{BMU VERLAG}

Vielen Dank für Ihr Vertrauen in uns!

Hallo Frau/Herr Sigismund,

Sie haben sich für ein Fachbuch vom BMU Verlag entschieden, und dafür möchten wir uns bei Ihnen recht herzlich bedanken. Mit diesem Dokument erhalten Sie nun das kostenfreie eBook zu Ihrem Buch.

Sollten Sie Fragen oder Probleme haben können Sie sich jederzeit gerne an uns oder den Autor wenden. Und nun wünschen wir Ihnen viel Erfolg beim Lernen mit diesem Buch.

Ihr Team vom BMU Verlag

KOMPLETT IN FARBE

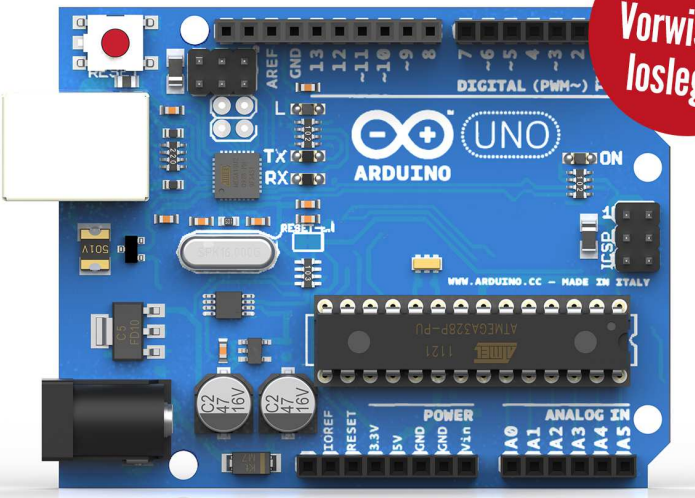
Andreas Sigismund asigismund@web.de

Danny Schreiter

ARDUINO KOMPENDIUM

ELEKTRONIK, PROGRAMMIERUNG UND PROJEKTE

Ohne Vorwissen loslegen



Die inoffizielle Anleitung zum Arduino!

- Grundlagen der Elektronik und des Programmierens verständlich erklärt
- Vorstellung vieler verschiedener Sensoren, Aktoren und Zubehörteile
- Umfangreiche und spannende Praxisprojekte zum Nachbauen

Inklusive eBook zum Download

{BMU VERLAG}

Andreas Sigismund asigismund@web.de

Arduino-Kompendium

Danny Schreiter

Andreas Sigismund asigismund@web.de

3. Auflage: Januar 2022

ISBN: 978-3-96645-039-3

Herausgegeben durch:
BMU Media GmbH
Koppenstraße 93
10243 Berlin

www.bmu-verlag.de
info@bmu-verlag.de

Coverfoto: CAD Rendering, Andrew Whitham
Lektorat: Markus Neumann
Druck: Wydawnictwo Poligraf

Andreas Sigismund asigismund@web.de

Arduino-Kompendium

Inhaltsverzeichnis

1. Einleitung 9

2. Geschichte 11

2.1 Die Geschichte von Mikrocontrollern ... 11
2.2 Entstehung der Arduino-Plattform ... 12
2.3 Überblick über verfügbare Hardware ... 15
2.4 Shields ... 19
2.5 Software-Überblick ... 20
2.6 Installation der Arduino-IDE ... 22
2.7 Bibliotheken ... 27

3. Hardware-Einführung 30

3.1 Grundlagen ... 30
3.2 Praktische Werkzeuge und Wissen ... 57

4. Grundlagen des Programmierens 69

4.1 Die Struktur eines Programmes ... 69
4.2 Blink ... 71
4.3 Konstanten ... 78
4.4 Bedingungen ... 79
4.5 Vergleichsoperatoren ... 81
4.6 Variablentypen ... 83
4.7 Schleifen ... 93
4.8 Ein- und Ausgabe am Bildschirm ... 97
4.9 Arrays ... 103
4.10 Zeiger ... 108
4.11 Funktionen ... 111
4.12 Objektorientierte Programmierung ... 115
4.13 Zeichenketten (Strings) ... 118

5. Ein- und Ausgänge 121

5.1 Digitale Ausgänge ... 121
5.2 Digitale Eingänge ... 125
5.3 Analoge Eingänge ... 130
5.4 Pulsweitenmodulation ... 134
5.5 Kommunikationsschnittstellen ... 138

6. Praxisprojekt: Modellbau-Ampel 162

6.1 Idee ... 162
6.2 Stromlaufplan ... 166
6.3 Versuchsaufbau ... 166
6.4 Programmcode ... 167

7. Anzeigeelemente 175

7.1 Leuchtdioden ... 175
7.2 RGB-LED ... 178
7.3 7-Segment-Anzeige ... 180
7.4 LED-Matrix ... 184
7.5 LCD ... 189
7.6 OLED-Display ... 195
7.7 Adressierbare LEDs ... 200

8. Praxisprojekt: Stoppuhr mit OLED-Display 208

8.1 Idee ... 208
8.2 Versuchsaufbau ... 208
8.3 Programmcode ... 209
8.4 Resultat ... 212

9. Sensoren und Eingabegeräte 215

9.1 Folientastatur ... 215
9.2 IR-Sensor/Fernbedienung ... 218
9.3 Fotowiderstand ... 222
9.4 Bewegungsmelder ... 224
9.5 Bodenfeuchte-Sensor ... 229
9.6 Temperatursensor ... 231
9.7 Ultraschall-Abstandssensor ... 235
9.8 Hall-Sensor ... 238
9.9 Beschleunigungssensor ... 242
9.10 Kompass ... 248
9.11 Echtzeitmodul ... 251

10. Praxisprojekt: LCD-Uhr mit Thermometer 258

10.1 Idee ... 258
10.2 Stromlaufplan ... 259
10.3 Versuchsaufbau ... 260
10.4 Programmcode ... 261
10.5 Resultat ... 267

11. Aktoren 269

11.1 Relais 269
11.2 Gleichstrommotor 273
11.3 Servomotor 278
11.4 Schrittmotor 282
11.5 Elektromagnet 288
11.6 Summer 291

12. Praxisprojekt: fernsteuerbares Auto 296

12.1 Idee 296
12.2 Stromlaufplan 297
12.3 Versuchsaufbau 299
12.4 Programmcode 303
12.5 Resultat 312

13. Datenverarbeitung 315

13.1 Permanente Speicher 315
13.2 Processing 330

14. Praxis-Projekt: LED-Matrix mittels Processing steuern 350

14.1 Idee 350
14.2 Konzeption 350
14.3 Versuchsaufbau 353
14.4 Programmcode (Arduino) 354
14.5 Programmcode (Processing) 358
14.6 Resultat 372

15. Objektorientierte Programmierung 377

15.1 Gründe für OOP 377
15.2 Klassen und Objekte 378
15.3 Klassendefinition 379
15.4 Beispiel: Schrittmotorsteuerung 386
15.5 Bibliotheken erstellen 399
15.6 Weitere Möglichkeiten 409

16. Arduino & Internet 410

16.1 Grundlagen 410
16.2 IoT-Webserver 418
16.3 MQTT 453

17. Arduino Clones, minimaler Arduino 493

17.1 Clones ... 493
17.2 Minimaler Arduino ... 495
17.3 In-System-Programmer ... 499

18. Erstellung eigener Platinen 507

18.1 Fritzing ... 508
18.2 EAGLE ... 517
18.3 Professionelle Platinenherstellung ... 525

19. Fehlersuche und Programmoptimierung 531

19.1 Fehler im Programmcode ... 531
19.2 Fehler außerhalb des Programmcodes ... 540
19.3 Speicheroptimierung ... 541
19.4 Zeitoptimierung ... 547

20. Der Anfang ist getan 552

21. Anhang: Verwendete Komponenten / Bezugsquellen 554

22. Anhang: Codereferenz 558

23. Anhang: Bildquellen 563

24. Anhang: Softwareverzeichnis 566

25. Anhang: Index 568

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 1
Einleitung

Hätte man vor vierzig Jahren Passanten auf der Straße gefragt, was ihr Leben mit Mikrocontrollern zu tun hat, hätten die meisten vermutlich nur verwundert mit den Schultern gezuckt. Zu dieser Zeit waren sich wohl auch nur wenige Menschen überhaupt über den Begriff im Klaren. Die sichtbare Welt funktionierte analog, und zwar ein bisschen auch deshalb, weil kaum jemandem der Begriff „digital" überhaupt verständlich war.

Heute leben wir in einer anderen Welt. Die Digitalisierung veränderte unseren Alltag und tut es noch immer. In unseren Hosentaschen tragen wir Computer, die man noch vor wenigen Dekaden für unmöglich realisierbar gehalten hätte. Die rasante technische Entwicklung machte auch vor den Hobbykellern nicht halt. So entstand in vielen Bastlerköpfen der Wunsch, ein klein wenig die digitale Welt selbst mitzugestalten, etwas zu kreieren. Der Trend des *Physical Computing* war geboren, allerdings zunächst noch sehr beschwerlich – mit Handbüchern in Telefonbuchstärke und kaum verfügbaren Hardware-Komponenten.

Ein paar findigen Italienern gelang es, diese Misere zu beenden. Sie schufen mit Arduino eine *Physical Computing Plattform* für Jedermann. Wegen der dadurch entstandenen schier unzählbaren Möglichkeiten machten die kleinen Experimentierplatinen rasch Karriere und wuchsen über den Hobbykeller hinaus: Künstler bereichern dadurch heute ihre Installationen mit interaktiven Elementen, Forscher bauen Prototypen für Feldversuche, Schüler lernen an Arduinos das Programmieren.

Dieses Buch wird nun auch Ihnen einen Einstieg in diese faszinierende Welt vermitteln, Impulse geben und Möglichkeiten aufzeigen. Nach der Lektüre werden Sie in der Lage sein, eigene Projekte selbst zu planen und umzusetzen, Programmcode zu entwickeln und Fehler zu beseiti-

gen. Dabei wird stets Wert auf eine ausgewogene Mischung zwischen Theorie und Praxis gelegt. Sie werden notwendige technische Grundlagen kennenlernen ohne ermüdend tief in Formeln oder Syntaxregeln zu versinken. Zahlreiche Beispiele machen das vermittelte Wissen anschaulich und nachvollziehbar. Die Reihenfolge der Kapitel wurde so gewählt, dass Sie die besprochenen Sachverhalte Schritt für Schritt an Ihrem eigenen Arduino nachvollziehen können.

Dieses Buch erhebt somit auch keinerlei Anspruch auf Vollständigkeit. Stattdessen wurde der Fokus darauf gelegt, dass Sie die wichtigsten Zusammenhänge auch ohne technische Vorkenntnisse verstehen können. Sie werden einen breiten Überblick über Möglichkeiten und Vorgehensweisen in verschiedenen Gebieten erhalten – sei es Elektronik, Programmierung, Peripherie oder Vernetzung. Möchten Sie anschließend Ihr Wissen in einem bestimmten Teilbereich vertiefen, können Sie dann hierdurch bereits auf eine gute Grundlage zurückgreifen.

Kapitel 2
Geschichte

2.1 Die Geschichte von Mikrocontrollern

Die Geschichte von Mikrocontrollern beginnt mit der Geschichte der Mathematik. Mit dem Bestreben der Gelehrten, die Welt und ihre Mechanismen in Formeln und Verhältnissen zu beschreiben, entstand rasch auch der Wunsch, diese effizient berechnen zu können. Bereits in der Antike begannen die Menschen mit dem Gebrauch der Rechenmethoden und Zahlensysteme, welche die Grundlagen für unsere modernen digitalen Geräte bilden. Dabei entwickelte man schon Handlungsvorschriften zur schrittweisen Lösung eines Problems, wie zum Beispiel den *euklidischen Algorithmus* zur Ermittlung des größten gemeinsamen Teilers zweier Zahlen. Ein Algorithmus ist also quasi das Kochrezept, das Schritt für Schritt den Lösungsweg für ein bestimmtes mathematisches Problem beschreibt – egal, ob nun zur Berechnung der nächsten Sonnenfinsternis, einer Brückentragfähigkeit oder Ihres Nettolohns. Über Jahrhunderte führte man solche Rechenvorschriften von Hand bei aus.

Verständlicherweise entstand das Bestreben, derartige Problemlösungen zu automatisieren. Bereits ab dem 17. Jahrhundert kamen erste mechanische Rechenwerke auf; doch es sollte bis 1941 dauern, als mit der *Zuse Z3* der erste programmierbare Universalrechner, der beliebige Algorithmen ausführen konnte, in Betrieb genommen wurde. Die daraus entwickelten Großrechner in vielen Universitäten und Unternehmen füllten ganze Säle. Nun berechnete man Staudämme, Hängebrücken und Bilanzen mit der neuen Technik.

Diese Anfänge der elektronischen Rechenwerke nutzte man auch für die Einsatzerprobung verschiedener Zahlensysteme. Während die mechanischen Vorgängermaschinen fast immer das uns Anwen-

dern bestens vertraute Dezimalsystem verwendeten, erwies sich dies bereits im Einsatz der *Z3* als eher unvorteilhaft und umständlich. Neben einigen mutigen Versuchen mit exotischen Zahlensystemen (so zum Beispiel der auf dem Ternärsystem[1] basierende *Setun*-Rechner, welcher 1958 in der Sowjetunion gebaut wurde), kristallisierte sich bald das Binärsystem als Favorit der Hersteller heraus. Seine Reduktion auf nur zwei komplementäre Zustände (*0* oder *1*; *ein* oder *aus*) lässt sich in der Elektronik besonders einfach und robust realisieren. Wir werden im Kapitel 4 noch einen genaueren Blick auf das Binärsystem werfen.

Die aufkommende Halbleitertechnik verkleinerte elektronische Komponenten um mehrere Größenordnungen. Die daraus resultierende Entwicklung des Mikrochips Mitte der 1960-er Jahre ermöglichte fünf Jahre später schließlich auch den Bau des ersten Mikroprozessors. Nun konnten alle elektronischen Komponenten eines Rechners, der in der Relais-Bauweise der *Z3* noch ganze Sporthallen gefüllt hätte, auf einem daumennagelgroßen Silizium-Chip untergebracht werden. Die Bahn war frei für die Entwicklung unserer heutigen Computer.

Die fortschreitende Miniaturisierung ermöglichte es nun, neben dem eigentlichen Prozessor, also dem Gehirn des Rechners, auch weitere Komponenten mit auf dem Chip unterzubringen. Durch das Anfügen von Arbeits- und Programmspeicher sowie Schnittstellen zur Ein- und Ausgabe von Daten sowie optionalen weiteren Funktionsblöcken entsteht ein kompletter Computer auf nur einem Chip – der Mikrocontroller. Ohne diesen wäre heute Ihr Smartphone nicht smart und Ihre Kreditkarte nicht fälschungssicher.

2.2 Entstehung der Arduino-Plattform

Die stetig fallenden Preise für Mikrochips sorgten dafür, dass Mikrocontroller nach der Jahrtausendwende auch mehr und mehr in

1 Ein Ternärsystem, auch Dreiersystem, arbeitet nur mit drei möglichen Ziffern; je nach Definition 0, 1 und 2 oder -1, 0 und 1.

den heimischen Bastelkellern ankamen. Intelligente Roboter, fernsteuerbare Autos, automatische Murmelsortiermaschinen – die Miniaturcomputer machten vieles möglich, was mit herkömmlicher Elektronik schlicht zu aufwändig gewesen wäre. *Physical Computing* war der neue Trend: Man verband also das Basteln an der Hardware mit dem Programmieren an der Software, um kleine „intelligente" Selbstbau-Projekte zu realisieren. Diese können auf Ereignisse in unserer realen, analogen Welt reagieren, sie verarbeiten und sogar auf sie einwirken. Manch Bastler baut sich einen Blumengieß-Automat, ein anderer eine Lichtsteuerung oder gar einen selbstkreierten Staubsaug-Roboter.

Anfangs schien all dies jedoch noch als eine ingenieurtechnische Herausforderung: Zwar gab es preiswerte Mikrocontroller, diese waren jedoch auf den industriellen Markt ausgelegt. Um ihnen Herr zu werden, galt es umfangreiche Dokumentationen zu studieren, teils teure Software anzuschaffen und passende Platinen zu bauen – für den Heimgebrauch undenkbar.

Das wollten die Italiener Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino und David Mellis ändern. Sie trafen sich regelmäßig in einer Bar in Ivrea, die ihren Namen „Arduino" von einem einstigen italienischen König übernommen hatte. 2005 wählten sie diesen Namen für ihre selbstentwickelte Musterplatine, kurz darauf stellten sie auch die erste Version einer Computersoftware zur Programmierung des Controllers fertig. Erstmals gab es nun eine einfach zu nutzende Platine, die zahlreiche Anschlüsse eines Mikrocontrollers für verschiedenste Anwendungen zur Verfügung stellt, ohne dabei teures Zubehör zu erfordern – denn die Entwickler veröffentlichten das Projekt unter einer Creative-Commons-Lizenz und ermöglichten somit die einfache Nutzung für Jedermann.

Die fünf Studenten gründeten die *Arduino LCC* als gemeinsame Firma, um das Projekt weiter voranzutreiben. Die eigentliche Herstellung der Hardware übernahmen dabei externe Dienstleister. Schnell stellte sich wirtschaftlicher Erfolg ein; man schätzt, dass allein bis 2013 rund 700.000 Arduinos verkauft wurden. Jedoch entbrann-

te ein Markenrechtsstreit zwischen den Gründern, der 2015 sogar dazu führte, dass unter dem Namen *Genuino* eine neue Marke eingetragen wurde. Auch wenn der Streit 2016 offiziell beigelegt wurde, herrscht in manchen Bereichen noch Unsicherheit über die rechtliche Situation. Wundern Sie sich also nicht, wenn Sie nach „Arduinos" suchen, aber „Genuinos" angeboten bekommen – der Inhalt ist identisch. Im Sprachgebrauch blieb der ursprüngliche Name aber stets standhaft.

Es sei erwähnt, dass etwa zeitgleich zur Entwicklung des Arduino auch der Raspberry Pi entstand. Falls Sie sich nun nach dem Unterschied fragen: Es handelt sich dabei um einen Einplatinencomputer, der ebenfalls bei Selbstbau-Projekten äußerst beliebt ist. Seine Fähigkeiten übersteigen die eines Arduino aber um mehrere Größenordnungen. Es handelt sich um einen vollwertigen Computer mit Betriebssystem, Dateisystem, Monitoranschlüssen, USB-Ports und vielem mehr. Dadurch steigt allerdings auch das Level der Abstraktion, das heißt, man kann nicht mehr so „nah an der Hardware" agieren. Die Programmierung erfolgt wie bei einem normalen PC; auf dem Chip läuft umfangreiche Software, die von einer einzelnen Person kaum zu überblicken ist. Nicht zuletzt ist ein Raspberry Pi auch den gleichen Gefahren ausgesetzt wie ein normaler PC: Programmfehler sind mitunter schwer zu lokalisieren, er kann – je nach Betriebssystem – anfällig sein für Viren und unbedachte Updates können die Funktionalität unter Umständen komplett lahmlegen.

Im Gegensatz dazu benötigt Arduino weder ein herkömmliches Betriebssystem noch ein Dateisystem auf dem Chip. Er erlaubt den unmittelbaren Zugriff auf die Hardware (Ein- und Ausgangspins) und ist sehr zuverlässig, wenn Sie auf die Echtzeitfähigkeit, also die garantierte Reaktion in einem gewissen Zeitfernster, Wert legen. Der Befall mit Viren oder Malware ist nahezu ausgeschlossen. Diese minimalisierten Fähigkeiten schlagen sich auch im Preis nieder: Ein Arduino ist deutlich günstiger zu haben als ein Raspberry Pi. Kaufen Sie nur den zu Grunde liegenden Mikrocontroller-Chip, ist die Ersparnis noch größer.

Beide Plattformen haben also ihre Vor- und Nachteile. Als Faustregel gilt: Ist Ihr Projekt eher hardwareorientiert, also wollen Sie Lämpchen, Motoren und Sensoren steuern, ist Arduino sehr wahrscheinlich die beste Lösung für Sie. Erfordert Ihr Projekt jedoch umfangreiche Rechenleistung, zum Beispiel weil Sie ein Kamerabild auswerten möchten oder einen Computermonitor als grafische Anzeige benutzen, sollten Sie zu einem Raspberry Pi greifen. Natürlich gibt es Überschneidungen. So könnten Sie eine Wanduhr beispielsweise sehr elegant per Arduino mit angeschlossenen Schrittmotoren oder LED-Anzeige realisieren; aber Sie könnten auch einen Raspberry Pi nehmen und per Videobeamer eine animierte Uhr projizieren. Allein Ihre Fantasie entscheidet. Es gibt auch zahlreiche Projekte, die beide Plattformen zusammen einsetzen: So zum Beispiel ein Roboter, dessen „Gehirn" ein Raspberry Pi ist, welcher seine Sensordaten jedoch von mehreren verteilten Arduinos bezieht, die diese Daten zunächst aufbereiten. Beide Plattformen stehen also nicht in Konkurrenz zueinander, sie ergänzen sich vielmehr. Im Folgenden fokussieren wir uns auf Arduino-Plattform, zu Raspberry Pi ist aber ebenfalls entsprechende Literatur erhältlich.

2.3 Überblick über verfügbare Hardware

Was die Hardware betrifft, hat sich in den letzten Jahren der Arduino UNO als Standardmodell durchgesetzt. Alle wichtigen Anschlussmöglichkeiten sind direkt über kleine Buchsenleisten an der Platine abgreifbar, dadurch muss man beim Experimentieren nichts löten. Die Platine kann sowohl über USB als auch über ein Netzteil mit Niedervoltstecker mit Spannung versorgt werden. Die aktuelle überarbeitete Version „UNO R3" (Revision 3) unterscheidet sich nur in nebensächlichen Details von den Vorgängerversionen. Als Mikrocontroller kommt nach wie vor ein „ATmega328P" des Herstellers Microchip AVR (vormals Atmel) zum Einsatz. Für viele Anwendungen sind sogenannte *Shields* verfügbar. Das sind Erweiterungsplatinen, die dank passender Dimensionierung direkt auf den Arduino UNO aufgesteckt werden können und dadurch auch sofort mit Betriebsspannung und allen benötigten Signalen versorgt werden.

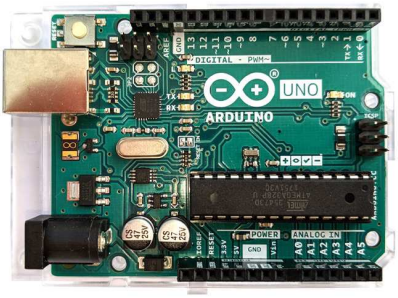


Abb. 2.1 Arduino UNO R3

Neben dem Arduino UNO wurden noch viele weitere Bauformen entwickelt und getestet. Nachfolgend seien nur die genannt, die es zu einer gewissen Relevanz geschafft haben.

Arduino Nano ist die Miniaturisierung des Arduino UNO. Er besitzt weiterhin alle Anschlusspins und den gleichen Mikrocontroller, benötigt jedoch nur noch ein Viertel der Grundfläche seines großen Bruders. Möglich wurde dies, indem die Bauform der Mikrochips und des USB-Anschlusses verkleinert wurde, außerdem entfällt der separate Spannungseingang. Man kann ihn zum Experimentieren direkt auf ein Breadboard stecken. Er kann für alle Projekte genutzt werden, die auch mit Arduino UNO möglich sind, außer wenn Shields zum Einsatz kommen – durch die abweichende Bauform können sie nicht mehr direkt gesteckt werden.

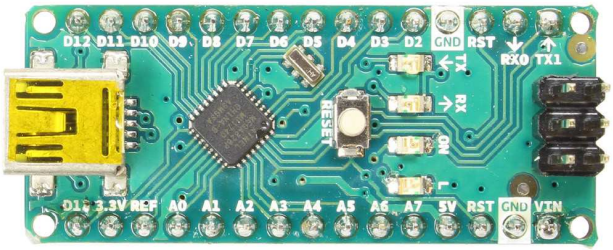


Abb. 2.2 Arduino Nano

Der *Arduino Mini* wurde durch den Wegfall des USB-Ports nochmals verkleinert. Zum Programmieren ist deshalb ein spezieller Adapter nötig. Ansonsten entspricht er dem Arduino Nano.

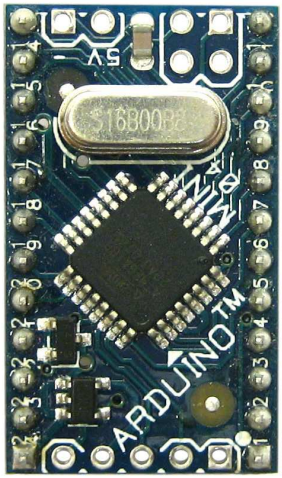


Abb. 2.3 Arduino Mini

Arduino Micro ist ein etwas leistungsstärkerer Bruder des Arduino Nano. Trotz der etwas längeren Abmessungen passt er noch auf ein Breadboard. Er bietet mehr Ein- und Ausgabepins sowie geringfügig mehr Arbeitsspeicher, da nun ein ATmega32U4-Mikrocontroller zum Einsatz kommt.

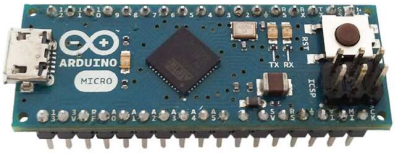


Abb. 2.4 Arduino Micro

Der *Arduino Mega2560* bietet den 8-fachen Speicher des Arduino UNO, seine Anschlusszahl wurde mit 54 digitalen und 16 Analogen Pins mehr als verdreifacht. Entsprechend ist auch die Bauform der Platine etwas größer. Die meisten Shields sind dennoch kompatibel.

Abb. 2.5 Arduino Mega2560

LillyPad Arduino ist der Name einer Reihe von Arduino-Boards, die durch ihre runde Bauform auffallen. Sie wurden dafür konzipiert in Textilien eingearbeitet zu werden. Ihr Durchmesser beträgt meist 5 cm, in den Anschlüssen orientieren sie sich weitgehend an den rechteckigen Boards, jedoch mit gewissen Einschränkungen.

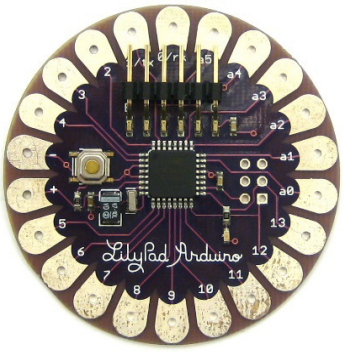


Abb. 2.6 LillyPad Arduino

Die kleineste Miniaturisierung stellt der *Arduino Gemma* dar. Auf einer kreisrunden Scheibe von nur 2,8 cm Durchmesser bietet er allerdings auch nur 3 Datenpins.

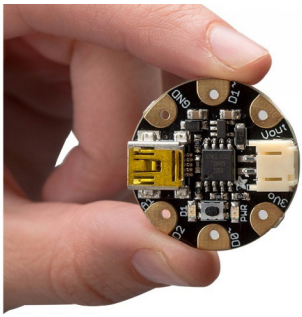


Abb. 2.7 Arduino Gemma

Es existieren zahlreiche weitere Sonderbauformen, beispielhaft genannt sei *Arduino Robot*, dessen Form und Anschlüsse bereits an seinen angestrebten Einsatz als Roboter angepasst wurden oder Platinen, die bereits als Grundplatte für ein kleines Auto genutzt werden können.

Die Tatsache, dass das Arduino-Projekt im Wesentlichen unter freien Lizenzen veröffentlicht wurde, sorgt dafür, dass es auch zahlreiche Nachbauten und alternative Boards gibt. Die meisten von ihnen sind nahezu vollständig kompatibel zum Original, allerdings unterscheiden sie sich in einigen kleinen Details. So verfügen preiswertere Boards oft über den identischen Mikrocontroller und exakt die gleichen Anschlüsse, allerdings wird für die USB-Verbindung zum PC ein anderer USB-zu-Seriell-Konverter verwendet. Das führt dazu, dass Sie unter Umständen zunächst einen Treiber für diesen Konverter-Chip mit der Bezeichnung CH340 aus dem Internet laden müssen, bevor Ihr PC das Board erkennt. Näheres dazu erfahren Sie in Kapitel 18.1.

2.4 Shields

Die zahlreichen Anschlusspins der Arduino-Platinen ermöglichen die Verbindung mit einer Vielzahl von weiteren Komponenten, in späteren

Kapiteln werden wir noch viele Beispiele kennenlernen. Eine Besonderheit der Arduino-Plattform sind dabei die Shields. Dabei handelt es sich um Erweiterungs-Platinen, welche in ihren Abmessungen so gestaltet wurden, dass sie direkt auf den Arduino UNO aufgesteckt werden können. Die Anschlusspins dienen dabei zugleich als elektrische und mechanische Verbindung.

Je nach Shield-Typ liegt die Funktion beispielsweise in der Steuerung von Motoren, der Herstellung einer Netzwerkverbindung oder der Verbindung mit einer SD-Karte. Die dafür benötigten Pins werden direkt abgegriffen, alle weiteren Anschlüsse werden durch das Shield einfach weitergeleitet und sind genau wie auf der Hauptplatine abgreifbar. Gibt es keine Dopplungen unter den benötigten Signalpins, können Shields sogar kaskadiert werden.

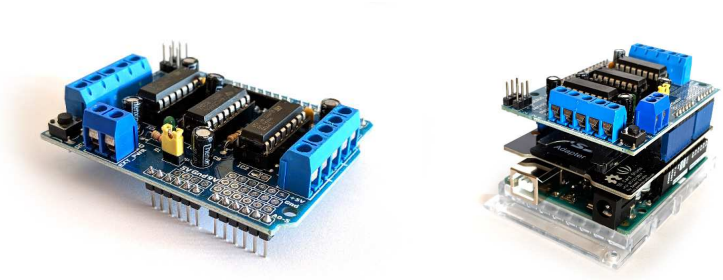


Abb. 2.8 links ein Shield zur Steuerung von bis zu 6 Motoren, rechts eine Kaskade aus einem Arduino UNO mit aufgestecktem SD-Karten-Shield und zusätzlichem Motor-Shield

Der Vorteil von Shields liegt darin, dass die Anzahl der manuell zu steckenden Verbindungen sinkt. Damit verringert sich auch die Gefahr von Verdrahtungsfehlern und der Versuchsaufbau wirkt aufgeräumter.

2.5 Software-Überblick

Die heute meistgenutzte Software zur Arduino-Programmierung ist die offizielle Arduino IDE. Das Kürzel steht für „Integrated Develop-

ment Environment“ (Integrierte Entwicklungsumgebung) – dieser etwas sperrige Begriff weist darauf hin, dass mit dieser Anwendung alle Schritte der Softwareentwicklung „an einem Ort“ möglich sind.

Ursprünglich entstand die Arduino IDE aus dem Projekt *Wiring*. Hernando Barragán entwickelte es ab 2003 im Rahmen seiner Masterarbeit mit dem Ziel, Designern und Künstlern die Arbeit mit Elektronik zu erleichtern. Diese schreckten häufig vor den relativ komplizierten Details der Schaltungen und Prozessoren zurück – Barragán wollte diese „quälenden Details“ für die Anwender ausblenden, so dass ein Fokus auf das eigentliche Kunstprojekt möglich wird. Es entstand eine javabasierte Software-Plattform, aus der schließlich auch die heutige Arduino IDE abgeleitet wurde. Als Anwender profitieren Sie dabei heute sehr von Barragáns Bestrebungen zur Vereinfachung, deshalb werden wir uns in den folgenden Kapiteln stets auf die Arduino IDE beziehen.

Für den Überblick sei jedoch erwähnt, dass die Programmierung der Arduino-Boards auch mit zahlreichen anderen, teils umfangreicheren und komplexeren Anwendungen möglich ist. Die Plattform ist quelloffen und verwendet einen Standard-Mikrocontroller, somit gibt es kaum Schranken. Die folgenden Alternativen richten sich an erfahrenere Anwender mit komplexeren Projekten:

- *Atmel Studio* stammt direkt vom Hersteller des auf dem Board verwendeten Mikrocontrollers. Es läuft nur unter Windows und ermöglicht eine sehr hardwarenahe Entwicklung und kann dem Chip beispielsweise im Hinblick auf Rechengeschwindigkeit und Speicherplatzoptimierung noch Reserven entlocken.[2]
- *Microsoft Visual Studio* benötigt die *VisualMicro*-Erweiterung, um mit dem Arduino kompatibel zu sein. Dies empfiehlt sich, wenn man ohnehin schon andere Projekte in Visual Studio bearbeitet – so muss man nicht die Plattform wechseln.[3]

2 https://www.microchip.com/mplab/avr-support/atmel-studio-7

3 https://www.visualmicro.com/page/Arduino-Visual-Studio-Downloads.aspx

- *MariaMole* ähnelt in der Handhabung sehr der Arduino IDE, lässt sich jedoch in einigen Punkten besser an persönliche Vorlieben anpassen.[4]
- *embedXcode* erweitert die in der Macintosh-Welt verbreitete Plattform *Xcode* um die Arduino-Unterstützung[5]

Außerdem besteht die Möglichkeit, den Web-Editor auf der offiziellen Arduino Website[6] zu verwenden. Da dieser jedoch, wie oft bei Web-Anwendungen, weniger stabil läuft als vergleichbare Desktop-Software, werden wir uns hier im Buch stets auf die Desktop-Version beziehen.

2.6 Installation der Arduino-IDE

Die Arduino IDE ist für Windows, Mac OS X und Linux verfügbar. Im Folgenden werden wir uns auf die Installation unter Windows fokussieren, da dies sicherlich die meisten Leser betrifft. Die Installation in den anderen Betriebssystemumgebungen stellt jedoch ebenso keine größere Herausforderung dar, wenn Sie den Umgang mit Ihrem Betriebssystem gewohnt sind.

Zunächst laden wir die Installationsdatei von der offiziellen Arduino-Website:

https://www.arduino.cc/en/Main/Software

Die Software wird dort sowohl als ausführbare Exe-Datei als auch als Zip-Archiv angeboten. Die erstgenannte ist die einfachere Variante für Einsteiger, da somit zeitgleich auch alle nötigen Treiber und Komponenten installiert werden. Allerdings sind dafür Administrator-Rechte unter Windows notwendig. Das Zip-Archiv ist etwas kleinteiliger in der Installation, benötigt jedoch keine Admin-Rechte. Als weitere Alternative besteht seit kurzem die Möglichkeit, die Software als App aus dem

4 http://dalpix.com/mariamole
5 http://playground.arduino.cc/Main/EmbedXcode
6 https://create.arduino.cc/editor

Microsoft Store zu laden, ein entsprechender Link findet sich ebenfalls auf der Website. Unabhängig vom gewählten Weg wird ein Spendenhinweis angezeigt. Da die Software kostenlos ist, finanziert sich die Entwicklung zu großen Teilen aus Spenden. Ob Sie dazu beitragen möchten, bleibt selbstverständlich Ihnen überlassen. Auf den Download hat dies keinen Einfluss.

Wir beziehen uns im Folgenden auf die Installation per Windows-Installer (Exe-Datei). Laden Sie die Datei auf Ihren Computer und öffnen Sie diese.

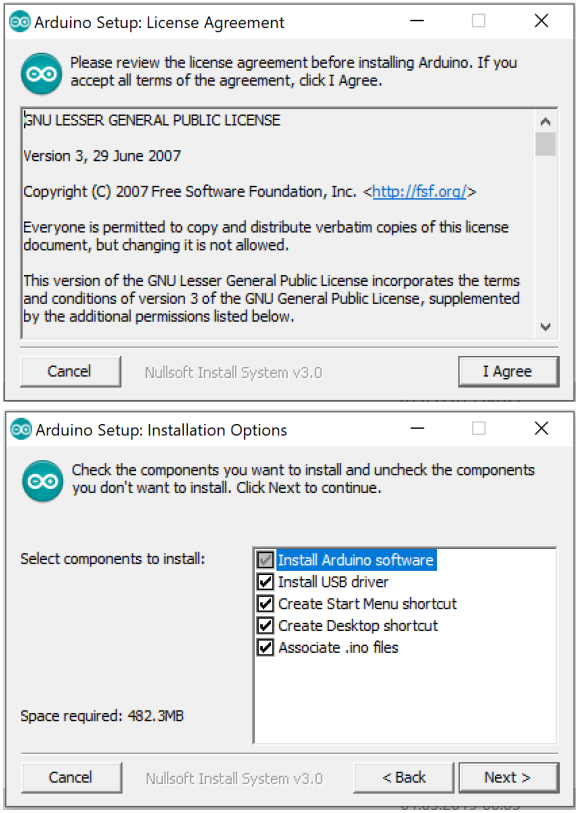


Abb. 2.9 Installation der Arduino IDE

Zunächst werden Sie gebeten, den Lizenzvertrag zu lesen. Anschließend können Sie wählen, welche Komponenten Sie installieren möchten. Die Arduino-Software und der USB-Treiber sollten auf jeden Fall angewählt werden. Ob Sie Desktop-, Startmenü- und Dateiverknüpfungen nutzen möchten, ist Ihren Vorlieben überlassen.

Anschließend startet die Installation, bei einem modernen Computer sollte sie nicht länger als 5 Minuten dauern.

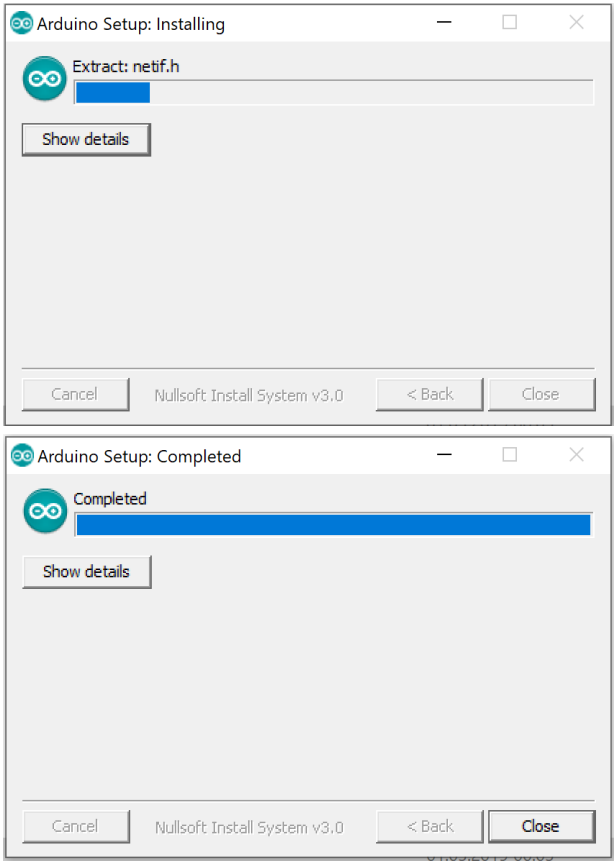


Abb. 2.10 Nach wenigen Minuten ist die Installation durchlaufen.

Ist die Installation vollständig abgeschlossen, lässt sich dieses Fenster schließen und Sie können die Arduino IDE nun über den Desktop oder das Startmenü öffnen.

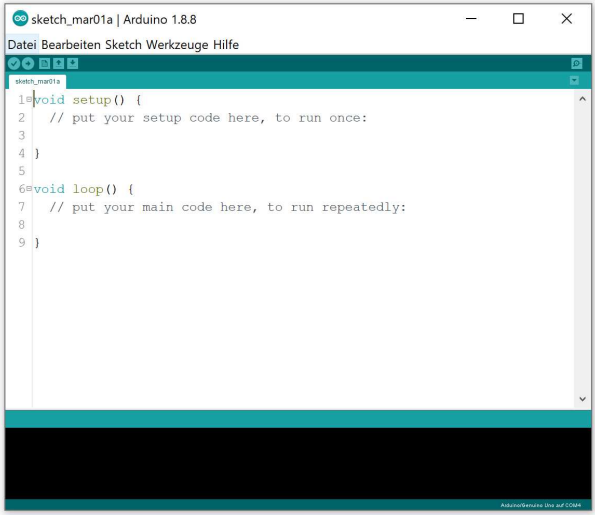


Abb. 2.11 Die Arduino IDE – der mittlere Bereich mit weißem Hintergrund ist für den Programmcode vorgesehen; im schwarzen Balken am unteren Rand erscheinen etwaige Fehlermeldungen oder Hinweise.

Zunächst wird Ihnen eine nahezu leere Seite angezeigt. In diesem Feld erstellen Sie später den Programmcode, bei Arduino auch *Sketch* genannt. Im Kapitel 4 werden wir uns ausführlich damit beschäftigen. Im unteren schwarzen Feld werden Sie auf Probleme aufmerksam gemacht, falls der Sketch aus irgendeinem Grund nicht auf den Mikrocontroller übertragen werden kann.

Bevor Sie mit dem Programmieren loslegen, sollten Sie noch kurz zwei wichtige Einstellungen prüfen. Stecken Sie dazu Ihre Arduino-Platine per USB an den Computer an und schauen Sie nach, ob unter *Werkzeuge -> Board* das korrekte Board ausgewählt ist, welches Sie benutzen

möchten. Im Regelfall sollte das „Arduino/Genuino Uno“ sein. Im Menüpunkt darunter muss ein COM-Port mit einem Häkchen versehen sein, die Zahl kann jedoch von diesem Beispiel abweichen.

