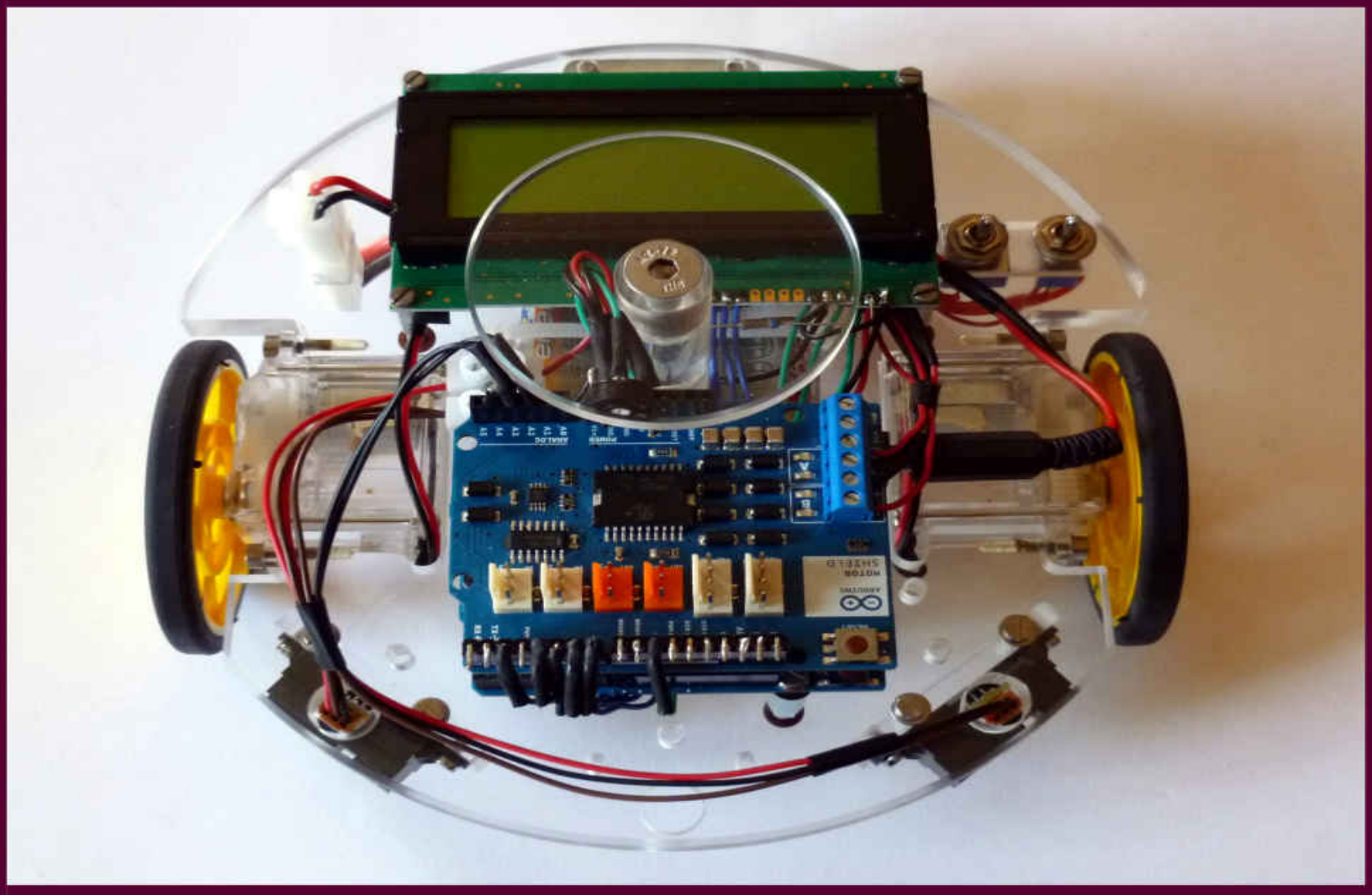
Mobiler Eigenbauroboter mit Arduino



Aufbau und Programmierung

3., überarbeitete und erweiterte Auflage

Klaus Röbenack

Mobiler Eigenbauroboter mit Arduino

Aufbau und Programmierung

Klaus Röbenack

Mobiler Eigenbauroboter mit Arduino

Mobiler Eigenbauroboter mit Arduino

Impressum

Röbenack, Klaus:
Mobiler Eigenbauroboter mit Arduino:
Aufbau und Programmierung.
3., überarbeitete und erweiterte Auflage.
Kindle Edition

Prof. Dr. Klaus Röbenack
Brucknerstr. 17, 01309 Dresden
Deutschland

Die Erstellung des Manuskripts sowie das Anfertigen der Schaltpläne und Programme erfolgten unter größter Sorgfalt. Dennoch können Fehler nicht ausgeschlossen werden. Der Autor übernimmt keine Garantie für die Korrektheit der beschriebenen Schaltungen und keine Haftung für entstandene Schäden.

Vorwort

Die im Buch beschriebene Roboterplattform entstand im Zusammenhang mit der Schülerprojektwoche des Martin-Andersen-Nexö-Gymnasiums (MANOS) in Dresden. Im Rahmen dieser Projektwoche wurden ab 2010 am Institut für Regelungs- und Steuerungstheorie der Technischen Universität Dresden Schülerpraktika zur Linien-Folge-Regelung eines mobilen Roboters durchgeführt.

Das Buch beschreibt den Aufbau und die Programmierung eines einfachen mobilen Roboters. Für die Steuerung des Roboters wurde die Arduino-Plattform, die sich durch leichte Handhabung auszeichnet, gewählt. Der Autor beschreibt den Anschluss und die Programmierung typischer Komponenten wie Motoren, LCD-Display und verschiedener Sensoren. Im Unterschied zu fertigen Roboterbausätzen wird dem Leser auch der nötige Freiraum zur Umsetzung und Ausgestaltung eigener Vorstellungen eingeräumt.

Dieses Buch ist für Leser gedacht, die bereits über erste Erfahrungen mit Mikrocontrollern im Allgemeinen oder der Arduino-Plattform im Besonderen verfügen. Zusätzlich werden geringe schaltungstechnische Grundkenntnisse erwartet sowie die Fähigkeit, einfache Programme in C bzw. C++ zu erstellen.

Es ist mir ein Bedürfnis, Herrn Dipl.-Ing. Carsten Knoll und Herrn Dipl.-Ing. Christian John für Ihren engagierten Einsatz in der Schülerprojektwoche zu danken. Mein Dank gilt ebenso Herrn Dr.-Ing. Jan Winkler, der mich in vielerlei Hinsicht unterstützt hat. Bezüglich der Durchführung der Schülerpraktika möchte ich zudem Frau Dipl.-Ing. Anja Lehmann, Herrn Dipl.-Ing. Matthias Schäfer und Herrn Dipl.-Ing. Mirko Franke danken. Dabei gebührt Herrn Schäfer zusätzlich Dank für die Durchsicht des Manuskripts und die damit verbundenen Anregungen.

Die Anregung zur Beschäftigung mit mobilen Robotern hat bei mir tiefere Wurzeln. Mit großer Faszination las ich als Jugendlicher den Beitrag „Kybernetische Tiere" von Reinhard Oettel im Elektronischen Jahrbuch für

den Funkamateur 1966. Der Beitrag behandelt ein Fahrmodell, welches mit Hilfe transistorierter Baugruppen einfache tierische Verhaltensformen nachbildet. Dem Autor möchte ich unbekannterweise für diesen inspirierenden Beitrag recht herzlich danken.

Das Buch erschien im November 2015 in der ersten Auflage. Die 2018 erschiene zweite Auflage berücksichtigt zahlreiche Korrekturen, Verbesserungen und Erweiterungen. So wurde beispielsweise der Abschnitt zu dem nicht mehr zeitgemäßen Arduino Duemilanove Board gestrichen. Beim Anschluss des LCD-Moduls wurde die Numerierung der Datenleitungen geändert. Die drahtlose Steuerung des Roboters ist jetzt auch über Smartphone via Bluetooth beschrieben. In dem neuen Kapitel zum I²C-Bus wird zudem eine inertiale Messeinheit zur Winkelbestimmung behandelt.

In der vorliegenden dritten Auflage wurden umfangreiche Erweiterungen vorgenommen. Neben der Vorstellung weiterer Programmierumgebungen (Arduino-Web-Editor, PlatformIO) haben auch die Kapitel zur Fernsteuerung (Bluetooth-Kopplung zwischen Arduino-Boards, Kommunikation über ZigBee bzw. XBee) und zu inertialen Messeinheiten (Magnetometer) deutliche Erweiterungen erfahren. Zusätzlich wird auf Boards mit ARM-Architektur eingegangen.

Hinsichtlich der Überarbeitung des Buches danke ich Herrn Univ.-Prof. Dr.-Ing. Frank Woittennek, Herrn Dipl.-Ing. Oliver Schnabel und Herrn Marcus Riesmeier, MSc., für die Weiterentwicklung der Roboterplattform. Ein besonderer Dank gilt auch Herrn Vincent Voigtländer, der zunächst im Rahmen einer wissenschaftlichen Schülerprojektarbeit und später als Besondere Lernleistung eine Roboter-Plattform mit Allseitenrädern entwickelt und aufgebaut hat.

Dieses Buch möchte ich — wie schon in der ersten Auflage — meinen Kindern widmen. Nicht zuletzt danke ich meiner Frau für Ihre tägliche Unterstützung.

Dresden, im März 2020

Klaus Röbenack

1 Arduino-Plattform

1.1 Allgemeines

Arduino® ist eine Mikrocontrollerplattform, die neben der Hardware auch quelloffene Software beinhaltet [1]. Zusätzlich ist die Hardware ist in dem Sinne quelloffen, dass die Schaltpläne, das Leiterplattenlayout usw. online verfügbar sind. Zu der Arduino-Plattform existieren etliche Bücher [7,8] und Starter-Sets.

Im Rahmen der Arduino-Plattform stehen zahlreiche verschiedene Boards zur Verfügung. Die Boards beherbergen den Mikrocontroller. In den meisten Fällen handelt es sich um einen Controller der ATmega-Serie von Atmel® [10,11]. Vom Mikrocontroller werden etliche analoge Eingänge sowie digitale Ein- bzw. Ausgänge herausgeführt. Zu den Arduino-Standard-Boards gibt es zahlreiche Erweiterungsleiterplatten, die *Shields* genannt werden.

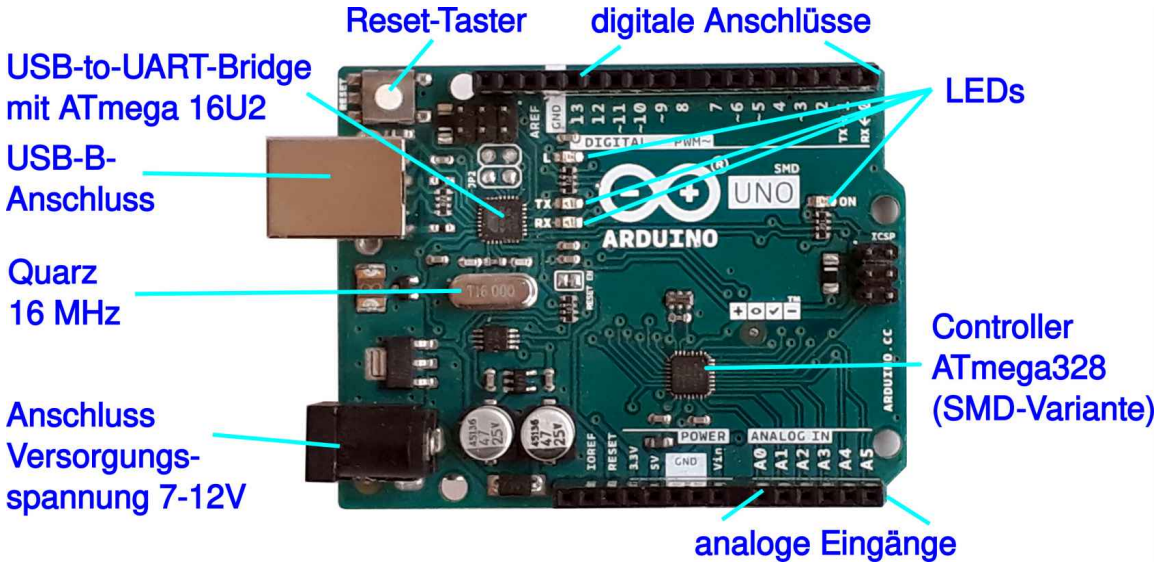
1.2 Arduino-Boards

In diesem Abschnitt werden die für den Roboter vorgesehenen Arduino-Boards kurz vorgestellt.

1.2.1 Arduino Uno R3

Das Arduino Uno Board war zum Zeitpunkt der Manuskripterstellung vermutlich das verbreitetste Arduino Board. Herzstück des Boards ist der Mikrocontroller ATmega328. Dieser Controller verfügt über 32 KiB Flash-Speicher, der weitgehend für eigene Programme genutzt werden kann. Die Erzeugung der Taktfrequenz erfolgt mit einem 16 MHz-Quarz. Die Kommunikation mit dem PC läuft über eine USB-B-Schnittstelle, deren Ansteuerung durch einen Atmega16U2 vorgenommen wird. Das Board verfügt ferner über 14 digitale Ein/Ausgabekanäle mit den Nummern 0,…,13 sowie 6 analoge Eingänge mit den Bezeichnungen A0,…,A5. Während ursprüng-

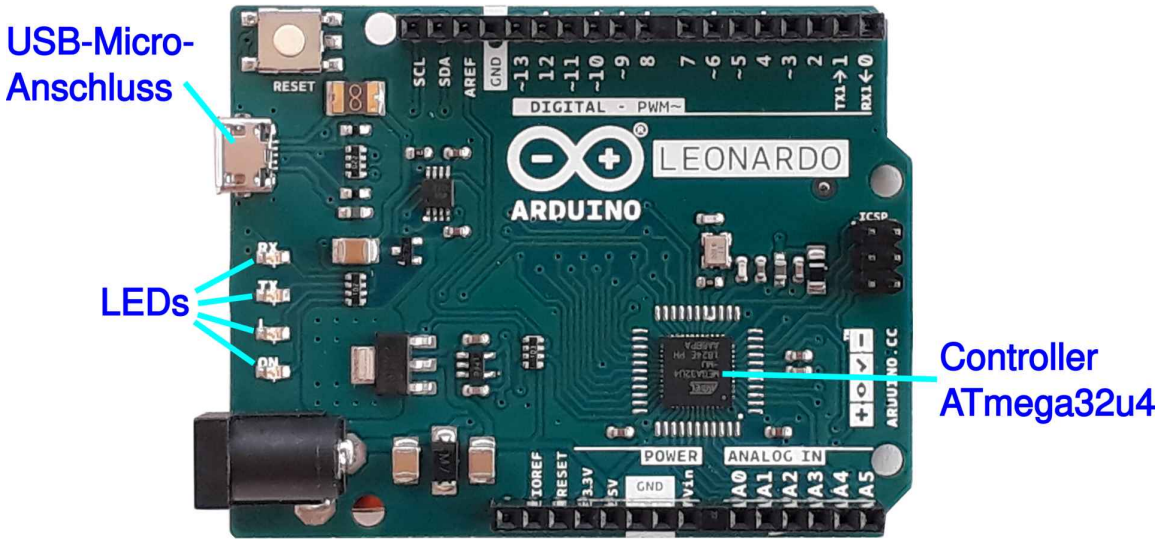
lich der Mikrocontroller ATmega328 in der DIP28-Gehäusevariante verbaut wurde, ist mittlerweile fast ausschließlich die SMD-Variante anzutreffen.



Arduino Uno Board R3, SMD-Version

1.2.2 Arduino Leonardo

Die Besonderheit des Arduino Leonardo Boards liegt in der Verwendung eines Mikrocontrollers mit integriertem USB-Anschluss. Dadurch spart man nicht nur die USB-to-UART-Bridge ein, sondern kann mit dem Leonardo-Board gegenüber dem PC auch eine Tastatur bzw. eine Maus emulieren. Zusätzlich ist es möglich, die analogen Eingänge auch als digitale Ein- bzw. Ausgänge zu nutzen. Macht man von dieser Möglichkeit Gebrauch, hat man insgesamt 20 digitale Ein- bzw. Ausgänge zur Verfügung. Die analogen Eingänge A0,…,A5 werden dann den digitalen Kanälen 18,…,23 zugeordnet. Umgekehrt lassen sich auch 6 der digitalen Kanäle als zusätzliche analoge Eingänge A6,…,A11 nutzen, so dass man insgesamt 12 analoge Eingänge verwenden kann. Diese zusätzlichen analogen Kanäle sind auf der Leiterseite des Boards angegeben.



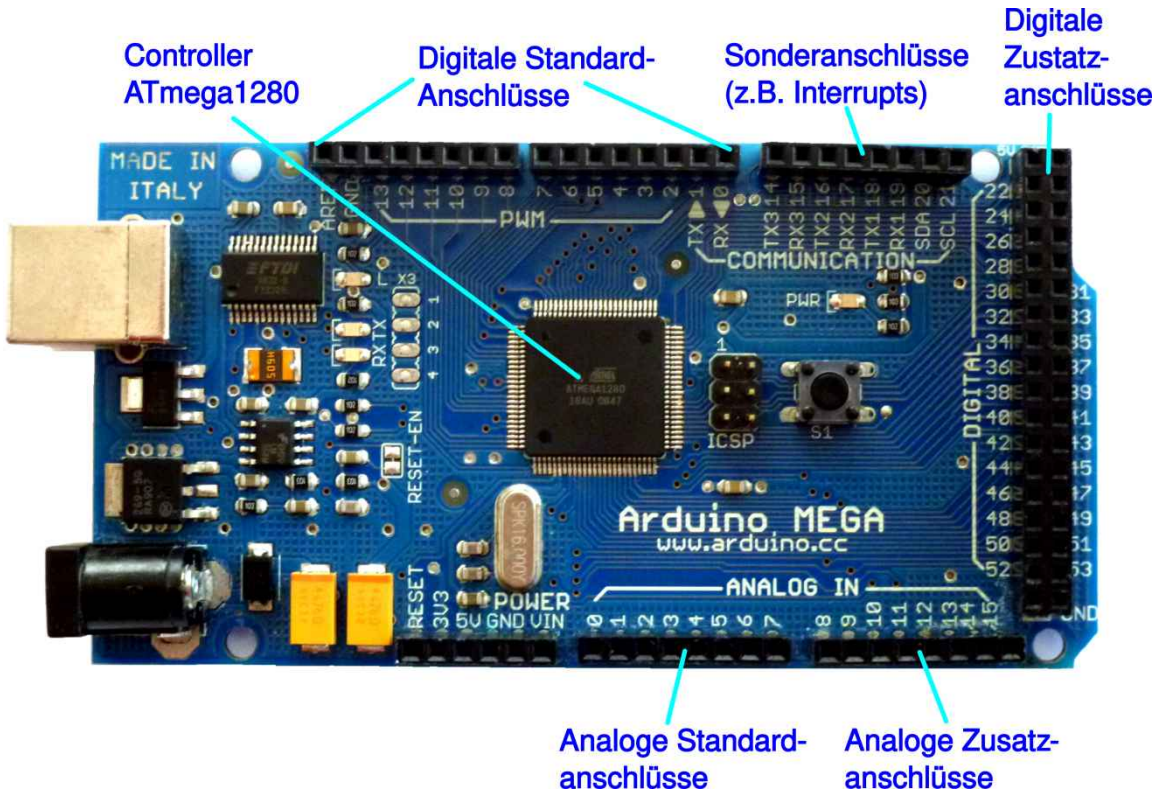
Arduino Leonardo Board

1.2.3 Alternative Boards

Zu den beschriebenen Standard-Boards gibt es zahlreiche Alternativen. Dieser Abschnitt soll dazu nur einige Anregungen liefen und erhebt daher keinen Anspruch auf Vollständigkeit.

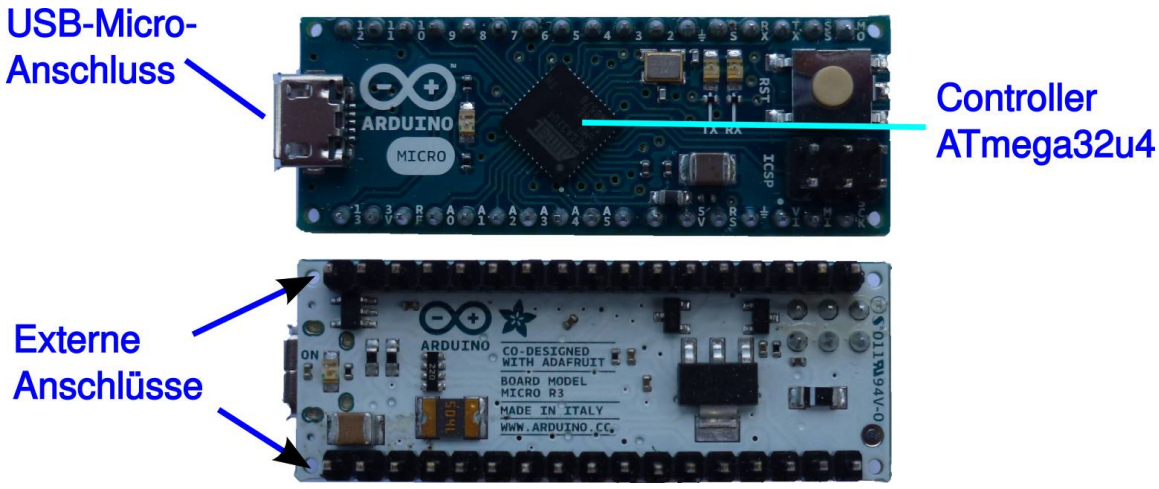
Boards anderer Hersteller: Einige Hersteller bieten alternative Boards an, die mit Arduino kompatibel sind, z.B. das Seeeduino Board von Seeed Technology Limited oder das RedBoard von SparkFun Electronics [4]. Die Firma Velleman vertreibt ebenfalls verschiedene Arduino-kompatible Boards [9].

Boards mit mehr Anschlüssen: Die 14 digitalen und 6 analogen Anschlüsse der Standard-Boards sind schnell belegt. Abhilfe schaffen hierbei Boards wie das Arduino Mega oder das Arduino Mega 2560, die mit 54 digitalen und 16 analogen Kanälen über deutlich mehr Ein- bzw. Ausgänge verfügen. Bei diesen Board kommen die Controller ATmega1280 bzw. ATmega2560 zum Einsatz.



Arduino Mega Board

Kleinere Boards: Für einen platzsparenden Aufbau gibt es auch etliche Boards mit einem kleineren Formfaktor, beispielsweise Arduino Micro, Arduino Mini oder Arduino Nano. Das Arduino Nano Board entspricht von der Funktionalität dem Arduino Duemilanove, das Arduino Micro Board dagegen dem Arduino Leonardo. Kleinere Boards werden auch von anderen Herstellern angeboten, z.B. Arduino Pro Mini und Pro Micro von SparkFun [4].



Arduino Micro Board (beide Seiten)

Boards mit leistungsfähigeren Controllern: Bei neueren Boards, wie beispielsweise Arduino Zero, M0 und Due, kommen 32-Bit Controller mit ARM-Kern zum Einsatz. Dadurch steht dem Nutzer eine deutlich höhere Rechenleistung und mehr RAM als bei den 8-Bit Boards zur Verfügung. Allerdings werden diese Boards mit 3,3V anstelle der sonst üblichen Spannung von 5V betrieben. Das bedeutet, dass Sensoren und Shields, die 5V ausgeben, nicht direkt an das Board angeschlossen werden können. In diesen Fällen wären Pegelanpassungen nötig.

1.3 Entwicklungsumgebung

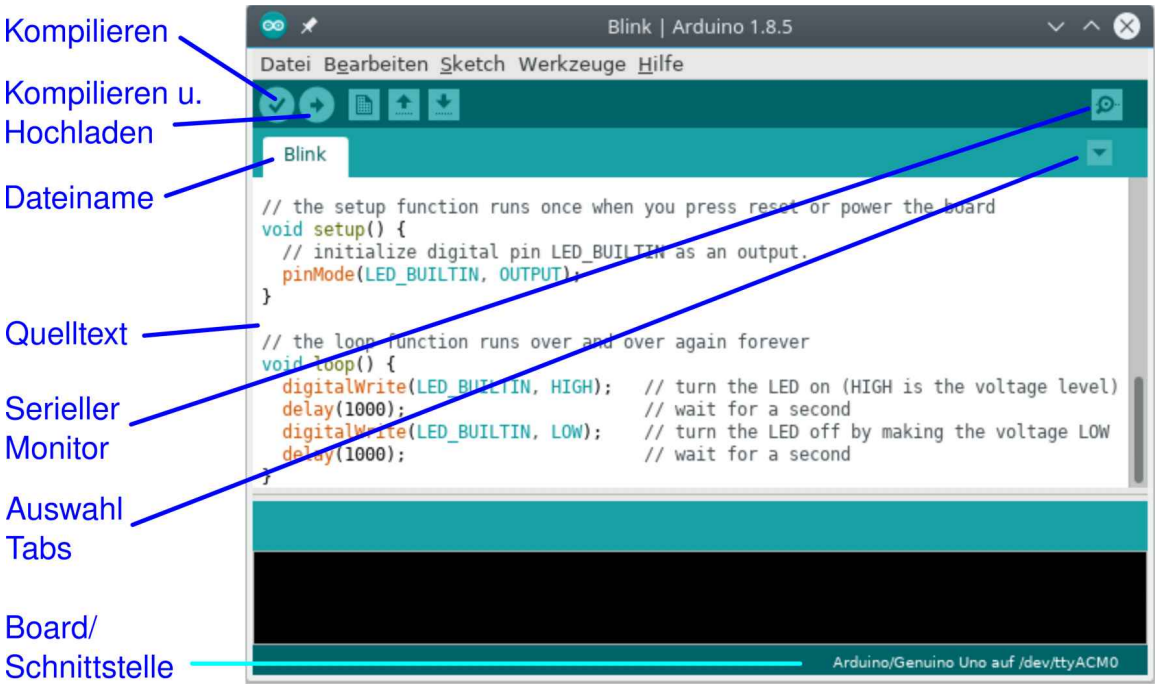
1.3.1 Arduino IDE

Die Entwicklungsumgebung (engl. *integrated development environment*, kurz IDE) ist quelloffen (engl. *open source*) und kann von der Arduino-Webseite heruntergeladen werden. Die Software steht für Windows, Mac OS X und Linux (32 Bit sowie 64 Bit) zur Verfügung. Für alle genannten Plattformen ist die Installation auf der Arduino-Webseite gut dokumentiert. In Abhängigkeit vom verwendeten Arduino-Board kann die Installation zusätzlicher Treiber für die USB-Schnittstelle notwendig sein. Unter Linux lässt sich die Installation über die gängigen Paketmanager (z.B. `rpm`, `apt`, `yum`) durchführen.

Die Arduino-IDE besteht im Wesentlichen aus einem Editor für den Quelltext sowie Funktionen zum Übersetzen des Quelltextes und zum Hochladen auf ein Arduino-Board. Zusätzlich hat man die Möglichkeit, sich Meldungen des Arduino-Boards auf dem PC anzeigen zu lassen. Bei früheren Version hatten die Quelltextdateien die Endung `.pde`, bei neueren Versionen `.ino`.

Zur Inbetriebnahme sollte man zunächst das Arduino-Board über ein USB-Kabel mit dem PC verbinden. Als nächstes wählt man auf der Arduino-IDE unter **Werkzeuge > Board** das verwendete Arduino-Board aus. Unter **Werkzeuge > Port** kann man die für die Übertragung zum Arduino-Board verwendete USB-Verbindung auswählen. Je nach PC-Betriebssystem und Arduino-Board können dabei auch mehrere Kanäle zur Auswahl stehen.

Auf der Arduino-IDE sind unter **Datei > Beispiele** zahlreiche Beispiele für gängige Programmieraufgaben, insbesondere aber für die Ansteuerung der Peripherie, hinterlegt. Für einen ersten Test eignet sich das unter **Datei > Beispiele > 01.Basics > Blink** aufzurufende Programm Blink, welches die auf nahezu allen Boards an Pin 13 angeschlossene LED zum Blinken bringt. Dieses Programm ist zunächst zu kompilieren (Strg+R) und muss danach auf das Board übertragen (hochgeladen) werden (Strg+U).

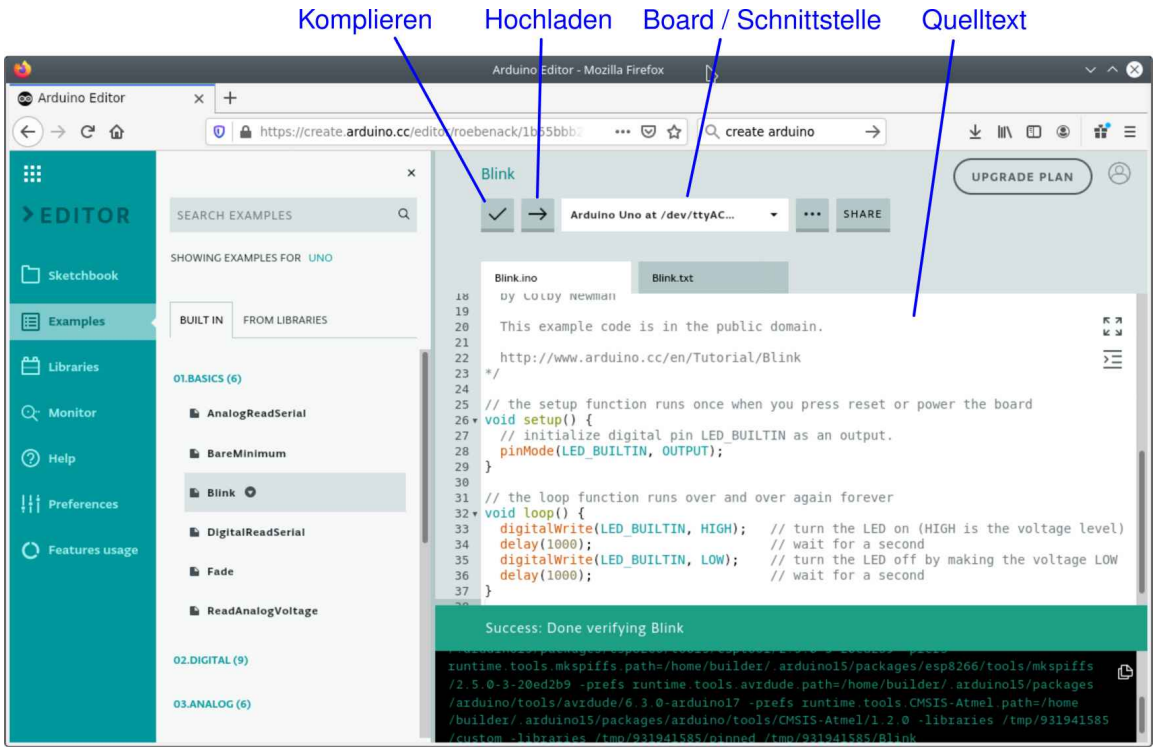


Entwicklungsumgebung mit Beispielprogramm `Blink`

Außerdem ist es möglich, Meldungen vom Arduino-Board zurück an den PC zu übertragen und dort anzeigen zu lassen. Diese Datenübertragung erfolgt seriell über die USB-Schnittstelle. Die Arduino-IDE verfügt über ein passendes Anzeigeprogramm, welches über **Werkzeuge > Serieller Monitor** aufgerufen werden kann.

1.3.2 Arduino-Web-Editor

Neben einer eigenständigen Programmierumgebung steht für die Entwicklung von Arduino-Programmen auch ein Web-Interface zur Verfügung [2]. Dazu muss man sich zunächst auf der Webseite [2] anmelden (mit Bestätigungs-Email). Für die Übertragung der Programme auf Arduino-Boards ist die Installation eines Plugins nötig. Auf der Arduino-Create-Webseite findet man unter Getting Started den Abschnitt Install Arduino Create Plugin. Der abgegebene Link führt auf eine Github-Seite [5], wo die Installation des Plugins für verschiedene Kombinationen aus Betriebssystem und Webbrowser beschrieben ist. Nach diesen Vorarbeiten kann man mit dem Arduino Web Editor in sehr intuitiver Weise Programme erstellen, kompilieren und hochladen.



Arduino-Web-Editor mit Beispielprogramm `Blink`

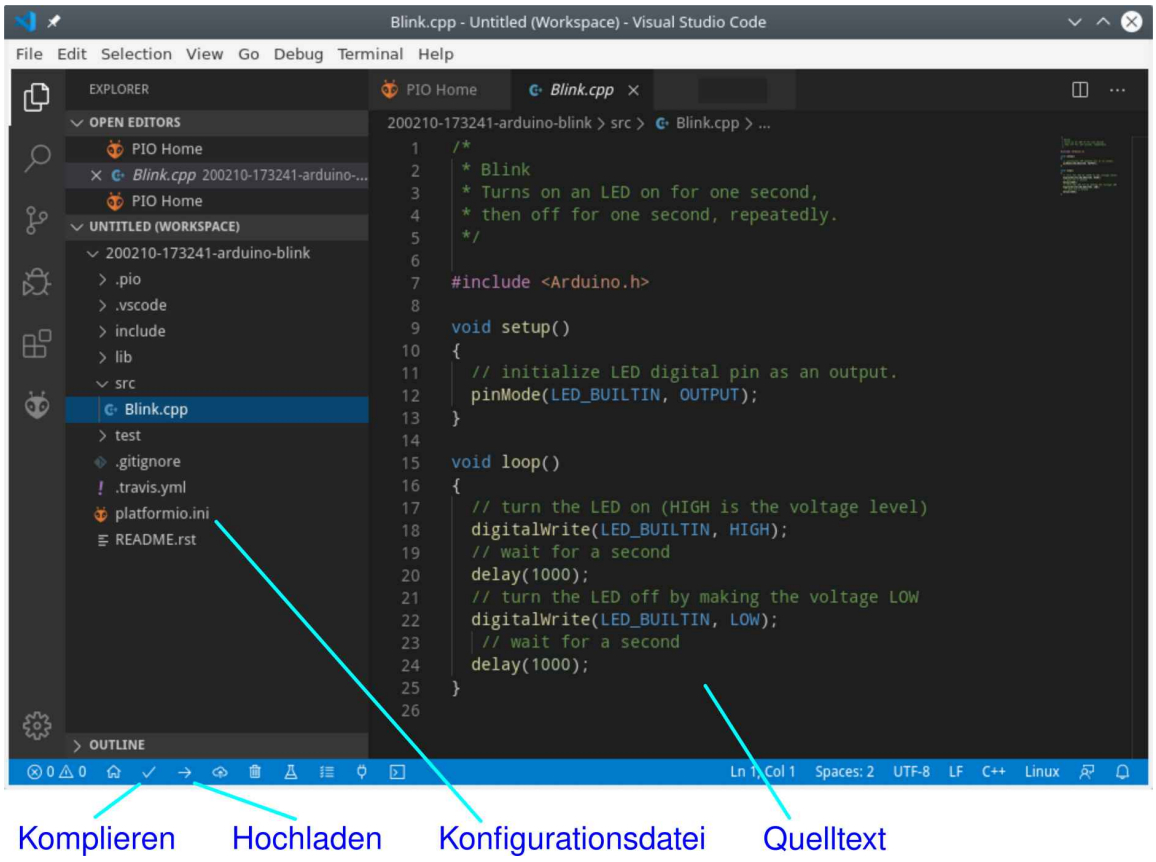
1.3.3 PlatformIO

PlatformIO ist eine Entwicklungsumgebung für zahlreiche Mikrocontrollerplattformen [3]. Neben der Atmel-AVR-Architektur der Arduino-Standardboards werden auch etliche andere Controllerarchitekturen unterstützt (u.a. Atmel SAM, ESP32, ESP8266, PIC32, RISC-V, STM32). Die PlatformIO IDE setzt auf Visual Studio Code von Microsoft auf und wendet sich daher eher an fortgeschrittene bzw. professionelle Anwender.

Die Installation ist auf der PlaformIO-Webseite gut beschrieben [3]. Dabei ist zuerst Visual Studio Code zu installieren [6]. Nach dem Aufruf von Visual Studio Code (mit dem Kommando `code`) installiert man die Erweiterung PlatformIO IDE (**View > Extensions**). Eine Beispielsammlung kann zusätzlich nachinstalliert werden. Für einen ersten Test wäre das Beispiel arduino-blink zu empfehlen. Die Arduino-Befehle werden über die Datei Arduino.h eingebunden. Die Controllerarchitektur bzw. das verwen-

dete Board sind in der Konfigurationsdatei platform.ini anzugeben. Für das Arduino Uno Board würde man folgende Beschreibung nutzen:

```
[env:uno]
platform = atmelavr
framework = arduino
board = uno
```

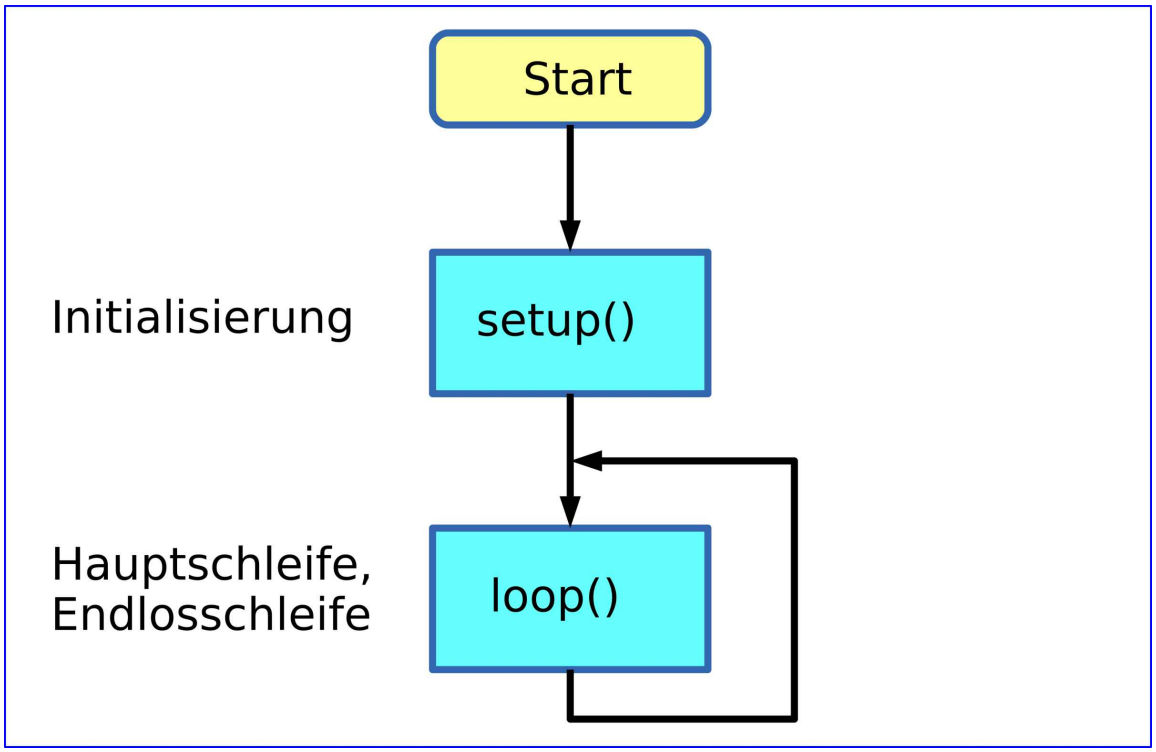


PlatformIO IDE mit Beispielprogramm `Blink`

1.4 Programmiersprache

Arduino wird im Wesentlichen in C/C++ programmiert. Ein C-Programm setzt sich in der Regel aus etlichen Funktionen zusammen. Bei den Standard-Varianten von C bzw. C++ für PCs bzw. Workstations wird mit dem Programmstart die Funktion `main` aufgerufen. Mit dem Ende bzw. der Abarbeitung der `main`-Funktion wird auch das Programm beendet.

Bei der C/C++ Variante für Arduino gibt es dagegen *zwei* besondere Funktionen. Die Funktion `setup` wird einmalig bei Programmstart aufgerufen. Dabei wird die `setup`-Funktion für die Initialisierung (z.B. der Ein- bzw. Ausgabekanäle) genutzt. Nach dem Abarbeiten der `setup`-Funktion wird die Funktion `loop` aufgerufen, die eine Endlosschleife durchläuft. Der darin enthaltene Code wird also zyklisch immer wieder abgearbeitet:



Programmstruktur mit Funktionen `setup` und `loop`

Das Grundgerüst eines Arduino-Programms ist unter **Datei > Beispiele > 01.Basics > BareMinimum** zu finden:

```
void setup() {
  // Einmalig auszuführender Code:
}

void loop() {
  // Zyklisch auszuführender Code:
}
```

In der C/C++ Variante für Arduino gibt es in Ergänzung zu dem Sprachumfang von Standard-C/C++ etliche zusätzliche Funktionen, die die Hardware des Mikrocontrollers unterstützen. Das betrifft beispielsweise das Einstellen der Datenrichtung bei den digitalen Kanälen (Ein- oder Ausgabe) bzw. die Datenabfrage der analogen wie digitalen Eingänge. Die entsprechenden Funktionen werden im Fortgang des Buches je nach Bedarf eingeführt.

1.5 Quellenangabe

1. Arduino – Home. URL: https://www.arduino.cc/.
2. Arduino Web Editor. URL: https://create.arduino.cc/editor.
3. PlatformIO. https://platformio.org/
4. SparkFun Electronics. URL: https://www.sparkfun.com/.
5. The Arduino Create Multi Platform Agent. https://github.com/arduino/arduino-create-agent
6. Visual Studio Code. https://code.visualstudio.com/
7. E. Bartmann: Die elektronische Welt mit Arduino entdecken. O'Reilly Verlag, 2. Auflage, Juni 2014.
8. J. Boxall: Arduino-Workshops: Eine praktische Einführung mit 65 Projekten (c't Hardware Hacks Edition). dpunkt.verlag, 2013.
9. Velleman: Arduino® kompatible Boards. https://www.velleman.eu/products/list/?id=435274
10. R. Walter: AVR Mikrocontroller Lehrbuch, Einführung in die Welt der AVR-RISC-Mikrocontroller am Beispiel des ATmega8. 2. Auflage, Juni 2004.
11. Seite „Atmel AVR". In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=Atmel_AVR.

2 Überlegungen zum Aufbau des Roboters

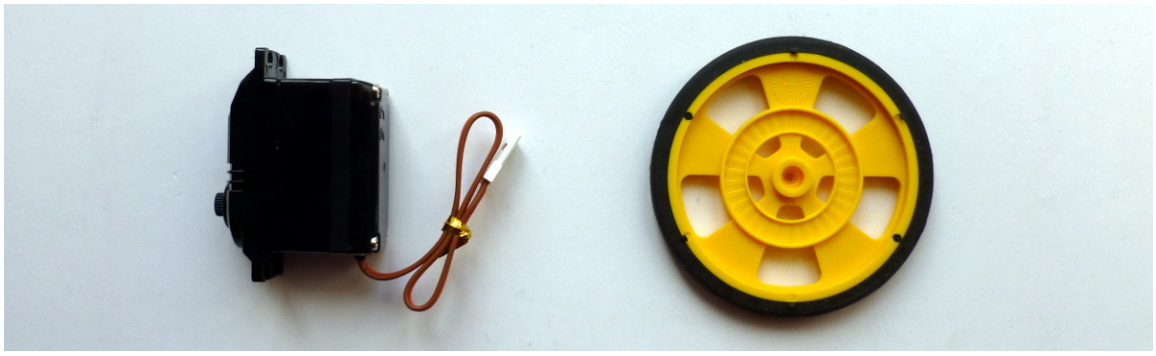
2.1 Auswahl der Motoren

Für eine einfache Ansteuerung setzt man bei mobilen Robotern praktisch immer permanenterregte Gleichstrommotoren ein. Gleichstrommotoren sind typischerweise für vergleichsweise hohe Drehzahlen ausgelegt, weshalb zusätzlich eine Untersetzung durch ein Getriebe erforderlich ist. Daher verwendet man vorzugsweise sogenannte *Getriebemotoren*, die Motor und Getriebe in einem Antriebsmodul kombinieren.

Ein in der mobilien Robotik anzutreffender Getriebemotor wird von der Firma Igarashi Motor GmbH unter der Typenbezeichnung 20GN152025-330-050 vertrieben. Dieser Motor ist für 4-12V ausgelegt. Das Getriebe weist ein Übersetzungsverhältnis von 1:50 auf. Ähnliche Motoren sind von Igarashi auch für andere Spannungen bzw. Übersetzungsverhältnisse im Angebot. Größere Motoren sind in der Reihe der Modelcraft RB-35 Getriebemotoren zu finden. Auch hier sind die Motoren für verschiedene Nennspannungen bzw. Übersetzungsverhältnisse verfügbar. Das abgebildete Modell ist ebenfalls für eine Nennspannung von 12V ausgelegt.

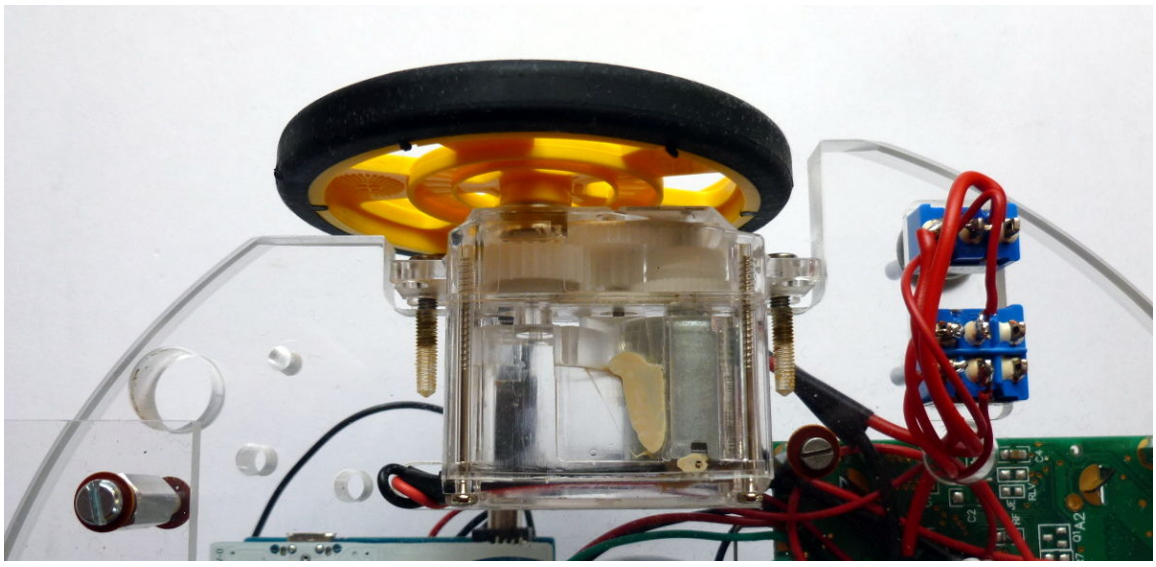
12V-Getriebemotoren: Motor von Igarashi (links), RB-35 Getriebemotor von Modelcraft (rechts)

Als Alternative zu teuren Getriebemotoren bietet sich der Umbau von Standardservos an. Servos sind Antriebe mit integrierter Lageregelung. Neben Gleichstrommotor und Getriebe enthalten sie auch eine Ansteuer-elektronik. Im normalen Einsatzfall verfügen Servos nur über einen endlichen Aussteuerbereich, der typischerweise im Bereich von ±90° liegt. Zum Umbau eines Servors in einen Getriebemotor muss man einerseits die Ansteuerelektronik entfernen bzw. deaktivieren. Die Anschlüsse des Gleichstrommotor sind dann direkt nach außen zu führen. Andererseits sind die mechanischen Anschläge, welche die Auslenkung des Servos begrenzen, zu entfernen. Das betrifft das für die Winkelmessung nicht mehr benötigte Potentiometer sowie mögliche Anschläge auf Zahnrädern bzw. der Achse. Neben dem günstigen Preis ist der Einsatz von Servos auch dahingehend vorteilhaft, dass dafür passende Kunststoffräder angeboten werden.



Standardservo von Modelcraft mit Kunststoffrad von Solarbotics [2]

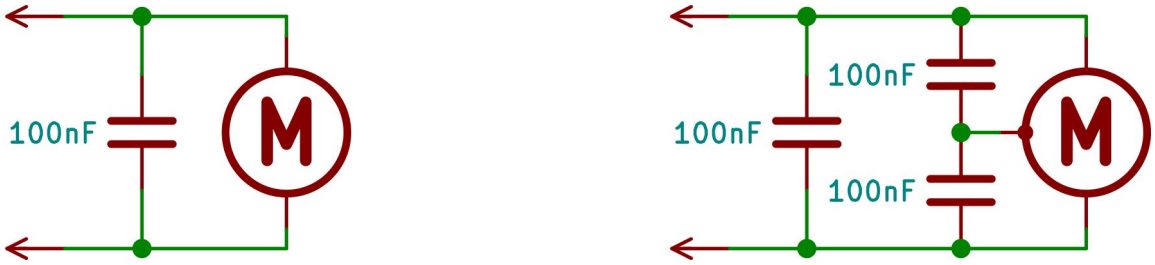
Servos sind normalerweise für einen Spannungsbereich von 4,5…6V ausgelegt. Dementsprechend darf man die aus umgebauten Servos entstandenen Getriebemotoren nicht mit zu hohen Spannungen betreiben. Ein Betrieb bis 9V sollte in der Regel kein Problem darstellen. Allerdings gibt es auch sogenannte High-Volt-Servos, die aber meist deutlich teurer sind. Den manuellen Servoumbau kann man vermeiden, indem man auf Spezialservos der Firma Solarbotics [2] zurückgreift.



Im Roboter verbauter Spezialservo von Solarbotics mit Rad

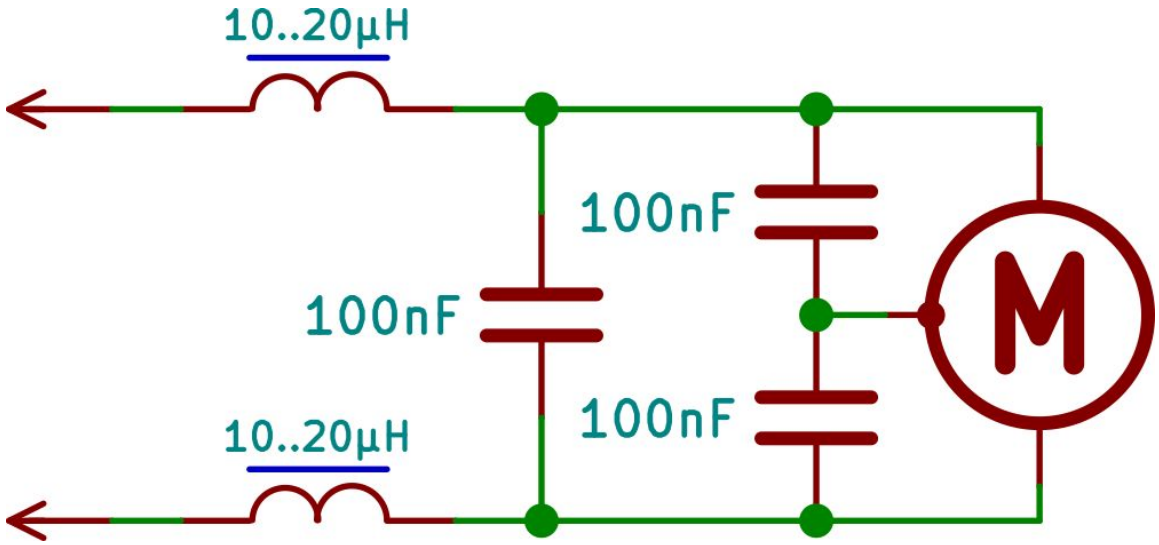
Bei permanenterregten Gleichstrommotoren werden die Wicklungen des Rotors mit Hilfe eines Kommutators umgepolt. Bei dieser Umpolung entstehen Funken und damit auch elektromagnetische Störungen. Diese

hochfrequenten Störsignale lassen sich mit Entstörkondensatoren reduzieren bzw. unterdrücken. Im einfachsten Fall schaltet man dazu einen Kondensator parallel zum Motor. Zusätzliche Entstörkondensatoren kann man mit dem Motorgehäuse verbinden:



Entstörung mit einem Kondensator (links), Entstörung mit drei Kondensatoren (rechts)

Eine noch stärkere Störunterdrückung erzielt man durch Hinzunahme von Entstördrosseln, deren Induktivität relativ unproblematisch ist:



Entstörung mit zusätzlichen Entstördrosseln

2.2 Akkus

Für die betrachteten Arduino-Boards wird ein Bereich von 6…20V für die bereitzustellende Versorgungsspannung angegeben. Ein Betrieb mit mehr als 12V ist allerdings nicht wirklich empfehlenswert. Auch für das später

beschriebene Arduino-Motor-Shield ist eine Maximalspannung von 12V angegeben. Wir schränken daher den Bereich der Versorgungsspannung auf 7…12V ein.

Bei einem häufigen Betrieb des mobilen Roboter ist der Einsatz von Primärzellen weder finanziell noch ökologisch sinnvoll. Man wird daher auf Akkumulatoren (Akkus) zurückgreifen. Im Modellbau sind zwei verschiedene Arten von Akkus verbreitet, nämlich Nickel-Metall-Hydrid-Akkus (NiMH) bzw. Lithium-Polymer-Akkus (LiPo).

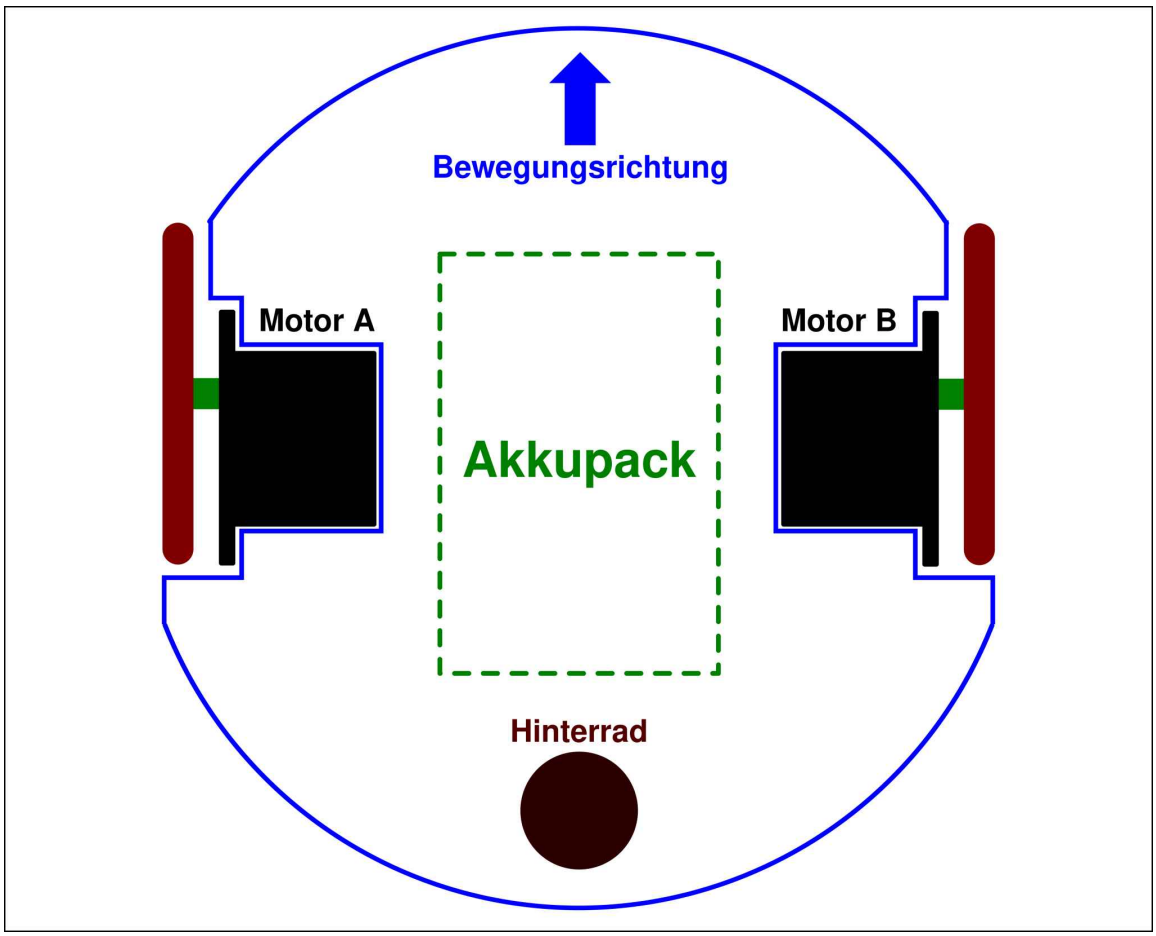
Eine einzelne NiMH-Zelle hat eine Nennspannung von 1,2V, unmittelbar nach dem Aufladen wird die Spannung eher bei 1,4…1,5V liegen. Für den angestrebten Spannungsbereich kann man entweder mehrere einzelne Akkus einsetzen oder einen Akkupack. Ein Akkupack mit 6 Zellen würde eine Nennspannung von 6×1,2V=7,2V bereitstellen, ein Akkupack mit 8 Zellen dagegen 8×1,2V=9,6V. Zum Einsatz beim mobilen Roboter wären grundsätzlich beide Varianten geeignet. Für den Anschluss von Akkupacks sind Tamiya-Stecker verbreitet. Man benötigt dafür allerdings auch ein anschlussseitig kompatibles Ladegerät.

Akkupack mit 2×4 NiMH-Zellen und Tamiya-Stecker

Anstelle von NiMH-Akkus kann man auch LiPo-Akkus verwenden, die eine deutlich höhere Energiedichte aufweisen. LiPo-Akkus werden daher vornehmlich bei Flugmodellen verwendet. Pro Zelle werden 3,7V zur Verfügung gestellt. Ein Akku mit zwei Zellen liefert dann 7,4V, einer mit drei Zellen 11,1V. Zum Aufladen sind sog. Balancer nötig, die die Ladespannung auf die einzelnen Zellen aufteilen.

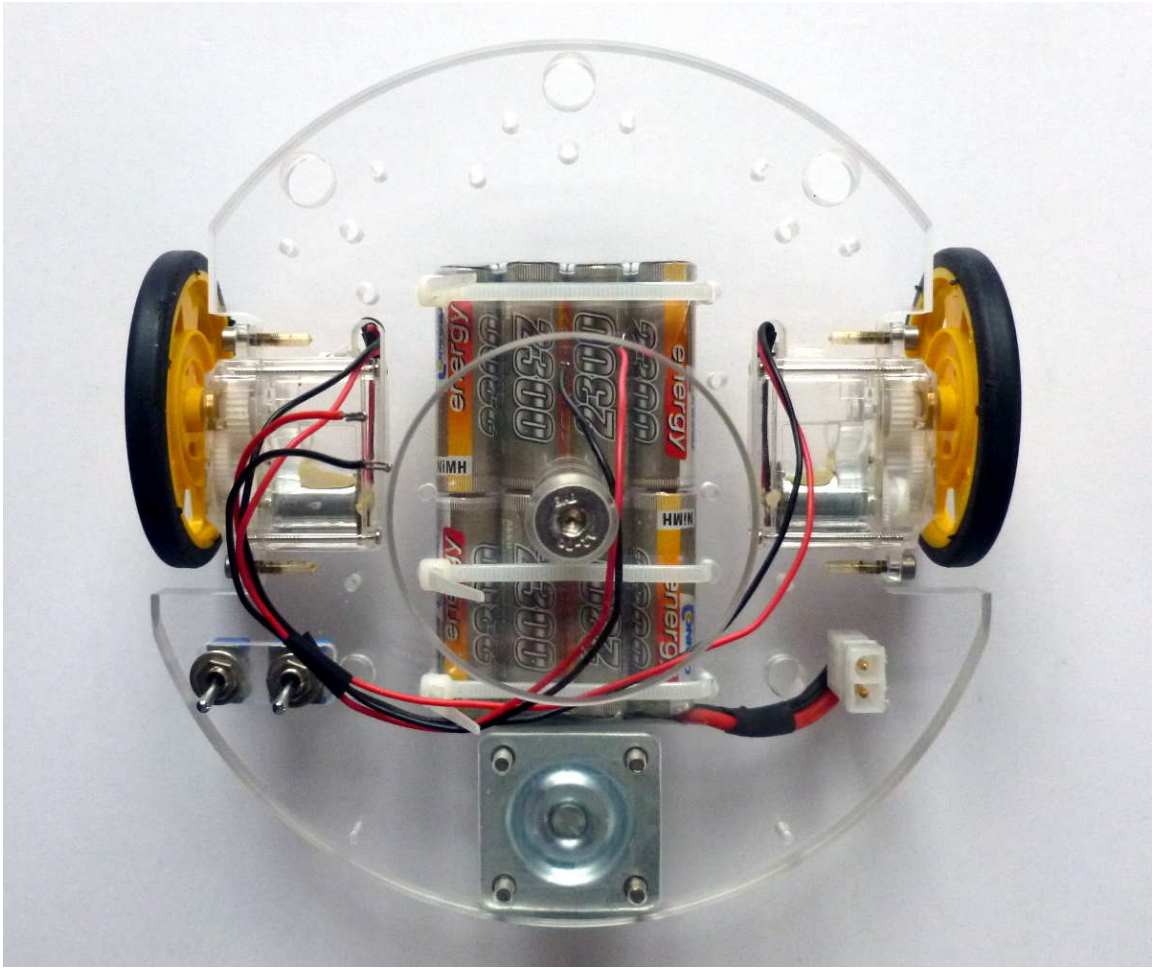
2.3 Mechanischer Aufbau

Für mobile Roboter gibt es verschiedene Antriebskonzepte. Bei dem hier beschriebenen Roboter werden zwei Motoren separat angesteuert. Damit kann der Roboter nicht nur vorwärts und rückwärts fahren, sondern sich auch auf der Stelle drehen. Die Plattform, an welcher die Motoren, Sensoren usw. befestigt sind, wird oft kreisförmig ausgeführt. Für unsere Roboterplattform ist ein Durchmesser von ca. 15…20cm sinnvoll.



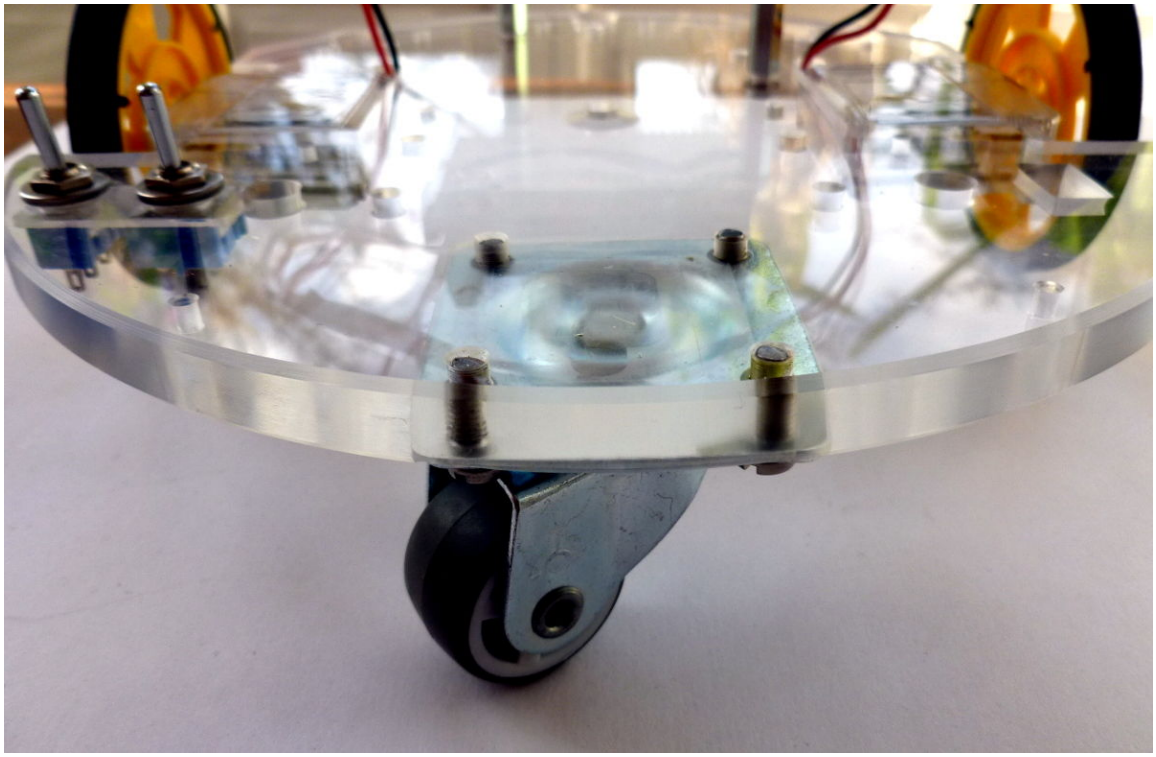
Roboterplattform

Im einfachsten Fall kann man die Plattform aus Holz fertigen. Optisch ansprechender, in der Verarbeitung aber auch anspruchsvoller ist eine Roboterplattform aus Acryl. Die Oberseite ist typischerweise für das Controllerboard und die LCD-Anzeige reserviert, den Akkupack kann man dagegen auch auf der Unterseite anbringen.