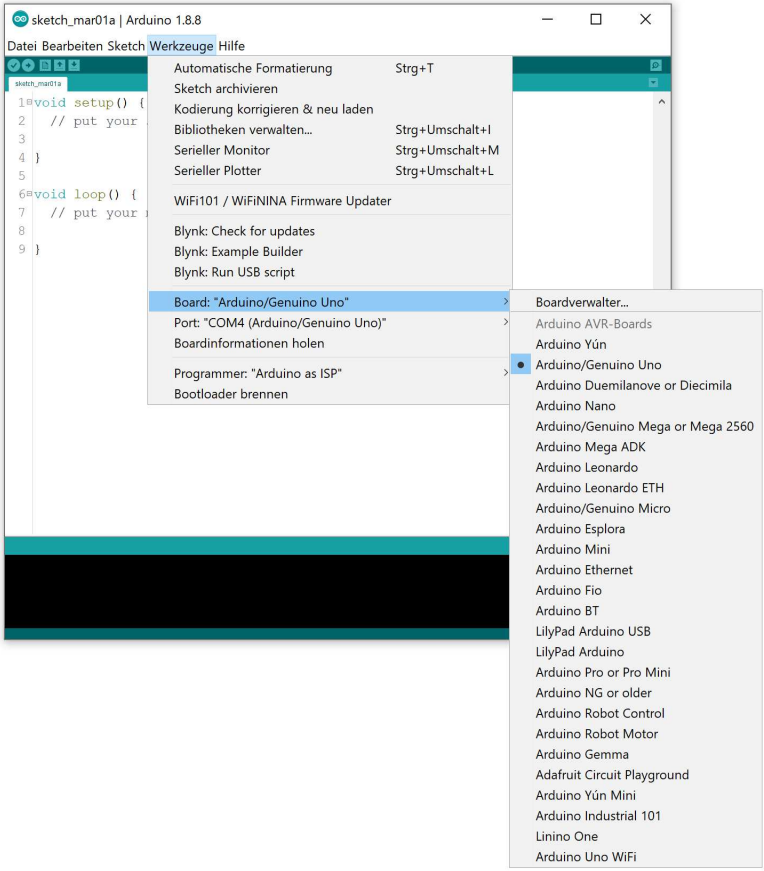


Abb. 2.12 Die korrekte Auswahl des verwendeten Arduino-Boards ist wichtig

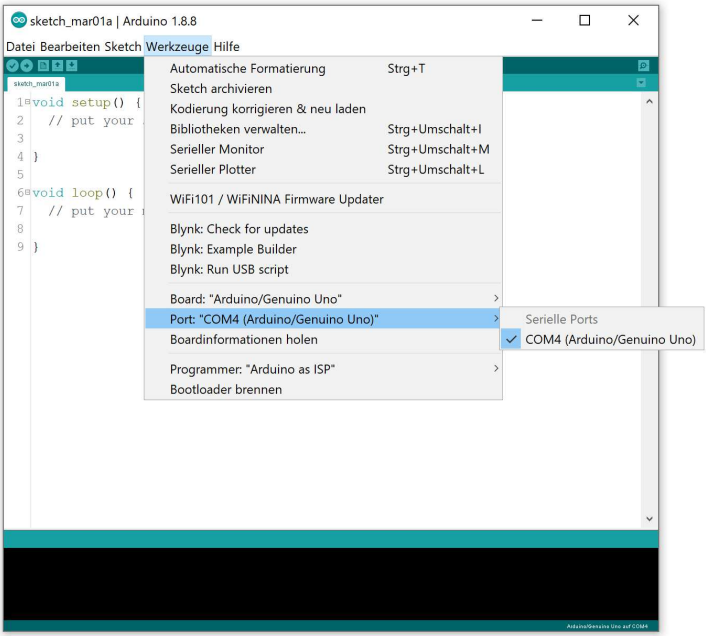


Abb. 2.13 Trotz der USB-Verbindung muss als Anschluss ein serieller Port ausgewählt werden, da die Verbindung zum eigentlichen Arduino-Mikrocontroller letztlich seriell (über einen Wandler auf dem Board) erfolgt

2.7 Bibliotheken

Die Arduino-Gemeinde lebt vom gegenseitigen Austausch. Daher ist es nicht selten, dass Nutzer ihren selbstgeschriebenen Programmcode für andere zur Verfügung stellen. Das geschieht entweder durch die Veröffentlichung des Sketches selbst oder einer Library-Datei, einer sogenannten Bibliothek. Diese können Sie sich wie ein Plug-In für Ihren Webbrowser vorstellen. Dementsprechend vielfältig sind die Einsatzzwecke: Einige dienen der Kommunikation mit angeschlossenen Sensoren oder Displays, andere wiederum erleichtern die Auswertung bestimmter Messdaten.

Über den Menüpunkt *Sketch -> Bibliothek einbinden -> Bibliotheken verwalten* kann in der Arduino IDE eine Übersicht der verfügbaren Bibliotheken angezeigt werden. Eine Suchfunktion erleichtert das Finden nach Stichworten. Fährt man mit der Maus über einen Eintrag, kann die entsprechende Bibliothek installiert oder aktualisiert werden.

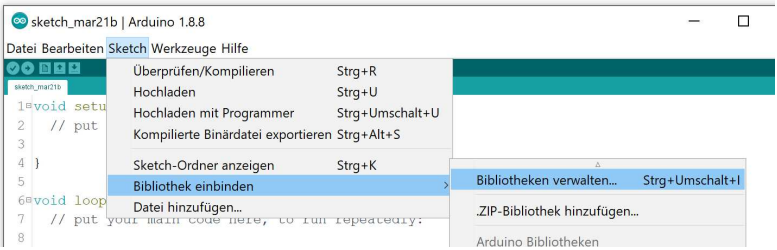


Abb. 2.14 Der Bibliotheksverwalter kann einfach über das Menü geöffnet werden

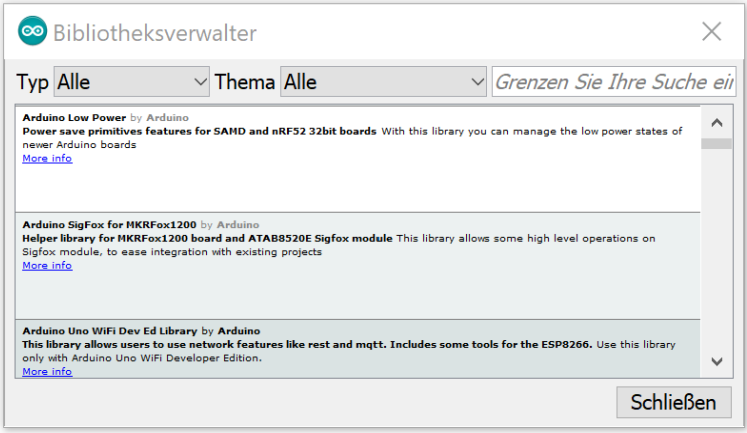


Abb. 2.15 Der Bibliotheksverwalter ähnelt einer Suchmaske, welche den Server der Arduino-Website durchsucht

Der Bibliotheksmanager durchsucht jedoch nur den Server des Arduino-Projektes. Es sind auf anderen Websites noch weitere Bibliotheken verfügbar, welche dann separat heruntergeladen und über den benach-

barten Menübefehl *.ZIP-Bibliothek hinzufügen* installiert werden können.

Grundsätzlich verhalten sich Bibliotheken passiv: Sie treten erst in Aktion, wenn sie explizit ins Programm eingebunden werden. Den dafür vorgesehenen Befehl `#include` lernen Sie in Kapitel 5.5.2 kennen, wenn er das erste Mal benötigt wird. Zusätzlich zum eigentlichen Programmcode sind oft noch Beispielsketche enthalten, welche automatisch dem Menü *Datei -> Beispiele* angefügt werden.

Die meisten Bibliotheken verfügen zudem über Online-Dokumentationen, welche über die Handhabung informieren. Der Link dazu findet sich entweder in den Beispielsketchen oder kann über eine Internet-Suchmaschine gefunden werden. In selteneren Fällen enthalten die Bibliotheksdateien selbst eine Dokumentation, welche dann im Unterordner *\libraries* des Arduino-Verzeichnisses gefunden werden kann. Wir werden in späteren Kapiteln noch detaillierter auf bestimmte Bibliotheken und deren Funktionen eingehen. Für den Anfang genügt es, wenn Sie den Begriff deuten können.

Nun sind Sie in Bezug auf die Software schon startklar. Das nächste Kapitel wird Ihnen nun noch ein paar elektronische Grundlagen vermitteln, bevor wir den ersten Sketch auf den Arduino laden.

Kapitel 3
Hardware-Einführung

3.1 Grundlagen

Die Arduino-Plattform bietet zahlreiche Möglichkeiten, mit anderen elektronischen Komponenten zu interagieren. Sie können beispielsweise Sensorwerte abfragen, Motoren regeln oder Displays ansteuern. Ab Kapitel 5 werden wir uns ausführlich damit beschäftigen. Um diese zu verstehen, ist jedoch ein grundlegendes Verständnis von Elektronik erforderlich. Deshalb klären wir im Folgenden einige grundlegende Begriffe und Zusammenhänge.

3.1.1 Strom und Spannung

Fundamental für unsere heutige Elektronik ist die Elektrizität, deshalb werfen wir zunächst einen Blick auf deren grundlegende Größen. Jeder hat im Alltag schon einmal von Strom und Spannung gesprochen, jedoch werden die Begriffe mitunter falsch verwendet. So zählt Ihr „Stromzähler“ zum Beispiel eigentlich gar nicht den elektrischen Strom, sondern die elektrische Energie, und wenn Sie Ihren Staubsauger „an den Strom anschließen“ meinen Sie eigentlich, dass sie ihn mit einer Spannungsquelle verbinden.

Als elektrischen Strom bezeichnet man den Fluss von elektrisch geladenen Teilchen. In einem Metall sind das die Elektronen. Stellen Sie sich einen Wasserschlauch vor. Die Menge des hindurchfließenden Wassers könnte man in Liter pro Minute messen. Messen wir die Menge der durch einen elektrischen Leiter fließenden Elektronen während einer bestimmten Zeit, erhalten wir den elektrischen Strom, manchmal auch als Stromstärke bezeichnet. Seine Einheit ist das Ampere[1].

1 benannt nach dem französischen Physiker André-Marie Ampère

Nun bedecken Sie die Öffnung des Wasserschlauches mit Ihrem Finger. Der Wasserfluss wird behindert, es baut sich ein gewisser Druck auf. Diesen können Sie regulieren, indem sie Ihren Finger etwas loslassen oder fester zudrücken. Analog zum Druck gibt es die elektrische Spannung. Sie ist die Kraft, welche die elektrischen Teilchen antreibt.

Wir erweitern unser Gedankenexperiment: Nun haben wir zwei wassergefüllte Gefäße auf gleicher Höhe mit einem Schlauch verbunden. Aufgrund der ausgeglichenen Höhe steht das Wasser im Schlauch still.

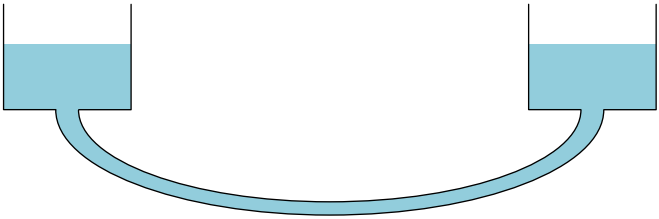


Abb. 3.1 Bei ausgeglichenem Potential fließt kein Strom

Heben wir nun eines der Gefäße an, wird Wasser über den Schlauch in das tiefere Gefäß fließen. Die elektrische Entsprechung zur Höhe des Wasserspiegels ist das elektrische Potential. Durch Potentialunterschiede entsteht eine Spannung, die einen Strom antreiben kann – so wie das Wasser im Schlauch zwischen den Gefäßen. Die Einheit für die Spannung ist das Volt[2].

Eine Schwäche hat unsere Analogie: Zerschneidet man den Schlauch, fließt das Wasser einfach heraus – die Elektronen bleiben jedoch im Metall, wenn man eine Leitung auftrennt. Das liegt an einer Eigenschaft unserer Umgebungsluft: Als Isolator leitet sie keinen elektrischen Strom, die Elektronen können sich in ihr nicht frei bewegen. Eine weitere Besonderheit der Elektrizität ist das Erfordernis eines geschlossenen Weges – der Ihnen sicher bereits bekannte Stromkreis. Im Wesentlichen ist seine Notwendigkeit darin begründet, dass eine Energiequelle (beispielsweise eine Batterie) die Elektronen im Leiter zwar antreibt, sie

2 benannt nach dem italienischen Physiker Alessandro Volta

aber nicht erzeugt. Deshalb müssen ihr auf der einen Seite die Elektronen wieder zugeführt werden, welche sie auf der anderen Seite abgibt. Ist der Stromkreis offen, fließen keine Elektronen, also kein Strom – und damit auch keine Energie.

Zur anschaulichen Darstellung der Verbindung elektrischer Komponenten nutzt man den Stromlaufplan, Sie kennen ihn umgangssprachlich womöglich als „Schaltplan". Er zeigt die Verdrahtung der Komponenten untereinander.

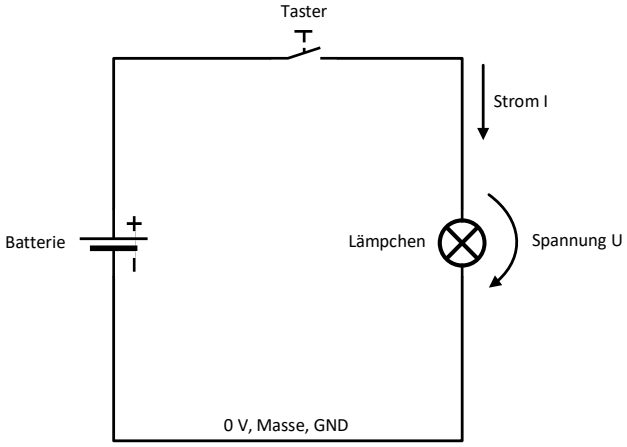


Abb. 3.2 ein einfacher Stromkreis

Die obige Abbildung zeigt einen einfachen Stromkreis bestehend aus einer Energiequelle, einem Taster und einem Glühlämpchen. In den meisten Fällen stammt die Energie aus einer *Spannung*squelle, zum Beispiel einer Batterie oder einem Netzteil. Nur zur Vollständigkeit sei erwähnt, dass es auch *Strom*quellen gibt, diese sind für uns als Einsteiger jedoch nicht von Belang.

Die Batterie stellt eine Gleichspannung bereit. Man hat sich in der Elektronik darauf verständigt, dass man ihren negativen Pol mit „0 V", „Masse" oder „GND" (für Englisch *Ground*) bezeichnet und andere Spannungen, wenn nicht anders angegeben, darauf bezieht. Den positi-

ven Pol bezeichnet man entweder mit seiner Spannung (beispielsweise 5 Volt) oder dem Kürzel Vcc[3].

Wird der Schalter geschlossen, kann die Batterie einen messbaren Strom (Formelzeichen I) antreiben. Ebenfalls lässt sich eine Potentialdifferenz zwischen den beiden Anschlüssen des Lämpchens messen – es steht also unter Spannung (Formelzeichen U). Durch Multiplikation beider Werte kann man nun die elektrische Leistung P bestimmen. Ihre Maßeinheit ist das Watt[4]. Sie ist ein direktes Maß dafür, wieviel Energie das betreffende Teil „verbraucht“[5].

$$P = U \cdot I$$

Aus der Formel wird ersichtlich: Um hohe Leistungen zu übertragen, kann man entweder hohe Ströme oder hohe Spannungen verwenden. Ein europäischer Wasserkocher, den bei 230 V Netzspannung ein Strom von 5 A durchfließt, wird das Wasser in exakt der gleichen Zeit erhitzen wie ein anderer Wasserkocher in Amerika, der bei 115 V Netzspannung 10 A Strom erhält, denn die Multiplikation ergibt immer 1150 Watt.

Multipliziert man die Leistung noch mit der Zeit, erhält man die Energie – also genau das, was Ihr umgangssprachlicher „Stromzähler“ eigentlich summiert. Energie ist das Maß dafür, wieviel Arbeit verrichtet wurde. Leistung ist das Maß dafür, wie schnell eine gewisse Arbeit verrichtet werden kann.

Als Vergleich: Wenn Sie eine Waschmaschine vorsichtig in den ersten Stock tragen wollen, benötigen Sie dafür eine gewisse Energie. Wenn Sie die Waschmaschine innerhalb von nur einer halben Minute in den

3 Englisch *Voltage at the Common Collector*, verweist auf die allgemeine Betriebsspannung einer Schaltung

4 benannt nach dem schottischen Ingenieur James Watt

5 Im physikalischen Sinne wird Energie nicht verbraucht, sondern nur in andere Energieformen umgewandelt – zum Beispiel in Wärme oder Licht.

ersten Stock tragen wollen, brauchen Sie zudem eine hohe Leistung – die benötigte Energie ist aber in beiden Fällen die gleiche.

Nun haben Sie bereits Strom und Spannung kennengelernt. Diese Größen haben übrigens auch weitere Analogien in der Mechanik: So können Sie den Strom zum Beispiel auch als Drehzahl einer rotierenden Welle auffassen und die Spannung als deren Drehmoment. In einem Auto übernimmt dann das Getriebe die Rolle eines „Transformators", um die Drehzahl an die Fahrzeuggeschwindigkeit anzupassen. Die Leistung sind die Pferdestärken Ihres Motors und die Energie schütten Sie als Treibstoff in den Tank.

Interne Betriebsspannung eines Computers	12 V
Betriebsspannung eines Smartphones	4 – 5 V
Maximale Stromaufnahme eines Smartphones	2 A
Durchschnittliche Leistungsaufnahme eines Smartphones	0,5 – 2 W
Spannung in Haushaltssteckdosen	230 V
Stromaufnahme eines Wasserkochers	4 – 8 A
Leistung eines Wasserkochers	1000 – 2000 W
Betriebsspannung der DB-Oberleitung	15.000 V
Maximale Stromaufnahme eines ICE	600 A
Maximale Leistungsaufnahme eines (anfahrenden) ICE	9.000.000 W

Tabelle 3.1 beispielhafte Werte für Spannung, Stromstärke und Leistung

Zur Verdeutlichung der Größenordnungen gibt obige Tabelle einen Überblick üblicher Spannungs- und Stromwerte. Da bereits bei Spannungen ab 50 Volt durch unbeabsichtigte Berührung eine Gesundheitsgefahr entstehen kann, sollten Sie mit höheren Spannungen grundsätzlich nur arbeiten, wenn Sie über entsprechende Fachkenntnis verfügen. Für unsere Anwendungen im Zusammenhang mit dem Arduino wird im Folgenden eine Spannung von 5 V stets ausreichend sein.

Schauen wir uns noch an, wie sich Strom und Spannung in Stromkreisen mit mehreren Elementen verhalten. Für die Spannung gilt: Sie teilt sich auf alle in Reihe (also hintereinander) geschalteten Elemente auf, bei Parallelschaltungen bleibt sie hingegen gleich:

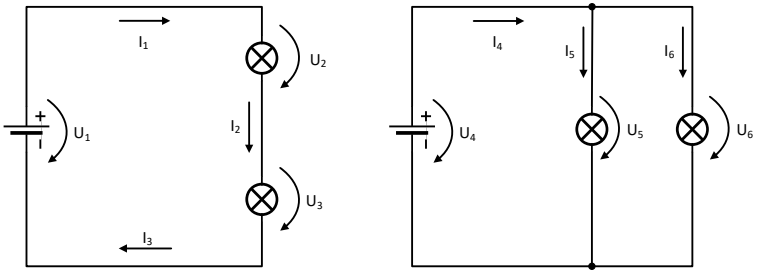


Abb. 3.3 Vergleich zwischen Reihenschaltung (links) und Parallelschaltung (rechts)

$$U_1 = U_2 + U_3 \qquad U_4 = U_5 = U_6$$
$$I_1 = I_2 = I_3 \qquad I_4 = I_5 + I_6$$

Für den Strom gilt es genau umgekehrt: Er ist in allen Elementen einer Reihenschaltung gleich, (denn er muss ja jedes dieser Bauteile durchfließen), teilt sich jedoch bei Parallelschaltungen auf.

3.1.2 Elektronische Komponenten

3.1.2.1 Widerstand

Im vorhergehenden Kapitel haben wir bereits einen kleinen Stromkreis mit einem Glühlämpchen kennengelernt. Die Spannung wurde in diesem Fall von der Spannungsquelle, einer Batterie, bestimmt. Es stellt sich die Frage, woraus sich nun die Stromstärke ergibt. Denken wir nochmal an unseren Wasser-Vergleich: Die Spannung entspricht einem bestimmten Wasserdruck im Schlauch. Wieviel Wasser nun fließt, hängt schlicht davon ab, wie durchlässig die Schlauchöffnung ist – oder anders gesagt, wieviel Widerstand sie dem Wasserfluss entgegensetzt. Damit haben wir auch schon die bestimmende Größe im Stromkreis:

Den elektrischen Widerstand (Formelzeichen R). Seine Einheit ist das Ohm[6]. Gehen wir in obigem Beispiel davon aus, dass der Glühdraht im Lämpchen einen Widerstand von 10 Ohm hat, so kann man die Stromstärke nach dem ohmschen Gesetz berechnen, es lautet:

$$I = \frac{U}{R}$$

Daraus ergibt sich bei einer Batteriespannung von 5 Volt ein Strom von 0,5 Ampere. Aus der Formel wird ersichtlich: Möchte man den Strom erhöhen, kann man entweder die Spannung steigern oder den Widerstand verringern.

Widerstand R = 10 Ω	
Spannung U	**Stromstärke I**
0,1 V	0,01 A = 10 mA
1 V	0,1 A = 100 mA
10 V	1 A
100 V	10 A

Spannung U = 5 V	
Widerstand R	**Stromstärke I**
1 Ω	5 A
10 Ω	0,5 A
1000 Ω = 1 kΩ	0,005 A = 5 mA
1000 kΩ = 1 MΩ	0,005 mA = 5 µA

Tabelle 3.2 Verhältnis zwischen Strom, Spannung und Widerstandswert bei konstantem Widerstand (links) oder konstanter Spannung (rechts)

Durch Multiplikation von Strom und Spannung erhalten wir nun noch die Leistung:

$$P = U \cdot I = 5\ V \cdot 0{,}5\ A = 2{,}5\ W$$

Das Lämpchen im Beispiel setzt also eine Energie von 2,5 Watt um. Diese wird sie teilweise als Licht, teilweise als Wärme abgeben. Um den Stromfluss zu regulieren, nutzt man in der Praxis natürlich keine Glühlämpchen, sondern Bauteile, welche als einzige Funktion ihren elektri-

6 benannt nach dem deutschen Physiker Georg Simon Ohm

schen Widerstand haben, deshalb heißen sie auch genauso. Im Stromlaufplan werden sie durch ein simples Rechteck gekennzeichnet.

Abb. 3.4 verschiedene Widerstände und das Symbol im Stromlaufplan

Ihre Hauptaufgabe in unserem Anwendungsgebiet besteht darin, Stromflüsse und Spannungsverteilungen zu beeinflussen. Im Gegensatz zum Glühlämpchen setzen sie sogar die komplette ihnen zugeführte Energie in Wärme um. Üblicherweise ist ihre Heizleistung jedoch vernachlässigbar gering. Den konkreten Wert eines Widerstandes kann man durch Ablesen seiner Farbringe nach einheitlichen Tabellen[7] bestimmen oder ihn einfach mit einem Widerstandsmessgerät ermitteln.

Eine wichtige Schaltung, die Sie zum Verständnis der nachfolgenden Kapitel kennen sollten, ist der Spannungsteiler. Hierbei nutzt man zwei oder mehr Widerstände, um eine Spannung zu verringern.

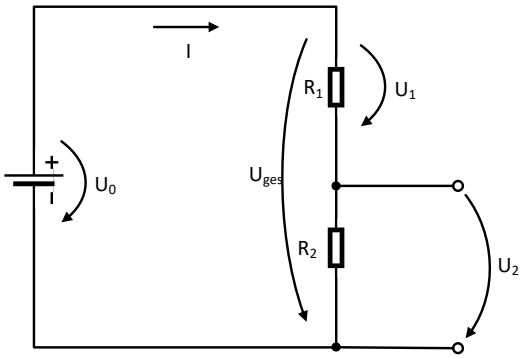


Abb. 3.5 der Spannungsteiler - eine grundlegende Schaltung

7 https://www.digikey.de/de/resources/conversion-calculators/conversion-calculator-resistor-color-code-4-band

Dabei schaltet man die Widerstände in Reihe, also im Stromkreis hintereinander. Da sie von ein und demselben Strom durchflossen werden, ist die Stromstärke in beiden gleich. Die Spannung teilt sich jedoch auf beide auf, und zwar im Verhältnis ihrer Widerstandswerte. Auf oben gezeigten Schaltplan bezogen bedeutet dies, dass sich die abgreifbare Spannung U_2 wie folgt berechnet:

$$U_{ges} = U_0 = U_1 + U_2$$

$$U_2 = \frac{R_2}{R_1 + R_2} \cdot U_{ges}$$

Haben wir also zum Beispiel eine Spannungsquelle von 5 Volt und nutzen Widerstände $R_1 = 22000\,\Omega = 22\,\text{k}\Omega$ und $R_2 = 47000\,\Omega = 47\,\text{k}\Omega$, so ergibt sich nach obiger Formel eine Spannung von

$$U_2 = \frac{47\,\text{k}\Omega}{22\,\text{k}\Omega + 47\,\text{k}\Omega} \cdot 5\,\text{V} = 3{,}4\,\text{V}$$

an den rechten Klemmen. Diese Schaltung eignet sich allerdings nur dann, wenn Sie an den Klemmen keinen oder nur einen verschwindend geringen Strom abgreifen wollen – zum Beispiel, weil Sie den Anschluss mit einem fremden Mikrocontroller verbinden möchten, dessen Eingang nicht für 5 V geeignet ist. Derartige Eingänge haben einen sehr hohen Eingangswiderstand, es fließt also nahezu kein Strom. Anders sieht es aus, wenn Sie mit obiger Schaltung beispielsweise einen weiteren Schaltkreis mit Betriebsspannung versorgen möchten. Da dann der Stromfluss durch die Abgriffklemmen deutlich höher ist, gilt obige Relation nicht mehr. Die Spannung U_2 wird deutlich sinken, da R_1 den Stromfluss begrenzt. Zur Energieversorgung sollten also stets Spannungsquellen (Batterien, Niederspannungsnetzteile oder Spannungsregler) verwendet werden, Spannungsteiler sind dafür nicht geeignet.

Eine besondere Rolle spielen verstellbare Widerstände (sogenannte Potentiometer) als Widerstandsteiler. Sie kennen Sie als Lautstärkeregler an einfachen Radios. Dabei wird ein Schleifkontakt über ein

Widerstandsmaterial geführt, wodurch das Verhältnis von R_1 zu R_2 stufenlos eingestellt werden kann. Der Gesamtwiderstand $R_1 + R_2$ ist dabei durch das verwendete Material vorgegeben und bleibt entsprechend konstant.

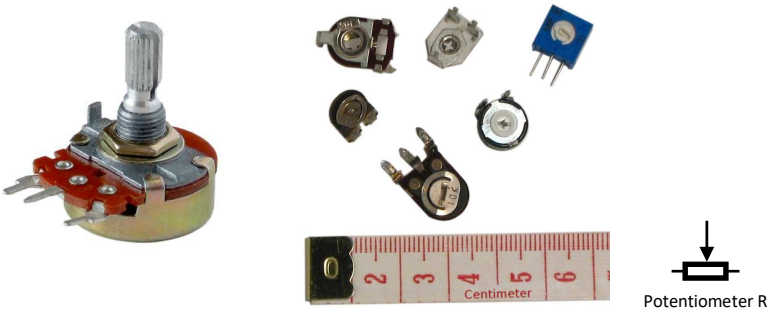


Abb. 3.6 verstellbare Widerstände (Potentiometer)

Des Weiteren sei noch erwähnt, dass auch jede andere Komponente einen elektrischen Widerstandswert aufweist. Manchmal ist er verschwindend gering, zum Beispiel weniger als $0{,}1\,\Omega$. bei einem Stückchen Draht – oder aber absichtlich sehr hoch, zum Beispiel mehrere Megaohm am Eingang von Spannungsmessgeräten. Wir werden uns in späteren Beispielen noch mehrfach damit beschäftigen.

Noch einmal zurück zu unserem einfachen Stromkreis mit dem Lämpchen: Was geschieht, wenn das Lämpchen entfernt und der Stromkreis einfach nur mit einem Draht geschlossen wird? Der Widerstand eines Drahtes ist nahezu 0 Ohm. Nach dem ohmschen Gesetz ergibt sich bei solch einem Kurzschluss ein quasi unendlich hoher Strom. In der Realität wird dieser Strom durch die Spannungsquelle begrenzt. Man kann dieses Verhalten nachbilden, indem man sich vorstellt, dass sich bereits in der Spannungsquelle ein (relativ kleiner) Widerstand befindet; der sogenannte Innenwiderstand R_i.

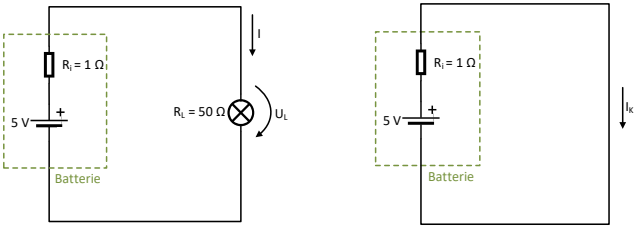


Abb. 3.7 Der modellhafte Innenwiderstand einer Spannungsquelle im Normalbetrieb (links) und bei Kurzschluss (rechts)

Gehen wir bei der obigen Spannungsquelle beispielhaft von einem Innenwiderstand von $R_i = 1\,\Omega$ aus, so ergibt sich für den normalen Betrieb kaum ein Unterschied – der Innenwiderstand bildet zwar einen Spannungsteiler mit dem Glühlämpchen $R_L = 50\,\Omega$, aber aufgrund des im Verhältnis geringen Wertes von R_i liegen am Lämpchen immer noch 4,9 V an. Kommt es aber nun zu einem Kurzschluss wie in der rechten Abbildung, lässt der Innenwiderstand bei 5 V Betriebsspannung nur einen Strom von.

$$I_K = \frac{5\,\mathrm{V}}{1\,\Omega} \cdot 5\,\mathrm{A}\,.$$

Das heißt aber auch, dass dieser Innenwiderstand (und damit die Spannungsquelle) sich mit

$$P = U \cdot I = 5\,\mathrm{V} \cdot 5\,\mathrm{A} = 25\,\mathrm{W}$$

erwärmt, woraus sogar eine Brandgefahr entstehen könnte. Aus diesem Grund sind Kurzschlüsse stets zu vermeiden! In den meisten Geräten sorgen Schmelzsicherungen für entsprechenden Schutz. In der Zuleitung Ihrer Steckdosen sorgt ein Leitungsschutzschalter (umgangssprachlich „Sicherung“) für entsprechende Sicherheit.

3.1.2.2 Kondensator

Ein weiteres oft genutztes Bauelement sind Kondensatoren. Sie können Energie, genauer gesagt elektrische Ladung, speichern. In unserem Vergleich mit dem Wasserkreislauf können Sie sich einen Kondensator

als zylindrisches Gefäß vorstellen, in welches Wasser gedrückt werden kann. Über eine Federmembran kann es dies dann wieder abgeben.

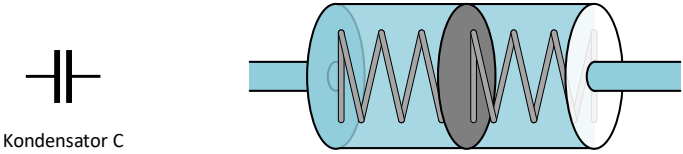


Abb. 3.8 Symbol eines Kondensators und schematischer Vergleich mit einem Membranzylinder

Ebenfalls gebräuchlich ist der nicht ganz korrekte Vergleich mit einem Akku[8]. Kondensatoren finden in der Elektronik vielfach Anwendungen bei hochfrequenten Signalen, Filterschaltungen und zur Vermeidung von Störeinflüssen. Wir werden sie hauptsächlich dazu verwenden, schwankende Spannungen „abzufedern" – wie der Stoßdämpfer in Ihrem Auto. Die folgende Schaltung „glättet" zum Beispiel die von einem Fahrraddynamo erzeugte, üblicherweise stark schwankende Spannung:

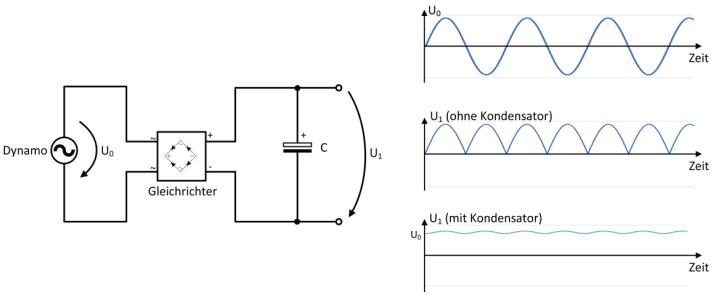


Abb. 3.9 Vereinfachte Darstellung der Spannungsglättung durch einen (Elektrolyt-)Kondensator

8 Ein Kondensator speichert viel weniger Energie, kann jedoch viel größere Ströme verkraften. Ein Akku speichert die Energie zudem chemisch (und nicht elektrostatisch), was ebenfalls zu deutlichen Unterschieden im Verlauf von Strom und Spannung zwischen Kondensator und Akku führt.

Rechts oben ist der Spannungsverlauf an den Anschlussklemmen des Dynamos dargestellt, er gibt Wechselspannung aus. Misst man den Spannungsverlauf nach dem Gleichrichter (ohne Kondensator), wechselt die Spannung zwar nicht mehr die Richtung; aber sie pulsiert noch (mittleres Diagramm). Mit angeschlossenem und ausreichend dimensioniertem Kondensator ergibt sich schließlich ein Verlauf, der nahezu einer Gleichspannung entspricht (unterer Graph).

Die Kapazität, also das „Fassungsvermögen" eines Kondensators wird in der Einheit Farad angegeben. Leider wurde diese nach Michael Faraday benannte Einheit bei ihrer Definition sehr groß gewählt, so dass typische Kondensatoren nur winzige Bruchteile eines Farad erreichen. Übliche Werte reichen von einigen Nanofarad (milliardstel Farad) bis hin zu wenigen Millifarad (tausendstel Farad).

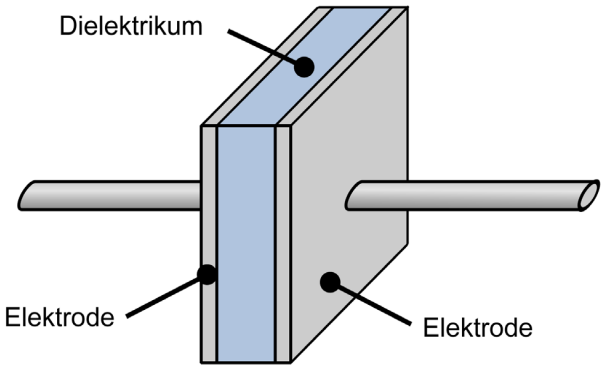


Abb. 3.10 schematischer Aufbau eines Kondensators

Alle Kondensatoren bestehen aus zwei sich gegenüberliegenden leitfähigen Flächen, welche von einem sehr dünnen Isolator (Dielektrikum) getrennt werden. Durch elektrische Feldeffekte lassen sich beim Anlegen einer Spannung Ladungsträger auf einer Seite sammeln und so die Ladung speichern. Bei der Herstellung von Kondensatoren mit relativ großen Kapazitäten (etwa ab 1 Mikrofarad) wird ein Elektrolyt als Isolator verwendet. Dieses Material ermöglicht die hohe Kapazität – leider zum Nachteil einer kürzeren Lebensdauer.

Abb. 3.11 diverse Bauformen von Kondensatoren; ganz rechts Elektrolytkondensatoren

Elektrolyt-Kondensatoren (kurz Elkos) altern (abhängig von Temperatur und Beanspruchung) und verlieren allmählich ihre Kapazität. Sie sind dadurch ein häufiger Grund für den Ausfall elektronischer Geräte. Durch die Wahl von höherwertigen Elkos kann dieser Effekt verringert werden. Bei fast allen Elektrolytkondensatoren ist im Übrigen auf die richtige Polarität zu achten; sie dürfen nur in eine Richtung aufgeladen werden. Bei Plattenkondensatoren mit viel kleinerer Kapazität spielt dies indes keine Rolle.

3.1.2.3 Diode

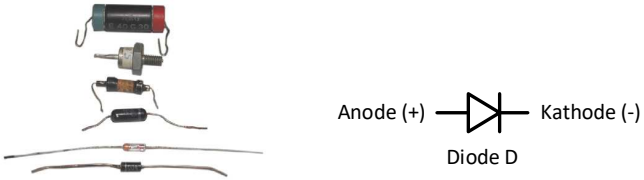


Abb. 3.12 Dioden dienen unter anderem als „elektrische Ventile“. Die Fließrichtung wird durch einen Ring an der Kathodenseite markiert.

Als nächstes richten wir unseren Blick auf die Diode. Dieses meist aus Silizium hergestellte Bauelement ist im wörtlichen Sinne „richtungsweisend“, denn es lässt den Strom nur in Pfeilrichtung passieren. Damit wirkt es wie ein Rückschlagventil in unserem Wasserschlauch-Vergleich. Die Besonderheit der Diode ist zudem ihre nichtlineare Kennlinie – das heißt vereinfacht gesagt, dass sie mit steigender Spannung ihre Durchlässigkeit ändert. In der folgenden Darstellung erkennt man gut den Unterschied zu einem normalen Widerstand (links) – in gleichem Maß wie die angelegte Spannung ändert sich bei ihm auch der Stromfluss. Die daraus entstehende Kennlinie ist gerade, also linear. Die Steilheit ihres Anstieges hängt vom Widerstandswert ab.

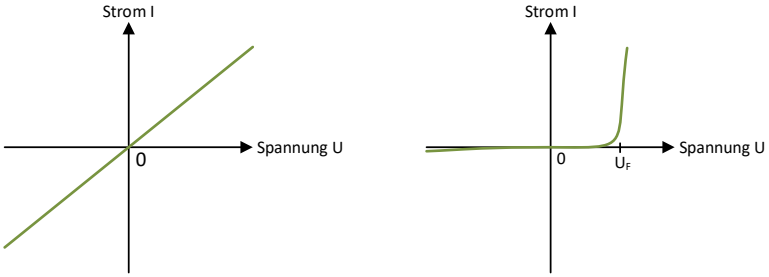


Abb. 3.13 schematischer Vergleich der Kennlinien eines Widerstandes (links) und einer Silizium-Diode (rechts)

Ganz anders ist dies bei der Diode (rechts): Legt man zunächst 0 V an und erhöht die Spannung dann langsam (in Durchlassrichtung), so fließt bis etwa 0,7 V (sogenannte Flussspannung U_F) fast überhaupt kein Strom. Die Diode wirkt in diesem Bereich wie es sehr großer Widerstand. Steigt die Spannung weiter, „öffnet“ die Diode plötzlich und der Strom kann stark ansteigen. Dieses Verhalten führt dazu, dass eine Siliziumdiode immer rund 0,7 V Spannung „schluckt“, was bei einer Reihenschaltung entsprechend berücksichtigt werden muss. In umgekehrter Richtung sperrt sie den Strom dauerhaft (wenn sie nicht durch eine zu hohe Spannung beschädigt wird). Dioden finden ihre Anwendung häufig in Gleichrichtern oder im Zusammenhang mit Spulen, die wir noch kennenlernen werden.

3.1.2.4 *Leuchtdiode*

Eine besondere Art der Diode ist die Leuchtdiode, welche als LED (“light-emitting diode“) besser bekannt sein dürfte. Aufgrund ihrer deutlich besseren Energieeffizienz und der mittlerweile preiswerten Herstellung hat sie in den letzten Jahren klassische Leuchtmittel wie Halogen- oder Leuchtstofflampen vielerorts verdrängt.

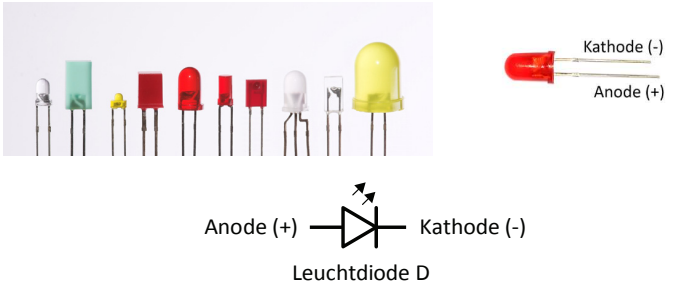


Abb. 3.14 Die Anode ist bei Leuchtdioden am längeren Anschlussdraht erkennbar

Sie verhält sich prinzipiell wie die oben beschriebene Silizium-Diode, jedoch gibt sie zusätzlich Licht einer bestimmten Farbe ab. Ihre Helligkeit ist direkt von der Stromstärke abhängig. Aufgrund ihrer bereits besagten nichtlinearen Kennlinie kann man sie nicht direkt an eine 5V-Spannungsquelle anschließen, da der Strom zu groß würde und die Diode zerstörte. Man nutzt daher einen Vorwiderstand, um den Stromfluss zu begrenzen.

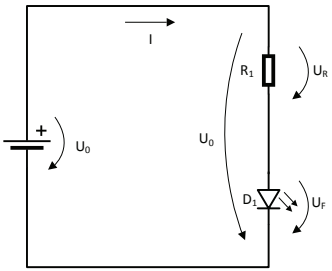


Abb. 3.15 Stromkreis mit einer Leuchtdiode und entsprechendem Vorwiderstand

Um den Wert des benötigten Vorwiderstandes zu berechnen, recherchiert man im Datenblatt der LED zunächst den zulässigen Strom sowie ihre Flussspannung U_F – bei der Siliziumdiode lag diese bei 0,7 V. In unserem Beispiel sei es eine grüne LED mit $U_F = 2{,}2\,\mathrm{V}$ und $I = 10\,mA$. Wir wissen, dass sich die Betriebsspannung U_o in der obigen Reihenschaltung auf U_R und U_F aufteilt.

$$U_o = U_R + U_F$$

Diese Formel lässt sich umstellen zu $U_R = U_o - U_F$, dann lässt sich der Wert des Widerstandes nun berechnen:

$$R = \frac{U_R}{I} = \frac{U_o - U_F}{I} = \frac{5\,\mathrm{V} - 2{,}2\,\mathrm{V}}{0{,}01\,\mathrm{A}} = 280\,\Omega$$

Da in der Praxis nicht alle beliebigen Widerstandswerte verfügbar sind, nimmt man einfach den nächsthöheren erhältlichen Widerstand – in diesem Fall wäre das $330\,\Omega$.

Die Farbe einer LED wird übrigens bestimmt durch die intern verwendeten Werkstoffe, welche auch direkte Auswirkungen auf die Kennwerte haben. So hat eine rote LED üblicherweise eine Flussspannung von etwa 1,9 V, eine blaue jedoch rund 3,2 V. Man könnte beide Farben daher nicht parallelschalten, es sind einzeln abgestimmte Vorwiderstände nötig. Beachtet man dies nicht, „schluckt" eine von beiden den gesamten Strom und könnte dadurch auch beschädigt werden.

3.1.2.5 Transistor

Die fortschreitende Entwicklung der Halbleitertechnik zum Ende der 1940-er Jahre hat unsere heutige miniaturisierte Elektronik erst möglich gemacht. Neben den gerade kennengelernten Dioden stammt noch ein weiteres essenzielles Bauteil aus dieser Zeit: Der Transistor. Man kann ihn als elektronischen Schalter verstehen, der durch einen Stromfluss gesteuert wird.

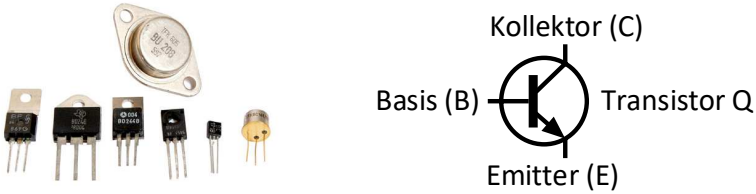


Abb. 3.16 Transistoren gibt es in zahlreichen Bauformen. Im Symbol werden die drei Anschlüsse deutlich.

Ein Transistor besitzt drei Anschlüsse, die mit Basis, Kollektor und Emitter bezeichnet sind. Der Basis-Anschluss dient als Steuer-Eingang. Fließt über ihn ein Strom, so wird auch der Kollektor geöffnet. Der Emitter dient dann als gemeinsamer „Abfluss". Dabei verhält sich der Basis-Anschluss so wie eine bereits beschriebene Siliziumdiode. Die Flussspannung liegt bei etwa 0,7 V. Bei höheren Spannungen reguliert man den Strom über einen Vorwiderstand. Dieser Strom öffnet dann den Kollektor für einen vielfach größeren Strom, übliche Verstärkungsfaktoren liegen bei etwa 500. So genügt also ein Basisstrom von beispielsweise 50 µA (0,05 mA), um einen Kollektorstrom von 25 mA zu gestatten. Steigert man den Basisstrom weiter, erreicht der Kollektorstrom irgendwann seinen Maximalwert (je nach Bauart können das mehrere Ampere sein) – das „Ventil" ist dann sozusagen komplett geöffnet. Wir werden den Transistor stets in dieser binären Funktion (ganz offen oder ganz geschlossen) nutzen, um ihn als Schalter zu verwenden. Die variable Stromverstärkungsfunktion ist aber zum Beispiel Grundlage für die einst weit verbreiteten Transistorradios.

Es sei noch erwähnt, dass sich die eben beschriebenen Punkte auf einen NPN-Transistor beziehen. Beim ebenfalls verfügbaren PNP-Transistor sind alle Strom- und Spannungsrichtungen umgekehrt. Außerdem gibt es noch Feldeffekt-Transistoren, welche nicht durch einen Strom, sondern durch eine Spannung gesteuert werden. Für unsere Anwendungen als Einsteiger genügt es zunächst, wenn wir uns auf den NPN-Transistor beschränken.

3.1.2.6 Spule und Relais

Ein weiteres grundlegendes Element der Elektronik sind Spulen. Sie bestehen meistens aus einem um Ferrit[9] gewickelten Draht. Wird dieser von einem Strom durchflossen, entsteht ein Magnetfeld, welches weitere Effekte mit sich bringt. So kann die im Magnetfeld gespeicherte Energie zum Beispiel genutzt werden, um sogenannte Schwingkreise zu konzipieren, welche elementar für drahtlose Übertragungen sind. Dass Spulen zudem grundlegenden Bestandteile von Transformatoren und Motoren sind, ist sicher bekannt. Für unsere Einsteiger-Anwendungen genügt es jedoch, wenn wir uns auf die Nutzung als Elektromagnet beschränken. Dabei wird das Magnetfeld genutzt, um einen anderen Körper anzuziehen – zum Beispiel einen kleinen Hebel, der einen elektrischen Kontakt schließen und öffnen kann. Diese fernsteuerbaren Schalter werden als Relais bezeichnet. Ihnen kommt in der Geschichte der Elektronik eine wichtige Rolle zu, sie bildeten Mitte des letzten Jahrhunderts die Grundlage für viele Rechenanlagen.

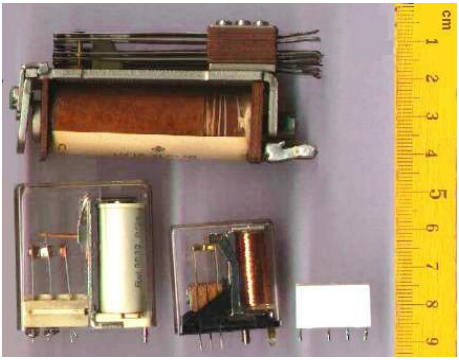


Abb. 3.17 verschiedene Relais, die Spule ist meist deutlich erkennbar

Das Funktionsprinzip ist einfach: Bei angelegter Signalspannung wird die Spule von einem Strom durchflossen. Das entstehende Magnetfeld zieht damit einen kleinen Hebel, den sogenannten Anker, an. Dieser ist mechanisch so gelagert, dass er einen Schaltkontakt eines anderen

9 Ferrit ist eine besonders kohlenstoffarme Eisenart, welche besonders günstige magnetische Eigenschaften aufweist.

Stromkreises schließt oder öffnet. Oft sind auch mehrere Kontakte angebracht, von denen einer geöffnet und der andere geschlossen wird (Wechselschalter). Fließt kein Strom mehr durch die Spule, verschwindet das Magnetfeld und der Anker wird über eine Feder zurück in seine Ausgangsposition gedrückt.

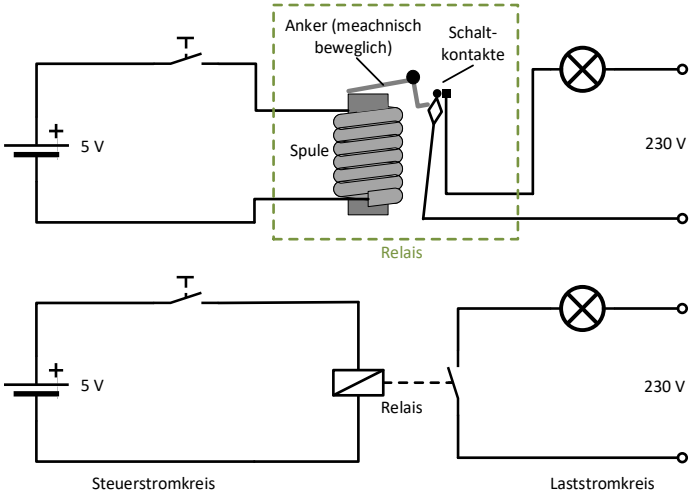


Abb. 3.18 Funktionsprinzip eines Relais (oben) und entsprechende Darstellung im Stromlaufplan (unten)

Bei dieser Verwendung ist die eigentliche Kenngröße von Spulen, ihre Induktivität mit der Einheit Henry, für uns nicht relevant. Da wir die Spulen stets nur mit Gleichspannung ansteuern, genügt es, wenn wir ihren ohmschen Widerstandswert beachten und damit prüfen, wie viel Strom durch sie maximal fließt, wenn wir eine Spannung anlegen.

Nehmen wir beispielsweise ein Standard-Relais, welches wir auch im Kapitel 11.1 noch im Zusammenhang mit dem Arduino verwenden werden. Mit einem Widerstandsmessgerät an den Klemmen seiner Spule ermitteln wir einen Widerstand von 70 Ohm. Gemäß dem ohmschen Gesetz aus dem vorigen Kapitel können wir den Strom bei einer angelegten Spannung von 5 V berechnen:

$$I = \frac{U}{R} = \frac{5\,\text{V}}{70\,\Omega} = 0{,}0714\,\text{A} = 71{,}4\,mA$$

Diese Stromstärke sorgt dafür, dass wir die Spule nicht direkt an einen Ausgang des Arduino anschließen können, denn diese Pins liefern maximal 20 mA. Nun hilft uns jedoch der NPN-Transistor, den wir vorhin bereits kennengelernt haben.

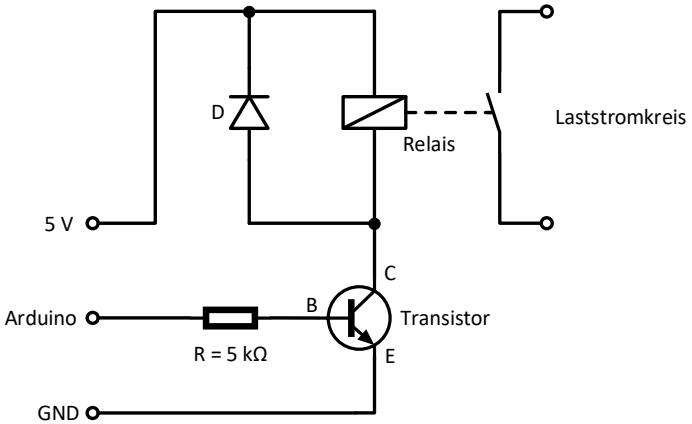


Abb. 3.19 Relais werden in der Regel über einen Transistor angesteuert, um den Mikrocontroller nicht durch zu hohen Stromfluss zu überlasten

Der Vorwiderstand am Basis-Eingang des Transistors wurde in obigem Beispiel so gewählt, dass bei 5 V Spannung vom Mikrocontroller gerade einmal 1 mA Strom in die Basis fließt. Dieser reicht jedoch schon aus, um den Transistor komplett durchzuschalten, so dass durch den Kollektor (C) problemlos der komplette Strom des Relais fließen kann. Bei abgeschaltetem Basis-Strom wird der Kollektor entsprechend blockiert und es fließt auch kein Strom mehr durch das Relais.

Sicher wundern Sie sich, was es mit der zusätzlichen Diode parallel zur Spule auf sich hat. Auf den ersten Blick scheint sie sinnlos, da sie in Sperrrichtung eingebaut wurde und somit auch keinen Strom in dieser

Richtung passieren lässt. Der Grund liegt jedoch in einer Eigenschaft der Spule: Der Aufbau des Magnetfeldes benötigt Energie, deshalb verhält sie sich etwas träge – so ähnlich wie ein schweres Schwungrad. Legt man eine Spannung an, beginnt der Strom erst allmählich durch sie zu fließen, bis er schließlich seinen Maximalwert erreicht. Entfernt man nun die Spannungszufuhr, muss zunächst das Magnetfeld wieder abgebaut werden. Dieser Vorgang erzeugt in der Spule einen Strom, der weiter in die bisherige Stromrichtung strebt – so wie ein Schwungrad durch seine Trägheit auch weiterlaufen will, wenn es nicht mehr angetrieben wird. In unserer Schaltung kann der Strom jedoch nicht mehr weiter durch den Kollektor fließen, wenn der Transistor sperrt. Die Diode sorgt nun dafür, dass dieser sogenannte Selbstinduktionsstrom auf kurzem Weg direkt um die Spule herum fließen kann. Würde man sie weglassen, „prallte" der Strom auf den geblockten Transistor und würde dort eine hohe Spannung erzeugen, die zur Zerstörung des Transistors führen könnte. In Anlehnung an ein Schwungrad mit Freilaufkupplung wird die Diode in dieser Funktion übrigens auch *Freilaufdiode* genannt.

Wenn Sie nach den folgenden Kapiteln Spaß am Experimentieren gefunden haben, bauen Sie obige Schaltung doch einmal nach und verwenden Sie eine LED als Freilaufdiode. Sie werden bemerken, dass sie bei jedem Abschalten des Relais kurz aufblitzt. Die Energie dafür stammt tatsächlich aus dem sich abbauenden Magnetfeld der Relaisspule.

Worin besteht nun der Unterschied zwischen Transistor und Relais, wo wir beides doch als gesteuerte Schalter verwenden wollen? In den Anfängen der Elektronik waren Relais die einzigen verfügbaren elektrisch gesteuerten Schalter – Konrad Zuse baute mit seinem *Z3* sogar einen kompletten Digitalrechner daraus[10]. Mit dem Aufkommen der Röhren- und Transistortechnik wurden sie aus vielen Anwendungen verdrängt. Heute kommen Relais noch dort zum Einsatz, wo entweder hohe Spannungen geschaltet werden sollen und/oder die galvanische Trennung der Stromkreise notwendig ist. „Galvanisch getrennt" bedeutet, dass

10 Einen funktionsfähigen Nachbau können Sie im Deutschen Museum in München besichtigen.

es keine leitende Verbindung zwischen dem steuernden Stromkreis und dem zu steuernden Stromkreis gibt. Besonders wenn darüber verschiedene, örtlich getrennte Anlagen verbunden werden, ist dies sinnvoll. Andernfalls könnten sich aufgrund verschiedener Massepotentiale störende Ausgleichsströme bilden. Bei einem Transistor ist diese galvanische Trennung aufgrund des gemeinsamen genutzten Emitter-Anschlusses nicht gegeben.

3.1.3 Mikrocontroller

Da wir nun die grundlegenden Bauelemente der Elektronik kennengelernt haben, können wir uns dem Gehirn unseres Arduino widmen. Der Mikrocontroller besteht tatsächlich auch aus diesen Grundelementen, hauptsächlich Transistoren – man könnte ihn also sogar selbst nachbauen, allerdings benötigt man dafür zehntausende Transistoren und natürlich den entsprechenden Platz, um sie zu verdrahten. Die moderne Halbleitertechnik macht es möglich, diese riesige Schaltung auf einer Siliziumscheibe mit nur wenigen Millimetern Durchmesser unterzubringen. Ein Gehäuse aus meist schwarzem Kunststoff oder Keramik schützt sie vor der Außenwelt und gibt den Mikrochips ihr typisches äußeres Erscheinungsbild.

Abb. 3.20 Der im Arduino verwendete ATmega328P-Mikrocontroller in dem für integrierte Schaltkreise typischen Dual-Inline-Package-Gehäuse

Die Hundertschaften von Transistoren sind dabei so verschaltet, dass der Chip alle wesentlichen Fähigkeiten eines Computers beherrscht: Es gibt einen kleinen Speicher für Zwischenergebnisse und einen deutlich größeren für das Programm. Das sogenannte Steuerwerk lädt die programmierten Befehle aus dem Speicher und arbeitet sie Schritt für Schritt ab. Ein Rechenwerk übernimmt dabei arithmetische Berechnungen, deren Ergebnisse über Datenleitungen wieder im Speicher abgelegt werden können. Die gewünschte Stelle im Speicher wird über Adressleitungen ausgewählt. Zudem gibt es Steuerleitungen, die den Programmablauf beeinflussen können.

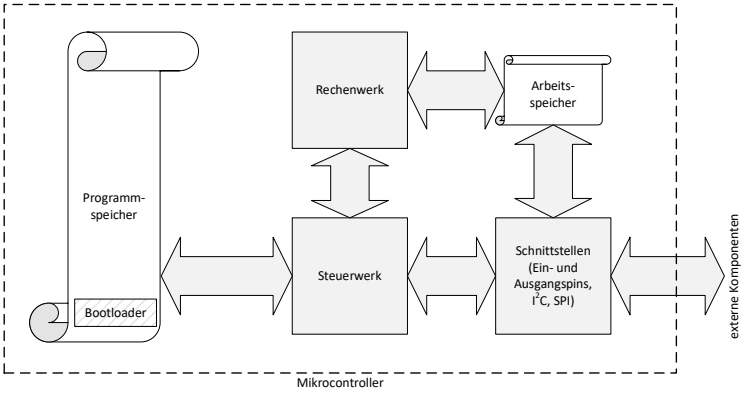


Abb. 3.21 vereinfachter schematischer Aufbau eines Mikrocontrollers

In einem speziell geschützten Speicherbereich ist der Bootloader hinterlegt. In älterer deutscher Literatur findet man dafür auch die Übersetzungen „Urlader“ oder „Startprogramm“. Direkt nach dem Einschalten hat der Mikrocontroller nämlich ein Problem: Im vorherigen, spannungslosen Zustand konnten sich in den verschiedenen Bereichen (Arbeitsspeicher, Rechenwerk, Steuerwerk usw.) willkürliche Zustände einstellen, die Leitungen und Register enthalten zufällige Daten. Der Chip ist sozusagen völlig durcheinander und weiß nicht, was er tun soll. Eine spezielle Schaltung zwingt ihn nun, die im Bootloader hinterlegten Befehle auszuführen. Diese räumen das Chaos auf und starten daraufhin das eigentliche, vom Anwender hinterlegte Programm.

Wichtig im Zusammenhang mit Mikrocontrollern sind deren verschiedene Betriebsspannungen. Während unser Arduino-Chip mit 5 Volt arbeitet, gibt es auch andere Varianten, die zum Beispiel auf 3,3 V ausgelegt sind und durch eine Spannung von 5 V bereits beschädigt werden könnten. Dies gilt es insbesondere dann zu beachten, wenn man verschiedenartige Mikrocontroller miteinander verbinden möchte. Näheres dazu im Kapitel 16.2.2.5.

3.1.4 Die Platine des Arduino UNO

Werfen wir nun einen ersten Blick auf die Platine des Arduino UNO, um einen Überblick über dessen Aufbau zu erhalten.

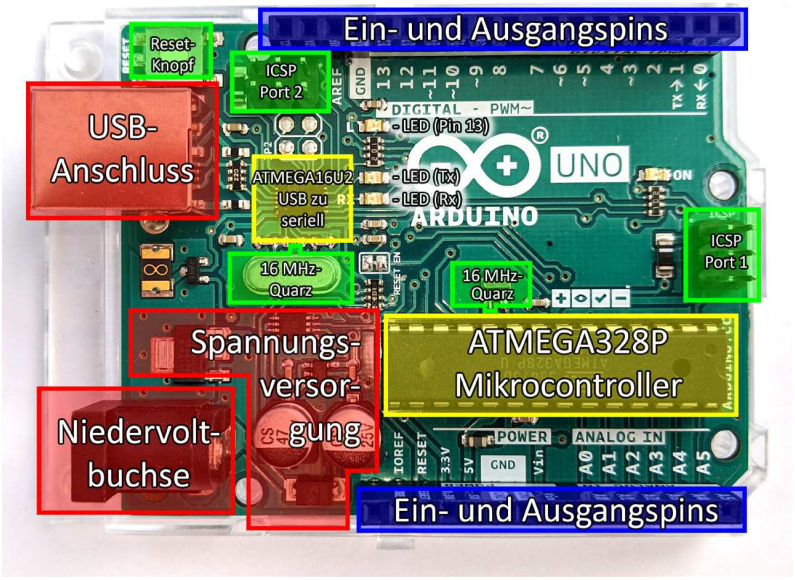


Abb. 3.22 die Funktionsgruppen der Arduino-Uno-Platine

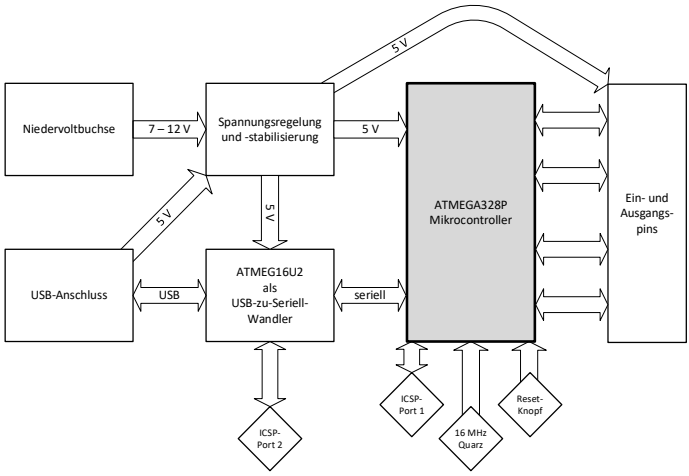


Abb. 3.23 Schematischer Aufbau der Arduino-Uno-Platine

Unser Mikrocontroller ist der bereits erwähnte ATmega328P, der sich in der Arduino-Welt zum Basis-Model entwickelt hat. Die meisten Anschlusspins der Platine sind tatsächlich direkt mit ihm verbunden. Eine kleine Besonderheit gibt es an Pin 13: Hier ist zusätzlich eine kleine LED angebracht. Dies wird uns den Einstieg ins Programmieren erleichtern, da wir mit diesem Lämpchen bereits experimentieren können, ohne irgendein zusätzliches Kabel anbringen zu müssen. Profis nutzen diese LED zum Beispiel als Indikator, an welcher Stelle sich der Programmablauf gerade befindet.

Des Weiteren fällt auf, dass die Platine sogar einen zweiten Mikrocontroller besitzt, einen ATmega16U2. Dieser hat die Aufgabe, das USB-Signal des Computers in das vom ATmega328P benötigte serielle Signal umzuwandeln. Er dient also sozusagen als Dolmetscher zwischen unserem eigentlichen Arduino und dem PC, somit können wir sehr bequem per USB auf die Platine zugreifen. An die serielle Sende- und Empfangsleitung zwischen beiden Mikrocontrollern ist jeweils eine LED angeschlossen (beschriftet mit Tx und Rx) – durch ihr Flackern kann ein gerade stattfindender Datentransfer beobachtet werden.

Alternativ zur Programmierung über USB ist aber auch der direkte Anschluss per ICSP möglich. Dieser sogenannte *In-Circuit-Serial-Programmer* ist ein spezieller Adapter, der auf die oben markierten ICSP-Pins der Platine gesteckt wird (siehe Kapitel 17.3). Sie sind direkt mit den Mikrocontrollern verbunden und erlauben entsprechend unmittelbaren Zugriff – das ist beispielsweise dann nötig, wenn der oben erläuterte Bootloader verändert werden soll. Für uns als Einsteiger ist dies jedoch nicht von Belang, wir können die ICSP-Anschlüsse also ignorieren.

Der einzige auf dem Board befindliche Knopf ist direkt mit dem Reset-Pin des Mikrocontrollers verbunden. Durch Knopfdruck verbindet er diesen Pin mit Masse, somit wird der Arduino zurückgesetzt und startet das Programm von vorn, so als hätte man ihn kurz von der Betriebsspannung getrennt.

Neben den bereits beschriebenen LEDs an Pin 13 und den seriellen Datenleitungen gibt es außerdem noch eine weitere Leuchtdiode auf der Platine: Sie zeigt schlicht an, ob die Betriebsspannung anliegt (Beschriftung „ON“).

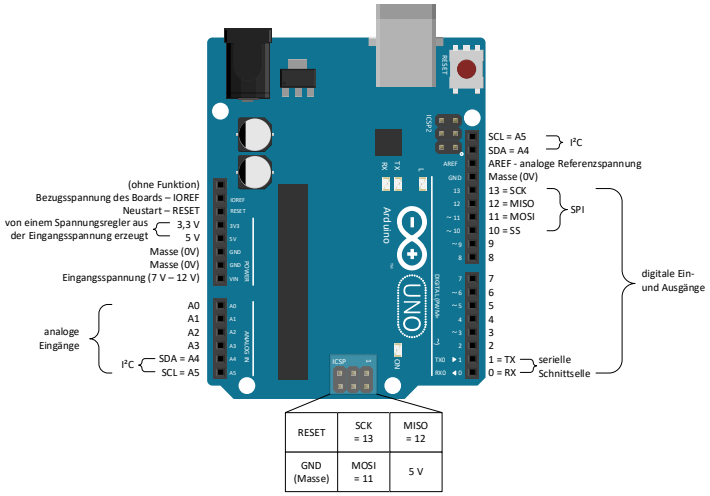


Abb. 3.24 Überblick der Anschlusspins

Es fällt auf, dass viele Pins mehrfach vorhanden sind, so zum Beispiel die Datenschnittstellen I²C oder SPI. Pins mit gleicher Bezeichnung sind auf dem Board direkt miteinander verbunden. Die Funktionen der einzelnen Ein- und Ausgangspins und Schnittstellen werden wir später noch kennenlernen. Die anderen Pins dienen überwiegend der Spannungsversorgung des Arduinos oder externer Komponenten. Dabei gibt es einen wichtigen Unterschied: Der Pin VIN ist direkt mit der auf dem Bord befindlichen Niedervoltbuchse verbunden. Hier dürfen Spannungen zwischen 7 Volt und 12 Volt angelegt werden. Auf dem Board wird diese Spannung einem Regler zugeführt, welcher daraus 5 V (für den Mikrocontroller) sowie 3,3 Volt (für optionale externe Komponenten) generiert. Dieser Regler „schluckt" prinzipbedingt etwas mehr als 1 Volt, daher wurde die untere Grenze der Eingangsspannung in der Spezifikation auf 7 Volt festgelegt. Möchten Sie den Arduino an einer Spannungsquelle mit nur 5 Volt betreiben, verbinden Sie diese nicht mit VIN, sondern direkt mit 5 V und umgehen dadurch den Spannungsregler.

Der Pin IOREF führt per Spezifikation immer die Spannung, welche auch vom Mikrocontroller verwendet wird. Es gibt seltenere Boards, welche einen Mikrocontroller mit 3,3 V statt 5 V verwenden. Durch das Abgreifen der IOREF-Spannung können externe Zusatzmodule dies erkennen und sich entsprechend darauf einstellen.

3.2 Praktische Werkzeuge und Wissen

3.2.1 Steckbrett

Im Folgenden wollen wir noch einen Blick auf einige praktische Werkzeuge und Hilfsmittel werfen, die uns das Experimentieren erleichtern. Der Liebling eines jeden Elektronikbastlers ist zweifellos die Steckplatine, oft auch Breadboard genannt. Diese kleine Kunststoffplatine gibt es in diversen Ausführungen und Größen. Sie bestehen jeweils aus einem Lochraster, in dem die Federkontakte gruppenweise verbunden sind (siehe folgende Abbildungen). Das Raster orientiert sich dabei am international üblichen 0,1-Zoll-Standard – dieser Lochabstand von 2,54

Millimeter sorgt dafür, dass fast alle Komponenten, die auch für die professionelle Platinenmontage geeignet sind, mühelos eingesteckt werden können. Drahtbrücken ermöglichen die direkte Verbindung zwischen den einzelnen Klemmengruppen. Mit den meisten Platinen werden auch vorkonfektionierte Drahtstücke geliefert, mit denen dann weitere Platinen oder andere Komponenten außerhalb der Steckplatine verbunden werden können.

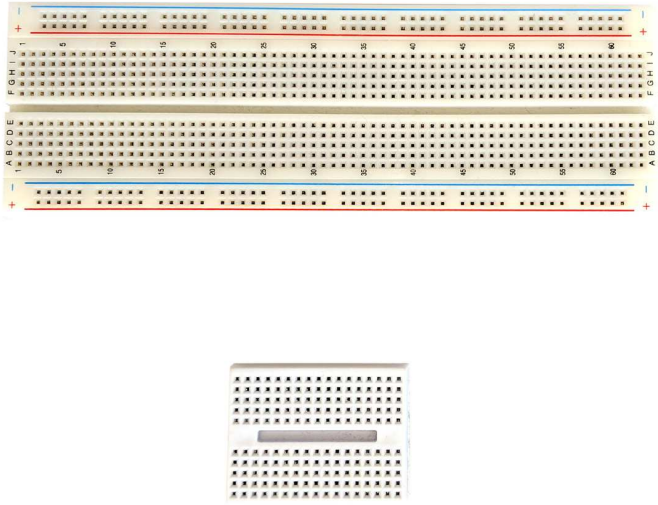


Abb. 3.25 Steckbretter (Breadbords) verschiedener Größen

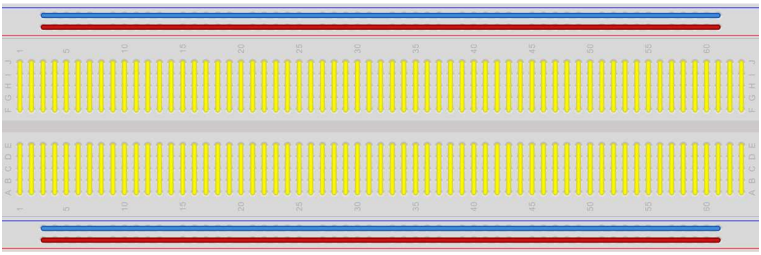


Abb. 3.26 Großes Breadboard mit markierten inneren Verbindungen. Die langen Querverbindungen am oberen und unteren Rand werden üblicherweise für die Spannungsversorgung genutzt.

So kann man Elektronikkomponenten unkompliziert durch Einstecken miteinander verbinden, es ist kein Löten erforderlich. Einfache Schaltungen lassen sich so schnell und effizient zusammenbauen und variieren. In der Mitte besitzen die meisten Ausführungen eine Trennung, die es ermöglicht, die üblicherweise zweireihig konstruierten Mikrochip-Bauelemente einzustecken und alle ihre Anschlüsse zu erreichen.

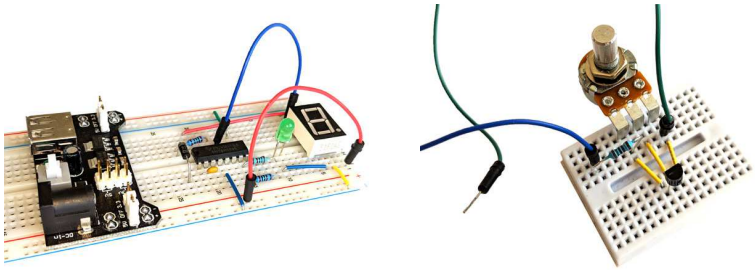


Abb. 3.27 zwei Breadboards mit beispielhaften Schaltungen, ganz links ein optionales Aufsteckmodul zur Spannungsversorgung

Breadboards sind in zahlreichen verschiedenen Größen erhältlich und oft auch über seitliche Nuten zusammensteckbar. Wenn Sie die Beispiele in diesem Buch nachvollziehen möchten, genügt ein Standard-Steckbrett mit Maßen von ca. 173 mm x 65 mm, in vielen Fällen reicht sogar ein deutlich kleineres.

3.2.2 Lötkolben

Für unsere Experimente in diesem Buch wird es genügen, die entsprechenden Komponenten mit dem Breadboard zu verbinden. Sollten Sie jedoch ein konkretes Projekt in einen dauerhaften Einsatz überführen wollen, kann dieser Aufbau aufgrund der relativ leicht zu lösenden Steckverbindungen zum Beispiel beim Transport ein Problem darstellen.

Zur dauerhaften und zuverlässigen Verbindung von elektronischen Komponenten wird deshalb üblicherweise das Lötverfahren angewendet. Das Lot besteht dabei aus einer Legierung verschiedener Metalle, welche sich durch Erhitzen verflüssigt und dann wie ein Kleber die Anschlussdrähte der Bauelemente miteinander oder mit einer Platine verbindet. Die in der Elektronik verwendeten Weichlote haben eine Schmelztemperatur von weit unter 450 Grad Celsius, diese liegt damit deutlich niedriger als die Schmelztemperatur der zu verbindenden Metallflächen. Die Legierungsbestandteile werden entsprechend ihrer mengenmäßigen Anteile in der Bezeichnung vermerkt. Das sehr häufig verwendete „SN99,3CU0,7"-Weichlot besteht also zu 99,3 Prozent aus Zinn und zu 0,7 Prozent aus Kupfer. Der stets hohe Zinnanteil hat den Weichloten auch ihre umgangssprachliche Bezeichnung „Lötzinn" eingebracht. Für den Hobbymarkt wird es meist in der Form eines auf eine Spindel gewickelten Lötdrahtes angeboten.

Eine wichtige und oft auch beworbene Eigenschaft von marktüblichen Weichloten ist ihr eutektisches Verhalten. Vereinfach gesagt bedeutet dies, dass beim Erhitzen alle Bestandteile zur gleichen Zeit schmelzen und beim Abkühlen zugleich erstarren. Nur dann lassen sich auch durch Laien stabile und sichere Verbindungen herstellen.

Abb. 3.28 Lötkolben, Hobby-Lötstation und semiprofessionelle Lötstation

Abb. 3.29 handelsübliches Lötzinn

Möchten Sie eine Verbindung löten, benötigen Sie lediglich das Lötzinn und einen Lötkolben, also einen stiftförmigen Heizstab zum Schmelzen des Lotes. Einfache Ausführungen sind bereits für unter 10 Euro zu haben und genügen völlig für erste Lötversuche. Sie haben jedoch keine genaue Temperaturregelung, somit hängt ihre Temperatur von diversen Einflüssen (Umgebungstemperatur, Aufheizzeit) ab und liegt meist im Bereich von ca. 350 Grad. Höherwertige Geräte haben oft einen einstellbaren Temperaturregler und austauschbare Kolbenspitzen, um sowohl feine als auch großflächige Verbindungen komfortabel löten zu können.

Obwohl die bereits angesprochene „SN99,3CU0,7"-Legierung einen Schmelzpunkt von 221 Grad Celsius besitzt, sollten Sie Ihren Lötkolben mindestens auf etwa 340 Grad einstellen, falls er regelbar ist. Der Temperatursensor sitzt in der Regel nicht direkt an der Spitze, diese ist dadurch bereits etwas kühler. Außerdem müssen Sie mit dieser Wärme nicht nur das Lot aufschmelzen, sondern auch die zu verbindenden Metallstücke über die Schmelztemperatur des Lotes hinaus erhitzen, damit sie benetzt werden können. Wird diese Temperatur nicht erreicht, schmilzt zwar das Lot, aber es perlt von den Kontaktflächen einfach ab und bildet keine stabile leitende Verbindung aus. Bei zu hoher Temperatur oder deutlich zu langer Kontaktzeit können empfindliche

Komponenten (Mikrochips, Elektrolytkondensatoren) Schaden nehmen.

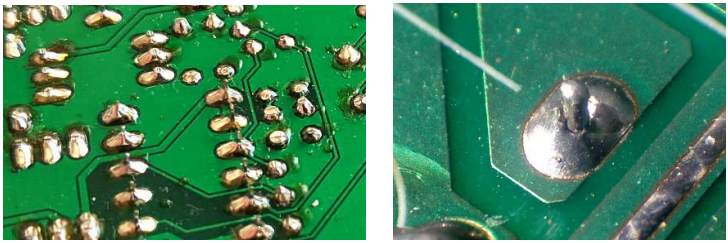


Abb. 3.30 sauber ausgeführte Lötstellen (links) und mangelhafte Lötstelle (rechts); hierbei hat das Lötzinn nur die Platine, aber nicht den Anschlussdraht benetzt - die Verbindung ist dadurch instabil

Der Vorgang selbst ist relativ einfach: Wenn der Lötkolben seine Temperatur erreicht hat, genügt es, die Kontaktflächen durch Berühren mit der Lötspitze für etwa ein bis zwei Sekunden zu erhitzen und dann gleichzeitig den Lötdraht zuzuführen. Dabei ist es hilfreich, wenn man die zu verlötenden Komponenten über eine Klammer oder „Dritte Hand“ am Tisch oder einer Werkbank befestigen kann, da man je eine Hand für den Lötkolben und das Lot benötigt. Je nach verwendeten Materialien können am Anfang einige Versuche nötig sein, bis man die ideale Kontaktzeit und Temperatur (falls einstellbar) findet. Legierungsreste an der heißen Lötspitze lassen sich vorsichtig mit einem feuchten Tuch oder Schwamm abstreifen, höherwertige Systeme bieten dafür einen Metallschwamm.

Falls Sie eine bereits vorhandene Lötstelle erweitern oder verändern möchten, empfiehlt es sich übrigens immer, noch ein wenig neues Lot hinzuzugeben – auch wenn bereits genug vorhanden war. Im Lötdraht ist ein sogenanntes Flussmittel enthalten, welches die Benetzungseigenschaften der Legierung deutlich verbessert. Es verbraucht sich jedoch beim Löten und muss immer neu zugeführt werden. Für den professionellen Einsatz ist es daher auch separat erhältlich.

Um überschüssiges Lot zu entfernen, können Sie es bei geringen Mengen einfach mit der heißen Spitze des Lötkolbens von der entsprechenden Stelle abstreifen. Größere Mengen entfernen Sie am besten mit einer Entlötsaugpumpe. Diese funktioniert ähnlich wie eine Spritze, mit der Sie durch das Hochziehen des Kolbens Wasser einsaugen können. Bei der Entlötsaugpumpe wird eine Feder mit dem Kolben gespannt. Wenn das Lot durch die Lötspitze erhitzt wurde, setzt man die Saugpumpe rasch auf die entsprechende Stelle (bevor das Lot wieder erstarrt) und löst die Feder per Knopfdruck aus. Der Kolben schnellt nach oben und der entstehende Unterdruck saugt das noch flüssige Lot von der zu reinigenden Stelle ab.

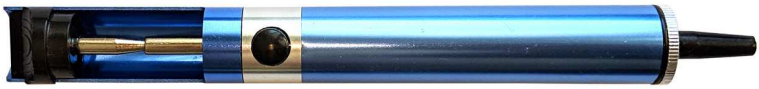


Abb. 3.31 Entlötsaugpumpe

Früher übliche Legierungen bestanden übrigens oft nur zu 60 Prozent aus Zinn und stattdessen bis zu 40 Prozent aus Blei. Aufgrund der Gesundheitsgefahren wurden Werkstoffe mit derart hohem Bleianteil von der EU mittels der RoHS[11]-Richtlinie jedoch verboten und dürfen seit 2019 nicht mehr an Privatanwender verkauft werden. Vielerorts gibt es jedoch noch alte Bestände von verbleitem Lötzinn – sollten Sie es verwenden wollen, können Sie die Temperatur des Lötkolbens um ca. 40 Grad absenken.

11 *Restriction of Hazardous Substances* – Verbot gefährlicher Substanzen

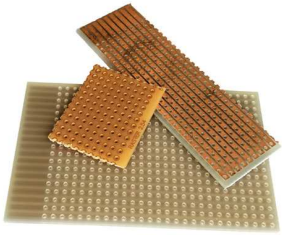


Abb. 3.32 Lochrasterplatinen mit Streifen- und Punktraster

Es sind Experimentierplatinen (auch Lochrasterplatinen genannt) erhältlich, mit denen Sie einfach Prototypen herstellen können. Die Bauelemente werden von der Oberseite hindurchgesteckt und lassen sich auf der Unterseite mit vorgedruckten Kupferflächen verlöten. Die Verbindungen der Elemente untereinander können Sie durch kurze Drahtstücke anlöten. Siehe dazu auch Kapitel 18.

3.2.3 Zangen und andere Werkzeuge

Prinzipiell ist für die Hobby-Bastelei mit Arduino und Breadboard kein spezielles Werkzeug notwendig. Wenn Sie jedoch Gefallen am Erschaffen kleiner elektronischer Wunderwerke finden und Ihre Projekte immer weiter ausbauen, werden Sie einige kleine Helferlein durchaus zu schätzen lernen.

Abb. 3.33 automatische Abisolierzange

Eine Abisolierzange macht das Konfektionieren von losen Drahtstücken zum Kinderspiel. In der Standard-Variante stellen Sie über eine Abstandsschraube den Drahtdurchmesser ein, zwei scharfe Kanten zerschneiden beim Herausziehen des Drahtes dann die Isolierung und entfernen sie. Die etwas teureren automatischen Abisolierzangen führen sogar den kompletten Vorgang mit nur einer Handbewegung durch, außerdem lässt sich an ihnen auch millimetergenau einstellen, wie weit der Draht abisoliert werden soll.

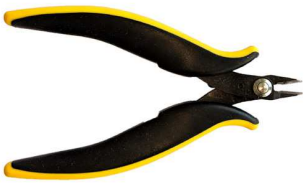


Abb. 3.34 Präzisions-Seitenschneider

Zum Kürzen von Anschlussdrähten bei Widerständen, LEDs und anderen Komponenten bewährt sich ein kleiner Seitenschneider. Beim Kauf sollten Sie darauf achten, dass es sich um einen Präzisions-Seitenschneider handelt. Bei diesem sitzt die Schneidkante bündig am Metallrand, so dass sich Drähte an ebenen Flächen (zum Beispiel Leiterplatinen) bündig abschneiden lassen. Ebenfalls empfehlenswert sind kleine Elektronikzangen, mit denen sich die Anschlussdrähte von Bauteilen mühelos in die gewünschte Form biegen lassen. Zudem helfen sie, wenn sich ein Bauteil auf einem voll bestückten Breadboard einmal schlecht greifen lässt.

Bei allen Werkzeugen sind leitende Griffhüllen empfehlenswert. Sie vermeiden elektrostatische Aufladungen (indem sie die Ladung langsam ableiten). Dies gilt allerdings nur für unseren Hobby-Bereich mit Spannungen in der Größenordnung von 5 Volt. Bei Arbeiten an möglicher Netzspannung sind isolierende Griffhüllen zwingend vorgeschrieben. Wie bereits bemerkt, sollten Sie daher an Projekten mit hohen Spannungen nur arbeiten, wenn Sie die entsprechende Qualifikation besitzen.