Roboterplattform mit Spezialservos und Akkupack

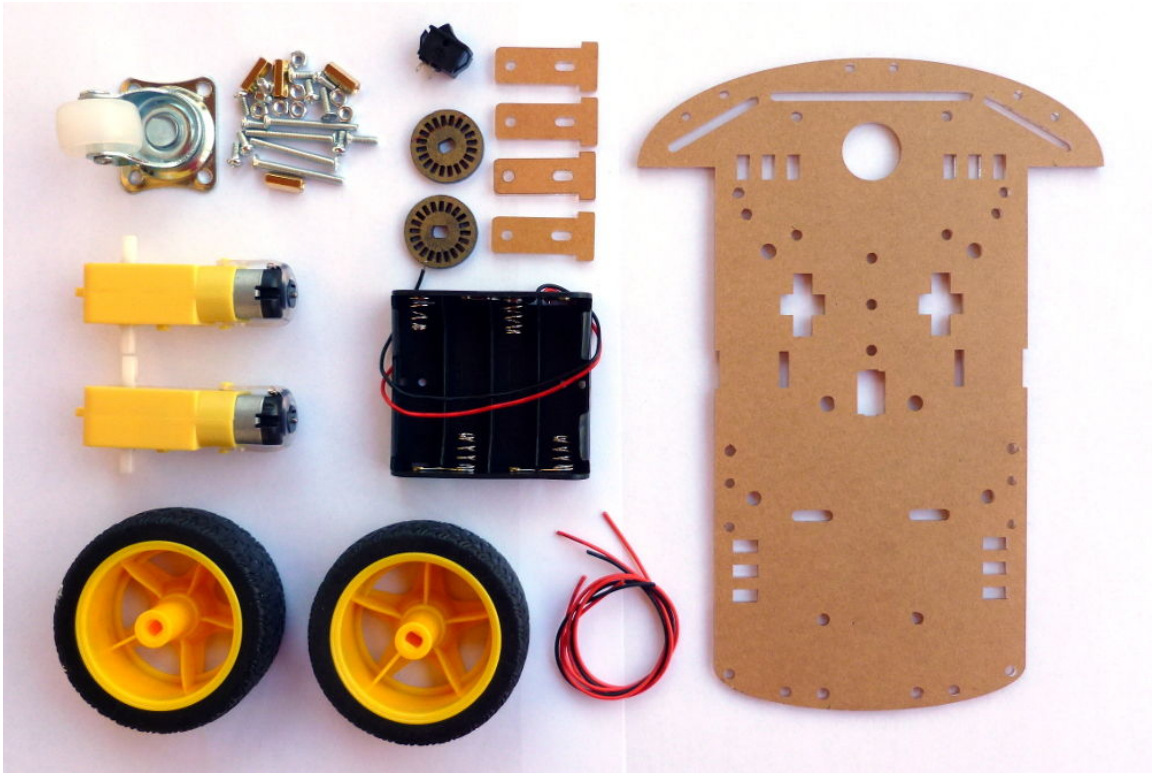
Um der Roboterplattform Stabilität zu verleihen müssen die zwei Antriebsräder durch einen dritten Auflagepunkt ergänzt werden. Hier ist der Einsatz einer Transport- bzw. Lenkrolle als Hinterrad naheliegend. Derartige Rollen sind in Baumärkten in verschiedenen Größen verfügbar. In Verbindung mit den Rädern von Solarbotics ist für die Lenkrolle eine Bauhöhe von ca. 34mm sinnvoll. Die erforderliche Bauhöhe hängt allerdings auch von der Montage bzw. Befestigung der Motoren ab. Bei manchen mobilen Robotern, z.B. beim Pololu 3pi Robot, wird für den dritten Auflagepunkt ein Plastikball verwendet [1].



Hinterrad des Roboters

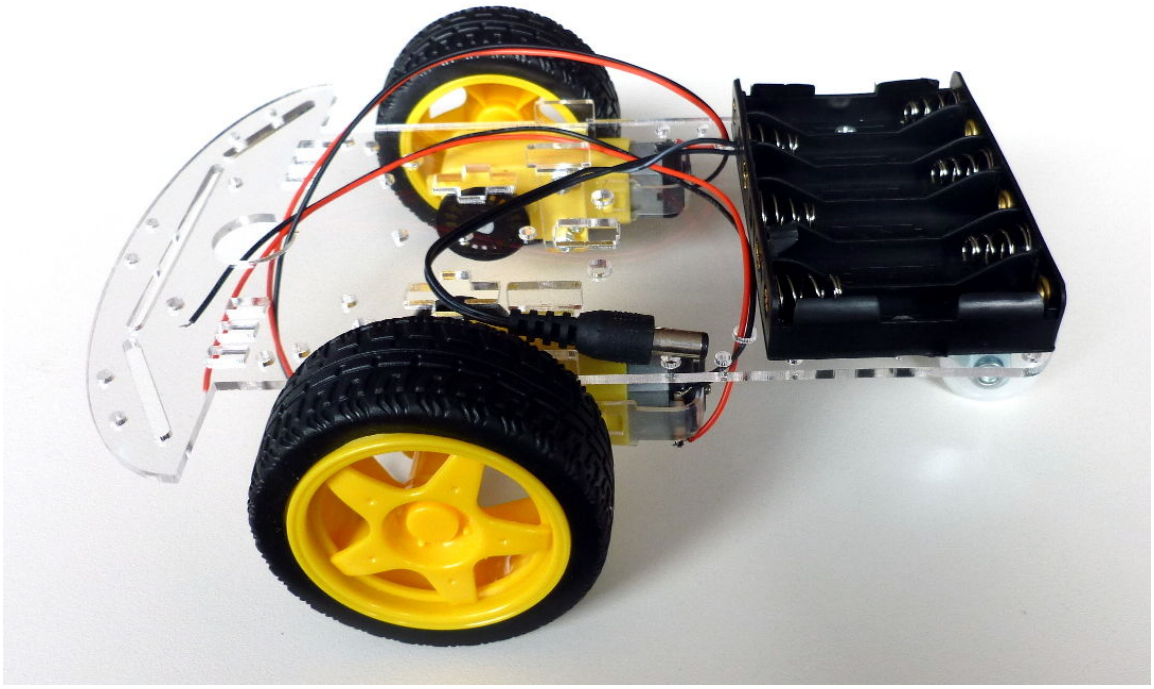
Als Bedienelemente sollte man mindestens zwei Schalter vorsehen: Einen als Hauptschalter, den anderen Schalter für die Motoren. Mit dem zweiten Schalter kann man Programme erproben, ohne dass der Roboter gleich losfährt und vielleicht vom Schreibtisch stürzt. Zusätzlich könnte man auch einige Taster als Bedienelemente vorsehen.

Als Zwischenstufe zwischen einem komplett eigenständigen Aufbau und einem vollständigen Roboterbausatz (siehe beispielsweise [3]) könnte man auch auf Bausätze bzw. Fertiggeräte für Fahrgestelle bzw. Roboterplattformen (Roboter Chassis) zurückgreifen. Die folgende Abbildung zeigt einen derartigen Bausatz, der alle wesentlichen mechanischen Komponenten einschließlich der Getriebemotoren enthält. Der Bausatz wird unter der Bezeichnung ZK-2WD vertrieben, wobei 2WD für *two wheels drive steht*.



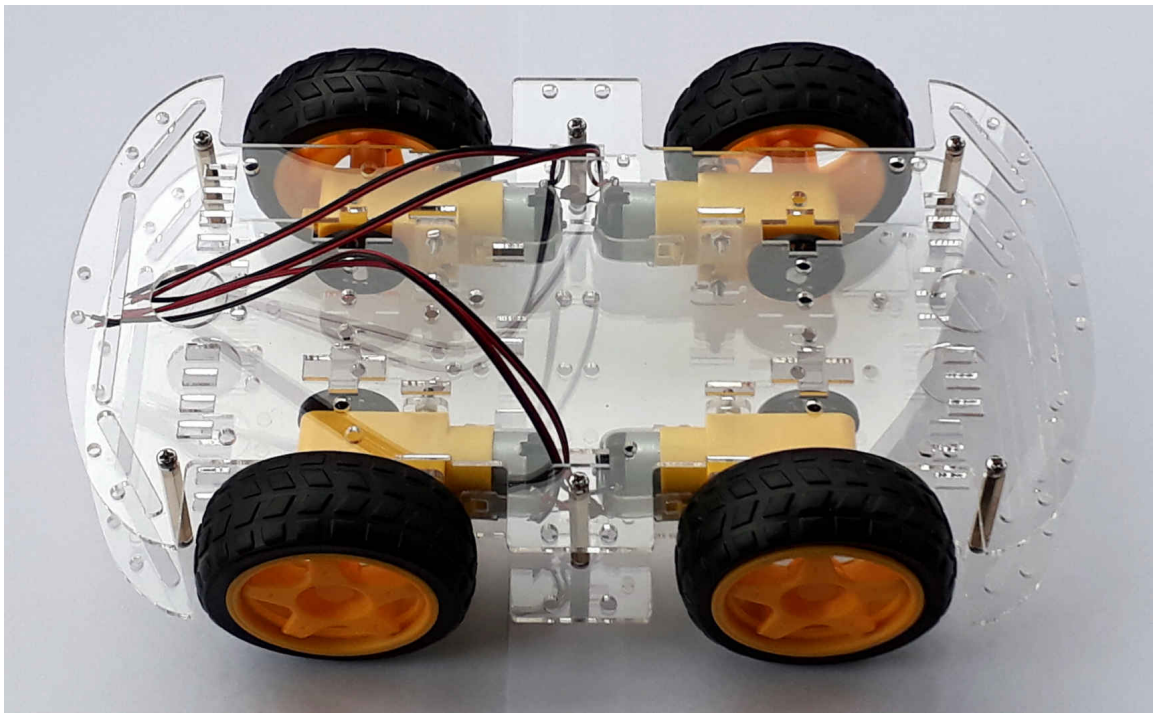
Bausatz für ein Roboter Chassis

Für den Betrieb mit einem Arduino-Board wäre allerdings der für 4 Zellen ausgelegte Batteriehalter durch einen für 6 Zellen zu ersetzen:



Aufgebautes Roboter Chassis mit Batteriehalter für 6 Zellen

Eine ähnliche Roboterplattform gibt es auch mit 4-Rad-Antieb. Hier würde man typischerweise die Motoren auf der linken bzw. rechten Seite jeweils parallel schalten:



Roboter Chassis mit 4-Rad-Antrieb

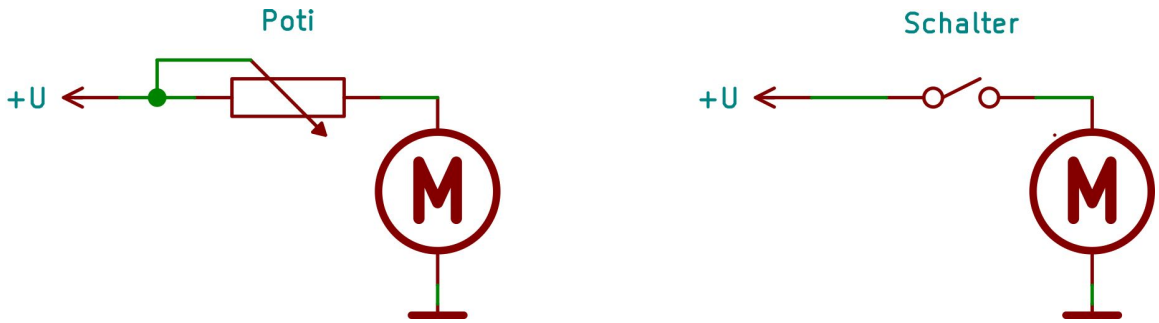
2.4 Quellenangabe

1. Pololu Robotics & Electronics. URL: https://www.pololu.com/.
2. Solarbotics. URL: https://solarbotics.com/.
3. Bachfeld, D.: Roboter-Kits. Make 2/2015, S. 28-36.

3 Motoransteuerung

3.1 Geschwindigkeitsvorgabe mit Pulsweitenmodulation (PWM)

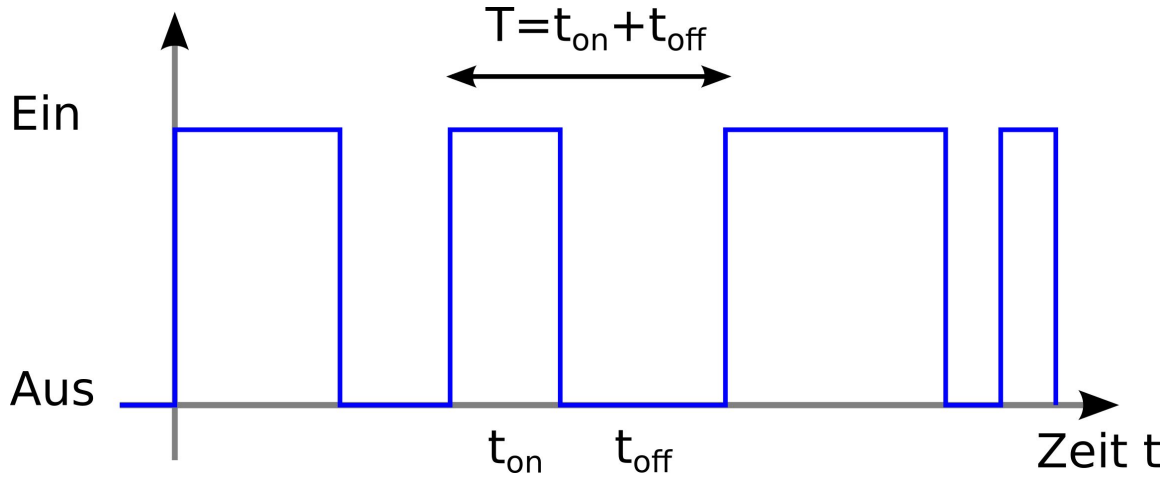
Die beim mobilen Roboter eingesetzten Gleichstrommotoren sollen in der Regel nicht immer nur mit voller Nenndrehzahl laufen, vielmehr möchte man die Drehzahl und damit die Geschwindigkeit kontinuierlich verstellen können. Dafür gibt es grundsätzlich zwei Herangehensweisen. Bei einer analogen Ansteuerung wird über einen vorgeschalteten veränderlichen Widerstand ein Spannungsabfall erzeugt. Bei einer digitalen Ansteuerung würde man nur schalten. Beide Varianten kann man auch mit Transistoren aufbauen:



Ansteuerung von Gleichstrommotoren: Analog mit Potentiometer (links), digital mit Schalter (rechts)

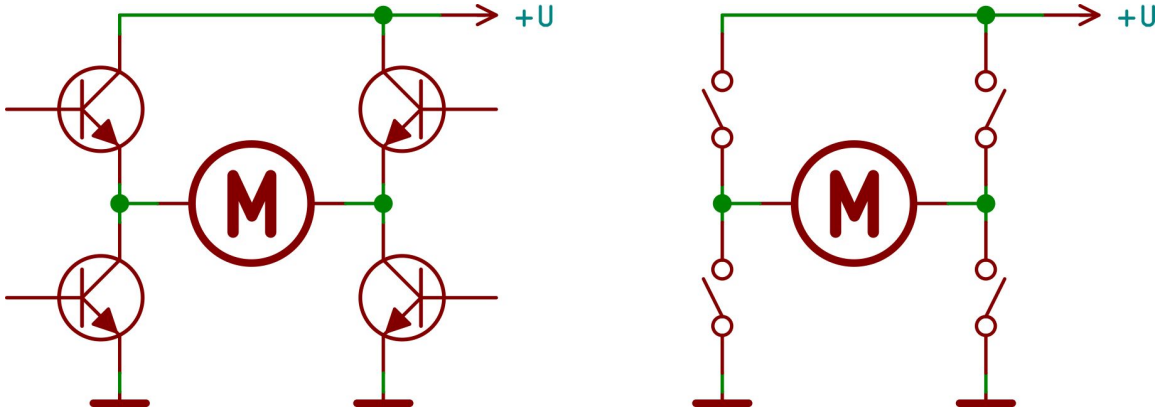
Stellt man bei der analogen Ansteuerung das Potentiometer so ein, dass am Motor nur die halbe Betriebsspannung anliegt, wird der Motor näherungsweise mit der halben Drehzahl arbeiten. Allerdings wird dann am Potentiometer die gleiche elektrische Leistung umgesetzt wie am Motor. Dadurch verringert sich einerseits der Wirkungsgrad, andererseits führt die am Widerstand eingeprägte Energie zu einer Erwärmung der Bauteils (egal, ob der Widerstand als Potentiometer oder als Transistor aufgebaut ist).

Die o.g. Probleme mit einer analogen Ansteuerung lassen sich mit einer digitalen Ansteuerung, also im geschalteten Betrieb, umgehen. Ist der Schalter offen, dann fließt kein Strom. Über einem geschlossenen Schalter fällt (theoretisch) keine Spannung ab. In beiden Fällen kommt es zu keinem (merklichen) Leistungsumsatz am Schalter. Im Normalfall kann man mit einem Schalter nur zwischen Stillstand und voller Drehzahl wählen. Mit einer *Pulsweitenmodulation* (PWM, engl. *pulse-width modulation*) kann man auch (fast beliebige) Zwischenwerte einstellen [1]. Dazu wird der Schalter mit einer Frequenz $f=1/T$ schnell ein- und ausgeschaltet. In der nachfolgenden Abbildung bezeichnet t_{on} die Einschaltzeit und t_{off} die Ausschaltzeit, so dass eine Schwingungsperiode die Zeitdauer $T=t_{on}+t_{off}$ aufweist. Das *Tastverhältnis d* (engl. *duty ratio* oder *duty cycle*) ist das Verhältnis der Einschaltzeit zur Periodendauer und kann Werte im Bereich von Null (immer aus) bis Eins (immer an) annehmen. Bei einem Tastverhältnis von $d=0{,}5$ ist der Schalter genauso lange an- wie ausgeschaltet. In diesem Fall würde dann im Mittel die halbe Spannung anliegen.



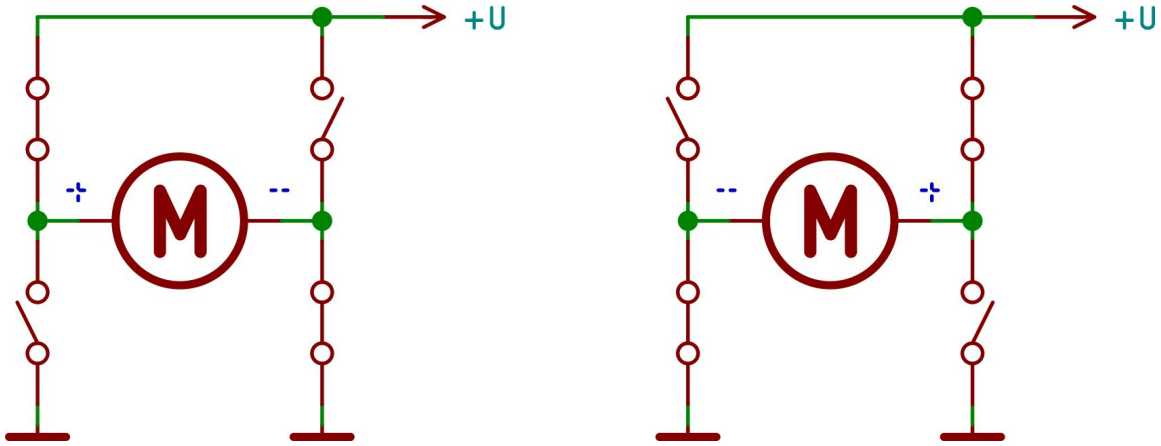
Signalverlauf bei Pulsweitenmodulation (PWM)

Die Ansteuerung der Motoren im Roboter erfolgt etwas anders, nämlich mit einer *Brückenschaltung*. Diese besteht aus vier Schaltern, die als Transistoren ausgeführt sind:



Brückenschaltung: Brücke mit NPN-Transistoren (links), Brücke mit Schaltern (rechts)

Eine Brückenschaltung erlaubt die Umpolung der Ausgangsspannung und ermöglich damit den Betrieb des Motors in beiden Richtungen:

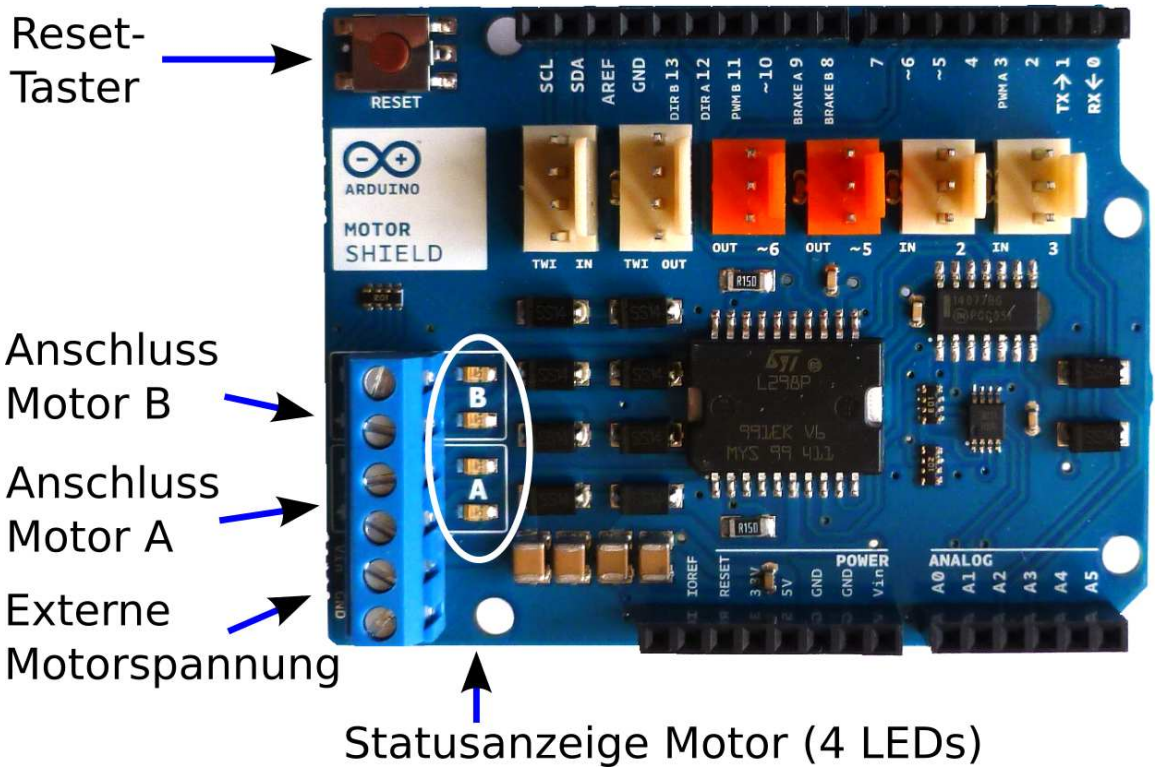


Richtungswechsel mit Brückenschaltung

Da der Roboter von zwei Gleichstrommotoren angetrieben werden soll sind dementsprechend auch zwei Brückenschaltungen nötig. Diese werden nicht diskret (aus einzelnen Bauteilen) aufgebaut, sondern mit Hilfe integrierter Schaltkreise realisiert. Bei älteren Motor-Shields wurde oft der Schaltkreis L293 eingesetzt, der für Ströme bis zu 1A vorgesehen ist [2]. Modernere Motor-Shields verwenden den für 2A vorgesehenen IC L298, der zusätzlich die Messung der Motorströme erlaubt [3].

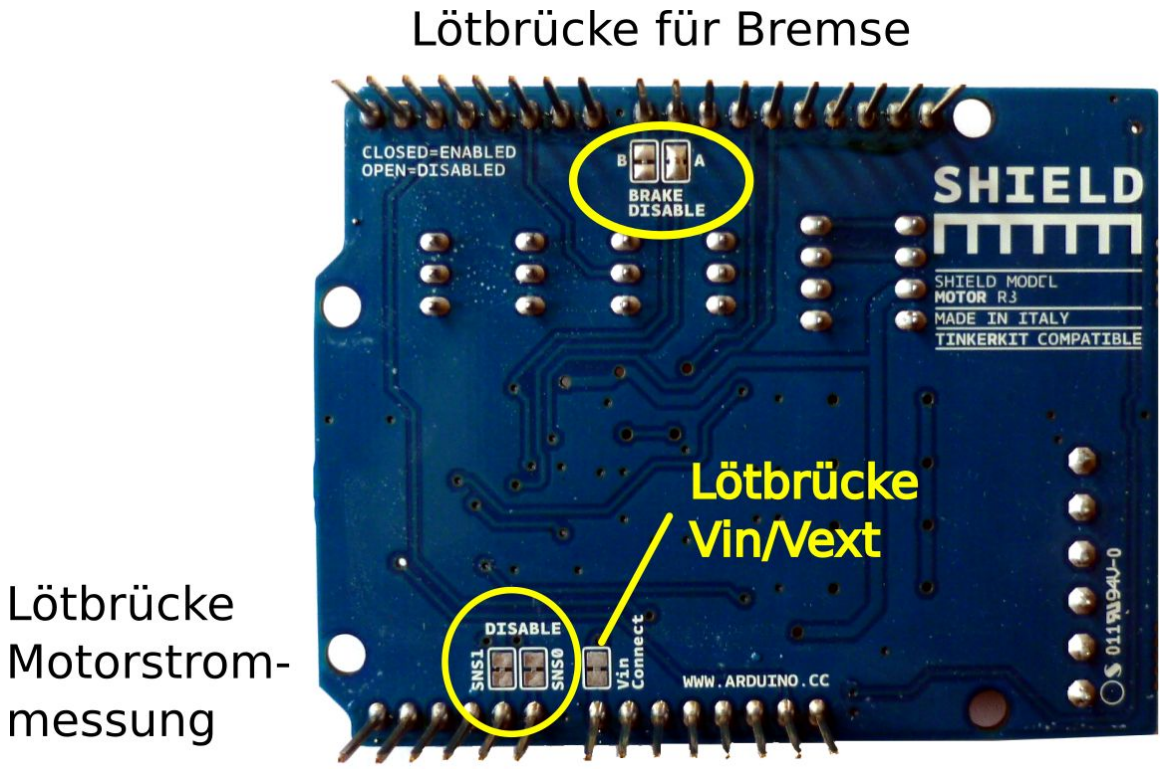
3.2 Arduino-Motor-Shield R3

Das Arduino-Motor-Shield R3 (Revision 3) ist zum Zeitpunkt der Manuskripterstellung wahrscheinlich die populärste Zusatzbaugruppe zur Motoransteuerung. Das Motor-Shield ist für die Ansteuerung von zwei Gleichstrommotoren vorgesehen. Alternativ ist auch die Ansteuerung eines Schrittmotors möglich [4].



Arduino Motor Shield Vorderseite

Außerdem ist es möglich, beide Motoren direkt zu bremsen (statt sie auslaufen zu lassen) und die Motorströme zu messen. Wenn man diese Zusatzfunktionen nicht benötigt, kann man die entsprechenden Lötbrücken auf der Rückseite durchtrennen und damit die dafür vorgesehenen Adressen (Anschlüsse bzw. Kanäle) wieder freigeben. Mit einer weiteren Lötbrücke ist es möglich, die Stromversorgung der Motoren von der Versorgungsspannung V_{in} des Arduino-Boards zu trennen und über eine externe Versorgungsspannung V_{ext} zu speisen.



Arduino-Motor-Shield Rückseite

Die verschiedenen Funktionen des Motor-Shields sind über mehrere digitale sowie analoge Kanäle ansprechbar. Die entsprechenden (fest verdrahteten) Adressen sind in der folgenden Tabelle aufgeführt:

Adressen des Arduino Motor-Shields R3

Funktion bzw. Signal	*Kanal A*	*Kanal B*
Richtung (Direction)	12	13
Pulsweitenmodulation (PWM)	3	11
Bremse (Brake)	9	8
Messung Motorstrom	A0	A1

Mit den zwei Kanälen A und B kann man je einen Gleichstrommotor ansteuern. Die Drehrichtung wird über Pin 12 und 13 des Arduino-Boards eingestellt, die Drehgeschwindigkeiten über die PWM-Ausgänge mit Pin 3 bzw.

11. Diese Adressen sind fest vergeben und können im Programm wie folgt als Konstante definiert werden:

```
const byte DIRA = 12;
const byte DIRB = 13;
const byte PWMA =  3;
const byte PWMB = 11;
```

In der Funktion `setup` sind diese vier digitalen Anschlüsse als Ausgänge zu definieren:

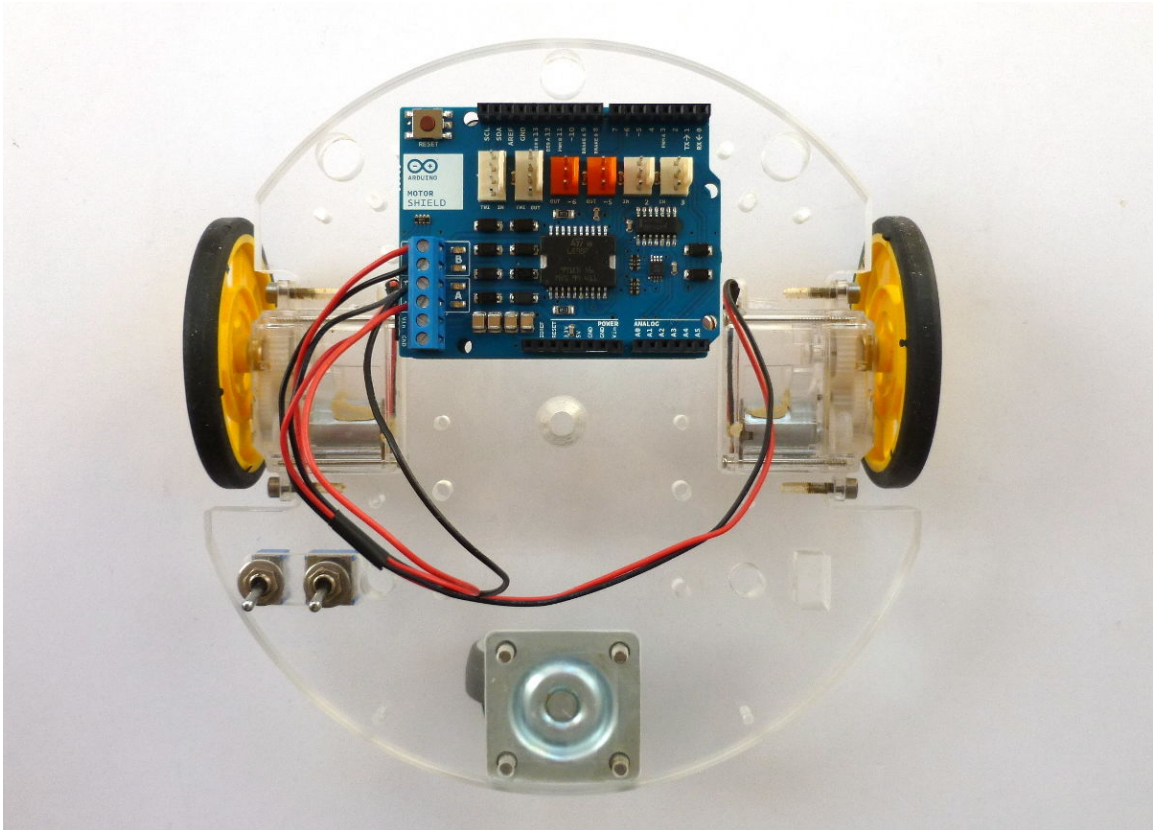
```
void setup()
{
  pinMode (DIRA, OUTPUT);
  pinMode (DIRB, OUTPUT);
  pinMode (PWMA, OUTPUT);
  pinMode (PWMB, OUTPUT);
}
```

Im Hauptprogramm, also in der Funktion `loop`, würde man Drehrichtung und -geschwindigkeit der Motoren nach den eigenen Wünschen einstellen bzw. anpassen. Im nächsten Programmbeispiel werden die Pins für die Drehrichtung auf LOW gesetzt. Die PWM-Ausgabe erfolgt über die Funktion `analogWrite`, die Werte im Bereich von 0 bis 255 erwartet. Dabei entspricht der Wert 0 dem Stillstand des Motors, der Wert 255 liefert die maximale Geschwindigkeit. Mit dem im Programmabschnitt verwendeten Wert von 127 ist mit etwa halber Geschwindigkeit zu rechnen:

```
void loop()
{
  // Drehrichtung
  digitalWrite (DIRA, LOW);
  digitalWrite (DIRA, LOW);
  // halbe Geschwindigkeit
  analogWrite (PWMA, 127);
  analogWrite (PWMB, 127);
}
```

Beim Prototypaufbau wird der linke Motor über Kanal A angesteuert, der rechte über Kanal B. Im mobilen Roboter sind die Motoren mit entgegengesetzter Orientierung eingebaut. Bei einer gradlinigen Vorwärts- bzw. Rück-

wärtsbewegung sollen sich beide Räder (rechts und links) in die gleiche Richtung drehen, wozu sich die Motoren in die jeweils entgegengesetzte Richtung drehen müssen. Die Motoren wurden beim Prototyp so angeschlossen, dass bei der im o.g. Programmabschnitt eingestellten Drehrichtung beide Motoren zu einer Vorwärtsbewegung beitragen.



Roboter-Prototyp mit Arduino-Motor-Shield

Zum Anhalten der Motoren gibt man entweder die Null als (quasi analoges) PWM-Signal aus

```
analogWrite (PWMA, 0);
analogWrite (PWMB, 0);
```

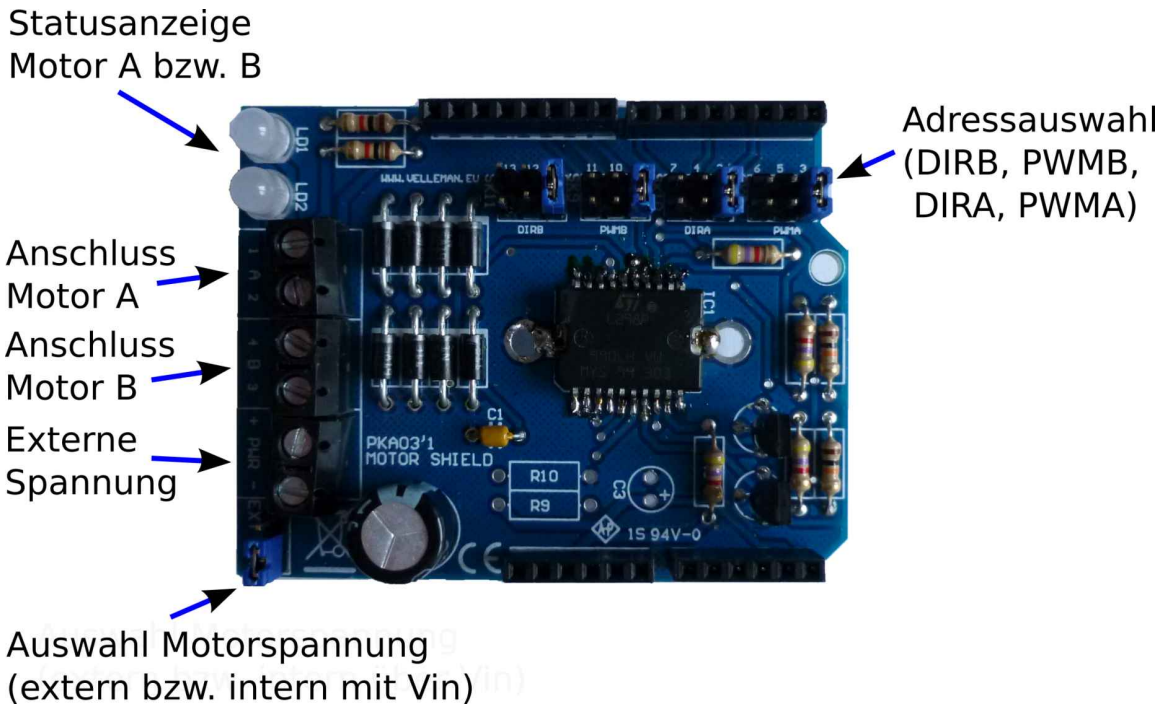
oder setzt die Ausgänge (als digitales Signal) auf LOW-Pegel:

```
digitalWrite (PWMA, LOW);
digitalWrite (PWMB, LOW);
```

Zum Einsatz der Bremsen muss man Pin 8 und 9 als (digitale) Ausgänge deklarieren. Bei `LOW`-Pegel ist die Bremse inaktiv, bei `HIGH`-Pegel wird aktiv gebremst. Bei der Nutzung von Getriebemotoren ist eine elektronische Bremse kaum erforderlich; durch die Untersetzung des Getriebes macht sich das normale Auslaufen des Motors kaum bemerkbar. Verzichtet man auf die Bremse, muss man Pin 8 und 9 nicht konfigurieren: Zwei auf dem Motor-Shield vorgesehenen Pull-Down-Widerstände ziehen die entsprechenden Pins auf Masse und damit auf `LOW`-Pegel. Will man Pin 8 bzw. 9 anderweitig nutzen, muss man die entsprechenden Lötbrücken auf der Rückseite des Motor-Shields durchtrennen.

3.3 Motor-Shield von Velleman

Ein ebenfalls verbreitetes Motor-Shield wird von Velleman angeboten. Dieses Motor & Power Shield ist sowohl als Bausatz (KA03) als auch als fertig montierte Platine (VMA03) verfügbar [5,6].



Motor-Shield KA03 von Velleman

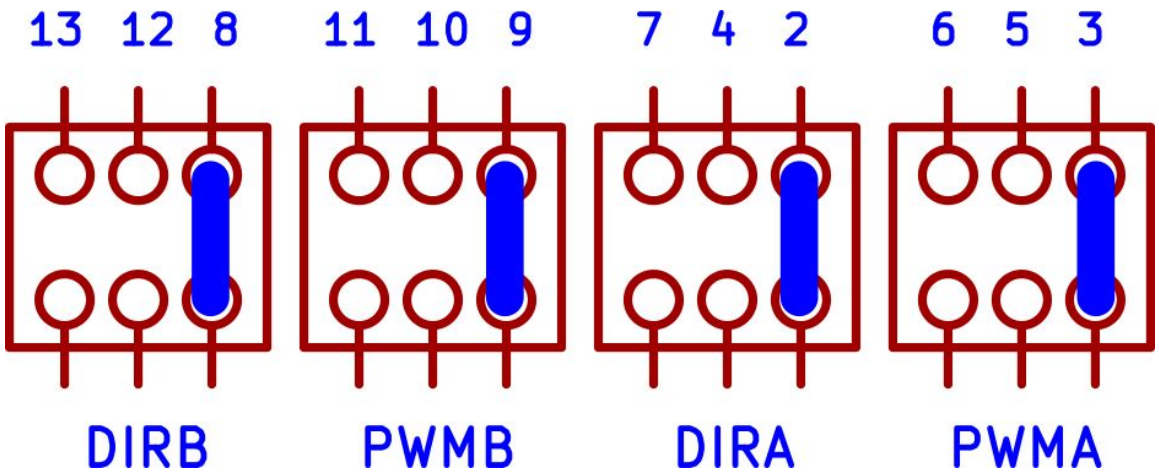
Das Shield erlaubt auch die Ansteuerung von zwei Gleichstrommotoren oder einem Schrittmotor. Zur Auswahl der Richtungen (DIRA, DIRB) bzw. der PWM-Ansteuerung (PWMA, PWMB) stehen verschiedene digitale Kanäle des Arduino-Boards zur Verfügung. Die Auswahl erfolgt über Jumper (Steckbrücken):

Adressen des Motor-Shields KA03 von Velleman

Signale	*Kanäle*
DIRA	2, 4, 7
DIRB	8, 12, 13
PWMA	3, 5, 6
PWMB	9, 10, 11
Motorspannung	A5

Ein weiterer Jumper erlaubt die Auswahl der Motorspannung, die entweder als externe Spannung V_{ext} oder intern über die Eingangsspannung V_{in} des Arduino-Boards bereitgestellt wird. Die Nutzung einer externen Spannung V_{ext} ist beispielsweise für 24V-Motoren interessant. Unabhängig von der Auswahl der Spannungsquelle kann die Motorspannung über den Analogkanal A5 gemessen werden. Dazu ist auf der Leiterplatte ein aus den Widerständen R9 und R10 bestehender Spannungsteiler vorgesehen, wobei das Messsignal zusätzlich mit dem Kondensator C3 geglättet wird. In der gezeigten Baugruppe wurden diese Bauelemente nicht bestückt.

Für Drehrichtung und PWM-Ansteuerung sind pro Kanal drei Möglichkeiten vorgesehen. Dadurch könnte man bis zu drei Motor-Shields übereinander stecken und so bis zu 6 Gleichstrommotoren ansteuern. Zur Durchführung eines ersten Tests wollen wir uns mit einem (einzigen) Motor-Shield begnügen und dazu folgende Pin- bzw. Adressauswahl treffen:

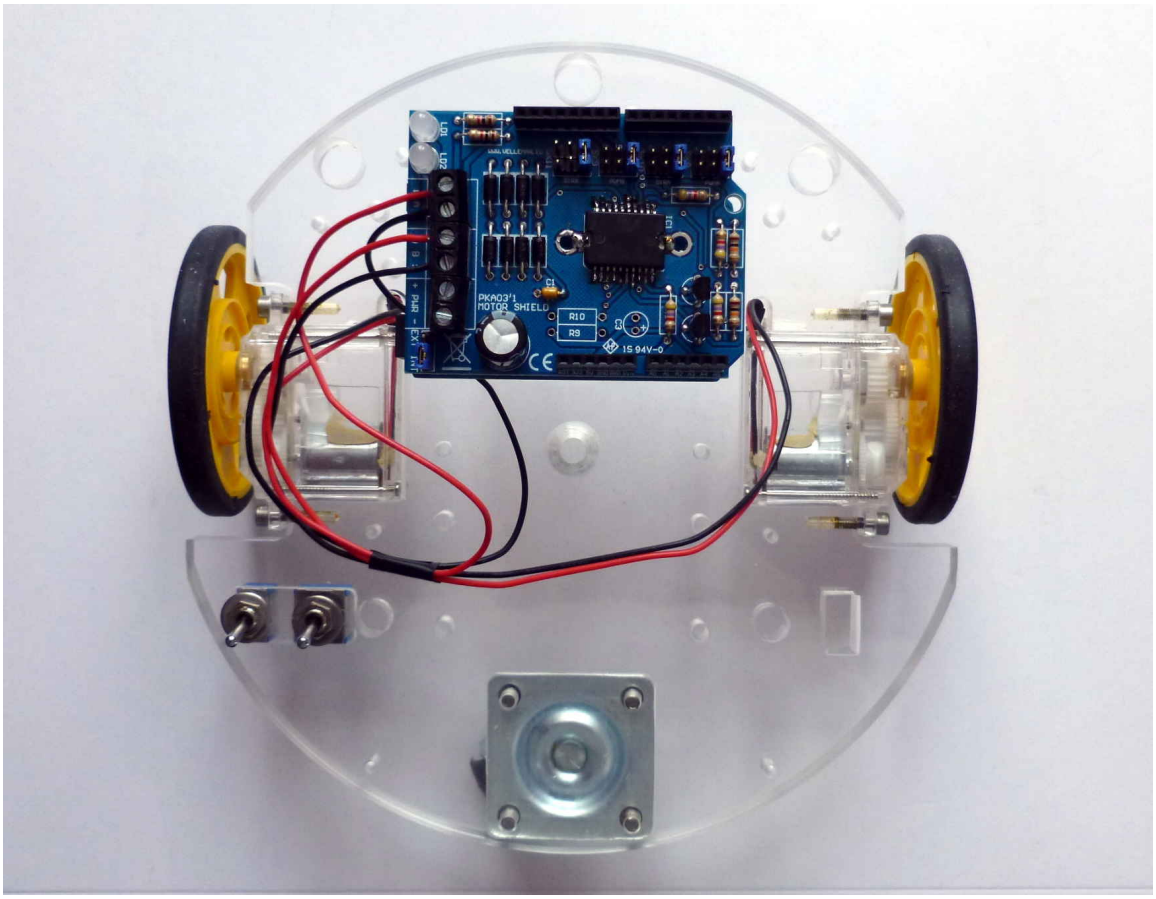


Standard-Adressauswahl für das Motor-Shield KA03 von Velleman

Diese Pinbelegung würde man im Programm so verankern:

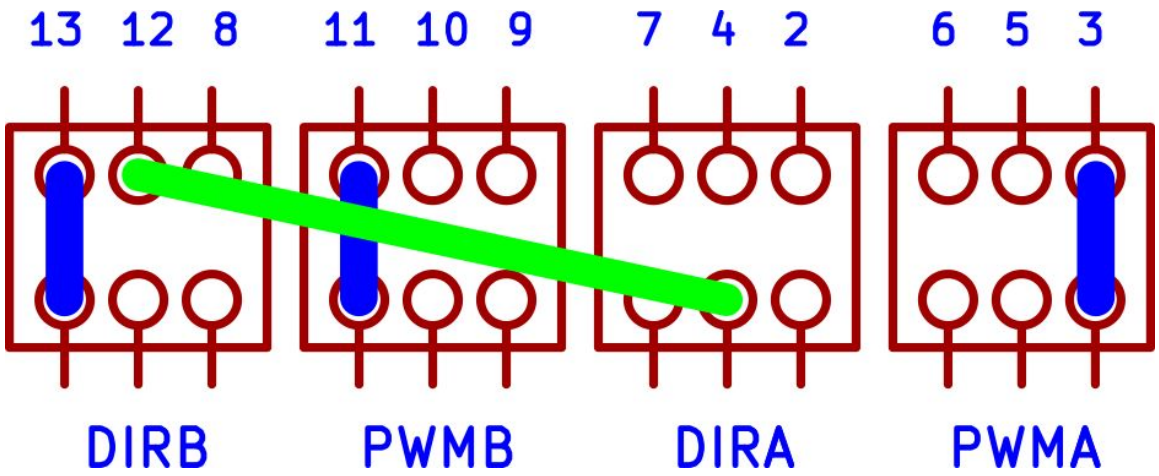
```
const byte DIRA = 2;
const byte DIRB = 8;
const byte PWMA = 3;
const byte PWMB = 9;
```

Mit dieser Adressanpassung kann man das für das Arduino-Motor-Shield erstellte Programm zur Geradeausfahrt auch für den Test des Velleman-Motor-Shields nutzen. Zur Anzeige der Drehrichtung sind auf dem Velleman-Motor-Shield die zweifarbigen LEDs LD1 und LD2 vorgesehen, die grün oder rot leuchten.



Roboter-Prototyp mit Velleman Motor Shield

Die unterschiedlichen Pinbelegungen zwischen dem offiziellen Arduino-Motor-Shield und dem Motor-Shield von Velleman wirken sich nicht nur auf die eigentliche Motoransteuerung aus, sondern auch auf die für andere Zwecke vorgesehenen Anschlüsse (z.B. für LCD-Modul, Taster, Sensoren, …). Da die Pins des offiziellen Arduino-Motor-Shields R3 fest belegt sind bietet es sich an, die Adressauswahl beim Motor-Shield von Velleman anzupassen. Bei drei der vier Signale ist die entsprechende Pinauswahl mit den vorgesehenen Steckbrücken unmittelbar möglich (siehe Abbildung). Lediglich für die Einstellung der Drehrichtung von Kanal A mit dem Signal `DIRA` ist eine gesonderte Verbindung vorzusehen. Dazu kann man beispielsweise auf der Leiterseite des Motor-Shields einen passenden Draht anlöten.



Adressauswahl für das Motor-Shield KA03 von Velleman in Übereinstimmung mit dem Arduino-Motor-Shield

3.4 C++ Klasse zur Motoransteuerung

Will man für die Bewegung des mobilen Roboters beide Motoren ansteuern, so sind bei der Verwendung elementarer Arduino-Kommandos vier Funktionsaufrufe nötig (zwei mit `digitalWrite` zur Richtungseinstellung und zwei von `analogWrite` zur PWM-Ausgabe). In diesem Abschnitt wird Schritt für Schritt eine C++ Klasse zur einfachen Handhabung der Motoransteuerung erstellt. Im nachfolgenden Abschnitt wird die dabei entstandene Klasse als Bibliothek in das Arduino-System eingepflegt.

Wir legen als erstes die Klasse `Motor` in einer Minimalfassung an:

```
class Motor {
  // Pins für Drehrichtung
  byte DIRA, DIRB;
  // Pins für PWM
  byte PWMA, PWMB;
  public:
  Motor (byte, byte, byte, byte);
};
```

Die Klasse beinhaltet zunächst die Attribute `DIRA`, `DIRB`, `PWMA` und `PWMB`, welche die zur Ansteuerung des Motor-Shields erforderlichen Pins charakterisieren. Die konkrete Adressbelegung wird mit dem Konstruktor

zugewiesen, dessen Aufruf vier Argumente vom Typ `byte` benötigt. Die beim Aufruf übergebenen Werte werden den o.g. Attributen zugewiesen:

```
Motor::Motor (byte pinDIRA, byte pinDIRB,
              byte pinPWMA, byte pinPWMB)
{
    DIRA = pinDIRA;
    DIRB = pinDIRB;
    PWMA = pinPWMA;
    PWMB = pinPWMB;
}
```

Alternativ kann man diese Zuweisungen auch über eine Initialisierungsliste implementieren (siehe z.B. [7]). Zusätzlich werden in dieser Fassung auch beide Motoren abgeschaltet:

```
Motor::Motor (byte pinDIRA, byte pinDIRB,
              byte pinPWMA, byte pinPWMB)
: DIRA(pinDIRA), DIRB(pinDIRB),
  PWMA(pinPWMA), PWMB(pinPWMB)
{
   digitalWrite (PWMA, 0);
   digitalWrite (PWMB, 0);
}
```

Im eigenen Programmcode würde man von dieser Klasse `Motor` eine Instanz (hier: `motor`) anlegen und dabei die zur Motoransteuerung relevanten Pins übergeben. Die in diesem Beispiel übergebenen Werte entsprechen der Adressbelegung des offiziellen Arduino-Motor-Shields R3:

```
Motor motor(12, 13, 3, 11);
```

Diese sehr verbreitete Pinbelegung könnte man beispielsweise auch für den ohne Argumente aufzurufenden Standard-Konstruktor vorsehen. Dazu würde man in der Klassendefinition für den o.g. Konstruktor entsprechende Standardwerte vorgeben:

```
Motor (byte=12, byte=13, byte=3, byte=11);
```

Der Aufruf ohne Argumente würde die Klasse mit der hinterlegten Standard-Belegung anlegen. Das Vorhandensein eines Standard-Konstruktors ist auch

dann günstig, wenn man eine von der Motor-Klasse abgeleitete Klasse anlegen will.

```
Motor motor;
```

Die benötigten Pins sind jetzt in der Klasse hinterlegt, müssen aber noch als Ausgänge konfiguriert werden. Dazu erweitert man die Klasse `Motor` um die Methode `begin`, die weder Argument und Rückgabewert besitzt:

```
void Motor::begin()
{
  pinMode (DIRA, OUTPUT);
  pinMode (DIRB, OUTPUT);
  pinMode (PWMA, OUTPUT);
  pinMode (PWMB, OUTPUT);
}
```

Der Aufruf muss in der `setup`-Funktion erfolgen:

```
void setup()
{
  motor.begin();
}
```

Als nächstes definieren wir die Methode `setValues`, mit der Richtung und Geschwindigkeit eingestellt werden. In der Klassendefinition wird diese Methode mit `private` markiert, so dass sie nur innerhalb der Klasse verwendet werden kann.

```
  private:
  void setValues (byte, byte, int);
```

Zusätzlich definieren zwei Konstante `VMAX` und `VMIN`, die den maximalen und minimalen Wert für die PWM und damit für die Winkelgeschwindigkeit bzw. Drehzahl beschreiben. Bei einer 8-Bit-PWM sind nur Werte von 0 bis 2^8-1=255 möglich. Die Boards mit ARM-Architektur erlauben auch eine 12-Bit-PWM, bei der Werte bis 2^{12}-1=4095 möglich sind. Falls man die Motoren nicht (dauerhaft) mit der maximalen Betriebsspannung betreiben möchte, kann man für `VMAX` auch einen kleineren Wert vorsehen, z.B. 127 für im Mittel halbe Betriebsspannung. Umgekehrt dreht sich bei zu kleiner

Spannung bzw. zu kleinem Tastverhältnis der Motor typischerweise nicht mehr; der Strom wird nur noch in Wärme umgesetzt. Dieser Fall ist ebenso nicht wünschenswert.

```
// Werte für max. und min. Geschwindigkeit
#define VMAX  255
#define VMIN   40
```

Für die Methode `setValues` sind drei Werte zu übergeben. In Ergänzung zu den relevanten Pins werden über die Variable `speed` Drehrichtung und -geschwindigkeit eingestellt. Das Vorzeichen des Arguments `speed` gibt die Drehrichtung an. Dabei ist ein positiver Wert für die Vorwärtsdrehung und ein negativer Wert für die Rückwärtsdrehung vorgesehen. Der absolute Betrag der Variablen `speed` entspricht der über die PWM eingestellten Geschwindigkeit, welche nach oben durch `VMAX` begrenzt ist und für Werte unterhalb von `VMIN` auf Null gesetzt wird.

```
void Motor::setValues (byte pinDir, byte pinPwm, int speed)
{
    // Drehrichtung
    digitalWrite (pinDir, speed>=0 ? LOW : HIGH );
    // Geschwindigkeit
    speed=abs(speed);
    speed=min(speed,VMAX);
    if (speed<VMIN) speed=0;
    analogWrite(pinPwm, speed);
}
```

Mit der Methode `setValues` ist man unter Berücksichtigung der als Klassenattribute hinterlegten Pinbelegung in der Lage, die zwei Motoren anzusteuern. Dazu werden direkt in der Klassendefinition die Methoden `writeA` und `writeB` als Inline-Funktionen definiert:

```
// Motor A: links
void writeA (int speed)
     { setValues (DIRA, PWMA, speed); };
// Motor B: rechts
void writeB (int speed)
     { setValues (DIRB, PWMB, speed); };
```

Mit der Methode `write` kann man durch einem einzigen Funktionsaufruf beide Motoren beeinflussen:

```
void Motor::write (int speedA, int speedB)
{
   writeA(speedA);
   writeB(speedB);
}
```

Der entsprechende Prototyp der Funktion muss natürlich auch in der Klassendefinition aufgeführt sein. Zudem ist darauf zu achten, dass die Funktionen `writeA`, `writeB` und `write` hinsichtlich ihrer Zugriffsrechts als `public` deklariert sind. Mit dem nachfolgenden Testprogramm zeigt der mobile Roboter die verschiedenen Bewegungsmöglichkeiten:

```
int V=200; // PWM-Wert
int T=500; // Zeit in ms
void loop()
{
    // vorwärts
    motor.write ( V, V); delay(T);
    // rechts
    motor.write ( V,-V); delay(T);
    // rückwärts
    motor.write (-V,-V); delay(T);
    // links
    motor.write (-V, V); delay(T);
}
```

3.5 Bibliothek zur Motoransteuerung

Bei einem mobilen Roboter werden die Motoren vermutlich bei fast jeder Anwendung zum Einsatz kommen. Um nicht jedes mal den entsprechenden Code für die Motoransteuerung kopieren zu müssen bietet es sich an, die im vorangegangenen Abschnitt erstellte Klasse `Motor` als Bibliothek zu hinterlegen. Die Bibliothek ist auf Github verfügbar [8]:

https://github.com/roebenack/Motor

Nach Herunterladen der ZIP-Datei kann die Bibliothek direkt über die Arduino IDE mittels **Sketch > Bibliotheken einbinden > .ZIP-Bibliothek hinzufügen** eingebunden werden. Zum besseren Verständnis wird nachfolgend der Aufbau der Bibliothek detailliert erläutert.

Mit der Installation der Arduino-Entwicklungsumgebung wird ein Ordner für eigene Programmskizzen – `sketchbook` – angelegt. Neben den selbst erstellten Programmen enthält der Ordner auch ein Verzeichnis namens `libraries`, welches zunächst keine weiteren Einträge enthält. In dem Verzeichnis `libraries` legen wir den neuen Ordner `Motor` an, der die Dateien für die Motor-Bibliothek enthalten soll (siehe [9]).

Die Datei `Motor.h` ist eine sogenannte *Headerdatei*, welche die Typdefinitionen der in der Bibliothek bereitgestellten Klassen bzw. Funktionen enthält. Durch die Sequenz

```
#ifndef Motor_h
#define Motor_h
...
#endif
```

wird verhindert, dass diese Definitionsdatei (aus Versehen) mehrfach abgearbeitet wird. Die für die Arduino-Umgebung spezifischen Deklarationen (wie beispielsweise die Funktion `digitalWrite`) werden über Headerdatei `Arduino.h` eingebunden:

```
#include "Arduino.h"
```

Vor Einbinden der Headerdatei kann man die Konstanten `VMIN` bzw. `VMAX` selbst definieren. Andernfalls werden sie in der Bibliothek definiert. Zusammen mit der Definition der Klasse `Motor` ergibt sich insgesamt die folgende Headerdatei, die in dem o.g. Ordner `Motor` abzuspeichern ist:

```
/*
    Motor.h - Library for motor control
    Created by Klaus Röbenack, 2015, 2019
*/

#ifndef Motor_h
#define Motor_h
```

```
#include "Arduino.h"

#ifndef VMAX
#define VMAX 255
#endif

#ifndef VMIN
#define VMIN  40
#endif

class Motor {
    protected:
    // Pins for direction
    byte DIRA, DIRB;
    // Pins for PWM
    byte PWMA, PWMB;
    public:
    // Constructor
    Motor (byte=12, byte=13, byte=3, byte=11);
    // Initialization
    void begin();
    // Motor A: left
    void writeA (int speed) { setValues (DIRA, PWMA, speed); };
    // Motor B: right
    void writeB (int speed) { setValues (DIRB, PWMB, speed); };
    // Motor A and B
    void write (int, int);
    // Typical maneuvers
    void forward  (byte speed=VMAX) {write (+speed,+speed); };
    void backward (byte speed=VMAX) {write (-speed,-speed); };
    void right (byte speed=VMAX)    {write (+speed,-speed); };
    void left  (byte speed=VMAX)    {write (-speed,+speed); };
    void stop ()    {write (0,0); };
    private:
    void setValues (byte, byte, int);
};

#endif
```

Im Unterschied zu der im vorangegangenen Abschnitt vorgestellten Klassendefinition wurde jetzt am Anfang das Zugriffsschlüsselwort `protected` ein-

gefügt. Ein direkter Zugriff auf die nachfolgenden Klassenelemente ist dann von außen nicht mehr möglich. Allerdings können abgeleitete Klassen auf diese Elemente zugreifen.

Während die Headerdatei `Motor.h` nur die Definitionen enthält, erfolgt die eigentliche Implementierung über die C++ Datei `Motor.cpp`, die ebenfalls im Ordner `Motor` abzulegen ist:

```
/*
    Motor.cpp - Library for motor control
    Created by Klaus Röbenack, 2015, 2019
*/

#include "Arduino.h"
#include "Motor.h"

// Constructor
Motor::Motor (byte pinDIRA, byte pinDIRB,
              byte pinPWMA, byte pinPWMB)
: DIRA(pinDIRA), DIRB(pinDIRB), PWMA(pinPWMA), PWMB(pinPWMB)
{
    digitalWrite (PWMA, 0);
    digitalWrite (PWMB, 0);
}

// Initialization
void Motor::begin()
{
    pinMode (DIRA, OUTPUT);
    pinMode (DIRB, OUTPUT);
    pinMode (PWMA, OUTPUT);
    pinMode (PWMB, OUTPUT);
}

// Internal function
void Motor::setValues (byte pinDir, byte pinPwm, int speed)
{
    // Direction
    digitalWrite (pinDir, speed>=0 ? LOW : HIGH );
    // Speed (PWM value)
    speed=abs(speed);
```

```
    speed=min(speed,VMAX);
    if (speed<VMIN) speed=0;
    analogWrite(pinPwm, speed);
}

// Speed for both motors
void Motor::write (int speedA, int speedB)
{
    writeA(speedA);
    writeB(speedB);
}
```

Für den Test der Bibliothek erstellen wir aus den verbleibenden Code-Schnipseln des letzten Abschnitts ein kleines, eigenständiges Programm. Nach einem Neustart der Arduino-Umgebung sollte die Bibliothek unter **Sketch > Import Library** aufgeführt sein. Wählt man darin die Bibliothek `Motor` aus, so wird die entsprechende Headerdatei (hier: `Motor.h`) in den Quelltext eingebunden:

```
/*
    Test of Motor Library
    Created 2015, 2019 by Klaus Röbenack
*/

#include <Motor.h>

Instanz der Klasse Motor erzeugen
Motor motor(12, 13, 3, 11);

void setup()
{
  motor.begin();
}

int V=200; // PWM-Wert
int T=500; // Zeit in ms

void loop()
{
  // vorwärts
  motor.write ( V, V); delay(T);
```

```
  // rechts
  motor.write ( V,-V); delay(T);
  // rückwärts
  motor.write (-V,-V); delay(T);
  // links
  motor.write (-V, V); delay(T);
}
```

Diesen Beispielcode kann man für die Bibliothek auch direkt hinterlegen. Dazu erstellt man im Ordner `Motor` das Verzeichnis `examples` und legt das Beispiel als neues Verzeichnis – z.B. `MotorTest1` – darin ab. Man kann für eine Bibliothek auch mehrere Beispiele vorsehen. Nach dem nächsten Start der Arduino-Umgebung ist das Beispiel in der allgemeinen Beispielsammlung zu finden, nämlich unter **Datei > Beispiele > Motor > Examples > MotorTest1**. In einem weiteren Beispiel `MotorTest2` kann man in der Hauptschleife die zusätzlichen Methoden für die gängigen Grundmanöver (Vorwärts- und Rückwärtsfahrt, Rechts- und Linksdrehung) nutzen:

```
void loop() {
    motor.forward();   delay(T);
    motor.right();     delay(T);
    motor.backward();  delay(T);
    motor.left();      delay(T);
}
```

Zu der erstellten Bibliothek sind verschiedene Erweiterungsmöglichkeiten denkbar. Anstelle einer direkten Erweiterung der Motor-Bibliothek könnte man zusätzliche Funktionen auch in einer abgeleiteten Klasse implementieren.

3.6 Quellenangabe

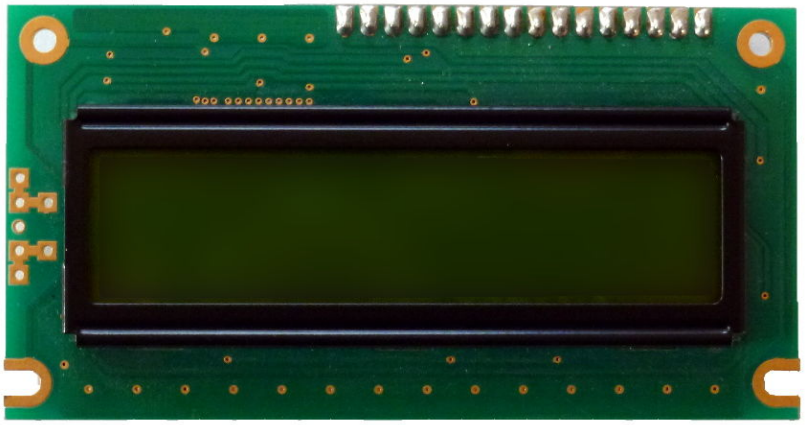
1. Pulsweitenmodulation. In: Wikipedia, Die freie Enzyklopädie. URL: http://de.wikipedia.org/w/index.php?title=Pulsweitenmodulation
2. L293, L293D; Quadruple Half-H Drivers. Datenblatt, Texas Instruments, 2002.
3. L298; Dual Full-Bridge Driver. Datenblatt, STMicroelectronics, 2000.

4. Arduino Motor Shield. URL: http://arduino.cc/en/Main/ArduinoMotorShieldR3
5. KA03 Motor & Power shield Arduino®. Illustrated assembly manual, Velleman.
6. VMA03 Motor & Power shield Arduino®. Manual, Velleman.
7. N. Josuttis: Objektorientiertes Programmieren in C++; Ein Tutorial für Ein- und Umsteiger. Addison-Wesley Verlag, 2. Auflage, 2001.
8. K. Röbenack: *Motor Library for Arduino*. URL: https://github.com/roebenack/Motor
9. Writing a Library for Arduino. URL: http://arduino.cc/en/Hacking/LibraryTutorial.

4 LCD-Anzeige

4.1 Anschluss der LCD-Anzeige

Zur Ansteuerung von alphanumerischen LCD-Displays hat sich der von Hitachi entwickelte Schaltkreis HD44780 als de facto Industriestandard herausgebildet [1]. Dieser Schaltkreis bzw. seine weitgehend oder voll kompatiblen Nach- bzw. Weiterentwicklungen werden heutzutage zur Ansteuerung von LCD-Modulen verschiedenster Größen verwendet. Gängige Displayformate sind 8×2, 16×2 bis 20×4 (Spalten × Zeilen).



LCD-Modul mit 16 Spalten und 2 Zeilen (16×2)

Bei den Prototypaufbauten des Roboters wurden Displaymodule der letztgenannten Größe, also mit 20 Spalten und 4 Zeilen, verwendet. Die meisten Displays weisen zudem auch die gleiche Anschlussbelegung auf:

Typische Anschluss- bzw. Adressbelegung von LCD-Modulen

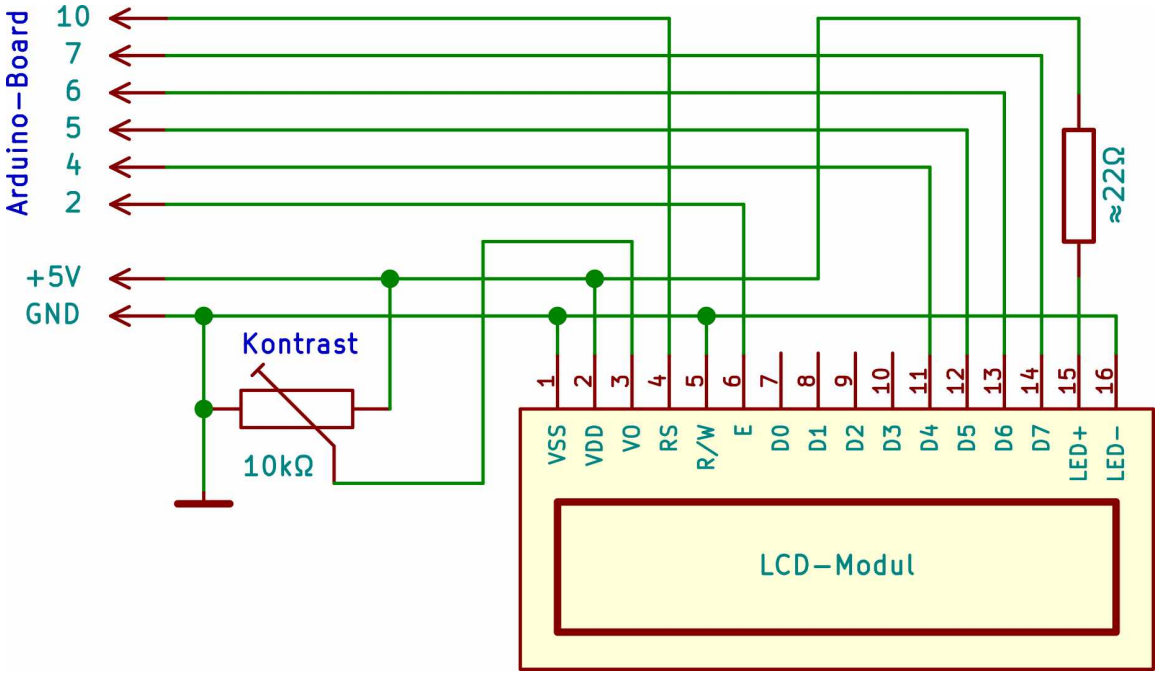
Pin-Nummer	*Symbol*	*Funktion bzw. Bedeutung*
1	V_{SS}	Masse (0V)
2	V_{DD}	Betriebsspannung (+5V)
3	V_0	Kontrast
4	RS	Registerauswahl (Befehl: 0, Daten: 1)
5	R/W	Read/Write (schreiben: 0, lesen: 1)
6	E	Enable
7-14	DB0-DB7	8-Bit Datenbus
15	A	Betriebsspannung Hintergrundbeleuchtung
16	K	Masse für Hintergrundbeleuchtung

Zur Ansteuerung sind 8 Datenbits vorgesehen in Verbindung mit einigen Steuersignalen. Neben dem 8-Bit-Betrieb ist auch ein 4-Bit-Betrieb möglich, bei dem man mit entsprechend weniger Anschlüssen auskommt. Typischerweise verzichtet man auf die Ansteuerung des R/W-Anschlusses. Damit werden nur 6 digitale Anschlüsse benötigt. Berücksichtigt man die bereits vom Motor-Shield belegten digitalen Anschlüsse, dann wäre das nachfolgend angegebene Belegungsschema denkbar:

Belegung der digitalen Signale am Arduino-Board für den Anschluss eines LCD-Moduls

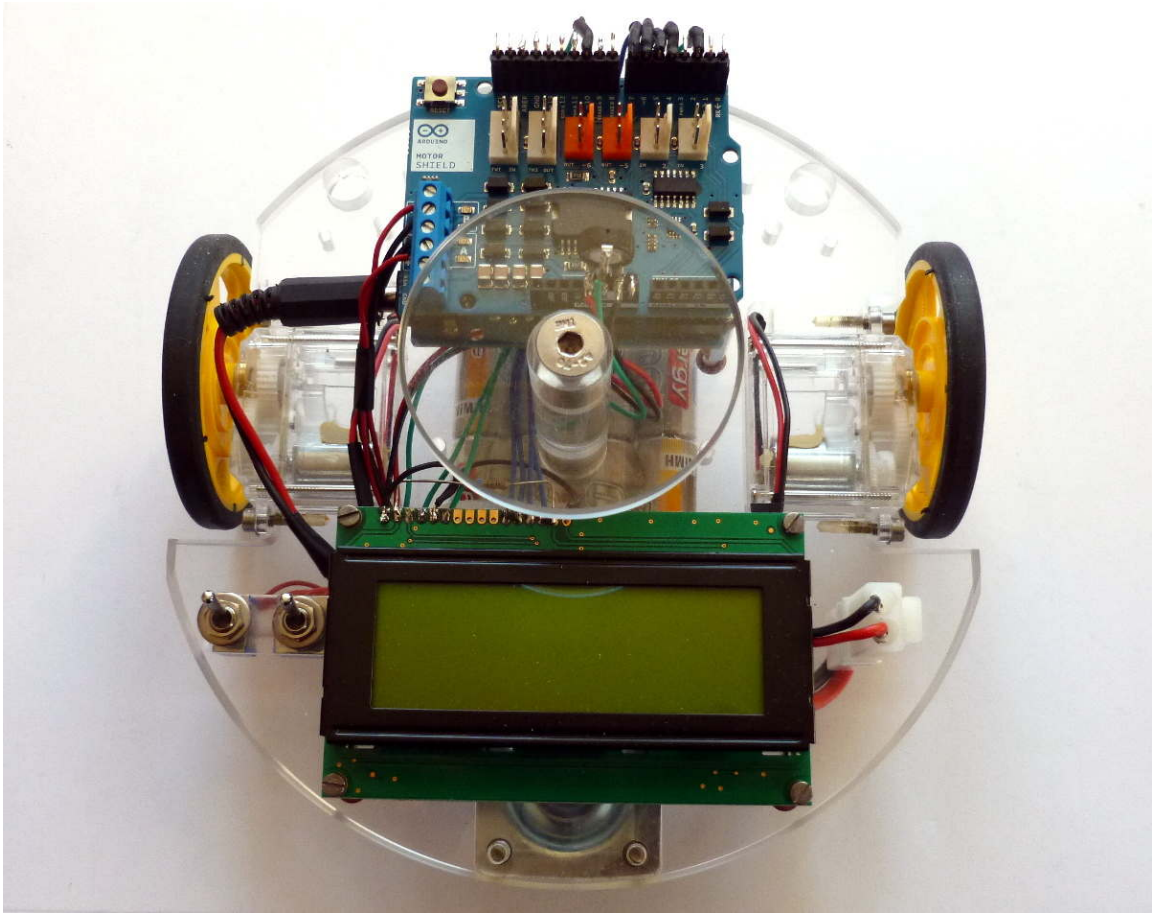
Digitale Anschlüsse (Arduino)	*Signale am LCD-Modul*
10	RS
2	E (Enable)
4	D4
5	D5
6	D6
7	D7

Beim Anschluss eines LCD-Moduls sind neben den eben genannten Datenleitungen auch Masse (GND) und Betriebsspannung (5V) zu berücksichtigen. Der Kontrast des Displays kann über einen 10kΩ-Einstellregler vorgegeben werden. Ggf. ist eine eventuell vorhandene Hintergrundbeleuchtung zu berücksichtigen. Diese wird typischerweise mit LEDs realisiert und benötigt noch einen Vorwiderstand im Bereich einiger Ohm [2].



Verdrahtung des LCD-Moduls

Das nachfolgende Bild zeigt einen Prototypaufbau des Roboters mit dem 20×4-Display AV2040 von ANAG VISION [2].



Roboter mit montiertem LCD-Modul

4.2 Ansteuerung der LCD-Anzeige

Zur Ansteuerung des LCD-Modules bindet man zunächst die Bibliothek für LCD-Anzeigen ein, welche die Klasse `LiquidCrystal` bereitstellt [3]:

```
#include <LiquidCrystal.h>
```

Anschließend wird von der Klasse `LiquidCrystal` das Objekt `lcd` instanziiert. Dabei gibt man auch die bei der Verdrahtung mit dem Arduino-Board verwendeten Pins an. Hierbei ist die Reihenfolge der verwendeten Signale einzuhalten (RS, Enable, D4, D5, D6, D7):

```
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

In der Funktion `setup` muss das LCD-Modul initialisiert werden. Dazu stellt die Klasse `LiquidCrystal` die Methode `begin` zur Verfügung, wobei Zeilen- und Spaltenzahl des Moduls zu übergeben sind, z.B. 20 Spalten und 4 Zeilen:

```
lcd.begin(20, 4);
```

Die Methode `print` übernimmt die Ausgabe. Zulässige Datentypen sind `char`, `byte`, `int`, `long` bzw. `string`.

```
lcd.print("Test LCD-Anzeige!");
```

Das gesamte Programm für den Test des LCD-Moduls sieht dann folgendermaßen aus:

```
// Bibliothek für LCD-Module
#include <LiquidCrystal.h>

// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);

void setup() {
  // Spalten- und Zeilenzahl
  lcd.begin(20, 4);
  // Textausgabe
  lcd.print("Test LCD-Anzeige!");
}

void loop() {}
```

Die Klasse `LiquidCrystal` enthält zahlreiche weitere Methoden zur Steuerung und Formatierung der Ausgabe. Die aus Sicht des Autors wichtigsten Funktionen sind nachfolgend aufgeführt:

Wichtige Methoden der Klasse LiquidCrystal [3]

Methode, Aufruf	*Beschreibung*
`begin(x,y)`	Initialisiert Display mit x Spalten und y Zeilen
`clear()`	Löscht Bildschirm und setzt den Kursor in die linke obere Ecke
`home()`	Setzt Kursor in die linke obere Ecke
`setCursor(x,y)`	Setzt Kursor auf Spalte x und Zeile y (beginnend ab Null)
`write(c)`	Ausgabe eines einzelnen Zeichens (`char`)
`print(daten)`	Ausgabe für zahlreiche Datentypen
`print(daten,Basis)`	Zahlenausgabe mit Wahl der Basis (`BIN`, `OCT`, `DEC`, `HEX`)

4.3 Möglichkeiten zur Formatierung der Ausgabe

Zur Formatierung der Ausgabe kann man die Standard C/C++ Bibliothek `stdio.h` nutzen [4]. Für diese Bibliothek muss man die zugehörige Headerdatei einbinden:

```
#include <stdio.h>
```

Die Ausgabe würde man auf einem PC mit der Funktion `printf` vornehmen, die aber nicht unmittelbar für die Ausgabe auf einem LCD-Display vorgesehen ist. Als Alternative kann man die sehr ähnliche Funktion `sprintf` einsetzen. Dazu deklariert man eine Zeichenkette der gewünschten (maximalen) Länge als Feld vom Typ `char`. Hierbei ist ein zusätzlicher Eintrag für das Ende der Zeichenkette vorzusehen. Das Ende einer Zeichenkette wird C-intern mit einer Null markiert. Für das verwendete LCD-Modul mit 20 Spalten pro Zeile würde man ein Feld der Länge 21 anlegen:

```
char zeile[21];
```

Die Funktion `sprintf` nimmt selber keine direkte Ausgabe vor, sondern speichert das Ergebnis in der im ersten Argument als Zeiger (Pointer) übergebenen Zeichenkette (hier: `zeile`). Das zweite Argument beim Aufruf von `sprintf` ist eine spezielle Zeichenkette, die neben "normalen" Zeichen auch Formatierungszeichen für die Ausgabe verschiedener Datentypen enthält. Die jeweilige Ausgabe wird mit dem Formatierungszeichen "%" eingeleitet, ggf. gefolgt von einer Zahl, welche die Länge angibt. Die weiteren Funktionsargumente sind die auszugebenden Größen.

```
sprintf(zeile,"Ganze Zahl: %4d",Wert);
lcd.print(zeile);
```

Zur Anzeige auf dem LCD-Display nutzt man die Methode `print`. Leider funktionierte die Ausgabe bzw. Formatierung von Gleitkommazahlen mit der Funktion `sprintf` nicht auf der Arduino-Plattform. Die Tabelle enthält die gängigen Formatierungsanweisungen.

Wichtige Symbole für die Formatierung mit der Funktion `sprintf`

Symbole	*Beschreibung*
%d, %i	Ausgabe ganzer Zahlen, ggf. mit Vorzeichen
%u	Ausgabe nichtnegativer ganzer Zahlen
%x, %X	Hexadezimale Ausgabe
%c	Ausgabe eines einzelnen Zeichens
%s	Ausgabe einer Zeichenkette

Die Ausgabe von Gleitkommazahlen ist mit der Funktion sprint jedoch über einen Umweg möglich. Dazu wandelt man die Gleitkommazahl mit der Funktion `dtostrf` der AVR-Standardbibliothek in eine Zeichenkette um, die dann mit der Formatierungsangabe "%s" von sprintf weiterverarbeitet werden kann. Der Funktion `dtostrf` sind neben der Gleitkommazahl auch die Zeichenanzahl der Ausgabe (einschließlich Komma), die Anzahl der Nachkommastellen und ein Feld vom Typ `char` für das Ergebnis zu übergeben:

```
// Gleitkommazahl
float f=3.14;
```

```
// Feld für konvertierte Gleitkommazahl
char str_zahl[5];
// Konvertierung Zahl->Zeichenkette
dtostrf(f, 4, 2, str_zahl);
// Ausgabe der Zeichenkette
sprintf(zeile, "Zahl: %s",str_zahl);
```

Die Methode `print` der Klasse `LiquidCrystal` erlaubt zwar die Ausgabe von verschiedenen Datentypen, muss aber für jeden Typen separat (und damit mehrfach) aufgerufen werden. Eine kombinierte Ausgabe von Text mit Zahlen der Typen `int` und `float` könnte beispielsweise folgendermaßen aussehen:

```
int i=4;
float f=3.14;
lcd.print("int: ");
lcd.print(i);
lcd.print(" float: ");
lcd.print(f);
```

In C++ hat man für die Ein- bzw. Ausgabe über die Konsole die Datenströme `cin` und `cout` zur Verfügung, die auf sehr einfache Weise die Ausgabe verschiedener Datentypen ermöglichen. Eine ähnliche Funktionalität stellt die Bibliothek `Streaming` für die Arduino-Plattform zur Verfügung [5].

Die `Streaming`-Bibliothek gehört nicht zu den mit der Arduino-Umgebung ausgelieferten Bibliotheken, kann jedoch über den Bibliotheksverwalter (**Werkzeuge > Bibliotheken verwalten**) direkt heruntergeladen und installiert werden. Alternativ kann man die Bibliothek auch als ZIP-Datei von der Webseite [5] herunterladen und anschließend durch **Sketch > Bibliothek einbinden > .ZIP-Biblithek hinzufügen** in die Arduino-IDE integrieren. Über **Sketch > Bibliothek einbinden** bindet man die Bibliothek in das eigene Programm ein. Im Quelltext wird dabei die Headerdatei eingefügt:

```
#include <Streaming.h>
```

Zu der oben angelegten Instanz `lcd` kann man jetzt die verschiedenen Ausgaben mit dem Operator `<<` leiten, z.B.:

```
lcd<<"int: "<<i<<" float: "<<f;
```

Die `Streaming`-Bibliothek übersetzt diese Zeile in einzelne `print`-Anweisungen und belegt daher zur Laufzeit keinen separaten Speicherplatz.

4.4 Quellenangabe

1. HD44780. In: Wikipedia, Die freie Enzyklopädie. URL: http://de.wikipedia.org/w/index.php?title=HD44780.
2. ANAG VISION AV2040. Datenblatt.
3. Arduino - LiquidCrystal. URL: http://arduino.cc/en/Reference/LiquidCrystal.
4. Printf. In: Wikipedia, Die freie Enzyklopädie. URL: http://de.wikipedia.org/w/index.php?title=Printf.
5. Streaming | Arduiniana". URL: http://arduiniana.org/libraries/streaming/.

5 Abfrage von Tastern und Schaltern

5.1 Tasteranschluss und -abfrage

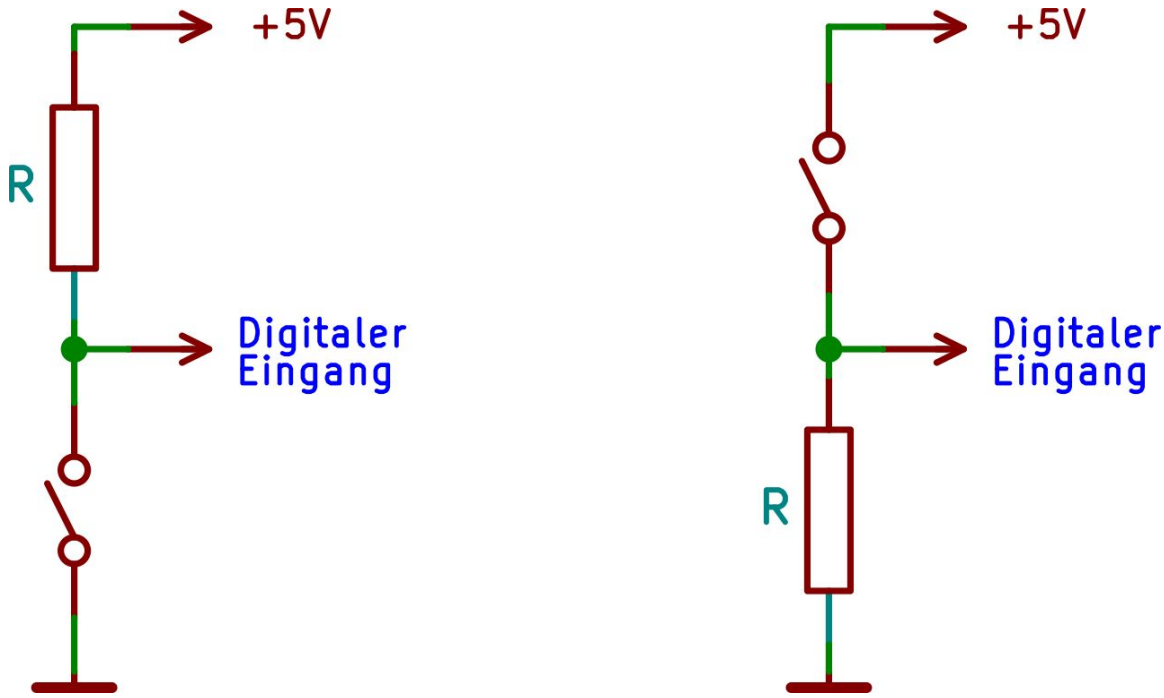
Am Roboter angebrachte Taster bzw. Schalter lassen sich zur Auswahl des jeweils genutzten Programmmodus nutzen. In diesem Kapitel werden Taster eingesetzt, um Begrenzungen des Bewegungsfeldes des Roboters zu erkennen. Zur Abfrage der Taster verwendet man die digitalen Anschlüsse, von denen allerdings schon etliche durch das Motor-Shield bzw. die LCD-Anzeige belegt sind. Freie Adressen wären die digitalen Anschlüsse 0 und 1. Diese Pins sind u.a. beim Uno- bzw. Duemilanove-Board für die Programmübertragung bzw. die serielle Kommunikation vorgesehen, was für den anvisierten Einsatz aber nicht weiter stört.

```
const byte pinTasterA = 0;
const byte pinTasterB = 1;
```

Verzichtet man beim Arduino-Motor Shield auf die Bremsfunktion, könnte man auch Pin 8 und 9 verwenden. Dazu müsste man auf der Rückseite des Motor Shields die mit **BRAKE DISABLE** gekennzeichneten Lötbrücken durchtrennen.

```
const byte pinTasterA = 8;
const byte pinTasterB = 9;
```

Bei der Abfrage von Tastern bzw. Schaltern mit einem digitalen Eingang ist für jede Schalterposition ein definierter Signalpegel zu gewährleisten. Das kann beispielsweise über Pull-Up- bzw. Pull-Down-Widerstände erfolgen. Im Fall eines Pull-Up-Widerstands liegt bei geöffnetem Schalter praktisch die Betriebsspannung und damit `HIGH`-Pegel an, ein geschlossener Schalter liefert `LOW`-Pegel. Mit einem Pull-Down-Widerstand sind die Verhältnisse umgekehrt: Bei geöffnetem Schalter zieht der Widerstand den Eingang auf `LOW`-Pegel, bei geschlossenem Schalter liefert die anliegende Betriebsspannung `HIGH`-Pegel:



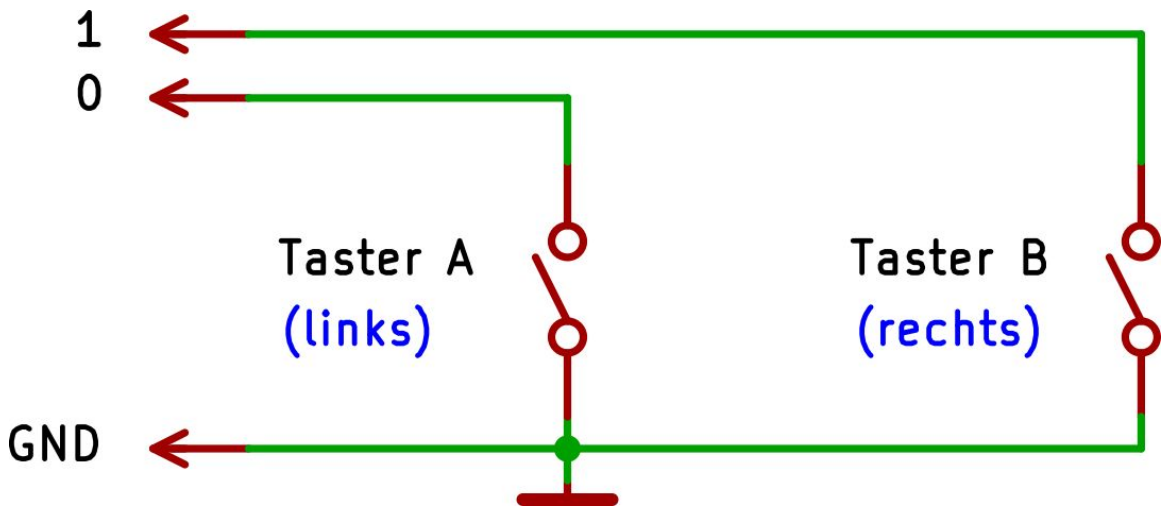
Taster mit Pull-Up-Widerstand (links) bzw. Pull-Down-Widerstand (rechts)

Bei dem in der Arduino-Entwicklungsumgebung vorgesehenen Beispiel, welches über **Datei > Beispiele > 0.2Digital > Button** aufgerufen werden kann, werden an Pin 2 ein Taster gegen 5V und ein 10kΩ-Pull-Down-Widerstand gegen Masse geschaltet. Das abgefragte Signal wird im Beispielprogramm über die LED an Pin 13 ausgegeben.

Wir wollen auf zusätzliche Widerstände verzichten. Bei den Arduino-Boards sind Pull-Up-Widerstände im Mikrocontroller vorgesehen und können bei Bedarf zugeschaltet werden. Dazu weist man in der Funktion `setup` den o.g. Pins den Eingangs-Modus mit Pull-Up-Widerständen zu [1]:

```
pinMode (pinTasterA,INPUT_PULLUP);
pinMode (pinTasterB,INPUT_PULLUP);
```

Die Taster sind dann von Pin 0 und 1 (oder bei Bedarf von den Pins 8 und 9) auf Masse zu legen.

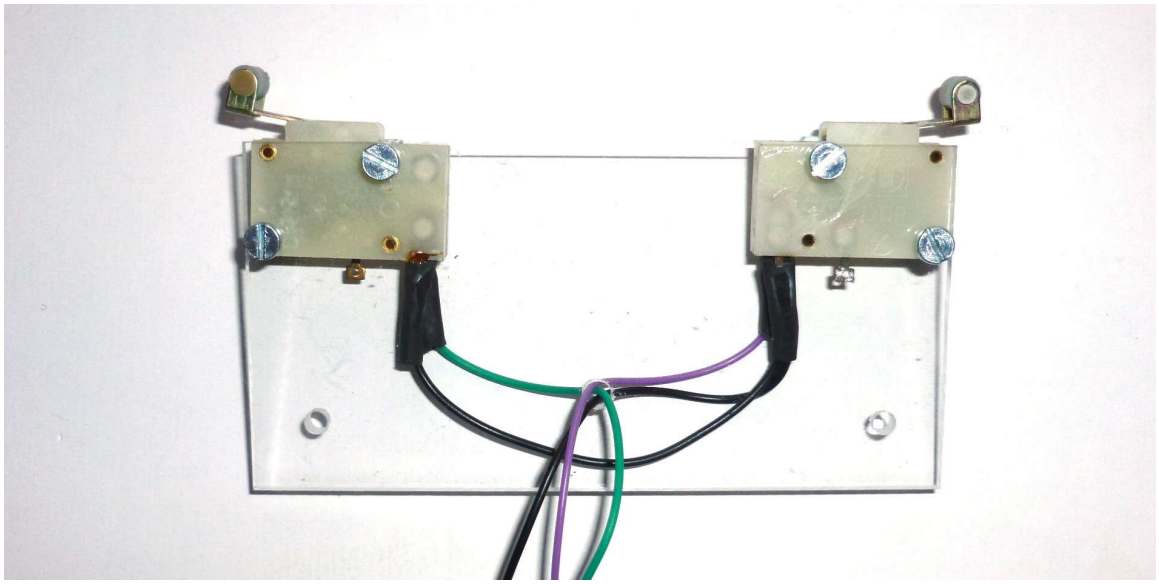


Anschluss von zwei Tastern an die über das Motor-Shield weitergeführten Anschlüsse des Arduino-Boards

Die Abfrage der digitalen Kanäle 0 und 1 erfolgt mit der Funktion `digitalRead`, welcher die Nummer des betreffenden Pins zu übergeben ist. Bei `LOW`-Pegel liefert diese Funktion den Zahlenwert 0, bei `HIGH`-Pegel den Wert 1. Diese Werte sind im Arduino-System als Konstante `LOW` und `HIGH` deklariert (siehe [1,2] bzw. Datei `Arduino.h`). Hat man das LCD-Modul in der üblichen Weise initialisiert, dann kann man sich das Ergebnis der Tasterabfrage mit folgender Hauptschleife im Programm anzeigen lassen:

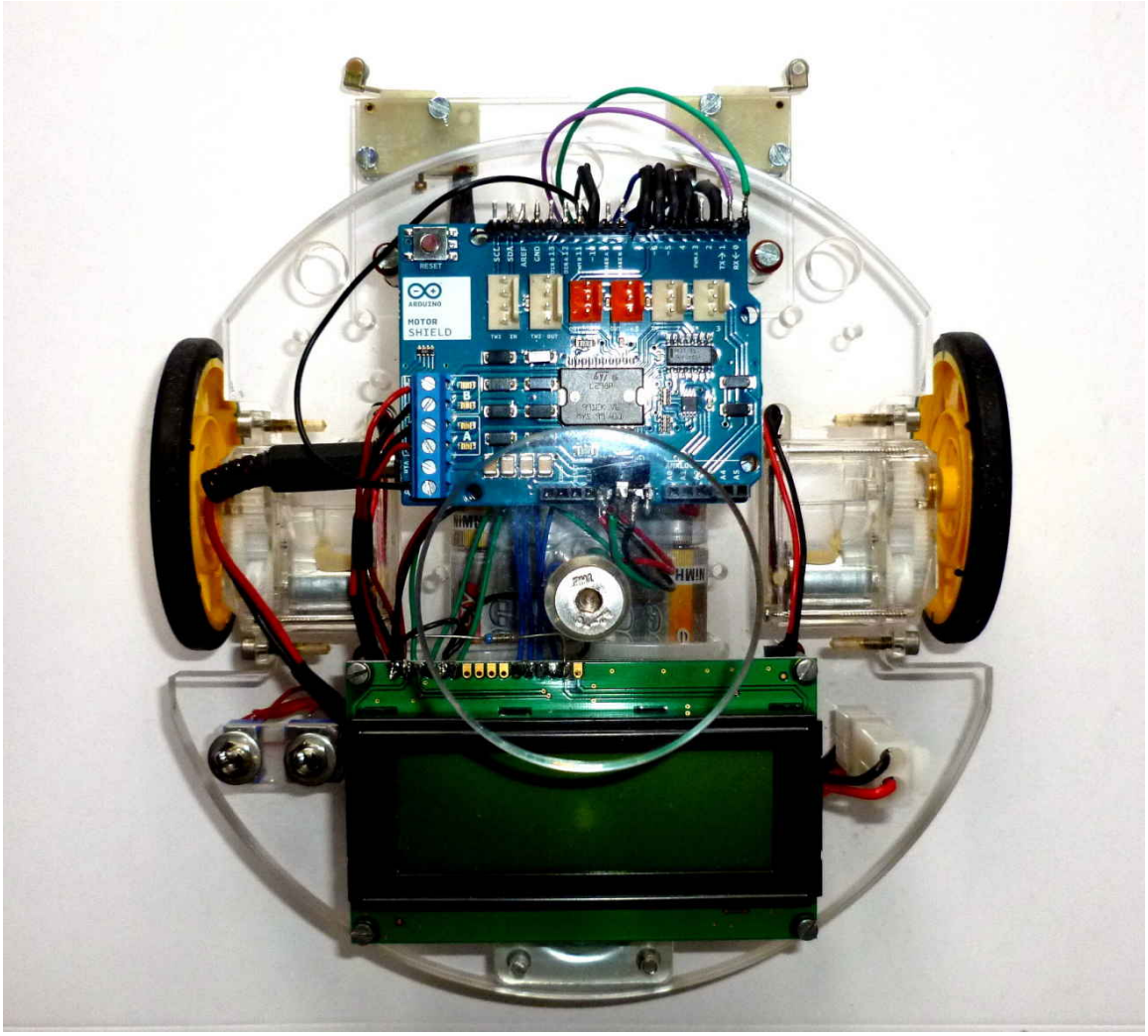
```
void loop()
{
    lcd.setCursor(0,0);
    lcd.print("Taster A: ");
    lcd.print(digitalRead(pinTasterA));
    lcd.setCursor(0,1);
    lcd.print("Taster B: ");
    lcd.print(digitalRead(pinTasterB));
}
```

Für den vorgesehenen Einsatz zur Hinderniserkennung sind Mikrotaster mit langem Hebel und Rollen am Hebelende empfehlenswert. Beim Prototypaufbau erfolgte die Montage auf einer kleinen Acryl-Platte:



Montage der zwei Mikrotaster auf einer Acryl-Platte

Die Acryl-Platte mit den Mikrotastern wird an der Vorderseite des Roboters montiert.



Mobiler Roboter mit Mikrotastern an der Vorderseite

Bei der Tasterabfrage mit Pull-Up-Widerstand entspricht der von `digitalRead` gelieferte Rückgabewert 0 einem gedrückten Taster, der Wert 1 einem geöffneten Taster. Bei zwei Tastern sind vier Zustände möglich (keiner gedrückt, entweder Taster A oder Taster B gedrückt, beide gedrückt). Diese Zustände könnte man mit verschachtelten `if`-Anweisungen bzw. `if-elseif`-Kombinationen abfragen. Aus Sicht des Verfassers ist eleganter, die beiden Abfrageergebnisse in einer neuen Integer-Variablen (hier: `Taster`) zusammenzufassen:

```
Taster  =   digitalRead(pinTasterA);
Taster += 2*digitalRead(pinTasterB);
```

Hierbei wird der Rückgabewert von Taster A mit dem um Faktor 2 skalierten Rückgabewert von Taster B addiert. Die vier Tasterzustände werden damit auf die Zahlen 0 bis 3 abgebildet. Die entsprechende Reaktion des Roboters auf den jeweiligen Zustand der zwei Taster kann dann sehr einfach über die `switch`-Anweisung implementiert werden. Dieser Ansatz wird in der nachfolgenden Hauptschleife genutzt, um die jeweilige Tasterbelegung auf dem LCD-Modul anzuzeigen:

```
void loop()
{
    lcd.home();
    lcd.print("Taster: ");
    Taster  =   digitalRead(pinTasterA);
    Taster += 2*digitalRead(pinTasterB);
    switch(Taster) {
        case 0: lcd.print("Beide "); break;
        case 1: lcd.print("Rechts"); break;
        case 2: lcd.print("Links "); break;
        case 3: lcd.print("Keiner"); break;
    }
}
```

5.2 Hindernisumfahrung mittels Tasterabfrage

In diesem Abschnitt soll ein Programm entstehen, welches durch Tasterabfrage Hindernisse erkennt und geeignete Ausweichmanöver durchführt. Dazu ist einerseits der Motor anzusteuern, andererseits sollen die jeweiligen Teilmanöver im LCD-Display angezeigt werden. Daher sind beide Bibliotheken – `LiquidCrystal` und `Motor` – einzubinden:

```
// Bibliotheken: LCD und Motor
#include <LiquidCrystal.h>
#include <Motor.h>
```

Außerdem ist von den entsprechenden Klassen jeweils eine Instanz zu erzeugen:

```
// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Motor motor(12, 13, 3, 11);
```

Im nächsten Schritt sind die Pins für die Taster festzulegen und eine Integer-Variable für das Abfrageergebnis anzulegen:

```
const byte pinTasterA = 0;
const byte pinTasterB = 1;
int Taster;
```

Die `setup`-Funktion initialisiert LCD-Modul und Motor-Ansteuerung. Zusätzlich werden die für die Tasterabfrage verwendeten Eingabekanäle mit internem Pull-Up-Widerstand konfiguriert.

```
void setup() {
    lcd.begin(20, 4);
    motor.begin();
    // Taster mit Pull-Up-Widerstand
    pinMode (pinTasterA,INPUT_PULLUP);
    pinMode (pinTasterB,INPUT_PULLUP);
}
```

Nach diesen Vorarbeiten können wir uns nun dem eigentlichen Problem widmen, nämlich einem Programm zur Hindernisumfahrung. Im Normalfall fährt der Roboter mit der durch die Größe `Speed` definierten Geschwindigkeit geradeaus. Bei der Berührung mit einem Hindernis soll der Roboter zunächst ein Stück zurück fahren. Registriert der linke Taster (Taster A) ein Hindernis, dann soll sich der Roboter ca. 90° nach rechts drehen, im Fall des rechten Tasters (Taster B) nach links. Sind beide Taster gleichzeitig gedrückt, d.h. trifft der Roboter frontal auf ein Hindernis, ist ein Wendemanöver (180°-Drehung) durchzuführen. Die für das Rückwärtsfahren, die seitliche Drehung bzw. das Wendemanöver vorgesehenen Zeiten sind experimentell zu bestimmen und unter `T_rueck`, `T_dreh` bzw. `T_wende` in ms zu hinterlegen.

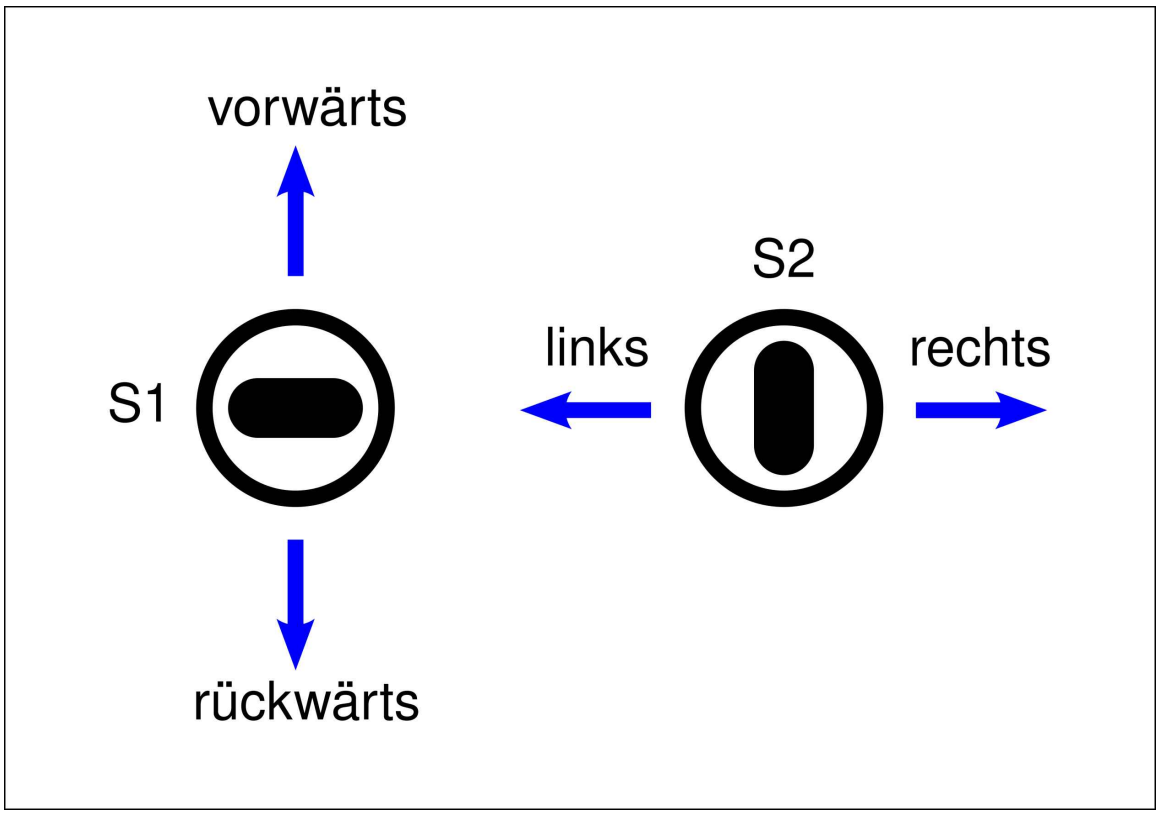
```
// Größen für Steuermanöver
int Speed = 180;
int T_rueck =  800;
int T_dreh  =  500; // 90°-Drehung
int T_wende = 1000; // 180°-Drehung
```

Die beschriebene Ausweichstrategie wurde in der Hauptschleife (`loop`) des Programms umgesetzt:

```
void loop() {
    lcd.home();
    Taster  =   digitalRead(pinTasterA);
    Taster += 2*digitalRead(pinTasterB);
    if (Taster==3) {
        // Vorwärtsfahrt
        lcd.print("Vorwaerts ");
        motor.write (Speed, Speed);
    }
    else {
        // Rückwärts fahren
        lcd.print("Rueckwaerts");
        motor.write (-Speed,-Speed);
        delay(T_rueck);
        lcd.home();
        // Ausweichmanöver
        switch(Taster) {
            case 0: lcd.print("Wenden ");
                motor.write (Speed,-Speed); delay(T_wende); break;
            case 1: lcd.print("Nach links ");
                motor.write (-Speed,Speed); delay(T_dreh);  break;
            case 2: lcd.print("Nach rechts");
                motor.write (Speed,-Speed); delay(T_dreh);  break;
        }
    }
}
```

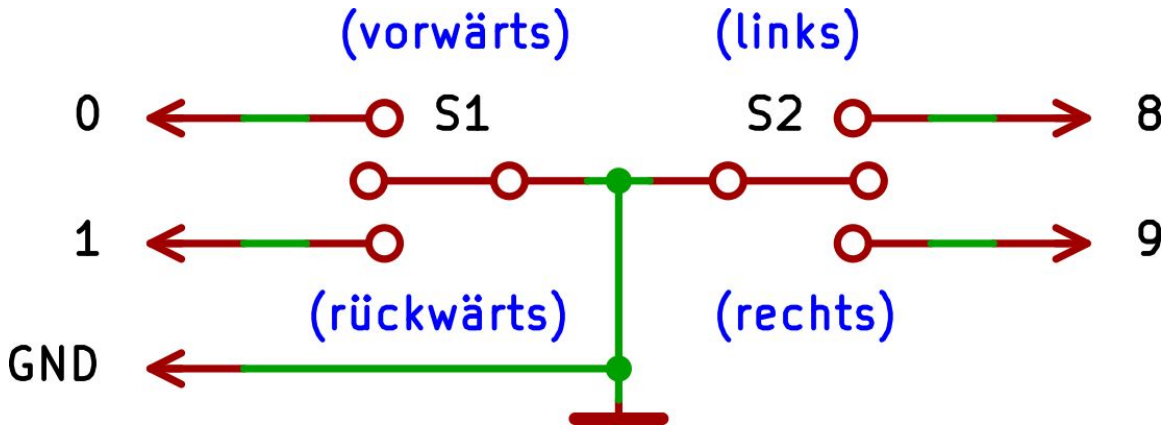
5.3 Drahtgebundene Steuerung mit zwei Umschaltern

Dieser Abschnitt beschreibt eine einfache drahtgebundene Fernsteuerung des mobilen Roboters. Dazu sind zwei Umschalter mit einer Mittelposition nötig, wobei für die Mittelposition kein nach außen geführter Anschluss erforderlich ist. Auf dem zur Fernsteuerung verwendeten Pult könnten die Schalter folgendermaßen angeordnet sein:



Anordnung der Umschalter für die Fernsteuerung

Zur Abfrage der Umschalter werden vier digitale Eingänge benötigt. Bei der nachfolgenden Schaltungsvariante werden dazu die digitalen Kanäle 0, 1, 8 und 9 verwendet:



Verdrahtung der Umschalter für die Fernsteuerung

An dieser Stelle sei daran erinnert, dass die digitalen Kanäle 8 und 9 nur dann für die Fernsteuerung eingesetzt werden können, wenn sie nicht bereits durch die Bremsfunktion des Arduino Motor Shields belegt sind. Beim Arduino Leonardo Board könnte man alternativ die Anschlüsse des ICSP-Adapters als zusätzliche digitale Eingänge nutzen. Die Fernsteuerung ist über eine fünfadrige flexible Leitung (Litze) mit dem Roboter zu verbinden (Masse und vier Signalleitungen).

Softwareseitig werden die verwendeten digitalen Kanäle in einem eindimensionalen Feld (Array) abgelegt:

```
const byte pinTaster[] = {0, 1, 8, 9};
```

Die Initialisierung dieser Kanäle als digitale Eingänge mit Pull-Up-Widerstand erfolgt in der `setup`-Funktion. Die Dimension des o.g. Feldes wird dabei über den Operator `sizeof` zurückgemeldet. Zusätzlich wird davon ausgegangen, dass die LCD-Anzeige in der bisher üblichen Weise eingebunden ist (Aufruf der Headerdatei, Anlegen der Instanz `lcd` der Klasse `LiquidCrystal`).

```
void setup() {
    lcd.begin(20, 4);
    // Taster mit Pull-Up-Widerstand
    for (byte i=0; i<sizeof(pinTaster); i++)
        pinMode (pinTaster[i],INPUT_PULLUP);
}
```

Durch die Konfiguration mit internen Pull-Up-Widerständen erhält man bei der Abfrage der digitalen Eingänge bei offenem Schalter die Zahl 1 und bei geschlossenem Schalter die Zahl 0. Den korrekten Anschluss der vier Schalter kann man bei einem vierzeiligen LCD-Modul mit einer einfachen Hauptschleife prüfen:

```
void loop() {
    lcd.clear();
    for (byte i=0; i<sizeof(pinTaster); i++) {
        lcd.setCursor(0,i);
        lcd.print("Pin ");
        lcd.print(pinTaster[i]);
        lcd.print(": ");
        lcd.print(digitalRead(pinTaster[i]));
```

```
    }
    delay(100);
}
```

Für die Weiterverarbeitung des Abfrageergebnisses fassen wir mit der Funktion `AbfrageTaster` die vier Bit der Schalter zu einem ganzzahligen Wert zusammen:

```
int AbfrageTaster () {
    int wert = 0;
    for (byte i=0; i<sizeof(pinTaster); i++)
        wert = (wert<<1) + digitalRead(pinTaster[i]);
    return wert;
}
```

Mit der folgenden Hauptschleife wird dieses Abfrageergebnis zyklisch ausgegeben:

```
void loop()
{
  lcd.clear();
  lcd.print("Abfrage: ");
  lcd.print(AbfrageTaster());
  delay(100);
}
```

Sind beide Umschalter in der Mittelstellung, so sind alle vier Bit gesetzt. In diesem Fall sollte sich später der Roboter im Stillstand befinden. Bei den vier Hauptbewegungsrichtungen ist jeweils genau ein Bit rückgesetzt. Die sinnvollen Kombinationen sind in der Tabelle abgelegt:

Wert	Bit 3	Bit 2	Bit 1	Bit 0	Manöver
15	1	1	1	1	stop
Hauptbewegungsrichtungen					
7	0	1	1	1	vorwärts
11	1	0	1	1	rückwärts
13	1	1	0	1	links
14	1	1	1	0	rechts
Kombinierte Bewegungsrichtungen					
5	0	1	0	1	links vorwärts
6	0	1	1	0	rechts vorwärts
9	1	0	0	1	links rückwärts
10	1	0	1	0	rechts rückwärts

Abfrageergebnis der Umschalter

Bisher erfolgte nur die Abfrage der Schalter, jetzt sollen auch die zwei Antriebsmotoren passend angesteuert werden. Dazu binden wir wieder die `Motor`-Bibliothek ein und erzeugen eine Instanz `motor` der zugehörigen `Motor`-Klasse. Bei einer auf Tastern basierenden Fernsteuerung ist natürlich keine Feindosierung der Geschwindigkeit möglich. Wir definieren den PWM-Wert für die Fahrgeschwindigkeit in der Konstanten `V`:

```
const int V=255;
```

In der Hauptschleife wird entsprechend des abgefragten 4-Bit-Wertes der Taster eine Fallunterscheidung durchgeführt. Das angegebene Programm berücksichtigt nur die Hauptbewegungsrichtungen. Kombinationen (z.B. links vorwärts), bei denen gleichzeitig zwei Kontakte geschlossen sind, führen momentan zum Stillstand. Allerdings lassen sich die vier kombinierten Bewegungsrichtungen leicht über zusätzliche `case`-Anweisungen ergänzen.

```
void loop() {
    int wert = AbfrageTaster();
    switch (wert) {
        // Hauptrichtungen
        case 7: // vorwärts
```

```
            motor.write(+V,+V); break;
        case 11: // rückwärts
            motor.write(-V,-V); break;
        case 13: // links
            motor.write(-V,+V); break;
        case 14: // rechts
            motor.write(+V,-V); break;
        default: // stop
            motor.write(0,0);
    }
}
```

Anstelle einer Fallunterscheidung könnte man die dem jeweiligen Abfragewert der Taster zugeordneten PWM-Werte für die Motoransteuerung auch in einem Feld ablegen. Bei 4 Bit gibt es insgesamt 2^4=16 mögliche Belegungen. Neben den in der o.g. Tabelle angegebenen 9 Werten (8 Bewegungsrichtungen sowie die Ruheposition, bei der beide Schalter in der Mittelstellung sind) verbleiben 7 Belegungen, die nicht sinnvoll sind und auch nicht auftreten sollten (z.B. gleichzeitig vorwärts und rückwärts oder links und rechts). Diese kennzeichnen wir mit "nicht definiert" und übergeben für beide Motoren den PWM-Wert Null, so dass der Roboter nicht bewegt wird. Für die PWM-Werte legen wir das zweidimensionale Feld `M` der Dimension 16×2 an. In jeder Zeile ist der erste Wert für den linken Motor und der zweite Wert für den rechten Motor vorgesehen. Die Zeilennummer entspricht dem Rückgabewert der Tasterabfrage:

```
const int M[][2] =
    {{ 0, 0}, //  0: nicht definiert
     { 0, 0}, //  1: nicht definiert
     { 0, 0}, //  2: nicht definiert
     { 0, 0}, //  3: nicht definiert
     { 0, 0}, //  4: nicht definiert
     { 0,+V}, //  5: links vorwärts
     {+V, 0}, //  6: rechts vorwärts
     {+V,+V}, //  7: vorwärts
     { 0, 0}, //  8: nicht definiert
     { 0,-V}, //  9: links rückwärts
     {-V, 0}, // 10: rechts rückwärts
     {-V,-V}, // 11: rückwärts
     { 0, 0}, // 12: nicht definiert
     {-V,+V}, // 13: links
```

```
      {+V,-V}, // 14: rechts
      { 0, 0}};// 15: stop
```

Mit diesem Feld kann man die Hauptschleife erheblich vereinfachen. Nach der Tasterabfrage werden die benötigten Werte zur Motoransteuerung unmittelbar dem Feld `M` entnommen.

```
void loop()
{
  int wert = AbfrageTaster();
  motor.write(M[wert][0],M[wert][1]);
}
```

Natürlich kann man das Programm auch noch um eine zusätzliche LCD-Ausgabe erweitern.

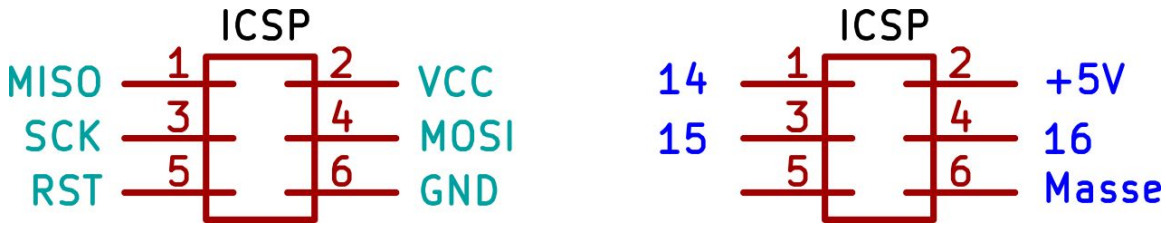
5.4 Zusätzliche digitale Ein- bzw. Ausgänge beim Arduino Leonardo Board

Das Arduino Leonardo Board verfügt zunächst über die standardmäßig vorgesehenen 14 digitalen Kanäle 0,…,13. Zusätzlich kann man die 6 analogen Kanäle A0,…,A5 bei Bedarf als digitale Kanäle 18,…,23 nutzen. Aufschluss über die hinsichtlich der Numerierung dazwischen liegenden Kanäle gibt die Definitionsdatei `pins_arduino.h`, die im Unterverzeichnis `hardware/arduino/avr/variants/leonardo` des Arduino-Programmordners zu finden ist [2]:

```
// Map SPI port to 'new' pins D14..D17
static const uint8_t SS   = 17;
static const uint8_t MOSI = 16;
static const uint8_t MISO = 14;
static const uint8_t SCK  = 15;
```

Dabei sind MISO, MOSI und SCK Anschlüsse der Schnittstelle *In-Circuit Serial Programming (ICSP)*, die auf (fast) allen Arduino Boards vorgesehen ist. Synonym wird auch die Bezeichnung *In-System Programming (ISP)* verwendet [3]. Die Besonderheit der In-System-Programmierung besteht darin, dass der zu programmierende Mikrocontroller im System bzw. auf der Leiter-

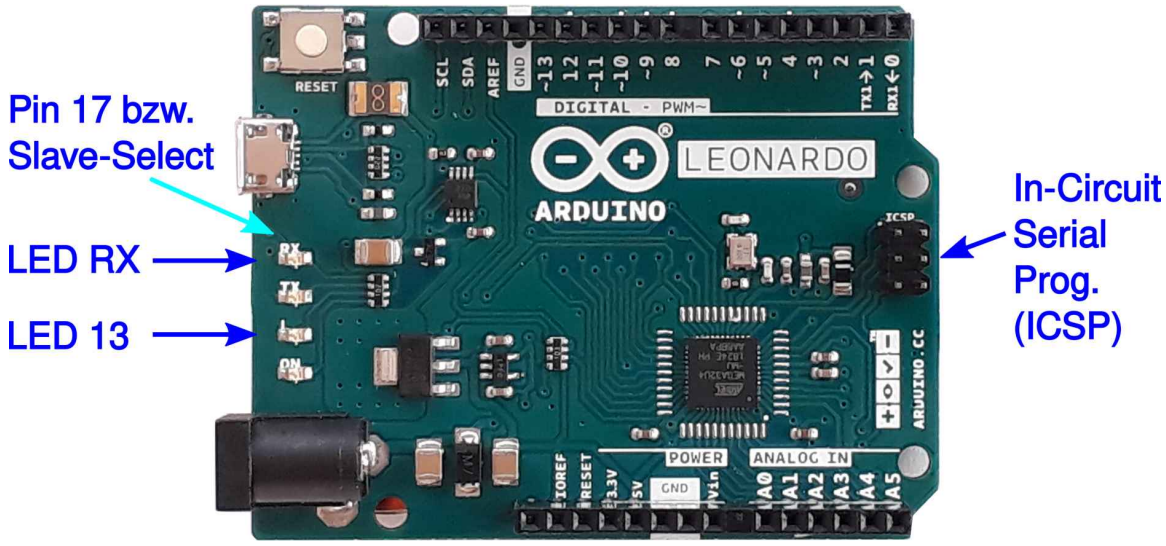
platte verbleiben kann. Die Datenübertragung erfolgt hier über einen seriellen Datenbus, der den Namen *Serial Peripheral Interface (SPI)* trägt [4,5].



Pinbelegung der ICSP-Schnittstelle (links), Bereitstellung zusätzlicher digitaler Kanäle beim Arduino Leonardo Board (rechts)

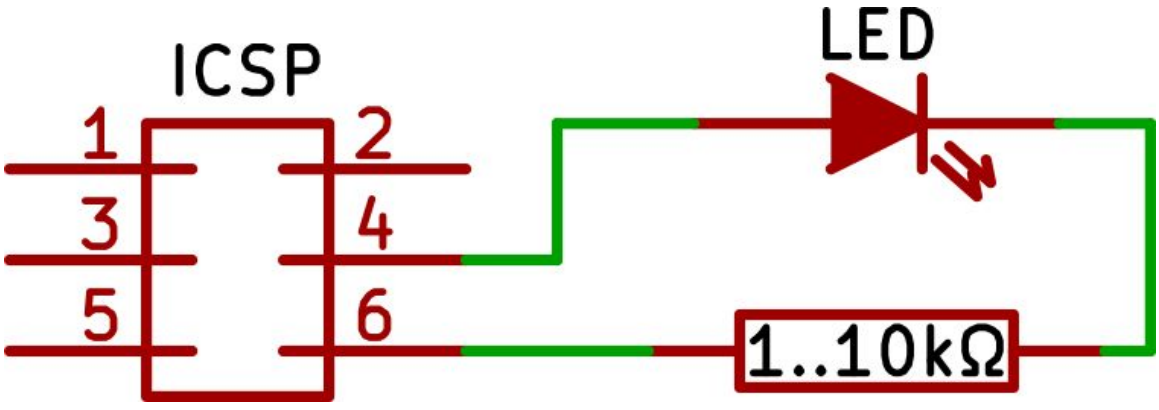
Über diese Schnittstelle könnte man die bei Arduino-Boards verwendeten Controller der ATmega-Serie programmieren [6]. Typischerweise wird man aber die Arduino-Programmierung über die auf den Boards vorgesehene USB-Schnittstelle in Verbindung mit dem Bootloader vornehmen. Beim Leonardo Board lassen sich die Anschlüsse der ICSP-Schnittstelle auch nutzen, um zusätzlich die digitalen Kanäle 14,…,16 bereitzustellen.

Der für den digitalen Kanal 17 verwendete Pin *Slave Select* ist nicht direkter Bestandteil des ICSP-Anschlusses, hängt aber mit dieser Schnittstelle zusammen. Der Anschluss wird gleichzeitig zur Ansteuerung der LED RX verwendet und ist nicht über einen Stecker herausgeführt. Allerdings gibt es für dieses Signal eine Durchkontaktierung auf der Leiterplatte, die sich nutzen lässt, um einen Draht bzw. einen (einpoligen) Stecker anzulöten.



Zusätzliche digitalen Anschlussmöglichkeiten beim Arduino Leonardo Board

Die zusätzlichen Kanäle kann man als digitale Ein- oder Ausgänge konfigurieren. Für einen einfachen Test bietet sich das in der Beispielsammlung mitgelieferte Programm `Blink` (**Datei > Beispiel > 0.1Basics > Blink**) an, bei dem man lediglich den verwendeten Pin ändert, beispielsweise auf Pin 16 für den MOSI-Anschluss. Nach Kompilieren und Hochladen des Programms sollte sich der Pegel des betreffenden Signals im Sekundentakt ändern. Das lässt sich mit einer LED in Verbindung mit einem Vorwiderstand überprüfen.



Testschaltung für den digitalen Kanal 16 beim Arduino Leonardo Board

Zur Überprüfung des digitalen Kanals 17 ist keine zusätzliche LED erforderlich, da dieser Kanal bereits auf dem Board mit der LED RX verbunden ist.

Das Arduino Mirco Board beruht auf der Architektur des Leonardo Boards und verwendet insbesondere den gleichen Prozessor ATmega32u4, welcher über eine integrierte USB-Schnittstelle verfügt. Dadurch hat man beim Arduino Micro Board in der gleichen Weise wie beim Arduino Leonardo Board vier weitere digitale Kanäle zur freien Verfügung. Beim Arduino Mirco Board ist zusätzlich sogar der Anschluss Slave Select herausgeführt, d.h. über einen Pin verfügbar.

Anstelle einer direkten Nutzung der ICSP-Signalleitungen wäre es auch denkbar, über SPI einen passenden Peripherieschaltkreis zur Porterweiterung (einen sog. *port expander*) anzusteuern, z.B. den MCP23S17 [7,8]. Mit diesem Schaltkreis könnte man 16 digitale Kanäle, die wahlweise als Ein- bzw. Ausgang konfigurierbar sind, betreiben.

Die beschriebene Erweiterung hinsichtlich der digitalen Kanäle lässt sich nicht auf das Board Arduino Uno übertragen, obwohl dieses Boards auch über eine ICSP-Schnittstelle verfügt. Bei diesem Board stimmen die Signalleitungen der Schnittstelle unmittelbar mit den digitalen Kanälen 11 bis 13 (und außerdem Kanal 10 mit Slave Select) überein, so dass man keine zusätzlichen Kanäle zur Verfügung hat [5]. Die Anschlüsse 11 bis 13 sind insbesondere schon durch das Arduino Motor Shield belegt. Allerdings könnte man über den I^2C-Bus eine Porterweiterung realisieren, z.B. mit dem Schaltkreis MCP23017 [7].

5.5 Quellenangabe

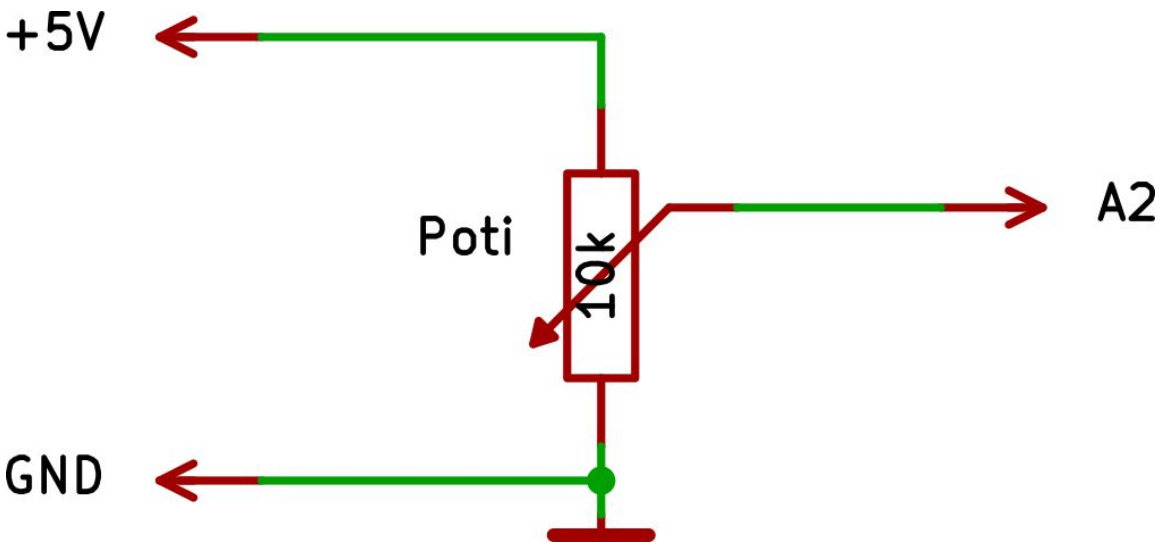
1. Arduino – DigitalPins. URL: https://www.arduino.cc/en/Tutorial/DigitalPins
2. Arduino – Home. URL: https://www.arduino.cc/
3. In-System-Programmierung. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=In-System-Programmierung
4. Serial Peripheral Interface. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=Serial_Peripheral_Interface
5. Arduino – SPI. URL: https://www.arduino.cc/en/Reference/SPI
6. Arduino – ArduinoISP. URL: https://www.arduino.cc/en/Main/ArduinoISP

7. Microchip: MCP23017/MCP23S17, 16-Bit I/O Expander with Serial Interface, 2007. Datenblatt.
8. Arduino Playground – MCP23S17 Class for Arduino. URL: https://playground.arduino.cc/Main/MCP23S17

6 Messung analoger Signale

6.1 Abfrage eines Potentiometers

Die betrachteten Arduino-Boards besitzen auf jeden Fall die analogen Eingangskanäle A0 bis A5. (Beim Leonardo-Board stehen weitere Analogkanäle zur Verfügung, für deren Betrieb man aber auf die entsprechenden digitalen Ein-Ausgabe-Kanäle verzichten muss.) In Verbindung mit dem Arduino Motor-Shield R3 ist zu erwarten, dass A0 und A1 für die Messung des Motorstroms vorgesehen und damit belegt sind. Für ein erstes Experiment wird daher der Anschluss A2 eingesetzt, an den ein Potentiometer angeschlossen wird:



Anschluss des Potentiometers

Zur Abfrage eines Analogkanals sind in der Arduino-Umgebung unter **Datei > Beispiele > 0.3Analog** einfache Beispiele zu finden. Bei dem Beispielprogramm `AnalogInOutSerial` werden die abgefragten Werte für die serielle Schnittstelle an ein Terminal gesendet, bei `AnalogInput` wird die Blinkfrequenz einer LED durch den abgefragten Spannungswert gesteuert. Wir wollen die gemessenen Werte auf dem LCD-Modul

anzeigen. Im Programm sind zunächst die Voraussetzungen für den Betrieb des LCD-Moduls zu schaffen:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

Der verwendete Analogkanal A2 wird der Konstanten `Poti` zugewiesen. Für den abzufragenden Wert des Analog-Digital-Umsetzers legt man die Integer-Variable `Wert` an.

```
const byte Poti = A2;
int Wert;
```

Der Analog-Digital-Umsetzer (engl. *analog-to-digital converter*, kurz *ADC*) bildet den Spannungsbereich von 0V bis 5V auf die (ganzzahligen) Werte von 0 bis 1023 ab. Mit dem nachfolgenden Umrechnungsfaktor kann man für den ausgelesenen Analogwert die zugehörige Spannung bestimmen:

```
const float Faktor=5.0/1023;
```

Für die Abfrage der analogen Kanäle muss in der `setup`-Funktion keine Initialisierung vorgenommen werden, wohl aber für die Anzeige auf dem LCD-Modul:

```
void setup() {
  lcd.begin(20, 4);
  lcd.print("Messung am Poti:");
}
```

Zur eigentlichen Abfrage der Analog-Digital-Umsetzer steht in der Arduino-Umgebung die Funktion `analogRead` zur Verfügung, welcher als Argument der jeweils verwendete Analogkanal zu übergeben ist:

```
Wert = analogRead(Poti);
```

Diese Anfrage wird in der Hauptschleife zyklisch durchgeführt. In der zweiten Zeile des LCD-Moduls wird der abgefragte Wert des ADC angezeigt, in der dritten Zeile die daraus in Verbindung mit dem o.g. Umrechnungsfaktor anliegende Spannung berechnet:

```
void loop() {
    Wert = analogRead(Poti);
    // Wert ADC
    lcd.setCursor(0, 1);
    lcd.print("ADC: ");
    lcd.print(Wert);
    // mind. 3 Leerzeichen
    lcd.print("   ");
    // Spannung in V
    lcd.setCursor(0, 2);
    lcd.print("Spg: ");
    lcd.print(Faktor*Wert);
    lcd.print(" V ");
}
```

Der ADC-Wert wird linksbündig ausgegeben und hat dementsprechend als Zeichenkette in Abhängigkeit vom anzuzeigenden Wert eine unterschiedliche Länge. Wird nach der Anzeige einer vierstelligen Zahl beispielsweise ein nur dreistelliger Wert ausgegeben, so würde die letzte Ziffer der vierstelligen Zahl stehen bleiben. Um diese "übrig gebliebenen" Ziffern zu löschen werden am Ende noch drei Leerzeichen ausgegeben.

Insgesamt ist die Ausgabe mit etlichen `print`-Befehlen vergleichsweise umständlich. Bindet man zusätzlich die Bibliothek `Streaming` ein, dann vereinfacht sich die Hauptschleife merklich:

```
void loop() {
    Wert = analogRead(Poti);
    lcd.setCursor(0, 1);
    lcd<<"ADC: "<<Wert<<"   ";
    lcd.setCursor(0, 2);
    lcd<<"Spg: "<<Faktor*Wert<<" V  ";
}
```

Der Umrechnungsfaktor `Faktor` ist eine Gleitkommazahl vom Typ `float`. Die eingesetzten Mikrocontroller der ATmega-Serie verfügen über keine hardwareseitige Gleitkommaeinheit, die entsprechenden Operationen werden (mühselig) softwareseitig nachgebildet. Das kostet Speicherplatz und Rechenzeit. Eine denkbare Alternative bestünde darin, die Spannung nicht als gebrochene Zahl in Volt (V), sondern als ganzzahligen Wert in Millivolt

(mV) auszugeben. Neben der schon vorhandenen Integer-Variablen `Wert` legt man eine zusätzliche Integer-Variable `Spg` für die Spannung in mV an:

```
int Wert, Spg;
```

Außerdem muss man den Bereich der ADC-Werte von 0 bis 1023 für die Ausgabe in mV auf 0 bis 5000 skalieren. Dazu steht in der Arduino-Umgebung die Funktion `map` zur Verfügung:

```
Spg = map (Wert, 0, 1023, 0, 5000);
```

Für die Spannungsangabe in Millivolt könnte man folgende Hauptschleife nutzen:

```
void loop() {
    Wert = analogRead(Poti);
    lcd.setCursor(0, 1);
    lcd<<"ADC: "<<Wert<<"    ";
    lcd.setCursor(0, 2);
    Spg = map (Wert, 0, 1023, 0, 5000);
    lcd<<"Spg: "<<Spg<<" mV    ";
}
```

Die ganzzahlig in mV abgelegte Spannung kann man mit überschaubarem Aufwand auch als Dezimalzahl in V ausgeben. Dazu bindet man die Standard-Bibiothek `stdio.h` ein und legt zwei Zeichenkettenvariable an:

```
char Zeile1[21];
char Zeile2[21];
```

Die Hauptschleife könnte dann so aussehen:

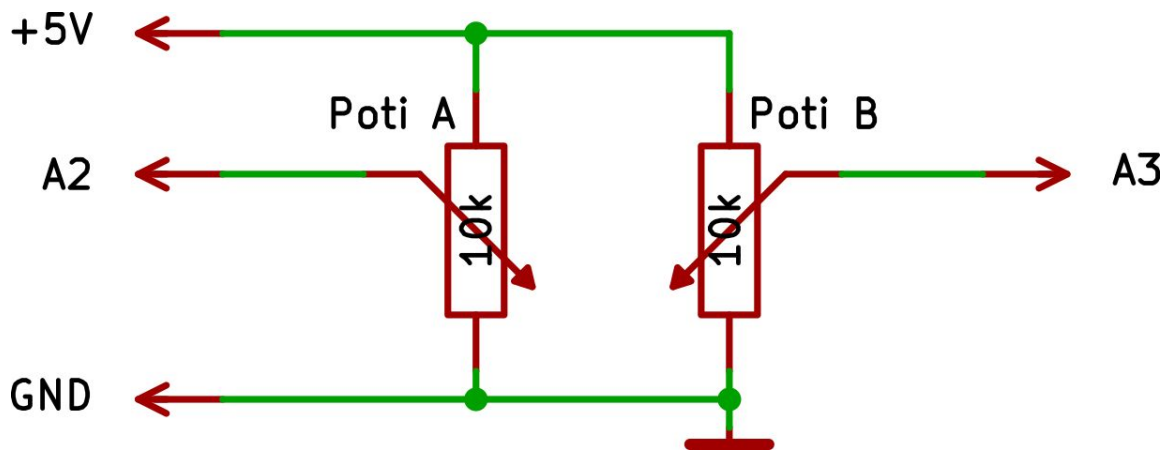
```
void loop() {
    Wert = analogRead(Poti);
    lcd.setCursor(0, 1);
    sprintf(Zeile1, "ADC: %5d", Wert);
    lcd.print(Zeile1);
    lcd.setCursor(0, 2);
    Spg = map (Wert, 0, 1023, 0, 5000);
    sprintf(Zeile2, "Spg: %1d,%03d", Spg/1000, Spg%1000);
```

```
    lcd.print(Zeile2);
}
```

Mit "`%5d`" bzw. "`%1d`" wird eine Integer-Variable mit 5 Stellen bzw. mit 1 Stelle ausgegeben. Der Formatstring "`%03d`" generiert eine 3-stellige Ausgabe, bei der die führenden Stellen mit Nullen aufgefüllt werden. Der Spannungsanteil in V wird mit der ganzzahligen Division `Spg/1000` ermittelt, der über den Divisionsrest `Spg%1000` ermittelte Anteil gibt die in mV angegebenen Nachkommastellen an.

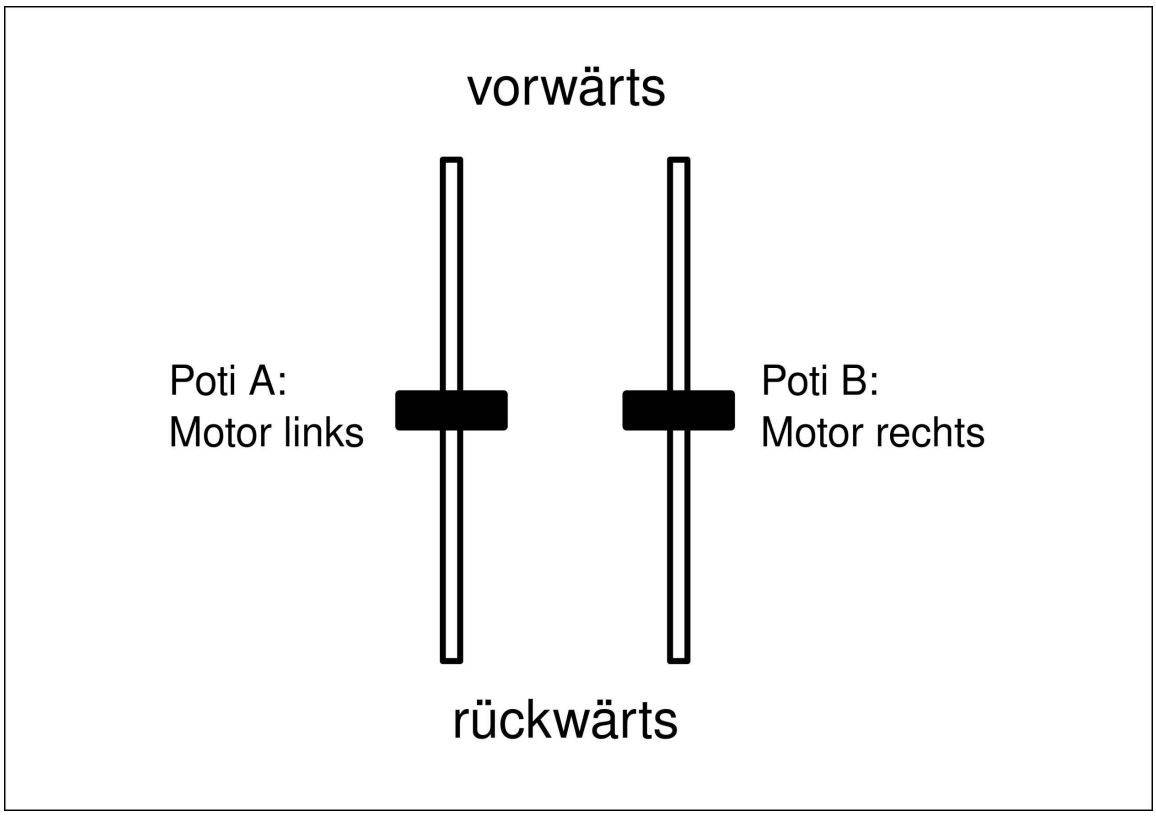
6.2 Drahtgebundene Steuerung über zwei Schiebepotentiometer

Mit der Abfrage von zwei Potentiometern kann man auf einfache Weise eine drahtgebundene Fernsteuerung aufbauen. Über eine vieradrige flexible Leitung werden die zwei Potis mit dem Roboter verbunden. Die Mittelanschlüsse sind mit zwei analogen Eingängen des Arduino-Boards zu verbinden. In der Schaltung werden hier die Anschlüsse A2 und A3 genutzt:



Anschluss der Potentiometer

Jedes der zwei Potentiometer soll die Drehrichtung bzw. Winkelgeschwindigkeit für einen Motor einstellen. Dabei ist das Poti A für den linken Motor und das Poti B für den rechten Motor vorgesehen:



Fernsteuerung mit zwei Potentiometern

Softwareseitig werden zuerst die benötigten Bibliotheken eingebunden:

```
// Bibliotheken: LCD und Motor
#include <LiquidCrystal.h>
#include <Motor.h>
#include <stdio.h>
```

Von den C++ Klassen zur LCD- und Motoransteuerung wird jeweils eine Instanz erzeugt:

```
// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Motor motor(12, 13, 3, 11);
```

Die für die zwei Potis verwendeten analogen Eingänge A2 und A3 werden als Konstante definiert:

```
// Pins für Potis
const byte PotiA = A2; // links
const byte PotiB = A3; // rechts
```

Für die im Zusammenhang mit den Potentiometern abgefragten Analogwerte und die daraus zu berechnenden Ausgabewerte für die Motoren legen wir Integer-Variable an:

```
// ADC-Signale
int WertA, WertB;
// Motorsignale
int KanalA, KanalB;
```

In der `setup`-Funktion werden das LCD-Modul und die `Motor`-Klasse initialisiert:

```
void setup() {
  lcd.begin(20, 4);
  lcd.print("Fernsteuerung: Poti");
  motor.begin();
}
```

Für beide Kanäle sollen die abgefragten Analogwerte der Potis als auch die Ausgabewerte für den Motor angezeigt werden. Da die Ausgabe dieser Werte für beide Kanäle nahezu gleich ist, wird für die Ausgabe eine Funktion angelegt. Die Auswahl der Kanäle erfolgt über die Variable `nr`, die für den Kanal A den Wert 0 und für den Kanal B den Wert 1 anzunehmen hat.