Wenn Sie mit einer größeren Anzahl von Widerständen oder Dioden arbeiten, müssen Sie womöglich viel Zeit damit verbringen, die Anschlussdrähte rechtwinklig zu verbiegen, um sie in ein Breadboard oder durch eine Platine stecken zu können. Abhilfe schafft hier eine Biegehilfe. In dieses keilähnliche Kunststoffteil lassen sich die Bauelemente einfach einlegen und mit einer simplen Handbewegung die Drähte passgenau biegen.

Abb. 3.35 Die Biegehilfe erleichtert bei größeren Projekten das Konfektionieren elektronischer Komponenten

Sollten Sie häufiger einzelne Drähte verlöten wollen, gibt es dafür auch eine Unterstützung: Die sogenannte „Dritte Hand“ ist ein kleines Gestell mit flexibel positionierbaren Klammern, an denen Sie Drähte, Platinen oder Bauteile zueinander ausrichten können, um beide Hände für das Löten frei zu haben. Meist gehört auch noch eine Lupe zu dem Helferlein.

Abb. 3.36 Eine „dritte Hand“ kann besonders beim Löten wertvolle Hilfe leisten

3.2.4 Messgeräte

Je komplexer Ihre Projekte werden, um so schwieriger kann sich auch die Fehlersuche gestalten. Messgeräte können Sie bei dieser Arbeit unterstützen. So sollte ein kleines Vielfachmessgerät (oft auch als Multimeter bezeichnet) zur Ausrüstung eines jeden Elektronikbastlers gehören. Sie können damit mühelos feststellen, ob und welche Spannung an einem Anschluss anliegt. Außerdem können Sie Verbindungen prüfen und Widerstandswerte messen. Simple Einsteigergeräte sind ab 10 Euro erhältlich, längere Freude haben Sie mit Geräten ab etwa 30 Euro.

Abb. 3.37 digitales Vielfachmessgerät

Wenn Sie sich später an das Feld der Datenübertragung wagen oder Signale mit einer gewissen Frequenz (zum Beispiel Infrarot-Fernbedienung) analysieren möchten, könnte ein Oszilloskop für Sie nützlich

sein. Es stellt den zeitlichen Verlauf eines Signals grafisch dar – so wie ein Elektrokardiogramm beim Hausarzt. Noch bis vor zehn Jahren waren solche Geräte nur als Laborutensilien verfügbar, der Preis war entsprechend hoch.

Abb. 3.38 labortaugliches Oszilloskop

Mittlerweile erhalten Sie bereits für unter 50 Euro mikroprozessorbasierte Einplatinengeräte. Diese „Mäusekinos“ sind nicht kalibriert und gewiss sehr eingeschränkt, so können sie in der Regel nur Signale bis zu einer Frequenz von 200 Kilohertz darstellen – für einfache Hobby-Anwendungen ist dies jedoch völlig ausreichend. Als nächste Preisstufe gibt es Geräte mit nahezu labortauglicher Messgenauigkeit, allerdings ohne Display. Diese Oszilloskop-Adapter werden per USB mit einem Computer verbunden, eine mitgelieferte Software übernimmt die Aufbereitung und Darstellung der Daten. Die Kosten liegen mit etwa 150 Euro im semiprofessionellen Bereich. Laborgeräte mit umfangreichen Analysemöglichkeiten und genauer Kalibrierung sind aktuell ab etwa 500 Euro zu haben.

Kapitel 4
Grundlagen des Programmierens

4.1 Die Struktur eines Programmes

Bevor wir mit dem Arduino unsere ersten Experimente starten können, müssen wir uns noch mit der Programmierung beschäftigen. In einem früheren Kapitel hatten wir bereits den Begriff *Algorithmus* kennengelernt und ihn mit einem Kochrezept verglichen. Algorithmen, also Handlungsvorschriften, sind Hauptbestandteile von Computerprogrammen. Sie sind die Schritt-für-Schritt-Anleitung zur Lösung eines Problems. Und unser Mikrocontroller wird sie später ebenso Schritt für Schritt abarbeiten.

Stellen Sie sich vor, Sie werden von einem Freund nach Ihrem Lieblingsrezept gefragt – womöglich ist es ein schmackhafter Eintopf. Sie werden zunächst die Zutaten aufzählen, danach werden Sie ihn vielleicht darauf hinweisen, dass bei der doppelten Personenzahl alle Mengenangaben zu verdoppeln sind. Anschließend beschreiben Sie, in welcher Reihenfolge die Zutaten hinzugegeben und gekocht werden. Nochmal umrühren, abschmecken und fertig! Das war Ihr Eintopf-Algorithmus.

Natürlich kann unser Arduino nicht direkt mit der Nahrungszubereitung loslegen. Aber vielleicht möchten Sie, dass er eine bestimme Eingabezahl (Zutat) nimmt, sie einer bestimmten Berechnung unterzieht (kocht) und dann das Resultat präsentiert? Auch dafür müssen Sie sich zunächst überlegen, wie Sie den ganzen Vorgang in kleinere Teilvorgänge zerlegen können. Wir werden in diesem Kapitel noch einige kleine Beispiele dafür kennenlernen.

Ein wesentlicher Verdienst des Mitbegründers der heutigen Arduino IDE Hernando Barragán ist es, dass die grundlegende Struktur eines Arduino-Programmes sehr simpel ist. Er sorgte dafür, dass verwirrende

oder irrelevante Elemente und Einstellungen vor dem Nutzer zunächst ausgeblendet werden, um den Einstieg zu erleichtern. Deshalb sieht der Programmcode eines neuen Arduino-Projektes zunächst so aus:

```
1  void setup()
2  { }
3
4  void loop()
5  { }
```

Man könnte diese Zeilen auch als Grundgerüst unserer Programme bezeichnen. Auf den Begriff `void` sowie die Klammern kommen wir später noch zu sprechen. An diesem ersten Beispiel sollen Sie nur sehen: Es gibt grundlegend stets zwei verschiedene Programmteile. Im `setup()`-Bereich legen Sie fest, was während der Initialisierung, also während des „Aufwachens" Ihres Programmes, passieren soll. Dieser Programmbereich wird nur einmal durchlaufen, und zwar direkt nach dem Einschalten[1]. Üblicherweise werden Sie dem Prozessor hier mitteilen, welche Anschlüsse benötigt werden und wie diese beschaltet sind. Außerdem werden Sie Verbindungen zu eventuell angeschlossenen Modulen herstellen. Wir schauen uns das alles noch im Detail an, wenn wir es benötigen.

Beim Kochen würden Sie im „Setup" Ihren Küchentisch freiräumen, den Topf bereitstellen und die Zutaten parat legen.

Im zweiten Bereich, dem `loop()`, beschreiben Sie die eigentliche Arbeit des Programmes – also das, was Ihre kleine Maschine immer und immer wieder durchführen soll, solange sie eingeschaltet ist. Diese Befehle werden in einer ständigen Schleife wiederholt, daher auch ihr englischer Name. In unserem Eintopf-Beispiel käme hier nun das eigentliche Rezept – welches man so lange neu kochen kann, bis alle satt sind. Ausgehende Zutaten kann man neu kaufen, weitere hungri-

1 Falls Sie es ganz genau wissen wollen: Nach dem Einschalten wird zunächst natürlich der Bootloader ausgeführt und dann erst die Setup-Befehle. Für uns als Einsteiger ist der Bootloader jedoch quasi unsichtbar, wir können ihn ignorieren.

ge Gaumen finden sich in der Nachbarschaft – die Schleife muss also nie enden. Natürlich kann man auch mal eine Pause einlegen – ob nun ein paar Sekunden oder zwei Wochen, ist im Programmcode kaum ein Unterschied. Die Anweisungen dafür werden Sie in diesem Kapitel kennenlernen. Das obige Beispiel zeigt nur die grundlegende Struktur und ist ansonsten ein leeres Blatt Papier, dementsprechend hat es keine Funktion.

4.2 Blink

Das soll sich nun ändern – mit unserem ersten funktionsfähigen Programm. Öffnen Sie die Arduino IDE und klicken Sie auf *Datei -> Beispiele -> 01. Basics -> Blink*

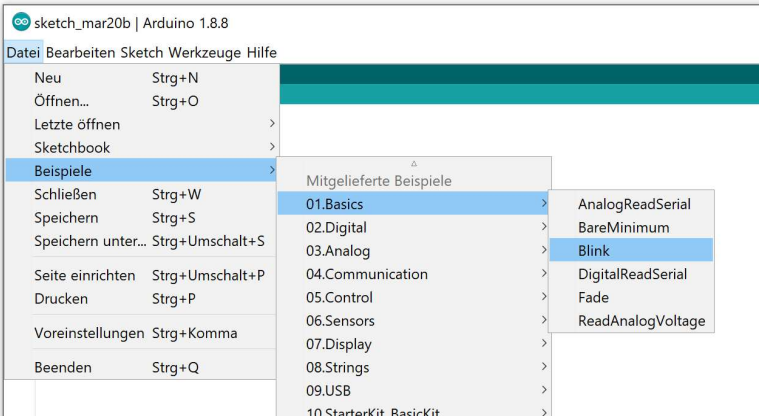


Abb. 4.1 Das mitgelieferte Blink-Beispiel eignet sich ideal als Einstieg

In der Software sind einige Beispielprogramme hinterlegt, Blink ist davon das einfachste. Bevor wir auf die Funktionsweise eingehen, lassen Sie uns zunächst einfach testen, was dieser *Sketch* – so nennt man Programmcode in der Arduino-Welt – bewirkt. Schließen Sie Ihren Arduino per USB an Ihren Computer an und laden Sie den Sketch per Menübefehl (*Sketch -> Hochladen*) auf den Mikrocontroller. Nach wenigen

Sekunden sollte eine der auf dem Board befindlichen LEDs (in der Nähe des Pins 13, beschriftet mit „L“) im Sekundentakt blinken.

Um die weiteren Abschnitte zu verstehen, müssen wir nun erst einmal klären, was genau passiert, wenn Sie auf „Hochladen“ klicken. Man könnte zunächst vermuten, dass der Programmcode (oft auch Quellcode genannt) einfach als Textdatei auf den Chip kopiert wird, etwa so wie bei einem USB-Stick. Doch das würde nicht funktionieren: Unser Mikrocontroller kann mit den Textbefehlen nichts anfangen, er versteht sie nicht. Zudem wird der Quellcode bei größeren Projekten schnell sehr umfangreich, so dass der sehr limitierte Speicherplatz (rund 32 Kilobyte, also etwa 8 Seiten Text) rasch verbraucht wäre. Die einzige Sprache, die unser Arduino versteht, ist der sogenannte Maschinencode. Dieser besteht aus für uns Anwender kryptisch anmutenden Zeichen, die im Steuerwerk des Prozessors bestimmte Aktionen auslösen, zum Beispiel das Abrufen eines Speicherwertes oder die Addition zweier Zahlen. Er wird Schritt für Schritt ausgeführt, kann aber auch Sprungbefehle enthalten.

Stellen Sie sich vor, Sie haben ein Backrezept (unseren Sketch) und sollen es so umformulieren, dass es auch ein Vorschulkind (unser Mikroprozessor) verstehen kann. Statt „Gib drei Eier hinzu“ werden Sie womöglich schreiben „Nimm ein Ei. Schlage vorsichtig gegen die Schale. Öffne nun die Schale. ...“ – mit dem Maschinencode verhält es sich ähnlich. Die Befehle (man spricht in ihrer Gesamtheit auch vom Befehlssatz) sind sehr elementar und simpel. Dafür kann der Chip sie wahnsinnig schnell ausführen – unser Arduino arbeitet mit einer Taktfrequenz von 16 Megahertz, er führt also 16 Millionen Befehlsschritte pro Sekunde aus. Der genaue Befehlssatz sowie die Größe des Programmspeichers und einige weitere Faktoren unterscheiden sich je nach Typ des Mikrocontrollers. Deshalb hatten wir der Arduino IDE bereits im Kapitel 2.6 über den Befehl *Werkzeuge -> Board* mitgeteilt, welchen Boardtyp wir benutzen, in unserem Standardfall also „Arduino/Genuino Uno“.

Die Übersetzung unseres Sketches in den Maschinencode übernimmt der *Compiler*, eine Software, die bereits in die Arduino IDE integriert ist und für uns im Hintergrund arbeitet. Der Compiler wird unterstützt

vom ebenfalls „unter der Haube“ der Arduino IDE installierten *Präprozessor*. Diese Software verarbeitet den Programmcode, bevor er an den Compiler weitergegeben wird. Dabei versucht sie, den Code zu vereinfachen – sie löscht Überflüssiges und Irrelevantes und prüft den Code auf grundlegende Vollständigkeit und Plausibilität. Wir können den Präprozessor auch anweisen, bereits vorhandene Bestandteile aus früheren Sketchen für unser neues Projekt zu übernehmen. Diese Funktion macht man sich auch bei sogenannten *Bibliotheken* zu Nutze – sie bestehen aus Programmcode in einer separaten Datei, welcher vom Präprozessor zum Sketch hinzugefügt wird.

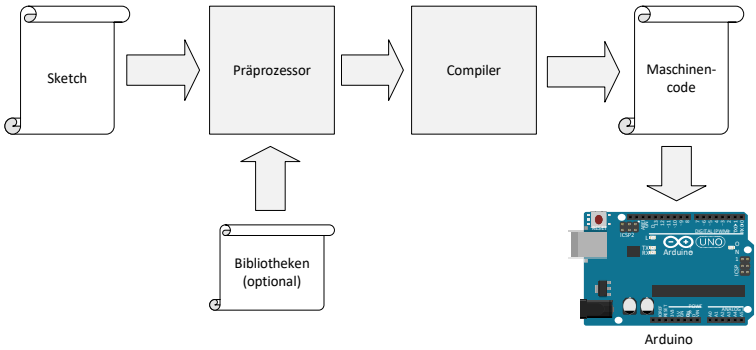


Abb. 4.2 Weg des Programmcodes beim Hochladen eines Sketches auf den Arduino

Der Klick auf den Befehl *Hochladen* löst nun zunächst den Präprozessor aus. Der von ihm verarbeitete Programmcode wird anschließend an den Compiler weitergegeben, welcher nun noch weitere Optimierungen durchführt. So versucht er beispielsweise, mathematische Ausdrücke zu vereinfachen und plant die Aufteilung des Arbeitsspeichers. Abschließend übersetzt der Compiler unser Programm in den Maschinencode, um ihn danach über das USB-Kabel in den Programmspeicher unseres Arduino zu schreiben, wobei stets das vorher auf dem Chip befindliche Programm überschrieben wird. Sie können also nicht einfach wie bei einem USB-Stick mehrere Dateien nebeneinander ablegen – es gibt zu jeder Zeit immer nur ein lauffähiges Programm im Speicher, welches aber natürlich mehrere Bestandteile haben kann.

Sollten Präprozessor oder Compiler Widersprüche oder andere Probleme im Sketch finden, werden Sie einen entsprechenden Hinweis im unteren schwarzen Feld der Arduino IDE finden. Je nach Schwere des Problems wird dadurch eventuell der Übersetzungsvorgang abgebrochen und das Hochladen wird nicht durchgeführt.

Werfen wir nun einen Blick auf den Programmcode und untersuchen ihn Zeile für Zeile.

```
1  // the setup function runs once when you press reset or power the
2  // board                                                    // (1)
3  void setup() {                                            // (2)
4    // initialize digital pin LED_BUILTIN as an output.
5    pinMode(LED_BUILTIN, OUTPUT);                           // (3)
6  }                                                         // (4)
7
8  // the loop function runs over and over again forever
9  void loop() {                                             // (5)
10   digitalWrite(LED_BUILTIN, HIGH);                        // (6)
11   delay(1000);                                            // (7)
12   digitalWrite(LED_BUILTIN, LOW);                         // (8)
13   delay(1000);                                            // (9)
14 }                                                        // (10)
```

Ein Hinweis zu den Zahlen am rechten Rand, denn diese werden Sie in Ihrem Sketch nicht vorfinden: In der Arduino IDE wird alles, was rechts von einem doppelten Schrägstrich steht, vom Programm ignoriert. Somit kann man die Kennzeichnung `//` nutzen, um dahinter einen Kommentar einzufügen – so wie einen Post-It-Klebezettel in einem Buch. Für alle nachfolgenden Codebeispiele nutzen wir diese Kommentarnotation, um Zeilen zu markieren, welche im Folgetext erklärt werden – wie nun auch hier:

1. In dieser Zeile steht nur ein englischer Kommentar, der von den Programmierern des Blink-Beispiels eingefügt wurde. Eventuell stehen bei Ihnen oberhalb dieser Zeile sogar noch weitere Kommentare. Mehrzeilige Kommentare werden durch die Zeichenkombination `/*` eingeleitet und mit `*/` beendet. Die Arduino IDE färbt Kommentare automatisch grau. Sie werden beim Hochladen bereits vom Präprozessor komplett entfernt und gelangen gar nicht zum Compiler. Kommentare dienen nur zu unserer eigenen Übersicht

im Sketch, sie werden vom Programm ignoriert und haben keine Funktion.

2. Hier beginnt nun der bereits erwähnte `setup()`-Programmteil, der direkt nach dem Einschalten einmalig durchlaufen wird. Das Schlüsselwort `void` und die runden Klammern beleuchten wir später, wenn wir auf Funktionen zu sprechen kommen. Wichtig ist an der jetzigen Stelle nur, dass der nachfolgende Funktionsblock (also der Inhalt der besagten Setup-Routine) durch geschweifte Klammern eingeschlossen wird. Diese Klammern werden immer genutzt, wenn mehrere Befehle gruppiert werden.

3. Nun fällt der Blick auf unseren ersten richtigen Befehl. Es handelt sich um die Funktion `pinMode()`. Wie nahezu jede Funktion löst sie eine gewisse Handlung im Mikrocontroller aus. In diesem Fall wird einem Ausgang (also einem Anschlussdraht) eine bestimmte Betriebsart zugewiesen. Natürlich sollte die Funktion dafür wissen, welcher Pin genau gemeint ist, und welche Betriebsart er nun erhalten soll. Um solche Informationen zu übergeben, schreibt man diese sogenannten Argumente in runde Klammern direkt hinter den Funktionsnamen, und zwar in einer durch den Befehl festgelegten Reihenfolge. Beim Befehl `pinMode()` lautet diese Reihenfolge „(Pin-Nummer, Betriebsart)“. An beiden Stellen erwartet die Funktion Zahlenwerte. Nun stehen in unserem Beispiel jedoch gar keine Zahlen. Grund ist, dass die Autoren dieses mitgelieferten Beispiels nicht wissen können, auf welchem Board der Blink-Sketch zum Einsatz kommen wird. Bei unserem Arduino UNO ist die LED mit Pin 13 verbunden, auf anderen Boards kann das jedoch ganz anders sein. Dafür nutzt man einen Trick: Der Begriff `LED_BUILTIN` ist eine sogenannte Präprozessor-Konstante, also eine Art Schlüsselwort. Während der Präprozessor das Programm optimiert, erkennt er dies und ersetzt es in Abhängigkeit vom voreingestellten Board-Typ mit der Nummer des Pins, an den eine Onboard-LED angeschlossen ist. Bei unserem Arduino UNO wird es also die 13 sein. Sie können dies einfach testen, indem Sie den Begriff `LED_BUILTIN` selbst durch die Zahl 13 ersetzen und den Sketch erneut hochladen. Das Programm sollte nach wie vor funktionieren. Der Begriff `OUTPUT` bedeutet, dass wir diesen Pin als Ausgangspin

nutzen wollen. Dies ist ebenso ein Schlüsselwort und wird vom Präprozessor durch eine festgelegte Zahl ersetzt, welche gegenüber der Funktion `pinMode()` den Betriebsmodus „Ausgang" symbolisiert. In Sketchen wird häufig mit solchen Schlüsselworten (Konstanten) gearbeitet, da so das Programm viel einfacher nachzuvollziehen ist.

4. Die schließende geschweifte Klammer schließt den `setup()`-Funktionsblock ab.

5. Hier beginnt die `loop()`-Funktion, also die ständig wiederholte Programmschleife.

6. Die Funktion `digitalWrite(Pin-Nummer, Wert)` leitet den Wert an einen bestimmen Ausgangspin. Da „digital" in diesem Bezug eigentlich „binär", also nur zwei mögliche Zustände meint, kann der Wert nur 1 (eingeschaltet) oder 0 (ausgeschaltet) sein. Alternativ können Sie für 1 auch `HIGH` oder `true` schreiben, und 0 können Sie auch durch `LOW` oder `false` ersetzen. Es verhält sich wie oben, der Präprozessor wird diese Schlüsselworte automatisch durch den hinterlegten Wert ersetzen. An dieser Stelle in unserem Sketch könnten Sie also auch schreiben „`digitalWrite(13, 1);`". Der Befehl führt dazu, dass der Ausgangspin 13 eingeschaltet wird, es werden also dort 5 Volt ausgegeben. Die (über einen Vorwiderstand angeschlossene) LED fängt an zu leuchten.

7. Die Funktion `delay(Millisekunden)` tut genau das, was ihr Name verspricht – sie verzögert den Programmablauf um die als Argument übergebene Zeitspanne in Millisekunden. Unser Arduino legt an dieser Stelle also eine Pause von genau einer Sekunde ein.

8. Hier nun wieder die uns bereits bekannte Funktion `digitalWrite()`[2]. Diesmal wird als zweites Argument `LOW` übergeben, also 0. Der Ausgangspin wird abgeschaltet (auf Masse), die LED erlischt.

---

2 Wenn man über eine Funktion spricht, deren Argumente bereits erläutert wurden, lässt man diese üblicherweise weg und schreibt nur noch leere Klammern als Hinweis darauf, dass es sich um eine Funktion handelt.

9. Erneut wird der Programmablauf um eine Sekunde verzögert.
10. Die sich schließende geschweifte Klammer beendet den Funktionsblock der Hauptschleife `loop()`. Die Schleife beginnt von vorn, nun geht es wieder mit dem Einschalten der LED weiter (6).

Zusammengefasst funktioniert unser Blink-Programm also wie folgt:

Der Mikrocontroller wird eingeschaltet (erhält Betriebsspannung) und konfiguriert daraufhin einmalig den Pin 13 als Ausgang. Danach wird der Ablauf [LED an] – [eine Sekunde warten] – [LED aus] – [eine Sekunde warten] einfach endlos wiederholt.

Bitte blicken Sie nun noch einmal auf die Struktur des Programmes als Ganzes. Wir wollen uns noch kurz der „Rechtschreibung" und der Form widmen. Wichtig ist, dass jeder Befehl korrekt geschrieben (Groß-/Kleinschreibung beachten) und mit einem Semikolon abgeschlossen wird. Sicher sind Ihnen auch die Einrückungen aufgefallen. Üblicherweise rückt man Funktionsblöcke (hier beispielsweise den kompletten Loop-Bereich) etwas nach rechts ein, um ihre Zusammengehörigkeit zu erkennen. Dies ist jedoch beliebig, da der Compiler Leerzeichen, Tabulatoren und Zeilenumbrüche ignoriert. Sie könnten also unser obiges Blink-Beispiel auch so schreiben:

```
1 void setup() {pinMode(LED_BUILTIN, OUTPUT); }
2 void loop() {digitalWrite(LED_BUILTIN, HIGH); delay(1000);
3 digitalWrite(LED_BUILTIN, LOW); delay(1000); }
```

Rein funktional ist in diesem Sketch immer noch alles Nötige enthalten und korrekt notiert, Sie können ihn also hochladen und die LED wird wie erwartet blinken. Jedoch wird sofort klar, dass dadurch jegliche Übersicht verloren geht – allein deshalb empfiehlt es sich, eine gewisse Ordnung im Programmcode zu halten.

Nutzen Sie nun die Gelegenheit und experimentieren Sie ein wenig mit diesem einfachen Beispiel. Wie verändert sich das Blinken, wenn Sie in einer der beiden `delay()`-Funktionen den Wert auf 100 absenken?

4.3 Konstanten

Als nächstes möchten wir nun selbst Schlüsselworte für den Präprozessor festlegen. Diese werden als Konstanten bezeichnet, da sie ihren Wert im Programm nicht mehr ändern können. So könnten wir unser Blink-Beispiel auch so erweitern:

```
1  #define WARTEZEIT 1000                                    // (1)
2
3  void setup() {
4    pinMode(LED_BUILTIN, OUTPUT);
5  }
6
7  void loop() {
8    digitalWrite(LED_BUILTIN, HIGH);
9    delay(WARTEZEIT);                                       // (2)
10   digitalWrite(LED_BUILTIN, LOW);
11   delay(WARTEZEIT);
12 }
```

1. Die Raute am Beginn der Zeile markiert eine Anweisung, die wir direkt an den Präprozessor richten, denn mit ihm müssen wir unsere Konstante vereinbaren.

   Wir weisen den Präprozessor an, das (von uns selbst gewählte) Schlüsselwort WARTEZEIT im Sketch zu suchen und es an allen Fundstellen durch den Wert 1000 zu ersetzen. So etwas ist immer dann sinnvoll, wenn der gleiche Wert oft vorkommt, und Sie ihn später eventuell einmal anpassen müssen – denn es ist wesentlich einfacher, ihn einmal am Programmbeginn zu ändern, statt den kompletten Sketch danach absuchen zu müssen. Dies ist ebenfalls sinnvoll, wenn Sie einen kryptisch anmutenden Zahlenwert mit seiner eigentlichen Bedeutung „beschriften“ möchten, so wie es zum Beispiel bei der Funktion `pinMode()` bereits mit dem Schlüsselwort `OUTPUT` getan wurde – das ist ebenfalls eine Konstante, für die eine gewisse Zahl hinterlegt wurde, welche von der Funktion als entsprechende Betriebsart (“Ausgang“) verstanden wird. Der Compiler bekommt später davon überhaupt nichts mit. Er erhält nur den Quellcode, in dem diese Ersetzung bereits durchgeführt wurde. Deshalb darf am Ende der `define`-Anweisung auch kein Semikolon stehen. Es würde nach der Bearbeitung durch den

Präprozessor alleinig stehenbleiben und im Compiler Fehlermeldungen erzeugen – probieren Sie es gern aus.

2. Nun können wir uns an jeder beliebigen Stelle des Programms auf die vereinbarte Konstante beziehen. Der Name der Konstante muss übrigens nicht unbedingt komplett aus Großbuchstaben bestehen, jedoch hat sich dies eingebürgert, um diese Art von Schlüsselwörtern schnell zu erkennen. In jedem Fall müssen Sie bei der Definition und beim späteren Aufruf der Konstante die exakt gleiche Schreibweise wählen.

4.4 Bedingungen

Wir wollen unser Blink-Beispiel weiter nutzen, um noch mehr Befehle und Kontrollstrukturen kennenzulernen. Dazu ändern wir den Sketch wie folgt ab:

```
1  void setup() {
2    pinMode(LED_BUILTIN, OUTPUT);
3  }
4
5  void loop() {
6    digitalWrite(LED_BUILTIN, HIGH);                    // (1)
7    if(millis() < 10000)                                // (2)
8      delay(50);                                        // (3)
9    else                                                // (4)
10     delay(1000);                                      // (5)
11   digitalWrite(LED_BUILTIN, LOW);
12   delay(1000);
13 }
```

Wenn Sie diesen Sketch nun hochladen, werden Sie ein verändertes Verhalten bemerken: Die LED blitzt im Sekundentakt nur kurz auf, allerdings geht sie nach 10 Sekunden zum normalen Blinken (wie in den vorherigen Beispielen) über. Wenn Sie den Reset-Knopf auf der Platine betätigen, beginnt dieses Verhalten von vorn. Ursache dafür ist eine `if`-Verzweigung ab der mit (1) markierten Zeile. Dabei handelt es sich – trotz des ähnlichen Aussehens – im Gegensatz zu den bereits kennengelernten Befehlen (wie zum Beispiel `delay()` oder `pinMode()`) nicht um eine Funktion, sondern um eine Kontrollstruktur. Das sind

Anweisungen, die den Programmfluss steuern – wie Weichen auf einer Bahnstrecke.

1. Das Einschalten der LED erfolgt wie beim vorherigen Sketch, bis hier hat sich nichts geändert.

2. Bei einer `if`-Anweisung wird eine bestimmte Bedingung geprüft. Diese steht in runden Klammern hinter dem Wort `if`. Es handelt sich dabei immer um einen Ausdruck, der „auf Wahrheit geprüft" werden kann. Dies kann beispielsweise, wie in unserem Fall, ein mathematischer Vergleich sein. Hier lernen wir auch gleich noch die Funktion `millis()` kennen. Die Klammern bleiben leer, da diese Funktion keine Argumente benötigt. Sie ist aber unsere erste Funktion mit Rückgabewert. Das heißt, dass sie später, wenn das Programm auf dem Arduino aufgespielt wird, nach ihrem Aufruf einen Wert als „Antwort" liefert, der an ihrer Stelle eingesetzt werden kann. Die Funktion `millis()` liefert als Rückgabewert die Zahl der Millisekunden, die seit dem Programmstart bereits vergangen sind. Diese Zahl wird nun mit 10.000 verglichen. Ist sie kleiner, sind also weniger als 10 Sekunden vergangen, ist die Aussage wahr. Bitte beachten Sie, dass an dieser Stelle aufgrund des besonderen Charakters der Kontrollstruktur kein Semikolon nach der runden Klammer folgen darf!

3. Im Falle einer wahren Bedingung wird direkt mit dem auf die `if`-Bedingung folgenden Befehl fortgefahren. In unserem Fall also eine nur sehr kurze Pause von 50 Millisekunden – die LED blitzt nur kurz auf. Sind für diesen Zweig mehrere Befehle vorgesehen, müssen diese in geschweiften Klammern zusammengefasst werden (so wie unsere `loop()`-Schleife als Ganzes auch geklammert wird). In unserem Fall ist es nur ein Befehl, deshalb können die Klammern entfallen.

4. Die optionale Anweisung `else` gehört auch noch zur `if`-Bedingung – an ihrer Übersetzung „andernfalls" kann man bereits erkennen, dass die folgende Anweisung (oder der folgende Anweisungsblock) nur ausgeführt wird, wenn die bei (2) angegebene Bedingung falsch

ist. In unserem Fall also genau dann, wenn seit dem Programmstart bereits 10 Sekunden oder mehr vergangen sind.

5. In diesem Fall wird dann genau eine Sekunde pausiert, die LED bleibt also länger hell.

   Auch in diesem Zweig könnten mehrere Befehle stehen, die dann durch geschweifte Klammern eingeschlossen werden müssten, denn sonst endet der `else`-Zweig direkt nach dem nächsten Semikolon, hier also nach diesem `delay()`-Befehl.

Der `else`-Zweig kann auch komplett entfallen, wenn er nicht benötigt wird. In diesem Fall wird bei einer unwahren Bedingung einfach nur der direkt auf die `if`-Bedingung folgende Befehl oder Befehlsblock (geschweifte Klammern) übersprungen.

4.5 Vergleichsoperatoren

Im Zusammenhang mit der gerade vorgestellten `if` -Bedingung werfen wir nun einen Blick auf die grundlegenden Operatoren, welche die Software für derartige Vergleiche zur Verfügung stellt.

Operator	Funktion	Beispiel	Resultat
<	„kleiner als“ – zahlenmäßiger Größenvergleich	`3 < 5`	wahr
>	„größer als“ – zahlenmäßiger Größenvergleich	`3 > 5`	falsch
<=	„kleiner gleich“ – zahlenmäßiger Größenvergleich	`4 <= 4`	wahr
>=	„größer gleich“ – zahlenmäßiger Größenvergleich	`4 >= 4`	wahr
==	Testet auf Gleichheit	`1 == 6`	falsch
!=	Testet auf Ungleichheit	`1 != 6`	wahr

Tabelle 4.1 häufig verwendete Vergleichsoperatoren

Ein häufig vorkommender Fehler ist es übrigens, bei einem gewünschten Vergleichstest versehentlich nur „=“ statt „==“ zu schreiben. Dadurch ergibt sich jedoch eine andere Bedeutung. Wir werden sie noch kennenlernen, wenn wir uns Wertzuweisung von Variablen anschauen. Meist erzeugt dies jedoch keine Fehlermeldung im Compiler, weshalb solche Fehler später oft schwer zu lokalisieren sind.

Des Weiteren ist es möglich, mehrere Teilausdrücke mit *und* beziehungsweise *oder* zu verknüpfen oder zu negieren:

Operator	Funktion	Beispiel	Resultat
&&	und-Verknüpfung (beide Teilausdrücke müssen wahr sein, um als Resultat wahr zu erhalten)	`(3<5) && (8>9)`	falsch
||	oder-Verknüpfung (mindestens einer der Teilausdrücke muss wahr sein, um als Resultat wahr zu erhalten)	`(3<5) || (8>9)`	wahr
!	Negiert den darauffolgenden Ausdruck	`!(2<=5)`	falsch

Tabelle 4.2 häufig verwendete Verknüpfungsoperatoren

Die Ausdrücke können beliebig weiter verschachtelt werden. Ähnlich wie in der Mathematik, wo der Merksatz „Punktrechnung vor Strichrechnung“ gilt, gibt es auch unter den Operatoren eine Reihenfolge, nach der sie aufgelöst werden. Dabei gilt als Faustregel: Zuerst wird negiert, dann verglichen und dann verknüpft.

In den obigen Beispielen zum Und- und Oder-Operator könnte man daher die Klammern auch weglassen, bei der Negation jedoch nicht. Der Ausdruck „`!2<=5`“ wäre in dem Falle wahr, denn die „2“ wird (wie jede von 0 verschiedene Zahl) als „wahr“ gewertet. „!2“ ergibt dementsprechend falsch. Falsch ist wiederum gleichwertig mit 0 – und damit kleiner also 5, somit ist der Gesamtausdruck also wahr.

Eine vollständige Auflistung der Operatoren mit jeweiliger Priorität finden Sie im Anhang dieses Buches.

4.6 Variablentypen

4.6.1 Zahlensysteme

Bevor wir unser Beispiel weiterentwickeln, sollten wir nun noch einen Ausflug in die Mathematik unternehmen, um den Umgang unseres Mikrocontrollers mit Zahlen zu verstehen. Dazu wollen wir zunächst unser vertrautes Dezimalsystem etwas näher beleuchten.

In der Grundschule haben Sie die einzelnen Stellenwerte unseres Zahlensystems vermutlich einmal als „Einer", „Zehner", „Hunderter" und so weiter kennengelernt. Mathematisch gesehen handelt es sich um Zehnerpotenzen, denn die „Hunderter" kann man als Faktor $10^2 = 100$ ausdrücken, die „Zehner" sind logischerweise als Faktor $10^1 = 10$, und selbst die Einer passen in dieses System, denn 10^0 ergibt per Definition[3] 1.

Sollte man eine Regel aufstellen, nach der wir aus den einzelnen Ziffern eine konkrete Zahl bilden, dann könnte sie also lauten:

"Multipliziere die rechte Ziffer mit 10^0, gehe dann einen Schritt nach links und multipliziere diese Ziffer mit 10^1, gehe dann wieder einen Schritt nach links und multipliziere diese Ziffer mit 10^2 – und so weiter, bis keine Ziffern mehr übrig sind. Addiere am Ende alle Ergebnisse."

Die Zahl 242 lautet in dieser Zerlegung also

$$242 = 2 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$$

Im Alltag macht man sich darüber natürlich keine Gedanken, es ist für uns schlicht selbstverständlich. Nur müssen wir uns darüber im Klaren sein, dass die 10 als Basis für das Dezimalsystem[4] einst völlig willkürlich gewählt wurde, sehr wahrscheinlich aufgrund der Anzahl unserer Finger. Das bedeutet im Umkehrschluss auch, dass man theoretisch jede

3 Jede beliebige Zahl ergibt 1, wenn sie mit 0 potenziert wird.

4 Aus dem Lateinischen: *decem* - zehn

beliebige Zahl als Basis für ein Zahlensystem nehmen kann, und es lassen sich immer noch alle beliebigen Werte darstellen.

Testen wir es mit der Basis 16:

$$242 = 15 \cdot 16^1 + 2 \cdot 16^0$$

Es funktioniert: Die Zahl im Beispiel lässt sich ebenso zerlegen – nur leider haben wir nicht genug Ziffern, um die vordere „15" griffig in einer Zahl darstellen zu können. Da dieses sogenannte Hexadezimalsystem[5] aber tatsächlich in der Informatik relativ häufig genutzt wird, hat man sich darauf geeinigt, für die fehlenden Ziffernwerte einfach die ersten Buchstaben des Alphabets zu nutzen.

Ziffernwert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexadezimalwert	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Falls Sie sich wundern, warum die Ziffern im wörtlichen übersetzten „Sechzehnersystem" nur bis „15" reichen: Die 0 muss natürlich mitgezählt werden. In unserem Zehnersystem ist die größte Ziffer schließlich auch die 9, nicht die 10. Somit ergibt sich für unser Beispiel die hexadezimale Schreibweise „F2", es genügen nun also sogar zwei Ziffern:

$$242 = 15 \cdot 16^1 + 2 \cdot 16^0 = F2$$

Wir ändern die Basis unseres Zahlensystems ein weiteres Mal: Nun soll die 2 die Grundlage unseres Zählens sein. Wir benötigen jetzt deutlich mehr Ziffern, in diesem Binärsystem[6] nennt man diese auch „Bit":

5 von griechisch *hexa* - sechs und lateinisch *decem* - zehn

6 Von lateinisch *bini* – „je zwei"; oft auch *Dualsystem* genannt

$$242 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 11110010$$

Wie Sie sehen, ist nun nicht mehr eindeutig erkennbar, ob die dargestellte 11110001 nun als Binärzahl oder als Dezimalzahl verstanden werden soll. Im Programmcode müssen wir dieses somit besonders kenntlich machen, so dass der Compiler erkennt, wie er diese Zahl interpretieren soll. Im Falle von Binärzahlen stellt man ein „B“ als Präfix links voran. Weitere Notationen sind in der folgenden Tabelle aufgeführt:

Zahlenbasis	Präfix	Beispiel
10 (dezimal)	*nicht notwendig*	242
2 (binär)	B	B11110010
8 (oktal)	0	0362
16 (hexadezimal)	0x	0xF2

Sie könnten also im Programm schreiben

```
...
delay(242);
delay(B11110010);
delay(0362);
delay(0xF2);
...
```

Das Ergebnis jeder dieser vier `delay()`-Funktionen ist jeweils exakt das gleiche: Das Programm wird immer für 242 Millisekunden pausiert.

Ein häufiger Fehler in diesem Zusammenhang sind Werte, die im Quellcode versehentlich mit führender Null angegeben wurden und vom Compiler dadurch als Oktalzahl missverstanden werden. Dies kann zum Beispiel bei Telefonnummern passieren, da die Vorwahl stets mit 0 beginnt. Abhilfe schafft die Speicherung als Zeichenkette, dazu später mehr.

Da alle Zahlensysteme in der Mathematik absolut gleichberechtigt sind, lassen sich auch alle Rechenarten zu jeder beliebigen Zahlenba-

sis anwenden, und zwar nach den gleichen Methoden, wie man sie im Dezimalsystem auch benutzt. So kann man beispielsweise auch binär mühelos eine Addition durchführen, wenn man sich an das schriftliche Rechnen aus der Schule erinnert:

```
  11110010          242
+        11        +  3
----------        -----
= 11110101        = 245
```

Beim Blick auf die Geschichte der Mikrocontroller wurde bereits erwähnt, dass sich das Binärsystem als Rechengrundlage für Mikroprozessoren durchgesetzt hat, weil es sich in elektrischen Schaltkreisen besonders einfach und robust nachbilden lässt. Da es nur zwei mögliche Ziffern gibt, repräsentiert man die „1" einfach durch eine anliegende Spannung, während „0" durch den spannungslosen Zustand dargestellt wird.

Möchten wir nun in unserem Arduino-Programm mit veränderlichen Zahlen (sogenannten Variablen) rechnen, so ist es erforderlich, dass wir dem Compiler mitteilen, wieviel Arbeitsspeicher er für eine bestimmte Variable vorsehen muss. Beim Beispiel mit unserem Eintopf würden Sie sich vor dem Kochen entsprechend auch Gedanken machen, wie groß der Topf sein muss. Natürlich könnten Sie auch einfach stets den größten Topf nehmen und so auf Nummer sicher gehen – allerdings büßen Sie dadurch viel Platz auf dem Herd ein, und das Kochen dauert auch etwas länger.

Beim Programmieren verhält es sich kaum anders. Wir können Zahlen als Variablen speichern. Ganz egal, ob wir sie nun als Zwischenergebnis, als Zähler oder als Faktor in einer Formel nutzen wollen – wir müssen beim Erstellen des Programmcodes entscheiden, wieviel Speicherplatz wir für sie reservieren – also wie groß der Topf sein soll. Bei großen Projekten, in denen der Speicherplatz schnell knapp wird, will dies besonders gut bedacht sein.

Die folgende Tabelle zeigt die wichtigsten Variablentypen und den zugehörigen Speicherbedarf.