```
// LCD-Ausgabe
void ausgabe (short nr, int wert, int kanal)
{
  char Zeile[21];
  lcd.setCursor (0, 1+nr);
  sprintf(Zeile, "Kanal %c: %4u / %+3d", 'A'+nr, wert, kanal);
  lcd.print(Zeile);
}
```

In der Hauptschleife werden zunächst die Potentiometerwerte (0…1023) abgefragt und auf den Wertebereich von -200…200 für die Ansteuerung der Motoren skaliert. Zu Überwachungszwecken erfolgt die Ausgabe dieser Werte auf dem LCD-Modul. Anschließend werden die Motoren angesteuert.

Mit einer kleinen Zeitverzögerung vermeidet man zu hektische Reaktionen des Roboters:

```
void loop() {
    // Einlesen ADC
    WertA = analogRead(PotiA);
    WertB = analogRead(PotiB);
    // Skalieren auf Motorsignale
    KanalA = map (WertA, 0, 1023, -200, 200);
    KanalB = map (WertB, 0, 1023, -200, 200);
    // LCD-Ausgabe
    ausgabe(0, WertA, KanalA);
    ausgabe(1, WertB, KanalB);
    // Motoransteuerung
    motor.write (KanalA, KanalB); delay(100);
}
```

6.3 Strommessung am Arduino-Motor-Shield

Das offizielle Arduino Motor-Shield R3 verfügt über die Möglichkeit der Motorstrommessung. Für Motor A steht der analoge Eingang A0 zur Verfügung, für Motor B der Eingang A1. Die Motorstrommessung wird für Motor A erläutert, das gleiche Vorgehen ist auch bei Motor B zielführend. Zur Verifikation des Messergebnisses sollte man für die Versuche ein als Amperemeter betriebenes Multimeter in Reihe zu Motor A schalten.

Der Motorstrom soll auf dem LCD-Modul angezeigt werden. Dafür bindet man die entsprechenden Bibliotheken ein und legt die Instanz `lcd` der Klasse `LiquidCrystal` an:

```
#include <LiquidCrystal.h>
#include <Streaming.h>
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

Die Motoransteuerung soll hier mit den gängigen Funktionen der Arduino-Umgebung erfolgen, d.h. ohne die Bibliothek `Motor`. Sowohl für die Ansteuerung als auch für die Strommessung werden daher zunächst die Adressen festgelegt:

```
const byte DIRA = 12;
const byte PWMA =  3;
const byte StromA = A0;
```

In der Funktion `setup` initialisiert man in der üblichen Weise das LCD-Modul, deklariert die zur Motoransteuerung verwendeten digitalen Kanäle als Ausgänge und versetzt Motor A in eine geeignete Bewegung (hier: Vorwärtsbewegung mit PWM-Wert 200):

```
void setup() {
  lcd.begin(20, 4);
  lcd.print("Messung Motorstrom:");
  // Signale für Motoransteuerung
  pinMode (DIRA, OUTPUT);
  pinMode (PWMA, OUTPUT);
  // Motor A einschalten
  digitalWrite (DIRA, LOW);
  analogWrite (PWMA, 200);
}
```

Der vom ADC abgefragte Wert ist eine Integerzahl, den Strom könne man als Gleitkommazahl auffassen. Aus Effizienzgründen soll allerdings auch der Strom durch einer Integerzahl repräsentiert werden. Dazu deklariert man zwei Variable:

```
int wert;
int strom;
```

Die Abfrage des zur Strommessung vorgesehenen analogen Pins erfolgt in der gleichen Weise wie bei dem am Anfang des Kapitels betrachteten Potentiometer:

```
wert = analogRead(StromA);
```

Beim Arduino-Motor-Shield wird der zulässige Maximalstrom von 2A auf die Spannung von 3,3V abgebildet [1]. Das entspricht 1,65V/A. Der ADC bildet den Spannungsbereich von 0V bis 5V auf die Werte 0 bis 1023 ab, womit sich für diesen Gesamtbereich ein Strom von 0mA bis 3030mA=3,03A ergeben würde. Die Umrechnung des Bereichs von 0 bis

1023 auf den in mA angegebenen Strom von 0 bis 3030 kann man wieder mit der Funktion `map` erledigen:

```
strom = map(wert, 0, 1023, 0, 3030);
```

Die Hauptschleife zur Strommessung und -anzeige ist nachfolgend dargestellt. Zusätzlich wird jeder Schleifendurchlauf um 500ms verzögert, damit die Anzeige nicht nur flimmert.

```
void loop() {
    wert = analogRead(StromA);
    lcd.setCursor(0, 1);
    lcd<<"ADC:   "<<wert<<"   ";
    // Strom in mA
    strom = map(wert, 0, 1023, 0, 3030);
    lcd.setCursor(0, 2);
    lcd<<"Strom: "<<strom<<" mA   ";
    delay(500);
}
```

Im Versuch schwankten die gemessenen Werte vom einem Abtastschritt zum nächsten sehr stark. Hier muss man sich verdeutlichen, dass bei einem Gleichstrommotor der Strom durch den Kommutator zyklisch unterbrochen wird. Bei diesen Schaltvorgängen können zudem durch die Wicklungs-induktivitäten Spannungsspitzen verursacht werden. Abhilfe kann hier ein Tiefpassfilters schaffen, mit dem die Messwerte geglättet werden. Vor der erneuten Zuweisung der Variablen `Wert` wird auf der rechten Seite das arithmetische Mittel zwischen dem alten Wert und dem aktuellen Messwert gebildet:

```
wert = (wert + analogRead(StromA)) / 2;
```

Mit dieser Glättung verhält sich der angezeigte Wert merklich ruhiger. Hält man das Rad kurzzeitig fest, sollte ein signifikant größerer Strom fließen.

6.4 Hindernisumfahrung mittels Motorstrom-messung

Zur Hinderniserkennung setzt man meist Taster oder Entfernungssensoren ein. Ein Hindernis lässt sich aber auch über den Motorstrom erkennen. Wird der Roboter angehalten oder ausgebremst, so steigt der Motorstrom merklich an. Dieser Effekt wird in diesem Abschnitt ausgenutzt, um eine Hindernisumfahrung ohne zusätzlichen Sensor zu ermöglichen. Die Messwerterfassung erfolgt dabei ausschließlich über das Arduino-Motor-Shield, welches zwei Kanäle zur Motorstrommessung bereitstellt.

Zunächst werden wieder die Bibliotheken für die LCD-Anzeige und den Motor eingebunden:

```
// Bibliotheken: LCD und Motor
#include <LiquidCrystal.h>
#include <Motor.h>
```

Die dadurch bereitgestellten C++ Klassen werden entsprechend der Verschaltung von LCD-Anzeige und Motor initialisiert:

```
// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Motor motor(12, 13, 3, 11);
```

Beim Ardunio-Motor-Shield sind die analogen Eingänge A0 und A1 für die Abfrage des Motorstroms vorgesehen:

```
// Pins für Motorstrommessung
const byte StromA = A0;
const byte StromB = A1;
```

Bei der Geradeausfahrt ohne Behinderung traten (nach Filterung) für den Motorstrom ADC-Werte im Bereich 20…45 auf, beim Blockieren der Räder dagegen Werte von 100 und mehr. Der Wert 70 für die Konstante `Imax` liegt dazwischen und dient zur Unterscheidung bzw. Erkennung dieser zwei Situationen.

```
// maximaler Wert für Motorstrom
const byte Imax = 70;
```

Die ADC-Werte für die Motorströme werden in den Variablen `iA` bzw. `iB` abgelegt. Die Einführung dieser Hilfsvariablen ist nötig wegen der

Filterung.

```
// Motorströme
int iA = 0;
int iB = 0;
```

Für das Ausweichmanöver werden weitere Konstante benötigt. Die Größe `speed` gibt den PWM-Wert für die Motoransteuerung an. Hier sind Werte bis 255 möglich. Der Wert `T_rueck` gibt die Zeit für das Rückwärtsfahren (nach dem Erkennen eines Hindernisses) in ms an. Die für eine seitliche Drehung von ca. 90° benötigte Zeit (wieder in ms) ist in `T_dreh` hinterlegt:

```
// Größen für Steuermanöver
int Speed   = 180; // Wert für Geschwindigkeit
int T_rueck = 800; // Zeit rückwärts
int T_dreh  = 500; // Zeit für Drehung
```

Die Initialisierung der Klassen zur LCD- und Motoransteuerung erfolgt wie üblich in der Funktion `setup`:

```
void setup() {
  lcd.begin(20, 4);
  motor.begin();
}
```

Als nächstes folgt die Hauptschleife:

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

In der Hauptschleife werden beide analogen Kanäle abgefragt und zur Glättung einer Mittelung unterworfen:

```
  iA = (iA + analogRead(StromA)) / 2;
  iB = (iB + analogRead(StromB)) / 2;
```

Die von den ADCs erfassten Werte sollen in der ersten Zeile des LCD-Moduls zur Anzeige kommen:

```
lcd.home();
lcd.print("Ia: ");
lcd.print(iA);
lcd.print(" Ib: ");
lcd.print(iB);
lcd.print("   ");
```

Falls beide (gefilterten) Messwerte für die Motorströme die vorgegebene Schranke `Imax` nicht überschreiten, so wird der Roboter in seiner Fahrt nicht beeinträchtigt und kann sich weiterhin geradeaus vorwärts bewegen.

```
if (max(iA, iB)<=Imax) {
  // Vorwärtsfahrt
  lcd.setCursor(0,1);
  lcd.print("Vorwaerts    ");
  motor.write (Speed, Speed);
  delay(100);
}
```

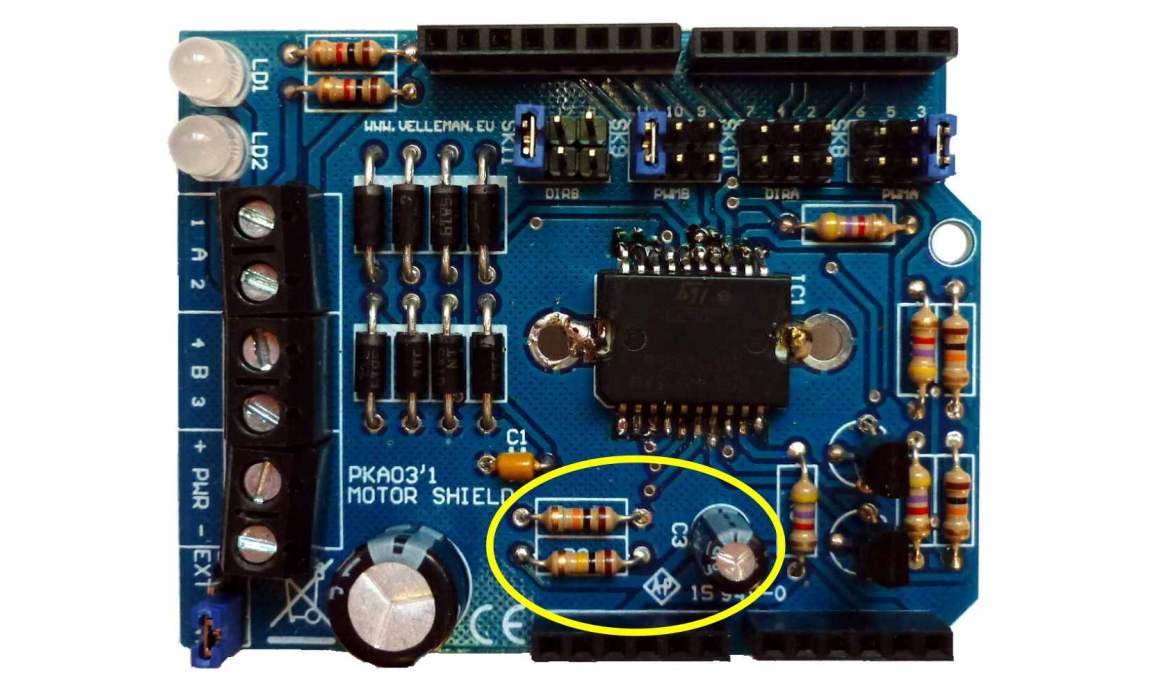
Andernfalls, wenn also mindestens ein Wert für den Motorstrom oberhalb der Schranke `Imax` liegt, wird der Roboter in seiner Bewegung beeinträchtigt. Dieser Fall wird als Hindernis eingestuft. Der Roboter bewegt sich zunächst eine Zeit rückwärts und dreht dich dann zur Seite. Für den nächsten Durchlauf wird der Motor kurz abgeschaltet, damit die durchgeführte Richtungsänderung sich nicht auf die nächste Strommessung auswirkt.

```
else {
  // Rückwärts fahren
  lcd.setCursor(0,1);
  lcd.print("Rueckwaerts");
  motor.write (-Speed,-Speed);
  delay(T_rueck);
  lcd.setCursor(0,1);
  lcd.print("Nach rechts");
  motor.write (Speed,-Speed);
  delay(T_dreh);
  motor.write (0,0);
  delay(100);
  iA=0;
```

```
    iB=0;
  }
```

6.5 Motorspannungsmessung am Velleman-Motor-Shield

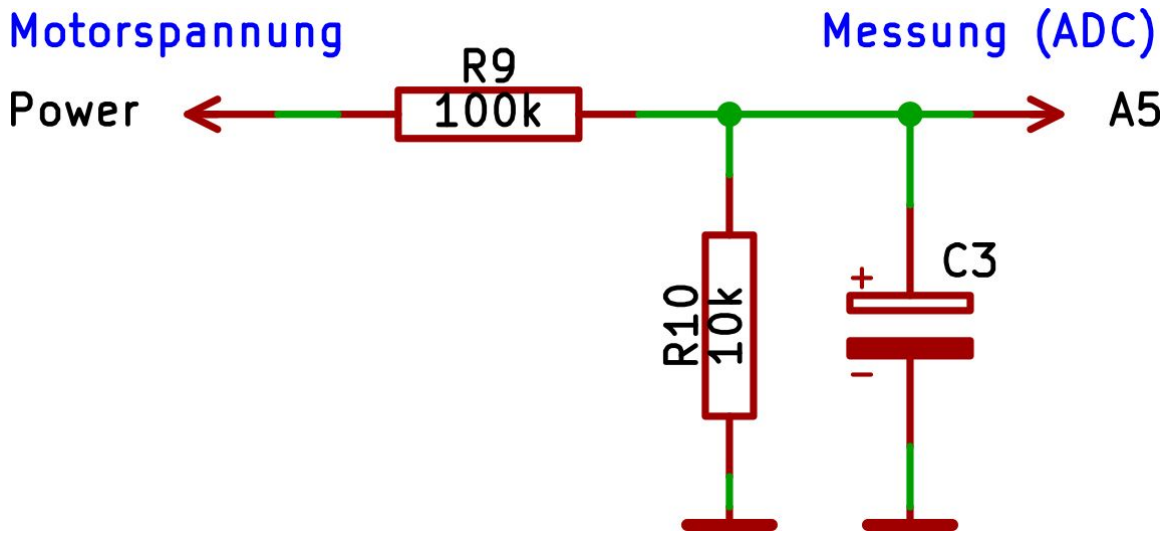
Beim Motor-Shield von Velleman wurde die Möglichkeit zur Messung der Motorspannung vorgesehen. Je nach Stellung des Jumpers SK7 wird der Motor entweder über die Versorgungsspannung V_{in} des Arduino-Boards oder über eine externe Spannung V_{ext} versorgt. Die motorseitige Versorgungsspannung sei im Folgenden mit V_{Power} bezeichnet. Für die Erzeugung des Messsignals sind auf dem Motor-Shield die Widerstände R9 und R10 sowie der Elektrolytkondensator C3 zuständig [2,3]:



Velleman Motor-Shield mit hervorgehobenen Bauelementen R9, R10 und C3

Die Versorgungsspannung V_{Power} wird über den aus R9 und R10 bestehenden Spannungsteiler an den analogen Eingang A5 geführt, dessen Span-

nung wir mit V_{ADC} bezeichnen. Zur Glättung ist zusätzlich der Elektrolytkondensator C3 vorgesehen:



Schaltung zur Messung der Motor- bzw. Versorgungsspannung beim Velleman Motor-Shield

An dem Spannungsteiler entsteht folgender Spannungsabfall:

V_{ADC} / V_{Power} = R10 / (R9+R10) = 10kΩ / 110kΩ = 1/11.

Die am ADC anliegende Spannung V_{ADC} ist somit um Faktor 11 kleiner als die Versorgungsspannung V_{Power}. Anders formuliert: Multipliziert man die am ADC ermittelte Spannung mit dem Faktor 11, dann erhält man die Versorgungsspannung. Die Messung der motorseitigen Versorgungsspannung würde somit im Wesentlichen wie die am Anfang des Kapitels behandelte Spannungsmessung am Potentiometer erfolgen. Anstelle des beim Potentiometer verwendeten Umrechnungsfaktors

```
const float Faktor=5.0/1023;
```

würde man jetzt den um Faktor 11 vergrößerten Umrechnungsfaktor

```
const float Faktor=55.0/1023;
```

verwenden.

Hinweis:

Beim Arduino Uno Board wird der Anschluss A5 (zusammen mit A4) auch für den I²C-Bus verwendet. Für die Nutzung des I²C-Busses dürfte man dabei die entsprechenden Bauelemente (R9, R10, C3) nicht bestücken bzw. müsste sie wieder entfernen.

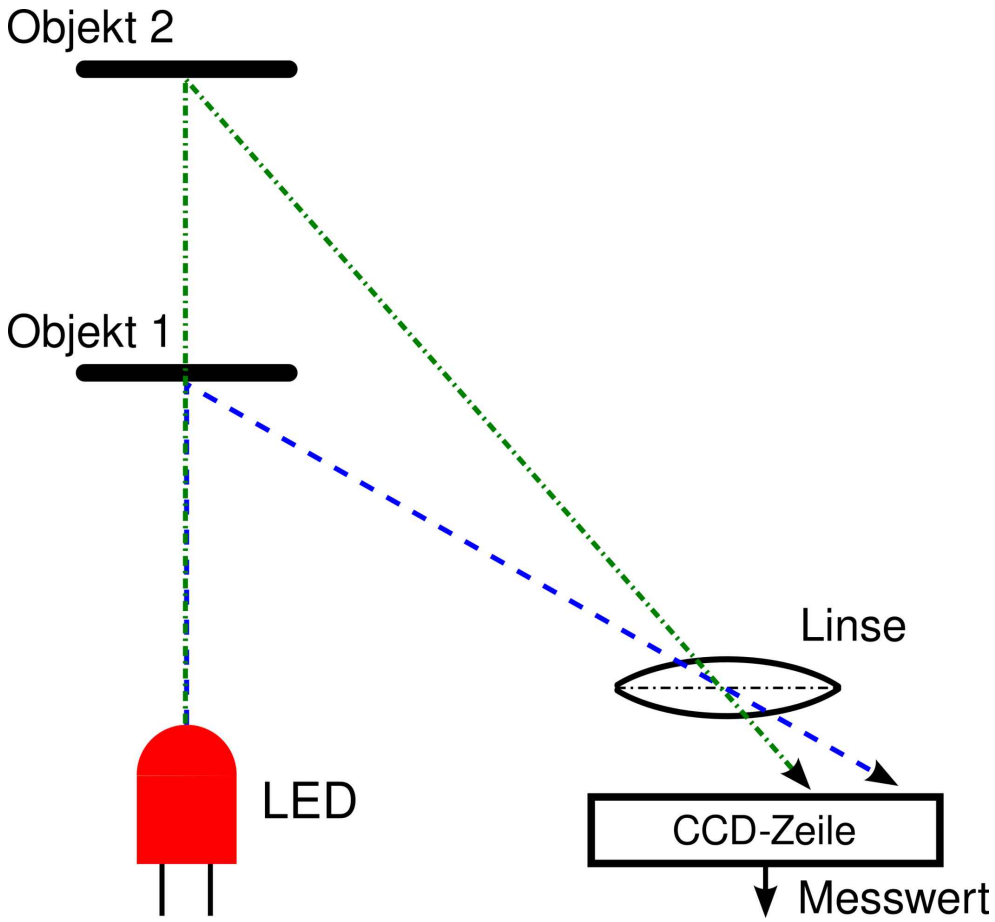
6.6 Quellenangabe

1. Arduino Motor Shield. URL: http://arduino.cc/en/Main/ArduinoMotorShieldR3.
2. KA03 Motor & Power shield Arduino®. Illustrated assembly manual, Velleman.
3. VMA03 Motor & Power shield Arduino®. Manual, Velleman.

7 Abstands- bzw. Entfernungsmessung

7.1 Entfernungssensoren auf Basis optischer Triangulation

Mit einem Entfernungsmesser bzw. Abstandssensor ist der Roboter in der Lage, Hindernisse vor einem Kontakt bzw. vor einer Kollision zu erkennen und zu umgehen. Zur Entfernungsmessung gibt es zahlreiche Messprinzipien bzw. Sensorausführungen. Eine Möglichkeit zur Abstandsbestimmung ist die *optische Triangulation*. Dabei sendet man einen Lichtstrahl aus, welcher vom zu vermessenden Objekt reflektiert wird. Über ein lichtempfindliches Element wird diese Reflektion wahrgenommen. Typischerweise wird der reflektierte Lichtstrahl über eine Linse oder ein Prisma auf einen CCD-Zeilensensor geleitet. In Abhängigkeit von der Entfernung ändert sich der Winkel des reflektierten Lichtstrahls und damit die Position auf der CCD-Zeile. Der CCD-Sensor ist ein lichtempfindliches integriertes Bauteil, wobei die Abkürzung CCD für *charge coupled device* steht. In Digitalkameras setzt man CCD-Sensoren mit einer matrixartigen (zweidimensionalen) Anordnung der einzelnen Fotoelemente ein, für die Entfernungsmessung reicht dagegen eine zeilenartige (eindimensionale) Anordnung aus.



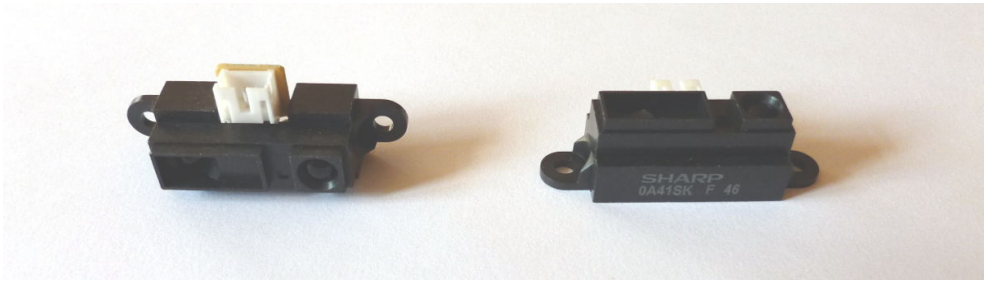
Entfernungsmessung mittels optischer Triangulation

In der mobilen Robotik sind Entfernungssensoren der Firma Sharp sehr populär. Gebräuchlich sind die in der nachfolgenden Tabelle angegebenen Typen, welche die Entfernungsmessung im Bereich einiger Dezimeter erlauben.

Gängige Entfernungssensoren von Sharp [1,2]

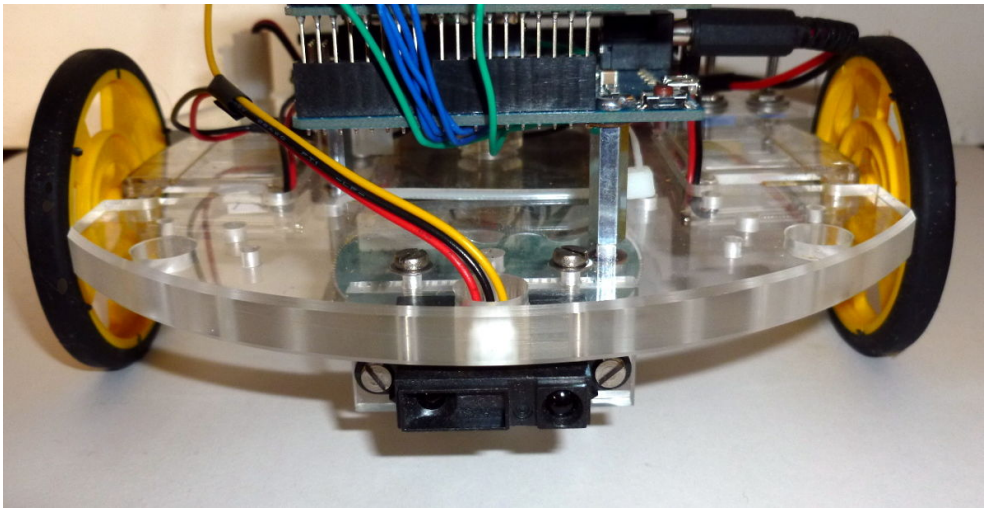
Neue Bezeichnung	*Alte Bezeichnung*	*Messbereich*
Sharp GP2Y0A41SK0F	Sharp GP2D120	4…30 cm
Sharp GP2Y0A21YK0F	Sharp GP2D12	10…80 cm
Sharp GP2Y0A02YK0F	—	20…150 cm

Hinsichtlich der Bauform sind die Sensoren GP2Y0A21YK0F und GP2Y0A41SK0F praktisch nicht zu unterscheiden. Der Sensor GP2Y0A02YK0F hat eine ähnliche Bauform, ist aber in einer Richtung etwas größer.



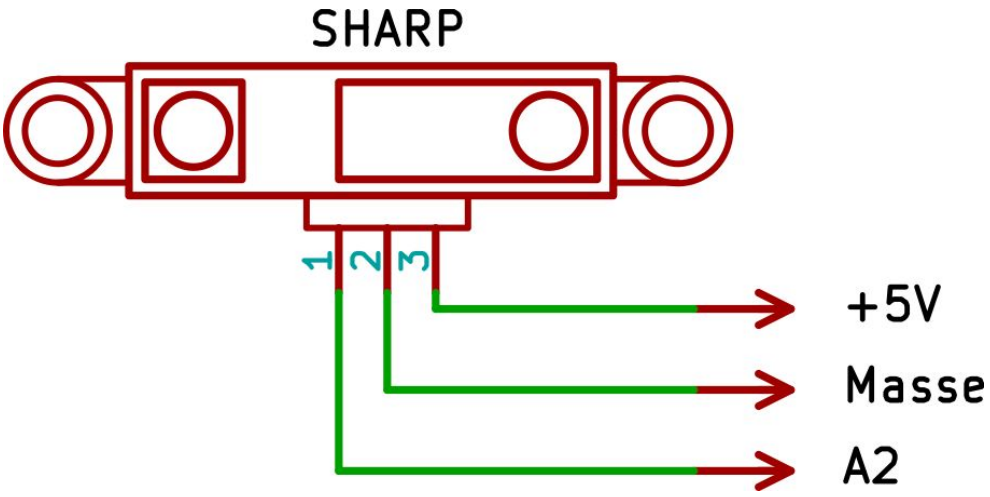
Sharp-Abstandssensoren GP2Y0A21YK0F (links) und GP2Y0A41SK0F (rechts)

Das nächste Bild zeigt einen der o.g. Sensoren, den GP2Y0A41SK0F, an der Vorderseite des mobilen Roboters.



Sharp-Abstandssensor im Frontbereich des mobilen Roboters

Die Verdrahtung ist denkbar einfach. Der Sensor verfügt über drei Anschlüsse. Zwei dieser Anschlüsse dienen der Spannungsversorgung (Masse und +5V), der dritte Anschluss liefert das Messsignal als analogen Spannungswert. Für den letzten Anschluss wurde der analoge Eingang A2 des Arduino-Boards vorgesehen.



Anschluss eines Sharp-Abstandssensors

Für die softwareseitige Abfrage des Entfernungssensors brauchen wir natürlich die für die LCD-Ausgabe relevanten Bibliotheken:

```
#include <LiquidCrystal.h>
#include <stdio.h>
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

Als nächstes definieren wir den analogen Kanal (Pin) für die Sensorabfrage und legen die Variable `x` für den Wert an:

```
const byte pinSensor = A2;
unsigned x=0;
char Zeile[21];
```

In der Funktion `setup` muss lediglich das LCD-Display initialisiert werden:

```
void setup() {
  lcd.begin(20, 4);
}
```

Die eigentliche Sensorabfrage erfolgt mit der Funktion `analogRead`. Zusätzlich wird das Sensorsignal gefiltert. Ausgegeben wird nur der vom ADC erfasste und gefilterte Wert, noch nicht die Entfernung. Da die Ausgangsspannung des Sensors im Bereich von etwa 0…3,1V liegt, wird nur etwa der halbe Messbereich des ADCs ausgeschöpft. Die ausgegebenen ADC-Werte sollten somit im Bereich 0…635 liegen. Damit die LCD-Ausgabe nicht zu stark flimmert, wurde für jeden Durchlauf der Hauptschleife eine Wartezeit von 200ms eingefügt:

```
void loop() {
    x = (4*x + analogRead(pinSensor)) / 5;
    lcd.home();
    sprintf(Zeile, "ADC: %4u", x);
    lcd.print(Zeile);
```

```
    delay(200);
}
```

Für die Inbetriebnahme des Entfernungssensors kann der Roboter durchaus mit dem PC verbunden bleiben. In diesem Fall könnte man sich die Signalwerte auch seriell über das USB-Kabel an den PC übertragen lassen und auf die Ansteuerung der LCD-Anzeige verzichten. Die Einbindung der Bibliotheken `LiquidCrystal.h` und `stdio.h` wäre dann nicht mehr nötig. In der Funktion `setup` würde man anstelle der LCD-Anzeige die serielle Verbindung initialisieren:

```
void setup() {
  Serial.begin(9600);
}
```

In der Hauptschleife wäre lediglich die Ausgabe anzupassen:

```
void loop() {
    x = (4*x + analogRead(pinSensor)) / 5;
    Serial.println(x);
    delay(200);
}
```

Die Anzeige der Messwerte ist mit der Arduino-Entwicklungsumgebung über **Werkzeuge > Serieller Monitor** möglich.

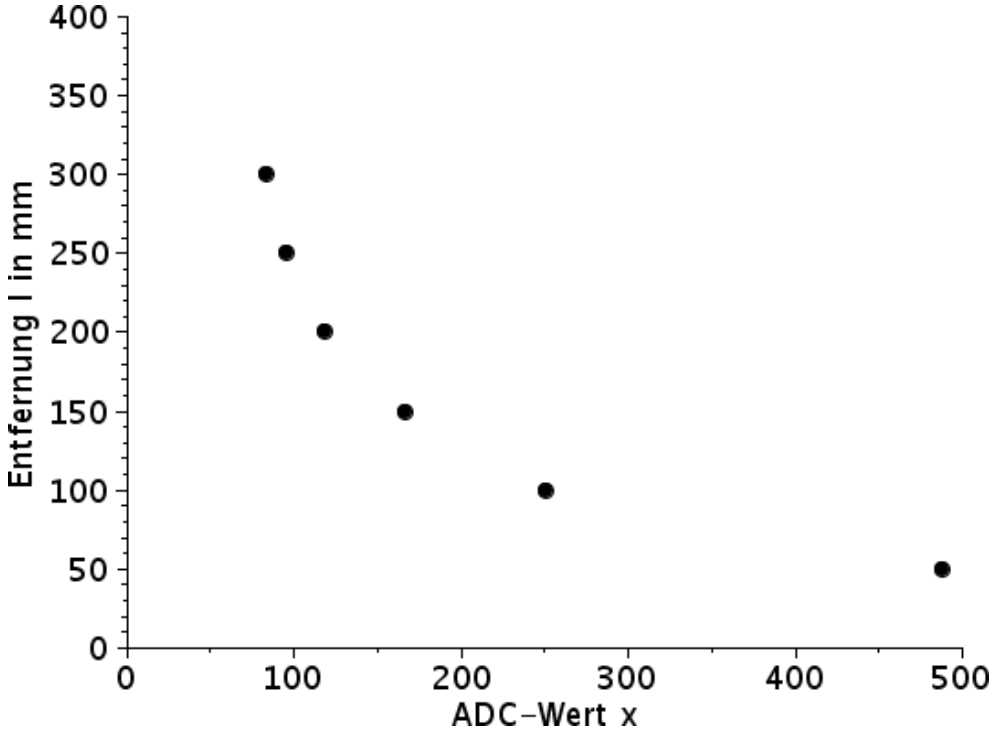
7.2 Kalibrierung des Sensors

Die vom ADC für den Entfernungssensor erfassten Werte lassen sich auch in gängige Längenmaße, z.B. mm oder cm, umrechnen. Dazu kann man entweder auf die in den Datenblättern angegebenen Kennlinien zurückgreifen oder muss eigene Messungen durchführen. Im letzteren Fall kann man das im vorangegangenen Abschnitt vorgestellte Programm zur Anzeige der ADC-Werte verwenden. In der nächsten Tabelle sind N=6 Messwerte erfasst. Für die weitere Verarbeitung wurden diese Messwerte für i=0,…,N-1 durchnumeriert.

Messungen zur Kalibrierung des Entfernungssensors GP2Y0A41SK0F

Nummer i	Entfernung l_i in mm	Sensorwert x_i (ADC)
0	50	488
1	100	250
2	150	166
3	200	118
4	250	95
5	300	83

In den Datenblättern [1,2] findet man eine grafische Darstellung der Sensorausgangsspannung als Funktion der Entfernung. Im nachfolgenden Diagramm ist dagegen die Entfernung l als Funktion des ADC-Wertes x dargestellt.



Messwerte des Sensors Sharp GP2Y0A41SK0F

Die im Diagramm gezeigten Messwerte liegen ganz klar *nicht* auf einer Geraden. Daher ist ein linearer Ansatz zur Beschreibung der Kennlinie nicht sinnvoll. Allerdings könnten die Messwerte näherungsweise auf einer Hyperbel liegen, so dass zunächst folgender Ansatz denkbar wäre:

$$l = K / x$$

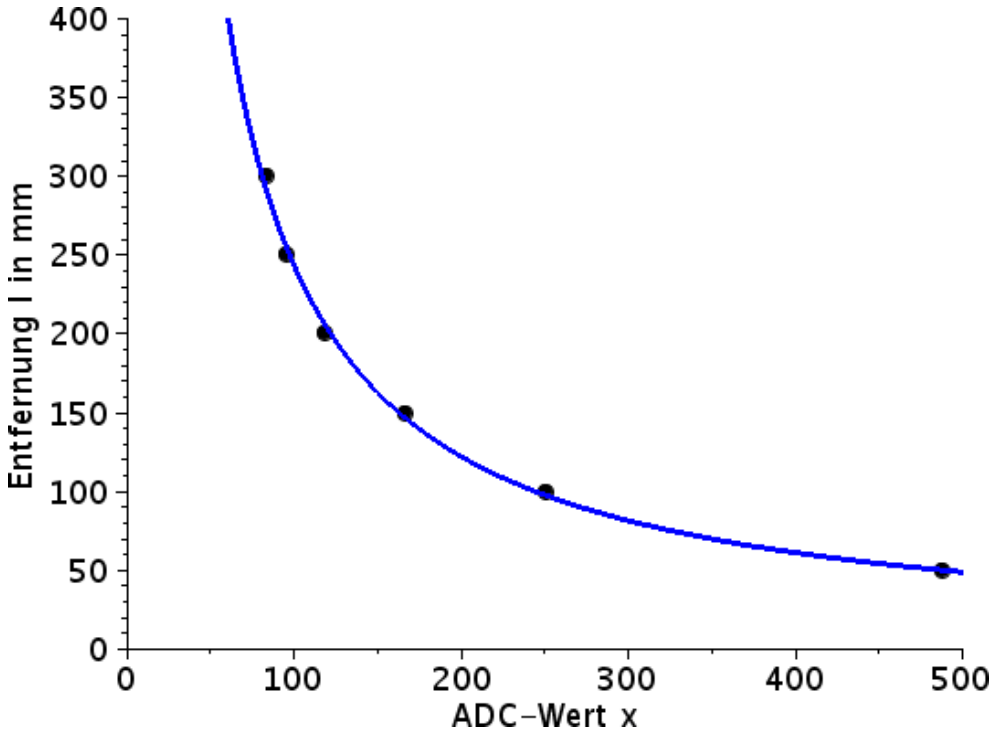
Als ein vom ADC gelieferter Wert liegt x im Bereich von 0…1023, könnte also auch den Wert Null annehmen. Um diese Singularität (Division durch Null) zu vermeiden, korrigieren wir den Ansatz wie folgt:

$$l = K / (x+1)$$

Der Koeffizient *K* lässt sich dann mit folgender Formel bestimmen:

$$K = \left(\sum_{i=0}^{N-1} \frac{l_i}{x_i + 1} \right) / \left(\sum_{i=0}^{N-1} \frac{1}{(x_i + 1)^2} \right)$$

Für die in der o.g. Tabelle dargestellten Messwerte wurde der Koeffizient *K*=24565 ermittelt. Das nachfolgende Bild zeigt eine außerordentlich gute Übereinstimmung zwischen den Messwerten und der mit dem ermittelten Koeffizienten *K* berechneten Hyperbel.



Messwerte und approximierte Kennlinie des Entfernungssensors

Verwendet man eigene Messwerte oder einen Sharp-Sensor für einen anderen Messbereich, so muss man die o.g. Formel zur Berechnung des Koeffizienten *K* selber auswerten. Diese Berechnung kann man aber auch auf dem Arduino-Board durchführen lassen. Dazu hinterlegt man als erstes die Messwerte in passenden Datenfeldern:

```
const unsigned N=6; // Anzahl Messwerte
int L[N] = { 50, 100, 150, 200, 250, 300};
int X[N] = {488, 250, 166, 118,  95,  83};
```

Für den Koeffizienten *K* wird eine globale Variable angelegt:

```
// Skalierungsfaktor
double K;
```

Die eigentliche Berechnung nach o.g. Formel muss nur einmal durchgeführt werden und erfolgt daher in der Funktion `setup`. Für den Zähler gibt es die Hilfsvariable `b`, für den Nenner die Hilfsvariable `a`. Zur Ausgabe wird die serielle Schnittstelle initialisiert:

```
void setup () {
    // Berechnung
    double a=0, b=0, c;
```

```
    for (int i=0; i<N; i++) {
        c=X[i]+1.0;
        a+=1/(c*c);
        b+=L[i]/c;
    }
    K=b/a;
    // Initialisierung Ausgabe
    Serial.begin(9600);
}
```

Die eigentliche Ausgabe erfolgt in der Hauptschleife. Der Wert für *K* wird im Sekundenrythmus ausgegeben.

```
void loop () {
    Serial.println(K);
    delay(1000);
}
```

Mit einem neuen Programm könnte man den Roboter zu einem Entfernungsmesser umfunktionieren. Den Koeffizienten *K* würde man dazu im Programm als Konstante hinterlegen:

```
// Konstante für Hyperbelkennlinie
const unsigned K = 24565;
```

Neben der Variablen `x` für den Sensorwert wäre auch eine Variable `l` für die Entfernung in mm anzulegen:

```
unsigned l,x=0;
```

In der Hauptschleife wird der Sensor abgefragt und etwas weniger stark als im vorangegangenen Programm gefiltert. Mit der o.g. Formel wird anschließend der Sensorwert *x* vom ADC in die Länge *l* umgerechnet:

```
void loop() {
    x = (2*x + analogRead(pinSensor)) / 3;
    l = K / (x+1);
    lcd.home();
    sprintf(Zeile, "L in mm: %5u", l);
    lcd.print(Zeile);
    delay(200);
}
```

7.3 Hindernisumfahrung mit einem Entfernungssensor

In diesem Abschnitt wird der mobile Roboter so programmiert, dass er in Fahrtrichtung befindliche Hindernisse erkennt und gegebenenfalls seitlich ausweicht. Die Bibliotheken zur Ansteuerung der LCD-Anzeige und des Motors werden dazu in üblicher Weise eingebunden. Für die Anzeige wurde zusätzlich die `Streaming`-Bibliothek vorgesehen:

```
// Bibliotheken: LCD und Motor
#include <LiquidCrystal.h>
```

```
#include <Streaming.h>
#include <Motor.h>
```

Auch die Einrichtung von LCD-Moduls und Motoransteuerung wird wie bisher vorgenommen:

```
// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Motor motor;
```

Die `setup`-Funktion weist dagegen keine Besonderheiten auf:

```
void setup() {
  lcd.begin(20, 4);
  motor.begin();
}
```

Der Entfernungssensor wird wieder über den analogen Anschluss A2 abgefragt. Bei dem verwendeten Sensortyp entspricht der ADC-Wert `xMax=200` einem Abstand im Bereich von 10… 15cm. Für den ADC-Wert steht die ganzzahlige Variable `x` zur Verfügung.

```
const byte pinSensor = A2; // Pin für Sensor
const int  xMax = 200;     // Bereich 10 .. 15 cm
int x;
```

In der Hauptschleife sollen die Bewegungen mit maximaler Fahrgeschwindigkeit (PWM-Wert 255) ausgeführt werden. Eine 90°-Drehung erfordert beim verwendeten Roboter ca. 400ms.

```
// Konstante für Manöver bei max. Geschw.
const int T_rueck =  800; // Zeit rückwärts
const int T_dreh  =  400; // Zeit für Drehung
```

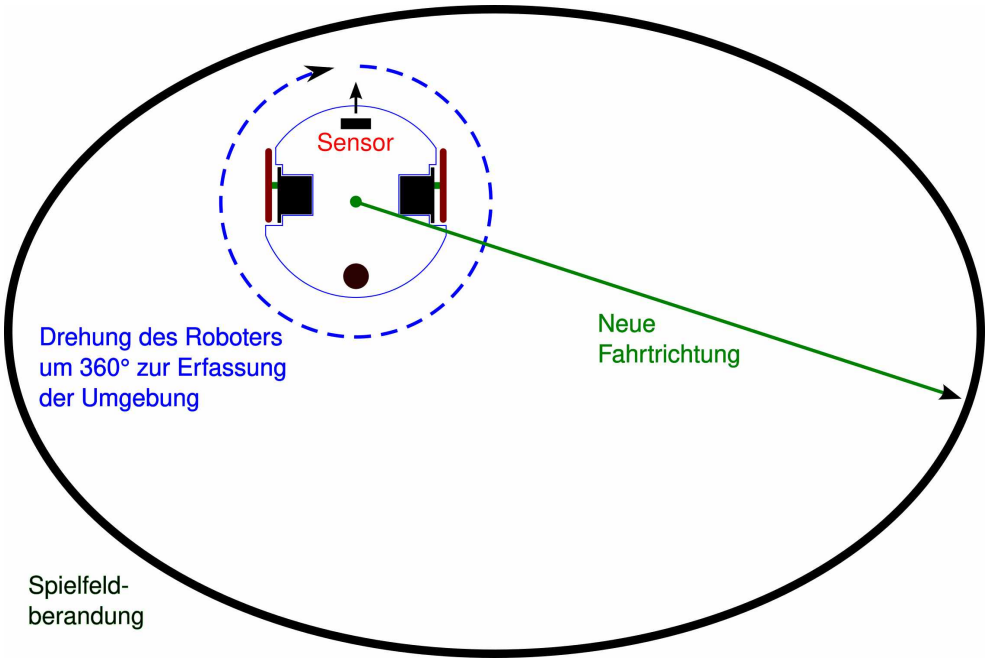
Insgesamt erhält man eine sehr übersichtliche Hauptschleife, wobei auf eine Glättung bzw. Filterung des Sensorwertes verzichtet wurde. Anstelle einer direkten Motoransteuerung mit PWM-Werten kommen jetzt die Methoden `forward`, `backward` und `right` der Klasse `Motor` zum Einsatz.

```
void loop() {
    x = analogRead(pinSensor);
    lcd.home();
    lcd<<"ADC: "<<x<<" ";
    if (x <= xMax) {
        // Vorwärtsfahrt
        motor.forward();
    }
    else {
        // Rückwärts fahren
        motor.backward();
        delay(T_rueck);
        // Rechtsdrehung
        motor.right();
        delay(T_dreh);
    }
```

```
    delay(100);
}
```

7.4 Erfassung der Umgebung und Suche nach neuer Fahrtrichtung

Mit einem einzigen Entfernungssensor kann der Roboter durch eine 360°-Drehung sein gesamtes Umfeld erfassen. In diesem Abschnitt wird dafür ein Programm entwickelt, welches nach der Erfassung der Umgebung den Roboter in die Fahrtrichtung bewegt, wo Hindernisse am weitesten entfernt sind.



Bestimmung der neuen Fahrtrichtung

Zunächst zur Vorbereitung. Die `Motor`-Bibliothek muss eingebunden werden, bei Bedarf auch die Bibliothek für die LCD-Anzeige.

```
// Instanziierung/Initialisierung
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Motor motor;
```

Der Entfernungssensor ist wieder an dem analogen Eingang A2 angeschlossen:

```
const byte pinSensor = A2;
```

Die nächsten Größen beschreiben die 360°-Drehung. Dabei wird an `N` Stellen nach jeweils der in `T_Schritt` hinterlegten Wartezeit (hier: 10ms) ein Messwert aufgenommen und in ein Element des eindimensionalen Feldes `X` geschrieben.

```
const int T_Schritt =  10;
const unsigned N = 180;
unsigned X[N];
```

Die folgende C-Funktion erfasst die Abstandsinformation für das Umfeld des Roboters. Dabei wird zunächst eine Rechtsdrehung (also im Uhrzeigersinn) des Roboters ausgelöst. Während der Drehung wird zyklisch der Sensor ausgelesen und der Messwert im Datenfeld abgespeichert. Nach der in `T_Schritt` definierten Wartezeit kommt der nächste Wert an die Reihe. Nach `N` Schritten werden die Motoren angehalten und damit die Drehung beendet.

```
// Erfassung der Sensorwerte bei 360-Grad-Drehung
void Erfassung() {
    motor.right();
    for (int i=0; i<N; i++) {
        X[i]=analogRead(pinSensor);
        delay(T_Schritt);
    }
    motor.stop();
}
```

Die Konstante `N` ist in Abhängigkeit von der Motorgeschwindigkeit und der Wartezeit `T_Schritt` so zu wählen, dass die Drehung nach möglichst 360° beendet ist. Durch den Aufruf `motor.right()` ohne Argument wird der Standardwert `VMAX` verwendet, ggf. kann dabei auch die Motorgeschwindigkeit angepasst werden. Der hier verwendete Werte N=180 bedeutet, dass der Vollkreis in Schritten von je 2° erfasst wird:

$$360° / N = 360° / 180 = 2°$$

Als nächstes erfolgt die Auswertung der im Feld `X` erfassten Messwerte. Der größte Abstand zu einem Hindernis entspricht bei den Sharp-Entferungssensoren dem kleinsten eingelesenen Wert. Nach Durchlauf der `for`-Schleife ist dieser Wert in der Variablen `xmin` hinterlegt, der entsprechende Feldeintrag in `imin`. Die C-Funktion `Auswertung` liefert die Nummer `imin` dieses Feldeintrags für die neue Richtung zurück.

```
// Bestimmung der neuen Richtung
int Auswertung() {
    unsigned imin=0, xmin=1024;
    for (int i=0; i<N; i++) {
        if (X[i]<xmin) { imin=i; xmin=X[i]; }
    }
    return imin;
}
```

Nun ist die berechnete Richtung durch Drehung einzustellen. Anschließend soll der Roboter ein Stück in die neue Richtung fahren. Diese Fahrzeit wird mit der nachfolgenden Konstanten definiert. Diese Zeitkonstante sollte nicht zu groß gewählt werden, weil der Roboter sonst während dieses Manövers das erfasste Umfeld verlässt.

```
// Fahrzeit für neue Richtung
const int T_Vorwaerts=500;
```

Das eigentliche Fahrmanöver ist in der Funktion `Fahrt` implementiert. Das Funktionsargument `i` aus 0 bis N-1 gibt die Segmentnummer für den in `N` gleiche Teile zerlegten Vollkreis an. In unserem Fall wäre der einzustellende Winkel das `i`-te Vielfache von 2°. Die Einstellung des Winkels erfolgt dadurch, dass sich der Roboter um `i` Zeiteinheiten (`T_Schritt`) nach rechts dreht. Dem schließt sich die Fahrt in die neue Richtung für die Zeit `T_Vorwaerts` an.

```
// Fahrt in neue Richtung
void Fahrt (int i) {
    // in die Richtung drehen
    motor.right();
    delay(i*T_Schritt);
    // ein Stück weiterfahren
    motor.forward();
    delay(T_Vorwaerts);
    motor.stop();
}
```

In der Hauptschleife werden die drei C-Funktionen `Erfassung`, `Auswertung` und `Fahrt` der Reihe nach aufgerufen. Zusätzlich erfolgt noch eine knapp gehaltene Ausgabe auf dem LCD-Display.

```
void loop() {
    unsigned imin;
    lcd.clear();
    lcd.print("Werte einlesen ...");
    // Werte einlesen
    Erfassung();
    // Auswertung: Minimum
    imin=Auswertung();
    lcd.clear();
    lcd<<"i= "<<imin;
    // in die Richtung drehen
    Fahrt(imin);
}
```

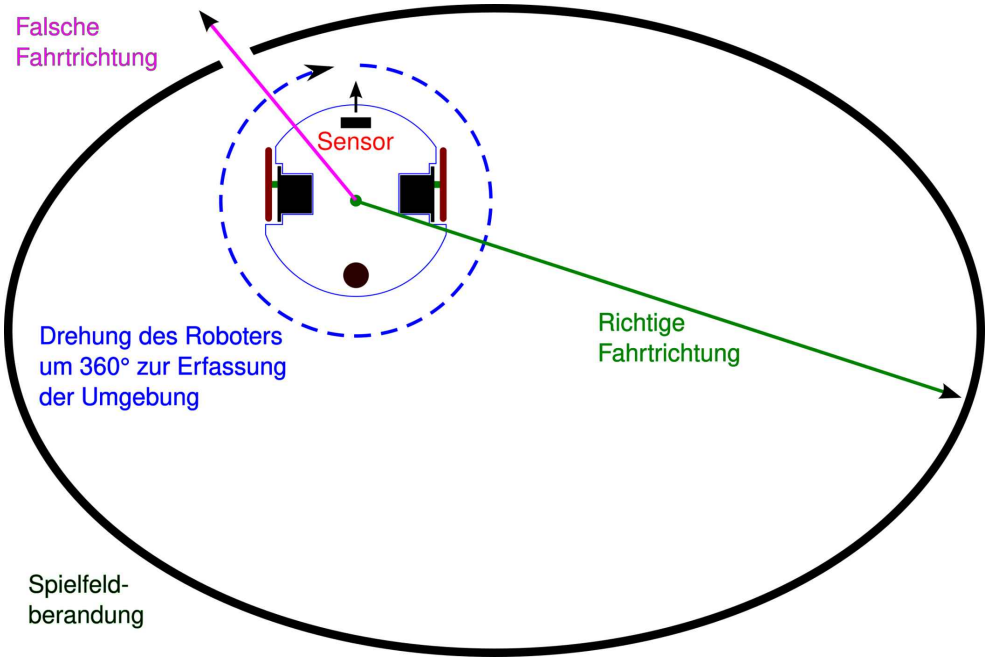
Zu Beginn ist der Roboter so auf dem Spielfeld zu positionieren, dass der beim verwendeten Sensor angegebene Mindestabstand gewährleistet ist (4cm beim Sharp GP2Y0A41SK0F bzw. 10cm beim Sharp GP2Y0A21YK0F). Hält man diesen Mindestabstand am Anfang nicht ein, so bekommt man ein Messsignal, das einen sehr großen Abstand suggeriert. Der Roboter würde dann fälschlicherweise gegen die Spielfeldberandung fahren und nicht in die Richtung, wo eigentlich Platz wäre.

Die beschriebene Implementierung kann in verschiedener Hinsicht verbessert werden. Das Einstellen der neuen Fahrtrichtung erfolgt immer durch Rechtsdrehung. Bei Winkeln über 180° kann der gewünschte Winkel schneller durch eine Linksdrehung erreicht werden. Dazu könnte man die Routine `Fahrt` folgendermaßen ändern:

```
// Fahrt in neue Richtung
void Fahrt (int i) {
    // Auswahl der Drehrichtung
```

```
    if (2*i<N) {
        motor.right();
        delay(i*T_Schritt);
    }
    else {
        motor.left();
        delay((N-i)*T_Schritt);
    }
    // ein Stück weiterfahren
    motor.forward();
    delay(T_Vorwaerts);
    motor.stop();
}
```

Bei der oben angegebenen Implementierung der Routine `Auswertung` wurde davon ausgegangen, dass das Spielfeld vollständig umrandet ist. Bei einer kleinen Unterbrechung der Berandung würde der Sensor einen großen Abstand feststellen und die Routine `Auswertung` könnte dies als neue Richtung wählen.



Problem bei der Bestimmung der neuen Fahrtrichtung

Dieses Problem lässt sich umgehen, in dem man für bei der Bestimmung der neuen Richtung nicht nur die den Messwert im i-ten Schritt einbezieht, sondern ein ganzes Winkelsegment einbezieht. Bei der verbesserten Implementierung wurden die 20 vorangegangenen und 20 nachfolgenden Feldeinträge hinzugezogen. Kommt man bei dem Feldindex `i+j` mit $-20 \leq j \leq 20$ auf einen Wert unter 0 oder über N-1, so wird die Feldnummer durch Modulorechnung (Division mit Rest) korrigiert:

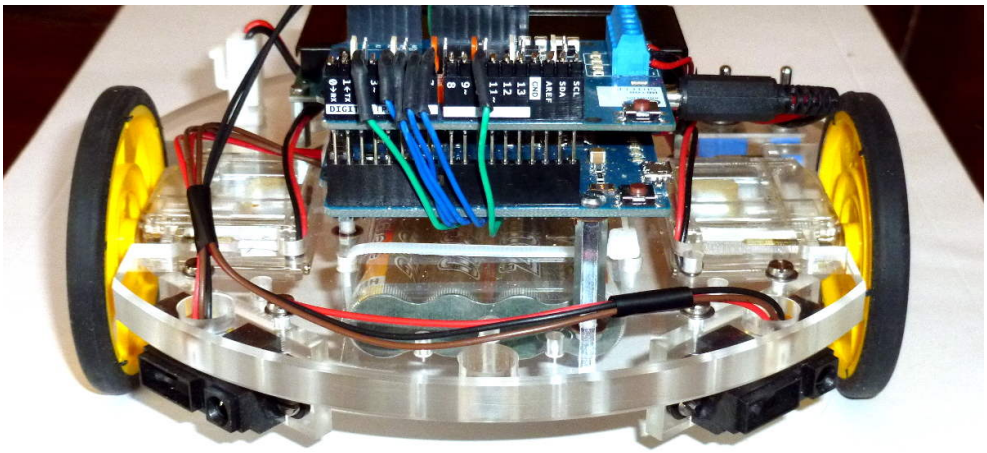
```
// Bestimmung der neuen Richtung
int Auswertung() {
```

```
    unsigned imin=0, xmin=1024, jmax;
    for (int i=0; i<N; i++) {                // Winkel i
        jmax=0;
        for (int j=-20; j<=20; j++)          // Bereich um i
            jmax=max(jmax,X[(i+j) % N]);
        if (jmax<xmin) {                                  // kein zu großer Messwert
            imin=i;
            xmin=jmax;
        }
    }
    return imin;
}
```

Die Funktionen `Erfassung`, `Auswertung` und `Fahrt` könnte man mit dem Datenfeld `X` auch in einer Klasse zusammenfassen. Wenn man zudem diese Routinen als *virtuelle* Funktionen deklariert, könnte man in abgeleiteten Klassen leicht entsprechende Verbesserung implementieren.

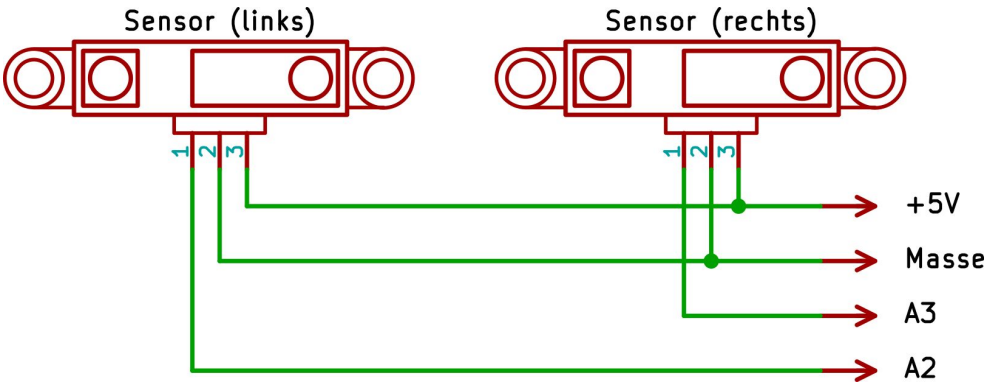
7.5 Hindernisumfahrung mit zwei Entfernungssensoren

Eine in der mobilen Robotik verbreitete Aufbauvariante ist die Montage zweier Abstandssensoren an der Vorderseite des Roboters.



Zwei Sharp-Abstandssensoren im Frontbereich des mobilen Roboters

Die Sensoren werden mit +5V versorgt und lassen sich über die analogen Eingänge A2 und A3 abfragen. Der Eingang A2 ist für den linken Sensor vorgesehen, der Eingang A3 für den rechten Sensor.



Anschluss von zwei Sharp-Abstandssensoren

Softwareseitig werden die analogen Eingänge wieder als Konstante definiert.

```
const byte pinSensorA = A2; // links
const byte pinSensorB = A3; // rechts
```

Wie bei der Hindernisumfahrung mit einem Entfernungssensor definiert die Konstante `xMax` den Sensorwert, ab welchem ein Hindernis erkannt wird.

```
// Bereich 10 .. 15 cm
const int  xMax = 200;
```

Die Abfrage beider Sensoren erfolgt mit der C-Funktion `Sensorabfrage`. Gibt diese Funktion den Wert 0 zurück, dann wurde bei keinem der Sensoren ein nahes Hindernis erkannt. Bei den Rückgabewerten 1 oder 2 wurde entweder am linken Sensor oder am rechten Sensor ein Hindernis erkannt. Erkennen beide Sensoren ein Hindernis, dann wird der Wert 3 zurückgegeben.

```
int Sensorabfrage()
{
  int x;
  x = analogRead(A2)<xMax ? 0 : 1;
  x+= analogRead(A3)<xMax ? 0 : 2;
  return x;
}
```

Für die visuellen Ausgabe der Sensorabfrage werden die vier Möglichkeiten in Textform als Feld abgelegt:

```
char *Meldung[] = {"Keins", "Links",
                   "Rechts", "Geradeaus"};
```

Die entsprechende Textausgabe zur Hinderniserkennung bzw. -klassifikation erfolgt in der Hauptschleife, so dass man die Funktion der Sensoren überprüfen kann:

```
void loop() {
    int x=Sensorabfrage();
```

```
    lcd.clear();
    lcd.print("Hindernis:");
    lcd.setCursor(0,1);
    lcd.print(Meldung[x]);
    delay(200);
}
```

Nach der Überprüfung der Sensoren können wir zur Implementierung des Ausweichmanövers übergehen. Dabei sind natürlich die Motoren des Roboters anzusteuern. Die jeweilige Fahrtrichtung wurde bisher entweder direkt über PWM-Werte der Methode `write` oder über eine der Methoden `forward`, `backward`, `right` oder `left`, die ebenfalls in der Klasse `Motor` definiert sind, übergeben. Ein anderer Weg ist die Definition benötigter Standardrichtungen als reine Datenstruktur. Die Struktur `Richtung` enthält zwei Zahlen, welche die PWM-Werte für die Motoren darstellen:

```
struct Richtung {int A; int B; };
```

Die gängigen Fahrtrichtungen sind damit leicht über die zugehörigen PWM-Werte für die Motoren zu deklarieren:

```
Richtung Vorwaerts   = {+255, +255};
Richtung Rueckwaerts = {-255, -255};
Richtung Rechts      = {+255, -255};
Richtung Links       = {-255, +255};
Richtung Stop        = {   0,    0};
```

Für die Motorsteuerung definieren wir die neue Klasse `MotorNew`. In dieser Klasse wird die Methode `drive` definiert, die als Übergabewert (Argument) die Struktur `Richtung` erhält. Die neue Methode wird als Inline-Funktion angelegt, welche die PWM-Werte aus der Struktur `Richtung` an die Methode `write` der Basisklasse `Motor` weiterleitet:

```
class MotorNew : public Motor {
    public:
    void drive (Richtung R) {
      Motor::write(R.A, R.B);
    };
};
```

Die Instanz `motor` der abgeleiteten Klasse `MotorNeu` wird über den Standard-Konstruktor initialisiert. Verwendet man eine andere Pin-Belegung als beim Arduino-Motor-Shield, dann müsste man für die abgeleitete Klasse einen weiteren Konstruktur vorsehen, der die entsprechenden Pins über den mit vier Argumenten aufzurufenden Konstruktur der Basisklasse einstellt.

```
MotorNeu motor;
```

Als nächstes sind die gewünschten Ausweichmanöver in Abhängigkeit vom Rückgabewert der Routine `Sensorabfrage` festzulegen. Solange kein Hindernis erkannt wird fährt der Roboter geradeaus. Bei einem Hindernis auf nur einer Seite (links oder rechts) weicht der Roboter in die jeweils andere Richtung aus. Wird geradeaus ein Hindernis erkannt, dann soll sich der Roboter um 180° drehen.

Sensorabfrage und resultierende Ausweichmanöver

Sensorabfrage	*Text*	*Manöver*
0	Kein	Geradeausfahrt
1	Links	Ausweichen nach rechts
2	Rechts	Ausweichen nach links
3	Geradeaus	180°-Drehung (nach rechts)

Die für die Ausweichmanöver vorgesehenen Fahrtrichtungen werden entsprechend der Numerierung der Sensorabfrage (0…3) in dem Feld `Ausweich` abgelegt.