Typ	Bedeutung	Speicherbedarf	Darstellbare Werte
`byte`	Ganzzahl ohne Vorzeichen	8 Bit = 1 Byte	0 ... 255
`int`	Ganzzahl (engl. Integer) mit Vorzeichen	16 Bit = 2 Byte	-32.768 ... 32.767
`unsigned int`	Ganzzahl ohne Vorzeichen	16 Bit = 2 Byte	0 ... 65.535
`long`	Ganzzahl mit Vorzeichen	32 Bit = 4 Byte	-2,147,483,648 ... 2,147,483,647
`unsigned long`	Ganzzahl ohne Vorzeichen	32 Bit = 4 Byte	0 ... 4.294.967.295
`char`	Ganzzahl; repräsentiert in der Regel einen Buchstaben	8 Bit = 1 Byte	-128 ... 127
`float`	Fließkommazahl	32 Bit = 4 Byte	-3.4028235E+38 ... 3.4028235E+38
`boolean`	Wahr/Falsch-Aussage	1 Bit	1 (true) oder 0 (false)

Tabelle 4.3 häufig genutzte Variablentypen

Im obigen Beispiel war gut zu erkennen, dass die im Dezimalsystem dreistellige 242 im Binärsystem ganze 8 Ziffern, also 8 Bit, benötigt. Sie könnte also gerade noch durch eine Variable vom Typ `byte` gespeichert werden. Des Weiteren sehen Sie, dass bei manchen Variablentypen ein Bit reserviert wird, um ein eventuelles Minuszeichen vor der Zahl darzustellen. Entsprechend verschiebt sich ihr möglicher Wertebereich gegenüber gleich großen Variablentypen ohne Vorzeichen.

Doch welche Bedeutung haben diese Wertebereiche für uns? Nehmen wir nochmal unser Additionsbeispiel von oben, diesmal zählen wir allerdings eine etwas größeren Zahl hinzu:

```
  11110010          242
+   10100        +  20
----------       -----
= 100000111      = 262
```

In der binären Darstellung benötigt das Ergebnis nun 9 Bit. Haben wir dafür jedoch eine Variable vom Typ `byte` verwendet, sind nur 8 Bit Speicherplatz verfügbar. Das führt dazu, dass das höchstwertige (also linke) Bit einfach abgeschnitten wird, man nennt diesen Effekt *Überlauf*. Als Ergebnis der Berechnung erhalten Sie dann nur noch „00000111", was dezimal dem Wert 7 entspricht – also ein unbrauchbares Ergebnis.

Den Wertebereich einer solchen Ganzzahl-Variablen können Sie sich wie ein Ziffernblatt vorstellen, auf dem die möglichen Werte im Kreis angeordnet sind. Ein Zeiger zeigt den aktuellen Wert. Durch mathematische Berechnungen bewegt er sich weiter. Überschreitet dieser jedoch seine Wertebereichsgrenze (im Falle einer `byte` -Variablen also die Zahl 255), so beginnt er einfach wieder beim kleinsten Wert. Dies gilt natürlich in beide Richtungen: Die Subtraktion „3 – 5" würde mit `byte` -Variablen das Ergebnis 254 liefern.

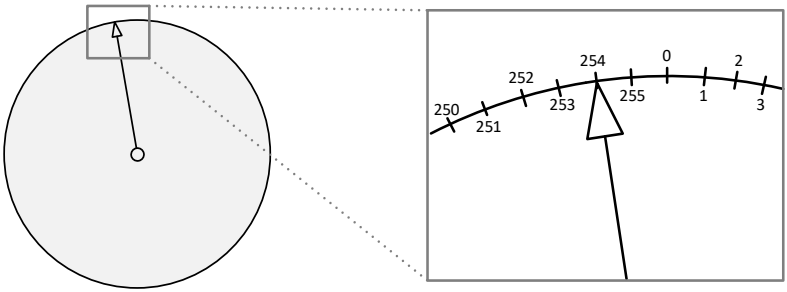


Abb. 4.3 schematische Darstellung einer Byte-Variablen als Ziffernblatt

Es sei noch bemerkt, dass Sie im Falle eines solchen Überlaufs keine Fehlermeldung sehen werden. Als Mikrocontroller-Programmierer erhalten Sie sehr hardwarenahen Zugriff. Sie bestimmen also sehr direkt, was in dem kleinen Rechner passiert, ohne dass im Hintergrund noch ein Betriebssystem oder eine zwischengeschaltete Software läuft. Diese Freiheit bietet viele Vorteile und Möglichkeiten, nimmt Sie im Gegen-

zug aber auch in die Pflicht, sich selbst um gewisse Vorsichtsmaßnahmen zu kümmern – so zum Beispiel auch die Vermeidung des eben beschriebenen Werteüberlaufs.

Obige Tabelle wird dominiert von den Ganzzahlvariablen. Diese werden in der Programmierung auch am häufigsten benötigt. Daneben gibt es noch einige Spezialtypen, die wir kurz kennenlernen wollen:

Variablen vom Typ `char` werden wie eine Byte-Ganzzahl mit 8 Bit (inklusive Vorzeichen) gespeichert, jedoch repräsentieren sie eigentlich einen Buchstaben (genauer gesagt: ein Zeichen). Die Zuordnung ergibt sich aus der standardisierten ASCII[7]-Tabelle, so steht die Zahl 65 zum Beispiel für den Großbuchstaben „A“.

`Float`-Werte sind Fließkommazahlen, welche häufig in wissenschaftlichen Berechnungen vorkommen. Sie werden mit 32 Bit gespeichert, daraus ergibt sich eine Genauigkeit von etwa 6 bis 7 Nachkommastellen. Da die Darstellung als Mantisse und Exponent (Kommazahl und Zehnerpotenz) einen anderen Weg der Berechnung erfordert, benötigen mathematische Formeln mit Fließkommazahlen im Mikrocontroller deutlich mehr Zeit. Außerdem wirkt sich die beschränkte Bitzahl oft als Rundungsfehler aus, der durch ungünstige Rechenwege (Multiplikation oder Division um mehrere Größenordnungen) schnell relevante Beträge erreichen kann. Fließkommazahlen sollten daher, besonders durch Einsteiger, sehr zurückhaltend eingesetzt werden.

Eine Variable von Typ `boolean` kann nur zwei Werte annehmen: 1 oder 0. Gleichwertig kann stattdessen auch `true` oder `false` geschrieben werden. Sie benötigt dadurch nur ein Bit Speicherplatz und kann schnell und effizient verarbeitet werden. Man benutzt boolesche[8] Variablen daher meist als eine Art Schalter, um zum Beispiel die Antwort auf eine Ja/Nein-Frage zu speichern.

7 *American Standard Code for Information Interchange* – standardisierte US-Zeichentabelle

8 benannt nach George Boole, einem englischen Mathematiker

4.6.2 Variablen definieren

Um dem Compiler mitzuteilen, von welcher Art eine bestimmte Variable sein soll, gibt man diesen Typ bei der ersten Nennung dieser Variable im Programmcode an. Zur Verdeutlichung modifizieren wir unser Blink-Beispiel wieder etwas:

```
1  int Wartezeit;                                          // (1)
2
3  void setup() {
4    pinMode(LED_BUILTIN, OUTPUT);
5    Wartezeit = 1;                                        // (2)
6  }
7
8  void loop() {
9    digitalWrite(LED_BUILTIN, HIGH);
10   delay(Wartezeit);                                     // (3)
11   digitalWrite(LED_BUILTIN, LOW);
12   delay(Wartezeit);
13   Wartezeit = Wartezeit + 1;                            // (4)
14 }
```

Wenn Sie diesen Sketch auf Ihren Arduino übertragen, werden Sie beobachten, dass die kleine LED zunächst sehr schnell blinkt und ihre Blinkfrequenz dann stetig verlangsamt. Durch Druck auf den Reset-Knopf können Sie diesen Vorgang neu starten.

1. Hier wird die Variable `Wartezeit` deklariert. Das Schlüsselwort `int` zeigt dem Compiler an, dass er dafür 16 Bit im Arbeitsspeicher des Mikrocontrollers reservieren soll. Diese Nennung direkt am Anfang des Sketches ist nicht unbedingt notwendig – man könnte die Variable auch später noch deklarieren. Oft wird ein Programm jedoch übersichtlicher, wenn zunächst alle wichtigen Variablen „vorgestellt“ und eventuell durch Kommentare erläutert werden. Eine solche Deklaration ist ein Befehl, der ausnahmsweise auch außerhalb der beiden wichtigen Programmteile `setup()` und `loop()` stehen darf. Das liegt daran, dass der Compiler ihn komplett für seine Zwecke verarbeitet (weil er Speicher reservieren muss etc.) und im späteren Maschinencode kein gesonderter Befehl mehr daraus resultiert.

2. In der `setup()`-Routine, also direkt nach dem Einschalten, wird der Wert zunächst auf 1 gesetzt. Das Gleichheitszeichen dürfen Sie demnach nicht als „Istgleich“ im mathematischen Sinne verstehen (für diesen Zweck haben wir bereits den Vergleichsoperator „==“ kennengelernt). Das Zeichen fungiert stattdessen als Zuweisungsoperator, und zwar immer von rechts nach links. Der Variablen mit dem Namen `Wartezeit` wird also der Wert 1 zugewiesen. Konkret bedeutet das im Mikrocontroller: Der binäre 16-Bit-Wert 0000000000000001 wird in den Arbeitsspeicher an die Stelle geschrieben, welche der Compiler für die Variable `Wartezeit` vorgesehen hat.
3. Die bereits bekannte Funktion `delay()` bekommt nun nicht mehr eine festgelegte Zahl übergeben, sondern den Wert, den die Variable `Wartezeit` aktuell hat. Es wird also die entsprechende Zahl aus dem Arbeitsspeicher gelesen und der Funktion als Argument übergeben.
4. Am Ende der Programmschleife wird nun zu der Zahl `Wartezeit` die Zahl 1 addiert, das Ergebnis wird wieder der Zahl `Wartezeit` zugewiesen. Somit steigt die im Arbeitsspeicher für unsere Variable `Wartezeit` hinterlegte Zahl mit jedem Programmdurchlauf um 1 an, sie zählt sozusagen die Schleifen.

Die Funktionsweise dieses Programmes ist nun offensichtlich: Die Wartezeit (in Millisekunden) zwischen den Umschaltungen der LED entspricht der Anzahl der bereits durchlaufenen Programmschleifen, deshalb blinkt die LED immer langsamer.

Welchen Namen Sie einer Variablen geben, ist übrigens Ihnen überlassen. Es gibt lediglich formale Vorgaben, so darf der Name nicht mit einer Ziffer beginnen und auch keine Leerzeichen enthalten. Verständlicherweise dürfen auch keine Schlüsselworte wie „if“ oder „byte“ verwendet werden. Im Wesentlichen dient der Name nur der Wiedererkennung durch den Programmierer, deshalb sind selbsterklärende Bezeichner stets von Vorteil. Nutzen Sie also lieber „Aussentemperatur“ anstatt „Wert“. Für den Compiler ist der Name lediglich ein Hinweis, um zu er-

kennen, an welchen Stellen eine Variable eingesetzt werden soll. Im erzeugten Maschinencode steht dann lediglich die Adresse des Speicherbereiches, der für die Variable reserviert wurde, der Name taucht dort nicht mehr auf. Längere Variablennamen benötigen also nicht mehr Speicher auf dem Mikrocontroller. Sie sollten allerdings darauf achten, dass Sie den Namen auch in Bezug auf Klein- und Großbuchstaben immer exakt gleich schreiben. „`Wartezeit`“ und „`wartezeit`“ sind für den Compiler zwei verschiedene Variablen.

Obiges Codebeispiel wurde zur besseren Nachvollziehbarkeit relativ ausführlich geschrieben. Für geübte Programmierer gibt es auch hier wieder die Möglichkeit, den Sketch etwas zu verkürzen, ohne die Funktion zu verändern:

```
1  int Wartezeit = 1;                                          // (1)
2
3  void setup() {
4    pinMode(LED_BUILTIN, OUTPUT);
5  }
6
7  void loop() {
8    digitalWrite(LED_BUILTIN, HIGH);
9    delay(Wartezeit);                                         // (2)
10   digitalWrite(LED_BUILTIN, LOW);
11   delay(Wartezeit++);                                       // (3)
12 }
```

1. Variablen dürfen bei ihrer Deklaration auch gleich initialisiert (also mit einem bestimmten Anfangswert belegt) werden. Dadurch entfällt in diesem Sketch die spätere Wertzuweisung in der Setup-Routine.

2. An diesem Befehl ändert sich nichts, der Wert von `Wartezeit` wird aus dem Speicher geladen und der `delay()`-Funktion übergeben.

3. Hier kommt nun der sogenannte Inkrementaloperator „++“ zum Einsatz. Wird er an eine Variable angehängt, vollzieht er das Gleiche wie die Zeile (4) im vorherigen Beispiel. Er erhöht also den Wert von `Wartezeit` im Arbeitsspeicher um 1, übergibt ihn aber zusätzlich an die Funktion `delay()`. Dadurch kann die separate Addition entfallen.

Von der Position des Inkrementaloperators ist abhängig, welchen Wert er an die übergeordnete Funktion übergibt. „`Wartezeit++`“ gibt den Wert vor der Erhöhung weiter, „`++Wartezeit`“ übergibt den bereits erhöhten Wert. Die den Arduino-Sketches zugrundeliegende Programmiersprache C++ hat ihren Namen übrigens von genau diesem Operator, welcher einst mit ihr eingeführt wurde. Es gibt auch den Dekrementaloperator `--`, welcher in exakt gleicher Weise eine Verringerung um 1 ausführt, sowie noch weitere Verkürzungen, die an entsprechender Stelle in den Codebeispielen erläutert werden, wenn wir sie zum ersten Mal einsetzen.

Übrigens: Wenn Sie testweise einmal den Variablentyp `int` in Zeile (1) durch `byte` ersetzen, können Sie beobachten, was der bereits erläuterte Wertüberlauf hier bewirkt: Nach 255 Durchläufen (rund 65 Sekunden) beginnt `Wartezeit` wieder von 0, da der Speicher nicht mehr ausreicht. Die LED beginnt also erneut mit dem schnellen Blinken. Unsere stattdessen genutzte `int`-Variable läuft zwar auch irgendwann über, allerdings in diesem Beispiel rechnerisch erst nach 23 Tagen. Nutzte man an dieser Stelle den Variablentyp `long`, würde es sogar 63 Jahre dauern, bis es zum Überlauf kommt.

4.7 Schleifen

Widmen wir uns nun wieder unserem Blink-Beispiel und erweitern es, um noch mehr Kontrollstrukturen kennenzulernen. Laden Sie den folgenden Sketch auf Ihren Arduino:

```
1  void setup() {
2    pinMode(LED_BUILTIN, OUTPUT);
3  }
4
5  void loop() {
6    for(byte Zaehler = 1; Zaehler <= 10; Zaehler++)          // (1)
7    {                                                        // (2)
8      digitalWrite(LED_BUILTIN, HIGH);
9      delay(300);
10     digitalWrite(LED_BUILTIN, LOW);
11     delay(300);
```

```
12    }                                                    // (3)
13    delay(5000);
14  }
```

Nun wird die LED jeweils 10-mal schnell hintereinander blinken und dann für 5 Sekunden dunkel bleiben.

1. Hier benutzen wir das erst mal eine sogenannte `for`-Schleife als Kontrollstruktur. Schleifen sind immer dann günstig, wenn Sie einen Befehl oder eine Gruppe von Befehlen mehrfach hintereinander ausführen möchten. Ihr Aufbau ähnelt der `if`-Anweisung, auch sie wird wesentlich bestimmt durch den Inhalt der runden Klammern. Dieser „Schleifenkopf" besteht bei der `for`-Schleife aus drei Teilen, welche durch Semikolon getrennt werden: Den ersten Teil nennt man auch Initialisierungsteil, er wird nur einmalig durchlaufen, und zwar vor dem ersten Schleifendurchgang. Hier deklarieren Sie eine Variable, die zum Zählen der Schleifendurchläufe genutzt werden soll, und weisen ihr dabei einen Anfangswert zu. Wir nutzen hier den beliebig gewählten Namen `Zaehler`, da sie für uns die Blinkzeichen zählen wird. Für unser Beispiel reicht eine 8-Bit Variable, denn wir möchten vorerst nur bis 10 zählen. Im nach dem Semikolon folgenden zweiten Teil steht die Durchlaufbedingung. Bei jedem Schleifendurchlauf wird sie erneut geprüft. In unserem Fall wird also untersucht, ob der Wert der Variable `Zaehler` kleiner oder gleich 10 ist. Ist diese Bedingung wahr, wird die Schleife weiter durchlaufen. Ist sie falsch, wir die Schleife abgebrochen und mit der normalen Programmabarbeitung fortgefahren. Der letzte Teil innerhalb der runden Klammern ist der Anweisungsteil. Er wird nach jedem Schleifendurchlauf einmal ausgeführt. Man nutzt ihn üblicherweise, um die Zählvariable zu aktualisieren. In den meisten Fällen genügt dafür der Inkrementaloperator ++, den wir ja bereits kennengelernt haben. Es wäre aber ebenso erlaubt, den Zähler stattdessen immer in Dreierschritten hochzuzählen – oder sogar rückwärts zu zählen.

2. Hinter der schließenden runden Klammer der `for`-Anweisung folgt der „Schleifenrumpf". Hier gilt genau wie bei der `if`-Struktur: Besteht dieser Block aus mehreren Anweisungen, müssen diese von

geschweiften Klammern eingeschlossen werden. Fehlen diese Klammern, wird nur der direkt auf die `for`-Anweisung folgender Befehl als Schleifenrumpf gewertet.

3. Diese geschweifte Klammer schließt die Schleife entsprechend. Ist die oben genannte Schleifenbedingung nicht mehr erfüllt, wird das Programm direkt nach dieser Klammer fortgesetzt.

Nun wird auch ersichtlich, warum unsere LED 10-mal schnell blinkt: Die Befehle in der `for`-Schleife werden zehnmal hintereinander durchgeführt, und die per `delay()` eingefügten Pausen betragen nur 300 Millisekunden. Nach den zehn Durchläufen wird die Schleife verlassen und es folgt ein Pausenbefehl, der 5 Sekunden andauert. Danach endet die übergeordnete `loop()`-Programmschleife und beginnt somit von vorn. Wieder wird in der `for`-Schleife der Variable `Zaehler` der Wert 1 zugewiesen und so weiter.

Die `for`-Schleifen können Sie auch wieder mit dem Rezept vergleichen – statt fünfmal hintereinander zu schreiben „Geben Sie ein Ei hinzu", werden Sie einfach vermerken „geben Sie 5 Eier hinzu".

Somit eignet sich diese Schleifenanweisung besonders gut, wenn Sie bereits vor dem ersten Durchlauf genau wissen, wie oft die Schleife durchlaufen werden soll. In manchen Fällen ist die Anzahl der Durchläufe jedoch noch nicht genau bestimmt, sie ergibt sich erst während der Durchführung. Stellen Sie sich vor, Sie greifen mit der Hand in eine undurchsichtige Tüte und holen Gegenstände heraus. Hierbei wissen Sie vorher auch nicht, wie oft Sie hineingreifen müssen – die Abbruchbedingung lautet schlicht „bis nichts mehr drin ist".

Die `for`-Schleife hat daher auch noch zwei „Verwandte", die hier noch kurz erwähnt seien: Eine `while`-Schleife funktioniert sehr ähnlich, jedoch steht in der runden Klammer ausschließlich die Bedingung. Ist sie wahr, wird die Schleife durchlaufen und danach die Bedingung erneut geprüft. Ist sie falsch, wird der Schleifenrumpf übersprungen und die normale Programmbearbeitung fortgesetzt. Jede `for`-Schleife ließe sich somit auch durch eine `while`-Schleife reali-

sieren. Die Programmschleife unseres obigen Beispiels würde dann so aussehen:

```
1  ...
2  void loop() {
3    byte Zaehler = 1;                                    // (1)
4    while(Zaehler++ <= 10)                               // (2)
5    {                                                    // (3)
6      digitalWrite(LED_BUILTIN, HIGH);
7      delay(300);
8      digitalWrite(LED_BUILTIN, LOW);
9      delay(300);
10   }
11   delay(5000);
12 }
```

1. Da die die `while`-Schleife keinen Initialisierungsteil besitzt, müssen wir die Variable separat deklarieren und initialisieren, ihr also einen Anfangswert geben.
2. In den runden Klammern hinter der `while`-Anweisung steht nun nur noch die Bedingung. Der Inkrementaloperator erspart uns in diesem Beispiel einen separaten Befehl zur Erhöhung des Zählers. Dieser müsste ansonsten innerhalb des Schleifenrumpfs erfolgen.
3. Die zur Schleife gehörenden Befehle werden wieder, wie bisher, von geschweiften Klammern eingeschlossen.

Wie Sie sehen, ist bei feststehender Anzahl der Durchläufe die `for`-Schleife etwas eleganter. Die Stärken der `while`-Schleife werden wir in späteren Beispielen jedoch auch noch kennenlernen, zum Beispiel wenn es um Eingabesignale geht.

Ist die Bedingung von Anfang an falsch, wird die Schleife übrigens überhaupt nicht ausgeführt, sondern einfach übersprungen. Soll ein mindestens einmaliger Durchlauf erzwungen werden, kann dafür eine `do-while`-Schleife genutzt werden – hierbei wird die Bedingung erst am Schleifenende geprüft. Diese Schleifenart findet jedoch eher selten Verwendung, deshalb soll es in unserem Fall bei dieser kurzen

Erwähnung bleiben. Das obige Beispiel sähe in solch einem Fall wie folgt aus:

```
...
void loop() {
  byte Zaehler = 1;
  do {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(300);
    digitalWrite(LED_BUILTIN, LOW);
    delay(300);
  } while(Zaehler++ <= 9);
  delay(5000);
}
```


4.8 Ein- und Ausgabe am Bildschirm

Aktuell sind unsere Kommunikationsmöglichkeiten mit dem Arduino noch stark eingeschränkt – als Ausgabe haben wir nur eine simple LED, Eingabemöglichkeiten haben wir bisher noch gar keine. Um die nächsten Codebeispiele besser nachvollziehen zu können, bauen wir nun eine Datenverbindung mit dem Computer auf, um dessen Tastatur und Display nutzen zu können. In den Kapiteln 5 und 6 werden wir unseren Arduino natürlich noch direkt mit Displays und Tasten versehen, an aktueller Stelle soll es uns zunächst nur um das Verständnis von grundlegendem Programmcode gehen.

Wir verändern unser Beispiel abermals:

```
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);                                    // (1)
}

void loop() {
  for(byte Zaehler = 1; Zaehler <= 5; Zaehler++)
  {                                                      // (2)
    digitalWrite(LED_BUILTIN, HIGH);
    delay(300);
    digitalWrite(LED_BUILTIN, LOW);
    delay(300);
    Serial.print(" Zähler: ");                           // (2)
    Serial.print(Zaehler);                               // (3)
```

```
15    }
16    delay(5000);
17    Serial.println("--- Loop-Ende ---");                    // (4)
18  }
```

Klicken Sie nach dem Hochladen auf die lupenähnliche Schaltfläche „serieller Monitor“ oben rechts in der Arduino IDE.

Abb. 4.4 Die Schaltfläche zum Öffnen des seriellen Monitors befindet sich in der oberen rechten Ecke der Arduino IDE

Es öffnet sich ein neues Fenster, in welchem unten rechts der Wert „9600 Baud“ eingestellt werden muss. Ab dann sollten Sie in diesem Fenster fortlaufend die Ausgabe unseres Programmes sehen:

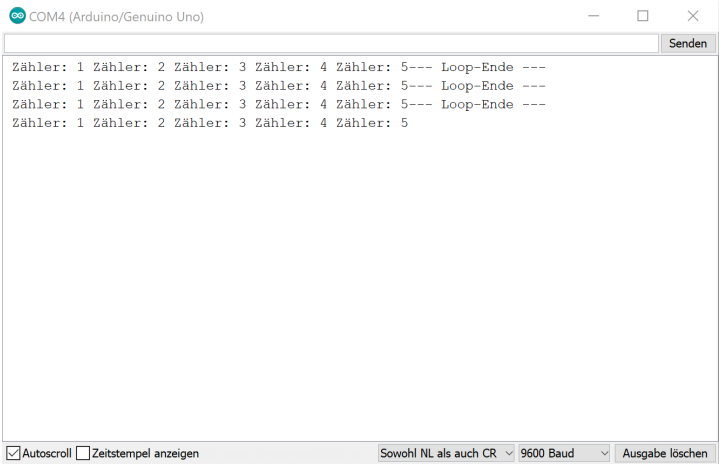


Abb. 4.5 Textausgabe auf dem Computermonitor

Der Arduino hat nun über das USB-Kabel eine Datenverbindung zum PC und kann in diesem Fenster Zeichen ausgeben – so wie eine Schreibmaschine oder ein Text-Ticker. Der Mikrocontroller selbst nutzt dafür seine serielle Schnittstelle – eine sehr einfache und bewährte Möglichkeit, über kurze Distanzen Daten zu übertragen. Dafür sind nur drei Leitungen nötig – *Tx* (Transceive – senden), *Rx* (Receive – empfangen) und die Masse-Verbindung als Gegenpol. Für moderne PCs ist die serielle Schnittstelle zu langsam und nicht mehr zeitgemäß, daher fehlt ihnen ein entsprechender Anschluss.

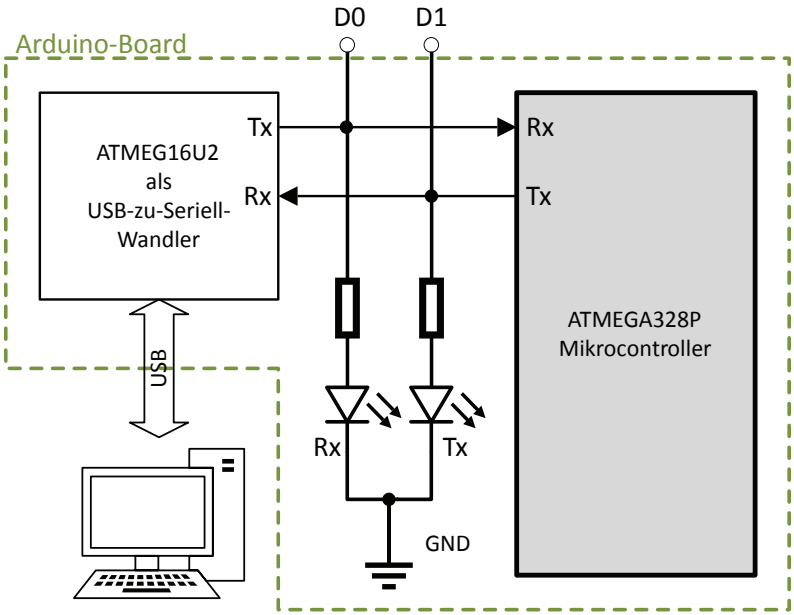


Abb. 4.6 Die seriellen Datenleitungen sind auch mit Pin D0 und D1 verbunden, deshalb können diese nicht anderweitig genutzt werden, wenn eine serielle Verbindung besteht

Die Arduino-Entwickler lösten dieses Problem, indem sie auf der Arduino-Platine einen weiteren Mikrocontroller platzierten, der sich per USB verbinden lässt und sich am Computer als serielle Schnittstelle meldet. Er ist sozusagen der verlängerte Arm des Computers und ermöglicht

so die serielle Kommunikation mit dem ATmega328, also dem eigentlichen Arduino-Mikrocontroller. Aus diesem Grund müssen Sie in der Arduino IDE als Anschluss einen COM-Port auswählen, denn ursprünglich wurden serielle Anschlüsse als *COM(munication)-Ports* bezeichnet.

Mit den seriellen Leitungen sind übrigens auch die mit „Rx" und „Tx" beschrifteten LEDs auf der Platine verbunden. Deshalb flackert die „Tx"-LED immer kurz auf, wenn eine Ausgabe von unserem Programm an den Bildschirm gesendet wird.

Schauen wir uns nun die Befehle des Codebeispiels im Detail an:

1. Mit dem Befehl `Serial.begin()` wird die Verbindung hergestellt. Der Arduino UNO hat dafür fest voreingestellte Anschlüsse, sie können am Pin 0 (Rx) und 1 (Tx) abgegriffen werden. Wir müssen jedoch nichts anklemmen, da sie bereits auf dem Board zusätzlich mit dem besagten USB-Konverter verbunden sind. Die als Argument übergebene Zahl 9600 stellt die gewünschte Datenübertragungsrate dar. Da der Verbindungsaufbau nur einmal nötig ist, steht er innerhalb des `setup()`-Programmteils. Falls Sie sich wundern, warum mitten im Funktionsnamen ein Punkt zu finden ist: Wir werden diese Schreibweise in Kapitel 4.12 (objektorientierte Programmierung) noch genauer beleuchten. An der aktuellen Stelle hat dies für uns noch keinen Belang, sehen Sie den Punkt vorerst einfach als Bestandteil des Funktionsnamens.
2. Die Funktion `Serial.print()` können wir nun einfach nutzen, um Zeichen oder Werte an ein seriell angeschlossenes Gerät (hier also den PC) zu schicken. Als Argument wird hier einfach eine Zeichenkette (also ein Textschnipsel) übergeben.
3. Diesmal wird der Funktion der Wert der Variable `Zaehler` übergeben. Auf der Bildschirmausgabe des Computers erkennen Sie, dass diese Zahl erwartungsgemäß hochgezählt wird.
4. Hier heißt die Ausgabefunktion geringfügig anders: Das angehängte „`ln`" bei `Serial.Println()` steht für *Line*, also Zeile. Es bedeutet lediglich, dass im Anschluss an die Ausgabe noch ein Steuerzei-

chen für einen Zeilenwechsel gesendet wird. Deshalb beginnt in der Ausgabe auf dem seriellen Monitor danach immer eine neue Zeile. Dies dient nur der besseren Übersicht.

Die bei der Initialisierung angegebene Geschwindigkeit (Baudrate) von 9600 beschreibt vereinfacht gesagt die Anzahl der übertragenen Bit pro Sekunde. Da Text meist mit 8 Bit pro Zeichen übertragen wird, können wir also grob geschätzt etwa 1.200 Buchstaben pro Sekunde senden. Real ist dieser Wert etwas geringer, da auch noch Steuer- und Kontrollbits übertragen werden, um den Datenfluss zu regeln. Für unsere Anwendungen reicht diese Geschwindigkeit aber auf jeden Fall. Der Arduino Mikrocontroller unterstützt Geschwindigkeiten bis zu 115200 Baud. Hohe Geschwindigkeiten machen das Signal allerdings anfälliger für Störungen. Mehr dazu im Kapitel 5.5, wenn wir uns den Schnittstellen widmen.

Natürlich funktioniert die Übertragung auch in die andere Richtung. Dazu nehmen wir nun ein neues Beispiel:

```
1  void setup()
2  {
3    pinMode(LED_BUILTIN, OUTPUT);
4    Serial.begin(9600);
5    Serial.println("### LED-Steuerung ###");                  // (1)
6    Serial.println("e: einschalten / a: ausschalten");
7  }
8
9  void loop() {
10   if (Serial.available()) {                                 // (2)
11     char Eingabe = Serial.read();                           // (3)
12     if(Eingabe == 'e')                                      // (4)
13     {
14       digitalWrite(LED_BUILTIN, HIGH);
15       Serial.println("Die LED wurde eingeschaltet.");
16     }
17     if(Eingabe == 'a')
18     {
19       digitalWrite(LED_BUILTIN, LOW);
20       Serial.println("Die LED wurde ausgeschaltet.");
21     }
22   }
23 }
```

Nach dem Hochladen dieses Sketches öffnen Sie einfach den seriellen Monitor, wieder mit der Einstellung „9600 Baud“. Zunächst bleibt die Anzeige leer. Klicken Sie nun mit der Maus in das Eingabefeld am oberen Rand, tippen Sie ein „e“ ein und bestätigen mit Enter. Nun sollte die LED aufleuchten und ein Bestätigungstext am Bildschirm erscheinen. Mit dem Buchstaben „a“ können Sie sie in gleicher Weise abschalten.

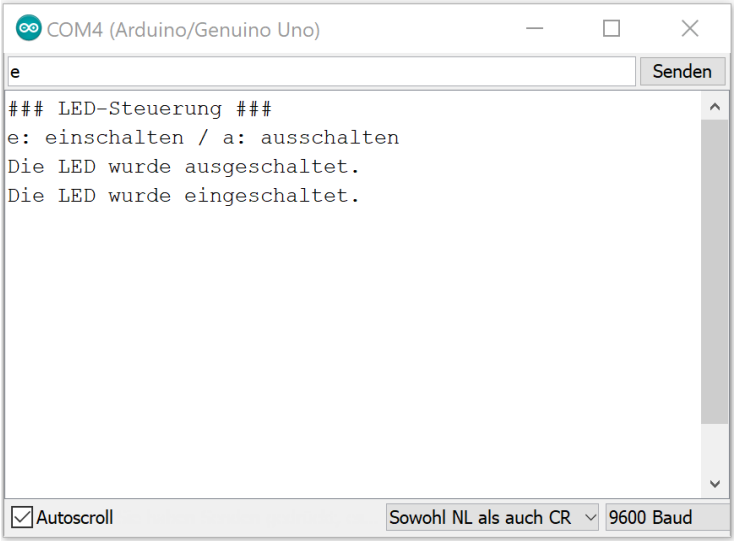


Abb. 4.7 Das Programm reagiert auf Tastatureingaben

1. Der `setup()`-Bereich wird genutzt, um einmalig eine Überschrift nebst kurzer Anleitung auf dem Bildschirm auszugeben.
2. Werden Daten über die serielle Schnittstelle empfangen, schreibt der Mikrocontroller sie zunächst in einen kleinen Zwischenspeicher, aus dem wir sie abrufen können. Dieser Abruf ist aber nur sinnvoll, wenn überhaupt Daten vorliegen. Deshalb prüft diese `if`-Bedingung, ob etwas empfangen wurde. Die Funktion `Serial.available()` gibt in diesem Fall den Wert `true` zurück, ansonsten `false`. Sie wundern sich womöglich, wieso als Bedingung nicht "`Serial.available()) == true`" notiert wurde.

Diese ausführliche Schreibweise ist natürlich ebenfalls richtig. Die Verkürzung ist allerdings erlaubt. Wenn Sie keinen Operator verwenden, wird die Bedingung automatisch so ausgewertet, als wenn Sie mit "`! = 0`" verknüpft worden wäre. Das bedeutet: Jeder Wert ungleich 0 gilt als erfüllte Bedingung, lediglich 0 (oder `false`) gilt als nicht erfüllte Bedingung.

3. Wenn etwas empfangen wurde, wird das nächste verfügbare Empfangsbyte mittels der Funktion `Serial.read()` aus dem Zwischenspeicher geladen und der Variable Eingabe zugewiesen. Da diese Variable an dieser Stelle das erste Mal vorkommt, wird hier ihr Typ deklariert, in diesem Fall `char`.

4. Diese `if`-Bedingung prüft nun, ob es sich bei der Eingabe um ein „e" handelte, indem es den Wert der Variable mit dem entsprechenden Zeichen vergleicht. Im Hintergrund werden in Wirklichkeit die Zahlen verglichen, die dem entsprechenden Buchstaben zugeordnet sind. Deshalb steht das „e" in Hochkommata. Ist die Bedingung erfüllt, also wurde ein „e" eingegeben, wird die LED eingeschaltet und der Vorgang entsprechend am Monitor bestätigt.

Die restlichen Programmschritte kennen Sie bereits. Nun haben wir also unser erstes interaktives Programm realisiert!

4.9 Arrays

Beschäftigen wir uns nun mit der Frage, wie sich eine größere Anzahl von Variablen günstig handhaben lässt. Als Beispiel gehen wir davon aus, dass ein anderer Teil unseres Programmes verschiedene Sensordaten (in Form von Zahlen) erfasst hat. Da wir bisher noch keine Sensoren kennengelernt haben, nehmen wir diese Werte zunächst einfach als gegeben an. Auf Knopfdruck soll das folgende Programm nun eine von zwei Messdatenreihen anzeigen:

```
1  int MessreiheA[5] = {503, 287, 903, 15, 91};                  // (1)
2  int MessreiheB[] = {63, 48, 720, 612, 102};                   // (2)
3
4  void setup()
5  {
```

```
  Serial.begin(9600);
  Serial.println("### Messergebnisanzeige ###");
  Serial.println("a: Messreihe A / b: Messreihe B");
}

void loop() {
  if (Serial.available()) {
    char Eingabe = Serial.read();
    if(Eingabe == 'a')
    {
      Serial.println();
      Serial.println("Datenreihe A:");
      for(byte i=0; i < 5; i++)                              // (3)
      {
        Serial.print(MessreiheA[i]);                         // (4)
        Serial.print("  ");                                  // (5)
      }
      Serial.println("# Ende der Datenreihe #");            // (6)
    }
    if(Eingabe == 'b')
    {
      Serial.println();
      Serial.println("Datenreihe B:");
      for(byte i=0; i < 5; i++)
      {
        Serial.print(MessreiheB[i]);
        Serial.print("  ");
      }
      Serial.println("# Ende der Datenreihe #");
    }
  }
}
```

Die Ausgabe sieht beispielhaft so aus:

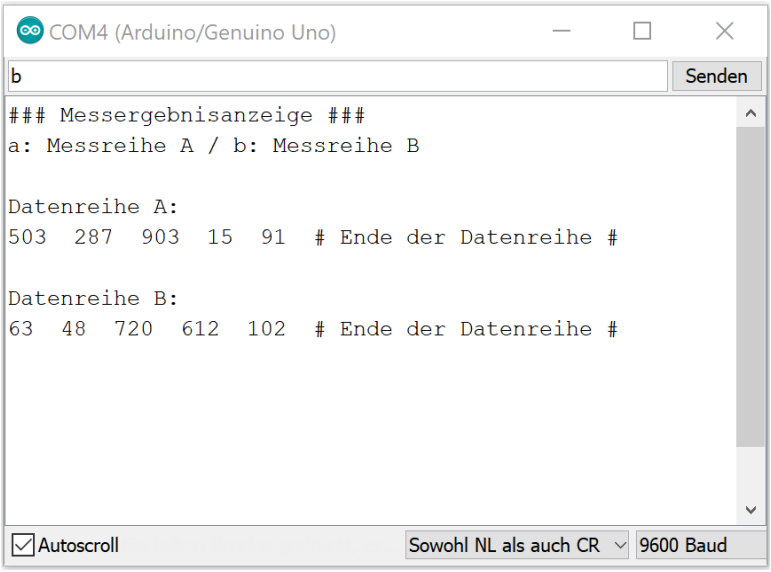


Abb. 4.8 Demonstration von Feld-Variablen

Schauen wir uns den Sketch genauer an:

1. Hierbei handelt es sich um eine Variablendeklaration vom Typ `int` – allerdings nicht nur eine, sondern gleich 5. Um für gleichartige Daten nicht immer neue Namen erfinden zu müssen, kann man sie einfach durchnummerieren, analog zu den kleinen Ziffern (Indizes) bei den Widerständen R_1 und R_2 unseres Spannungsteilers im Kapitel 3. Dafür muss man dem Compiler bei der Deklaration genau mitteilen, wie viele solcher Variablen benötigt werden – denn er muss ja den Speicher dafür reservieren. In unserem Fall wird er 5 x 16 Bit (Typ `int`) vormerken. Im Programm kann man die Variablen dann einfach über ihre Nummer (Index) abrufen, man beginnt dabei immer mit der 0. Somit heißen die hier deklarierten Variablen „`MessreiheA[0]`, `MessreiheA[1]`, … , `MessreiheA[4]`“. Man nennt diese Gruppe gleichartiger Variablen auch *Array* (Feld). Da Variablen bei der Deklaration auch gleich initialisiert werden dürfen, weisen wir ihnen hier beispielhafte Werte zu.

2. Die ebenfalls fiktive Messreihe B wird nach der gleichen Art deklariert und initialisiert. Jedoch wurde hier die Größenangabe des Arrays, also die 5 in den eckigen Klammern, weggelassen. Dies ist erlaubt, da der Compiler an den direkt zugewiesenen Werten erkennt, dass es sich um ein Array mit 5 Elementen handeln soll.
3. Diese `for`-Schleife wird fünfmal durchlaufen, als Zählvariable verwenden wir `i` – dieser Buchstabe wird in Anlehnung an die Mathematik häufig verwendet, wenn einfach lokal eine Zählvariable benötigt wird. Da die im Schleifenkopf deklarierte Variable ohnehin nur für die jeweilige Schleife gilt, und außerhalb dieser Kontrollstruktur gar nicht existiert, können wir sogar in beiden `for`-Schleifen dieses Sketches den Namen `i` verwenden.
4. An die Ausgabefunktion `Serial.print()` wird nun der Wert der Variablen `Messreihe[0]` bis `Messreihe[4]` übergeben, indem die Schleife fünfmal durchlaufen wird und dabei der jeweils aktuelle Wert der Zählvariablen `i` als Index benutzt wird.
5. Die Leerzeichen werden an dieser Stelle nur ausgegeben, um am Bildschirm einen gewissen Abstand zwischen den Zahlen zu erreichen.
6. Wurde die Schleife fünfmal durchlaufen, wird ein Hinweis auf das Ende der Datenreihe ausgegeben und der Block der `if`-Bedingung danach verlassen.

Bei der Arbeit mit Arrays ist es wieder nützlich, wenn man grob versteht, wie der Compiler Speicheradressen aufteilt. Wir hatten bereits festgestellt, dass Variablen bei ihrer ersten Nennung deklariert werden müssen – das heißt man muss mitteilen, wie viel Speicherplatz sie belegen sollen. Stellen Sie sich den Speicher vor wie ein großes Lager mit Zehntausenden Regalen, jedes hat acht Schubladen. Jede Schublade kann genau ein Bit, also 0 oder 1, speichern. Ein Regal speichert damit genau ein Byte (8 Bit). Nun werden die Regale durchnummeriert, so dass jedes eine eindeutige Adresse hat.