```
Richtung Ausweich[] =
  {Vorwaerts, Rechts, Links, Rechts};
```

Die für eine 180°-Drehung benötigt Zeit ist experimentell zu bestimmen und als Wert in ms in der Konstanten `T_wende` abzulegen:

```
// 180°-Drehung
const int T_wende = 1000;
```

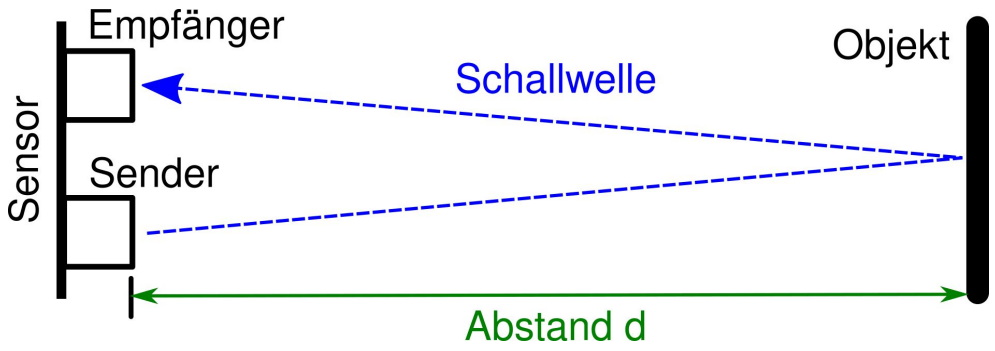
In der Hauptschleife werden zunächst die Sensoren abgefragt und das Ergebnis der qualitativen Messwertauswertung über das LCD-Display angezeigt. Anschließend erfolgt die Vorgabe der Fahrtrichtung mit der Methode `drive`. Bei einem frontal erkanntem Hindernis wird zusätzlich die entsprechende Zeit für eine 180°-Drehung abgewartet.

```
void loop() {
    int x=Sensorabfrage();
    lcd.clear();
    lcd.print(Meldung[x]);
    motor.drive(Ausweich[x]);
    if (x==3) {
        delay(T_wende);
    }
    delay(100);
}
```

7.6 Abstandsmessung mit Ultraschallsensoren

7.6.1 Prinzip der Ultraschall-Entfernungsmessung

Zur Abstandsmessung stehen neben der optischen Triangulation etliche weitere Messprinzipien zur Auswahl. Bei mobilen Robotern ist insbesondere auch die Abstandsmessung mit Ultraschallsensoren verbreitet. Bei der Messung gibt ein Ultraschallgeber (d.h. ein Sender, engl. *transmitter*) ein Signal aus, welches vom zu vermessenden Objekt (Hindernis) reflektiert wird. Ein Ultraschallmikrofon (d.h. ein Empfänger, engl. *receiver*) erfasst die ankommende Schallwelle.



Prinzip der Ultraschall-Entfernungsmessung

Weist das betrachtete Objekt den Abstand *d* (Distanz) vom Ultraschallsensor auf, so muss der Schall die doppelte Entfernung (also 2*d*) überwinden. Bezeichnet man mit *t* die Laufzeit und mit *c* die Ausbreitungsgeschwindigkeit der Schallwellen (Schallgeschwindigkeit), so kommt man auf folgenden Zusammenhang:

$$2\,d = c \cdot t$$

Der Abstand *d* ergibt sich somit aus der Formel:

$$d = t \cdot c/2$$

Die Schallgeschwindigkeit hängt von verschiedenen Faktoren ab (z.B. vom Medium und dessen Temperatur) und beträgt für Luft bei 20°C ca. c=343m/s. In der o.g. Formel wird die halbe Schallgeschwindigkeit benötigt. Dazu definieren wir die Konstante *k*:

$$k = c/2 = (343/2)\ \text{m/s} = 171{,}5\ \text{m/s}$$

Erfasst man die Laufzeit *t* der Schallwelle in Mikrosekunden (µs) und den Abstand *d* in cm, so gilt:

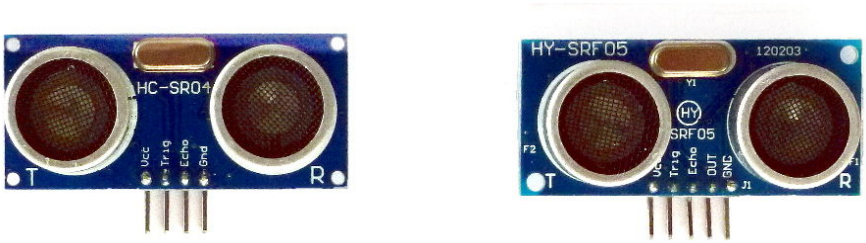
$$k = 171{,}5\ \text{m/s} = 0{,}01715\ \text{cm/µs}$$

Der Abstand in cm ergibt sich folglich näherungsweise dadurch, dass man die in µs gemessene Laufzeit durch 58 teilt:

$$k = 0{,}01715\ \text{cm/µs} \approx (1/58{,}3)\ \text{cm/µs}$$

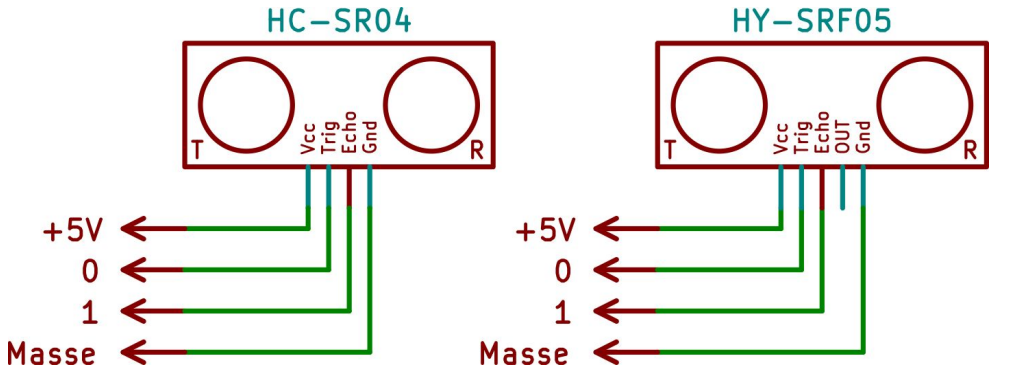
7.6.2 Betrieb der Ultraschall-Entfernungssensoren HC-SR04 und HY-SRF05

Ein gängiger Sensor ist der HC-SR04. Dieser beherbergt neben Ultraschallsender und -empfänger auch die komplette Ansteuerelektronik. Der Entfernungssensor ist für einen Messbereich von 2cm bis 4m vorgesehen. Der Ultraschall-Entfernungssensor HY-SRF05 kann als (marginale) Weiterentwicklung des HC-SR04 aufgefasst werden.



Ultraschall-Entfernungssensoren HC-SR04 (links), HY-SRF05 (rechts)

Der Sensor HC-SR04 verfügt über 4 Pins, der HY-SRF05 über 5 Pins. Neben Masse (Gnd) und Betriebsspannung (Vcc) sind zwei TTL-Signale zur Ansteuerung vorgesehen. Mit Trig wird der Messvorgang ausgelöst. Dazu ist an diesem Pin für 10µs HIGH-Pegel vorzugeben. Die Laufzeit wird als Impuls mit HIGH-Pegel über Pin Echo rückgemeldet. Beim Versuchsaufbau wurden für die Kommunikation mit dem Ultraschall-Entfernungssensor die digitalen Anschlüsse 0 und 1 des Arduino-Boards verwendet:



Anschluss des Entfernungssensors HC-SR04 (links) bzw. des HY-SRF05 im Kompatibilitätsmodus (rechts)

Für den Betrieb des Sensors HY-SRF05 im Kompatibilitätsmodus (Mode 1) zum HC-SR04 bleibt der zusätzliche Pin unbeschaltet. In den nachfolgenden Beschreibungen wird nur auf den HC-SR04 eingegangen (der ohnehin deutlich verbreiteter ist), obwohl die gleiche Software ohne Änderungen auch für den HY-SRF05 genutzt werden kann.

Zur Entfernungsmessung mit dem HC-SR04 kann man auf eine passende Arduino-Bibliothek zurückgreifen. Dazu lädt man sich die ZIP-Datei `HC_SR4_Demo_Arduino.zip` von der Firmenwebseite herunter [5]. Mit dem Menüpunkt **Sketch > Bibliothek einbinden > .ZIP-Bibliothek hinzufügen** wählt man die heruntergeladene ZIP-Datei aus und bindet sie dadurch in die

Arduino-Entwicklungsumgebung ein. Dabei wird im Arduino-Verzeichnis `libraries` das Unterverzeichnis `HC_SR4_Demo_Arduino` angelegt, welches neben den C++ Quellen (`Ultrasonic.h` und `Ultrasonic.cpp`) auch ein Arduino-Beispiel enthält.

Das Beispielprogramm wird nachfolgend (mit kleinen Anpassungen) beschrieben [5]. Zuerst werden die benötigten Headerdateien eingebunden:

```
#include <Ultrasonic.h>
#include <LiquidCrystal.h>
```

Sowohl für den Sensor als auch für die LCD-Anzeige sind zu den bereitgestellten Klassen geeignete Instanzen anzulegen. Hier ist das Beispielprogramm hinsichtlich der verwendeten Pin-Belegung zu modifizieren. Für die Abfrage des Sensors werden die digitalen Anschlüsse 0 und 1 genutzt, die dem Objekt `ultrasonic` bei der Instanziierung übergeben werden:

```
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
Ultrasonic ultrasonic(0,1);
```

In der `setup`-Funktion ist nur das LCD-Display in der üblichen Weise zu initialisieren.

```
void setup() {
  lcd.begin(20, 4);
}
```

In der Hauptschleife wird die Messung vorgenommen und das Messergebnis ausgegeben:

```
void loop() {
    lcd.home();
    lcd.print(ultrasonic.Ranging(CM));
    lcd.print(" cm ");
    delay(100);
}
```

Die Methode `Ranging` der Klasse `Ultrasonic` gibt bei dem Argument `CM` die Entfernung in cm aus (wie im Beispiel), bei dem Argument `INC` dagegen in Zoll (engl. *inch*).

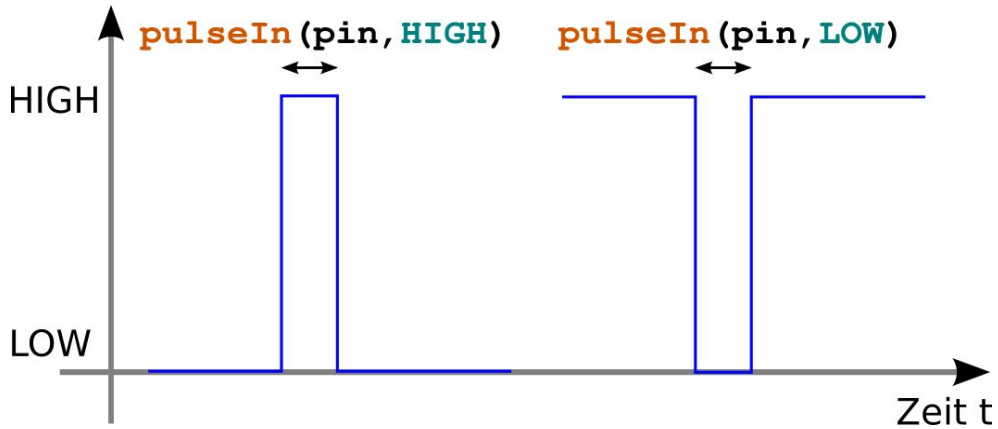
Die Sensorabfrage lässt sich aber auch ohne Installation der o.g. Zusatzbibliothek bewerkstelligen [4]. Im Sinne eines guten Programmierstils definieren wir die zur Abfrage des HC-SR04 verwendeten digitalen Kanäle als Konstanten:

```
const int pinTrig = 0;
const int pinEcho = 1;
```

In der Funktion `setup` wird der zum Auslösen der Messung verwendete Pin als Ausgang und der für die Signallaufzeit genutzte Pin als Eingang konfiguriert:

```
void setup() {
  lcd.begin(20, 4);
  pinMode(pinTrig, OUTPUT);
  pinMode(pinEcho, INPUT);
}
```

Zur Abfrage einer Impulslänge steht in der Arduino-Umgebung die Funktion `pulseIn` zur Verfügung [6]. Dieser Funktion ist als erstes Argument die Nummer des verwendeten digitalen Anschlusses zu übergeben. Das zweite Argument legt fest, ob die Länge eines HIGH- oder LOW-Impulses zu messen ist.



Messung der Breite eines HIGH- bzw. LOW-Impulses mit der Arduino-Funktion `pulseIn`

Die Funktion `pulseIn` gibt die Impulsdauer (Laufzeit der Schallwelle) in µs als Integer-Zahlentyp `unsigned long` zurück. Dafür legen wir die Variable `zeit` an:

```
long unsigned zeit;
```

Zusätzlich kann der Funktion `pulseIn` optional ein drittes Argument für den Abbruch der Messung bei Überschreiten einer bestimmten Zeit übergeben werden. Bei Erreichen des maximalen Messbereichs von 4m muss der Schall 8m zurücklegen. Mit der oben angegebenen Schallgeschwindigkeit von ca. 343m/s entspricht das einer Laufzeit von ca. 23,3ms bzw. 23300µs. Wir legen daher fest, dass die Messung nach 25ms bzw. 25000µs abgebrochen wird:

```
const long unsigned TimeOut = 25000;
```

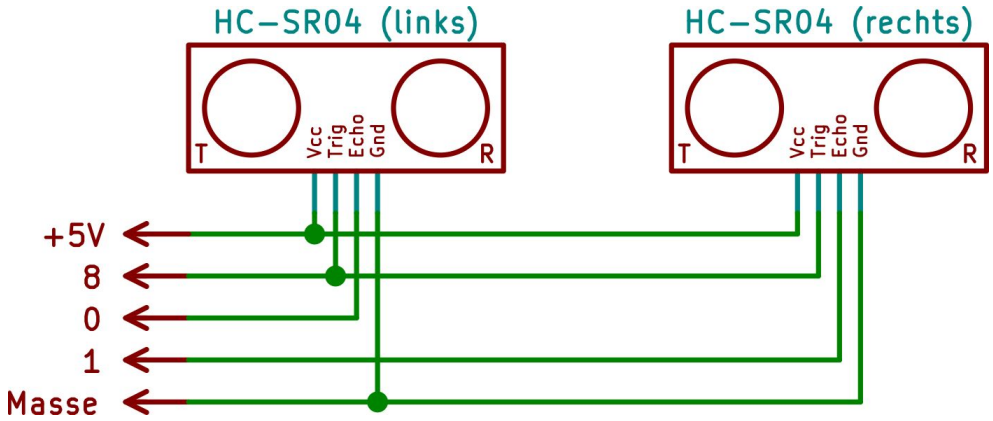
In der Hauptschleife wird der Anschluss Trig zunächst (kurz) auf LOW-Pegel gesetzt. Danach wird auf diesem Anschluss ein 10µs dauernder Impuls mit HIGH-Pegel ausgegeben, der den Messvorgang auslöst. Mit `pulseIn` wird die Laufzeit eingelesen. Dem folgen Umrechnung und Ausgabe. Hier wird der Abstand in mm angegeben, so dass die mit `pulseIn` gemessene Laufzeit durch 5,83=583/100 zu teilen ist (statt durch 58,3 wie für die Angabe in cm).

```
void loop() {
    lcd.home();
    // Messvorgang starten
    digitalWrite(pinTrig,LOW);
    delayMicroseconds(2);
    digitalWrite(pinTrig,HIGH);
    delayMicroseconds(10);
    digitalWrite(pinTrig,LOW);
    // Messwerterfassung
    zeit=pulseIn(pinEcho,HIGH,TimeOut);
```

```
    // Ausgabe
    lcd.print("Wert : ");
    lcd.print(zeit);
    lcd.print("   ");
    lcd.setCursor(0,1);
    lcd.print("Abstand : ");
    lcd.print((100*zeit)/583);
    lcd.print(" mm ");
    delay(100);
}
```

Beim Überschreiten der unter `TimeOut` definierten Laufzeit wird der Wert Null ausgegeben. Diesen Fall könnte man auch programmtechnisch abfangen und dann den maximal zulässigen Abstand (also 4m bzw. 4000mm) ausgeben.

Für den Betrieb von zwei Sensoren HC-SR04 würde man normalerweise 4 digitale Kanäle belegen. Neben den bereits verwendeten Kanälen 0 und 1 könnte man, sofern man auf die Bremsfunktion des Arduino-Motor-Shields verzichtet, die Anschlüsse 8 und 9 nutzen. Dabei lässt sich ein Kanal einsparen, indem man die Anschlüsse Trig beider Sensoren zusammenlegt. Bei der nachfolgenden Schaltungsvariante startet man über Pin 8 gleichzeitig die Messung bei beiden Sensoren und fragt einzeln über Pin 0 oder 1 den linken bzw. rechten Wert ab:



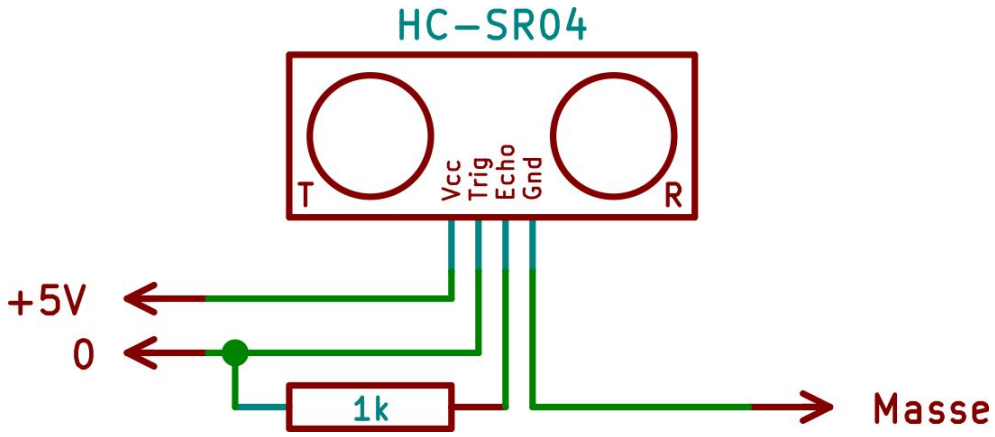
Betrieb von zwei Entfernungssensors HC-SR04 über drei digitale Anschlüsse

7.6.3 Eindrahtbetrieb des Ultraschall-Entfernungssensors HC-SR04

Laut Datenblatt [7] besitzt der HY-SRF05 neben dem Kompatibilitätsmodus (Mode 1) eine weitere Betriebsart (Mode 2), bei welcher über den gleichen Pin die Messung ausgelöst und dann eingelesen wird. Anstelle von zwei digitalen Kanälen würde man dann mit einem Kanal auskommen. Trotz intensiver Versuche gelang es dem Autor nicht, in dieser Betriebsart eine Entfernungsmessung durchzuführen. Daraus entstand die Idee, eine derartige Betriebsart für den deutlich populäreren Sensor HC-SR04 zu realisieren.

Wenn man beide Signale (Trig und Echo) über eine Leitung übertragen möchte muss verhindert werden, dass zwei Ausgänge gegeneinander arbeiten. Das Signal Trig ist beim Sensor ein Eingang, so dass wir diesen Anschluss direkt mit dem digitalen Kanal des Arduino Boards ver-

binden können. Der Anschluss Echo ist beim Sensor ein Ausgang, der über einen 1kΩ-Widerstand mit dem digitalen Kanal des Arduino Boards verbunden wird. Ist der Anschluss am Arduino Board zum Auslösen des Messvorgangs als Ausgang konfiguriert, so wird der Stromfluss zum Anschluss Echo durch den Widerstand auf 5V/1kΩ=5mA begrenzt:



Eindrahtbetrieb des Entfernungssensors HC-SR04

Für den Betrieb des Sensors ist jetzt nur ein einziger Pin erforderlich:

```
const int pinSensor = 0;
```

In der `setup`-Funktion ist lediglich die LCD-Anzeige zu initialisieren. Der digitale Kanal für den Sensor wird in der Hauptschleife konfiguriert bzw. umkonfiguriert.

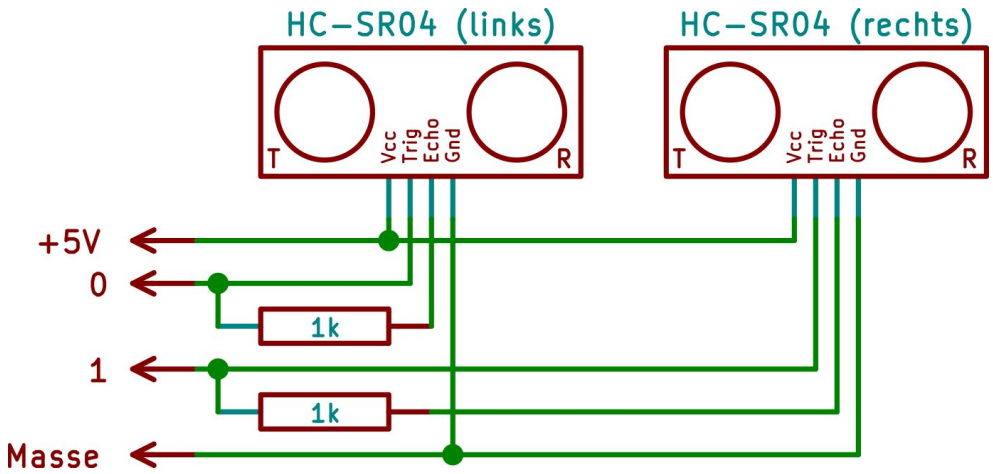
```
void setup() {
  lcd.begin(20, 4);
}
```

Zum Auslösen des Messvorgangs wird der Sensoranschluss als digitaler Ausgang konfiguriert. Die Messung wird dann in der üblichen Weise durch einen 10µs dauernden HIGH-Implus gestartet. Vor dem Auslesen des Messwertes wird der Sensoranschluss zum einem digitalen Eingang umkonfiguriert.

```
void loop() {
    lcd.home();
    // Pin als Ausgang konfigurieren
    pinMode(pinSensor, OUTPUT);
    // Messvorgang starten
    digitalWrite(pinSensor,LOW);
    delayMicroseconds(2);
    digitalWrite(pinSensor,HIGH);
    delayMicroseconds(10);
    digitalWrite(pinSensor,LOW);
    // Pin als Eingang umkonfigurieren
    pinMode(pinSensor, INPUT);
    // Messwerterfassung
    zeit=pulseIn(pinSensor,HIGH,TimeOut);
```

```
    // Ausgabe
    lcd.print("Wert : ");
    lcd.print(zeit);
    lcd.print("   ");
    lcd.setCursor(0,1);
    lcd.print("Abstand : ");
    lcd.print((100*zeit)/583);
    lcd.print(" mm ");
    delay(100);
}
```

Für den Betrieb von zwei Abstandssensoren, die man typischerweise links und rechts an der Vorderseite des Roboters anbringt, belegt man nun nur noch zwei digitale Kanäle.



Eindrahtbetrieb mit zwei Entfernungssensoren HC-SR04

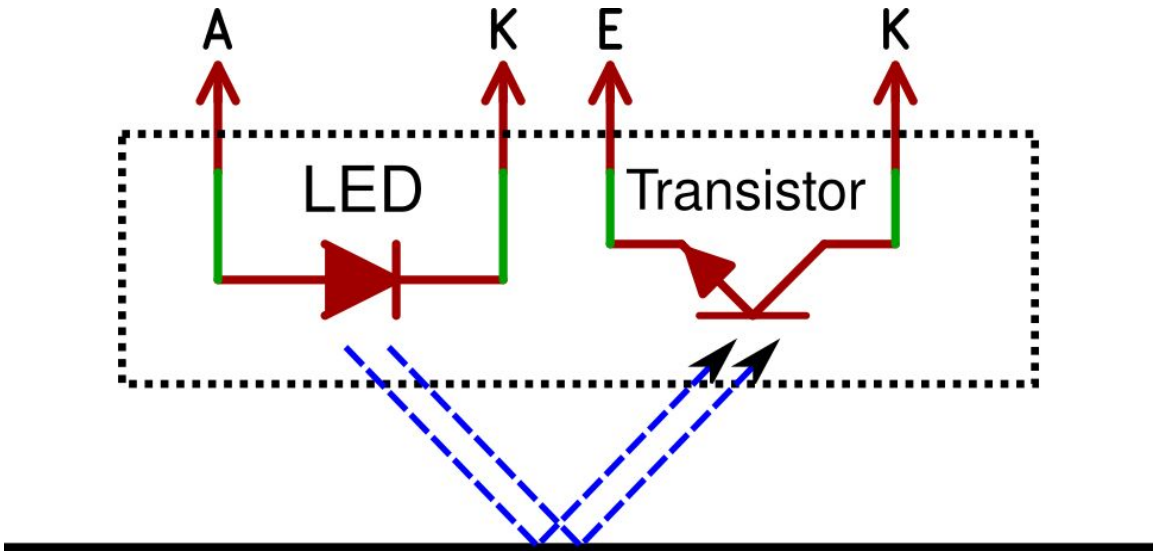
7.7 Quellenangabe

1. Sharp: GP2Y0A41SK0F, Distance Measuring Sensor Unit, Measuring distance: 4 to 30 cm, Analog output type. Datenblatt.
2. Sharp: GP2Y0A21YK0F, Distance Measuring Sensor Unit, Measuring distance: 10 to 80 cm, Analog output type. Datenblatt.
3. Sharp: GP2Y0A02YK0F, Distance Measuring Sensor Unit, Measuring distance: 20 to 150 cm, Analog output type. Datenblatt.
4. F. Maier: Bewegende Momente – Abstands-Sensoren mit Arduino programmieren. Make 2/2015, S. 54-60.
5. Cytron Technologies – Ultrasonic Ranging Module. URL: http://www.cytron.com.my/p-sn-hc-sr04
6. Arduino – Home. URL: https://www.arduino.cc/
7. SRF05 Technical Documentation. URL: http://www.robot-electronics.co.uk/htm/srf05tech.htm

8 Spielfeld- bzw. Linienerkennung

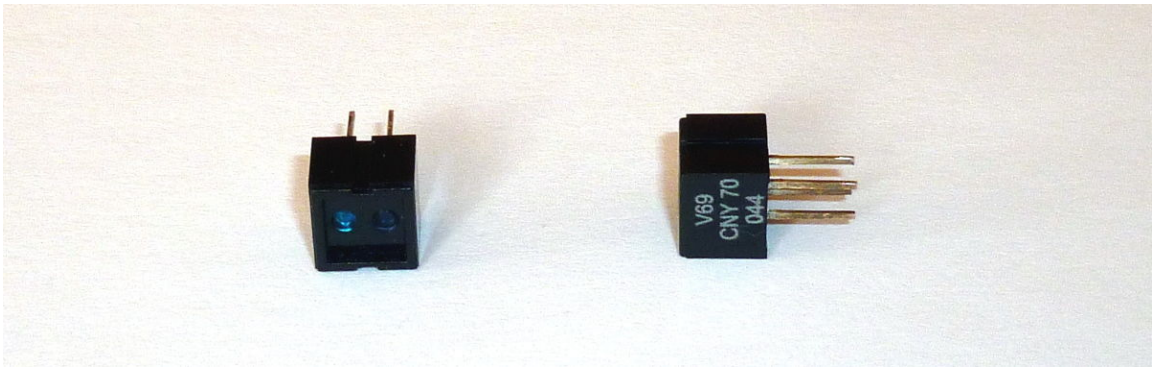
8.1 Linienerkennung mit Reflexlichtschranke

Spielfeld- bzw. Fahrbahnmarkierungen lassen sich mit einer Reflexlichtschranke erkennen. Eine LED strahlt dabei einen Lichtstrahl ab, dessen Reflexion über einen Fototransistor erfasst wird. Durch die Verwendung von infrarotem Licht kann man den Einfluss des Tageslichts unterdrücken.



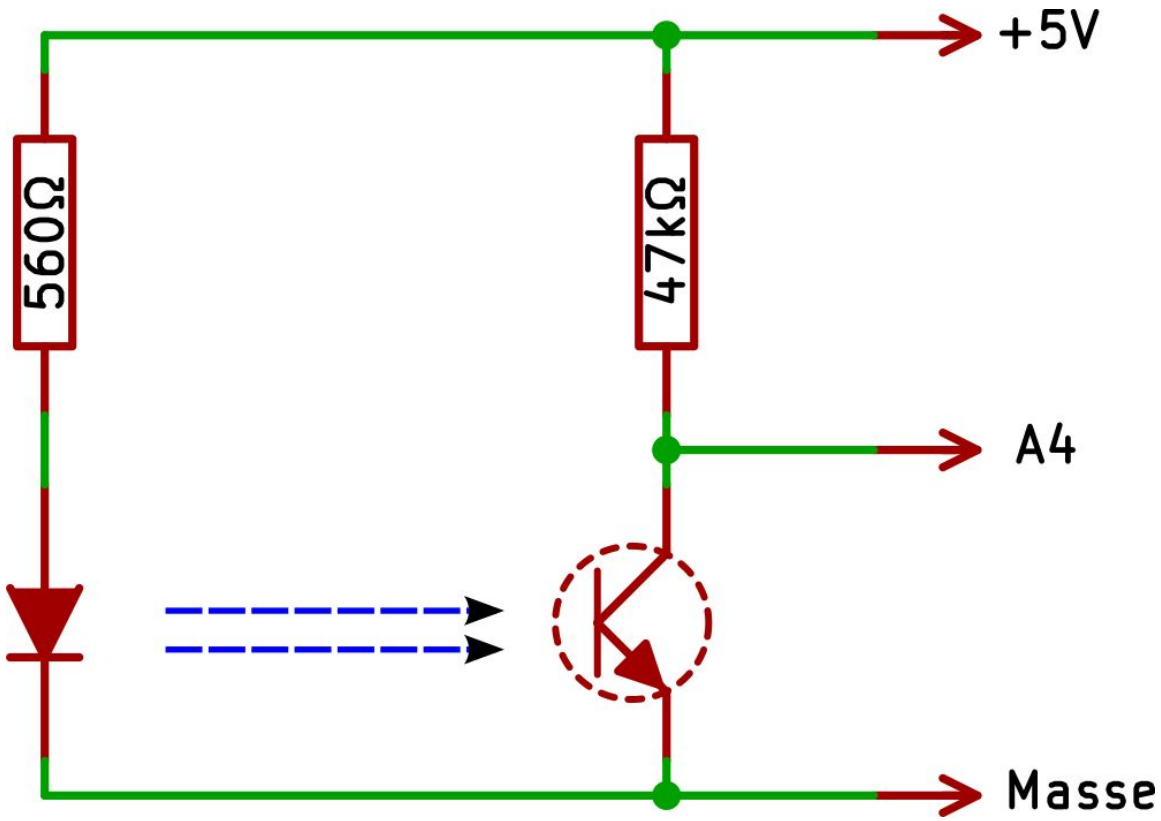
Prinzip einer Reflexlichtschranke

Sehr verbreitet ist die integrierte Reflexlichtschranke CNY70 [1]. LED und Fototransistor sind bei diesem Bauteil in einem 4-poligen Gehäuse untergebracht.



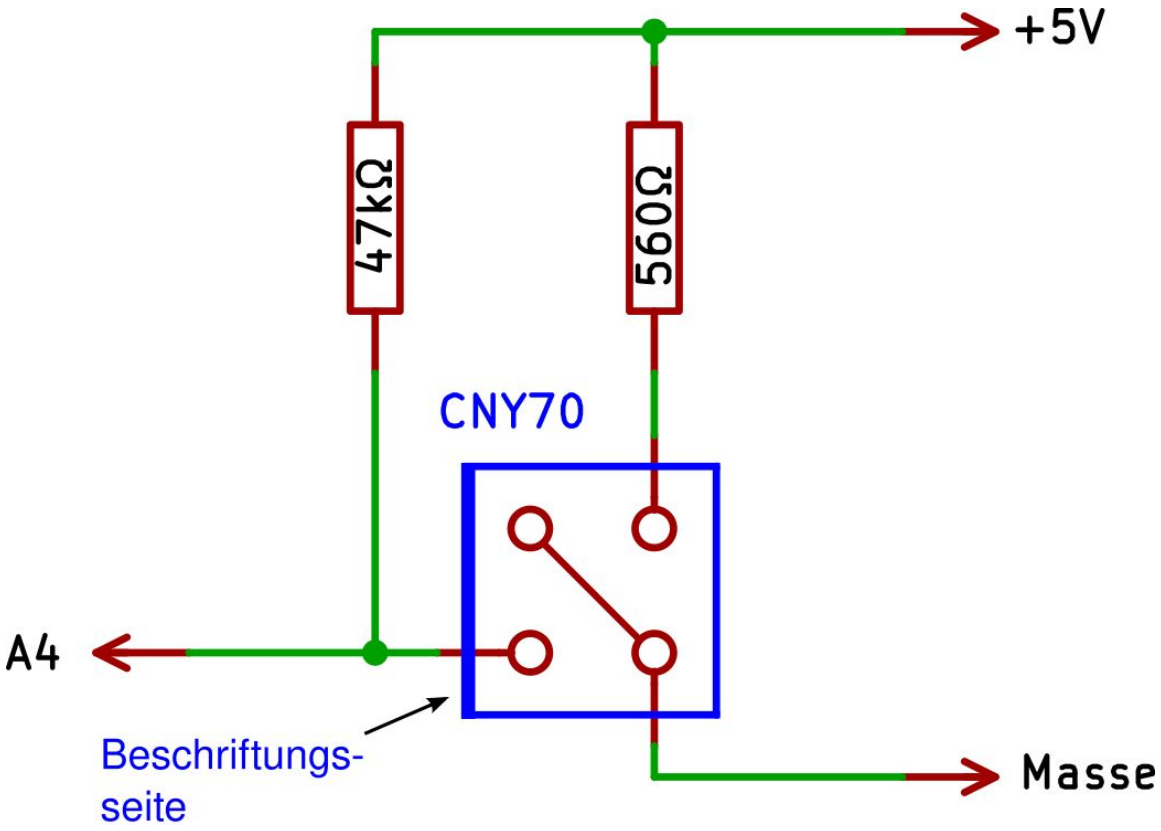
Zwei Reflexlichtschranken CNY70

Die LED wird über einen 560Ω-Vorwiderstand mit der Betriebsspannung von +5V verbunden, der Fototransistor über einen 47kΩ-Widerstand. Die Kollektorspannung des Fototransistors wird zu einem analogen Eingang (hier: A4) des Arduino-Boards geleitet. Bei hellem Untergrund wird viel Licht reflektiert, so dass der Transistor damit durchgesteuert wird und leitet. Damit liegt an A4 eine niedrige Spannung an. Bei schwarzem Untergrund wird dagegen kaum Licht reflektiert. In diesem Fall ist der Transistor gesperrt und an A4 liegt (fast) die Betriebsspannung an.



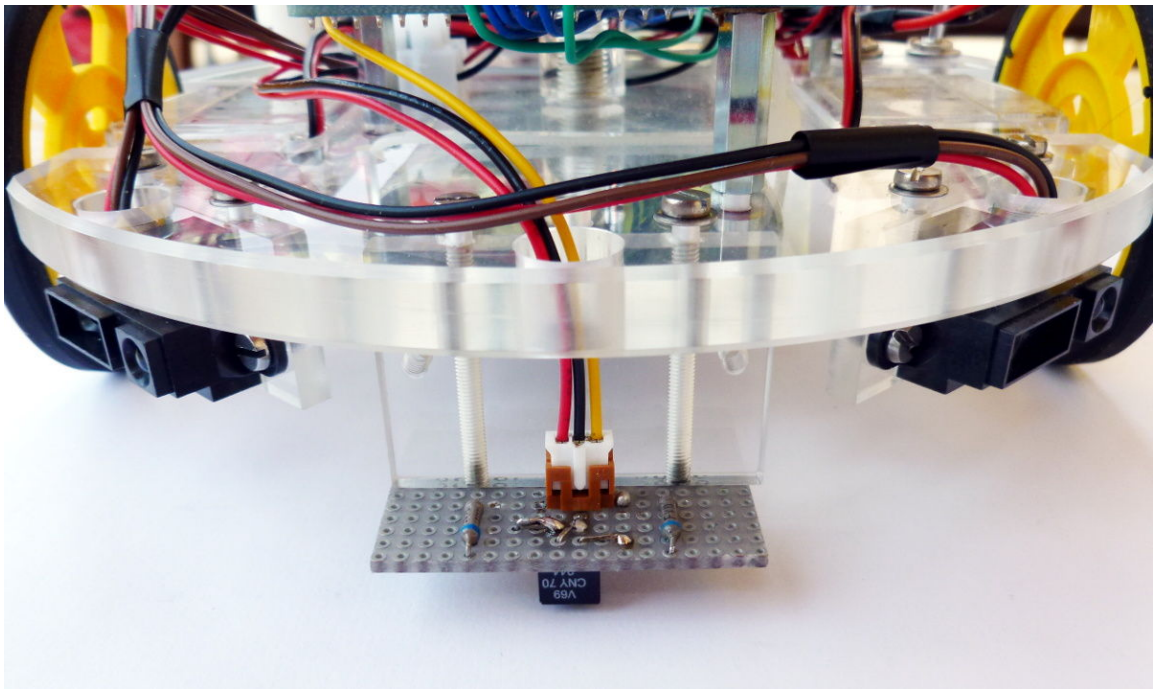
Beschaltung der Reflexlichtschranke

Das nächste Bild zeigt den Anschluss der Reflexlichtschranke CNY70 unter Beachtung der Pinbelegung [1]:



Anschluss der Reflexlichtschranke CNY70 (Blick von Anschlussseite, Beschriftung links)

Konstruktiv sollte die Reflexlichtschranke einen Abstand von höchstens 5mm gegenüber dem Spielfeld bzw. dem Untergrund haben.



Liniensensor mit Reflexlichtschranke CNY70 an der Vorderseite des Roboters

Die Sensorabfrage gestaltet sich praktisch genauso wie beim Sharp-Entfernungsmesser. Wir gehen davon aus, dass die LCD-Anzeige in der üblichen Weise softwareseitig eingebunden wurde. Der für den Sensor verwendete analoge Eingang wird als Konstante `pinLinie` hinterlegt, für Messergebnis und Ausgabe legt man die Variablen `x` bzw. `Zeile` an:

```
const byte pinLinie = A4;
unsigned x=0;
char Zeile[20];
```

In der Hauptschleife wird der analoge Eingang abgefragt (ohne Filterung) und über die LCD-Anzeige ausgegeben:

```
void loop() {
    x = analogRead(pinLinie);
    lcd.home();
    sprintf(Zeile, "ADC: %4u", x);
    lcd.print(Zeile);
    delay(200);
}
```

Bei weißem Untergrund wurden ADC-Werte von maximal 100 eingelesen, bei schwarzem Untergrund von ca. 800 und mehr. Diese Werte können aber in Abhängigkeit von den verwendeten Widerstandswerten bzw. dem Abstand der Reflexlichtschranke vom Spielfeld variieren.

8.2 Linienfolge mit einer Reflexlichtschranke

Mit dem Liniensensor lässt sich eine Linienfolgeregelung realisieren. In diesem Abschnitt wird eine einfache Implementierung vorgestellt. Zur Beschreibung der Teilmanöver, aus denen sich die Linienfolgeregelung zusammensetzt, führen wir wieder die Struktur `Richtung` ein, welche die PWM-Werte für die Motoransteuerung enthält:

```
struct Richtung {int A; int B; };
```

Auf Basis dieser Struktur definieren wir die benötigten Richtungen. Die Vorwärtsfahrt soll nicht mit voller Geschwindigkeit erfolgen. Bei `Leicht_Rechts` bzw. `Leicht_Rechts` dreht sich ein Motor in Vorwärtsrichtung, der andere bleibt stehen. Damit hat man einerseits eine Drehung in die entsprechende Richtung, andererseits auch anteilig eine leichte Vorwärtsbewegung.

```
Richtung Vorwaerts   = {200, 200};
Richtung Leicht_Rechts = {200, 0};
Richtung Leicht_Links  = {0, 200};
```

Zur Übergabe der als Struktur definierten Richtung an die Motoren leiten wir wie im letzten Kapitel aus der Basisklasse `Motor` die Klasse `MotorNeu` ab und legen die Methode `drive` an. Dazu muss natürlich vorher die Headerdatei `Motor.h` eingebunden werden.

```
class MotorNeu : public Motor {
  public:
  void drive (Richtung R) {
    Motor::write(R.A, R.B);};
};
```

Die Nummer des für den CNY70 verwendeten analogen Eingangs ist in der nächsten Konstanten abgelegt:

```
const byte pinLinie = A4;
```

Der Roboter soll solange geradeaus fahren, bis er eine Linie bzw. Spur erkennt. Danach soll er dieser Linie folgen. Zur Unterscheidung dieser zwei Zustände — Suche der Linie bzw. Linienfolge — wird die Variable `spur` verwendet. Dieser Variablen wird zunächst der Wert `false` zugewiesen, bei Erkennung einer Linie ist der Wert `true` einzustellen:

```
bool spur = false;
```

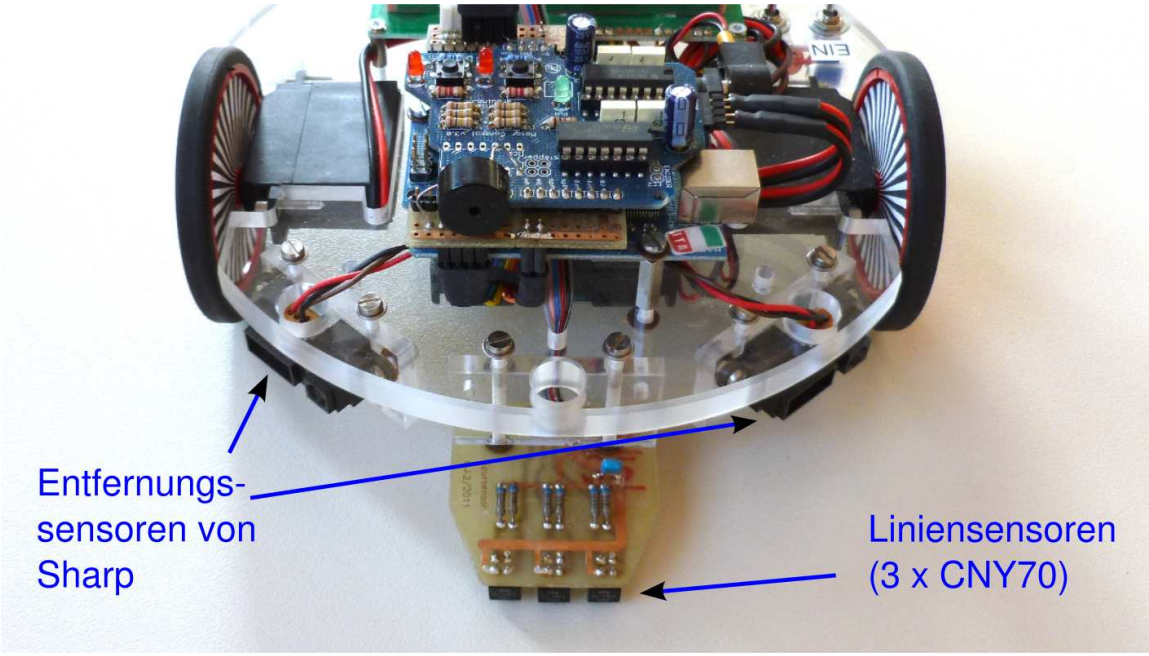
Bei weißem bzw. hellem Untergrund fährt der Roboter entweder geradeaus (wenn die Spur noch nicht erkannt wurde) oder leicht rechts (um wieder auf die schwarze Linie zu kommen). Nach dem Erkennen der schwarzen Linie hangelt sich der Roboter sozusagen an der weiß-schwarzen Kante entlang.

```
void loop()
{
    // Schwarze Linie gefunden?
    if (analogRead(pinLinie) > 500) {
        motor.drive(Leicht_Links);
        spur = true;
    }
    else {
        if (spur)
            motor.drive(Leicht_Rechts);
        else
            motor.drive(Vorwaerts);
    }
}
```

Selbstverständlich kann man das Programm noch um eine passende LCD-Ausgabe bereichern.

8.3 Linienfolge mit mehreren Reflexlicht-schranken

Mit mehreren Liniensensoren lässt sich die Linienfolgeregelung auch subtiler gestalten. Sinnvoll erscheint der Einsatz von zwei oder drei Linien-sensoren. Das nächste Bild zeigt einen Aufbau mit drei Liniensensoren:



Roboter mit zwei Entfernungs- und drei Liniensensoren

Möchte man mehrere Liniensensoren zusammen mit mehreren Entfernungssensoren verwenden, dann reichen eventuell die sechs analogen Eingänge der Standard-Boards nicht mehr aus. Hier wären folgende Lösungen denkbar:

- Man verzichtet auf die Signale zur Motorstrommessung des Arduino-Motor-Shields. Mit Durchtrennen der entsprechenden Lötbrücken auf der Leiterseite des Motor-Shields würde man die Eingänge A0 und A1 für die Sensoren nutzen können.
- Man verwendet ein Arduino-Board mit mehr analogen Eingängen, z.B. Arduino Mega oder Arduino Mega 2560.
- Man verwendet ein Board, bei dem digitale Kanäle auch für die analoge Abfrage nutzbar sind. Das trifft auf die Boards Arduino Leonardo und Arduino Micro zu.

Mit drei Sensoren müsste man sich insbesondere nicht mehr an der weiß-schwarzen oder schwarz-weißen Kante orientieren, sondern könnte auch direkt auf der Linie fahren. Im Programm würde man zunächst die verwendeten analogen Eingänge als Konstante deklarieren:

```
// Analoge Eingänge: Liniensensoren
const byte pinLinieL = A3; // Links
const byte pinLinieM = A4; // Mitte
const byte pinLinieR = A5; // Rechts
```

Ähnlich wie bei der Abfrage mit zwei Entfernungssensoren kann man die qualitative Messwertauswertung über eine einzige Funktion realisieren:

```
byte Liniensensor() {
    const int MAX=500;
    byte x;
    x = analogRead(pinLinieL)<MAX ? 0 : 1;
    x+= analogRead(pinLinieM)<MAX ? 0 : 2;
    x+= analogRead(pinLinieR)<MAX ? 0 : 4;
    return x;
}
```

Die nachfolgende Tabelle klassifiziert die möglichen Sensorzustände und gibt Vorschläge für die zu implementierenden Manöver an. Wenn beispielsweise der Roboter genau auf der Linie fährt, also entweder nur der mittlere Sensor oder alle drei Sensoren schwarz erkennen (Rückgabewert 2 bzw. 7 der Funktion `Liniensensor`), kann der Roboter auch weiterhin schnell geradeaus fahren.

Abfragezustände bei drei Liniensensoren

Wert	*Sensor links*	*Sensor mitte*	*Sensor rechts*	*Manöver, Zustand*
0	weiß	weiß	weiß	Linie suchen
1	schwarz	weiß	weiß	stark rechts
2	weiß	schwarz	weiß	schnell geradeaus
3	schwarz	schwarz	weiß	leicht rechts
4	weiß	weiß	schwarz	stark links
5	schwarz	weiß	schwarz	Kreuzung?
6	weiß	schwarz	schwarz	leicht links
7	schwarz	schwarz	schwarz	schnell geradeaus

Diese Anregungen müssen noch in ein lauffähiges Programm übersetzt werden. Hierbei gibt es auch zahlreiche Möglichkeiten, um eigene Ideen unterzubringen.

8.4 Quellenangabe

1. Vishay Telefunken: CNY70, Reflective Optical Sensor with Transistor Output. Datenblatt.

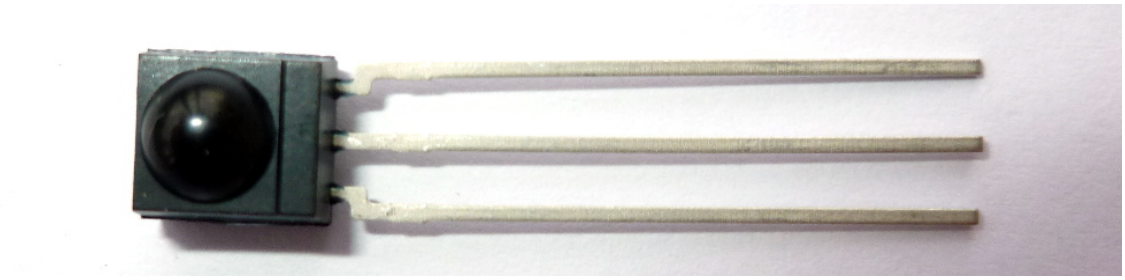
9 Drahtlose Steuerung des Roboters

9.1 Infrarot-Fernbedienung

Infrarot-Fernbedienungen findet man praktisch in jedem Haushalt, beispielsweise für den Fernseher und die Stereoanlage. Auch viele Kleinmodelle (z.B. Mini-Helikopter) werden mittels Infrarot (IR) gesteuert. Daher bietet es sich an, die ohnehin vorhandenen IR-Fernbedienungen auch zur Steuerung des mobilen Roboters einzusetzen. Diese Fernbedienungen senden über eine IR-Leuchtdiode ein moduliertes Signal aus, dessen Trägerfrequenz typischerweise im Bereich von 30…56 kHz liegt.

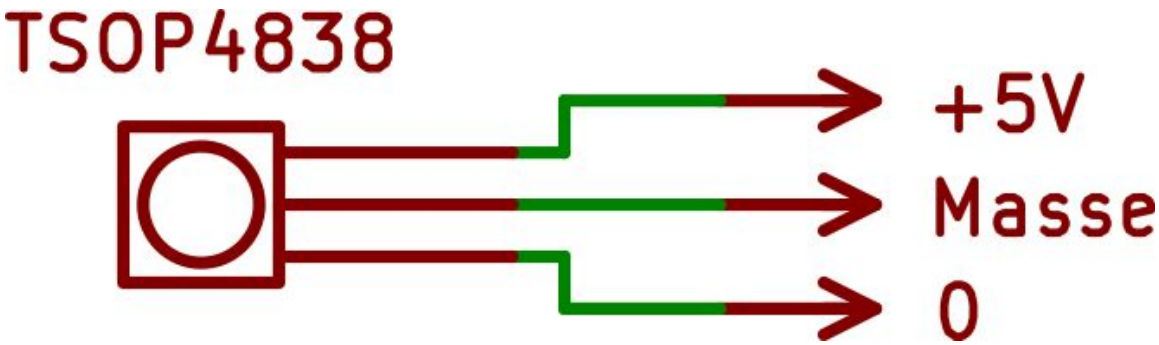
9.1.1 Anschluss des Empfängers und Abfrage der IR-Codes

Für den Empfang eines IR-Signals könnte man prinzipiell eine IR-Fotodiode oder einen IR-Fototransistor nutzen. Besser ist jedoch der Einsatz eines integrierten IR-Empfängers wie dem PNA4602 oder dem TSOP4838, wo das empfangene Signal verstärkt und vorverarbeitet wird. Die angegebenen Empfänger sind für die sehr verbreitete Trägerfrequenz von 38 kHz vorgesehen und dürften daher mit den meisten Fernbedienungen funktionieren. Im Fachhandel werden zahlreiche ähnliche IR-Empfänger angeboten.



Infrarot-Empfänger TSOP4838

Zum Anschluss des IR-Empfängers TSOP4838 ist neben Betriebsspannung und Masse seitens des Arduino-Boards ein digitaler Eingang nötig. In nachfolgenden Abbildung wurde dafür Pin 0 vorgesehen. Manche IR-Empfänger weisen allerdings eine andere Anschlussbelegung auf.



Anschluss des Infrarot-Empfängers TSOP4838

Die Auswertung des vom IR-Empfänger gelieferten Signals erfolgt mit Hilfe der Bibliothek Arduino-IRremote. Diese Bibliothek unterstützt übrigens auch die Ansteuerung eines IR-Senders, so dass man beispielsweise eine Universalfernbedienung mit Arduino aufbauen kann [1]. Bei der Installation der Bibliothek ist wie folgt vorzugehen:

1. Herunterladen der Bibliothek Arduino-IRremote als ZIP-Datei, z.B. über GitHub [2].
2. Zur Einfügen der Bibliothek in die Arduino-Entwicklungsumgebung wählt man entweder über den Menüpunkt **Sketch > Bibliothek einbinden > .ZIP-Bibliothek hinzufügen** die ZIP-Datei aus oder entpackt die ZIP-Datei direkt im Arduino-Verzeichnis `libraries`.
3. Aus dem Arduino-Programmverzeichnis `libraries`, welches unter Linux typischerweise im Verzeichnis `/usr/share/arduino` liegt, ist das Unterverzeichnis `RobotIRremote` zu löschen.

Der dritte Schritt ist nötig, da es bei den Dateien `IRremote.h` und `IRremote.cpp` einen Konflikt mit den gleichnamigen Dateien der Bibliothek `RobotIRremote` des wenig verbreiteten offiziellen Arduino-Roboters gibt [3]. Die Bibliothek `Arduino-IRremote` sollte dann (ggf. nach einem Neustart der Arduino-Entwicklungsumgebung) verfügbar sein. Unter dem Menüpunkt Examples sind für diese Bibliothek verschiedene Beispielprogramme hinterlegt. Durch Aufruf des Beispielprogramms `IRrecvDemo` kann man den IR-Empfänger testen, wobei lediglich der verwendete digitale Pin des Arduino-Boards anzupassen ist. (Im Beispielprogramm ist der Pin 11 vorgesehen, der bei uns jedoch vom Arduino-Motor-Shield belegt ist.) Beim Betätigen einer IR-Fernbedienung wird der empfangene Steuer-Code über den seriellen

Monitor der Arduino-Entwicklungsumgebung ausgegeben (**Werkzeuge > Serieller Monitor**).

Basierend auf dem Beispielprogramm der Bibliothek soll die Ausgabe der IR-Codes in ähnlicher Weise über die LCD-Anzeige des Roboters erfolgen. Dazu bindet man zuerst die benötigten Bibliotheken ein:

```
#include <LiquidCrystal.h>
#include <IRremote.h>
```

Die Anzeige wird in der gewohnten Weise behandelt:

```
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

Bei der Initialisierung des für die Dekodierung des IR-Signals zuständigen Objekts ist der für den Empfänger verwendete digitale Pin zu übergeben:

```
const int pinIR = 0;
IRrecv IRempfaenger(pinIR);
```

Zum Abspeichern der beim Dekodieren erhaltenen Werte ist ein weiteres Objekt anzulegen:

```
decode_results ergebnis;
```

In der `setup`-Routine wird die LCD-Anzeige initialisiert und die Abfrage des IR-Empfängers gestartet:

```
void setup()
{
  lcd.begin(20, 4);
  IRempfaenger.enableIRIn();
}
```

Die Hauptschleife gibt jedes dekodierte Signal auf der LCD-Anzeige aus:

```
void loop() {
    if (IRempfaenger.decode(&ergebnis)) {
        IRempfaenger.resume();
        lcd.clear();
        lcd.print("IR-Code: ");
        lcd.print(ergebnis.value, HEX);
```

```
    }
}
```

9.1.2 Motoransteuerung mittels IR-Fernbedienung

Das beschriebene Beispielprogramm zur Anzeige der IR-Codes soll hinsichtlich einer Bewegungssteuerung des Roboters erweitert werden. Für die Ansteuerung des Motors binden wir die Bibliothek Motor ein, um dann auf die Methoden `forward`, `backward`, `right`, `left` und `stop` zugreifen zu können:

```
Motor motor;
```

Zur Steuerung des Roboters muss man auf einer vorhandenen Fernbedienung insgesamt fünf Tasten für die Vorwärts- und Rückwärtsfahrt, die Rechts- und Linksdrehung sowie das Anhalten auswählen. Die Steuer-Codes, welche die Fernbedienung bei der Betätigung der entsprechenden Tasten aussendet, können mit dem im letzten Abschnitt beschriebenen Programm angezeigt werden und sind jetzt als Konstante zu hinterlegen. Bei der im Test verwendeten Fernsteuerung wurden die folgenden Codes ermittelt:

```
// Abgelesene IR-Codes
const long IR_vor    = 0xB4B4E21DL;
const long IR_rueck  = 0xB4B412EDL;
const long IR_links  = 0xB4B49A65L;
const long IR_rechts = 0xB4B45AA5L;
const long IR_stop   = 0xB4B41AE5L;
```

Für das aktuelle Abfrageergebnis wird die Variable `IRcode` angelegt:

```
long IRcode;
```

In der Hauptschleife wird zunächst der IR-Code ausgewertet und auf der LCD-Anzeige ausgegeben. Bei Übereinstimmung mit einem der hinterlegten IR-Codes wird die entsprechende Bewegung des Roboters eingeleitet:

```
void loop() {
    // Abfrage des IR-Codes
    if (IRempfaenger.decode(&ergebnis)) {
        IRcode = ergebnis.value;
        IRempfaenger.resume();
        lcd.clear();
        lcd.print("IR-Code: ");
```

```
            lcd.print(IRcode, HEX);
            // Anzeige und Motoransteuerung
            lcd.setCursor(0,1);
            switch(IRcode) {
                case IR_vor:
                    lcd.print("vorwaerts");
                    motor.forward();
                    break;
                case IR_rueck:
                    lcd.print("rueckwaerts");
                    motor.backward();
                    break;
                case IR_links:
                    lcd.print("links");
                    motor.left();
                    break;
                case IR_rechts:
                    lcd.print("rechts");
                    motor.right();
                    break;
                case IR_stop:
                    lcd.print("stop");
                    motor.stop();
                    break;
                default:
                    lcd.print("IR-Code unbekannt");
            }
        }
}
```

Als Erweiterung des beschriebenen Testprogramms kann man beispielsweise kombinierte Bewegungen (z.B. rechts vorwärts) vorsehen oder über zwei weitere Tasten der Fernbedienung (schneller, langsamer) die Fahrgeschwindigkeit einstellen.

9.2 Funkfernsteuerung

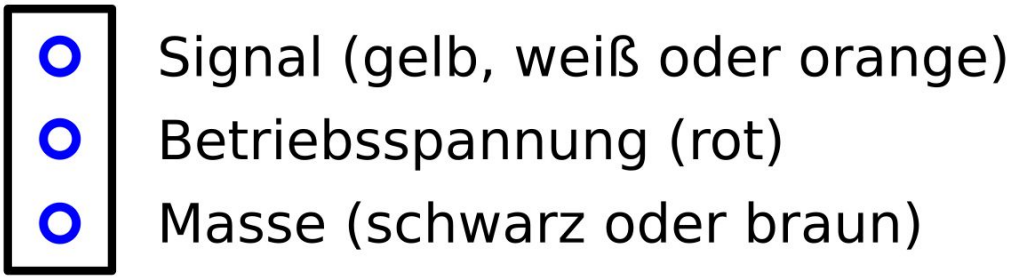
9.2.1 Anschluss und Abfrage des Empfängers

Funkfernsteuerungen verwendet man u.a. im Auto-, Schiffs- bzw. Flugzeugmodellbau. Für den Funkbetrieb von Fernsteuerungen sind verschiedene Frequenzbänder freigegeben, z.B. 27 MHz, 35 MHz oder 2,4 GHz. Bei dem in

diesem Kapitel beschriebenen Aufbau kam der Sender HT-4 in Verbindung mit dem Empfänger HR-4 von Reely zum Einsatz [4]. Tatsächlich kann auch fast jede andere Fernsteuerung verwendet werden. Wichtig ist nur, dass Sender und Empfänger korrekt miteinander kommunizieren.

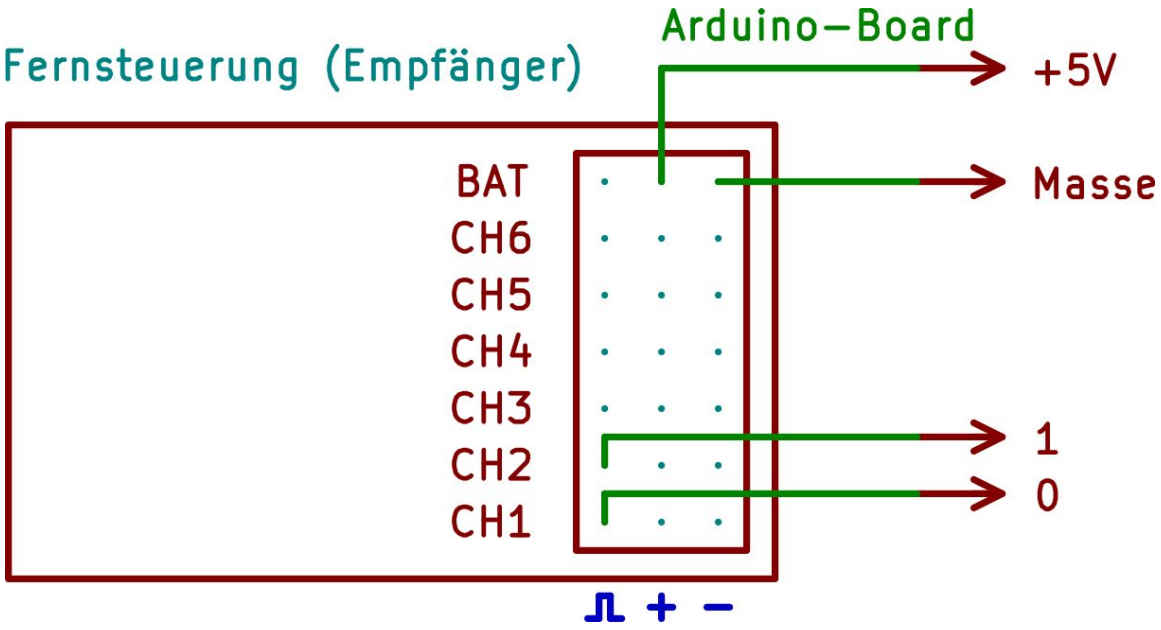
Empfänger HR-4 der Firma Reely

Die üblichen Fernsteuerungen sind primär für die Ansteuerung von Servos ausgelegt. Servos werden über drei Anschlüsse (Betriebsspannung, Masse, Signal) mit dem Empfänger der Fernsteuerung verbunden. Auch wenn es etliche verschiedene Anschlussvarianten gibt [5,6], so folgen die gängigen Servos dem in der nachfolgenden Abbildung angegebenen Anschlussschema mit dem mittleren Anschluss für die Betriebsspannung. Dieses Schema wird übrigens auch für die Ansteuerung von Motorreglern genutzt.



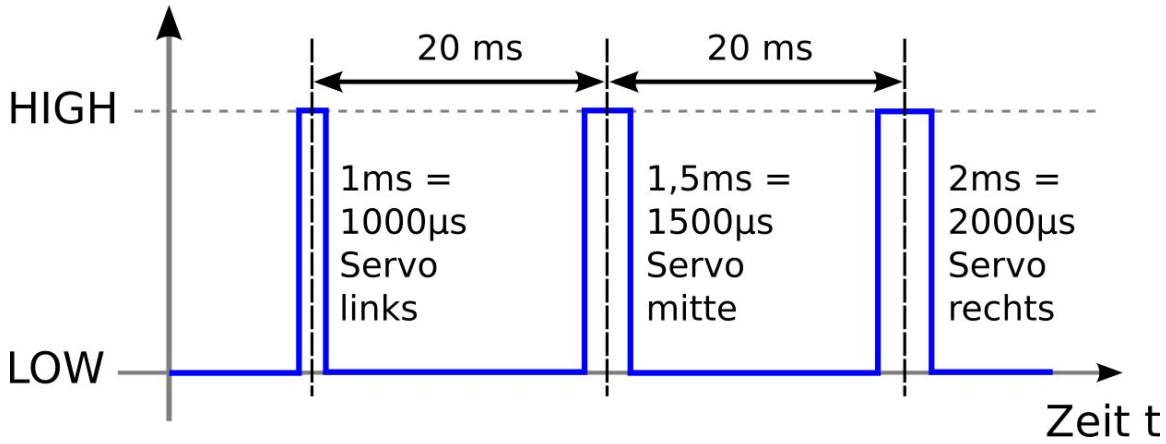
Typische Konstellation der Servoanschlüsse

Bei der verwendeten Fernsteuerung handelt es sich um eine 4-Kanal-Anlage, auch wenn die Bauform des Empfängers mit den Anschlüssen CH1 bis CH6 eine 6-Kanal-Anlage suggeriert. Dabei steht die Abkürzung CH für Kanal (engl. *channel*). Für den Roboter werden nur zwei Kanäle benötigt. Wir verwenden die Kanäle CH1 und CH2, die bei Flugzeugmodellen typischerweise mit dem Quer- und dem Seitenruder belegt sind. Die Servoausgänge werden mit den digitalen Anschlüssen 0 und 1 des Arduino-Boards verbunden. Über den zusätzlichen Anschluss BAT verbindet man den Empfänger mit Masse und Betriebsspannung. Dieses generelle Anschlussschema ist auch bei zahlreichen anderen Empfängern anzutreffen.



Anschluss des Empfängers HR-4

Die Ansteuerung der Servos erfolgt über Pulsbreitenmodulation (PWM). Die Mittelstellung des Servos wird über einen HIGH-Impuls mit einer Dauer von 1,5ms eingestellt. Für den linken bzw. rechten Anschlag erwartet der Servo HIGH-Impulse mit einer Dauer von 1ms bzw. 2ms. Die betreffenden Impulse werden mit einer Frequenz von ca. 50Hz, also in einem Zeitraster von ca. 20ms, generiert.