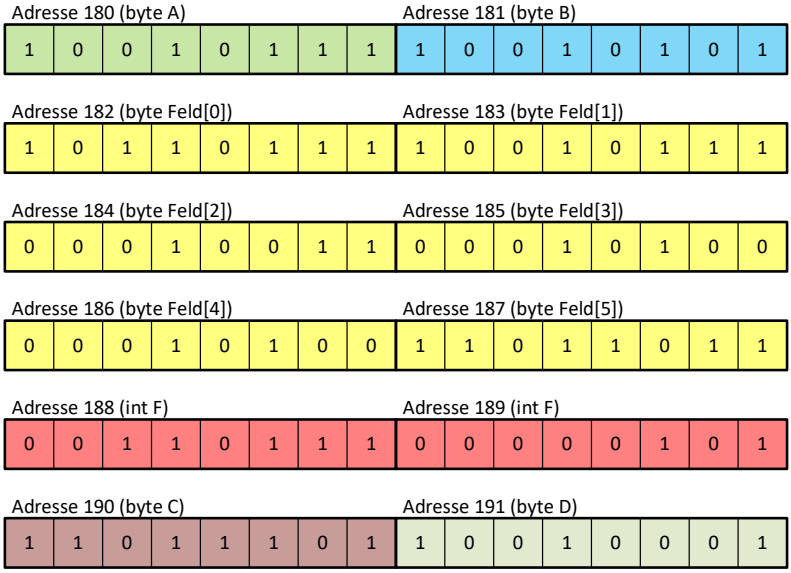


Abb. 4.9 schematischer Ausschnitt aus dem Arbeitsspeicher

Deklarieren Sie nun die Variable A als `byte`, wird der Compiler einen konkreten Bereich von 8 Bit dafür reservieren und dessen Adresse im Maschinencode an allen Programmstellen vermerken, wo Sie auf „A“ zugreifen. Obige Abbildung zeigt schematisch einen Ausschnitt des Arbeitsspeichers in welchem zusätzlich noch weitere Variablen abgelegt wurden, unter anderem auch das Array `Feld[]` vom Typ `byte` mit 6 Elementen. Wenn der Compiler dies vorbereitet, reserviert er den benötigten Platz (Bitzahl des Variablentyps mal Anzahl der Elemente) und vermerkt dennoch nur die Adresse des allerersten Elements (`Feld[0]`) im Maschinencode. Greifen Sie nun an einer beliebigen Programmstelle beispielsweise auf die Variable `Feld[3]` zu, so berechnet der Mikrocontroller die Adresse dieses Elementes selbst, indem er die vom Compiler hinterlegte Adresse des ersten Elementes nimmt und dazu 3 addiert.

Dies birgt allerdings eine Gefahr: Ruft das Programm versehentlich durch eine zu große Zahl ein Element ab, welches es gar nicht gibt (z.B. Feld[7]), so wird der Mikrocontroller dennoch eine Adresse berechnen, und somit Daten von einer Speicherstelle laden, an der irgendetwas anderes (in diesem Beispiel einige Bits der `int`-Variable F) gespeichert ist. Dies sorgt für unvorhersehbare Ergebnisse. Noch ungünstiger ist es, wenn dadurch Daten an eine entsprechend falsche Speicherstelle geschrieben werden. Das Ergebnis ist oft der Absturz des Mikrocontrollers, welcher daraufhin automatisch neu startet. Falls Sie ein Nutzer von Windows 95 waren, erinnern Sie sich sicherlich an die gefürchteten „Blue-Screen-Abstürze“. Adressierungsfehler wie die eben beschriebenen waren eine häufige Ursache für diese Abstürze.

Um weitere Messreihen nicht immer neu Deklarieren zu müssen, können wir das Array auch in ein mehrdimensionales Array verwandeln. Einzelne Elemente werden dann über zwei Indizes angesprochen, an der prinzipiellen Funktion ändert sich jedoch nichts.

4.10 Zeiger

Ausgehend von der soeben erörterten Speicherverwaltung des Compilers nehmen wir nun einen kleinen Exkurs in die fortgeschrittene Programmierung, da wir manche merkwürdige Schreibweise zumindest verstehen wollen, auch wenn wir sie selbst zunächst nicht anwenden werden. Dafür besinnen wir uns nochmal auf die Art, wie Variablen gespeichert werden. Wir hatten bereits festgestellt, dass der Compiler, abhängig vom Variablentyp, einen gewissen Speicherplatz reserviert. Bei der Benennung von Variablen dürfen wir die Bezeichnung selbst wählen – der Compiler wird sie sowieso entfernen und durch die zugewiesene Speicheradresse ersetzen. In Wirklichkeit steht an einer Programmstelle, an der wir eine Variable verwenden, im Maschinencode also eigentlich ein Verweis auf die Position im Arbeitsspeicher – ein sogenannter Zeiger. Unter bestimmten Umständen kann es nötig sein, diese Speicheradresse zu ermitteln. Testen wir dazu den folgenden Sketch, indem das Vorgehen an zwei Beispielvariablen demonstriert wird:

```
int a = 12345;
int b = 10000;

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

void loop() {
  Serial.print("Wert von a: "); Serial.println(a);
  int adresse_a = &a;                                    // (1)
  Serial.print("Adresse von a: "); Serial.println(adresse_a);

  Serial.print("Wert von b: "); Serial.println(b);
  int* adresse_b = &b;                                   // (2)
  Serial.print("Adresse von b: "); Serial.println((int)
  adresse_b);                                            // (3)
  Serial.print("Wert von b: "); Serial.println(*adresse_b);// (4)

  delay(100000);
}
```

Nach dem Hochladen gibt er Folgendes auf dem seriellen Monitor aus:

Abb. 4.10 Speicheradressen stehen nicht für einen Zahlenwert, sondern nur für eine konkrete Position im Arbeitsspeicher

Der Sketch gibt die Adressen von beiden Testvariablen aus. Bei Ihnen können die Adressen dabei auch völlig anders lauten – abhängig davon, welche Speicherposition Ihr Compiler den beiden Werten zugewiesen

hat. In obigem Screenshot hatte der Compiler beide Variablen auf benachbarten Plätzen abgelegt: 256 und 258. (Der Abstand von 2 ergibt sich daraus, dass eine `int`-Variable zwei Byte im Speicher belegt; siehe *Abbildung 4.9*.)

1. Stellt man den Operator `&` vor einen Variablennamen, dann wird an dieser Stelle statt des Wertes der Variable die entsprechende Speicheradresse zurückgegeben. Diese wird hier einer Variablen vom Typ `int` zugewiesen. Dieses Vorgehen funktioniert, ist allerdings keine saubere Programmierung! Würde dieser Sketch auf einen Mikrocontroller mit sehr großem Arbeitsspeicher ausgeführt, würde der Typ `int` nicht mehr ausreichen, um alle Speicheradressen aufzunehmen, und das Programm könnte abstürzen.
2. So sieht die korrekte Schreibweise aus. Die Hilfsvariable `adresse_b` hat nun nicht den Typ `int`, sondern den Typ „Zeiger auf eine int-Variable“, dies wird durch das Sternchen signalisiert. Natürlich sind Zeiger auch auf alle anderen Variablentypen, sogar auf andere Zeiger, möglich.
3. Da die Variable `adresse_b` eigentlich ein Zeiger ist, erzeugt der Compiler eine Fehlermeldung, wenn wir sie einfach per `println()` ausgeben wollen. Das vorangestellte `(int)` weist den Compiler an, sie für die dortige Anwendung in eine Variable vom Typ `int` umzuwandeln und erst dann an `println()` zu übergeben.
4. Hier wird nun der umgekehrte Weg genutzt: Das vorangestellte Sternchen greift auf den Wert zu, welcher an der in der Variable `adresse_b` hinterlegten Speicheradresse zu finden ist. Erwartungsgemäß ist dies der Wert der Variablen `b`.

Das Arbeiten mit Zeigern sollten Sie sich aufheben, bis Sie einige Routine in der Programmierung haben. Es ergeben sich daraus viele Fehlerquellen, die schwer zu lokalisieren sind. Dennoch sollten Sie an Schreibweisen wie `&zahl_a` zumindest erkennen, dass es sich um einen Zeiger handelt – beim Arbeiten mit vorgefertigtem Code für gewisse externe Komponenten lässt sich das unter Umständen nicht vermeiden.

4.11 Funktionen

Wir verfolgen weiter unser Beispiel aus Kapitel 4.9. Nun wollen wir die Möglichkeit schaffen, zu jedem Messwert seine Quersumme anzuzeigen. Dafür schreiben wir ein kleines Unterprogramm – unsere erste selbsterstellte Funktion. Deren Prinzip wurde abgeleitet von mathematischen Funktionen. Vielleicht erinnern Sie sich noch an die Sinus-Funktion. Taucht in einer Formel beispielsweise der Ausdruck „ $\sin(50)\cdot 4$ “ auf, so ist zunächst über eine Berechnung der Sinuswert der Zahl 50 zu bestimmen und deren Ergebnis dann mit 4 zu multiplizieren. Ähnlich verhält es sich auch mit Funktionen in der Informatik: Man kann ihnen in runden Klammern Werte übergeben (sogenannte Argumente) und sie können das Ergebnis einer Berechnung zurückliefern. Beides ist jedoch als Option zu sehen, nicht als Verpflichtung. Sie kennen bereits Funktionen, die ein Argument benötigen, aber keinen Rückgabewert liefern – so zum Beispiel `delay()`. Andererseits kennen Sie auch Funktionen, die eine Rückgabe liefern, aber kein Argument erwarten, zum Beispiel `millis()`.

Schauen wir uns zunächst das Beispiel an und testen seine Funktionalität:

```
1  int Messreihe[2][5] = {{503, 287, 903, 15, 91},            // (1)
2                         {63, 48, 720, 612, 102}};           // (2)
3
4  void setup()
5  {
6    Serial.begin(9600);
7    Serial.println("### Messergebnisanzeige ###");
8    Serial.println("a: Messreihe A / b: Messreihe B");
9  }
10
11 void loop() {
12   if (Serial.available()) {
13     char Eingabe = Serial.read();
14     if(Eingabe == 'a')
15     {
16       Serial.println();
17       Serial.println("Datenreihe A:");
18       for(byte i=0; i < 5; i++)
19       {
20         Serial.print(Messreihe[0][i]);                      // (3)
```

```
21          Serial.print(" (Quersumme: ");
22          Serial.print(QSumme(Messreihe[0][i]));
23          Serial.print(")  ");
24        }
25        Serial.println("# Ende der Datenreihe #");
26      }
27      if(Eingabe == 'b')
28      {
29        Serial.println();
30        Serial.println("Datenreihe B:");
31        for(byte i=0; i < 5; i++)
32        {
33          Serial.print(Messreihe[1][i]);
34          Serial.print(" (Quersumme: ");
35          Serial.print(QSumme(Messreihe[1][i]));              // (4)
36          Serial.print(")  ");
37        }
38        Serial.println("# Ende der Datenreihe #");
39      }
40    }
41  }
42
43  byte QSumme(int Zahl)                                      // (5)
44  {                                                          // (6)
45    byte Ergebnis = 0;                                       // (7)
46    while(Zahl > 0)                                          // (8)
47    {
48      Ergebnis = Ergebnis + Zahl%10;                         // (9)
49      Zahl = Zahl / 10;                                     // (10)
50    }
51    return Ergebnis;                                       // (11)
52  }
```

Die Ausgabe dieses Sketches sieht nun beispielhaft so aus:

Abb. 4.11 Funktionen können häufig wiederkehrende Vorgänge vereinfachen

1. Nun wurde das Array wie besprochen mit zwei Indizes angelegt. Der Compiler reserviert jetzt also 2 x 5 x 8 Bit im Arbeitsspeicher. Die Initialisierung sieht jetzt geringfügig anders aus: Die Werte werden nun in geschweiften Klammern gruppiert und ineinander verschachtelt.
2. Da der Compiler ja Zeilenumbrüche ignoriert, können wir die Wertzuweisung zur besseren Übersicht auf zwei Zeilen aufteilen.
3. Der Abruf der Array-Variablen hat sich durch die zweite Indexzahl kaum verändert. Ihre Verwendung ist naheliegend: In unserem Beispiel repräsentiert der Index 0 die vorherige Messreihe A, der Index 1 steht entsprechend für Messreihe B.
4. Um die Quersumme auszugeben, wird hier die Funktion `QSumme()` aufgerufen. Allerdings kennt die Arduino IDE keine solche Funktion. Wir müssen sie deshalb in diesem Sketch noch definieren – also erklären, was mit dieser Funktion überhaupt gemeint ist.
5. Hier wird die eigentliche Funktion `QSumme()` programmiert. Derartige Funktionsdefinitionen erfolgen stets außerhalb der `setup()`- und `loop()`-Blöcke. Ob nun davor oder dahinter, ist jedoch egal. Wie bei einer Variablen auch, benötigt der Compiler genaue Informationen, wieviel Speicher er reservieren muss. Deshalb hat auch eine Funktion einen bestimmten Typ. Dieser bezieht sich dabei auf den Rückgabewert, in unserem Fall also die Quersumme. Der Typ `byte` reicht dafür völlig aus, schließlich ist selbst die Quersumme von 99999 nur 45. Hat eine Funktion keinen Rückgabewert, macht man dies durch das Schlüsselwort `void` kenntlich. Es folgt der Name der Funktion. Wieder dürfen keine Leerzeichen, Sonderzeichen oder bereits reservierte Worte genutzt werden. Direkt anschließend werden die Argumente deklariert. Auch für sie ist der Datentyp anzugeben. Mehrere Argumente können durch Kommata getrennt werden. Benötigt die Funktion keine Argumente, bleiben die runden Klammern einfach leer. Unsere beiden Haupt-Programmteile `setup()` und `loop()` sind also auch Funktionsdeklarationen – ohne Argumente und ohne Rückgabewert.

6. Da die Funktion üblicherweise aus mehreren Befehlen besteht, werden diese wieder in geschweifte Klammern eingeschlossen.

7. Für unsere Berechnung benötigen wir eine Hilfsvariable, die wir einfach innerhalb der Funktion deklarieren können. Diese Variable ist dann außerhalb der Funktionsdeklaration nicht „sichtbar" – wenn Sie innerhalb der `loop()`-Programmteils also auf die Variable `Ergebnis` zugreifen würden, erhielten Sie eine Fehlermeldung des Compilers, da er in diesem Gültigkeitsbereich keine Variable mit diesem Namen findet.

8. Die eigentliche Quersummenberechnung führen wir durch, indem wir die Einer der übergebenen Variable `Zahl` zur Variable `Ergebnis` hinzuaddieren. Dann verschieben wir alle Ziffern von `Zahl` nach rechts und wiederholen den Vorgang so lange, bis keine Ziffern übrig sind. Dies ist somit auch unsere Schleifenbedingung: Solange `Zahl` größer als 0 ist, gibt es noch etwas zu tun, sonst ist die Berechnung fertig.

9. Der Zahlenwert der Variable `Zahl` wird einer Modulodivision[9] durch 10 unterzogen. Der Rest sind die Einer. Diese Zahl wird zum bisherigen Wert der Variable `Ergebnis` addiert und anschließend wieder der Variable `Ergebnis` zugewiesen. Als verkürzte Schreibweise für diese Zeile wäre übrigens auch folgendes zulässig: `Ergebnis+= Zahl%10;` Der Operator += addiert den rechten Wert zum Wert der linken Variable und weist ihr diese Summe wieder zu.

10. Nun wird der Zahlenwert der Variable `Zahl` ganzzahlig[10] durch 10 geteilt, um alle Ziffern nach rechts zu verschieben. Das Ergebnis

9 Diese Rechenart kennen Sie aus der Grundschule: Es geht um die stets ganzzahlige „Division mit Rest". Eine Modulodivision liefert dabei nur den Rest zurück. Als Rechenzeichen nutzt man das Prozent-Symbol. Ein Beispiel: 13 % 6 = 1, denn „13 durch 6" ergibt "2 Rest 1".

10 Da der Datentyp int gewählt wurde, also ein ganzzahliger Wert vorgesehen ist, lässt der Compiler automatisch nur ganzzahlige Divisionen zu. Ein eventueller Rest wird einfach verworfen. Ein Beispiel: 208 / 10 = 20.

wird ihr wieder zugewiesen. Analog zu (9) ließe sich auch diese Zeile wie folgt verkürzen: `Zahl/= 10;`

11. Nachdem die `while`-Schleife verlassen wurde, enthält die Variable Ergebnis die fertige Quersumme. Der Befehl `return` gibt diesen Wert nun als Rückgabewert weiter und beendet gleichzeitig die Bearbeitung der Funktion, der Programmablauf wird nun an der Stelle fortgesetzt, von der aus die Funktion aufgerufen wurde.

    Die Funktion `QSumme()` stellt also einen einfachen mathematischen Algorithmus zur Quersummenberechnung dar.

4.12 Objektorientierte Programmierung

Stellen Sie sich vor, Sie planen ein Projekt zur Gebäudeautomatisierung. Das Gebäude hat dutzende Räume. In jedem möchten Sie die Temperatur erfassen, die Heizung steuern, das Licht schalten und so weiter. Nun könnten Sie in Ihrem Programmcode für all diese Werte und Aufgaben entsprechende Variablen und Funktionen definieren.

```
1  float Temperatur_Lobby;
2  float Temperatur_Teekueche;
3  float Temperatur_Raum1;
4  float Temperatur_Raum2;
5
6  ...
7
8  void SchalteLichtEin_Lobby() {
9  ...
10 }
11
12 void SchalteLichtEin_Teekueche() {
13 ...
```

Es ist offensichtlich, dass hierbei trotz disziplinierter Namenswahl schon bald die Übersicht verlorengeht. Und dann beginnt das Chaos: Womöglich wird innerhalb der Funktion `SchalteLichtEin_Teekueche()` versehentlich das Licht der Lobby mitgesteuert, weil der Programmcode zuvor kopiert wurde. Oder es wird später die falsche Temperatur angezeigt, da `Raum1` mit `Raum2` verwechselt wurde.

Um größere Projekte besser zur strukturieren und Fehler zu vermeiden, wurde vor rund 30 Jahren die objektorientierte Programmierung eingeführt. Ziel war es, die Strukturen in einem Programm näher an die reale Welt anzulehnen und dadurch besser nachvollziehbar zu machen. Eine der zentralen Methoden dafür ist die Nutzung von sogenannten *Klassen* als Vorlagen für Objekte.

In unserem Beispiel könnte es eine Klasse „`Raum`“ geben, welche zunächst einfach nur definiert, dass jeder Raum eine Temperatur (vom Typ `float`) hat, außerdem verfügt jeder Raum über eigene Funktionen zum Schalten der Beleuchtung und so weiter. Im weiteren Quellcode müssen wir dann nur noch für jeden konkreten Raum eine Ausprägung dieser Klasse, also eine sogenannte *Instanz*, anlegen.

Dies kann man an einem weiteren Beispiel beschreiben: Sie möchten die Ersatzteile in Ihrem Bastelkeller katalogisieren. Nun könnten Sie für jedes gefundene Teil ein Blatt Papier nehmen und darauf „Artikel: Kabelrolle. Wert: 5 Euro. Zustand: ...“ notieren. Sie könnten Sich aber auch einen Stempel anfertigen, der bereits eine Tabelle nach dem Muster

Artikel: _____
Wert: _____
Zustand: _____
...

auf das Papier druckt. In diesem Beispiel steht der Stempel für die Klasse und die damit bedruckten Papierseiten für dessen Ausprägungen (Instanzen).

Zurück zu unserem Beispiel der Gebäudeautomation. Wir nutzen unsere Klasse „`Raum`“, um damit für jeden realen Raum eine Instanz, wir nennen sie auch *Objekt*, zu erstellen. Dies sieht im Quelltext ganz ähnlich aus wie eine Variablendeklaration:

```
1  Raum Lobby;                                          // (1)
2  Raum Teekueche(ON);                                  // (2)
3  Raum Raum1;
4  Raum Raum2;
5
6  ...
7  void loop() {
8  ...
9    Lobby.SchalteLichtEin();                           // (3)
10   Serial.print(Teekueche.Temperatur);                // (4)
11 ...
```


1. Hier wird das Objekt Lobby erzeugt, es ist eine Instanz der Klasse `Raum`. Bildlich gesprochen drückt der Compiler dabei einen Stempel in den Arbeitsspeicher, der dadurch auf einen Schlag den Platz für die Variablen eines Raumes (Temperatur, ...) und auch für dessen Funktionen reserviert.
2. Klassen dürfen während der Erstellung eines Objektes auch Argumente annehmen (wenn sie entsprechend definiert wurden). Hier könnte man beispielsweise damit das Licht in der Teeküche standardmäßig Einschalten. Manche Klassen lassen auch eine Zuweisung mittels „=“ (wie bei einer Variablen) zu.
3. Um nun auf eine Funktion eines konkreten Objektes zuzugreifen, nutzt man dessen Name gefolgt von einem Punkt. Dahinter kann man auf Funktionen und
4. Variablen zugreifen, welche die Klasse öffentlich bereitstellt.

Wohlgemerkt: Die eigentliche Klassendefinition wurde hier bewusst nicht dargestellt, dies würde den Rahmen sprengen. Das Thema objektorientierte Programmierung (OOP) ist derart umfangreich, dass es komplette Bücher füllt. Als Einsteiger genügt es zunächst, wenn Sie wissen, dass es Klassen und daraus erzeugte Objekte gibt und es Ihnen kein Kopfzerbrechen bereitet, wenn in folgenden Beispielen Funktionsnamen mit Punkt auftauchen. Genau genommen haben Sie ja schon ein Objekt benutzt: `Serial` wird von der Arduino IDE benutzt, um die serielle Schnittstelle zu repräsentieren. `Serial.print()` ist eine Funktion dieses Objektes.

Die objektorientierte Programmierung kann ihre Vorteile bei großen und komplexen Projekten (zum Beispiel der Entwicklung von Anwendersoftware am Computer oder der Programmierung von Web-Portalen) richtig ausspielen. Für unsere Arduino-Projekte mit dem im Vergleich dazu winzigen Programmspeicher und maximal einigen hundert Zeilen Quelltext bietet sie hauptsächlich dann Vorteile, wenn Programmcode für spätere Projekte wiederverwendet werden soll. Daher werden Sie bei der Nutzung von Bibliotheken häufig damit konfrontiert, dass beispielsweise angeschlossene Sensoren als Objekt dargestellt sind. Die Beispiele in den folgenden Kapiteln helfen Ihnen, sich schnell daran zu gewöhnen. In Kapitel 15 werden wir dann auch selbst in die OOP einsteigen und sogar eine eigene Bibliothek erstellen.

4.13 Zeichenketten (Strings)

Abschließend richten wir den Blick auf Zeichenketten (englisch Strings). Sie kommen oft zur Anwendung, wenn Textausgaben auf Displays oder dem PC generiert werden sollen. Ursprünglich mussten Zeichenketten immer als Feld vom Typ `char` angelegt werden. Mittlerweile gibt es die Klasse `String`, welche einen komfortableren Umgang damit erlaubt – die Zeichenketten werden dabei zu Objekten.

```
1  char TextA[7] = {'H', 'a', 'l', 'l', 'o', '!'};            // (1)
2  char TextB[] = "Hallo";                                    // (2)
3  String TextC = "Hall";                                     // (3)
4  String TextD = "o Welt!";
5
6  void setup() {
7    Serial.begin(9600);
8  }
9
10 void loop() {
11   Serial.print("TextA: "); Serial.println(TextA);
12   Serial.print("TextB: "); Serial.println(TextB);
13   Serial.print("TextC und D: "); Serial.println(TextC + TextD);
14                                                              // (4)
15   Serial.print("Länge von C: "); Serial.println(TextC.length());
16                                                              // (5)
17   delay(100000);
18 }
```

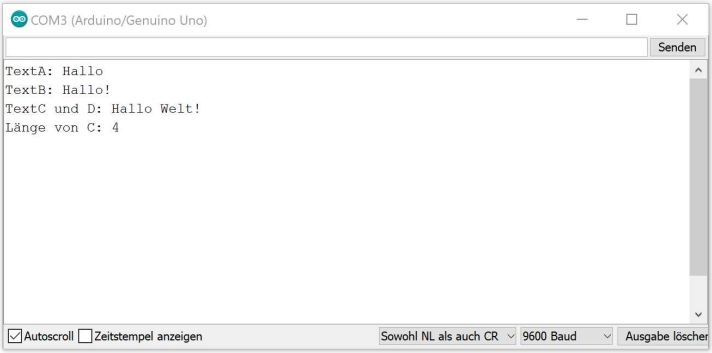


Abb. 4.12 Zeichenketten sind elementar für die Interaktion mit einem Nutzer per Bildschirm oder Display

1. Das ist die ursprüngliche Vorgehensweise zur Erzeugung einer Zeichenkette: Man legt ein Array vom Typ `char` an und legt jeden Buchstaben einzeln darin ab. Die Elementanzahl 7 ergibt sich daraus, dass der Compiler bei `char`-Feldern am Ende noch ein spezielles Byte zur Markierung des Endes der Zeichenkette einfügt.
2. In dieser vereinfachten Variante erkennt der Compiler selbständig die Länge und erzeugt das Array aus der Zeichenkette. Wichtig hierbei sind die Anführungszeichen (im Gegensatz zu den Hochkommata bei (1))
3. Deutlich komfortabler wird es, indem wir ein Objekt aus der Klasse `String` erzeugen, denn
4. Aneinanderreihungen durch einfache (scheinbare) Addition und weitere
5. Funktionen, wie zum Beispiel das Ermitteln der Länge, sind mit der `String`-Klasse sehr einfach realisierbar. Wenn Sie die gleiche Funktionalität mit dem `char`-Feld erreichen möchten, wäre das viel komplexer.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 5
Ein- und Ausgänge

Unsere bisherigen Versuche beschränkten sich auf einfache Programme, deren Ein- und Ausgabe stets über einen Computer erfolgte. Jedoch ist der selbständige Einsatz ohne einen PC eigentlich gerade eine der Stärken von Mikrocontrollern. Nehmen wir nun also unserem Arduino die „Augenbinde“ ab und lernen ihm, sich zu äußern. Er soll nun selbständig mit der Umwelt interagieren. Dazu beschäftigen wir uns in diesem Kapitel zunächst mit den Schnittstellen, also den Anschlüssen unserer Platine, bevor wir uns in den folgenden Kapiteln mit konkreten Bauelementen zur Ein- oder Ausgabe von Daten auseinandersetzen.

5.1 Digitale Ausgänge

Abb. 5.1 Die digitalen Ausgangspins tragen die Nummern 0 bis 13

Das Arduino-Board stellt uns 14 digitale Ausgänge zur Verfügung, die wir uns zunächst wie Schalter vorstellen können, welche wir durch das Programm steuern. *Digital* meint in diesem Zusammenhang eigentlich sogar *binär*, also mit nur zwei möglichen Werten – nämlich 5 V (Be-

triebsspannung) und 0 V (Masse). Zwischenwerte sind nicht vorgesehen, allerdings lernen wir in Kapitel 5.4 noch ein Verfahren kennen, um dennoch die Spannung quasi gleitend zu verändern.

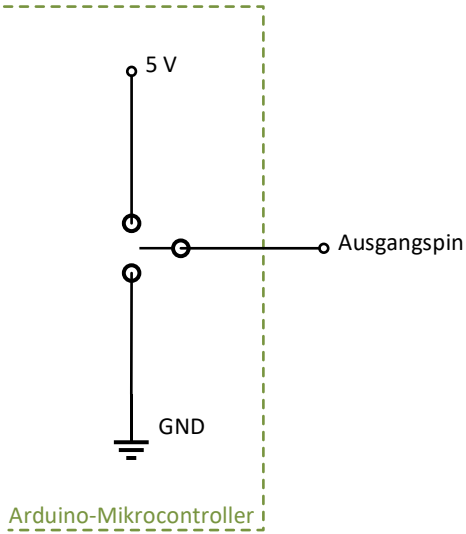


Abb. 5.2 Modellvorstellung – digitaler Ausgangspin als steuerbarer Umschalter

Wichtig im Zusammenhang mit digitalen Ausgängen ist es, ihre Strombelastbarkeit zu beachten. Jeder Ausgangspin unseres Arduino ist für eine Stromstärke von maximal 20 mA ausgelegt. Das war auch der Grund, warum wir in Kapitel 3.1.2.6 das Relais über einen Transistor anschließen mussten, statt es direkt mit dem Ausgangspin zu verbinden.

Anders gesagt: Wir müssen dafür sorgen, dass alles, was wir an einen Ausgangspin anschließen, einen Widerstand hat, der den Strom ausreichend begrenzt. Schließen wir also beispielsweise eine LED ohne Vorwiderstand an, können sowohl die LED als auch der Arduino beschädigt werden, da die Stromstärke zu hoch wird, sobald der Ausgangspin auf 5 V schaltet. Nutzen wir jedoch einen passenden Vorwiderstand (siehe Kapitel 3.1.2.4), wird es kein Problem geben.

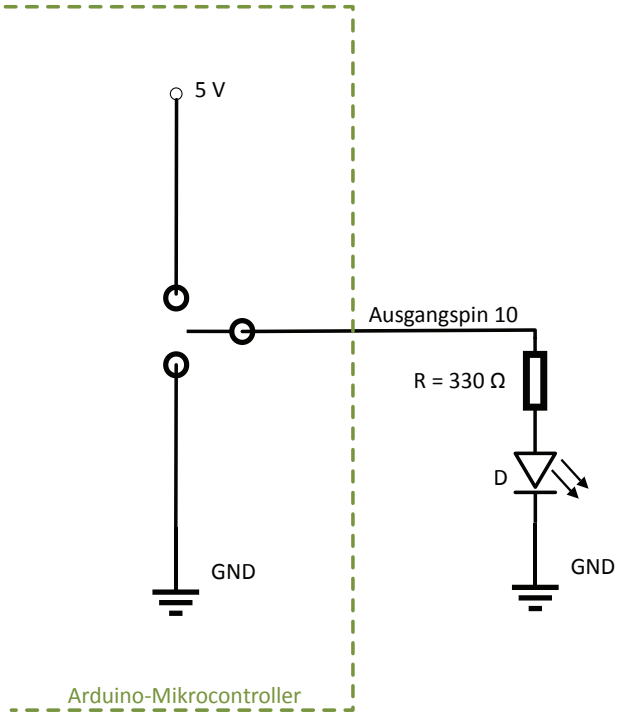


Abb. 5.3 Der Vorwiderstand begrenzt den Stromfluss durch die LED für den Fall, dass der Ausgangspin auf 5 V geschaltet ist

Diesen Versuchsaufbau wollen wir auch gleich für einen ersten Test verwenden. Wir nutzen dazu eine grüne LED und einen Widerstand von 330 Ohm als Vorwiderstand. Diesen Wert hatten wir in Kapitel 3.1.2.4 ermittelt. Sollten Sie eine andersfarbige LED oder eine größere Bauform nutzen, ergeben sich entsprechend der zulässigen Stromstärke und Flussspannung andere Widerstandswerte. Aufgrund der Toleranzen und der in unserer Berechnung niedrig angesetzten Stromstärke von 10 mA können Sie jedoch testweise auch einfach den gleichen Widerstand verwenden, gegebenenfalls erreicht Ihre LED dann nicht die volle Leuchtkraft – für unser hiesiges Beispiel ist dies jedoch zunächst nicht relevant.

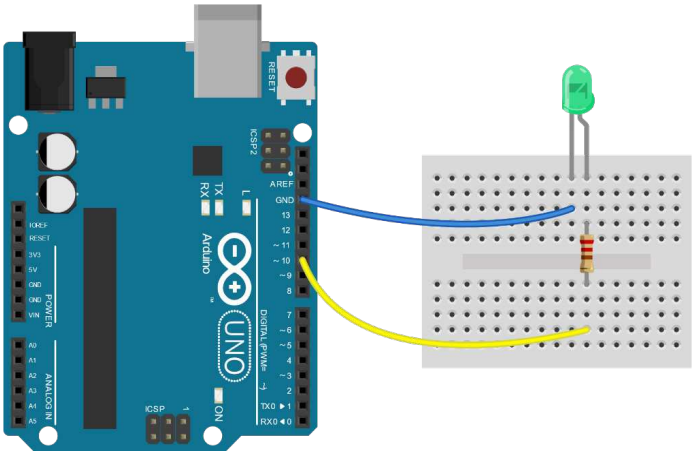


Abb. 5.4 Für dieses Experiment genügt ein kleines Breadboard. Hier sind intern jeweils 5 Löcher vertikal miteinander verbunden; so führt also der gelbe Draht zum unteren Pin des Widerstands, dessen andere Seite ist direkt mit dem rechten Pin (Anode) der LED verbunden. Der Kathodenanschluss der LED führt über den blauen Draht auf Masse.

Als Testprogramm können wir einfach wieder unser Blink-Beispiel aus Kapitel 4.2 nutzen und es leicht modifizieren.

```
1  void setup() {
2    pinMode(10, OUTPUT);                                    // (1)
3  }
4  void loop() {
5    digitalWrite(10, HIGH);                                 // (2)
6    delay(1000);
7    digitalWrite(10, LOW);
8    delay(1000);
9  }
```

Die Befehle `pinMode()` und `digitalWrite()` kennen wir bereits, da wir unseren ersten Tests mit Hilfe der auf dem Board verbauten LED durchgeführt hatten. Unser Beispiel unterscheidet sich tatsächlich nur in der Nummer des angesprochenen Pins, hier wurde (willkürlich) Pin 10 gewählt.

1. Als erstes Argument wird dem Befehl `pinMode()` die Nummer des betreffenden digitalen Ausgangspins mitgeteilt, hier beispielhaft die 10.
2. Gleiches gilt entsprechend auch für die Funktion `digitalWrite()`.

Den Programmablauf kennen wir bereits aus dem Blink-Beispiel. Dass die LED nach dem Hochladen anfängt, im Sekundentakt zu blinken, ist somit keine Überraschung. Sie können diesen kleinen Versuch gern auch noch einmal mit Pin 13 wiederholen – dann werden erwartungsgemäß sowohl die onboard-LED als auch die LED auf dem Breadboard im Gleichtakt blinken.

5.2 Digitale Eingänge

Um mit anderen Komponenten interagieren zu können, sind neben den gerade beschriebenen Ausgängen natürlich auch Eingänge wichtig. Zunächst beschäftigen wir uns mit den *digitalen* Eingangspins.

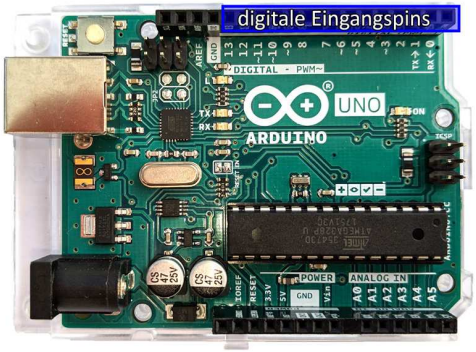


Abb. 5.5 Digitale Ein- und Ausgangspins sind identisch, ihre Nutzungsart wird im Programm festgelegt

Mit Blick auf das Board fällt auf, dass diese Pins 0 bis 13 exakt den vorhin beschriebenen Ausgangspins entsprechen. Diese Doppelbelegung

wurde von den Konstrukteuren bewusst gewählt, um Platz zu sparen. Ob ein bestimmter Pin nun als Aus- oder Eingang fungiert, kann man im Programm festlegen – und sogar während der Programmausführung ändern.

In Kapitel 4 haben wir bereits einen genaueren Blick auf die Zahlensysteme geworfen. Entsprechend der internen binären Rechnung verwendet der Arduino auch für die digitalen Eingänge das binäre Zahlensystem. Es sind also nur zwei verschiedene Werte zulässig: `LOW` und `HIGH`, repräsentiert durch entsprechende Spannungen:

Binärer Wert	zulässige Spannungen
`0 / LOW / false`	-0,5 ... 1,5 V
`1 / HIGH / true`	3,0 ... 5,5 V

Tabelle 5.1 Spannungsbereiche und zugeordnete Logikwerte

Bewegt sich die Spannung am Pin außerhalb dieser Grenzen, ist der Status nicht definiert – das heißt, es kann nicht vorausgesagt werden, ob der Arduino dies nun als „1" oder als „0" wertet. Solche undefinierten Zustände können zu ungewollten Ergebnissen führen und sollten daher vermieden werden. Spannungen unter -0,5 V beziehungsweise über 5,5 V können den Mikrocontroller zudem dauerhaft beschädigen.

Der Eingang selbst verhält sich gegenüber der Außenwelt wie ein sehr großer Widerstand (etwa 5 Megaohm) gegen Masse. Es fließt also fast gar kein Strom, somit belastet er die Quelle seines Signals nicht. Man kann ihn dadurch auch direkt mit 5 V oder GND verbinden, ohne die Gefahr eines Kurzschlusses betrachten zu müssen.[1]

Nun möchten wir einen simplen Taster mit dem Eingang verbinden. Intuitiv könnte man auf die Idee kommen, den Taster einfach mit dem Eingangspin und auf der anderen Seite mit der 5 V oder GND zu verdrahten. Allerdings ergibt sich dadurch ein Problem: Ist der Taster of-

1 Dies gilt natürlich nicht, wenn der Pin (womöglich versehentlich) als Ausgang konfiguriert wird.

fen (also nicht gedrückt), ist der Pin überhaupt nicht mehr verbunden, man spricht dann von einem *offenen* Eingang. In diesem Fall wird sein Potential (und damit die Spannung) von quasi zufälligen Einflüssen wie elektrischen Feldern in der Umgebung bestimmt und ist damit unvorhersehbar. Wir möchten jedoch, dass sich unsere Spannung am Eingang stets innerhalb der definierten Werte für LOW oder HIGH bewegt. Abhilfe schafft ein Widerstand, der in dieser Funktion auch als Pull-Up-Widerstand[2] bezeichnet wird.

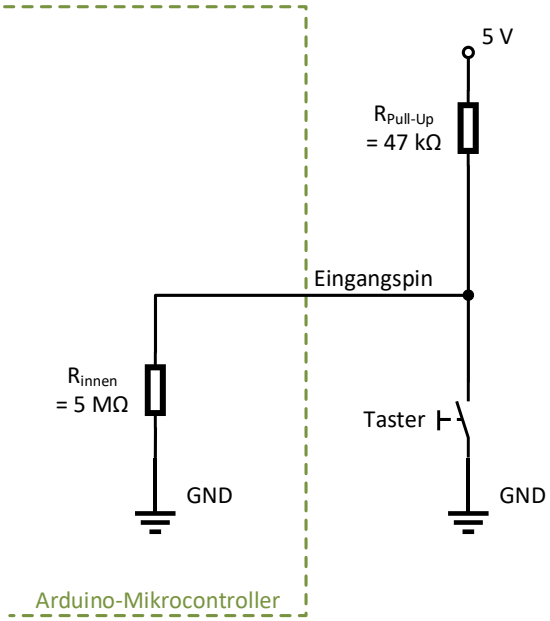


Abb. 5.6 mögliche Beschaltung eines Eingangspins mit separatem Pull-Up-Widerstand

Ist der Taster offen, bildet der Pull-Up-Widerstand zusammen mit dem im Vergleich sehr großen imaginären Innenwiderstand des Arduino

2 Der Begriff rührt von der bildlichen Vorstellung, dass dieser Widerstand das vorher undefinierte Potential am Eingangspin bei offenem Taster „nach oben zieht".

einen Spannungsteiler (siehe Kapitel 3.1.2.1), somit ist die Spannung am Eingang nahezu 5 V. Bei gedrücktem Taster wird diese Spannung zur Masse hin kurzgeschlossen. Durch den Pull-Up-Widerstand fließt dennoch nur ein sehr kleiner, vernachlässigbarer Strom (in diesem Beispiel 0,5 mA). Nun haben wir in jedem Fall eine definierte Spannung im zulässigen Bereich am Eingang: 5 V bei offenem Taster und 0 V bei geschlossenem Taster

Da ähnliche Anwendungsfälle in der Praxis relativ häufig vorkommen, haben die Konstrukteure bereits optionale Pull-Up-Widerstände in den Arduino-Chip eingebaut, die einfach im Programm aktiviert werden können. Somit genügt es für unser Beispiel tatsächlich, den Taster mit einem Eingangspin und GND zu verbinden.

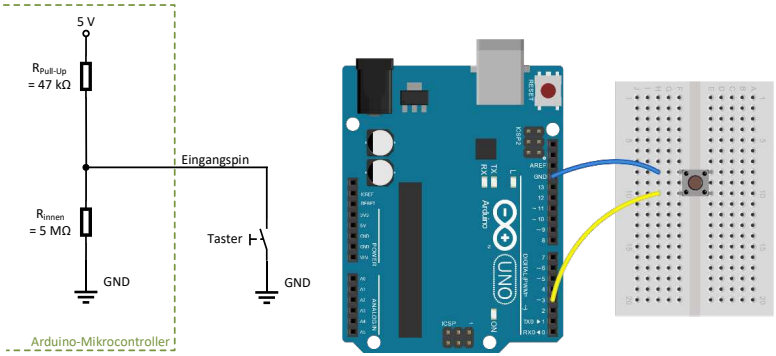


Abb. 5.7 Nutzung des internen Pull-Up-Widerstandes

Analog zu den Ausgangspins im vorigen Kapitel werden auch Eingangspins über die Funktion `pinMode(Pinnummer, Modus)` konfiguriert. Die Modi für Eingänge sind

- `INPUT_PULLUP` – Der Pin wird als Eingangspin genutzt, der interne Pull-Up-Widerstand ist aktiv.
- `INPUT` - Der Pin wird als Eingangspin ohne internen Pull-Up-Widerstand genutzt.

Die Abfrage des Eingangssignals erfolgt über die Funktion `digitalRead(Pinnummer)`, sie liefert entweder `false` (0) oder `true` (1) als Rückgabewert. Das folgende kleine Beispielprogramm liest den Zustand eines Tasters am beliebig gewählten Pin 3 und schaltet bei gedrücktem Taster (0 V am Eingang, Rückgabewert `false`) die interne LED des Arduino-Boards ein und bei offenem Taster aus.

```
1  void setup() {
2    pinMode(LED_BUILTIN, OUTPUT);
3    pinMode(3, INPUT_PULLUP);                                  // (1)
4  }
5
6  void loop() {
7    if(digitalRead(3) == false)                                // (2)
8      digitalWrite(LED_BUILTIN, HIGH);
9    else
10     digitalWrite(LED_BUILTIN, LOW);
11 }
```

1. Nachdem der Pin mit der onboard-LED als Ausgang definiert wurde, wird nun zusätzlich Pin 3 als Eingang konfiguriert und der interne Pull-Up-Widerstand aktiviert.
2. In der Programmschleife wird fortwährend der Status des Eingangspins 3 abgefragt. Ist der Taster gedrückt, liegen 0 V an und die Funktion `digitalRead()` liefert den Rückgabewert `false`. In diesem Fall wird die LED eingeschaltet, sonst aus.

Da die Funktion `digitalWrite()` als zweites Argument einen booleschen Wert erwartet und `digitalRead()` als Rückgabewert einen ebensolchen liefert, kann man obige Programmschleife sogar sehr stark verkürzen:

```
1  ...
2
3  void loop() {
4    digitalWrite(LED_BUILTIN, !digitalRead(3));                // (1)
5  }
```

1. Nun wird einfach der von `digitalRead()` zurückgegebene Wert durch den Operator `!` negiert und direkt als Argument an die Funktion `digitalWrite()` übergeben.

5.3 Analoge Eingänge

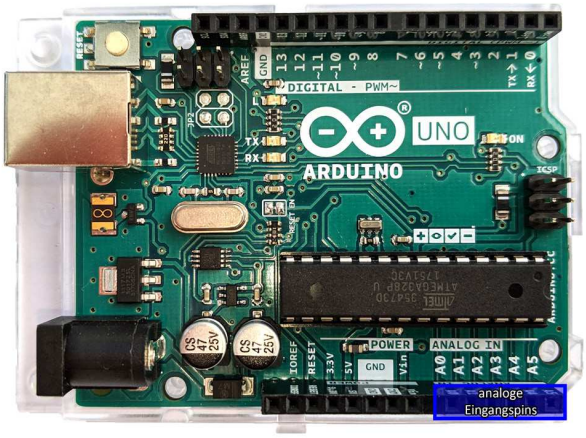


Abb. 5.8 Es stehen 6 analoge Eingangspins zur Verfügung: A0 bis A5

Neben den eben beschriebenen 14 digitalen Ein- und Ausgangspins bietet unser Arduino noch 6 analoge Eingänge, welche beliebige Spannungswerte zwischen 0 V und 5 V verarbeiten können. Ebenso wie die digitalen Eingänge weisen sie einen sehr hohen Eingangswiderstand auf, es fließt damit nahezu kein Strom. Das ist günstig, um empfindliche Signalgeber oder Sensoren nicht zu belasten.

Für das folgende Beispiel nutzen wir ein Potentiometer, also einen verstellbaren Widerstand, wie wir ihn bereits in Kapitel 3.1.2.1 kennengelernt haben. Wir setzen ihn als Spannungsteiler ein, um über einen Drehknopf eine beliebige Spannung zwischen 0 V und 5 V an den analogen Eingang zu bringen.

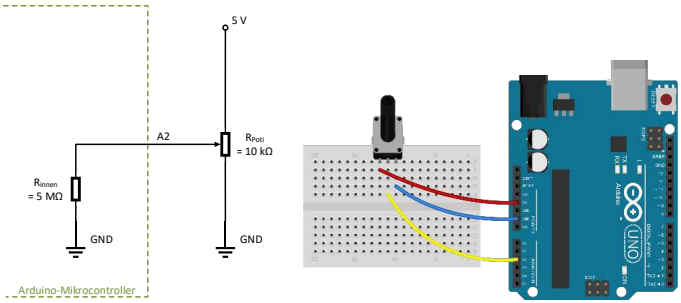


Abb. 5.9 Potentiometer als verstellbarer Widerstand

Das hier genutzte Modell hat einen Gesamtwiderstand von $10\,\text{k}\Omega$, für unser Beispiel sind jedoch auch andere Potis im Bereich von etwa $1\,\text{k}\Omega - 100\,\text{k}\Omega$ geeignet. Die Spannung am Mittelabgriff ergibt sich bekanntermaßen aus dem Verhältnis von R_1 zu R_2, also aus dem Drehwinkel des Potentiometers, und lässt sich daher immer in gleicher Weise einstellen.

Ein analoger Eingang arbeitet nun in ähnlicher Weise wie ein Spannungsmessgerät – er wandelt die anliegende Spannung in einen Zahlenwert um. Der in diesem Mikrocontroller integrierte Analog-Digital-Wandler hat eine Auflösung von 10 Bit, demnach kann die Ausgabe $2^{10} = 1024$ verschiedene Werte annehmen.

Spannung	Digitalwert
0 V	0
0,5 V	102
1 V	205
2 V	410
3 V	614
4 V	818
5 V	1023

Tabelle 5.2 Beispiele für durch Digitalwerte repräsentierte Eingangsspannungen

Die Spannung kann somit theoretisch auf etwa $\frac{5\,\text{V}}{1024} \approx 0{,}005\,\text{V}$ genau gemessen werden. Allerdings ist hierbei noch zu beachten, dass je nach Güte der verwendeten Bauelemente und des genutzten Verfahrens zur Analog-Digital-Wandlung gewissen Toleranzen in Kauf genommen werden müssen, so dass die tatsächliche Genauigkeit geringer ist. Beim Arduino-Mikrocontroller hängt die Umwandlungsskala zum Beispiel auch von der Betriebsspannung ab – ein Arduino, welcher aufgrund einer ungenauen Spannungsquelle an 5,3 V Betriebsspannung angeschlossen ist wird bei gleichem Eingangssignal einen geringfügig niedrigeren Zahlenwert zuordnen als ein anderer Arduino mit genau 5,0 V Betriebsspannung. Alternativ kann auch festgelegt werden, dass statt der Betriebsspannung eine andere Bezugsspannung (zwischen 0 und 5 Volt) genutzt werden soll, welche an den Pin AREF angelegt wird. Dazu muss im `setup()`-Teil des Sketches einmalig die Funktion `analogReference(EXTERNAL)` aufgerufen werden.

In unserem folgenden Beispiel nutzen wir die Standardeinstellung, also den Bezug auf die Betriebsspannung, denn wir wollen den Eingang nur verwenden, um die Position eines Drehreglers zu erfassen.

Nach dem Hochladen des Sketches können Sie die Blinkfrequenz der onboard-LED mit dem Drehregler einstellen. Zusätzlich wird am seriellen Monitor der aktuell ausgelesene Analogwert dargestellt.

```
1  void setup() {
2    pinMode(LED_BUILTIN, OUTPUT);
3    Serial.begin(9600);                                    // (1)
4  }
5
6  void loop() {
7    int Wartezeit = analogRead(2);                         // (2)
8    Serial.println(Wartezeit);                             // (3)
9    digitalWrite(LED_BUILTIN, HIGH);
10   delay(Wartezeit);                                      // (4)
11   digitalWrite(LED_BUILTIN, LOW);
12   delay(Wartezeit);
13 }
```

1. Die Funktionen zur seriellen Ausgabe kennen Sie bereits aus Kapitel 4.8. Wir nutzen sie hier nur, um die Funktion des analogen Einganges durch die Anzeige des Auslesewertes besser nachvollziehen zu können.
2. In gleicher Weise wie im vorigen Beispiel mit `digitalRead()` nutzen wir nun die Funktion `analogRead()`, um das Eingangssignal auszulesen. Diesmal ist der Rückgabewert allerdings kein boolescher `true`/`false`-Wert, sondern eine ganzzahlige Variable mit mindestens 10 Bit. Somit ist `int` (16 Bit) ein geeigneter Variablentyp. Da hier, im Gegensatz zum Beispiel mit den digitalen Pins, nicht vorher per `pinMode()` explizit eine Betriebsart zugewiesen wurde, verhält sich der Pin (wie erwartet) als Eingang.
3. Um die Wirkungsweise in diesem Beispiel besser nachvollziehen zu können, geben wir den ausgelesenen Zahlenwert auf den seriellen Monitor aus.
4. Die Pausenzeit zwischen den Ein- und Ausschaltbefehlen der onboard-LED wird nun direkt durch den zuvor ausgelesenen Analogwert bestimmt und liegt demnach zwischen 0 und 1023 Millisekunden.

Sicherlich haben Sie auch bemerkt, dass der angezeigte Wert am seriellen Monitor geringfügig schwanken kann, obwohl Sie den Regler gar nicht berühren. Diese Schwankungen resultieren aus den bereits genannten Toleranzen.

Übrigens: Es ist möglich, die analogen Eingänge über den Befehl `pinMode(Pinnummer, OUTPUT)` auch als digitale Ausgänge zu konfigurieren und entsprechend anzusteuern. Verwenden Sie dazu die Pinnummern 14 (für A0) bis 19 (für A5). Alternativ können Sie auch einfach die Bezeichner „`A0`“ bis „`A5`“ verwenden, der Präprozessor ersetzt sie entsprechend. Des Weiteren können Sie mit der Funktion `digitalRead()` ebenso auf die Pins (im Eingangsmodus) zugreifen, dann wird der eingelesene Spannungswert binär interpretiert – genau wie bei digitalen Eingangspins. Verstehen Sie diese Kontakte also einfach als weitere digitale Ein- und Ausgabepins, welche sich standard-

mäßig im INPUT-Modus befinden und um die Zusatzfunktion der analogen Einlesemöglichkeit per `analogRead()` erweitert wurden.

5.4 Pulsweitenmodulation

Das binäre Verhalten der digitalen Ausgänge kann ein Nachteil sein, wenn man einen Zwischenwert als Ausgangssignal benötigt – zum Beispiel, weil man eine LED dimmen möchte, oder einen Gleichspannungsmotor eines Lüfters mit variabler Drehzahl ansteuern will. In der Elektronik greift man in einem solchen Fall zu einem Trick namens Pulsweitenmodulation[3]. Wir hatten bereits festgestellt, dass unsere Ausgänge lediglich zwischen 5 V und 0 V hin- und herschalten können, jedoch keine Zwischenwerte beherrschen. Allerdings können diese Ausgänge sehr schnell hin- und herschalten. Um also eine LED mit verminderter Helligkeit leuchten zu lassen, genügt es auch, sie rasch ein- und auszuschalten. Geschieht dies schneller als etwa 100-mal pro Sekunde (der Arduino nutzt sogar eine Frequenz von knapp 1.000 Hertz, also 1.000-mal pro Sekunde), kann unser Auge dieses Flackern nicht mehr wahrnehmen und deutet es stattdessen als konstante Lichtquelle. Verändert man das Verhältnis der Einschaltdauer zur Gesamtdauer des Zyklus (das sogenannte Tastverhältnis $\frac{t_1}{T}$), kann man damit die wahrgenommene Helligkeit beeinflussen.

3 Dies ist der in Deutschland übliche Begriff. Ausgehend von der Herkunft vom englischen *„Pulse Width Modulation“* wäre allerdings die Übersetzung „Puls*breiten*modulation“ sprachlich treffender, weshalb diese auch mehr und mehr Verwendung findet – mit der entsprechenden Abkürzung *PBM*. Gemeint ist immer dasselbe Verfahren.

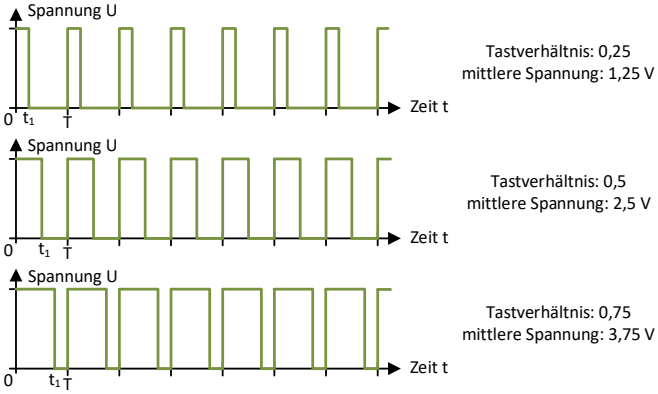


Abb. 5.10 Beispiel für ein pulsweitenmoduliertes Signal mit verschiedenen Tastverhältnissen

Einige der digitalen Ausgänge unseres Arduinos beherrschen dieses sogenannte Pulsweitenmodulationsverfahren. Konkret sind es die Pins 3, 5, 6, 9, 10 und 11. Sie sind auf der Platine durch eine Tilde (~) neben der Zahl gekennzeichnet. Im folgenden Beispiel machen wir uns dies zu Nutze. Der Aufbau entspricht exakt dem Beispiel aus Kapitel 5.1:

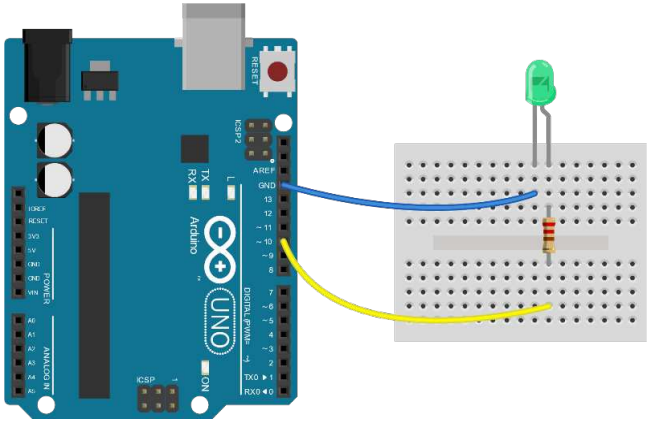


Abb. 5.11 Die Tilde (~) kennzeichnet Ausgänge, welche für das Pulsweitenmodulationsverfahren vorgesehen sind

Die Ansteuerung der Ausganspins mit PWM ist einfach – statt der Funktion `digitalWrite()` verwendet man nun `analogWrite()` und übergibt als zweites Argument einen Zahlenwert. Entsprechend dieser Zahl wird das Ausgangssignal im entsprechenden Verhältnis gepulst, wobei `0` für *immer aus* und `255` für *immer an* steht.

```
1  void setup() {
2    pinMode(10, OUTPUT);                                          // (1)
3  }
4
5  void loop() {
6    for(int Helligkeit = 0; Helligkeit <= 255; Helligkeit++) // (2)
7    {
8      analogWrite(10, Helligkeit);                                // (3)
9      delay(5);                                                   // (4)
10   }
11   for(int Helligkeit = 255; Helligkeit >= 0; Helligkeit--) // (5)
12   {
13     analogWrite(10, Helligkeit);
14     delay(5);
15   }
16 }
```

Im Gegensatz zu den abrupten Lichtsignalen des Blink-Beispiels lässt dieser Sketch wiederholt die angeschlossene LED langsam aufleuchten und danach wieder langsam verlöschen.

1. Wie bereits im Beispiel zu den digitalen Ausgängen in Kapitel 5.1 verwenden wir wieder Pin 10. Er ist auf dem Board mit einer Tilde markiert und unterstützt also die Ausgabe per Pulsweitenmodulation.

2. `for`-Schleifen haben wir bereits kennengelernt. Hier wird die Schleife dazu genutzt, die Variable `Helligkeit` von 0 bis 255 kontinuierlich hochzuzählen.

3. Die Funktion `analogWrite()` dient der Ausgabe eines besagten Pulsweitemodulationssignals. Sie erwartet als erstes Argument den betreffenden Pin und als zweites Argument einen 8-Bit-Wert, also eine Zahl zwischen 0 und 255. Sie erzeugt am betreffenden Ausgangspin ein Signal im entsprechenden Tastverhältnis. Dieses Signal wird so lange dort beibehalten, bis die Funktion `analogWrite()`

erneut aufgerufen wird und dem entsprechenden Pin einen neuen Ausgabewert zuweist.

4. Da die Schleife insgesamt 256-mal durchlaufen wird, bevor die LED ihre volle Helligkeit erreicht, genügt eine relativ kurze Pause von nur 5 ms. Falls Sie sich wundern, warum man diese verschwindend kurze Pause nicht einfach ganz weglässt: Dann würde die Schleife so schnell durchlaufen, dass man die Helligkeitsänderungen an der LED gar nicht mehr wahrnehmen kann – probieren Sie es gern aus.

5. Die zweite `for`-Schleife zählt nun rückwärts von 255 auf 0, um die `Helligkeit` wieder zu reduzieren, ansonsten ist sie identisch mit der ersten Schleife.

An diesem Beispiel sehen Sie, wie einfach es ist, mit Hilfe der PWM-Ausgänge ein „gleitend regelbares" Ausgangssignal zu erzeugen. In gleicher Weise könnte man zum Beispiel auch einen Lüfter steuern, selbst wenn er aufgrund seiner erhöhten Stromaufnahme über einen Transistor angesteuert wird (wie auch das Relais im Kapitel 3.1.2.6). Die mechanische Trägheit des Rotors sorgt dann dafür, dass die schnellen Schaltvorgänge nicht mehr bemerkbar sind und lediglich in einer entsprechend des Tastverhältnisses variablen Rotationsgeschwindigkeit resultieren.

Die Tatsache, dass nicht alle Ausgangspins diese Funktion unterstützen, liegt im internen Aufbau des Arduino-Mikrocontrollers begründet. Natürlich ist auch möglich, die anderen Pins in ähnlicher Weise zu benutzen, indem man durch ein entsprechendes Programm das Verhalten eines PWM-Pins nachbildet – so wie der Blink-Sketch, nur mit deutlich verkürzten Pausen. Allerdings ergibt sich dadurch der Nachteil, dass das Programm dann entweder ausschließlich damit beschäftigt ist, den Ausgang im korrekten Tastverhältnis ein- und auszuschalten, oder das Programm muss noch andere Befehle erledigen und das gewünschte PWM-Signal kommt dadurch ins Stocken. Aus diesem Grund ist es also immer zu empfehlen, die dafür vorgesehenen Pins zu verwenden, wenn eine Pulsweitenmodulation angestrebt wird. An ihnen wird das gepulste Signal unabhängig vom Programmablauf durch eine spezielle Schaltung erzeugt.

Des Weiteren gilt es zu beachten, dass das Ausgangssignal kein echtes Analogsignal ist – auch wenn die variable Helligkeit der LED aufgrund der Trägheit unserer Augen diesen Anschein erweckt. Dies kann in einigen Fällen zu unerwarteten Ergebnissen führen, zum Beispiel wenn das Signal auf einen analogen Eingang eines anderen Mikrocontrollers geleitet wird. Mit Hilfe eines Kondensators kann die gepulste Spannung jedoch geglättet werden:

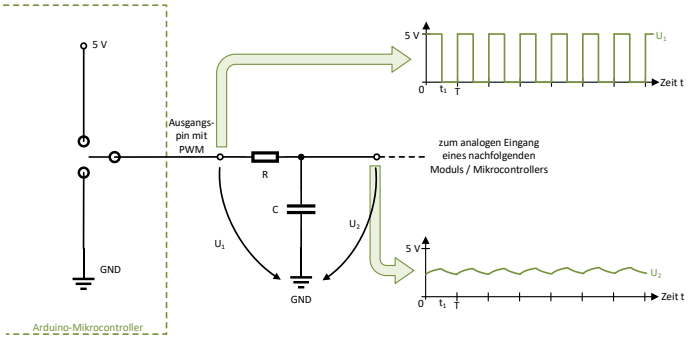


Abb. 5.12 Ein sogenanntes R-C-Glied (bestehend aus einem Widerstand und einem Kondensator) kann das pulsweitenmodulierte Signal in eine nahezu konstante Spannung umwandeln. Das obere Diagramm zeigt den zeitlichen Spannungsverlauf vor der Glättung, das untere danach.

Das gepulste Signal lädt den Kondensator abwechselnd auf und entlädt ihn wieder. Der Widerstand begrenzt dabei die Stromstärke, um den Ausgang des Mikrocontrollers nicht zu überlasten. Durch die relativ hohe Kapazität des Kondensators reicht die Zeit jedoch bei Weitem nicht, um ihn vollständig zu laden oder zu entladen. Daher stellt sich an ihm nach kurzer Zeit eine Spannung ein, die dem zeitlichen Mittelwert der gepulsten Spannung entspricht.

5.5 Kommunikationsschnittstellen

Die in den letzten Kapiteln betrachteten Ein- und Ausgangspins sind vielseitig und einfach einsetzbar, allerdings gestaltet sich die Übertra-

gung komplexer Daten (zum Beispiel Messwerte oder Anzeigetexte) damit relativ schwierig. Zu diesem Zweck ist der Arduino mit mehreren Schnittstellen ausgestattet, welche – ähnlich wie der USB-Anschluss bei einem PC – für den Datenaustausch mit anderen Komponenten wie Sensoren, Displays oder weiteren Mikroprozessoren konzipiert sind.

5.5.1 Serielle Schnittstelle

Eines der ältesten Verfahren zur Datenübertragung zwischen Rechnern ist die serielle Schnittstelle, sie wurde bereits in den 1960-er Jahren unter dem Begriff RS-232[4] standardisiert und über Jahrzehnte als Maus- oder Modemanschluss bei PCs verwendet, bevor sie um die Jahrtausendwende von USB verdrängt wurde. In Anlehnung an die Bezeichnung des verwendeten Schnittstellen-Moduls wird sie im englischsprachigen Raum auch als UART[5] bezeichnet.

Sie benötigt zur Datenübertragung nur jeweils einen Draht pro Richtung (sowie eine gemeinsame Masseverbindung). Die Bits werden nacheinander (seriell) als Spannungspegel übertragen, wobei eine negative Logik gewählt wurde. Das heißt, ein hoher Pegel (+3 V bis +15 V) repräsentiert eine 0, ein niedriger Pegel (-3 V bis -15 V) steht für eine 1. Je nach Wandlertyp werden aber auch abweichende Pegel toleriert, deshalb funktioniert sie in unserer Arduino-Welt auch mit 0 V (für eine 1) und 5 V (für eine 0).

4 *RS – recommended standard*, empfohlener Standard

5 UART – *Universal Asynchronous Receiver Transmitter* – universeller asynchroner Sender/Empfänger

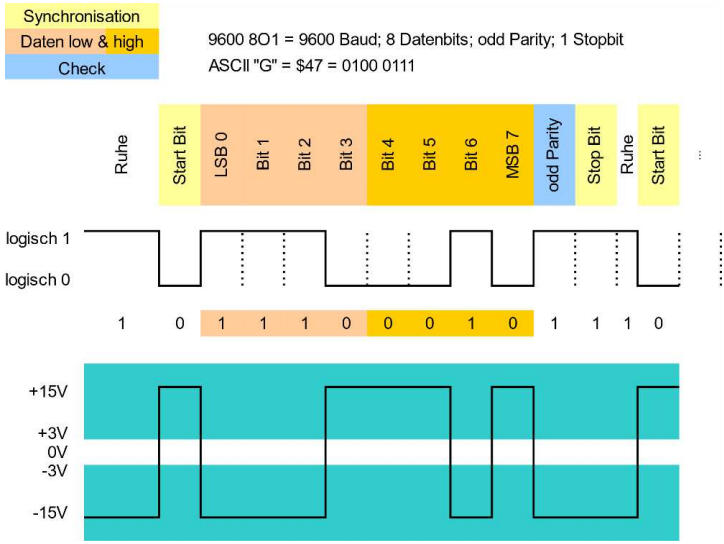


Abb. 5.13 Aufbau eines seriellen Datensignals (RS-232)

Die Übertragung erfolgt in Bitgruppen (sogenannte Wörter) Bit für Bit. Ein Wort besteht üblicherweise aus 8 Bit, es sind jedoch je nach Implementierung auch Abweichungen möglich. Zusätzlich wird der Beginn eines Bytes von einem sogenannten Startbit und das Ende von einem Paritätsbit (zur Fehlererkennung) und einem Stoppbit markiert. Dadurch kann der Empfänger erkennen, wann ein neues Byte beginnt. Allerdings muss er von vornherein auch die Übertragungsgeschwindigkeit kennen, da er nach dem Empfang des Startbits in einem festgelegten zeitlichen Abstand das Signal abtasten muss, um die einzelnen Bits zu erkennen. Erwartet der Empfänger eine andere Geschwindigkeit als die vom Sender verwendete, kann das Signal nicht verarbeitet werden.

Wir haben die serielle Schnittstelle bereits kennengelernt, als wir im Kapitel 4.8 die Datenausgabe auf dem seriellen Monitor nutzten – auch hier mussten wir sicherstellen, dass die eingestellte Baudrate (Geschwindigkeit) mit der im Arduino-Sketch gewählten Baudrate übereinstimmt. Übliche Datenraten reichen von 300 bis

115200 Baud[6], in Bastler-Anwendungen hat sich ein Wert von 9600 Baud bewährt.

Die Arduino IDE auf dem Computer nutzt die serielle Schnittstelle auch, um unsere Sketche auf den Arduino zu übertragen, nachdem der Compiler sie in Maschinencode übersetzt hat. Dazu ist auf dem Arduino-Board ein USB-zu-Seriell-Wandler angebracht, welcher die Signale in beide Richtungen übersetzt. Wenn kein PC an die USB-Buchse des Boards angeschlossen ist, können wir diesen Wandler jedoch ignorieren. Die Signale lassen sich auch an den digitalen Pins 0 (Senderichtung) und 1 (Empfangsrichtung) abgreifen, da sie sich deren Belegung teilen. Das bedeutet im Umkehrschluss auch, dass die digitalen Ein- und Ausgangspins 0 und 1 nicht genutzt werden können, wenn die serielle Schnittstelle im Programm verwendet wird. Siehe dazu auch *Abbildung 4.6* in Kapitel 4.8.

Vorteilhaft an der RS-232-Schnittstelle ist ihre historisch bedingte Kompatibilität mit fast allen Arten von Mikrocontrollern und Computern (gegebenenfalls über Wandler). Aufgrund des fehlenden Taktsignals ist sie jedoch besonders bei höheren Übertragungsgeschwindigkeiten oder langen Leitungswegen anfällig für Störungen und daher nicht besonders robust. Wir werden ihre Nutzung in weiteren Beispielen daher auf die bereits erprobte Datenausgabe auf dem Computer beschränken.

5.5.2 I²C

Aufgrund dieser Unzulänglichkeiten entwickelte Philips Anfang der 1980-er Jahre die I²C-Schnittstelle (sprich: „I-Quadrat-C“ oder englisch

6 Die Einheit Baud wird gewählt, da es sich hierbei genau genommen um eine Symbolrate handelt. Ein Symbol ist dabei ein „Zeichen“ auf der Leitung. Bei anderen Schnittstellen kann ein solches Symbol auch mehrere Bits übertragen. In unserem Zusammenhang entspricht es einem Bit pro Sekunde, wobei jedoch Start- und Stoppbits mitgezählt werden. Als Faustregel kann man in unserer Anwendung daher 10 Baud als 1 Byte pro Sekunde annehmen.

„I-square-C“, auch „I-two-C“; ursprünglich für „Inter-Integrated Circuit“) zum Datenaustausch zwischen verschiedenen Komponenten eines Gerätes. Im Gegensatz zur seriellen Verbindung, welche stets nur zwei Kommunikationspartner verbindet, handelt es sich hierbei um einen Datenbus, welcher optional deutlich mehr Komponenten (ursprünglich bis zu 112, in neueren Versionen nochmals deutlich mehr) miteinander verbinden kann.

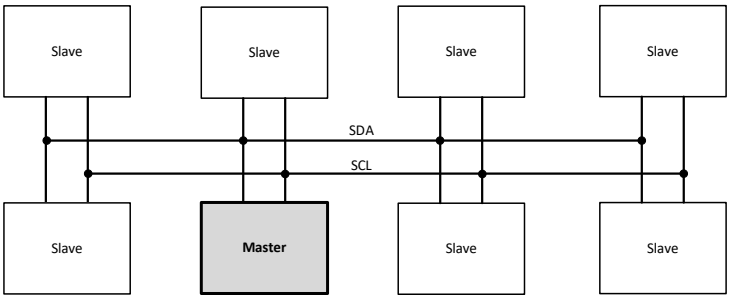


Abb. 5.14 I^2C-Bus mit 8 Teilnehmern

Elektrisch gesehen besteht der Bus aus lediglich zwei Leitungen, an die alle Busteilnehmer angeschlossen sind, sowie einem gemeinsamen Massebezug. Eine der Leitungen dient als Taktsignal (*Clock* - CLK), die andere wird zum Datenaustausch (*Serial Data* – SDA) verwendet.

Dabei fungiert stets ein Teilnehmer als Master, er generiert das Taktsignal und koordiniert die Kommunikation – etwa so wie ein Moderator in einer Gesprächsrunde. Alle anderen Komponenten agieren als Slaves, sie ordnen sich also dem Master unter. Jeder Slave benötigt eine eindeutige Adresse (in Form einer Zahl), über die er vom Master angesprochen werden kann. Der Master kann den Slaves jederzeit Daten senden. Die einzelnen Slaves dürfen jedoch nur senden, wenn sie vom Master abgefragt werden, also quasi eine Sendeerlaubnis erhalten. Dadurch wird vermieden, dass mehrere Komponenten gleichzeitig auf die einzige vorhandene Datenleitung senden, denn dann wäre das Signal gestört.

Üblicherweise agiert unser Arduino natürlich als Master und kommuniziert mit Komponenten wie Sensoren oder Displays als Slaves. Aller-

dings ist es auch möglich, über I²C Verbindungen zwischen mehreren Arduinos herzustellen. Dies nutzen wir auch in folgendem Beispiel: Ein Arduino fungiert als Master und sendet die von einem analogen Eingang gemessene Position eines Potentiometers per I²C-Bus an einen zweiten Arduino, welcher als Slave mit diesem Wert die Helligkeit einer angeschlossenen LED steuert.

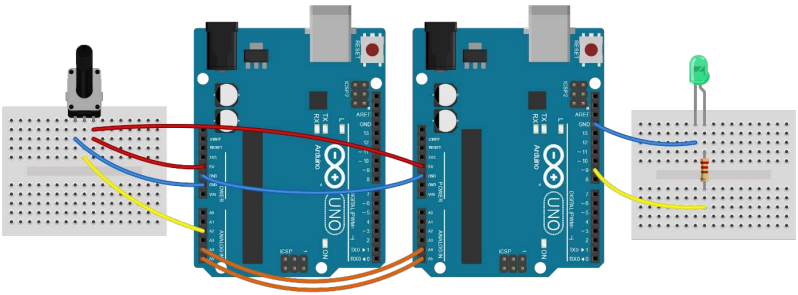


Abb. 5.15 Zwei Arduinos, verbunden per I²C – links der Master, rechts der Slave. Werden beide per USB mit Strom versorgt, entfällt die rote 5V-Verbindung zwischen den Arduinos. Die GND-Verbindung ist als Massebezug für die Signalübertragung immer notwendig.

Wie auch schon zuvor beim seriellen Port, hat unser ATmega328P-Mikrocontroller keine separaten Pins für die I²C-Schnittstelle – stattdessen werden nun die analogen Pins A4 (SDA) und A5 (SCL) verwendet. Dies bedeutet wiederum, dass die beiden Anschlüsse nicht mehr als analoge Eingänge zur Verfügung stehen, solange der Arduino die I²C-Schnittstelle nutzt. Zusätzlich sind beide Signale auch auf der gegenüberliegenden Seite der Platine nahe dem Pin 13 abgreifbar.

Nun laden wir folgenden Sketch auf den linken Arduino:

```
1  #include "Wire.h"                                          // (1)
2
3  void setup() {
4    Wire.begin();                                            // (2)
5  }
6
7  int Drehregler = 0;                                        // (3)
8
```

```
 9  void loop() {
10    Drehregler = analogRead(2);                          // (4)
11    Drehregler /= 4;                                     // (5)
12    Wire.beginTransmission(8);                           // (6)
13    Wire.write(Drehregler);                              // (7)
14    Wire.endTransmission();                              // (8)
15
16    delay(500);
17  }
```

1. Der `include`-Befehl ist für uns neu. An der vorangestellten Raute erkennen wir, dass er sich an den Präprozessor richtet. Er soll an dieser Stelle den Programmcode aus der Datei „Wire.h“ einfügen, bevor der Sketch an den Compiler übergeben wird. Es handelt sich dabei um eine Bibliothek, welche uns die Funktionen zur Verfügung stellt, die wir für die Kommunikation per I^2C brauchen. Der Inhalt dieser Datei sind also Funktionsdefinitionen, wie wir sie selbst bereits in Kapitel 4.11 kennengelernt haben. Bibliotheken werden uns in kommenden Beispielen noch häufig das Programmieren erleichtern, indem sie Funktionen für oft gebrauchte, aber komplexe Aufgaben (wie beispielsweise die Kommunikation mit anderen Komponenten) bereitstellen. Würde man auf sie verzichten, müsste man im hiesigen Fall zum Beispiel auch die komplette Steuerung des I^2C-Busses selbst im Programm regeln – also das Taktsignal erzeugen, die Daten bitweise zerlegen und seriell ausgeben, die korrekte Übertragung prüfen usw. – es wird schnell ersichtlich, dass es sinnvoll ist, dafür auf vorgefertigten und von Profis optimierten Code zurückzugreifen. Die Datei Wire.h[7] ist dabei eine Bibliothek, die bereits bei der Installation der Arduino IDE mitgeliefert wird. Es ist aber auch möglich, weitere Bibliotheken für spezielle Anwendungen aus dem Internet zu laden. Im Kapitel 5.5.4 wird dies demonstriert.
2. Der Bibliothek stellt das Objekt `Wire` zur Verfügung, dem alle I^2C-betreffenden Funktionen zugeordnet sind. Dies wurde zur besseren Übersicht von den Autoren so gewählt. Die Funktion

7 Der Name rührt daher, dass der I^2C-Bus im englisch umgangssprachlich auch als „Two-Wire“, also „Zweidraht“, bezeichnet wird.

`Wire.begin()` wird einmalig in der `setup()`-Routine aufgerufen und startet damit im Arduino integrierte I²C-Schnittstelle. Wird ihr, wie in obigem Fall, kein Argument übergeben, fungiert der Arduino als Master.

3. Wir definieren eine `int`-Variable, um den vom Poti ausgelesenen Wert zwischenzuspeichern.

4. Das Einlesen des Signals vom analogen Eingang haben wir bereits kennengelernt. Allerdings müssen wir daran denken, dass dieser Wert mit 10 Bit dargestellt wird, also zwischen 0 und 1023 liegen kann.

5. Da wir per I²C aber nur byteweise (also 8 Bit) übertragen können und die PWM-Ausgabe später auch nur 8 Bit unterstützt, teilen wir den Wert einfach ganzzahlig durch 4. So erhalten wir wie gewünscht einen Wertebereich von 0 bis 255.

6. Die Funktion `Wire.beginTransmission()` leitet den Beginn einer Datenübertragung ein. In Klammern ist die Adresse des Empfängers (Slave) anzugeben. Im Normalfall dürfen Slaves Adressen zwischen (einschließlich) 8 und 119 haben, es stehen also 112 Möglichkeiten zur Verfügung. Hier wurde willkürlich die 8 gewählt.

7. Mittels `Wire.write()` kommen nun die eigentlichen Nutzdaten ins Spiel. Hierbei muss man beachten, dass die Funktion standardmäßig nur ein Byte überträgt. Übergibt man ihr Variablen, die mehr Speicherplatz belegen, so werden nur die 8 niedrigwertigsten Bit übertragen. Möchte man mehr Daten übertragen, kann man sie aufteilen, in ein `byte`-Array schreiben und dieses übergeben. Auch in unserem Fall ist die Variable `Drehregler` vom Typ `int` (welcher zwei Byte belegt) eigentlich zu groß. Da wir in (5) den Wertebereich aber auf Zahlen bis maximal 255 eingegrenzt haben, reicht es aus, die 8 niedrigwertigsten Bit zu übertragen.

8. `Wire.endTransmission()` leitet letztlich die eigentliche Übertragung der Daten aus dem Sendespeicher ein und beendet danach die Verbindung.

Obiger Sketch liest also zweimal pro Sekunde die aktuelle Stellung des Potentiometers am analogen Eingang A2 ein und sendet den Wert per I²C an den Slave, dessen Sketch wir im Folgenden betrachten:

```
1  #include "Wire.h"
2
3  volatile byte Empfangswert;                                  // (1)
4
5  void setup() {
6    Wire.begin(8);                                             // (2)
7    Wire.onReceive(Datenempfang);                              // (3)
8    pinMode(9, OUTPUT);
9  }
10
11 void loop() {
12   analogWrite(9, Empfangswert);
13 }
14
15 void Datenempfang(int Anzahl) {                              // (4)
16   if(Wire.available())                                       // (5)
17     Empfangswert = Wire.read();                              // (6)
18 }
```

Hier begegnen wir zum ersten Mal einer weiteren Besonderheit in der Programmierung: Der Empfang von I²C-Daten wird über sogenannte Interrupts realisiert. Diese wörtlichen Unterbrechungen sind in der Hardware des Mikrocontrollers verankert und sorgen dafür, dass bestimmte Aktionen (in unserem Fall der Empfang von I²C-Daten vom Master) den regulären Programmablauf unterbrechen dürfen. So als wenn bei Ihnen der Postbote an der Tür klingelt und sie kurz alles stehen und liegen lassen, um rasch die Post entgegenzunehmen.

Schauen wir auf die Befehle im Einzelnen:

1. Zur Definition der Variable `Empfangswert` verwenden wir hier zusätzlich das Schlüsselwort `volatile`. Es signalisiert der Compiler, dass sich diese Variable auch außerhalb des normalen Programmablaufs ändern kann (durch den bereits erwähnten Interrupt). Dadurch unterlässt der Compiler bestimmte Optimierungen im

Programmcode, welche ansonsten zu unerwarteten Ergebnissen führen könnten.

2. Da dieser Arduino nun als Slave agieren soll, übergeben wir der Funktion `Wire.begin()` nun einen Wert, und zwar die gewünschte Slave-Adresse.

3. Mittels der Funktion `Wire.onReceive()` wird festgelegt, was passieren soll, wenn Daten per I^2C empfangen werden. Ihr muss der Name einer Funktion übergeben werden, welche sofort aufgerufen wird, sobald Daten empfangen wurden. Solch eine Funktion nennt man auch *Interrupt Service Routine* (ISR), da sie sich um den Grund der Programmunterbrechung (Interrupt), kümmert. Bildlich gesprochen erklären Sie dem Mikrocontroller damit, was er tun soll, wenn es an der Tür klingelt.

4. Hier wird die besagte ISR nun deklariert. Derartige Funktionen haben niemals einen Rückgabewert (`void`). Von der Wire.h-Bibliothek ist in diesem Fall zusätzlich vorgegeben, dass sie als Argument eine `int`-Variable bekommt, welche die Anzahl der empfangenen Byte repräsentiert. Ob wir diese Information innerhalb der Funktion auswerten, ist uns überlassen. ISR sollten stets möglichst kurz gehalten werden, da sie ja den eigentlichen Programmablauf „stören". Außerdem reagiert der Mikrocontroller während der Ausführung einer ISR nicht auf mögliche andere Interrupts. Wir beschränken uns daher auf die simple Entgegennahme des empfangenen Helligkeitswertes.

5. Es hat sich eingebürgert, innerhalb derartiger Routinen noch einmal kurz zu prüfen, ob tatsächlich Daten empfangen wurden. Die Funktion `Wire.available()` gibt in diesem Fall `true` zurück. Es wäre ja auch denkbar, dass die Funktion `Datenempfang()` versehentlich von einer anderen Programmstelle aus aufgerufen wurde – ohne diese Prüfung könnte dies zu Problemen führen.

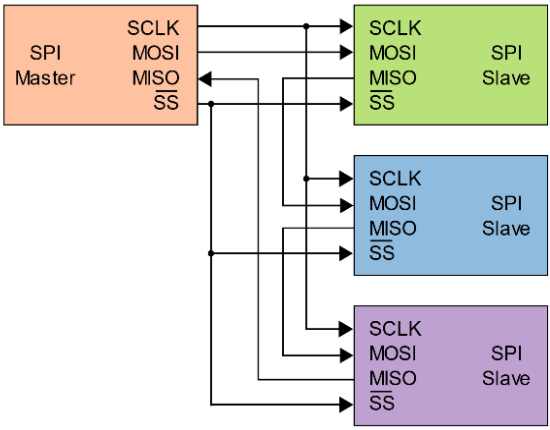
6. Wir erhalten das empfangene Byte als Rückgabewert der Funktion `Wire.read()` und weisen es der Variable `Eingabewert` zu. Natürlich könnten wir es auch gleich noch per `analogWrite()` ausgeben, aber wir wollen die ISR ja möglichst kurz halten und insbe-

sondere Funktionsaufrufe innerhalb einer ISR vermeiden. Daher nehmen wir die Ausgabe lieber in der normalen `loop()`-Hauptschleife vor.

Der Vorteil der Nutzung in Interrupts ist in diesem Fall, dass der Master nicht warten muss, bis der Slave an einer Programmstelle angekommen ist, an der er die Daten entgegennehmen kann. Kommuniziert ein Master mit mehreren Dutzend Slaves, könnte dies ansonsten die Kommunikation erheblich behindern.

5.5.3 SPI

Eine weitere Schnittstelle zum Datenaustausch auf Mikrocontroller-Ebene ist das 1987 entwickelte *Serial Peripherial Interface* (SPI). Auch hier gibt es wieder einen Master sowie einen oder mehrere Slaves. Der Master erzeugt wieder ein Taktsignal (*Serial Clock* – SCL), welches die komplette Datenübertragung synchronisiert. Im Gegensatz zum I^2C-Bus erfolgt der Datenempfang und das Versenden von Daten nun aber auf zwei getrennten Leitungen: MOSI (*Master Out, Slave In* – vom Master zum Slave) und MISO (*Master In, Slave Out* – die umgekehrte Richtung).