Servosignale (PWM)

Für einen ersten Test der Fernsteuerung wird das LCD-Display in der üblichen Weise eingebunden:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(10, 2, 4, 5, 6, 7);
```

Die zur Abfrage der Servosignale verwendeten digitalen Kanäle 0 und 1 werden als Konstante deklariert:

```
// Pins für Empfänger
const int Kanal1 = 0;
const int Kanal2 = 1;
```

In der `setup`-Funktion stellt man wie gewohnt die Größe des verwendeten LCD-Moduls ein. In Hinblick auf die Fernsteuerung werden die betreffenden Kanäle als Eingänge konfiguriert:

```
void setup() {
  lcd.begin(20, 4);
  pinMode(Kanal1, INPUT);
  pinMode(Kanal2, INPUT);
}
```

Die Impulslängen der Servosignale werden in der Hauptschleife mit der Funktion `pulseIn` abgefragt und über die LCD-Anzeige ausgegeben:

```
void loop() {
    // Abfrage des Empfängers
```

```
    long unsigned x = pulseIn(Kanal1,HIGH);
    long unsigned y = pulseIn(Kanal2,HIGH);
    // Ausgabe
    lcd.home();
    lcd.print("Kanal 1: ");
    lcd.print(x);
    lcd.print("     ");
    lcd.setCursor(0,1);
    lcd.print("Kanal 2: ");
    lcd.print(y);
    lcd.print("     ");
}
```

Bei korrekt funktionierender Fernsteuerung sollten für die Impulslängen von 1000µs bis 2000µs der beiden Fernsteuerkanäle auch Zahlenwerte im Bereich von ca. 1000 bis 2000 angezeigt werden.

9.2.2 Umrechnung der Signale zur Motoransteuerung

Im letzten Abschnitt wurde die Verknüpfung zwischen dem Empfänger der Fernsteuerung und dem Arduino-Board sowohl hard- als auch softwareseitig hergestellt. Die abgefragten Signale müssen für die Motoransteuerung angepasst werden. Dieser Abschnitt befasst sich mit der benötigten Anpassung bzw. Umrechnung. Es wird davon ausgegangen, dass der Motor und ggf. die LCD-Anzeige softwareseitig in der üblichen Weise eingebunden sind.

Der Empfänger der Fernsteuerung sollte PWM-Signale mit Impulslängen im Bereich von 1000µs bis 2000µs liefern. Die tatsächlich auftretenden Werte können sich je nach Fernsteuerung etwas unterscheiden. Mit dem im vorangegangenen Abschnitt vorgestellten Programm können wir nicht nur die Funktion der Fernsteuerung überprüfen, sondern auch die gemessenen Impulslängen ablesen. Bei der verwendeten Fernsteuerung wurden Impulslängen im Bereich von 1300 bis 2500 (beides in µs) erfasst. Die Grenzwerte (minimaler und maximaler Werte) erfassen wir in zwei Konstanten:

```
// Impulslängen in µs
const int TMIN = 1300;
const int TMAX = 2500;
```

In der Hauptschleife `loop` würde man zuerst den bereits im vorherigen Abschnitt vorgestellten Programmteil zur Abfrage der Impulslängen einfügen:

```
// Abfrage des Empfängers
long unsigned x = pulseIn(Kanal1,HIGH);
long unsigned y = pulseIn(Kanal2,HIGH);
```

Bei Bedarf kann man die abgefragten Werte natürlich wieder über die LCD-Anzeige ausgeben. Die eingelesenen Signale würde man unter Nutzung der Funktion `constrain` auf den durch die Konstanten `TMIN` und `TMAX` beschriebenen Wertebereich beschränken:

```
// Beschränkung
int u = constrain(x,TMIN,TMAX);
int v = constrain(y,TMIN,TMAX);
```

Im nächsten Schritt wird der Bereich von `TMIN` und `TMAX` auf den Wertebereich der Motorsignale (typischerweise -255 bis +255) mit Hilfe der Funktion `map` umskaliert:

```
// Skalierung
u = map(u,TMIN,TMAX,-255,+255);
v = map(v,TMIN,TMAX,-255,+255);
```

Zum Testen des bisher erstellten Programms könnte man die berechneten Werte unmittelbar auf den Motor leiten:

```
motor.write(u,v);
```

Die verwendeten Kanäle 1 und 2 (CH1 und CH2) der Fernbedienung sind oft mit einem der zwei Steuerknüppel verknüpft. Bei dieser einfachen Ansteuervariante sollten in der Mittelstellung des Steuerknüppels beide Motoren in Ruhe verharren. Ist der Knüppel rechts oben, so bewegt sich der Roboter vorwärts, bei der Knüppelstellung links unten dagegen rückwärts. Ist der Steuerknüppel links oben bzw. rechts unten, dann dreht sich der Roboter nach links bzw. nach rechts. Stellt sich ein anderes Verhalten ein, so muss man ggf. die Orientierung (links/rechts) des jeweiligen Kanals ändern. Dazu sind bei vielen Fernsteuerungen zusätzliche Schalter vorgesehen. Die Orientierung kann man auch softwareseitig ändern, indem man z.B. für die Variable u den Aufruf `map(u,TMIN,TMAX,-255,+255)` in `map(u,TMIN,TMAX,+255,-255)` ändert (Vorzeichenwechsel).

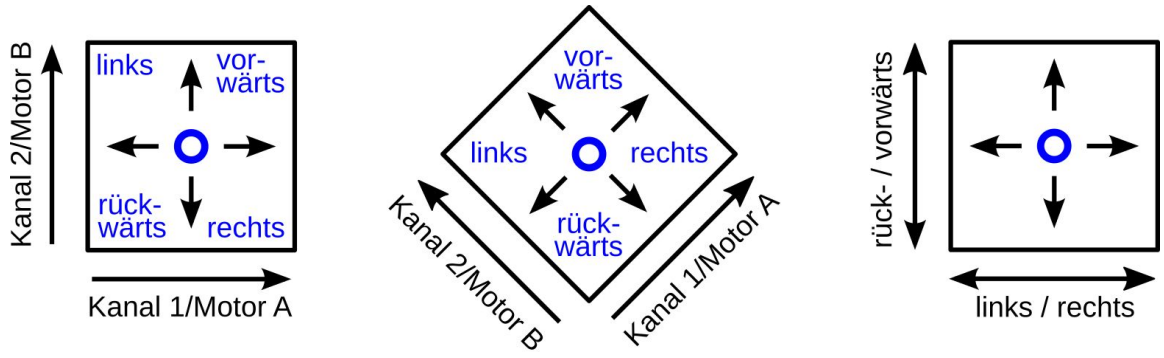


Abbildung eines Knüppels der Fernsteuerung auf die Motorsignale, einfache Variante (links), um 45° gedrehte Fernsteuerung (Mitte), korrigierte Variante (rechts)

Die zunächst realisierte einfache Ansteuerung der Motoren ist etwas unüblich. Dreht man dagegen den Sender der Fernsteuerung um 45° entgegen dem Uhrzeigersinn, so stellt sich eine viel intuitivere Bedienung ein. Eine Drehung entgegen dem Uhrzeigersinn (also in mathematisch positivem Drehsinn) mit dem Winkel α von einer Position (x,y) in die neue Position (x',y') lässt sich durch folgende Formel beschreiben [7]:

$$\begin{aligned} x' &= x\,\cos(\alpha) + y\,\sin(\alpha) \\ y' &= -x\,\sin(\alpha) + y\,\cos(\alpha) \end{aligned}$$

Die Winkelfunktionen Sinus und Kosinus haben bei einem Winkel von 45° den folgenden Funktionswert:

$$\sin(45°) = \cos(45°) \approx 0{,}7071 \approx 0{,}7.$$

In unserem Programm wird die durch (u,v) beschriebene Position in die neuen Variablen (a,b) überführt. Die gebrochenrationale Zahl 0,7=7/10 ist dabei als Dezimalbruch implementiert:

```
// Drehung um 45°
int a = ( 7*u + 7*v) / 10;
int b = (-7*u + 7*v) / 10;
```

Die Variablen `u` und `v` sind auf den Bereich von -255 bis +255 beschränkt. Mit der Umrechnung in die Variablen `a` und `b` kann dieser Zahlenbereich um bis zu 40% überschritten werden. Das ist bei der in diesem Werk vorgestellten Klasse `Motor` kein Problem, da automatisch eine Beschränkung auf den zulässigen Bereich von ±255 erfolgt. Bei Verwendung einer anderen Bibliothek zur Motoransteuerung muss ggf. noch eine passende Beschränkung erfolgen, z.B. mit der Arduino-Funktion `constrain`:

```
// Beschränkung Motorsignale
a = constrain(a,-255,255);
b = constrain(b,-255,255);
```

Anstelle der Variablen `u` und `v` setzen wir jetzt die Variablen `a` und `b` zur Motoransteuerung ein:

```
// Motoransteuerung
motor.write(a,b);
```

Mit dieser Umrechnung würde man bei der Vorwärtsstellung des Steuerknüppels auch vorwärts fahren, bei der Rückwärtsstellung rückwärts und so weiter.

Bisher wurde stillschweigend davon ausgegangen, dass der Fernsteuerempfänger die entsprechenden Impulse ausgibt. Das ist beispielsweise bei ausgeschaltetem Sender oder einer Unterbrechung zwischen Empfänger und Arduino-Board nicht der Fall. Zur Erkennung dieser Ausnahmesituation definieren wir die Konstante `TIMEOUT` derart, dass die Impulserfassung spätestens nach 50ms abgebrochen wird. (Das ist mehr als die doppelte Zyklenzeit der PWM.) Die Konstante `TMITTEL` gibt die mittlere Impulsdauer an, was der Mittelstellung des Steuerknüppels entspricht:

```
// Timeout und Mittelwert
const unsigned long TIMEOUT = 50000;
const int TMITTEL = (TMIN + TMAX)/2;
```

In der Hauptschleife `loop` würde man die Funktion `pulseIn` mit einem zusätzlichen dritten Argument aufrufen:

```
// Abfrage des Empfängers
long unsigned x = pulseIn(Kanal1,HIGH,TIMEOUT);
long unsigned y = pulseIn(Kanal2,HIGH,TIMEOUT);
```

Wird in der durch die Konstante `TIMOUT` vorgegebenen Zeit kein HIGH-Impuls erkannt, so gibt die Funktion `pulseIn` den Wert Null zurück. In diesem Fall setzen wir die Variablen `x` und `y` auf die mittlere Impulsdauer:

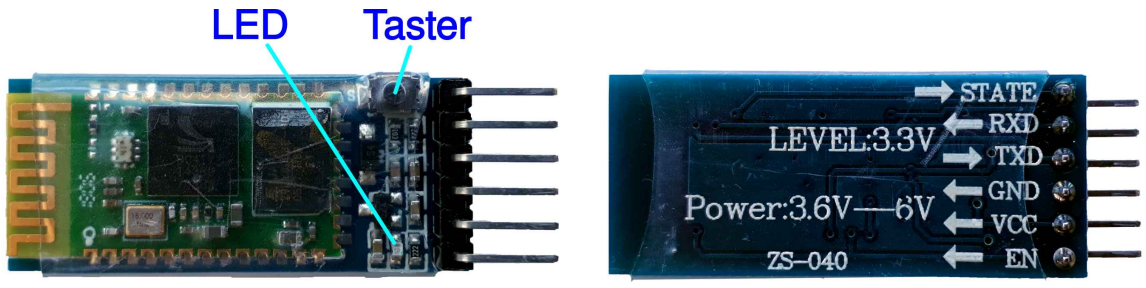
```
// Mittelwert setzen bei Timeout
if (x==0) x = TMITTEL;
if (y==0) y = TMITTEL;
```

Nach der entsprechenden Umrechnung auf die Motorsignale würden bei ausgeschalteter Fernsteuerung beide Motoren stillstehen.

9.3 Fernsteuerung über Bluetooth

9.3.1 Anschluss des Moduls HC-05

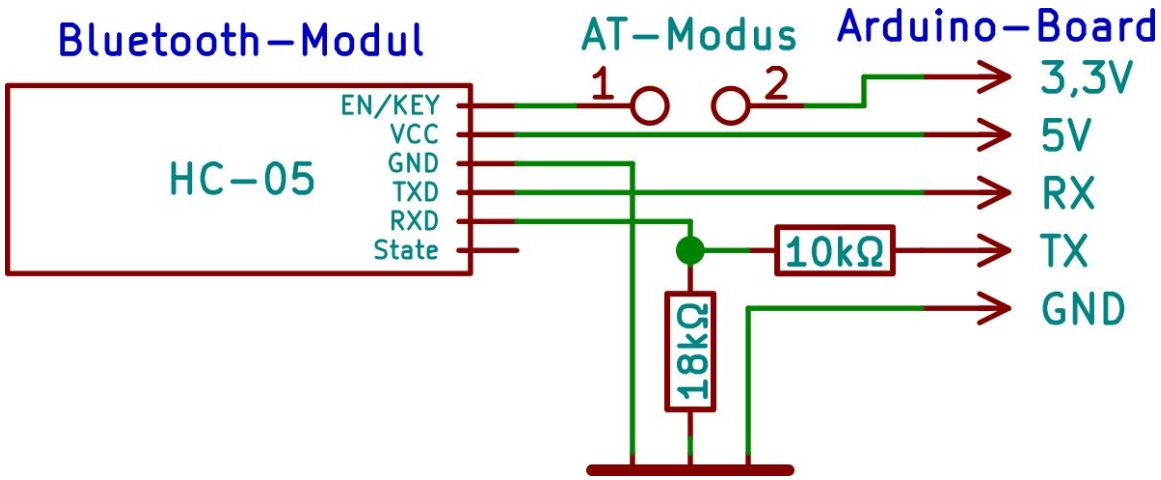
Für die Kommunikation über Bluetooth bieten sich die Module HC-05 und HC-06 an. Das Modul HC-05 kann man als Master bzw. Slave betreiben, das Modul HC-06 nur als Slave. Das bedeutet, dass zwar beide Module eine Bluetooth-Verbindung annehmen können, aber nur das Modul HC-05 eine solche Verbindung (als Master) auch initiieren kann. Da sich beide Module preislich kaum unterscheiden, wird hier ein HC-05 eingesetzt. Zur Steuerung des Roboters sollte das Modul HC-06 in ähnlicher Weise funktionieren.



Bluetooth Modul HC-05, Vorderseite (links), Rückseite (rechts)

Das Arduino-Board kommuniziert mit dem Modul HC-05 über eine serielle Schnittstelle mit den Signalen RX bzw. RXD (Receive Data) und TX bzw. TXD (Transmit Data). Die Signale sind dabei über Kreuz zu verbinden, d.h. der Sender der einen Baugruppe mit dem Empfänger der anderen. Bei den Standard-Boards Arduino Uno, Leonardo, Mega usw. ist jedoch eine Pegelanpassung erforderlich. Diese Boards werden mit 5V betrieben. Das

Bluetooth-Modul kann sowohl mit 3,3V als auch mit 5V betrieben werden, verwendet als Signalpegel aber 3,3V. (Bei den Boards Arduino Zero und Due ist keine Pegelanpassung nötig, da diese Boards bereits mit 3,3 V betrieben werden.) Für die Versorgungsspannung haben die Arduino-Boards ohnehin einen 3,3V-Anschluss. Der Ausgang TXD des Funkmoduls kann vom Eingang des Arduino-Boards korrekt verarbeitet werden, da der Signalpegel von knapp 3,3V deutlich oberhalb der halben Betriebsspannung des Boards liegt und damit klar als HIGH-Pegel zu erkennen ist. Der Ausgang TXD des Arduino-Boards könnte mit einem 5V-Signalpegel allerdings den Eingang RXD des Bluetooth-Models beschädigen. Die erforderlich Spannungsreduktion erfolgt über einen aus zwei Widerständen bestehenden Spannungsteiler.



Anschluss des Bluetooth-Funkmoduls HC-05

Der Anschluss KEY bzw. EN (Enable) bleibt im normalen Betrieb offen und wird nur zum Konfigurieren des Bluetooth-Modus auf HIGH-Pegel. Bei einem Betrieb des Moduls mit 5V ist die Verbindung mit dem 3,3V-Anschluss des Arduino-Boards nötig.

Für die serielle Kommunikation gibt es seitens der Arduino-Boards verschiedene Möglichkeiten:

1. *Nutzung der hardwareseitig vorgesehenen Standardschnittstelle:* Der Schaltkreis ATmega328 besitzt eine Funktionsgruppe USART (Universal Synchronous/Asynchronous Receiver Transmitter) für die serielle Kommunikation. Die Signale RX und TX liegen bei dem Arduino Uno Board auf den digitalen Kanälen 0 und 1. Softwareseitig ist für diese Schnitt-

stelle das Objekt `Serial` vorgesehen. Diese Schnittstelle wird aber auch zum Hochladen des kompilierten Programmcodes bzw. zur Ausgabe von Meldungen auf dem PC mit Hilfe des seriellen Monitors genutzt. Bei der Verwendung dieser Standardschnittstelle ist zum Hochladen des Programms das Funkmodul vom Arduino-Board zu trennen. Außerdem ist beim Betrieb des Funkmoduls eine Ausgabe über den seriellen Monitor nicht möglich.

2. *Nutzung zusätzlicher Hardware-Schnittstellen:* Manche Boards verfügen über mehrere serielle Schnittstellen. Das Arduino Leonardo Board hat eine weitere Schnittstelle, die softwareseitig über `Serial1` angesprochen werden kann. Die Standardschnittstelle `Serial` ist für die Kommunikation mit dem PC bzw. für den Anschluss externer USB-Geräte an das Board (z.B. Maus oder Keyboard) vorgesehen. Die Boards Arduino Mega und Due besitzen drei zusätzliche serielle Schnittstellen, die über die Objekte `Seriel1`, `Serial2` und `Serial3` ansprechbar sind (siehe Tabelle).
3. *Softwareseitige Emulation einer seriellen Schnittstelle:* In der Arduino-Umgebung kann man über die Bibliothek `SoftwareSerial` eine serielle Schnittstelle emulieren. Dafür kann man (mit einigen Einschränkung) normale digitale Kanäle verwenden.

Da die erste Variante mit Einschränkungen im Betrieb verbunden ist und die zweite Variante nur für bestimmte Boards funktioniert, wird zunächst die dritte Variante beschrieben. Für die Signalübertragung bieten sich die digitalen Kanäle 8 und 9 an, sofern diese nicht vom Arduino-Motor-Shield zum Bremsen genutzt werden.

Arduino Boards	`Serial`		`Serial1`		`Serial2`		`Serial3`	
	RX	TX	RX	TX	RX	TX	RX	TX
Uno	0	1						
Leonardo	(USB)		0	1				
Mega	0	1	19	18	17	16	15	14

Pins für die seriellen Schnittstellen

Wichtige Methoden zur softwareseitigen Abwicklung einer seriellen Übertragung sind in der nächsten Tabelle aufgeführt.

Wichtige Methoden von `Serial` bzw. `SoftwareSerial`

Methode, Aufruf	*Beschreibung*
`begin(baudrate)`	Eröffnet Übertragung mit vorgegebener Baudrate
`end()`	Beendet Übertragung
`print(daten)`	Ausgabe für zahlreiche Datentypen
`print(daten,Basis)`	Zahlenausgabe mit Wahl der Basis (BIN, OCT, DEC, HEX)
`println(...)`	Wie print, jedoch mit Zeilenumbruch
`write(daten,n)`	Sendet Datenfeld mit n Bytes
`flush()`	Wartet auf Abschluss des Schreibvorgangs
`read()`	Liest Byte vom Slave
`readBytes(daten,n)`	Liest Datenfeld mit n Bytes ein
`available()`	Anzahl der zum Lesen verfügbaren Bytes

Für die Überprüfung der Bluetooth-Verbindung bzw. für die Konfiguration des Moduls ist ein kleines Terminalprogramm hilfreich. Die Baudrate kann je nach Konfiguration oder Betriebsmoduls variieren wird wird daher zentral als Konstante definiert.

```
const long Baudrate = 9600;
```

Zur Emulation der seriellen Schnittstelle ist die Bibliothek `SoftwareSerial` einzubinden:

```
#include <SoftwareSerial.h>
```

Die für die Signale RX und TX verwendeten digitalen Kanäle definieren wir als Konstante:

```
const byte pinRx = 8;
const byte pinTx = 9;
```

Von der Klasse `SoftwareSerial` wird das Objekt `bt` (für Bluetooth) angelegt, wobei die für die serielle Übertragung verwendeten Pins übergeben werden:

```
SoftwareSerial bt(pinRx,pinTx);
```

In der Funktion `setup` ist einerseits die Baudrate für die serielle Übertragung zum Bluetooth-Modul einzustellen. Die empfangenen Signale sollen über den seriellen Monitor ausgegeben werden. Dazu ist auch für das System-Objekt `Serial` die Baudrate einzustellen:

```
void setup() {
    bt.begin(Baudrate);        // Bluetooth
    Serial.begin(Baudrate);    // Serieller Monitor
}
```

In der Hauptschleife wird zunächst überprüft, ob auf der für das Bluetooth-Modul verwendeten Schnittstelle Daten vefügbar sind. Ist das der Fall, so wird ein Zeichen eingelesen und auf dem seriellen Monitor ausgegeben. Umgekehrt werden Eingaben beim seriellen Monitor über Bluetooth ausgegeben:

```
void loop() {
  // Bluetooth -> Arduino
    if (bt.available()>0) {
        Serial.write(bt.read());
    }
    // Arduino -> Bluetooth
    if (Serial.available()>0) {
    bt.write(Serial.read());
    }
}
```

Funktionstest über Smartphone

Für den Test der Bluetooth-Verbindung bzw. später für die Steuerung des Roboters kam die Android-App „Arduino bluetooth controller“ zum Einsatz; es lassen sich aber auch zahlreiche andere Apps verwenden. Nach dem Start sucht die App zunächst nach Bluetooth-Geräten. Das Modul sollte unter seiner Bezeichnung HC-05 o.ä. sichtbar sein. Bei der ersten Verbindungsaufnahme des Moduls ist die PIN zu übergeben (Standardeinstellung: „1234“). Für Abfrage bzw. Änderung von Gerätenamen und PIN sei auf den nächsten Abschnitt verwiesen. Für den Verbindungsmodus stehen verschiedene Möglichkeiten zur Auswahl (Joystick, Schalter, Terminal). Über Terminal kann man die Bluetooth-Verbindung in beide Richtungen überprüfen.

Arduino Leonardo Board

Bei diesem Board bietet sich die Verwendung der hardwareseitig unterstützen seriellen Schnittstelle mit Pin 0 (RX) und 1 (TX) an. Diese Schnittstelle wird über `Serial1` angesprochen, der serielle Monitor über `Serial`. Zusätzlich wird noch gewartet, bis der serielle Monitor verfügbar ist. Die Bibliothek `SoftwareSerial` ist dabei nicht nötig.

```
void setup() {
  Serial1.begin(9600);  // Bluetooth
  Serial.begin(9600);     // Serieller Monitor
  while (!Serial) {}
}
```

Die empfangenen Daten werden in der Hauptschleife auf dem seriellen Monitor ausgegeben:

```
void loop() {
  if (Serial1.available()>0) {
    Serial.write(Serial1.read());
  }
}
```

9.3.2 Konfiguration bzw. Konfigurationsabfrage

Dieser Abschnitt befasst sich mit der Konfiguration (bzw. dem Auslesen der eingestellten Konfiguration) des Moduls. Für die Steuerung des Roboters per Smartphone ist die Kenntnis dieses Abschnitts nicht unbedingt erforderlich.

Die Konfiguration des Moduls kann man im sogenannten AT-Modus auslesen bzw. ändern. Die Steuerkommands (AT-Kommandos) kann man über eine serielle Schnittstelle senden. Um in den AT-Modus zu gelangen, muss der Pin KEY (bzw. EN) auf HIGH (3,3V) gesetzt werden. Dazu ist die in entsprechende Verbindung zu überbrücken oder der Taster auf dem Modul zu drücken (siehe vorangegangener Abschnitt). Hinsichtlich der Baudrate gibt es dabei zwei Möglichkeiten:

1. Der Pin KEY bzw. EN liegt schon beim oder vor dem Einschalten auf HIGH-Pegel. In diesem Fall erfolgt die Übertragung mit 38400 Baud.

2. Der Pin KEY bzw. EN wird deutlich nach dem Einschalten auf HIGH gelegt. Dann erfolgt die Übertragung mit der eingestellten Standardbaudrate (hier: 9600 Baud).

Im ersten Fall wäre bei dem im vorangegangenen Abschnitt verwendeten Terminalprogramm die Baudrate anzupassen:

```
const long Baudrate = 38400;
```

Für die Benutzung ist im seriellen Monitor neben der o.g. Datenrate von 38400 Baud zusätzlich die Einstellung „Sowohl NL als auch CR“ auszuwählen. Die Konfigurationskommandos fangen alle mit der Sequence „AT“ an. Die wichtigsten AT-Kommandos sind in der nachfolgenden Tabelle aufgeführt [8]. Kommandos, die mit einem Fragezeichen enden, dienen der Abfrage bestimmter Parameter. Durch Weglassen des Fragezeichens kann man die entsprechenden Parameter auch neu setzen.

Wichtige AT-Kommandos zur Konfiguration des Bluetooth-Moduls

AT-Kommando	*Beschreibung*
`AT`	Testkommando
`AT+ORGL`	Rücksetzen auf Standardeinstellungen
`AT+VERSION?`	Ausgabe der Programmversion
`AT+UART?`	Abfrage der seriellen Parameter (Baudrate, …)
`AT+NAME?`	Abfrage des Modulnames
`AT+PSWD?`	Abfrage der PIN
`AT+ADDR?`	Abfrage der Adresse
`AT+ROLE?`	Rolle in der Komunikation (Slave: 0, Master: 1)
`AT+CMODE?`	Abfrage Verbindungsmodus (feste Adresse: 0, beliebige Adresse: 1)
`AT+BIND`	Adresse für Partnermodul festlegen
`AT+RMAAD`	Bindung an Partnermodul aufheben

Ist das Bluetooth-Modul im AT-Modus, dann blinkt die LED etwa im Zwei-Sekunden-Takt. Jetzt kann man mit dem Terminalprogramm aus dem vorangegangenen Abschnitt die Konfiguration abfragen bzw. ändern. Über den seriellen Monitor gibt man die Kommandos ein und kann die Antwort des

Systems ansehen. Die nächste Tabelle gibt die Ein- und Ausgabe des verwendeten Blutooth-Moduls an. Mit dem Kommando `AT` überprüft man zunächst die Verbindung zum Bluetooth-Modul. Im AT-Modus erhält man die Antwort `OK`. Anschließend kann man mit `AT+VERSION?` die Versionsnummer der Firmware abfragen bzw. mit `AT+UART?` die Einstellungen zur seriellen Übertragung auslesen und ggf. anpassen. Typisch sind die angezeigten 9600 Baud. Die zwei weiteren Parameter stehen für Stop- sowie Paritätsbits und sollten nicht verändert werden. Die Rückmeldung „0“ auf die Anfrage `AT+ROLE?` bestätigt, dass sich das Moduls bereits im Slave-Modus befindet. Sollte das Moduls bereits als Master konfiguriert sein (Rückmeldung „1“), kann man den Slave-Moduls über `AT+ROLE=0` einstellen. Die Abfrage des Namen mit `AT+NAME?` kann hilfreich sein, um das Gerät bei der Verbindung mit dem Smartphone zu identifizieren. Das mit `AT+PSWD?` abgefragte Passwort (hier: 1234) ist die Standardeinstellung. Für die Kopplung mit dem Master-Modul liest man mit `AT+ADDR?` die Adresse des Moduls aus.

Konfigurationsabfrage des Slave-Moduls

Eingabe	*Ausgabe*
`AT`	`OK`
`AT+VERSION?`	`+VERSION:2.0-20100601`
`AT+UART?`	`+UART:9600,0,0`
`AT+ROLE?`	`+ROLE:0`
`AT+NAME?`	`+NAME:DSD TECH HC-05`
`AT+PSWD?`	`+PSWD:1234`
`AT+ADDR?`	`+ADDR:14:3:60f85`

9.3.3 Motoransteuerung über Smartphone

In diesem Abschnitt wird beschrieben, wie der mobile Roboter über die Bluetooth-Verbindung mit einem Smartphone gesteuert wird. Dazu sind zunächst die benötigten Bibliotheken einzubinden:

```
#include<SoftwareSerial.h>
#include <Motor.h>
```

Zur Ansteuerung der Motoren instanziieren wir ein Objekt der Klasse `Motor`:

```
Motor motor;
```

Für die serielle Kommunikation mit dem Bluetooth-Modul legen wir das Objekt `bt` an:

```
const byte pinRX = 8;
const byte pinTX = 9;
SoftwareSerial bt(pinRX,pinTX);
```

In der Funktion `setup` erfolgt die Initialisierung für die Okjekte `bt` und `motor` (Baudrate und ggf. Pin-Konfiguration beim Motor):

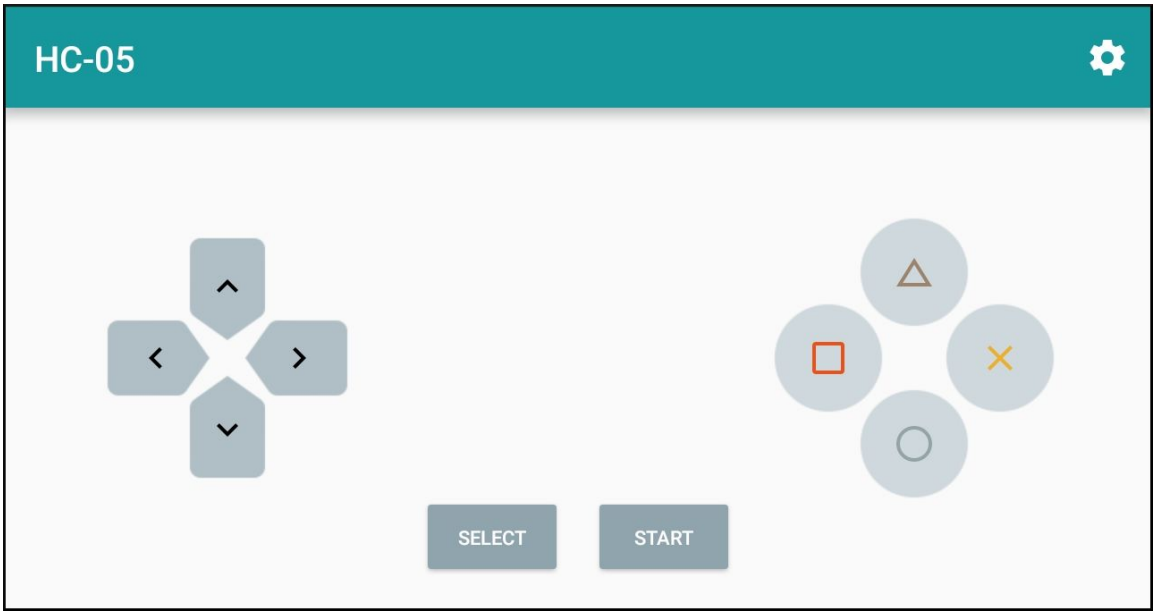
```
void setup()
{
  bt.begin(9600);
  motor.begin();
}
```

Die Buchstaben `o`, `u`, `l` und `r` sind für die Vorwärts, Rückwärts-, Links- bzw. Rechtsbewegung vorgesehen. Bei jedem anderen Zeichen wird der Roboter angehalten.

```
void loop() {
    if (bt.available()>0) {
        char wert = bt.read();
        switch (wert) {
            case 'o': // vorwärts
                motor.forward();
                break;
            case 'u': // rückwärts
                motor.backward();
                break;
            case 'l': // links
                motor.left();
                break;
            case 'r': // rechts
                motor.right();
                break;
            default:  // stop
                motor.stop();
        }
    }
}
```

Steuerung über Smartphone

Für die Ansteuerung kann man die bereits angesprochene Android-App „Arduino bluetooth controller“ nutzen. Nach der Auswahl des Bluetooth-Gerätes wählt man den Modus „Joystick“ aus. Die Abbildung zeigt einen Screenshot. Die im o.g. Programm für die Auswahl der Fahrtrichtung verwendeten Buchstaben sind entsprechend zu konfigurieren (Zahnradsymbol rechts oben).



Screenshot der Android-App „Arduino bluetooth controller“ im Joystick-Modus

9.3.4 Fernsteuerung über zweites Arduino-Board

Dieser Abschnitt befasst sich mit der Steuerung des mobilen Roboters über ein weiteres Arduino-Board mit dem Bluetooth-Modus HC-05. Wir gehen davon aus, dass die Signalleitungen TX und RX mit der Pegelwandlung mit den Arduino-Kanälen 8 und 9 verbunden sind. Zunächst ist das Bluetooth-Modul zu konfigurieren. Dabei greifen wir auf das beschriebene Terminalprogramm (mit Baudrate auf 38400 Baud) zurück und versetzen das Modul in den AT-Modus.

Im seriellen Monitor überprüfen wir mit dem Kommando `AT`, ob das Modul im Konfigurationsmodus ist. Nach Überprüfung der Baudrate für den normal

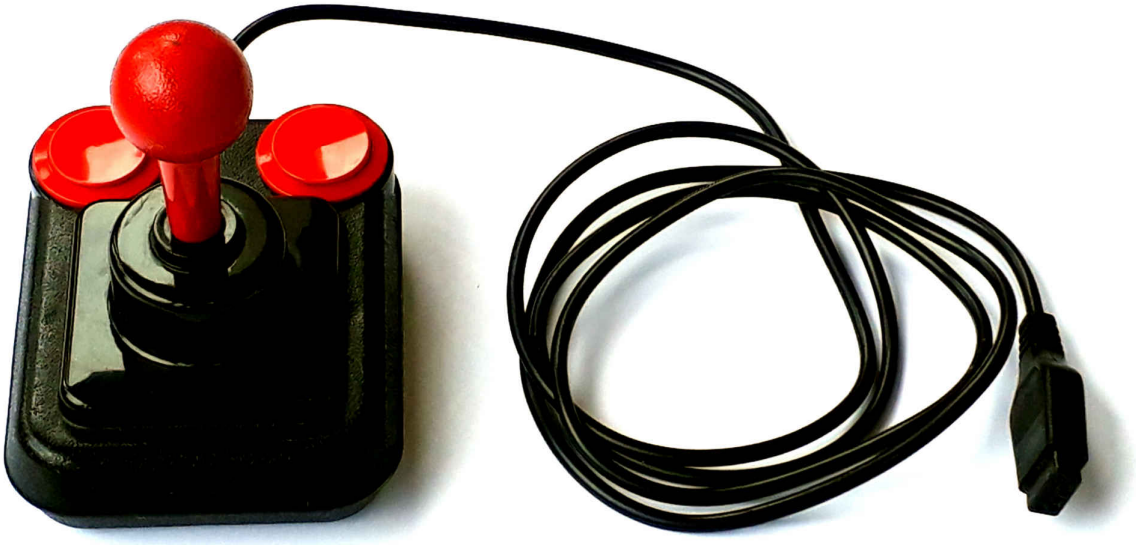
Datenaustausch und der PIN wird das Modul mit `AT+ROLE=1` als Master konfiguriert. Mit `AT+CMODE=0` (Standardeinstellung ist „1“) sorgt man dafür, dass der Verbindungsaufbau nur noch zum vorgesehenen Partnermodul erfolgt. Über das AT-Kommando `AT+BIND` mit der vom Slave-Modul vorher ausgelesenen Adresse erfolgt die Zuordnung zu diesem Slave-Modul. Wichtig: Bei der Adresse sind die Doppelpunkte durch Kommata zu ersetzen. Damit ist die Konfiguration des Master-Moduls abgeschlossen und wir können den AT-Modus verlassen.

Konfiguration des Master-Moduls

Eingabe	*Ausgabe*
`AT`	`OK`
`AT+UART?`	`+UART:9600,0,0`
`AT+PSWD?`	`+PSWD:1234`
`AT+ROLE=1`	`OK`
`AT+CMODE=0`	`OK`
`AT+BIND=14,3,60f85`	`OK`

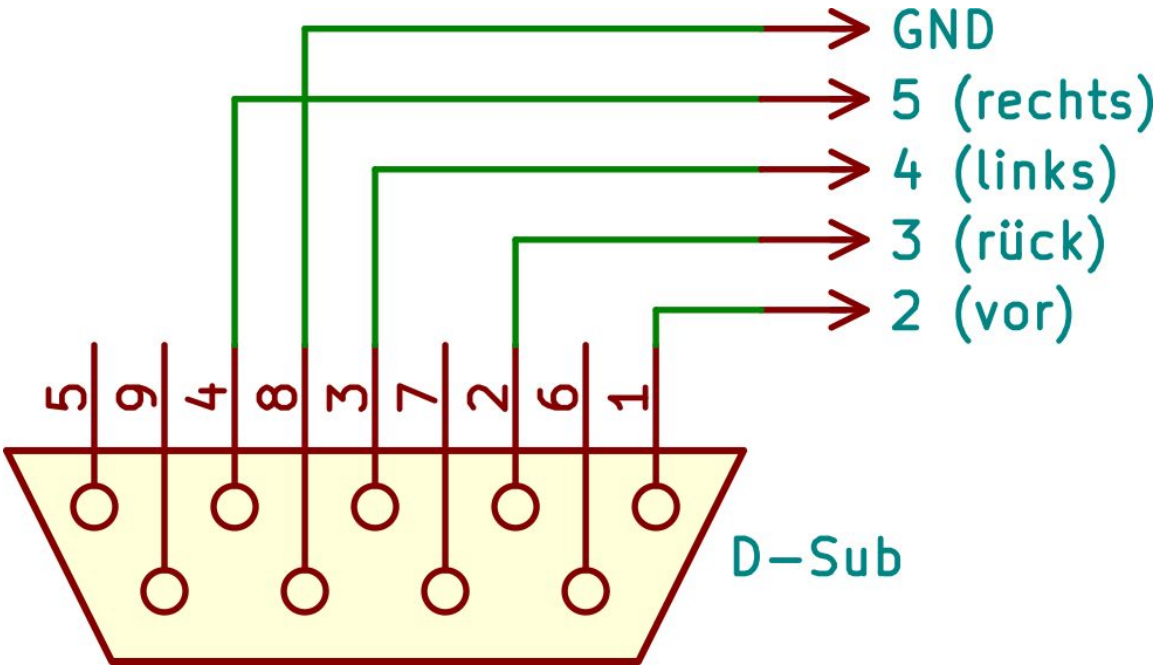
Für einen Test im normalen Datenmodus bietet sich erneut das Terminalprogramm an, jetzt wieder mit 9600 Baud. Im seriellen Monitor ist neben der Baudrate auch „Kein Zeilenende“ einzustellen. Durch Eingabe der Buchstaben `o`, `u`, `l` oder `r` einem Klick auf Senden kann man den mit dem Slave-Modul ausgestattenten mobilen Roboter steuern.

Die Fernsteuerung des mobilen Roboters wird über einen digitalen Joystick, wie sie in den 1980er Jahren bei Heimcomputer überlich waren, vorgenommen.



Digitaler Joystick

Der Abschluss erfolgt über einen 9-poligen D-Sub Steckverbinder:



Anschluss ds digitalen Joysticks an das zweite Arduino-Board

Für die Fernsteuerung des Roboters über den Joystick wird jetzt ein separates Programm erstellt. Zur Kommunikation mit dem Bluetooth-Modul wird wieder das Objekt `bt` angelegt:

```
#include <SoftwareSerial.h>
const byte pinRx = 8;
const byte pinTx = 9;
SoftwareSerial bt(pinRx,pinTx);
```

Der Joystick hat vier Richtungstasten, für welche vier digitale Kanäle benötigt. Die verwendeten Pins werden in einem Feld angelegt:

```
const byte pinTaster[] = {2, 3, 4, 5};
```

Die hier verwendeten digitalen Kanäle 2, 3, 4, 5 sind den Bewegungsrichtungen vorwärts, rückwärts, links und rechts zugeordnet. In der Funktion `setup` wird die serielle Verbindung zum Bluetooth-Modul mit 9600 Baud initialisiert. Die Taster des Joysticks sind gegen Masse zu schalten. Die digitalen Kanäle werden daher als Eingänge mit Pull-Up-Widerstand konfiguriert:

```
void setup() {
    bt.begin(9600);
    for (byte i=0; i<=3; i++)
        pinMode (pinTaster[i],INPUT_PULLUP);
}
```

Die Abfrage der Taster erfolgt in naheliegender Weise mit der folgenden Funktion:

```
int AbfrageTaster () {
    int wert = 0;
    for (byte i=0; i<=3; i++)
        wert = (wert<<1) + digitalRead(pinTaster[i]);
    return wert;
}
```

In der Hauptschleife werden der Taster zyklisch abgefragt. Wird eine der Hauptrichtungen erkannt, sendet das Programm den entsprechenden Buchstabencode:

```
void loop() {
    int wert = AbfrageTaster();
    switch (wert) {
        // Hauptrichtungen
        case  7:    // vorwärts
            bt.print('o');  break;
        case 11:    // rückwärts
            bt.print('u');  break;
        case 13:    // links
            bt.print('l');  break;
        case 14:    // rechts
            bt.print('r');  break;
        default:    // stop
            bt.print(' ');
    }
    delay(100);
}
```

Das Programm kann man leicht hinsichtlich der kombinierten Richtungen (z.B. rechts vorwärts) erweitern. Die entsprechenden Buchstabencodes sind dann natürlich auch im Programm des mobilen Roboters zu berücksichtigen.

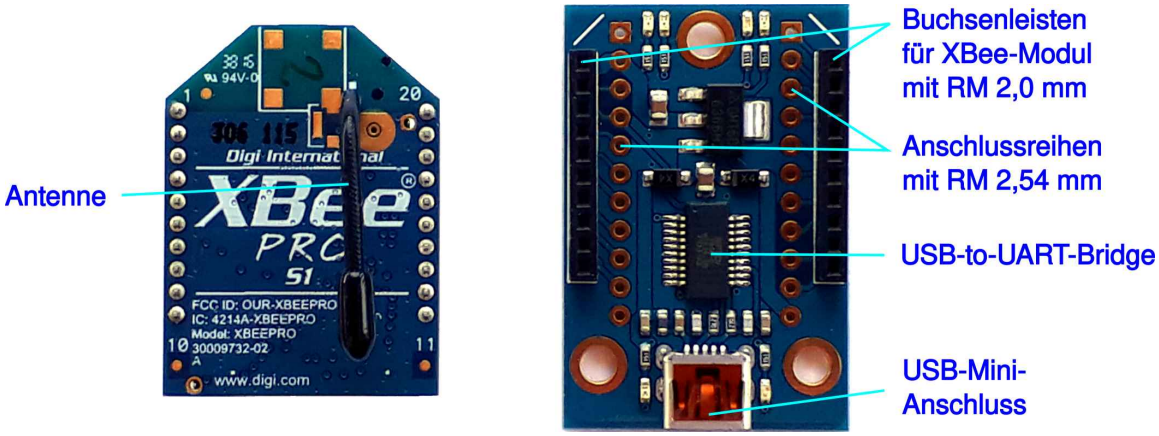
9.4 Fernsteuerung über ZigBee bzw. XBee

9.4.1 Konfiguration der Funkmodule

ZigBee ist ein Standard für drahtlose Kommunikation mit niedrigem Datenaufkommen. Dieser Funkstandard wird von XBee-Modulen der Firma Digi International unterstützt. XBee-Module nach dem Kommunikationsprotokoll IEEE 802.15.4 (Serie 1, kurz S1) sind für Punkt-zu-Punkt-Verbindungen oder für sternförmige Verbindungen geeignet, die Module der Serie 2 (S2) erlauben den Aufbau komplexerer Netzwerktopologien. Für die Kommunikation mit dem mobilen Roboter setzen wir Module der Serie 1 ein. Das Standard-XBee-Modul hat eine Sendeleistung von 1mW mit einer Reichweite von 30m (Innen) bzw. 100m (Außen). XBee-PRO-Module haben dagegen eine Sendeleistung von 60mW und Reichweite von 100m (innen) bzw. 1,5km (außen).

Die nächste Abbbildung (links) zeigt ein XBee PRO-Modul der Serie 1. Die Anschlüsse sind als zwei Stiftleisten mit je 10 Pins angeordnet. Die Stiftleisten

sind im Rastermaß (RM) 2mm ausgeführt und daher leider nicht kompatikel mit gängigen Lochrastplatten, bei denen RM 2,54mm Verwendung findet. Für die Konfiguration über den PC gibt es Adapter-Module, die über einen USB-Anschluss mit Spannungsregler (zur Anpassung von 5V auf 3,3V) und einem Schaltkreis zur seriellen Kommunikation (USB-to-UART-Bridge) verfügen. Diese Module werden meist unter der Bezeichnung USB-Explorer angeboten (Abbildung (rechts)). Bei diesen Adapter sind in der Regel auch Anschlussreihen für das Standardrastermaß von 2,54mm vorgesehen.



XBee-PRO Modul (links), USB-Explorer (rechts)

Zur Konfiguration der Module stellt der Hersteller Digi International das Programm XCTU für Windows, MaxOS X und Linux zur Verfügung [9]. Das Modul wird über den USB-Explorer mit dem PC verbunden. Die Konfiguration des Moduls erfolgt über verschiedene Register bzw. Parameter:

Wichtige Register der XBee-Module

Register	*Bedeutung*
CH	Kanal
ID	Netzwerk-ID
DH	Zieladresse, HIGH-Teil
DL	Zieladresse, LOW-Teil
MY	Adresse des Moduls
SH	Seriennummer, HIGH-Teil
SL	Seriennummer, LOW-Teil
BD	Baudrate (0 ≙ 1200 Baud, …, 7 ≙ 115200 Baud)

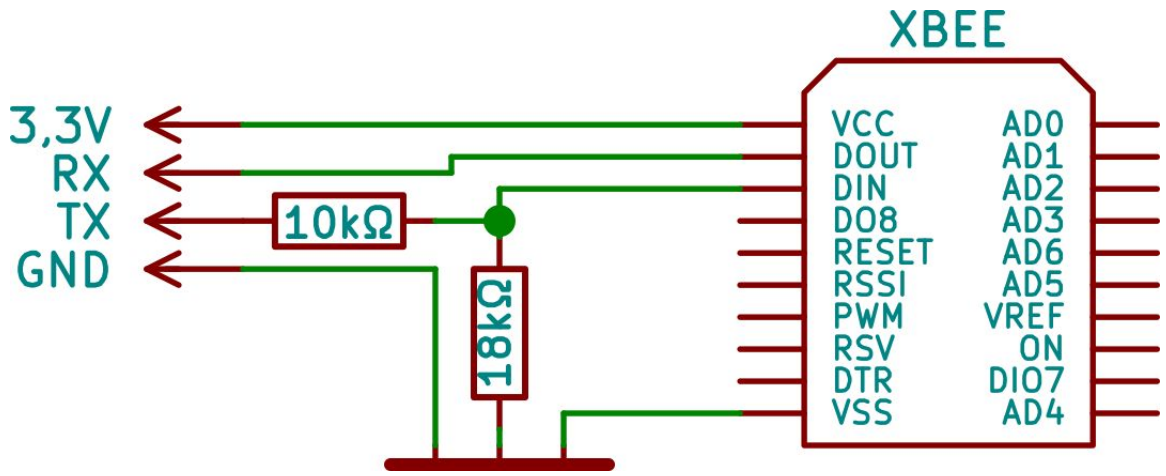
Das Register CH für den Kanal sollte den Wert 0CH aufweisen. Zusätzlich müssen die Modul im selben Netzwerk agieren, also die gleiche ID aufweisen (Grundeinstellung: 3332H). Zur gegenseitigen Verkopplung der Module gibt es verschiedene Möglichkeiten. Die einfachste Variante besteht darin, den HIGH-Teil DH der Zieladreese auf 0000H zu setzen. Die Register MY und DL setzt man dann derart, dass die eigene Adresse mit der Zieladresse des anderen Moduls übereinstimmt. Dabei muss das Register DL Werte kleiner als FFFFH annehmen. (Der WertFFFFH ist für sternförmige Verbindungen reserviert.) Die nächste Tabelle zeigt ein mögliches Schema für zwei Module (Roboter und Fern-steuerung). In dieser Konfiguration wird zwischen den Modulen eine gleichberechtigte Punkt-zu-Punkt-Verbindung aufgebaut, d.h. es liegt keine Master-Slave-Struktur vor. Zusätzlich wird die Baudrate 9600 eingestellt.

Beispielkonfiguration zweier XBee-Module

Register	*Modul 1*	*Modul 2*
CH	0CH	0CH
ID	1000H	1000H
DH	0000H	1002H
DL	1002H	1001H
MY	1001H	1002H
BD	3 ≙ 9600 Baud	3 ≙ 9600 Baud

9.4.2 Anschluss und Betrieb der Fernsteuerung

Die Verdrahtung der XBee-Module ist in der nächsten Abbildung dargestellt. Die Beschaltung kann man über eine 10-polige Buchsenleiste mit RM 2,0 mm vornehmen (z.B. Conrad Electronic SE, Artikelnr. 001311370). Für die Signale RX und TX bieten sich (wie schon im Fall der Bluetooth-Module) die digitalen Kanäle 8 und 9 an. Für den Roboter kann man das gleiche Programm wie beim Blueooth-Modul eingesetzen.



Anschluss des XBee-Moduls

Das zweite XBee-Modul wird in der gleichen Weise mit dem zur Fernsteuerung vorgesehenen Arduino-Board verbunden. Zusätzlich ist der Joystick entsprechend anzuschließen. Das zugehörige Arduino-Programm kann ebenfalls unverändert übernommen werden.

9.5 Quellenangabe

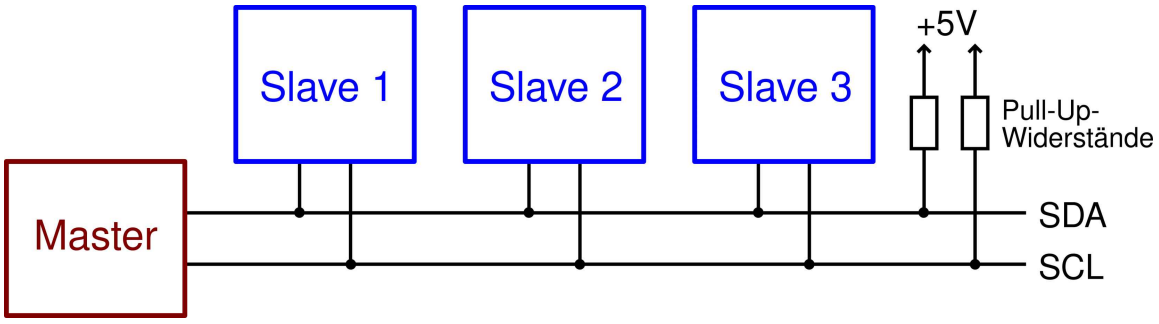
1. Schmidt, M.: Arduino. Ein schneller Einstieg in die Microcontroller-Entwicklung. dpunkt.verlag, Heidelberg, 2. Auflage, 2015.
2. Arduino-IRremote. *Infrared remote library for Arduino: send and receive infrared signals with multiple protocols*. URL: https://github.com/z3t0/Arduino-IRremote
3. Arduino – Robot. URL: https://www.arduino.cc/en/Main/Robot
4. Reely: Fernsteuerung „HT-4" 2,4 GHz, Bedienungsanleitung.
5. Schlechtriem, G.: Zeit, dass sich was bewegt. c't Hacks 1/2014, S. 64-70.

6. Schlechtriem, G.: Ansteuerung von Modellbauservos. Kindle Edition, 2014/2015, ASIN: B00ERC3INI.
7. Bronstein, I. N.; K. A. Semendjajew; G. Musiol; H. Mühlig: Taschenbuch der Mathematik. Verlag Harri Deutsch, 8. Auflage, 2012.
8. ITead Studio: *HC-05 Bluetooth to Serial Port Module*. 2010.
9. XCTU. URL: https://www.digi.com/xctu

10 I²C-Bus

10.1 Anschluss und Adressermittlung

Der I²C-Bus ist ein serieller Datenbus, der auch unter der Bezeichnung TWI (Two-Wire-Interface) bekannt ist [5]. Die Kommunikation erfolgt über die zwei Signalleitungen SDA (Serial Data) und SCL (Serial Clock). Zusammen mit Masse (GND) und Versorgungsspannung (VCC oder VDD) benötigt man somit vier Leitungen. Beim I²C-Bus wird der Zugriff auf einzelne Geräte über eine Master-Slave-Struktur gesteuert. Eine Übertragung wird dabei von einem Master ausgelöst. In der Regel wird das Arduino-Board als Master fungieren, womit verschiedene Peripheriebausteine, -baugruppen bzw. -module angesprochen werden. Der I²C-Bus kann auch zur Kommunikation zwischen mehreren Arduino-Boards verwendet werden.



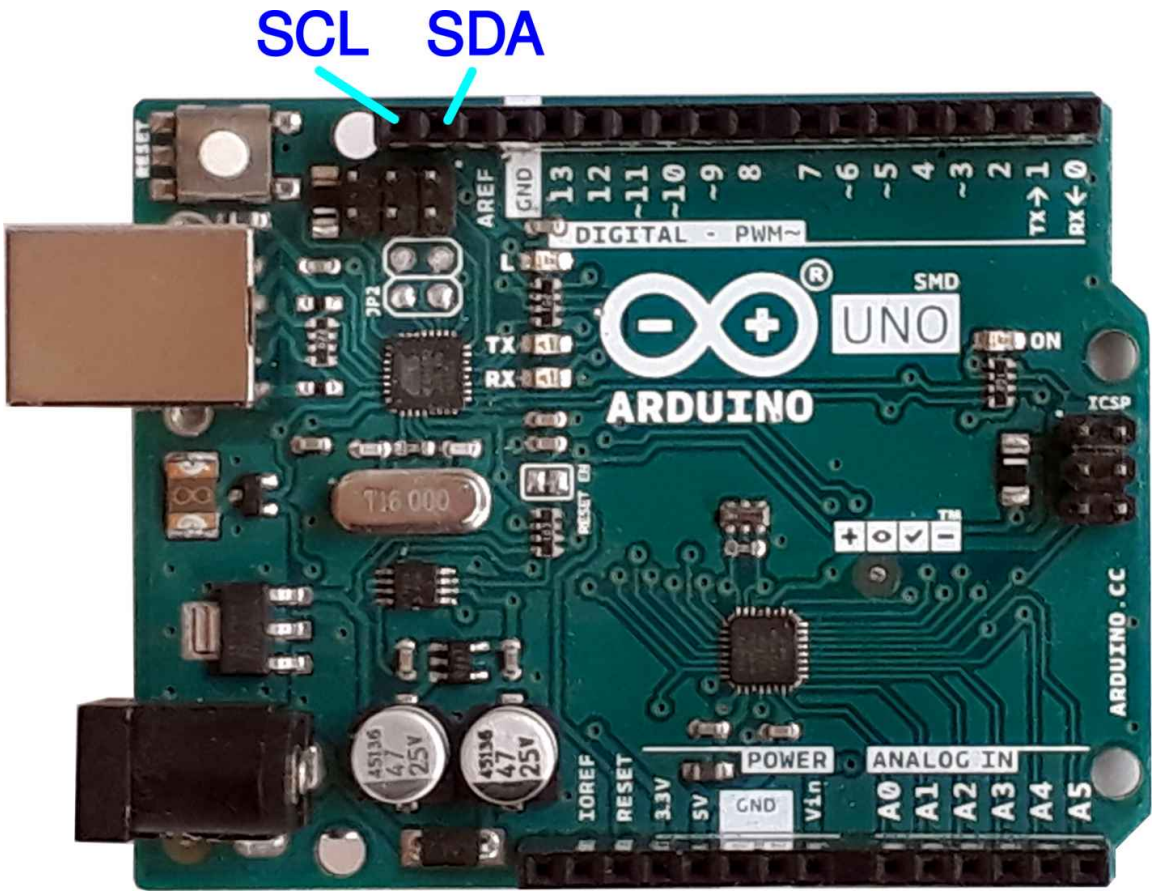
Master-Slave-Struktur des I²C-Busses

Den gleichzeitigen Anschluss mehrerer Ausgänge auf der gleichen Signalleitung ermöglicht man dadurch, dass man Open-Kollektor- oder Open-Drain-Ausgänge verwendet. Daher sind Pull-Up-Widerstände erfolderlich, die prinzipiell aber auch vom Mikrocontroller intern bereitgestellt werden können. Bei Arduino Uno, Leonardo und Mega 2560 beträgt die Versorgungsspannung 5V. (Es gibt aber auch viele I²C-Schaltkreise, die für 3,3V vorgesehen sind.) In der folgenden Tabelle sind die für die Signalleitungen des I²C-Busses verwendeten Pins der o.g. Arduino-Boards aufgeführt:

Pins für I²C-Bus [1]

Arduino Board	*SDA*	*SCL*
Uno	A4	A5
Leonardo	2	3
Mega 2560	20	21

Für die Verdrahtung des I²C-Busses ist die in der Tabelle angegebene Pin-Belegung von untergeordneter Bedeutung, da die Signale SCL und SDA auf den gängigen Arduino-Boards separat herausgeführt sind. Man muss jedoch beachten, dass man die für den I²C-Bus verwendeten Kanäle nicht gleichzeitig anderweitig benutzen kann. Die beim Leonardo Board verwendete Pin-Belegung ist problematisch, da Pin 3 bereits vom Arduino-Motor-Shield fest belegt wird. Pin 2 ist zwar für das LCD-Modul vorgesehen, könnte dort aber auch einem anderen digitalen Kanal zugeordnet werden.



Anschlüsse des I²C-Busses auf dem Arduino Uno Board R3

Die Auswahl der am I²C-Bus angeschlossenen Geräte ist über eine 7-Bit- oder eine 10-Bit-Adressierung möglich. Hier kommt die 7-Bit-Adressierung zum Einsatz. Dabei steht theoretisch der Adressbereich 0,…,127 zur Verfügung. Die untersten 8 Adressen sind für Spezialzwecke reserviert, die obersten 8 Adressen werden für den Übergang zu einer 10-Bit-Adressierung genutzt. Damit ist der nutzbare Adressraum auf den Bereich 8,…,119 mit 112 Adressen beschränkt.

Die an den I²C-Bus anschließbaren Peripheriegeräte haben meistens eine feste (vom Hersteller vorgegebene) Adresse oder zumindest eine vorgegebene Grundadresse, die über Jumper oder Lötbrücken in einem gewissen Rahmen modifiziert werden kann. Ist die verwendete Adresse nicht bekannt, so kann sie softwareseitig ermittelt werden. Mit dem nachfolgend beschriebenen Programm wird der verfügbare Adressraum

durchsucht. Zugleich kann man damit die korrekte Anbindung der jeweiligen Baugruppe an den I²C-Bus überprüfen.

Die Ansteuerung des I²C-Busses erfolgt in der Arduino-Umgebung über die Bibliothek `Wire`:

```
#include <Wire.h>
```

Damit wird das Objekt `Wire` angelegt. In der Funktion `setup` wird zunächst dieses Objekt initialisiert. Die Ausgabe der belegten Adressen erfolgt über den seriellen Monitor (**Werkzeuge > Serieller Monitor**). Die zugehörige serielle Verbindung muss ebenfalls softwareseitig initialisiert werden:

```
void setup() {
    Wire.begin();
    Serial.begin(9600);
    while (!Serial) {}; // Warten auf serielle Verbindung
}
```

Im Hauptprogramm wird für die Adressen 8,…,119 eine Datenübertragung initiiert. Bei erfolgreichem Verbindungsaufbau wird die jeweilige Adresse angezeigt. Die Abfrage der am I²C-Bus belegten Adressen wird nach 10s wiederholt.

```
void loop() {
    byte status, adr;
    Serial.println("Suche I2C-Adressen...");
    for(adr = 8; adr < 120; adr++ ) {
        Wire.beginTransmission(adr);
        status = Wire.endTransmission();
        if (status == 0)
        {
            Serial.print("I2C-Adresse (HEX): ");
            Serial.println(adr,HEX);
        }
    }
    delay(10000);
}
```

In der Bibliothek Wire ist die Klasse `TwoWire` implementiert. Von dieser Klasse wird durch die Anweisung `extern` eine globale Instanz als Objekt

`Wire` angelegt. Die wichtigsten Methoden sind in der nächsten Tabelle aufgeführt.

Wichtige Methoden der Klasse `TwoWire` bzw. des Objekts `Wire` [1]

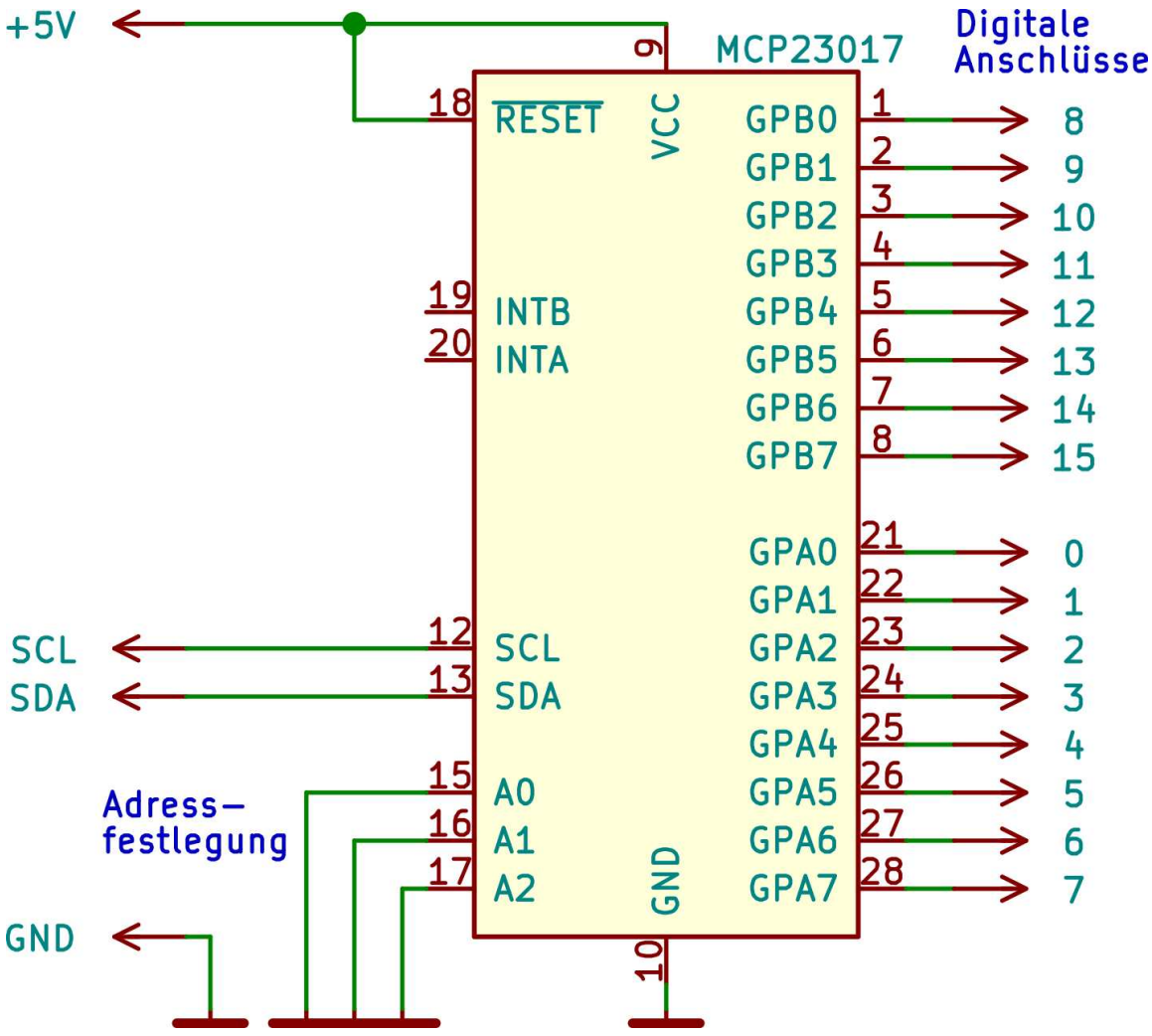
Methode, Aufruf	*Beschreibung*
`begin()`	Initialisiert Bibliothek
`begin(adr)`	Eröffnet zusätzlich Übertragung zu Adresse adr
`beginTransmission(adr)`	Eröffnet Übertragung zu Adresse adr
`endTransmission()`	Beendet Übertragung
`write(daten)`	Sendet Byte oder Zeichenkette an Slave
`write(daten,n)`	Sendet Datenfeld mit n Bytes
`requestFrom(adr,n)`	Fordert n Bytes von Adresse adr an
`read()`	Liest Byte vom Slave
`available()`	Anzahl der zum Lesen verfügbaren Bytes

10.2 Bereitstellung weiterer digitaler Anschlüsse mit MCP23017

Wenn man mehr digitale Anschlüsse benötigt, als von den Standard-Boards (Arduino Uno, Leonardo, …) bereitgestellt werden, aber nicht zu den größeren Boards (Arduino Mega, Mega 2560) wechseln möchte, bietet sich der Einsatz eines sog. *I/O-Port-Expanders* an. Ein gängiger, für diesen Verwendungszweck geeigneter Schaltkreis ist der MCP 23017-E/SP [4]. Dieser Schaltkreis stellt 16 digitale Kanäle bereit. Die Ansteuerung erfolgt über den I²C-Bus. Der IC MCP23S17 der gleichen Schaltkreisfamilie ist dagegen für die Kommunikation über SPI vorgesehen.

Für den Betrieb des Schaltkreises MCP23017 werden Masse, Versorgungsspannung und die zwei I²C-Signalleitungen (SCL, SDA) benötigt. Der RESET-Anschluss ist mit der Versorgungsspannung zu verbinden. Über die Anschlüsse A0, A1, A2 lassen sich 3Bit der I²C-Adresse einstellen. Grund-

adresse des Schaltkreises ist 20H. Das ist auch die im Schaltbild eingestellte Adresse (A0, A1, A2 auf LOW). Damit stehen 2^3=8 verschiedene Adressen zur Verfügung (20H…27H), so dass man maximal 8 dieser Schaltkreis an einem I²C-Bus betreiben kann. Für den Nutzer ist ein 16-Bit GPIO-Modul (*general purpose input/output*) verfügbar, welches in zwei 8-Bit-Kanäle (GPIOA, GPIOB) aufgeteilt ist.

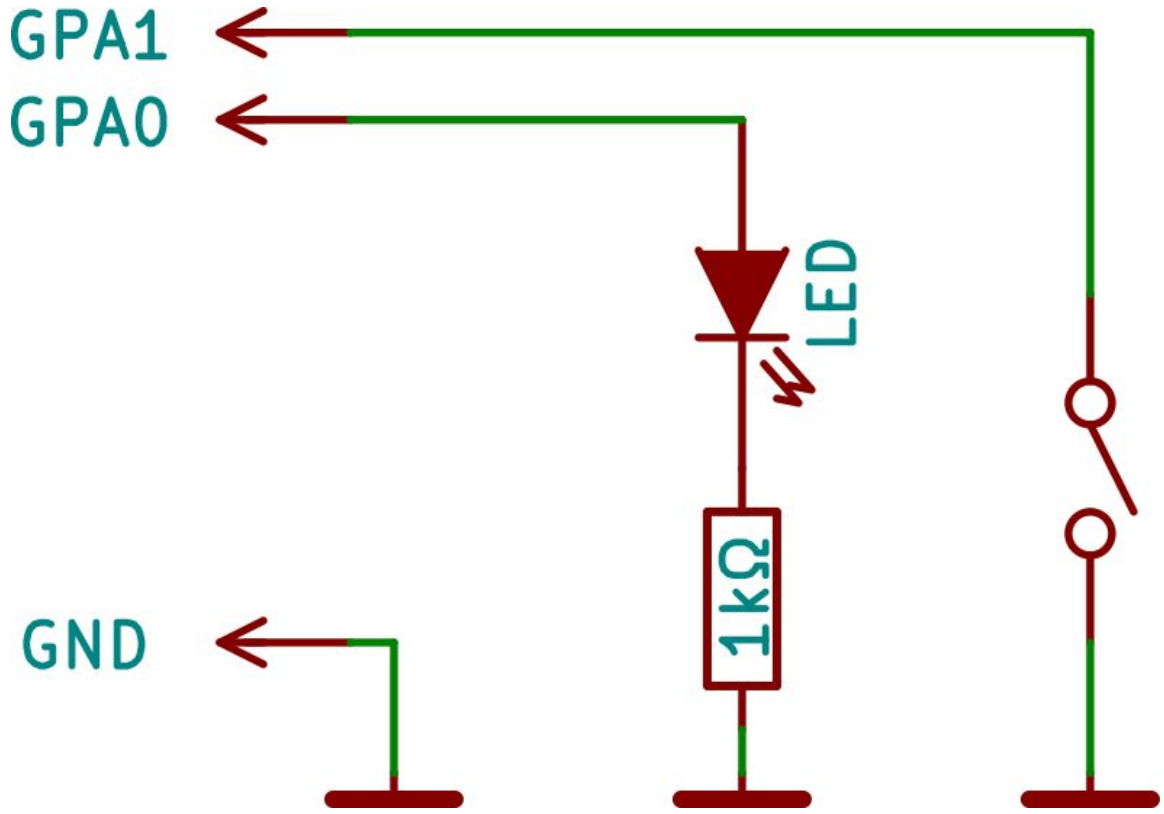


Anschluss des Port-Expanders MCP23017 am I²C-Bus

Den MCP23017 kann man elementar über den I²C-Bus mit Hilfe der Arduino-Bibliothek `Wire` ansprechen. Dazu wäre eine detaillierte Kenntnis der internen Regester nötig [4]. Wesentlich einfacher gestaltet sich die Ansteuerung über die Adafruit MCP23017 Arduino Library [2], die man

zunächst (in der üblichen Weise) herunterladen und in die Arduino-Umgebung einbinden muss.

Der Betrieb des Schaltkreises und die Funktion der Bibliothek sollen mit Hilfe einer Testschaltung erprobt werden:



Einfache Testschaltung für Port-Expander MCP23017

In einem Programm zur Ansteuerung des Schaltkreises sind als zuerst die Header-Dateien einzubinden:

```
#include <Wire.h>
#include <Adafruit_MCP23017.h>
```

Danach legt man für eine Instanz `mcp` der Klasse `Adafruit_MCP23017` an:

```
Adafruit_MCP23017 mcp;
```

Zunächst soll die LED zum Blinken gebracht werden. Die Nummer des verwendeten Anschlusses GPA0 hinterlegen wir in der folgen Konstanten:

```
const byte pinLED = 0;
```

Mit dem Aufruf der Methode `begin` ohne Argument wird als I²C-Adresse die Grundadresse `20H` eingestellt. Mit `pinMode` konfiguriert man für den jeweiligen Anschluss die Datenrichtung:

```
void setup() {
    mcp.begin();
    mcp.pinMode(pinLED, OUTPUT);
}
```

Die Hauptschleife entspricht weitgehend dem Beispielprogramm `Blink`, wobei jetzt kein Kanal des Arduino-Boards, sondern der entsprechende digitale Kanal des MCP23017 angesprochen wird.

```
void loop() {
    mcp.digitalWrite(pinLED, HIGH);
    delay(1000);
    mcp.digitalWrite(pinLED, LOW);
    delay(1000);
}
```

Die angeschlossene LED wird damit im Sekundenrythmus ein- bzw. ausgeschaltet.

Beim nächsten Test soll der Taster abgefragt und damit die LED geschaltet werden. Dazu definieren wir die verwendeten Anschlüsse (GPA0 und GPA1):

```
const byte pinLED = 0;
const byte pinKey = 1;
```

Der zusätzliche Kanal für den Taster wird als Eingang konfiguriert. Ferner wird der interne Pull-Up-Widerstand mit der Methode `pullUp` zugeschaltet:

```
void setup() {
    mcp.begin();
    mcp.pinMode(pinLED, OUTPUT);
    mcp.pinMode(pinKey, INPUT);
    mcp.pullUp(pinKey, HIGH);
}
```

In der Hauptschleife wird zyklisch der Eingangskanal abgefragt und das Abfrageergebnis zu dem mit der LED beschalteten Ausgabekanal ausgegeben:

```
void loop() {
  mcp.digitalWrite(pinLED,
   mcp.digitalRead(pinKey));
}
```

Will man mehrere ICs MCP23017 am I²C-Bus betreiben, so ist für jeden Schaltkreis ein Objekt der Klasse `Adafruit_MCP23017` anzulegen:

```
Adafruit_MCP23017 mcp1;
Adafruit_MCP23017 mcp1;
```

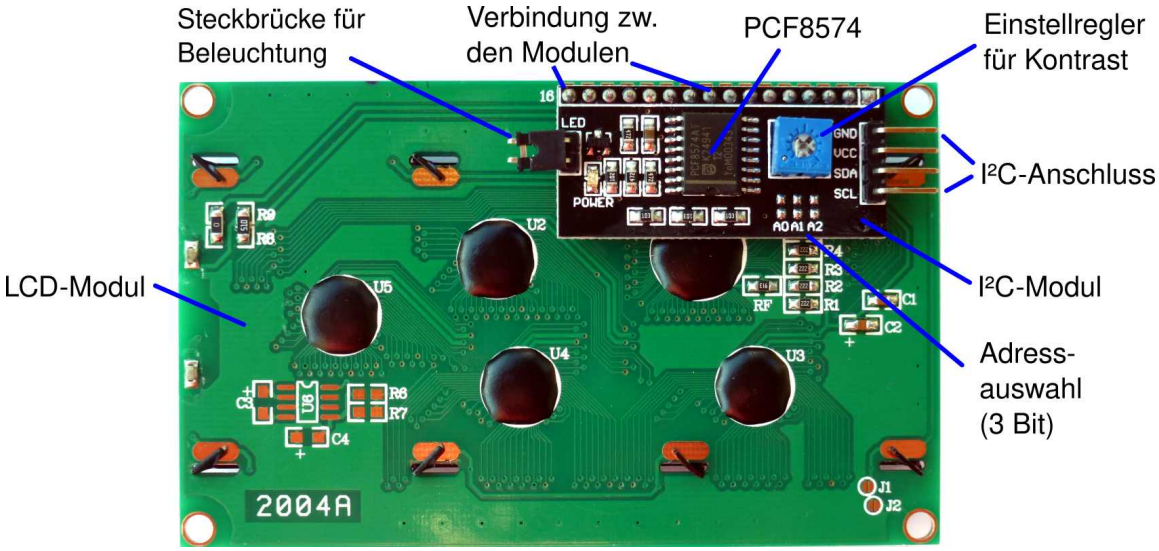
Über die Adresssignale A0, A1, A2 ist jedem Schaltkreis eine individuelle Adresse zuzuweisen. Die jeweiligen Subadressen sind über die Methode `begin` einzustellen:

```
void setup() {
  mcp1.begin(0);  // Adresse 20H
  mcp2.begin(1);  // Adresse 21H
  ...
}
```

10.3 LCD-Ansteuerung über I²C-Bus

Die Ansteuerung eines LCD-Moduls belegt auch im 4-Bit-Modus noch 6 digitale Kanäle des Arduino-Boards. Bei den Standard-Boards (Arduino Uno, Leonardo, …) ist damit fast die Hälfte der digitalen Anschlüsse belegt. Eine mögliche Alternative wäre die serielle Ansteuerung des LCD-Moduls über den I²C-Bus. Für LCD-Module auf Basis des HD44780 oder

kompatibler Schaltkreise gibt es entsprechende Adaptermodule, die über eine 16-polige Stiftleiste auf der Rückseite des LCD-Moduls angelötet werden können. Diese Adapter beruhen oft auf den Schaltkreis PCF8574.



LCD-Modul mit I²C-Adapter

Softwareseitig bietet sich für den Betrieb des LCD-Moduls die Bibliothek `New LiquidCrystal` an [3]. Diese Bibliothek ist nicht Bestandteil der Arduino-Umgebung und muss daher zunächst heruntergeladen und in der üblichen Weise installiert werden. Mit dieser Bibliothek ist auch die effiziente Ansteuerung eines LCD-Moduls im 4-Bit-Betrieb möglich. In diesem Fall wäre die systemseitig vorhandene Bibliothek `LiquidCrystal` zu löschen, da die gleichen Dateinamen verwendet werden. Zusätzlich ist mit der neuen Bibliothek auch die Ansteuerung über SPI möglich. Für die Kommukation über den I²C-Bus ist außerdem die Bibliothek `Wire` einzubinden:

```
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

Zur Ansteuerung ist zunächst die Adresse des I²C-Adaptermoduls festzustellen. Gängige Adressen für LCD-Module sind `27H` oder `3FH`. Bei dem gezeigten Adaptermodul sind über die Lötbrücken A0, A1, A2 weitere Adressen einstellbar. Dadurch kann man beispielsweise mehrere LCD-Module am gleichen I²C-Bus betreiben. Zur Feststellung der tatsächlich

belegten Adresse lässt sich das am Anfang des Kapitels beschriebene Programm einsetzen. Beim Instanziieren eines Objektes der Klasse `LiquidCrystal_I2C` ist mindestens die verwendete Adresse des I²C-Busses zu übergeben, z.B.:

```
LiquidCrystal_I2C lcd(0x3F);
```

Das eingesetzte LCD-Modul funktionierte damit noch nicht. In solchen Fällen muss zusätzlich die Pinzuordnung zwischen I²C-Adapter und dem LCD-Modul übergeben werden:

```
LiquidCrystal_I2C lcd(0x3F, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
```

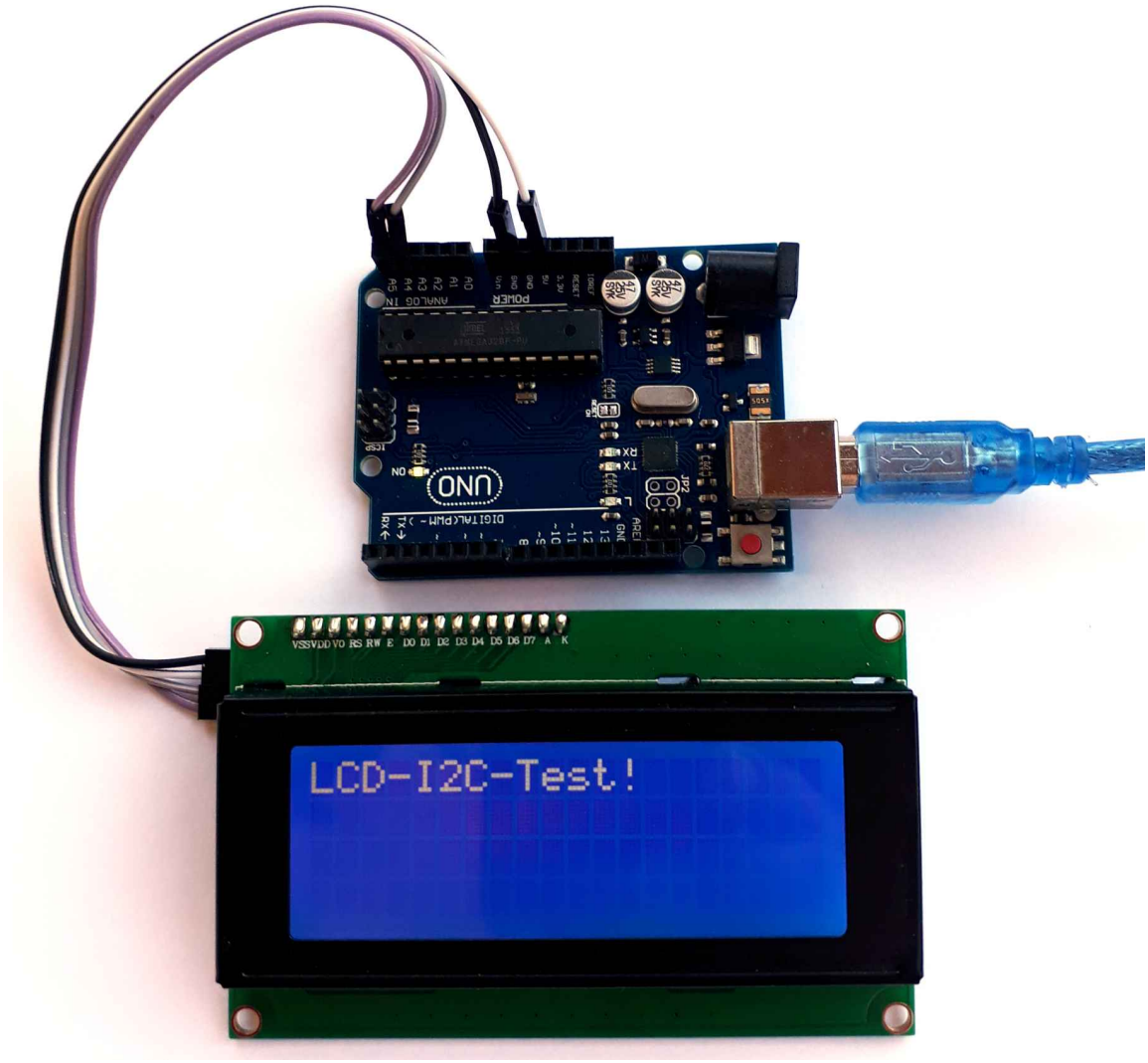
Nach der I²C-Adresse sind die Pins für Enable, R/W, RS, D4, …,D7 anzugeben. Bei Verwendung einer Hintergrundbeleuchtung sind außerdem der verwendete Pin und die Polarität zum Einschalten der Beleuchtung (`POSITIVE` oder `NEGATIVE`) zu übergeben. In der Funktion `setup` sind die Anzahl der Spalten und Zeilen des LCD-Moduls festzulegen. Mit den Methoden `backlight` bzw. `noBacklight` schaltet man die Hintergrundbeleuchtung an bzw. aus.

```
void setup()
{
  lcd.begin(20,4);
  lcd.backlight();
  lcd.home();
  lcd.print("LCD-I2C-Test!");
}
```

In der Hauptschleife des Testprogramms ist dann nichts mehr zu tun:

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

Die nächste Abbildung zeigt den Testbetrieb des LCD-Moduls:



LCD-Modul im Testbetrieb mit Uno-Board

10.4 Quellenangabe

1. Arduino – Wire. URL: https://www.arduino.cc/en/Reference/Wire.
2. Fried, L.: *Adafruit MCP23017 Arduino Library*. URL: https://github.com/adafruit/Adafruit-MCP23017-Arduino-Library.
3. Malpartida, F.: *New LiquidCrystal*. URL: https://bitbucket.org/fmalpartida/new-liquidcrystal.
4. Microchip: *MCP23017/MCP23S17, 16-Bit I/O Expander with Serial Interface*, 2007.

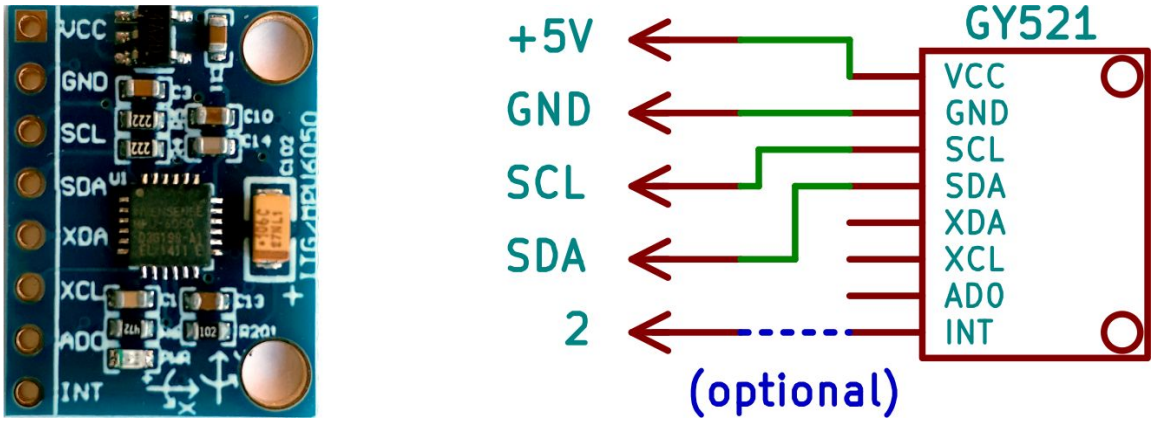
5. I²C. In: Wikipedia, Die freie Enzyklopädie. URL: https://de.wikipedia.org/w/index.php?title=I%C2%B2C.

11 Inertiale Messeinheiten und Magnetometer

11.1 Inertiale Messeinheit MPU-6050

11.1.1 Anschluss und Auslesen der Rohdaten

Eine *inertiale Messeinheit* (engl. *inertial measurement unit*, kurz *IMU*) besteht aus Beschleunigungs- und Drehratensensoren. Dieser Abschnitt widmet sich dem IMU-Schaltkreis MPU-6050 von InvenSense Inc. [7]. Dieser Schaltkreis kommt bei verschiedenen IMU-Modulen zum Einsatz. Die nachfolgenden Versuche wurden mit dem Modul GY-521 durchgeführt. Auf der Leiterplatte sind *x*- und *y*-Achse sowie die zugehörigen Rotationswinkel gekennzeichnet.



Modul GY-521 mit MPU-6050 (links), Anschluss des Moduls (rechts)

Der Schaltkreis MPU-6050 ist für den Betrieb mit 3,3V ausgelegt. Das Modul GY-521 besitzt einen Spannungsregler und erlaubt dadurch auch den Betrieb mit 5V. Dadurch kann das Modul direkt an die Leitungen SCL und SDA des I²C-Busses der gängigen Arduino-Boards angeschlossenen werden. Bei Verwendung der I2C Device Library (`i2cdevlib`) von Jeff Rowberg [12] ist zusätzlich die Interrupt-Nutzung möglich. Das Modul informiert dabei das Arduino-Board, wenn Daten anliegen. Beim Arduino Uno ist Pin 2 für

Interrupt vorgesehen. Der Anschluss AD0 ermöglicht die Auswahl zwischen den I²C-Adressen 68H und 69H. Durch einen 4,7kΩ Pull-Down-Widerstand ist dieser Anschluss im Normalfall auf LOW, was der Adresse 68H entspricht. Zur Auswahl der Adresse 69H ist der Anschluss AD0 auf HIGH-Pegel zu legen, entweder direkt (durch einen Draht) auf 3,3V oder mit einem 2,7kΩ-Widerstand auf 5V.

Die nächste Tabelle gibt Adressen und Bezeichnungen der wichtigsten Register des Schaltkreises MPU-6050 an [8]. Die Register `ACCEL_XOUT`, `ACCEL_YOUT`, `ACCEL_ZOUT` geben die Beschleunigungen für die *x*-, *y*- bzw. *z*-Achse an, die Register `GYRO_XOUT`, `GYRO_YOUT`, `GYRO_ZOUT` die Winkelgeschwindigkeiten (Drehraten) für die Rotation um diese Achsen. Zusätzlich lässt sich mit `TEMP_OUT` die Temperatur auslesen. Diese Datenregister sind 16-Bit-Register, die intern als zwei 8-Bit-Register hinterlegt sind (z.B. `ACCEL_XOUT` als `ACCEL_XOUT_H` für den HIGH-Teil und `ACCEL_XOUT_L` für den LOW-Teil). Die 16-Bit-Zahlen sind im Zweierkomplement kodiert und umfassen damit einen Wertebereich von -32768,…,+32767. In der Grundeinstellung des Schaltkreises werden Beschleunigungen von ±*g* bzw. Winkelgeschwindigkeiten von ±250°/s auf diesen Zahlenbereich abgebildet. Dabei gibt $g \approx 9{,}81\text{m/s}^2$ die (mittlere) Fallbeschleunigung an der Erdoberfläche an. Mit den 8-Bit-Konfigurationsregistern `GYRO_CONFIG` und `ACCEL_CONFIG` kann man bei Bedarf die Wertebereiche auf ±500°/s, ±1000°/s, ±2000°/s bzw. ±4*g*, ±8*g*, ±16*g* verstellen. Über das 8-Bit-Register `PWR_MGMT_1` wird der Schaltkreis aktiviert bzw. kann umgekehrt über dieses Register auch in den Schlafmodus versetzt werden.

Wichtige Register des ICs MPU-6050 [8]

Adresse	*Register*
1BH	GYRO_CONFIG
1CH	ACCEL_CONFIG
3BH-3CH	ACCEL_XOUT
3DH-3EH	ACCEL_YOUT
3FH-40H	ACCEL_ZOUT
41H-42H	TEMP_OUT
43H-44H	GYRO_XOUT
45H-46H	GYRO_YOUT
47H-48H	GYRO_ZOUT
6BH	PWR_MGMT_1

Das nachfolgende Programm soll die Rohdaten abfragen und über den seriellen Monitor ausgeben. Für die Kommunikation mit der IMU über den I²C-Bus ist die benötigte Bibliothek einzubinden:

```
#include <Wire.h>
```

Das Modul ist standardmäßig auf die I²C-Adresse 68H eingestellt. Auszulesen sind die in der Tabelle angebenen 16-Bit-Datenregister `ACCEL_OUT` bis `GYRO_ZOUT`. Das erste Register (`ACCEL_XOUT` bzw. `ACEEL_XOUT_H`) beginnt mit Adresse 3BH. Zusätzlich wird die Adresse des 8-Bit-Registers `PWR_MGMT_1` zum Aktivieren des Schaltkreises benötigt:

```
const int MPU_ADDR  = 0x68; // I2C-Adresse
const int ACCEL_XOUT = 0x3B; // 1. Datenregister
const int PWR_MGMT_1 = 0x6B; // Statusregister
```

Die einzulesenden Daten sind 16-Bit-Werte. Für die Darstellung im Zweierkomplement ist die korrekte Größe des verwendeten Zahlentyps relevant. Um auch hinsichtlich neuerer Arduino-Architekturen mit ARM- oder Intel-Prozessoren die korrekte Datengröße sicherzustellen, wird der Typ `int16_t` verwendet:

```
int16_t data;
```

In der Funktion `setup` wird die Datenübertragung zum IC MPU-6050 über den I²C-Bus gestartet. Mit dem Zugriff auf das Register `PWR_MGMT_1` aktiviert man zudem den IMU-Schaltkreis. Außerdem wird der Datenkanal zum seriellen Monitor geöffnet.

```
void setup() {
    Wire.begin();
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(PWR_MGMT_1);
    Wire.write(0);
    Wire.endTransmission(true);
    Serial.begin(9600);
}
```

In der Hauptschleife wird die Datenübertragung dadurch eingeleitet, dass zunächst die Adresse des ersten Datenregisters übermittelt wird. Als nächstes werden 14 Bytes (7 Register mit je 8 Bit) angefordert. Bei den 16-Bit-Registern enthält das erste Byte die höheren 8 Bit (HIGH-Teil), das zweite Byte die niedrigeren 8 Bit (LOW-Teil). Zuerst wird der HIGH-Teil eingelesen und um 8 Bit nach links verschoben. Danach wird der LOW-Teil eingelesen und mit dem vorherigen Wert ODER-verknüpft. Die eingelesenen Werte werden über den seriellen Monitor ausgegeben und sind durch jeweils einen Tabulator getrennt. Dieser Ablauf wird nach 500ms Wartezeit wiederholt.

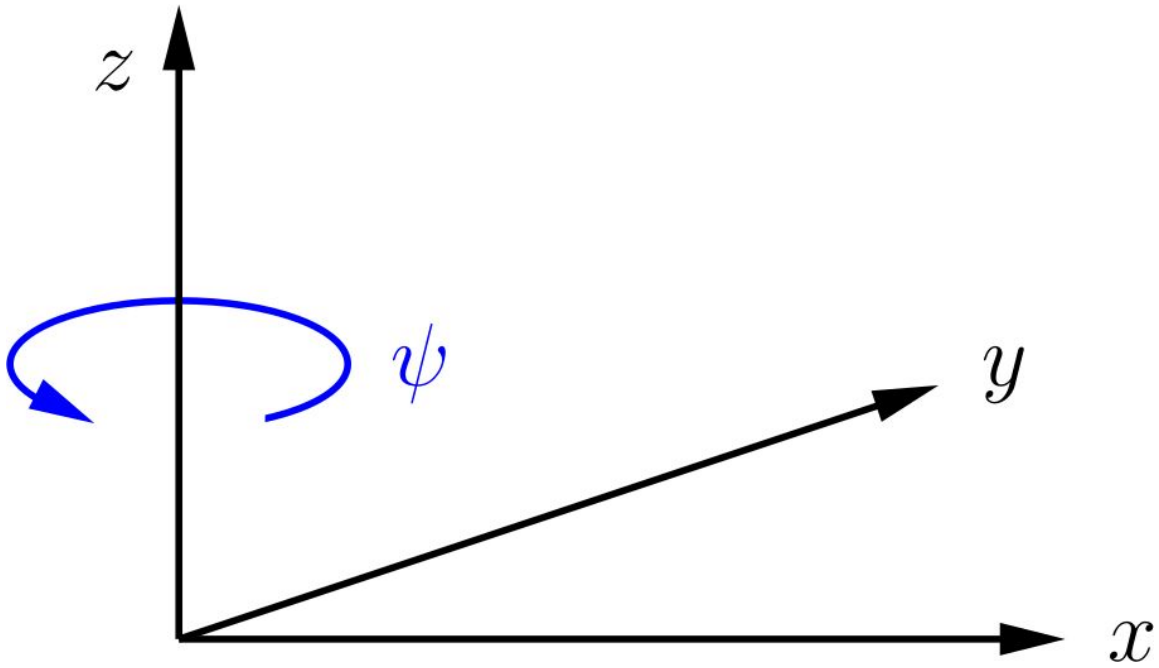
```
void loop() {
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(ACCEL_XOUT);  // 1. verwendete Datenregister
    Wire.endTransmission(false);
    Wire.requestFrom(MPU_ADDR,14,true);  // 14 8-Bit-Werte
    for (int i=0; i<7; i++) {
        data=Wire.read()<<8 | Wire.read();
        Serial.print(data);
        Serial.print("\t"); // Tabulator
    }
    Serial.println();
    delay(500);
}
```

Liegt das Modul horizontal und verharrt in Ruhe, dann sollten die ersten zwei Werte (Beschleunigungen in *x*- bzw. *y*-Richtung klein (im Vergleich zu

2^{15} = 32768) sein. Der dritte Wert gibt (bis auf Fehler durch leichte Schräglage des Moduls) die Beschleunigung in z-Richtung, also senkrecht zur Leiterplatte, an. Bei einem Messbereich von ±2*g* entspricht die Fallbeschleunigung *g* dem halben Messbereich, also etwa einem Wert von 2^{14} = 16384. Der vierte Wert entspricht der Temperatur. Die letzten drei Werte sind die Winkelgeschwindigkeiten, die bei ruhendem Modul kleine Werte und bei Rotation um die jeweilige Achse entsprechend größere Werte annehmen sollten.

11.1.2 Bestimmung des Gierwinkels

In der Regel wird der mobile Roboter für einen ebenen Boden vorgesehen sein und sich daher nur horizontal bewegen. Ist die Leiterplatte des Moduls GY-521 ebenfalls horizontal auf dem Roboterchassis angebracht, dann wird die Orientierung des Roboters durch den Gierwinkel ψ beschrieben, also als Rotation um die *z*-Achse



Gierwinkel ψ

Die intertiale Messeinheit ermittelt leider keine Winkel, sondern nur Winkelgeschwindigkeiten. Bezeichne ω die Winkelgeschwindigkeit der Rotation um

die *z*-Achse. Dann ist ω die sog. *Gierrate*, d.h. die zeitliche Ableitung des Gierwinkels:

$$\omega(t) = d\psi(t) / dt$$

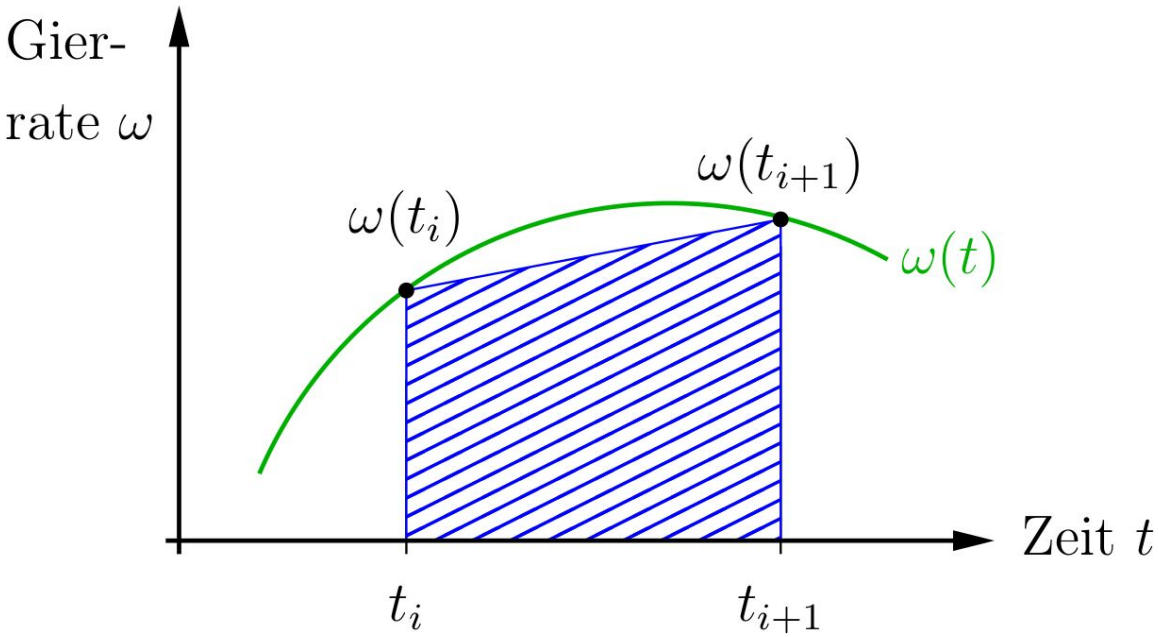
Umgekehrt kann man aus einem bekannten Verlauf der Gierrate den Gierwinkel durch zeitliche Integration bestimmen

$$\psi(t) = \psi(t_0) + \int_{t_0}^{t} \omega(\tau)\,\mathrm{d}\tau,$$

wobei zu dem betrachteten Anfangszeitpunkt t_0 der zugehörige Anfangswinkel ψ_0 bekannt sein muss (oder festgelegt wird). In der Regel setzt man $t_0 := 0$. Der Übergang von einem Zeitpunkt t_i zum nächsten Zeitpunkt t_{i+1} lässt sich durch numerische Integration mit der *Trapezregel* bzw. *Trapezformel* folgendermaßen formulieren [3]:

$$\psi(t_{i+1}) = \psi(t_i) + \frac{\omega(t_i) + \omega(t_{i+1})}{2}\,(t_{i+1} - t_i)$$

Die nächste Abbildung veranschaulicht den numerischen integrationsvorgang:



Numerische Integration der Gierrate mittels Trapezregel

Bei der Implementierung wird zuerst wieder die Bibliothek für den Zugriff zum I²C-Bus eingebunden:

```
#include <Wire.h>
```

Bei den Konstanten kommt die Adresse `47H` des Registers `GYRO_ZOUT` für die Drehrate bezüglich der *z*-Achse (Gierrate) hinzu:

```
// I2C-Adresse für MPU 6050
const int MPU_ADDR   = 0x68;
// Registeradresse
const int GYRO_ZOUT  = 0x47;
// Statusregister
const int PWR_MGMT_1 = 0x6B;
```

Die Systemzeit wird für aufeinanderfolgende Abtastzeitpunkte jeweils als ganze Zahl gespeichert. Für die Zeitdifferenz in Sekunden wird eine Gleitkommazahl bereitgestellt:

```
long unsigned t_old;
long unsigned t_new;
double dt; // Zeitdifferenz
```

Die Winkelgeschwindigkeiten werden einerseits als 16-Bit-Registerwerte gespeichert, andererseits auch als Gleitkommazahlen in °/s:

```
int16_t omega;
int16_t omega0=0;
double omega_old;
double omega_new;
```

Der Gierwinkel ψ wird mit dem Anfangswert Null initialisiert:

```
double psi=0; // Gierwinkel
```

Das Auslesen des Registers `GYRO_ZOUT` für die Gierrate erfolgt mit der Funktion `read_omega`:

```
int16_t read_omega() {
    int16_t data;
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(GYRO_ZOUT);  // Register Drehrate z-Achse
    Wire.endTransmission(false);
    Wire.requestFrom(MPU_ADDR,2,true);
    data=Wire.read()<<8|Wire.read();
    return data;
}
```

In der Funktion `setup` wird die Datenübertragung sowohl zur IMU als auch zum seriellen Monitor aktiviert. Außerdem wird der Wert der Gierrate 100 mal eingelesen, gemittelt und in der Variablen `omega0` gespeichert. Unter der Voraussetzung, dass der mobile Roboter während dieser Kalibrierung in Ruhe verharrt, lässt sich der gespeicherte Wert zur Offsetkompensation verwenden. Zusätzlich wird die Systemzeit (in ms) ausgelesen und gespeichert.

```
void setup() {
    Wire.begin();
    Wire.beginTransmission(MPU_ADDR);
    Wire.write(PWR_MGMT_1);
    Wire.write(0);
    Wire.endTransmission(true);
    Serial.begin(9600);
    // Kalibrierung
    const unsigned N=100;
    for (unsigned i=0; i<N; i++)
```

```
        omega0+=read_omega();
    omega0/=N;
    Serial.print("Kalibirierung (Omega0) = ");
    Serial.println(omega0);
    t_new = millis();
    omega_new = 0;
}
```

Mit jedem Durchlauf der Hauptschleife wird zunächst die Systemzeit ausgelesen. Daraus bestimmt man die Zeitdifferenz (in s) zum vorangegangenen Durchlauf. Als nächstes wird der Registerwert der Gierrate mit der Funktion `read_omega` ausgelesen. Mit Hilfe des in der Funktion `setup` bestimmten und in der Variablen `omega0` gespeicherten Wertes erfolgt eine Offsetkompensation. Nach einer Umskalierung des Registerwertes in °/s erfolgt die Integration entsprechend der Trapezregel. Das Ergebnis dieser Integration ist der Gierwinkel ψ, der zusammen mit einigen Zwischenergebnissen zyklisch ausgegeben wird.

```
void loop() {
    t_old=t_new;
    t_new=millis();
    dt=(t_new-t_old)/1000.0; // Zeitdifferenz in s
    omega=read_omega()-omega0;
    Serial.print("Omega (roh) = ");
    Serial.print(omega);
    omega_old=omega_new;
    omega_new=250.0*omega/32768.0; // Skalierung in °/s
    Serial.print(" | Omega (°/s) = ");
    Serial.print(omega_new);
    psi+=0.5*(omega_old+omega_new)*dt; // Integration
    Serial.print(" | Winkel (°) = ");
    Serial.println(psi);
    delay(100);
}
```

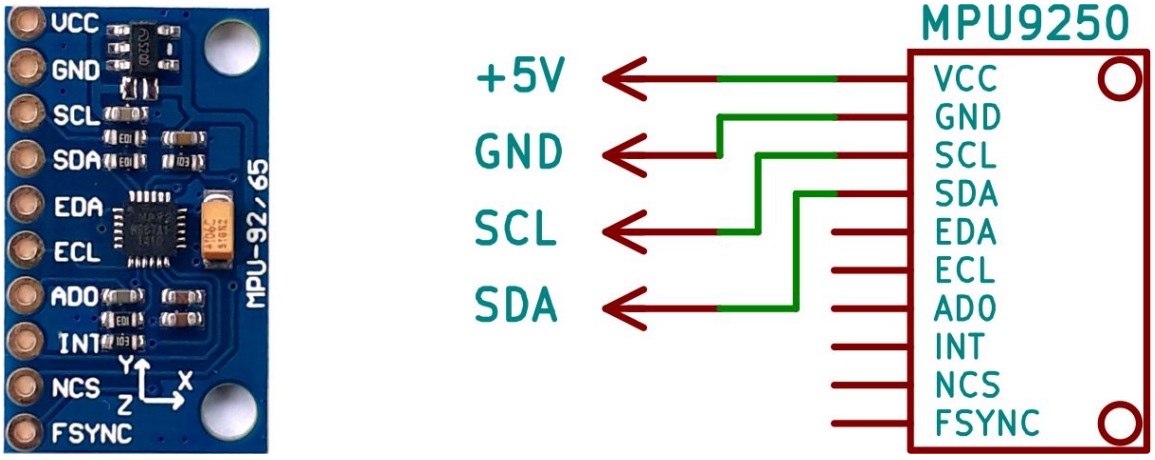
Neben der beschriebenen direkten Implementierung zur Berechnung des Gierwinkels kann man alternativ auch die I2C Device Library (`i2cdevlib`) nutzen. Dazu lädt man die Gesamtbibliothek als ZIP-Datei herunter [12]. Nach dem Entpacken kopiert man aus dem Unterverzeichnis `Arduino` der Bibliothek die Unterverzeichnisse `I2Cdev` und `MPU6050` in das lokale Bibliotheksverzeichnis `libraries` der Arduino-Umgebung. Unter **Datei > Beispiele > MPU6050 > MPU6050DMP6** findet man ein Beispielprogramm zur Berechnung der

Euler-Winkel (*Gear-*, *Nick-* und *Rollwinkel*, engl. *yaw*, *pitch*, *nick angle*). Dabei ist die Interruptleitung entsprechend der o.g. Abbildung zu verbinden.

11.2 Inertiale Messeinheit MPU-9250 mit Magnetometer

11.2.1 Anschluss und Auslesen der Rohdaten

Die inertiale Messeinheit MPU-9250 wird wie der im vorangegangenen Abschnitt behandelte IC MPU-6050 von InvenSense Inc. angeboten [10]. In Ergänzung zu Beschleunigungs- und Drehratensensoren verfügt diese Messeinheit auch über ein 3-Achsen-Magnetometer, mit dem die magentische Flussdichte bestimmt wird. Die Abbildung (links) zeigt ein gängiges Modul mit dem IC MPU-9250. Der Anschluss an den I²C-Bus wird in der Abbildung (rechts) gezeigt. Außerdem wäre ein Betrieb mit SPI möglich. Wir betreiben das Modul hier aber über den I²C-Bus.



cc

Die Registerbelegung zur Ansteuerung der Beschleunigungs- und Drehratensensoren stimmt mit dem Schaltkreis MPU-6050 überein. Daher kann man die Programmbeispiele unmittelbar verwenden. Etwas schwieriger gestaltet sich die Ansteuerung bzw. Abfrage der integrierten Magnetometereinheit AK8963. Die benötigten Register des Magnetometers sind in der nächsten Tabelle aufgeführt.

Wichtige Register des Magnetometers AK8963 im MPU-9250 [9]

Adresse	*Register*	*Beschreibung*
02H	ST1	Status 1
05H-06H	HY	Daten y-Richtung
03H-04H	HX	Daten x-Richtung
07H-08H	HZ	Daten z-Richtung
09H	ST2	Status 2
0AH	CNTL	Control

Für die Abfrage des Magnetometers ist zuerst die Bibliothek zur Ansteuerung des I²C-Buses einzubinden:

```
#include <Wire.h>
```

In der Grundeinstellung ist auf dem I²C-Bus anfänglich nur der IC MPU-9250 unter der Adresse `068H` sichtbar. Das integrierte Magnetometer AK8963 verfügt über die separate I²C-Adressse 0CH. Beide Adressen werden im Programm als Konstante hinterlegt:

```cpp const byte MPU_ADDR = 0x68; const byte MAG_ADDR = 0x0C;
```

Für die Durchleitung der I²C-Kommunikation des AK8963 ist im IC MPU-9250 das Register `INT_PIN_CFG` zu konfigurieren. Diese Adresse wird zusammen mit den Adressen der Register des Magnetometer deklariert:

```
const byte INT_PIN_CFG = 0x37; // I2C-Konfiguration
const byte ST1  = 0x02;  // Status 1
const byte HX   = 0x03;  // Daten
const byte CNTL = 0x0A;  // Control
```

Da nun mehrere 8-Bit-Register zu konfigurieren sind, wird dafür die Hilfsfunktion `I2CWriteReg` bereitgestellt:

```
void I2CwriteReg(uint8_t address, uint8_t reg, uint8_t data) {
    Wire.beginTransmission(address);
    Wire.write(reg);
    Wire.write(data);
```

```
    Wire.endTransmission();
}
```

Die Funktion `I2CreadReg` liest in ähnlicher Weise ein 8-Bit-Register ein:

```
uint8_t I2CreadReg(uint8_t address, uint8_t reg) {
    Wire.beginTransmission(address);
    Wire.write(reg);
    Wire.endTransmission(false);
    Wire.requestFrom(address, (uint8_t) 1);
    uint8_t data = Wire.read();
    return data;
}
```

In der Funktion `setup` wird der IC MPU-9250 zur Durchleitung des I²C-Signals der Magnetometereinheit konfiguriert (engl. *bypass mode*). Das Magnetometer wird für eine kontinuierliche Messung mit 16 Bit Genauigkeit eingestellt:

```
void setup() {
    Wire.begin();
    Serial.begin(9600);
    Serial.println("Magnetometer MUP-9250");
    I2CwriteReg(MPU_ADDR, INT_PIN_CFG, 2); // I2C-Durchleitung
    I2CwriteReg(MAG_ADDR, CNTL, 0x16);     // Kont. Messung 16Bit
}
```

Zum Auslesen der Rohdaten des Magentometer legen wir die Funktion `readMag` an. Zunächst wird durch Abfrage des Statusregisters `ST1` solange gewartet, bis ein gültiger Messwert vorliegt. Anschließend werden 7 Byte eingelesen. Davon entfallen die ersten 6 Byte auf die drei 16 Bit-Messwerte für *x*-, *y*- und *z*-Achse. Im Unterschied zu den Daten bei den Beschleunigungs- bzw. Drehratensensoren ist beim Magnetometer zuerst der LOW-Teil und dann der HIGH-Teil abgelegt. Mit dem 7. Byte wird das Statusregister `ST2` abgefragt, womit man den nächsten Messzyklus ausgelöst [9].

```
void readMag(int16_t &x, int16_t &y, int16_t &z) {
    uint8_t r;  // Wert für Registerabfrage
    // Messung abgeschlossen?
    while (!(I2CreadReg(MAG_ADDR,ST1) & 0x01));
    // Daten anfordern
    Wire.beginTransmission(MAG_ADDR);
```

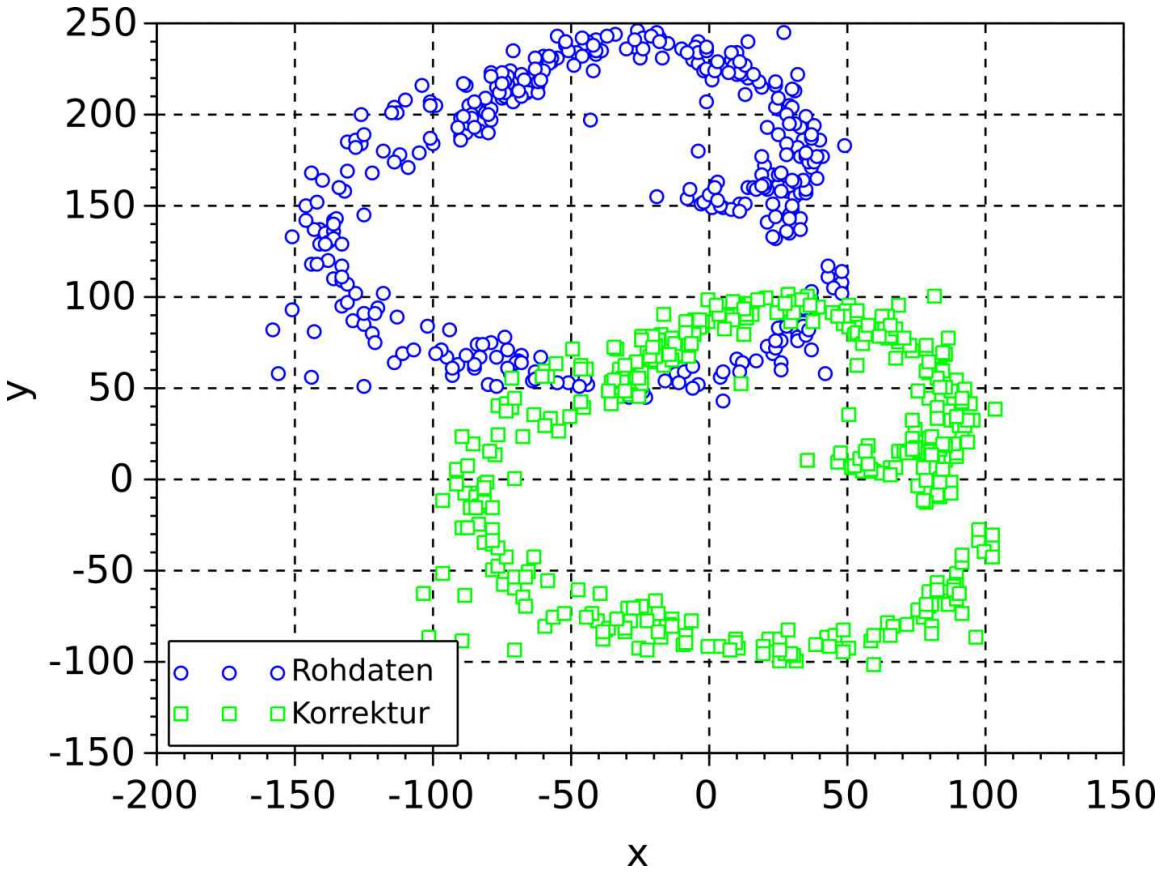
```
    Wire.write(HX); // 1. Datenregister (x-Richtung)
    Wire.endTransmission(false);
    Wire.requestFrom(MAG_ADDR, (byte) 7);
    // Daten verfügbar?
    while (Wire.available()<7) {};
    // Datenabfrage
    x = Wire.read() | Wire.read() << 8;
    y = Wire.read() | Wire.read() << 8;
    z = Wire.read() | Wire.read() << 8;
    r = Wire.read(); // Status-Register 2
}
```

In der Hauptschleife werden die Magnetometerdaten mit der Funktion `readMag` abgefragt und anschließend über den seriellen Monitor ausgegeben.

```
void loop() {
    int16_t x, y, z;   // Daten Magnetometer
    readMag(x, y, z);
    Serial.print(x);  Serial.print("\t");
    Serial.print(y);  Serial.print("\t");
    Serial.println(z);
    delay(100);
}
```

11.2.2 Kalibrierung des Magnetometers und Winkelberechnung

Die von Magnetometer erfasste magnetische Flussdichte ist eine vektorielle, d.h. gerichtete Größe. Sieht man von Störeinflüssen (z.B. durch Permanent- oder Elektromagnete sowie ferromagnetische Materialien wie Eisen) ab, so wird die Flussdichte maßgeblich vom Erdmagnetfeld bestimmt. Eine Drehung des Sensors würde nicht den Betrag bzw. die Stärke der Flussdichte ändern, sondern nur zu einer anderen Zuordnung der (x,y,z)-Anteile führen. Bei einer Drehung in der (x,y)-Ebene wäre zu erwarten, dass die zugehörigen Messwerte auf einem Kreis um den Nullpunkt liegen. Das ist bei den in der nächsten Abbildung dargestellten Rohdaten (blau), die über mehrere Drehungen des Robters aufgenommen wurden, nicht der Fall. Die Messwertze für die x- bzw. y-Richtung weisen offensichtlich eine deutliche Verschiebung (Offset) auf. Für eine Bestimmung des Gierwinkels ist daher eine Offsetkompensation erforderlich. Die korrigierten Werte (grün) sind ebenfalls der Abbildung zu entnehmen.



Ausgelesene Rohdaten des Magnetometers (blau), durch Offsetkompensation korrigierte Werte (grün)

Zur Berechnung der Offsetwerte ermitteln wir aus den gemessenen Rohdaten des Magnetometers die Minima und Maxima in *x*- und *y*-Richtung. Für die Initialisierung der entsprechenden Variablen benötigen wir die folgende Bibliothek:

```
#include <limits.h>
```

Vor der Hauptschleife legen wir für die Extremalwerte globale Variablen an. Diese Variablen werden so initialisiert, dass bei der ersten Durchlauf die Bildung von Minimum bzw. Maximum den ersten Messwert liefert:

```
int xmin = INT_MAX;
int xmax = INT_MIN;
int ymin = INT_MAX;
int ymax = INT_MIN;
```

In der Hauptschleife werden nach dem Einlesen der Rohdaten die Minimal- und Maximalwerte aktualisiert. Zur Offsetkompensation wird für jede Achse aus Minimum und Maximum das arithmetische Mittel gebildet. Damit erhält man die (in jedem Schritt aktualisierten) Mittelpositionen für die Messwerte der *x*- und *y*-Achse. Aus den offsetkompensierten karthesischen Koordinaten bestimmt man mit der 4-Quadranten-Arkustangensfunktion `atan2` den zugehörigen Winkel im Wertebereich von ±π (also im Bogenmaß, Einheit: Radiant). Über die Systemkonstante `RAD_TO_DEG` wird der Winkel in den Bereich von ±180° skaliert. Ist der so berechnete Winkel negativ, so erfolgt eine Abbildung auf den Bereich 180°…360°, so dass man insgesamt Winkel im Bereich 0…360° erhält:

```
void loop() {
    int16_t x, y, z;   // Daten Magnetometer
    readMag(x, y, z);
    // Auswertung
    xmin = min(x,xmin);
    xmax = max(x,xmax);
    ymin = min(y,ymin);
    ymax = max(y,ymax);
    // Berechnung Offset
    float x0=(xmin+xmax)/2.0;
    float y0=(ymin+ymax)/2.0;
    // Winkelberechnung und Ausgabe
    float angle = atan2(y-y0,x-x0) * RAD_TO_DEG;
    if(angle < 0) angle = angle + 360;
    Serial.print("Winkel: ");
    Serial.println(angle);
    delay(100);
}
```

Nach einigen Drehung liefert diese Routine brauchbare Ergebnisse für den Gierwinkel. Anstelle einer fortlaufenden Aktualisierung wäre es auch denkbar, die entsprechenden Offsetwerte aus den Messdaten einiger Umdrehungen in der Routine `setup` zu ermitteln. Zur Verbesserung der Genauigkeit könnte man zusätzlich die Achsen skalieren bzw. die Messwerte filtern.

11.3 Magnetometermodule GY-271 bzw. GY-273

Die in der nächsten Abbildung gezeigten Module GY-271 und GY-273 beherbergen jeweils ein 3-Achsen-Magnetometer. Die Anschlüsse sind auf der Rückseite beschriftet. Beide Module verfügen über einen Spannungsregler, so dass die Versorgung über +5V erfolgen kann. Beim Modul GY-271 ist zusätzlich eine Pegelwandlung der Signalleitungen SCL und SDA vorhanden.

Magnetometermodule GY-271 (links) bzw. GY-273 (rechts)

Die Module sind eigentlich für den Schaltkreis HMC5883L vorgesehen [5]. Tatsächlich wird bei etlichen Modulen nicht der HMC5883L verbaut, sondern der pinkompatible IC QMC5883 [11]. Die Schaltkreise lassen sich anhand der I²C-Adresse bzw. der Beschriftung unterscheiden (siehe Tabelle).

Unterscheidung zwischen ICs HMC5883L und QMC5883L

IC	*I²C-Adresse*	*Beschriftung*
HMC5883L	`1EH`	L883
QMC5883L	`0DH`	DA5883

Der Betrieb mit dem HMC5883L ist beispielsweise in [4] beschrieben. Gibt man in der Arduino-IDE unter **Werkzeuge > Bibliotheken verwalten** den Suchbegriff „5883“, dann werden mehrere Bibliotheken zum HMC5883L, aber keine für den QMC5883L angezeigt. Allerdings existieren auch für den QMC5883 verschiedene Bibliotheken [6,13], die separat zu installieren sind.

Die vom Autor bei verschiedenen Anbietern bestellen Module waren ausschließlich mit dem QMC5883L bestückt. Daher wird nachfolgend dessen

Abfrage kurz beschrieben. Die wichtigsten Register des Magnetometers sind in der Tabelle aufgeführt [11].

Wichtige Register des Magnetometers QMC5883 [11]

Adresse	*Register/Beschreibung*
`00H-01H`	Daten x-Richtung
`02H-03H`	Daten y-Richtung
`04H-05H`	Daten z-Richtung
`06H`	Status Register
`09H`	Control Register 1
`0AH`	Control Register 2
`0BH`	Set/Reset-Time

Für den Zugriff auf den I²C-Bus wird die Bibliothek Wire benötigt. Zusätzlich gehen wir davon aus, dass die Funktion `I2CwriteReg` definiert ist. Die I²C-Adresse definieren wir als Konstante:

```
const byte MAG_ADDR = 0x0D;
```

In der Funktion `setup` erfolgt die Initialisierung des ICs. Dazu wird im Datenblatt empfohlen, dass Register `0BH` mit dem Wert `01H` zu beschreiben. Im Control Register 1 werden kontinuierlicher Messmodus mit 200Hz, maximaler Messbereich (±8 Gauß) und Oversamplingrate 256 eingestellt. Dazu ist folgender Code einzufügen:

```
I2CwriteReg(MAG_ADDR, 0x0B, B1);
I2CwriteReg(MAG_ADDR, 0x09, B11101);
```

Die Rohdaten für die drei Koordinatenrichtungen erhält man durch Einlesen von 6 Bytes ab Adresse `00H`:

```
void readMag(int16_t &x, int16_t &y, int16_t &z) {
    Wire.beginTransmission(MAG_ADDR);
    Wire.write(0); // erstes Datenregister
    Wire.endTransmission(false);
    Wire.requestFrom(MAG_ADDR, (byte) 6);
```

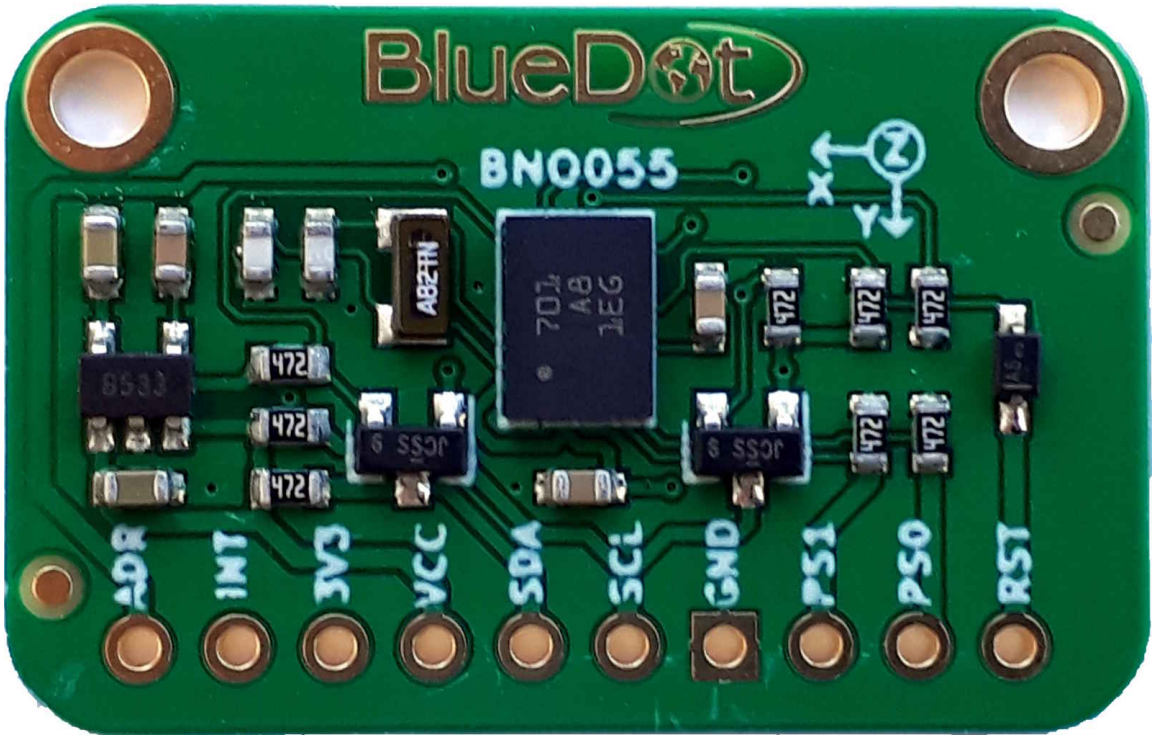
```
    x = Wire.read() | Wire.read() << 8;
    y = Wire.read() | Wire.read() << 8;
    z = Wire.read() | Wire.read() << 8;
}
```

Für eine saubere Implementierung müsste man zusätzlich den Abschluss des Messvorgangs über das Statusregister bzw. die Verfügbarkeit der Daten am I²C-Bus mit `Wire.available` abfragen. Alternativ könnte man über das Control Register 2 bei Verfügbarkeit neuer Messdaten auch einen Interrupt auslösen lassen. Für die Kalibrierung des Messwerte sei auf den vorangegangenen Abschnitt verwiesen.

11.4 Sensormodul BNO055

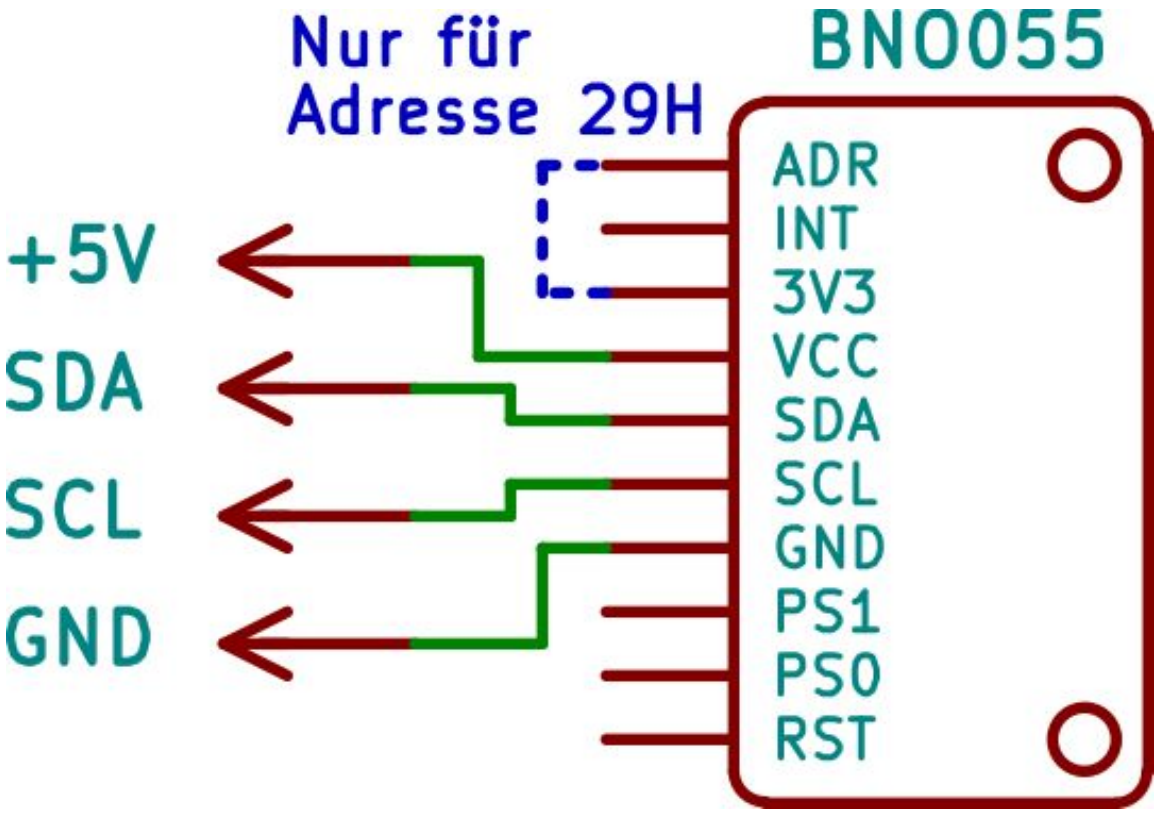
11.4.1 Anschluss und Test

Der von Bosch gefertigte Baustein BNO055 verfügt über Beschleunigungs- und Drehratensensoren sowie ein Magnetometer. Jeder dieser Sensoren erfasst Messwerte in 3-Achsrichtungen, so dass insgesamt ein 9-Achsen-Sensor vorliegt. Für die Signalauswertung ist zusätzlich ein 32-Bit Mikrocontroller integriert. Sensormodule werden beispielsweise von Adafruit und BlueDot angeboten:



Sensormodul BlueDot BNO055

Das Modul verfügt über einen Festspannungsregler und kann dadurch sowohl mit 3,3V als auch mit 5V betrieben werden. Bei den Arduino-Standard-Boards verbindet man den VCC-Anschluss des Sonsormoduls direkt mit dem 5V-Anschluss des Arduino-Boards. Lässt man den Pin ADR unbeschaltet, so erfolgt die Kommunikation über die Adresse `28H` des I²C-Busses. Verbindet man dagegen ADR mit dem Anschluss 3V3 (Hilfsspannung 3,3V), dann wird die Adresse `29H` genutzt. Die nächste Abbildung zeigt den Anschluss des BlueDot-Sensormoduls an ein Arduino-Standard-Board. Beim Adafruit-Sensormodul erfolgt der Anschluss prinzipiell nach dem gleichen Schema, allerdings sind die Pins anders angeordnet.



Anschluss des Sensormodul BlueDot BNO055 an den I²C-Bus

Für den Betrieb des Sensormoduls installiert man über **Werkzeuge > Bibliotheken verwalten** sowohl die `Adafruit Unified Sensor Library` als auch die `Adafruit BNO055 Library` [1,2]. Im Quelltext sind dann die benötigten Headerdateien einzubinden. Für die Ausgabe wird zusätzlich die LCD-Bibliothek `LiquidCrystal` benötigt.

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <LiquidCrystal.h>
```

Die Abfrage des Sensormoduls erfolgt über die Klasse `Adafruit_BNO055`. Im Normalfall wird das Modul auf der I²C-Standardadresse `28H` betrieben. In diesem Fall kann der Konstruktur bei der Instanziierung ohne weitere Argumente aufgerufen werden:

```
Adafruit_BNO055 bno = Adafruit_BNO055();
```

Bei einem Aufruf mit zwei Argumenten weist man dem Sensor eine Identifikationsnmummer und die I²C-Adresse zu, z.B.:

```
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);
```

Diese Variante ist insbesondere dann wichtig, wenn man nicht die Standardadresse `028H` verwendet, sondern die Ausweichadresse `29H`. Für die Ausgabe des Gierwinkels über das LCD-Modul wird die Instanz `lcd` der Klasse `LiquidCrystal` in der üblichen Weise angelegt. In der Funktion `setup` erfolgt die Initialisierung. Dabei wird auf die erfolgreiche Initialisierung des Sensormoduls gewartet:

```
void setup(void) {
    lcd.begin(40, 4);
    lcd.print("Sensor BNO055");
    while(!bno.begin()) {};
    bno.setExtCrystalUse(true);
    lcd.setCursor(0,1);
    lcd.print("Initialisiert!");
    delay(2000);
}
```

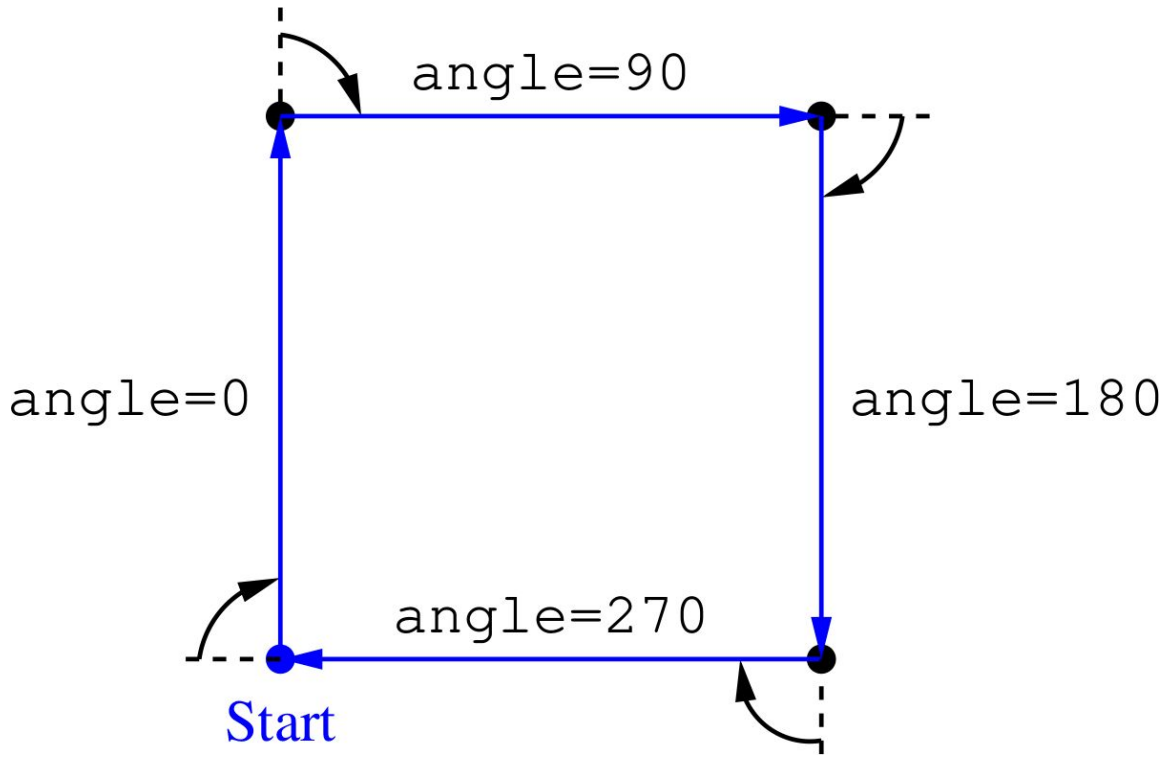
In der Hauptschleife wird der Sensor abgefragt und das zyklisch Ergebnis ausgegeben. Dabei ist zu beachten, dass hier der Gearwinkel im Uhrzeigersinn orientiert ist:

```
void loop(void) {
    sensors_event_t event;
    bno.getEvent(&event);
    float angle = event.orientation.x;
    lcd.setCursor(0,1);
    lcd.print("Winkel: ");
    lcd.print(angle);
    lcd.print("     ");
    delay(100);
}
```

11.4.2 Fahrt mit geregelter Orientierung

Der Gierwinkel beschreibt letztlich die Fahrtrichtung. Der Vergleich zwischen der gewünschten und der momentanen Fahrtrichtung liefert eine

Winkeldifferenz, die sich zur Regelung der Orientierung nutzen lässt. Dieses Vorgehen wird für das in der Abbildung gezeigte Manöver einer Fahrt im Karree behandelt.



Fahrmanöver (Fahrt im Karree)

Die Einbindung der Bibliotheken und die Initialisierung in der Funktion `setup` erfolgen im Wesentlichen wie im vorherigen Abschnitt. Zusätzlich muss man auch die Bibliothek zur Motoransteuerung einbinden, eine Instanz `motor` der Klasse `Motor` bereitstellen und in der Funktion `setup` passend initialisieren. Die Funktion `drive` übernimmt Lagebestimmung, Motoransteuerung und Regelung für einen Streckenabschnitt:

```
void drive (int angle, int V, unsigned T, float K) {
    long unsigned t = millis();
    do {
        // Winkelabweichung
        int dangle;
        sensors_event_t event;
        bno.getEvent(&event);
        dangle = round(event.orientation.x)-angle;
```

```
        dangle %= 360;
        if (dangle>180) dangle-=360;
        // Motorsignale
        int lmotor=V-round(K*dangle);
        int rmotor=V+round(K*dangle);
        motor.write(lmotor,rmotor);
        // LCD Ausgabe
        lcd.clear();
        lcd.print("L: ");
        lcd.print(lmotor);
        lcd.setCursor(0,1);
        lcd.print("R: ");
        lcd.print(rmotor);
    } while (millis()-t<T);
}
```

Das Argument `angle` gibt den Gierwinkel der gewünschten Fahrtrichtung vor (Sollwert). Über das Modul BNO055 wird der Gierwinkel des mobilen Roboters ermittelt (Istwert). Für die Differenz (Regelabweichung), die auf den Bereich von -180° bis +180° normiert wird, steht die Variable `dangle` zur Verfügung. Das Argument `V` beschreibt die nominelle (translatorische) Geschwindigkeit in Fahrtrichtung (Längsrichtung) als PWM-Signal zur Motoransteuerung. Unter Einbeziehung der Winkeldifferenz dangle und der Reglerverstärkung `K` ergeben sich daraus die Signale `lmotor` und `rmotor` zur Ansteuerung der beiden Motoren. Das Funktionsargument `T` gibt die für den Streckenabschnitt vorgesehene Fahrzeit in ms an.

Der Sollwert für den gewünschten Gierwinkel wird als globale Variable angelegt und initialisiert:

```
int angle=0;
```

In der Hauptschleife wird mit dem ersten Aufruf der Funktion `drive` das Fahrtzeug in die gewünschte Fahrtrichtung orientiert. Dafür ist eine Zeitdauer von 1s=1000ms vorgesehen. Dem schließt sich für 2s eine Geradeausfahrt in die eingestellte Richtung an. Anschließend wird der Sollwinkel um 90° erhöht, was einer Rechtsdrehung im rechten Winkel entspricht.

```
void loop(void) {
    drive(angle,   0, 1000, 2.0);
    drive(angle, 150, 2000, 2.0);
```

```
    angle=(angle+90) % 360;
}
```

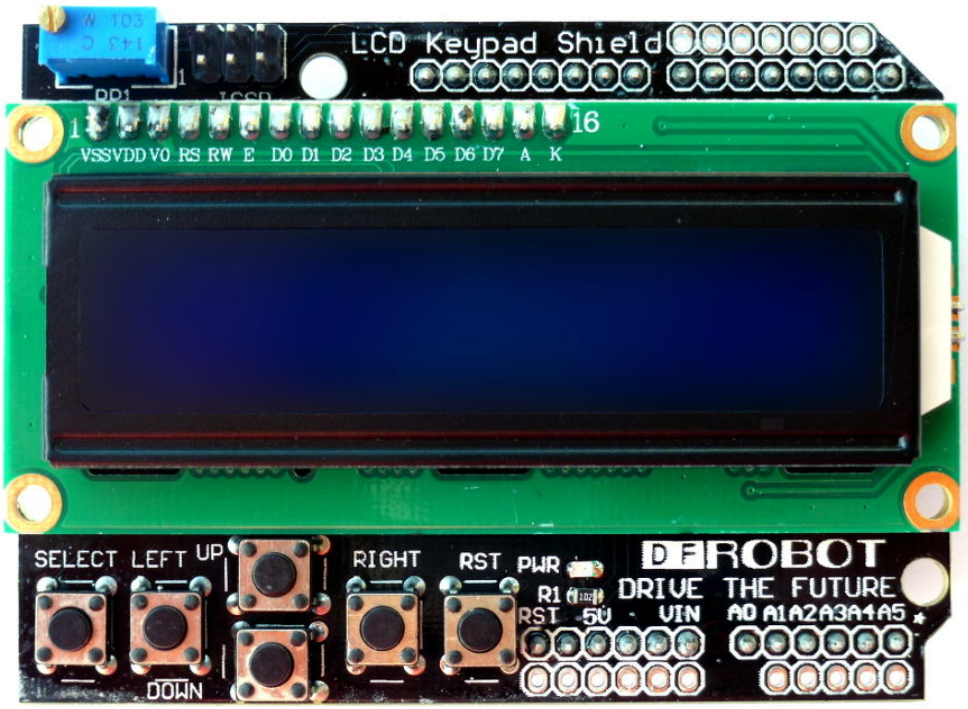
11.5 Quellenangabe

1. *Adafruit BNO055 Library*. URL: https://github.com/adafruit/Adafruit_BNO055
2. *Adafruit Unified Sensor Library*. URL: https://github.com/adafruit/Adafruit_Sensor
3. Bronstein, I. N.; K. A. Semendjajew; G. Musiol; H. Mühlig: *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 8. Auflage, 2012.
4. Th. Brühlmann: *Sensoren im Einsatz mit Arduino*. mitp Verlags GmbH & C. KG, 2017.
5. Honeywell: *Three-Axis Digital Compass IC HMC5883L*.
6. Y. Hong: *Mechasolution QMC5883L Library*. URL: https://github.com/keepworking/Mecha_QMC5883L
7. InvenSense Inc.: *MPU-6000 and MPU-6050 Product Specification*. Revision 3.4, 2013.
8. InvenSense Inc.: *MPU-6000 and MPU-6050 Register Map and Descriptions*. Revision 4.2, 2013.
9. InvenSense Inc.: *MPU-9250 Register Map and Descriptions*. Revision 1.6, 2015.
10. InvenSense Inc.: *MPU-9250 Product Specification*. Revision 1.1, 2016.
11. QST Corporation: *3-Axis Magnetic Sensor QMC5883L*. Februar 2016.
12. Rowberg, J.: *I2C Device Library*. URL: https://github.com/jrowberg/i2cdevlib.
13. D. Thain: *Arduino Library for the QMC5883L Magnetometer/Compass*. URL: https://github.com/dthain/QMC5883L

12 Zusätzliche Aufbauvarianten

12.1 LCD KeyPad Shield

Wer die beschriebene Verdrahtung eines LCD-Moduls mit dem Arduino-Board umgehen möchte, kann ein LCD KeyPad Shield einsetzen, welches von verschiedenen Anbietern zur Verfügung gestellt wird. Eine verbreitete Variante ist das in der nächsten Abbildung gezeigte LCD KeyPad Shield von DFRobot [2]. Dieses Shield verfügt über ein 16×2-LCD-Modul mit Hintergrundbeleuchtung (engl. *backlight*), 5 Funktionstasten und eine zusätzliche Reset-Taste.



LCD KeyPad Shield

Die Signalbelegung des LCD KeyPad Shield ist fest vorgegeben (siehe nachfolgende Tabelle). Sie stimmt nicht vollständig mit der beim bisherigen LCD-Modul verwendeten Adress- bzw. Signalbelegung überein.

Signalbelegung des LCD KeyPad Shields

Anschlüsse (Arduino)	*Signale am LCD-Modul*
4	D4
5	D5
6	D6
7	D7
8	RS
9	E (Enable)
10	Backlight (!)
A0	Funktionstasten

Zur softwareseitigen Ansteuerung ist die Bibliothek für die LCD-Ausgabe in der üblichen Weise einzubinden:

```
#include <LiquidCrystal.h>
```

Bei der Instanziierung ist das Objekt `lcd` mit den in der o.g. Tabelle angegebenen Adressen aufzurufen:

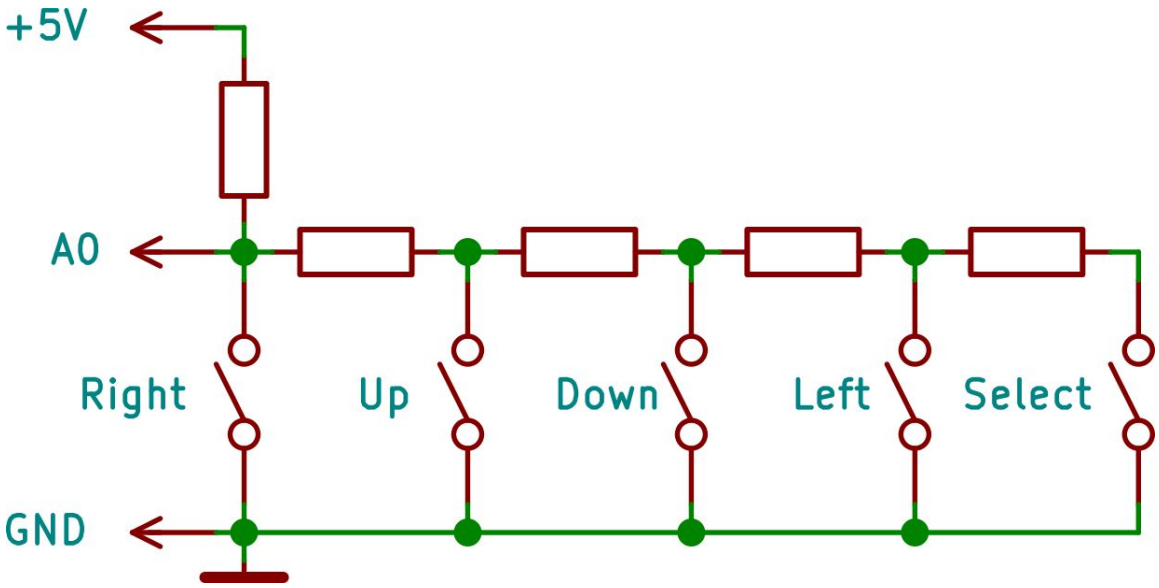
```
LiquidCrystal lcd(8, 9, 4, 5, 6, 7);
```

In der Funktion `setup` wird das LCD-Modul initialisiert. Dazu sind Zeilen- und Spaltenzahl der Anzeige zu übergeben:

```
void setup() {
  lcd.begin(16, 2);
}
```

Mit Pin 10 kann man prinzipiell die Hintergrundbeleuchtung ein- bzw. ausschalten. Soll die Hintergrundbeleuchtung immer aktiv sein, dann muss dieser Pin nicht konfiguriert werden, da auf dem Shield bereits ein passender Pull-Up-Widerstand vorgesehen ist.

Das Shield verfügt über 5 Funktionstasten, die mit dem analogen Kanal A0 entsprechend verbunden sind:



Verschaltung der Funktionstasten beim LCD KeyPad Shield

Einfache analoge Abfrage:

```
void loop() {
    int a = analogRead(A0);
    lcd.home();
    lcd.print("LCD KeyPad Shield");
    lcd.setCursor(0,1);
    lcd.print(a);
    lcd.print("   ");
}
```

Die entsprechenden Ausgabewerte des verwendeten Moduls sind der Tabelle zu entnehmen:

Abfragewerte der Taste des LCD KeyPad Shields

Taste	*ADC-Wert*	*Rückgabewert*
RIGHT (rechts)	0	0
UP (hoch)	100	1
DOWN (runter)	256	2
LEFT (links)	403	3
Select (Auswahl)	638	4
Keine Taste	1023	5

Die Werte 0 bis 4 sind für die verschiedenen Funktionstasten vorgesehen. Ist keine Taste gedrückt, so wird der Wert 5 zurückgegeben:

```
byte readKey() {
    int a = analogRead(A0);
    if (a>830) return 5;
    if (a>520) return 4;
    if (a>330) return 3;
    if (a>178) return 2;
    if (a> 50) return 1;
    return 0;
}
```

Mit dem folgenden Hauptprogramm lässt sich die Tastenabfrage überprüfen:

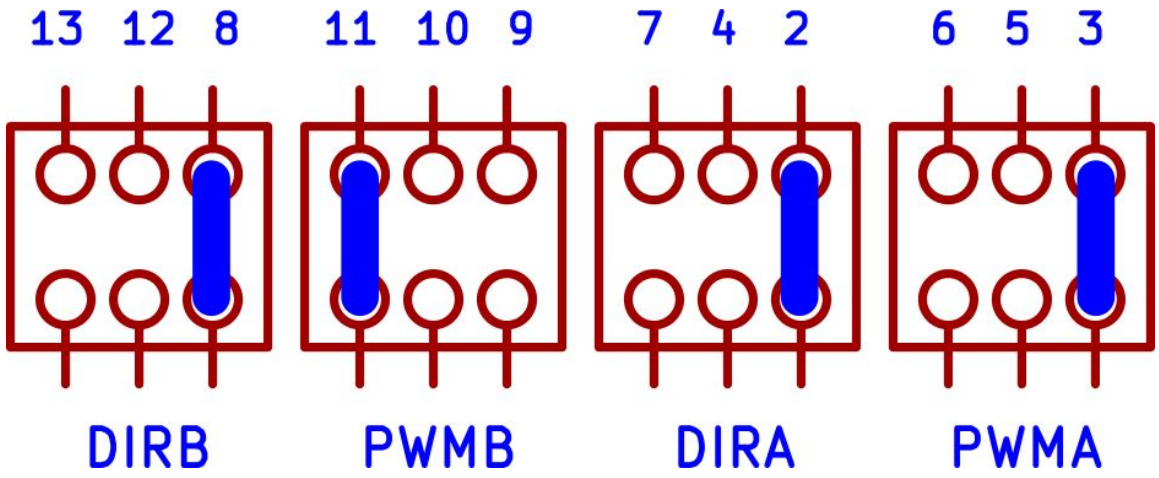
```
void loop() {
    byte a = readKey();
    lcd.home();
    lcd.print("LCD KeyPad Shield");
    lcd.setCursor(0,1);
    lcd.print("Taste: ");
    lcd.print(a);
}
```

Kombination mit dem Arduino-Motor-Shield R3

Die Signale RS und E des LCD KeyPad Shields belegen die gleichen digitalen Anschlüsse wie die Bremssignale (Brake A/B) des MotorShields. Das gleiche Problem tritt bei dem analogen Kanal A0 auf, der beim LCD KeyPad Shield zur Tastenabfrage und beim MotorShield zur Strommessung von Kanal A vorgesehen ist. Die betreffenden Signale sind auf der Rückseite des Motor-Shields als Lötbrücken ausgeführt und können leicht durchtrennt werden. Mit dieser Maßnahme kann man beide Shields gemeinsam betreiben.

Kombination mit dem Motor-Shield von Velleman

Die Signale PWMA und PWMB könnte man auf den Standard-Adressen 3 und 9 belassen. Da die digitalen Kanäle 4 und 7 vom LCD KeyPad Shield belegt sind, bleibt für DIRA nur Pin 2 übrig. In ähnlicher Wiese steht für DIRB nur der Pin 11 zur Verfügung:

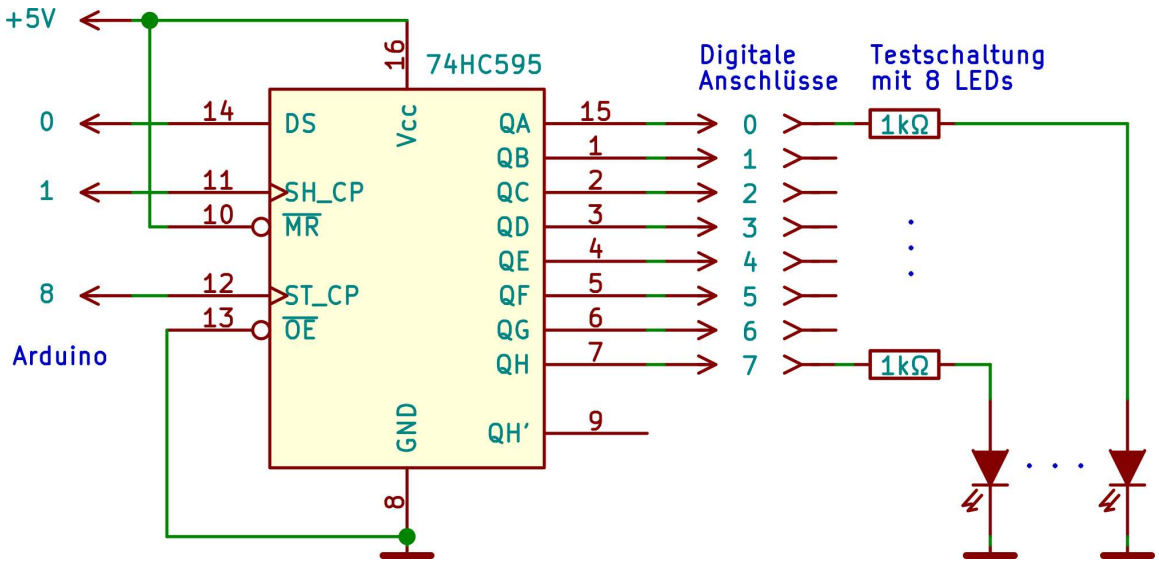


Adressauswahl des Motor-Shields KA03 von Velleman für den Betrieb mit dem LCD KeyPad Shield

12.2 Bereitstellung digitaler Ausgänge mit Schieberegister 74HC595

Mit Motor-Shield und LCD-Modul sind fast alle digitale Kanäle der Arduino-Standard-Boards belegt. Mit Hilfe eines Schieberegisters kann man weitere Ausgabekanäle bereitsstellen. In diesem Abschnitt kommt

dafür das 8-Bit-Schieberegister 74HC595 zum Einsatz [3]. Für die Ansteuerung des Schaltkreises werden drei digitale Kanäle benötigt. Die folgende Abbildung zeigt die Verdrahtung mit dem Arduino-Board sowie eine Testschaltung mit 8 LEDs.



Anschluss des Schieberegisters 74HC595 mit zusätzlicher Testschaltung

Die seriellen Daten werden über den digitalen Kanal 0 übertragen (DS, serial data input). Zusätzlich gibt es zwei Taktsignale, die über die digitalen Kanälen 1 und 8 angesteuert werden. Das Signal SH_CP (shift register clock input) ist ist für die bitweise serielle Übertragung zuständig, das Signal ST_CP (storage register clock input) für die Übernahme der übertragenen 8 Bits in den Ausgabepuffer. Für die softwareseitige Anbindung des Schieberegisters definieren wir diese Pins als Konstante:

```
byte latchPin = 8;  // ST_CP
byte clockPin = 1;  // SH_CP
byte dataPin  = 0;  // DS
```

In der Routine `setup` sind diese Pins als Ausgabekanäle zu deklarieren:

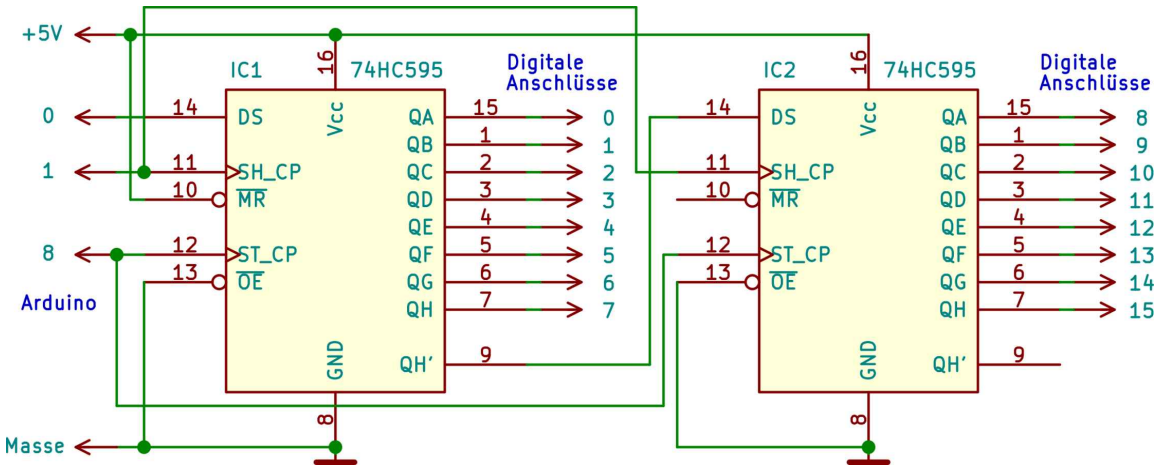
```
void setup() {
    pinMode(latchPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
```

```
    pinMode(dataPin,   OUTPUT);
}
```

Zur Ansteuerung des Schieberegisters steht in der Arduino-Umgebung die Funktion ShiftOut zur Verfügung [1]. Beim Aufruf dieser Funktion übergibt man zunächst digitalen Kanäle (Pins) für den Daten und (bitweisen) Takt. Das dritte Funktionsargument legt fest, in welcher Reihenfolge die Bits rausgeschoben werden. Dazu stehen die Optionen `MSBFIRST` (engl. *most significant bit first*) oder `LSBFIRST` (engl. *least significant bit first*) zur Verfügung. Das vierte Argument sind die auszugebenden Daten als 8-Bit-Wert (Datentyp `byte` bzw. `uint8_t`). Die Übernahme des 8-Bit-Wertes in den Ausgabepuffer des Schieberegisters erfolgt über eine LOW-HIGH-Flanke des Signals ST_CP. Das bedeutet, dass man das dieses Signal über den entsprechen Pin (`latchPin`) vor der bitweisen Ausgabe mit `ShiftOut` auf LOW und danach auf HIGH setzt. Für einen Testbetrieb mit den in o.g. Abbildung gezeigten 8 LEDs wurde nachfolgend ein Lauflicht implementiert:

```
void loop() {
    for (int i = 0; i<8; i++) {
        byte x=0;
        bitSet(x,i);
        digitalWrite(latchPin, LOW);
        shiftOut(dataPin, clockPin, MSBFIRST, x);
        digitalWrite(latchPin, HIGH);
        delay(100);
    }
}
```

Die Schieberegister 74HC595 kann man auch kaskadieren und damit mehr als 8 Bit ausgeben. Die nächste Abbildung zeigt den Anschluss von zwei Schieberegistern zur Bereitstellung von 16 zusätzlichen digitalen Ausgängen. Der Takt für die bitweise Übertragung (Signal SH_CP) liegt an beiden Schaltkreisen an. Der erste Schaltkreis erhält seine Daten direkt vom Mikrocontroller, der zweite Schaltkreis aus den Überlauf (Signal QH') des ersten.



Anschluss zweier Schieberegister 74HC595

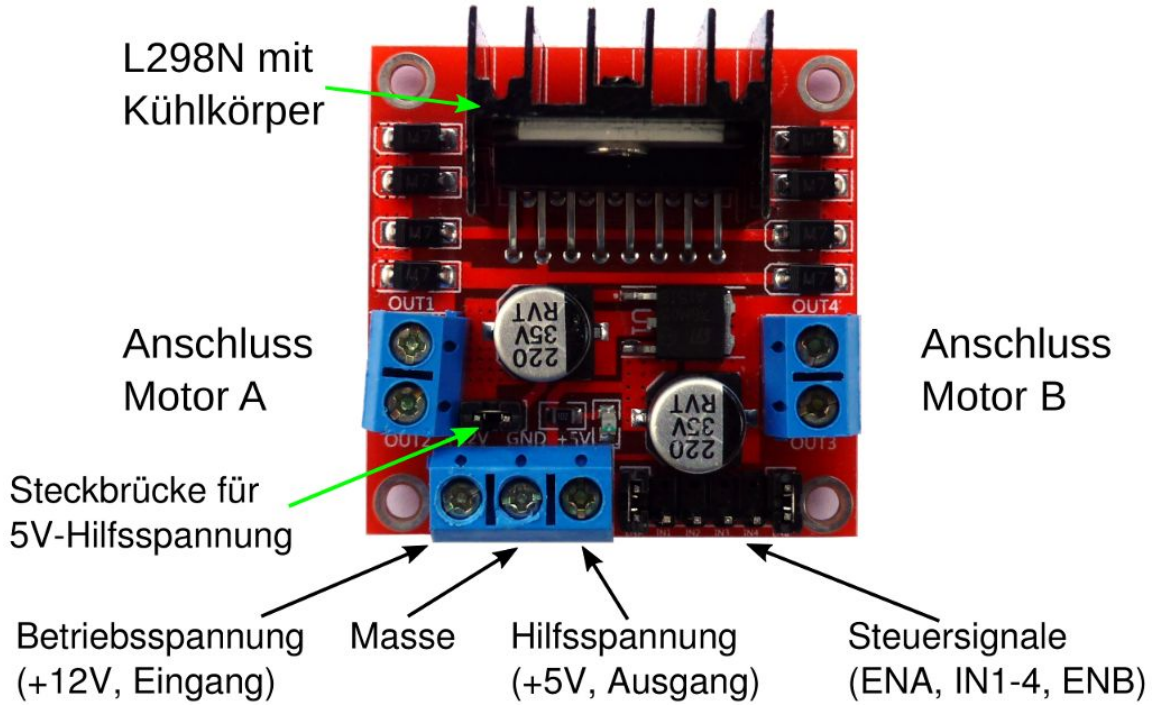
Die Funktion `ShiftOut` ist (vorerst) nur für die 8-Bit-Ausgabe vorgesehen. Ein 16-Bit-Wert (Typ `unsigned int` bzw. `uint16_t`) ist daher in zwei Schritten auszugeben, wobei die höherwertigen 8 Bits für die Ausgabe mit `ShiftOut` in den niederwertigeren Teil zu verschieben sind. Diese Vorgehensweise wird in der folgenden Hauptschleife für ein 16-Bit-Lauflicht illustriert. Dabei wären die Ausgänge der beiden Schieberegister in der üblichen Weise mit LEDs zu beschalten.

```
void loop() {
    for (int i = 0; i<16; i++) {
        unsigned int x=0;
        bitSet(x,i);
        digitalWrite(latchPin, LOW);
        shiftOut(dataPin, clockPin, MSBFIRST, x>>8);
        shiftOut(dataPin, clockPin, MSBFIRST, x);
        digitalWrite(latchPin, HIGH);
        delay(100);
    }
}
```

12.3 Motortreiber mit L298N

Neben dedizierten Motortreibern für die Arduino-Umgebung (den sog. *Motor-Shields*) gibt es auch Baugruppen zur Motoransteuerung, die für beliebige Mikrocontroller-Plattformen vorgesehen sind. Die nachfolgende

Abbildung zeigt einen solchen Motortreiber. Derartige Baugruppen sind auch in ähnlichen Ausführungen erhältlich. Kernstück der Leiterplatte ist ein Treiberschaltkreis der L298-Familie. Im Unterschied zu der bei beiden Motor-Shields (Arduino-Motor-Shield und Velleman-Motor-Shield) eingesetzten SMD-Variante L298P kommt hier die Multiwatt-Ausführung L298N zum Einsatz. Der Eingang für die zur Motoransteuerung verwendeten Betriebsspannung ist mit +12V bezeichnet. Der Treiberschaltkreis L298 ist für eine Betriebsspannung von maximal 46V vorgesehen, die Treiberbaugruppe allerdings nur bis 35V. Bei einer Betriebsspannung von bis zu 12V kann man mit der Baugruppe auch eine +5V-Hilfsspannung bereitstellen, die sich beispielsweise zur Versorgung von TTL-Logik-Schaltkreisen einsetzen lässt. Bei einer höheren Versorgungsspannung für die Motoren ist die entsprechende Steckbrücke zu entfernen.



Motortreiberbaugruppe mit L298N

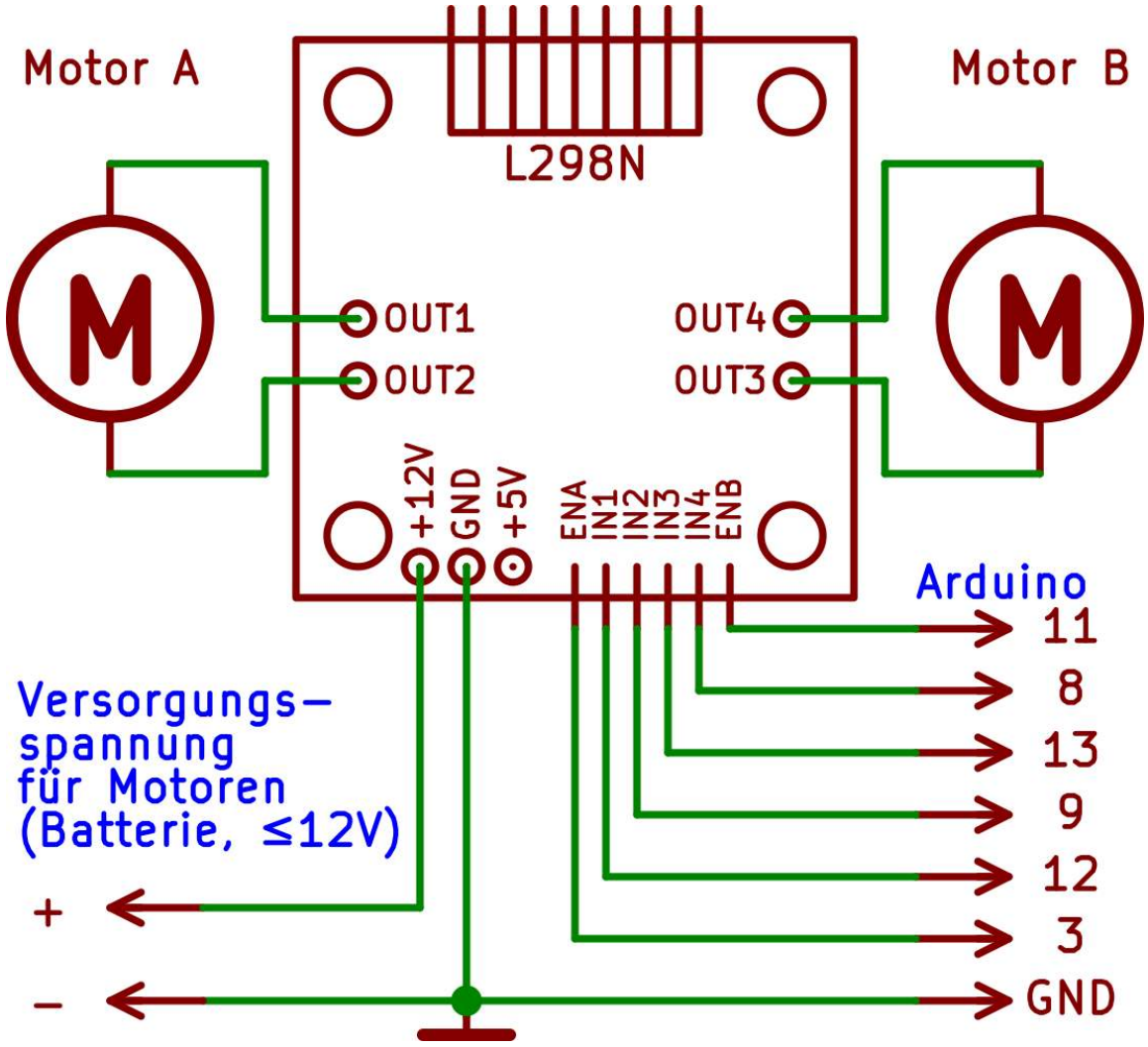
Bei der abgebildeten Variante der Motortreiberbaugruppe sind die Enable-Anschlüsse ENA und ENB mit Steckbrücken belegt. Für eine variable Geschwindigkeitseinstellung sind diese zwei Steckbrücken zu entfernen und die Anschlüsse ENA und ENB mit PWM-Signalen anzusteuern. Die Dreh-

richtung des einen Kanals (Motor A) wird über die Anschlüsse IN1 und IN2 eingestellt, die des anderen Kanals (Motor B) über IN3 und IN4. Für eine Drehbewegung des Motors müssen die Anschlüsse IN1, IN2 bzw. IN3, IN4 jeweils unterschiedlichen Pegel aufweisen. Bei gleichem Pegel befinden sich die Motoren entweder im Freilauf (Enable auf LOW) oder werden aktiv gebremst (Enable auf HIGH).

Anschlussbelegung der L298N-Motortreiberbaugruppe

Funktion bzw. Signal	*Kanal A*	*Kanal B*
PWM	ENA	ENB
Richtung (DIR)	IN1	IN3
Komplementäre Richtung	IN2	IN4

Die nächste Abbildung zeigt die Verdrahtung der Motortreiberbaugruppe. Die verwendete Pinbelegung stimmt weitestgehend mit der des Arduino-Motor-Shields überein. Für die komplementären Richtungssignale wurden die sonst für die Bremsfunktion vorgesehenen digitalen Anschlüsse verwendet.



Anschluss der separaten Motortreiberbaugruppe

Diese Pinbelegung würde man folgendermaßen im Programm verankern:

```
const byte PWMA =  3;
const byte PWMB = 11;
const byte DIRA = 12;
const byte DIRB = 13;
const byte DIRA2 = 9;
const byte DIRB2 = 8;
```

In der Funktion `setup` sind diese 6 digitalen Anschlüsse als Ausgänge zu definieren:

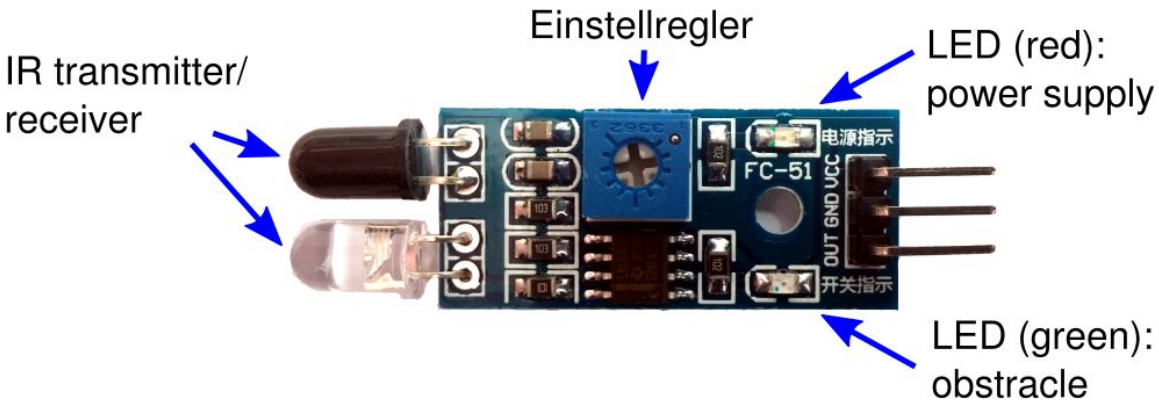
```
void setup()
{
    pinMode(PWMA,  OUTPUT);
    pinMode(PWMB,  OUTPUT);
    pinMode(DIRA,  OUTPUT);
    pinMode(DIRB,  OUTPUT);
    pinMode(DIRA2, OUTPUT);
    pinMode(DIRB2, OUTPUT);
}
```

In der Hauptschleife könnte die Ansteuerung der Motoren für halbe Geschwindigkeit folgendermaßen aussehen:

```
void loop()
{
    // Drehrichtung
    digitalWrite (DIRA,  LOW);
    digitalWrite (DIRA2, HIGH);
    digitalWrite (DIRB,  LOW);
    digitalWrite (DIRB2, HIGH);
    // halbe Geschwindigkeit
    analogWrite (PWMA, 127);
    analogWrite (PWMB, 127);
}
```

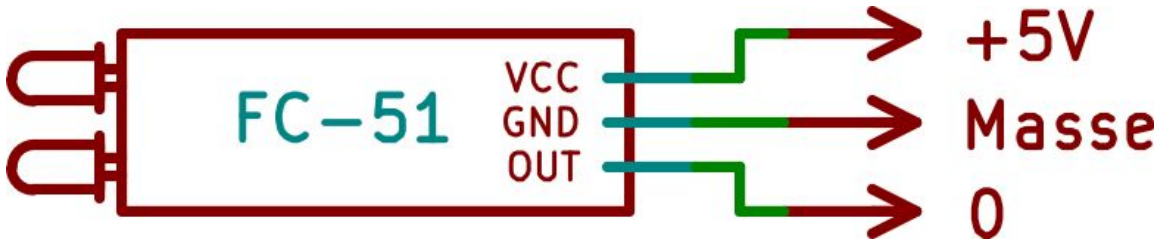
12.4 Hinderniserkennung mit Infrarotsensoren

Oft ist man nicht an einer präzisen Entfernungsangabe interessiert, sondern möchte einfach Hindernisse rechtzeitig erkennen, um ein geeignetes Ausweichmanöver einzuleiten. Für eine derartige Abfrage lässt sich das Sensormodul FC-51 verwenden:



Infrarotsensor FC-51

Das Sensormodul verfügt über drei Anschlüsse: Masse (GND), Betriebsspannung (VCC) und einen digitalen Signalausgang (OUT). Bei korrekt bereitgestellter Stromversorgung leuchtet die rote LED auf dem Sensormodul. Ein Hindernis wird durch LOW-Pegel am Ausgang OUT angezeigt. Zusätzlich leuchtet dann die grüne LED auf der Sensorplatine. Die gewünschte Entfernung, ab der ein Hindernis erkannt wird, lässt sich über den auf der Leiterplatte befindlichen Einstellregler vorgeben.



Anschluss eines Infrarotsensors FC-51

In der Regel wird man den Infrarotsensor in der Mitte der Vorderseite des mobilen Roboters anbringen. Für eine Hindernisumfahrung muss man nur das bereits für die Verwendung mit einem Sharp-Entfernungssensor angegebene Programm leicht modifizieren. Der für den Anschluss des Infrarotsensors verwendete Pin wird durch die Konstante `pinInfra` definiert. Für das Abfrageergebnis legen wir außerdem die Variable `sensor` an:

```
// Pin für Infrarotsensor
const byte pinInfra = 0;
```

```
byte sensor;
```

Die Hauptschleife wird hinsichtlich der Sensorabfrage marginal modifiziert. Bei einem HIGH-Pegel am Sensorausgang fährt der Roboter geradeaus, bei einem durch LOW-Pegel angezeigten Hindernis wird das Ausweichmanöver eingeleitet:

```
void loop() {
    sensor = digitalRead(pinInfra);
    if (sensor == HIGH) {
        // Vorwärtsfahrt
        motor.forward(); }
    else {
        // Rückwärts fahren
        motor.backward();
        delay(T_rueck);
        // Rechtsdrehung
        motor.right();
        delay(T_dreh); }
    delay(100);
}
```

12.5 Quellenangabe

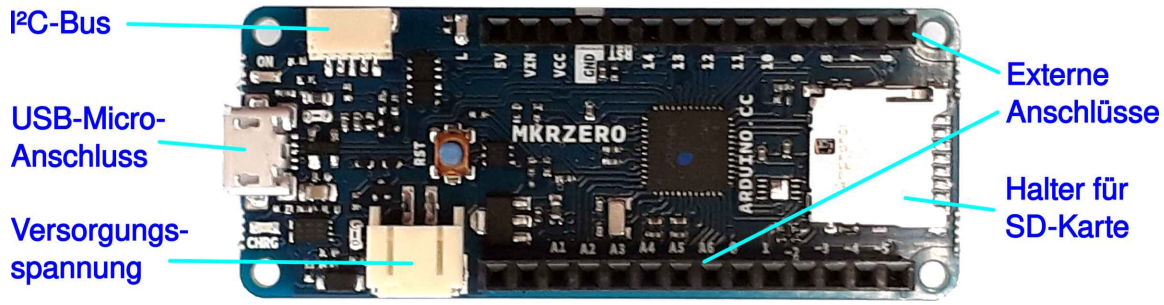
1. *Arduino — ShiftOut*. URL: https://www.arduino.cc/en/tutorial/ShiftOut
2. DFROBOT. URL: https://www.dfrobot.com/
3. Philips Semiconductors: *74HC/HCT595 8-bit serial-in/serial or parallel-out shift register with output latches; 3-state*. Juni 1998.

13 Arduino Boards mit ARM-Architektur

13.1 ARM-Boards

Bei den Arduino-Standard-Boards mit 8-Bit Mikrocontrollern der ATmega-Serie stößt man hinsichtlich Rechenleitung und Speicherplatz schnell an Grenzen. Abhilfe schafft der Einsatz neuerer Boards auf Basis der ARM®-Architektur. Dabei wären zuerst die Boards Arduino Zero, Arduino M0 bzw. M0 PRO zu nennen, welche über den gleichen Formfaktor wie Arduino Uno bzw. Leonardo verfügen. Nach Einschätzung des Autors erlangten diese Boards allerdings keine große Verbreitung. Für Anwendungen, bei denen sehr viele Anschlüsse anzusteuern sind, bietet sich das Board Arduino Due an, welches man als Weiterentwicklung des Arduino Mega Boards auffassen kann.

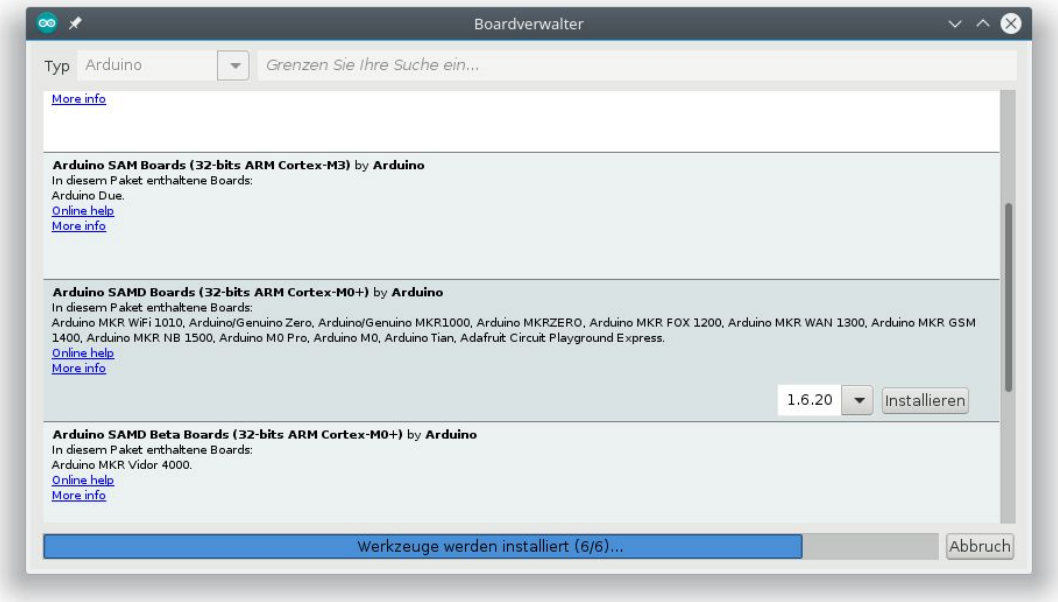
Für viele Anwendungen dürfte die neue MRK-Serie interessant sein. Diese Boards weisen einen kleineren Formfaktor auf und sind in zahlreichen Varianten (z.B. mit Wi-Fi, Lo-Ra, GSM) verfügbar. Ein Board innerhalb dieser Serie ist das Arduino MKR Zero, welches sich hinsichtlich der Funktionalität an das Arduino Zero Board anlehnt, zusätzlich aber über einen Adapter für MicroSD-Karten verfügt.



Arduino MKR Zero

13.2 Inbetriebnahme und Software

Die Arduino-IDE ist zunächst nur für Boards der AVR-Architektur vorgesehen. Die entsprechenden Tools für die ARM-Architektur müssen nachinstalliert werden. Dazu geht man auf **Werkzeuge > Board > Boardverwalter**. Unter dem Menüpunkt **Typ** kann man die Auswahl der Board-Architekturen einschränken, z.B. auf **Arduino**. Dann wählt man das zum Board gehörende Paket aus, beispielsweise Arduino SAMD Boards (32-bits ARM Cortex-M0+ by Arduino) für ein Arduino MKR ZERO Board und geht auf **Installieren**. Zur manuellen Einstellung des Boards würde man beispielsweise über **Werkzeuge > Boards > Arduino MKRZERO** das Board auswählen und die serielle Schnittstelle unter **Werkzeuge > Port** einstellen. Für die Programmierung wählt man **Werkzeuge > Programmer > Arduino as ISP** aus. Zum Test kann man auch hier das Beispielprogramm `Blink` verwenden.



Auswahl der ARM-Archtiktur im Boardverwalter

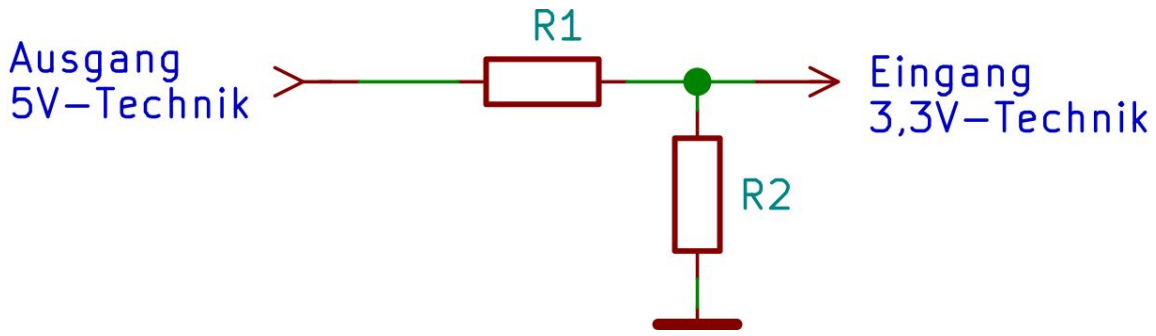
Die Arduino-Boards mit ARM-Architektur werden praktisch genauso wie die Boards mit AVR-Architektur programmiert. Besonderheiten ergeben sich für die analoge Ein- und Ausgabe. Während die AVR-Boards (z.B. Arduino Uno oder Leonardo) 10 Bit-ADCs aufweisen, verfügen die ARM-Boards über 12 Bit-ADCs. Die Auflösung für die analoge Eingabe kann bei den ARM-Boards mit dem Funtionsaufruf `analogReadResolution(n)` auf n Bit gesetzt werden. Eine ähnliche Situation liegt bei der digitalen Ausgabe vor, die bei den AVR-

Boards über eine 8 Bit-PWM erfolgt. Die ARM-Boards besitzen auch Ausgänge mit 10 bzw. 12 Bit Auflösung. Die Anpassung ist über den Funktionsaufruf `analogWriteResolution(n)` möglich.

13.3 Pegelanpassung

Die Arduino-ARM-Boards verwenden für die Ein- und Ausgabe einen Signalpegel von 3,3V anstelle von 5V. Bei Baugruppen, die ebenfalls 3,3V-Pegel nutzen (z.B. das behandelte Bluetooth-Modul HC-05), ist dann keine Pegelanpassung nötig. Auf eine Pegelanpassung kann man ebenfalls verzichten, wenn man mit einem 3,3V-Ausgang eines ARM-Boards einen für 5V ausgelegten CMOS-Eingang ansteuert. Dabei wird allerdings vorausgesetzt, dass der Eingang der anzuschließenden 5V-Baugruppe nicht über einen Pull-Up-Widerstand mit 5V verbunden ist.

Bei der Abfrage eines 5V-Ausgangs mit einem 3,3V-Eingang ist die Spannung zu reduzieren. Diese Reduktion lässt sich leicht mit Hilfe eines Spannungsteilers realisieren:



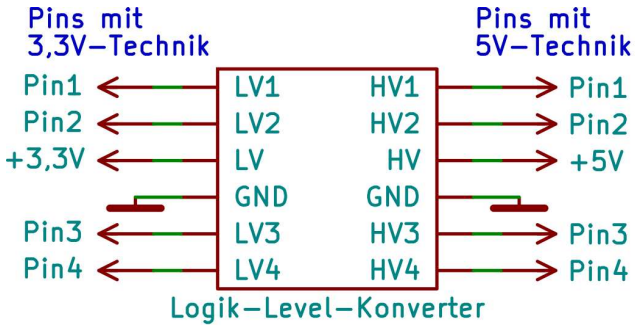
Pegelanpassung von 5V auf 3,3V

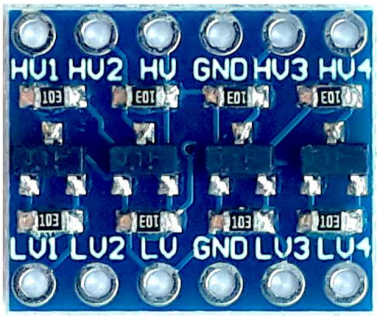
Für die gewünschte Pegelanpassung müssen die Widerstände des Spannungsteilers der Beziehung

$$R_2 / (R_1 + R_2) \approx 3{,}3\text{V} / 5\text{V}$$

genügen (z.B. $R_1 = 10\ \text{k}\Omega$ und $R_2 = 18\ \text{k}\Omega$).

Auf Basis aktiver Schaltungen ist auch eine bidirektionale Pegelanpassung möglich. Die Pegelanpassung ist dabei in beide Richtungen möglich. Derartige Schaltungen zur Pegelanpassung werden auch als Baugruppe für 4-Kanäle angeboten:



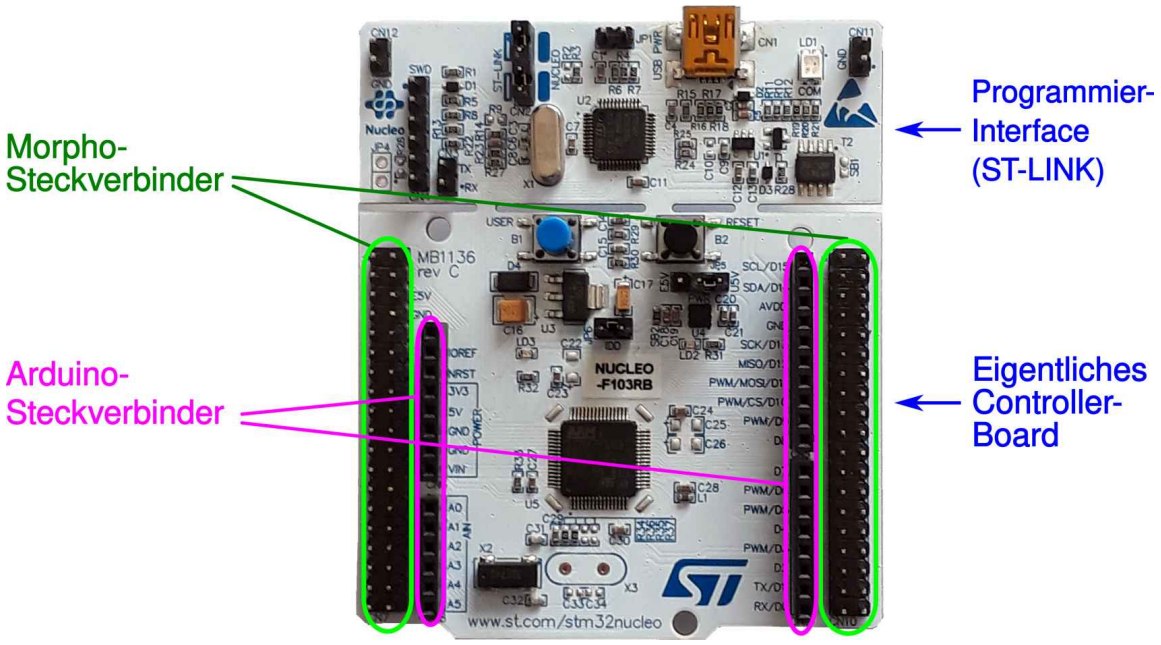


Bidirektionale Pegelanpassung mit 4-Kanal-Modul

Bei der Beschaltung steht LV für *low voltage* (3,3V) und HV für *high voltage* (5V).

13.4 Nucleo-Boards mit STM32-Controllern

Die Boards der Nucleo-Reihe mit STM32-Controllern von STMicroelectronics stellen eine preiswerte Alternative zu den Arduino-Boards dar. Die nachfolgende Abbildung zeigt die grundsätzliche Struktur der Nucleo-64-Boards am Beispiel des Boards NUCLEO-F103RB.



Struktur der Nucleo-64-Boards

Die Nucleo-Boards lassen sich grundsätzlich auch mit der Arduino-IDE programmieren. Der Betrieb der STM32-Boards innerhalb der Arduino-Umgebung wird über das STM32duino-Projekt ermöglicht [1]. Die Installation der erforderlichen Zusatzsoftware und die Inbetriebnahme eines derartigen Boards ist in [2] beschrieben.

Die Besonderheit der Nucleo-64-Boards besteht darin, über weitgehend zu Arduino kompatible Steckverbinder zu verfügen. Dabei sind auch etliche Pins für den Betrieb mit einem 5 V-Signalpegel ausgelegt. Zahlreiche weitere Pins sind über den Morpho-Steckverbinder zugänglich [3]. Neben den Boards der Nucleo-64-Reihe gibt es auch kleinere Nucleo-32-Boards und größere Nucleo-144-Boards.

13.5 Quellenangabe

1. STM32duino. URL: https://github.com/stm32duino
2. K. Röbenack: *STM32-Einstieg mit Nucleo-64-Board und Arduino-Umgebung*. Independently published, 2019. ISBN: 978-1674318387.
3. STMicroelectronics: *STM-Nucleo64 boards (MB1136)*. User manual UM 1724, 2019.

Anhang: Weitere Bücher des Autors

STM32-Einstieg mit Nucleo-64-Board und Arduino-Umgebung

Das Ardunio-Konzept ermöglicht den schnellen Einstieg in die Mikrocontrollertechnik. Die Standardboards (Arduino Uno, Arduino Leonardo usw.) basieren auf 8-Bit-Mikrocontrollern der AVR-Reihe von Atmel und sind dadurch hinsichtlich Rechenleistung und Speicherplatz für komplexere Anwendungen nur bedingt geeignet.

Röbenack, K.: *STM32-Einstieg mit Nucleo-64-Board und Arduino-Umgebung.*
Independently published, Dezember 2019 (ISBN: 978-1674318387). Auch als Kindle-Edition verfügbar.

Mikrocontroller mit ARM-Kern bieten in der Regel eine wesentlich höhere Rechenleistung. Die ARM-Architektur wird auch von neueren Arduino unterstützt (z.B. Arduino Zero, Due und die Arduino MRK1000-Reihe). Eine sehr preiswerte Alternative stellen die Boards der Nucleo-Reihe mit STM32-Controllern von STMicroelectronics dar. Dieses Buch bezieht sich maßgeblich auf die Nucleo-64-Boards, die auch Steckverbinder mit einem

zum Arduino Uno kompatibeln Formfaktor enthalten. Damit lassen sich Shield auch au dieser sehr leistungsfähigen Plattform nutzen. Für die im Buch beschriebenen Experimente wurde das Board NUCLEO-F103RB eingesetzt, welches zum Zeitpunkt der Manuskripterstellung für knapp 15€ verfügbar war.

Die Programmierung der STM32-Mikrocontroller ist trotz verfügbarer Tools sehr anspruchsvoll. Die Arduino-Umgebung bietet die Möglichkeit für einen schnellen Einstieg in die STM32-Technik, auch wenn man damit vielleicht nicht das volle Potential der Mikrocontroller ausschöpft.

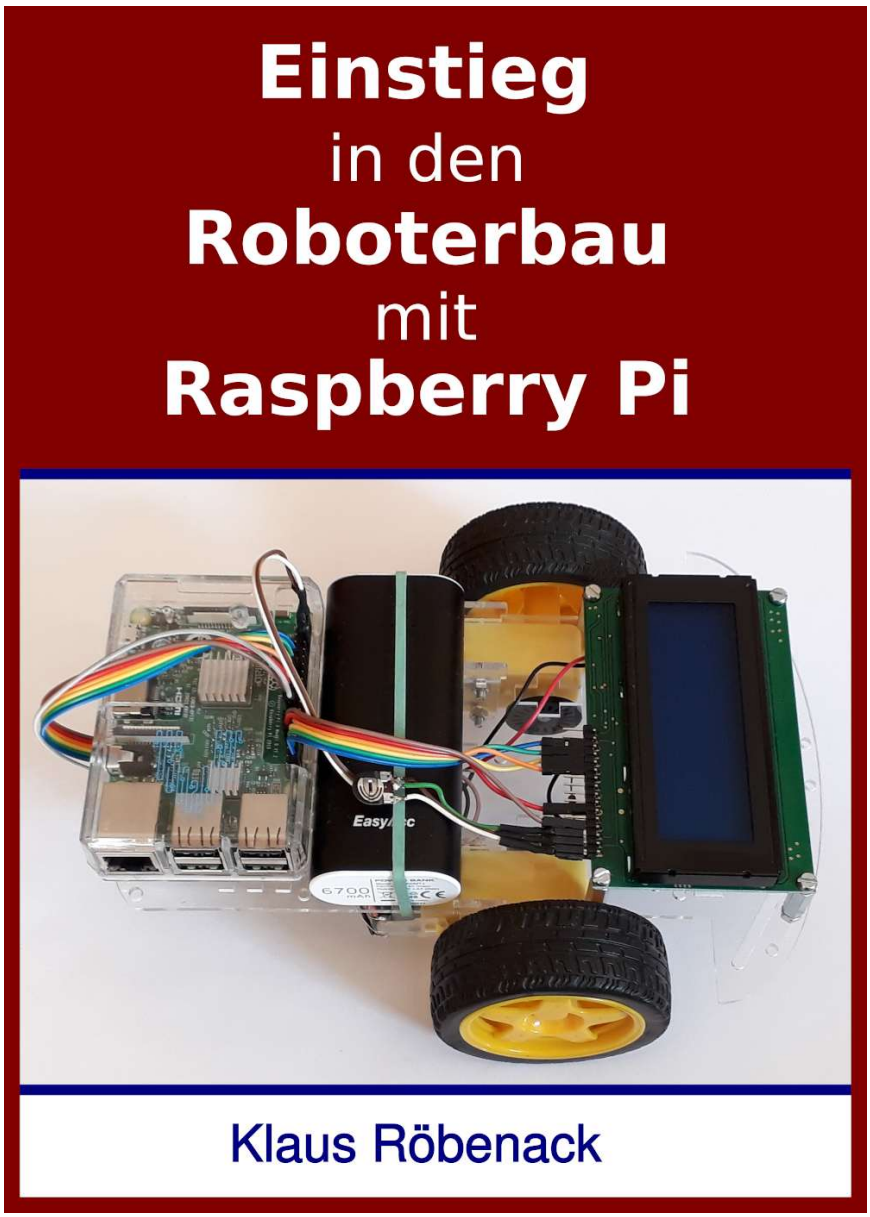
Dieses Buch widmet sich dem Einstieg in die STM32-Technik unter Nutzung der Arduino-Umgebung. Es richtet sich an Leser, die bereits über erste Erfahrungen mit der klassischen Arduino-Plattform verfügen. Zusätzlich werden geringe schaltungstechnische Grundkenntnisse erwartet sowie die Fähigkeit, einfache Programme in C bzw. C++ zu verstehen bzw. zu erstellen.

Bei der Programmierung von Mikrocontrollern geht es fast immer um Konfiguration und Betrieb angeschlossener externer Hardware-Komponenten, z.B. von Anzeigen. Dabei wird in diesem Buch sowohl auf die elementare Ansteuerung als auch die Nutzung gängiger Bibliotheken eingegangen.

Einstieg in den Roboterbau mit Raspberry Pi

Dieses Buch beschreibt erste Schritte auf dem Weg zu einem mobilen Roboter, welcher von einem Raspberry Pi gesteuert wird. Dabei werden geringe schaltungstechnische Grundkenntnisse vorausgesetzt. Die Programmierung des Roboters erfolgt in Python.

Nach einigen Hinweisen zur Inbetriebnahme des Raspberry Pi werden die Ansteuerung bzw. die Abfrage externer Hardware an einigen einfachen Beispielen illustriert. Im Zusammenhang mit dem Roboter werden die Ansteuerung der Motoren mit Pulsweitenmodulation und die Textausgabe auf einem LCD-Modul behandelt. Außerdem wird die Fernsteuerung des Roboters mit einem Notebook über WLAN beschrieben.



Röbenack, K.: *Einstieg in den Roboterbau mit Raspberry Pi*.
Kindle Edition, April 2020.

Audioverstärker mit Röhrenvorstufe - Einfache Schaltungen zum Selberbauen

Vollständig mit Röhren bestückte Verstärker sind sowohl schaltungstechnisch als auch mechanisch sehr aufwendig. Ein konzeptionell wie technisch interessanter Kompromiss ist die Verknüpfung einer Röhrenvorstufe mit einer transistorisierten bzw. integrierten Endstufe. Dieser Ansatz, der auch bei etlichen kommerziellen Geräten umgesetzt wurde, bildet den Schwerpunkt dieses Buches. Die beschriebenen Röhrenvorstufen können natürlich auch leicht mit anderen Verstärkern kombiniert werden.

Röbenack, K.: *Audioverstärker mit Röhrenvorstufe - Einfache Schaltungen zum Selberbauen.* Printed by CreateSpace, Juli 2014 (ISBN: 978-1500482206). Auch als Kindle-Edition verfügbar.

Dieses Buch wendet sich an Leser, die bereits über schaltungstechnische Grundkenntnisse verfügen, sich aber nicht zwangsläufig mit Röhren auskennen müssen. Dazu wird im Buch die Funktionsweise der gängigen Verstärkerröhren (Trioden und Pentoden) erläutert. Der Autor geht außer-

dem auf Verstärkergrundschaltungen (Kathodenbasis- und Anodenbasis-schaltung) ein.

Neben mehreren erprobten Verstärkerschaltungen finden sich im Buch auch Hinweise für eigene Experimente. In Ergänzung zu verschiedenen funktionsfähigen Grundschaltungen werden ebenso zahlreiche Erweiterungsmöglichkeiten diskutiert.

Radiobasteln mit Elektronenröhren - Detektor-empfänger und Audionschaltungen

In der ersten Hälfte des letzten Jahrhunderts waren Elektronenröhren die bestimmenden Bauteile der Elektronik und haben Rundfunk- bzw. Fernseh-technik geprägt. Obwohl Röhren in nahezu allen Anwendungsbereichen durch Transistoren verdrängt wurden, üben Elektronenröhren auch heute noch eine große Faszination auf Bastler und Anwender aus. Mit dem vorliegenden Buch wird Bastelfreunden die Möglichkeit geben, sich mit einfachen Radio- bzw. Röhrenschaltungen vertraut zu machen. Das Buch richtet sich an Leser, die bereits über Grundkenntnisse der Elektrotechnik und erste Erfahrungen mit elektronischen Schaltungen verfügen.

Röbenack, K.: *Radiobasteln mit Elektronenröhren - Detektorempfänger und Audionschaltungen*. Shaker Verlag, Aachen, 2013 (ISBN: 978-3-8440-1864-6).

In ihrer Schlichtheit fördern Röhrenschaltungen das Verständnis für wichtige schaltungstechnische Problemstellungen. Durch die Kombination von Röhren mit moderneren Halbleiterbauelementen sind schnell vorzeigbare Erfolge zu erzielen. Am Ende eines Bastelprojekts kann aber auch ein vollständig mit Röhren bestücktes Radio stehen.

Im ersten Kapitel wird der Leser mit den nötigsten Grundlagen der Röhrentechnik vertraut gemacht. In diesem Zusammenhang wird die Funktionsweise der wichtigsten Röhrentypen (Dioden, Trioden, Pentoden und Heptoden) beschrieben und an einfachen Schaltungsbeispielen erläutert. Die für die Signalverstärkung relevanten Kenngrößen werden an konkreten Rechenbeispielen erörtert.

Kapitel zwei und drei bilden den Hauptteil des Buches und behandeln verschiedene Geradeausempfänger für den amplitudenmodulierten Rundfunk. Die beschriebenen Radioschaltungen sind für den Mittelwellenempfang konzipiert, können aber weitestgehend ohne Probleme für den Langwellen- bzw. Kurzwellenbereich modifiziert werden.

Das zweite Kapitel ist Detektorempfängern gewidmet. Während die ersten Detektorschaltungen zunächst noch auf Halbleiterdioden, Transistoren bzw. einem Schaltkreis zurückgreifen, werden diese Bauteile Schritt für Schritt durch Röhren ersetzt. Dabei ist der Leser nicht starr an eine feste Bauanleitung gebunden, sondern kann in jedem Stadium zwischen verschiedenen Schaltungsvarianten bzw. Röhrentypen wählen.

Das dritte Kapitel befasst sich mit Audionschaltungen. Bei diesen Empfängerschaltungen, die in der ersten Hälfte des vergangenen Jahrhunderts üblich waren, übernimmt eine Röhre gleichzeitig Demodulation und Verstärkung. Dieses Empfangsprinzip wird an verschiedenen Röhrentypen illustriert. Neben dem Einsatz von konventionellen Radio- bzw. Fernsehröhren werden auch Schaltungen mit Batterieröhren vorgestellt.

Die Sockelschaltbilder der verwendeten Röhren sind im Anhang aufgeführt. Ein umfangreiches Literaturverzeichnis rundet das Buch ab.

Buchbesprechungen, Rezensionen:

- Funkamateur (Das Magazin für Amateurfunk, Elektronik und Funktechnik) 7/2013, S. 708
- Elektor Special Project Röhren 10, 2014, S. 85-86.
- Make: Magazin 5/2015, S. 140

Nichtlineare Regelungssysteme: Theorie und Anwendung der exakten Linearisierung

Das Buch behandelt fortgeschrittene Methoden der nichtlinearen Regelungstheorie. Die Darstellung ist einerseits in sich mathematisch schlüssig und nachvollziehbar, andererseits aber auch in einer für Ingenieure verständlichen Sprache formuliert. Die jeweilige Herangehensweise bzw. Entwurfsmethodik wird an verschiedenen Beispielen veranschaulicht bzw. durch den Einsatz des Open-Source-Computeralgebrasystems Maxima illustriert.

Röbenack, K.: *Nichtlineare Regelungssysteme: Theorie und Anwendung der exakten Linearisierung*. Springer Vieweg, 2017 (ISBN-10: 3662440903, ISBN-13: 978-3662440902).

Das Werk wendet sich an Doktoranden bzw. Ingenieure in der Industrie, die - z.B. im Bereich Automotive oder in der chemischen Industrie - mit der Regelung nichtlinearer Systeme konfrontiert werden. Das Buch richtet sich ebenso an Studierende der Elektrotechnik oder Mechatronik in der

Vertiefungsrichtung Automatisierungs- bzw. Regelungstechnik, die bereits über fortgeschrittene regelungstechnische Kenntnisse verfügen.

Inhalt:

- Einleitung mit Einführungsbeispielen
- Mathematische Grundlagen
- Wichtige Begriffe der Differentialgeometrie
- Normalformen für den Reglerentwurf mittels exakter Linearisierung
- Reglerentwurf mit verschiedenen Modifikationen und Erweiterungen
- Beobachter mit großer Verstärkung (High-Gain-Beobachter)
- Beobachterentwurf mittels exakter Linearisierung
- Zusammenfassung