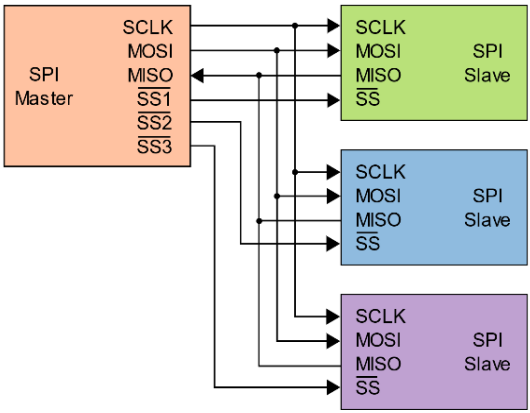


Abb. 5.16 Die Slaves können entweder kaskadiert (oben) oder parallelgeschaltet werden (unten), dann ist jedoch für jeden Slave ein separater Adresspin am Master erforderlich.

Des Weiteren haben die Slaves keine vordefinierte Adresse mehr, sondern werden entweder über separate Adresspins (*Slave Select* – SS) angesprochen oder (in selteneren Fällen) kaskadiert. Je nach Anschlussart muss das Programm entsprechend angepasst werden, da unter Nutzung der Adresspins jeweils nur der aktuell aktivgeschaltete Slave an der Kommunikation teilnimmt, während alle anderen die Signale ignorieren. Bei der Kaskadierung sind hingegen immer alle Slaves aktiv und reichen die Daten durch. Der Oberstrich an der Bezeichnung SS im Diagramm weist dabei auf die negative Logik hin. Das bedeutet, dass ein hoher Spannungspegel hierbei für *inaktiv* steht und ein niedriger Spannungspegel hingegen den Slave aktiviert.[8]

8 Derartige Inversionen finden sich in der Digitaltechnik an vielen Stellen. Die Ursache liegt im Aufbau der Komponenten, welche sich aus sogenannten Gatterschaltungen zusammensetzen. Diese Transistorschaltungen realisieren boolesche Funktionen (Und-Verknüpfung, Oder-Verknüpfung, Negation usw.) mit konkreten binären (Spannungs-)Signalen. In manchen Fällen ist diese Transistorschal-

Auch für die Verdrahtung per SPI werden wieder bereits vorhandene Pins benutzt. Auf dem Board sind diese sogar doppelt vorhanden, da der ICSP-Anschluss (zum Aufspielen eines neuen Bootloaders auf den Mikrocontroller) ebenfalls diese Schnittstelle verwendet.

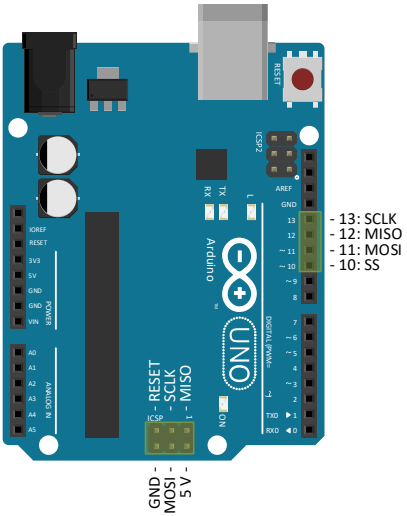


Abb. 5.17 Die für SPI benötigten Pins sind auf dem Board gleich doppelt vorhanden

Wir können unser Beispiel aus Kapitel 5.5.2 nun also auch per SPI statt I^2C realisieren:

---

tung in negativer Logik weniger aufwändig als in positiver ("normaler") Logik. Dies haben Sie auch bereits bei unserem Beispiel im Kapitel 5.2 gesehen: Der Druck auf den Taster erzeugte einen `LOW`-Pegel am Eingang, auch hier handelt es sich also um inverse Logik. Des Weiteren findet man die negative Logik gelegentlich in Bus-Systemen, bei denen ein Signal durch einen Pull-Up-Widerstand auf `HIGH` (0) gehalten und jeder Busteilnehmer das Signal (z.B. durch einen Transistor) aktiv auf LOW (1) „ziehen" kann. Ein Beispiel dafür ist der in der Industrie häufig verwendete CAN-Bus.

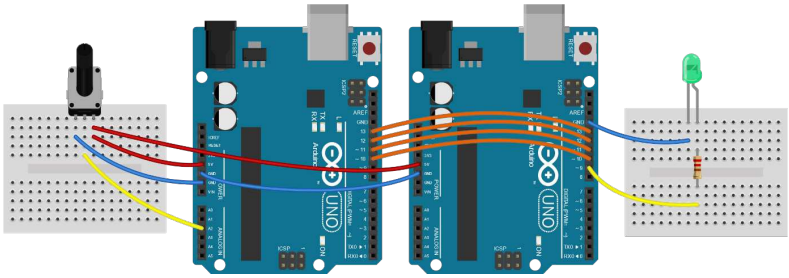


Abb. 5.18 Das vorangegangene Beispiel lässt sich identisch auch per SPI realisieren – einzig die orangen Verbindungen zwischen den Arduinos ändern sich dadurch. Werden beide Arduinos per USB mit Strom versorgt, entfällt die rote 5V-Verbindung zwischen den Platinen. Die GND-Verbindung ist als Massebezug für die Signalübertragung immer notwendig.

Die Funktionalität ist identisch, durch Drehen am Potentiometer lässt sich die Helligkeit der LED regeln. Natürlich hat sich durch die veränderte Schnittstelle auch der Programmcode geändert:

```
1  #include "SPI.h"                                          // (1)
2
3  void setup() {
4    SPI.begin();                                           // (2)
5    digitalWrite(SS, HIGH);                                // (3)
6  }
7
8  int Drehregler = 0;
9
10 void loop() {
11   Drehregler = analogRead(2);
12   Drehregler /= 4;
13   digitalWrite(SS, LOW);                                 // (4)
14   SPI.transfer(Drehregler);                              // (5)
15   digitalWrite(SS, HIGH);                                // (6)
16   delay(500);
17 }
```

1. In gleicher Weise wie bei I^2C gibt es auch hier eine Bibliothek, die uns wichtige Funktionen zur Verfügung stellt: Sie heisst SPI.h

2. Auch hier gibt es wieder eine Funktion, welche die Schnittstelle betriebsbereit macht: `SPI.begin()`, sie benötigt kein Argument.
3. Wir setzen die Slave-Select-Leitung zunächst auf `HIGH`, so dass der Slave (aufgrund der invertierten Logik) inaktiv ist, bis unser Programm etwas senden wird. Falls Sie sich wundern, warum nicht vorher die Konstante `SS` auf den Wert 10 definiert wurde und wieso dieser Pin 10 nicht erst per `pinMode()` in den Output-Modus geschaltet wurde: Die Konstantendefinition erfolgte bereits innerhalb der SPI.h-Bibiliothek und um den Output-Modus hat sich die Funktion `SPI.begin()` schon gekümmert.
4. Durch senken des Signals auf der Slave-Select-Leitung wird der Slave empfangsbereit geschaltet.
5. Die Funktion `SPI.transfer()` überträgt nun das ihr übergebene Datenbyte sofort an den Slave.
6. Das Empfangsmodul des Slaves wird wieder inaktiv gesetzt und kann nun mit der Verarbeitung beginnen.

Auch der Slave benötigt nun natürlich einen anderen Programmcode. Die SPI.h-Bibliothek ist hauptsächlich darauf ausgelegt, den Arduino als Master zu nutzen, sie stellt für Slave-Anwendungen kaum Funktionen bereit. Das Niveau des folgenden Sketches geht daher über das eines Einsteigers hinaus, also verzweifeln Sie bitte nicht, wenn Sie ihn nicht auf Anhieb nachvollziehen können. Die weiteren Beispiele werden wieder deutlich einfacher. Zur Vollständigkeit sei er dennoch hier aufgeführt.

```
1  #include "SPI.h"
2
3  volatile byte Empfangswert;
4
5  void setup() {
6    pinMode(9, OUTPUT);
7    SPCR |= bit(SPE);                                    // (1)
8    pinMode(MISO, OUTPUT);                               // (2)
9    SPI.attachInterrupt();                               // (3)
10 }
11
```

```
12 void loop() {
13   analogWrite(9, Empfangswert);
14 }
15
16 ISR(SPI_STC_vect)                                    // (4)
17 {
18   Empfangswert = SPDR;                               // (5)
19 }
```

1. Hier wird im *SPI-Control-Register* (ein 8-Bit-Speicherort, bei dem jedes Bit quasi ein Schalter für eine gewisse Funktionalität ist) der SPI-Modus als Slave aktiviert. Dies geschieht, indem jedes einzelne Bit dieses Registers mit einem „Schaltmuster" ODER-verknüpft wird. Dieses Schaltmuster (`SPE` – „SPI enable") hat die Form `01000000`. Die Verknüpfung bewirkt also, dass das zweite Bit in diesem Register eingeschaltet wird und alle anderen unverändert bleiben.
2. Der zum Master gerichtete Datenpin (Master In Slave Out, Pin 10) wird als Ausgang definiert. Da aber in unserem Beispiel ohnehin nichts vom Slave zum Master gesendet wird, könnten Sie diese Verbindung sogar auftrennen.
3. Es wird eine Interrupt-Service-Routine definiert. Dabei wird kein Name übergeben, da die SPI.h-Bibliothek diesen vorgibt.
4. Hier wird die ISR deklariert. Die etwas befremdliche Schreibweise ohne Typdefinitionen liegt darin begründet, dass `ISR()` die Standard-ISR der Programmiersprache C++ ist und die Bibliothek SPI.h keine andere akzeptiert.
5. Die Präprozessor-Konstante `SPDR` verweist auf die Speicheradresse des SPI-Empfangsregisters, dessen Inhalt wird der Variable Empfangswert zugewiesen.

Es ist offensichtlich, dass die Verwendung von SPI eher unvorteilhaft ist, wenn man Arduinos als Slaves einsetzen möchte. Bleibt der Arduino jedoch (wie in den meisten Anwendungsfällen) Master, so entscheidet man schlicht anhand der vorhandenen Schnittstellen der

anderen Komponenten, welche Variante man nutzt. Einige Sensoren und Displays verfügen zum Beispiel nur über I²C-Anschlüsse, manche RFID-Empfänger und Speicherbausteine können wiederum nur per SPI gesteuert werden. Da die Pinbelegung sich nicht überschneidet, kann ein Arduino auch beide Schnittstellen innerhalb eines Programms bedienen.

Es sollte noch erwähnt werden, dass sowohl bei SPI als auch bei I²C die Leitungslänge auf wenige Meter beschränkt ist. Zwar sind die Daten aufgrund des vorhandenen Taktsignals deutlich besser für den Empfänger zu rekonstruieren als beim seriellen Signal, aber dennoch können Störungen durch unvermeidliche äußere elektrische und magnetische Wechselfelder den Datenfluss behindern und zu falschen Ergebnissen führen. Diese Gefahr wächst mit der Leitungslänge. Da keines der beiden Übertragungssysteme über eine integrierte Fehlerkorrektur verfügt, ist es Aufgabe des Programmierers, auf der Empfängerseite die Plausibilität der empfangenen Daten zu prüfen und gegebenenfalls für eine entsprechende Fehlerkorrektur zu sorgen.

5.5.4 Drahtlose Signalübertragung (433 MHz)

Für die drahtlose Datenübertragung gibt es verschiedene Möglichkeiten. Die Nutzung von WLAN erscheint sicher jedem naheliegend, darauf werden wir in Kapitel 15.2.2 auch noch eingehen. Beschränkt sich die Übertragung jedoch auf simple Steuersignale oder Messwerte und ist Abhörsicherheit nicht erforderlich (weil es sich beispielsweise lediglich um die Daten eines Temperatursensors im Garten handelt), gibt es deutlich preiswertere und simplere Verfahren. Am gebräuchlichsten sind sogenannte 433-MHz-Module, welche meist als Set aus Sender und Empfänger vertrieben werden.

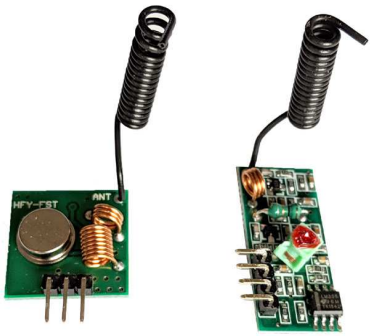


Abb. 5.19 433-MHz-Sendemodul (links) und -Empfangsmodul (rechts), jeweils mit Antenne

Die Funkverbindung nutzt das lizenz- und genehmigungsfreie Frequenzband von 433,05 MHz bis 434,79 MHz. Aufgrund der geringen Sendeleistung von maximal 10 Milliwatt darf jeder Nutzer in Deutschland diese Module betreiben, ohne dass dafür eine Lizenz oder Erlaubnis erforderlich wäre, es handelt sich um sogenannten *Jedermannfunk*. Die Reichweite liegt offiziell bei mehreren hundert Metern, allerdings ist diese neben anderen Faktoren auch von der Betriebsspannung des Sendemoduls (maximal 12 V) abhängig. Bei den hier stets verwendeten 5 V konnten durch den Autor im Test bei Abständen bis zu 20 Meter brauchbare Ergebnisse erzielt werden. Mitunter werden diese Module ohne Antenne vertrieben – man kann am entsprechenden Lötpin selbst eine Antenne ergänzen, indem man einen 17,3 Zentimeter langen Draht anlötet. Diese Länge entspricht einem Viertel der Funkwellenlänge und ist damit sehr günstig für die Übertragung. Ohne angelötete Antenne erreichen die Module Reichweiten im Bereich von einem Meter.

Für unser Beispiel verwenden wir den Versuch aus den vorherigen Kapiteln, nun soll aber gar keine Drahtverbindung mehr zwischen den Modulen bestehen. Der nachfolgende Versuchsaufbau erfordert daher, dass beide Arduino-Platinen mit Strom versorgt werden. Ist dies nicht

möglich, können auch einfach wieder zwei Drähte zur Verbindung der 5V- und GND-Anschlüsse verwendet werden. Zur genaueren Untersuchung wollen wir die Daten auf der Empfängerseite zusätzlich auf dem seriellen Monitor ausgeben.

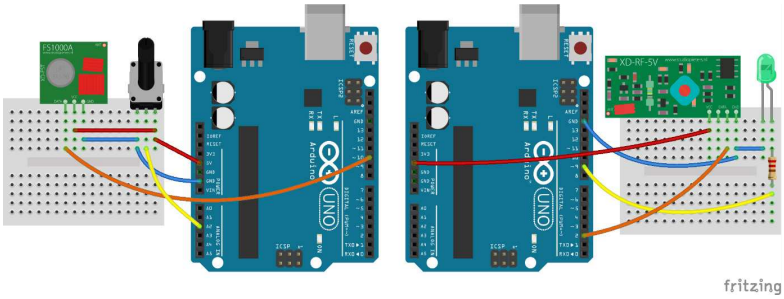
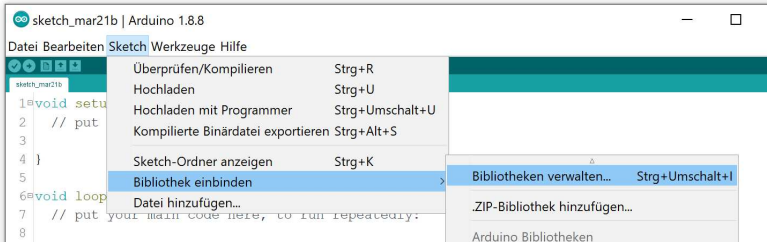


Abb. 5.20 Im Gegensatz zum vorhergehenden Foto sind auf diesem Schema Funkmodule ohne Antenne gezeigt. Sie kann zur Reichweitensteigerung jeweils nachträglich angelötet werden.

Bei der Programmierung unterstützt uns wieder eine Bibliothek, welche jedoch nicht im Installationsumfang der Arduino IDE enthalten ist – wir können sie aber problemlos nachinstallieren. Klicken Sie dazu auf *Sketch -> Bibliothek einbinden -> Bibliotheken verwalten*. Es öffnet sich ein Fenster, in dessen Suchfeld Sie „*RCSwitch*“ eingeben. In der angezeigten Ergebnisliste können Sie direkt auf *Installieren* klicken. Danach ist die Bibliothek sofort verfügbar und kann eingebunden werden.



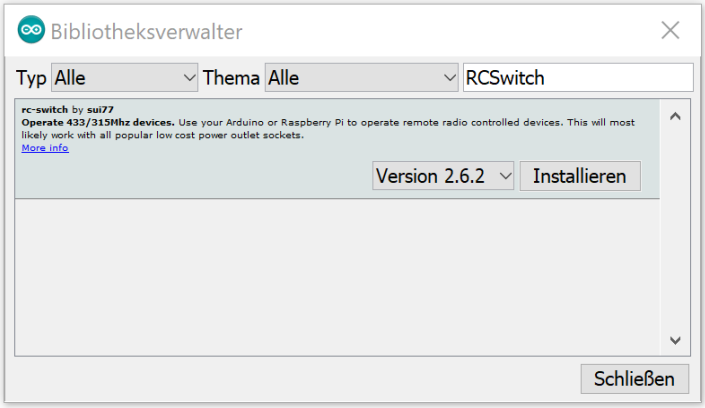


Abb. 5.21 Die erforderliche Bibliothek RCSwitch kann aus dem Internet heruntergeladen werden.

Schauen wir uns nun also die beiden Sketche an. Auf der Senderseite ist er sehr kompakt:

```
1  #include "RCSwitch.h"                                    // (1)
2
3  RCSwitch Sendemodul = RCSwitch();                        // (2)
4
5  void setup() {
6    Sendemodul.enableTransmit(10);                         // (3)
7  }
8
9  void loop() {
10   Sendemodul.send(analogRead(2)+1, 16);                  // (4)
11   delay(1000);
12 }
```

1. Wie auch bereits bei den vorinstallierten Bibliotheken erfolgt die Einbindung einfach über den Dateinamen. Die heruntergeladenen Dateien selbst finden Sie übrigens im Arduino-Verzeichnis auf Ihrer Festplatte im Unterordner *\libraries*.
2. Die eingebundene Bibliothek wurde objektbasiert programmiert (siehe Kapitel 4.12) und stellt die Klasse `RCSwitch` zur Verfügung.

Wir definieren hier eine Instanz dieser Klasse mit dem Namen `Sendemodul`. Dieses Objekt repräsentiert nun unsere kleine Sendeplatine.

3. Wir initialisieren unser Sendemodul über die Funktion `enableTransmit()` und übergeben ihr die Nummer des verwendeten Anschlusspins.

4. Für die eigentliche Datenübertragung steht hier die Funktion `send()` zur Verfügung. Sie erwartet als erstes Argument die Daten und als zweites Argument deren Länge in Bit. Da der Rückgabewert der Funktion `analogRead()` den Typ `int` aufweist, handelt es sich um 16 Bit Nutzdaten (auch wenn nur 10 Bit genutzt werden). Der eingelesene Wert wird hier um 1 erhöht, um den Wert 0 zu vermeiden. Die Übertragung einer 0 könnte bei der hier verwendeten Bibliothek vom Empfänger auch als „keine Daten" missinterpretiert werden.

Auf der Empfängerseite verwenden wir den folgenden Sketch:

```
#include "RCSwitch.h"

RCSwitch Empfangsmodul = RCSwitch();                          // (1)

void setup()
{
  Serial.begin(9600);                                         // (2)
  Empfangsmodul.enableReceive(0);                             // (3)
  pinMode(9, OUTPUT);
}

void loop() {
  if (Empfangsmodul.available())                              // (4)
  {
    int Empfangswert = Empfangsmodul.getReceivedValue()-1;    // (5)
    Serial.print("Empfangswert: ");
    Serial.println(Empfangswert);
    analogWrite(9, Empfangswert/4);                           // (6)
  }
  Empfangsmodul.resetAvailable();                             // (7)
}
```

1. Auch hier erzeugen wir ein Objekt als Instanz der Klasse `RCSwitch`, diesmal repräsentiert es jedoch unser Empfangsmodul, deshalb wählen wir auch diesen Namen.
2. Als Erweiterung zu den vorherigen Beispielen möchten wir hier die Empfangsdaten zur besseren Veranschaulichung auch seriell zum Computer übertragen und dort anzeigen.
3. Die Funktion `enableReceive()` stellt die Empfangsbereitschaft her. Die Besonderheit ist dabei, dass als Argument ein Interrupt-Pin[9] übergeben werden muss. Der Arduino UNO besitzt nur zwei davon: Den digitalen Pin 2, welcher als Interrupt-Pin 0 bezeichnet wird, und digital-Pin 3, bezeichnet als Interrupt-Pin 1. Hier ist also digital-Pin 2 gemeint, an den die Datenleitung des Empfangsmoduls angeschlossen ist.
4. Die Funktion `available()` gibt `true` zurück, wenn Daten empfangen wurden – ansonsten `false`. Hier liegt auch der Grund, warum wir auf der Senderseite die Daten um 1 erhöht haben – würden wir eine 0 empfangen, gäbe diese Funktion trotz des Empfangs weiterhin ein `false` zurück.
5. Im Falle eines Datenempfangs erhalten wir als Rückgabewert der Funktion `getReceivedValue()` die Nutzdaten, welche wir sogleich wieder um 1 verringern, um die ursprüngliche Zahl zu erhalten.
6. Wie bei den vorigen Beispielen, müssen wir den Wert ganzzahlig durch 4 teilen, um den `Empfangswert` (von bis zu 1023) auf den Wertebereich der Pulsweitenmodulation (maximal 255) zu beschränken.
7. Die genutzte Bibliothek erfordert es, nach dem Auslesen der Daten den Empfänger per `resetAvailable()` wieder auf Empfang zu schalten.

9 Diese Pins sind chipintern mit besonderen Schaltungen ausgestattet und können zur Unterbrechung des laufenden Programms (Interrupt) genutzt werden, wenn sie im Sketch dazu vorgesehen werden. Insbesondere beim Datenempfang kann dies relevant sein.

Die Ausgabe am seriellen Monitor sieht beispielhaft so aus:

Abb. 5.22 Bildschirmausgabe am Empfänger

Bei der Beobachtung der Ausgabe am Bildschirm fällt auf, dass manchmal Werte schnell hintereinander doppelt oder gar dreifach empfangen werden. Das liegt daran, dass die verwendete Bibliothek standardmäßig jeden Sendevorgang dreimal wiederholt, um die Chance auf eine erfolgreiche Übertragung zu erhöhen – schließlich erhält der Sender keine Rückmeldung vom Empfänger, ob die Nachricht angekommen ist. Außerdem kann es aufgrund von Funkstörungen durchaus vorkommen, dass die gesendeten Bits am Empfänger falsch interpretiert werden und somit falsche Werte ausgelesen werden. Diese Funkschnittstelle ist aus den genannten Gründen nicht sehr robust – für einfache Anwendungen, wie zum Beispiel das Steuern von Dekorationslicht, aber dennoch geeignet. Durch entsprechende Prüfbedingungen im Programmcode kann man die Anfälligkeit für Störungen stark minimieren. So könnte man beispielsweise dafür sorgen, dass nur Empfangsdaten verwendet werden, die mindestens zweimal hintereinander identisch empfangen wurden.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

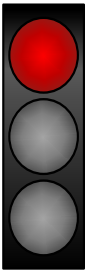
Kapitel 6
Praxisprojekt: Modellbau-Ampel

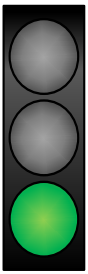
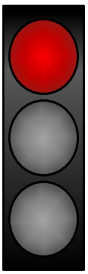
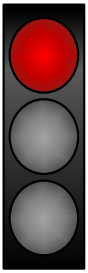
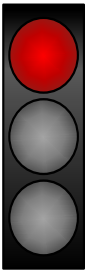
Langsam wird es Zeit, die bisher meist isoliert betrachteten Sachverhalte an etwas komplexeren Beispielen mit mehr Praxisbezug zu demonstrieren. Zu diesem Zweck gibt es in diesem Buch mehrere Praxisprojekte, welche Sie – wie auch alle anderen Beispiele – gern selbst nachbauen können. Hinweise zum Bezug der benötigten Bauteile finden Sie im Anhang.

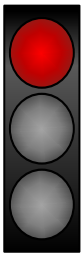
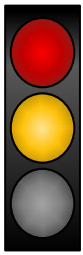
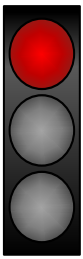
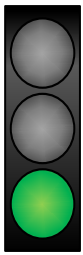
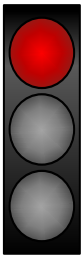
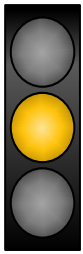
6.1 Idee

Unser erstes Projekt soll eine Ampelanlage sein, welche beispielsweise im Modellbau verwendet werden könnte, um eine Baustelle mit Fahrbahneinengung zu simulieren. Dementsprechend hat die Ampel zwei Standorte, um den Verkehr aus beiden Richtungen zu regeln. Zusätzlich soll es möglich sein, die Anlage per Knopfdruck abzuschalten, so dass beide Ampeln nur noch ein gelbes Blinklicht zur Warnung zeigen.

Die Ampelanlage durchläuft im Normalbetrieb also 8 Phasen in zyklischer Wiederholung. Die Dauer der einzelnen Phasen ist dabei unterschiedlich.

Phase	Ampel 1	Ampel 2
0 Dauer: 2 Sekunden		

Phase	Ampel 1	Ampel 2
1 Dauer: 1 Sekunde		
2 Dauer: 8 Sekunden		
3 Dauer: 2 Sekunden		
4 Dauer: 2 Sekunden		

Phase	Ampel 1	Ampel 2
5 Dauer: 1 Sekunde		
6 Dauer: 8 Sekunden		
7 Dauer: 2 Sekunden		
Zusätzlich definieren wir zwei weitere Phasen für den ausgeschalteten Zustand (gelbes Blinklicht):		
10 Dauer: 1 Sekunde		

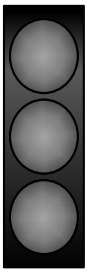
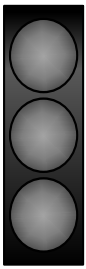
Phase	Ampel 1	Ampel 2
11 Dauer: 1 Sekunde		

Tabelle 6.1 Skizzierung der Ampelphasen

Natürlich eignen sich als Lichtzeichen am besten LEDs, welche problemlos in den Farben rot, gelb und grün erhältlich sind. Da unser Arduino genügend digitale Ausgangspins hat, können wir die sechs LEDs über jeweils einen Vorwiderstand an jeweils einen Ausgangspin anschließen. Einen weiteren digitalen Pin verwenden wir für den Taster.

Wie in Kapitel 3.1.2.4 gezeigt, können Sie die benötigen Vorwiderstände für die LEDs selbst berechnen. Haben Sie jedoch keine genauen Informationen über die Flussspannung und erlaubte Stromstärke Ihrer LEDs, so können Sie in der Arduino-Welt auch einfach Vorwiderstände von 330 Ohm verwenden. Dieser Wert ist so hoch gewählt, dass selbst bei LEDs mit äußerst niedriger Flussspannung von 1,2 V nur ein Strom von 11 mA fließen würde – das hält jede Standard-LED aus. Dennoch kann selbst im anderen Extremfall, bei LEDs mit Flussspannungen von 3,2 Volt, immer noch ein Strom von 5 mA fließen. Das reicht aus, um ein deutliches Leuchten wahrzunehmen. Da es uns in diesem Beispiel nur darum geht, ein deutliches Lichtsignal zu sehen – auch wenn es nicht die maximale Helligkeit der LED erreicht – setzen wir zur Vereinfachung ebenfalls alle Widerstandswerte auf 330 Ohm.

6.2 Stromlaufplan

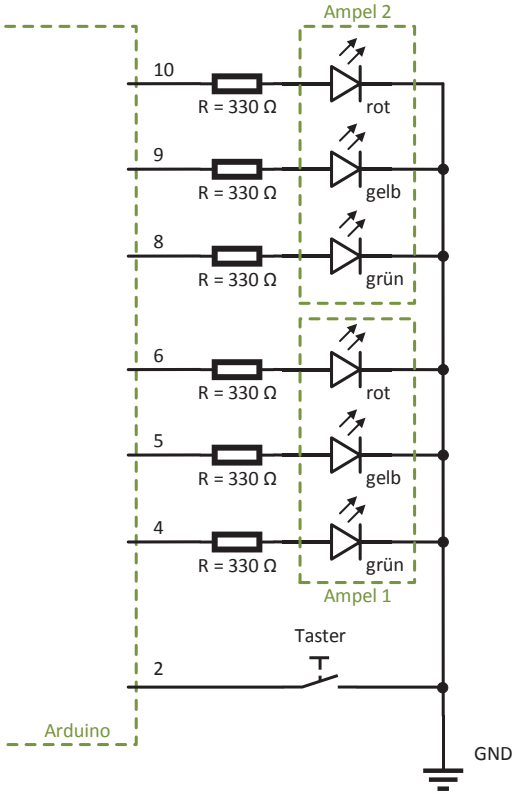


Abb. 6.1 Der Stromlaufplan ist einfach nachzuvollziehen

6.3 Versuchsaufbau

Wir benötigen:

- 1 kleines Breadboard
- 1 Arduino UNO
- 6 Widerstände 330 Ohm

- 2 rote LEDs
- 2 gelbe LEDs
- 2 grüne LEDs
- 1 Taster
- Verbindungsdrähte

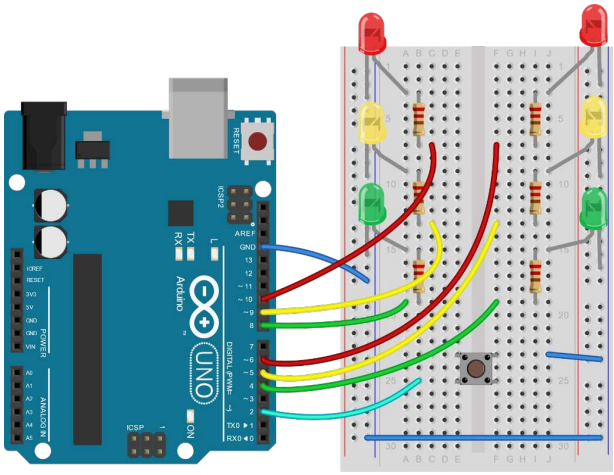


Abb. 6.2 Durch die Wahl passender Farben für die Verbindungsdrähte wird die Testschaltung übersichtlicher.

6.4 Programmcode

Der Sketch ist nun etwas länger, daher sind kurze Erläuterungen direkt im Sketch kommentiert:

```
#define AMPEL1ROT 6  // Die Pins der einzelnen LEDs und der Taste
                     //werden zur
#define AMPEL1GELB 5 // besseren Übersicht als Konstantennamen
                     //angelegt
#define AMPEL1GRUEN 4
#define AMPEL2ROT 10
#define AMPEL2GELB 9
```

```
#define AMPEL2GRUEN 8

#define TASTE 2

#define AUS 0         // Hier werden Zahlen vereinbart,
                      // welche ein
#define ROT 1         // bestimmtes Lichtzeichen
                      // darstellen sollen.
#define ROTGELB 2     // Wir verwenden sie später im Zusammenhang
#define GRUEN 3       // mit der Funktion setzeAmpel().
#define GELB 4

long Timer = 0;      // dient der Zeitmessung
byte Phase = 0;      // repräsentiert die aktuelle Ampelphase

void setup() {
  pinMode(AMPEL1ROT, OUTPUT);
  pinMode(AMPEL1GELB, OUTPUT);
  pinMode(AMPEL1GRUEN, OUTPUT);
  pinMode(AMPEL2ROT, OUTPUT);
  pinMode(AMPEL2GELB, OUTPUT);
  pinMode(AMPEL2GRUEN, OUTPUT);
  pinMode(TASTE, INPUT_PULLUP);
}

void loop() {

  while(millis() < Timer)                                    // (1)
  {
    if(!digitalRead(TASTE)) // Wird die Taste betätigt?
    {                       // falls ja -> Betriebsmodus
                            // umschalten:
      if(Phase < 10)        // Befinden wir uns gerade im normale
                            // Betrieb?
        Phase = 10;         // ja -> auf gelbes Blinken schalten
                            // else
        Phase = 0;          // nein -> den normalen Betrieb
                            // starten

      Timer = 0;            // Timer zurücksetzen, damit
                            // übergeordnete
                            // while-Schleife verlassen wird.

      while(!digitalRead(TASTE));                            // (2)
    }
  }

  switch(Phase)                                              // (3)
```

```
56   {
57     case 0: // Beide Ampeln rot
58       setzeAmpel(1, ROT);                                  // (4)
59       setzeAmpel(2, ROT);
60       Timer = millis() + 2000;                             // (5)
61       Phase++;              // Weiterschaltung der Ampelphase
62     break;
63
64     case 1: // Ampel 1 rotgelb
65       setzeAmpel(1, ROTGELB);
66       Timer = millis() + 1000;
67       Phase++;
68     break;
69
70     case 2: // Ampel 1 grün
71       setzeAmpel(1, GRUEN);
72       Timer = millis() + 8000;
73       Phase++;
74     break;
75
76     case 3: // Ampel 1 gelb
77       setzeAmpel(1, GELB);
78       Timer = millis() + 2000;
79       Phase++;
80     break;
81
82     case 4: // Beide Ampeln rot
83       setzeAmpel(1, ROT);
84       Timer = millis() + 2000;
85       Phase++;
86     break;
87
88     case 5: // Ampel 2 rotgelb
89       setzeAmpel(2, ROTGELB);
90       Timer = millis() + 1000;
91       Phase++;
92     break;
93
94     case 6: // Ampel 2 grün
95       setzeAmpel(2, GRUEN);
96       Timer = millis() + 8000;
97       Phase++;
98     break;
99
100    case 7: // Ampel 2 gelb
101      setzeAmpel(2, GELB);
102      Timer = millis() + 2000;
103      Phase = 0;    // nach Phase 7 beginnt Zyklus von vorn
```

```
104     break;
105
106     case 10: // beide Ampeln gelb
107       setzeAmpel(1, GELB);
108       setzeAmpel(2, GELB);
109       Timer = millis() + 1000;
110       Phase = 11;   // Phase 10 und 11 wechseln einander ab
111     break;          // und erzeugen so das gelbe Blinken
112
113     case 11: default: // beide Ampeln aus
114       setzeAmpel(1, AUS);
115       setzeAmpel(2, AUS);
116       Timer = millis() + 1000;
117       Phase = 10;
118     break;
119   }
120 }
121
122 void setzeAmpel(byte Ampelnummer, byte Zustand)             // (6)
123 {
124   byte rotPin = 0; byte gelbPin = 0; byte gruenPin = 0;     // (7)
125
126   if(Ampelnummer == 1)
127   {
128     rotPin = AMPEL1ROT;
129     gelbPin = AMPEL1GELB;
130     gruenPin = AMPEL1GRUEN;
131   }
132   if(Ampelnummer == 2)
133   {
134     rotPin = AMPEL2ROT;
135     gelbPin = AMPEL2GELB;
136     gruenPin = AMPEL2GRUEN;
137   }
138
139   switch(Zustand)
140   {
141     case AUS:
142       digitalWrite(rotPin, LOW);
143       digitalWrite(gelbPin, LOW);
144       digitalWrite(gruenPin, LOW);
145     break;
146     case GRUEN:
147       digitalWrite(rotPin, LOW);
148       digitalWrite(gelbPin, LOW);
149       digitalWrite(gruenPin, HIGH);
150     break;
151     case GELB:
```

```
152       digitalWrite(rotPin, LOW);
153       digitalWrite(gelbPin, HIGH);
154       digitalWrite(gruenPin, LOW);
155     break;
156     case ROTGELB:
157       digitalWrite(rotPin, HIGH);
158       digitalWrite(gelbPin, HIGH);
159       digitalWrite(gruenPin, LOW);
160     break;
161     case ROT: default:
162       digitalWrite(rotPin, HIGH);
163       digitalWrite(gelbPin, LOW);
164       digitalWrite(gruenPin, LOW);
165     break;
166   }
167 }
```


1. Statt wie bisher die Zeitverzögerung per `delay()` zu realisieren, verwenden wir hier eine `while`-Schleife, welche so lange immer wieder durchlaufen wird, bis der in der Variable `Timer` angegebene Zeitpunkt (angegeben in *Millisekunden seit Programmstart*) erreicht ist. Das hat den großen Vorteil, dass wir innerhalb dieser Verzögerungsschleife noch andere Dinge erledigen können – beispielsweise prüfen, ob die Taste gedrückt wird. Würden wir stattdessen `delay()` nutzen, wäre das Programm während der Wartezeit quasi handlungsunfähig[1].

2. Die Bedingung `if(!digitalRead(TASTE))` einige Zeilen weiter oben prüft, ob gerade die Taste gedrückt wird (die Negation ergibt sich aus der Schaltung als Pull-Up-Eingang, siehe Kapitel 5.2). Ist dies der Fall, wird die Betriebsart (`Phase`) entsprechend umgeschaltet. Da dies nur Bruchteile einer Millisekunde dauert, wäre die Taste vermutlich beim nächsten `loop()`-Durchlauf immer noch gedrückt und würde erneut eine Umschaltung auslösen. Um dies zu verhindern, nutzen wir diese `while()`-Schleife in Form einer „Falle“: Solange die Taste noch gedrückt ist, wird diese Schleife immer wieder durchlaufen. Der Schleifenrumpf ist leer (und deshalb über-

1 Eine Unterbrechung per Interrupt ist dennoch möglich, jedoch ist dieses Vorgehen für Einsteiger eher ungeeignet. Daher wollen wir hier nicht näher darauf eingehen.

haupt nicht vorhanden), daher steht direkt hinter der schließenden runden Klammer ein Semikolon. Das Programm bleibt also so lange hier „hängen", bis die Taste wieder losgelassen wird.

3. Die Kontrollstruktur `switch` kennen Sie noch nicht. Sie vereinfacht den Programmcode, indem sie mehrere `if`-Anweisungen ersetzen kann, falls diese zur Fallunterscheidung dienen. Statt mit

```
if(Phase == 0)
{…}
if(Phase == 1)
{…}
if(Phase == 2)
{…}
```

alle möglichen Ampelphasen „abzuklopfen", gestattet die `switch`-Anweisung eine deutlich kürzere Notation. In den runden Klammern ist die (ganzzahlige) Entscheidungsvariable angegeben. Im darauf folgenden, von geschweiften Klammern eingeschlossenen Rumpf wird dann an die Stelle gesprungen, an welcher der aktuelle Wert der Entscheidungsvariablen vermerkt ist – zum Beispiel `case 3:` im Falle, dass `Phase` gerade den Wert `3` hat. Von dort aus beginnt dann die Befehlsbearbeitung, bis der Rumpf endet oder über die Anweisung `break` verlassen wird.

Gibt es keine `case`-Markierung mit dem betreffenden Zahlenwert, so wird an die mit `default:` markierte Stelle gesprungen. Gibt es auch diese nicht, wird der Rumpf der `switch`-Struktur übersprungen. Darauf folgend sind für jede Ampelphase entsprechende Befehle notiert.

4. `setzeAmpel()` ist eine selbstdefinierte Funktion (siehe weiter unten), welche ein gewünschtes Lichtsignal auf eine der beiden Ampeln ausgeben kann. Als erstes Argument ist die Nummer der Ampel, als zweites das gewünschte Signal zu übergeben.

5. Die `Timer`-Variable gibt den Zeitpunkt des nächsten Signalwechsels an, dieser soll hier in `2` Sekunden (2000 Millisekunden) erfolgen und unterscheidet sich je nach Ampelphase. Der Zeitpunkt wird dabei in *Millisekunden seit Programmstart* angegeben.

6. Die selbsterstellte Funktion `setzeAmpel()` erledigt die Ausgabe eines bestimmten Lichtzeichens (`Zustand`) auf eine der beiden Ampeln. Beide Argumente sind ganzzahlige Variablen, deren Zahlenwerten willkürlich eine programmweit einheitliche Bedeutung zugewiesen wurde. Demnach steht also beispielsweise bei `Ampelnummer` der Wert `1` für die an den Pins 4, 5 und 6 angeschlossenen LEDs und für `Zustand` beschreibt die Zahl `3` ein grünes Lichtzeichen. Da insbesondere bei den Werten von `Zustand` schnell der Überblick verloren gehen kann, wurden für die Zahlen Konstanten definiert, welche die Bedeutung leichter erkennen lassen. Sie kennen dies bereits von anderen Funktionen, beispielsweise übergeben Sie der Funktion `pinMode()` in Wirklichkeit auch nur eine (im Hintergrund vereinbarte, für uns irrelevante) Zahl, wenn Sie einen Modus wie `OUTPUT` angeben.

7. Da Zeilenumbrüche für den Compiler ohnehin nicht relevant sind, kann man kurze triviale Befehle (wie diese Variablendeklarationen) auch einfach hintereinanderschreiben, um Platz zu sparen. Der Compiler erkennt das Ende eines Befehls wie gehabt am Semikolon.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 7

Anzeigeelemente

Nachdem wir nun alles Grundlegende über die Anschlussmöglichkeiten des Arduino kennengelernt haben, wollen wir uns in den folgenden Kapiteln den Komponenten widmen, die daran angeschlossen werden können. Den Anfang bilden diverse Bauelemente zur Anzeige von Informationen.

7.1 Leuchtdioden

Leuchtdioden haben wir ja bereits in mehreren Beispielen eingesetzt. Sie sind eine simple und effektive Art zur Ausgabe von Informationen, wenn es sich um schlichte ja/nein-Indikatoren handelt. Den üblichen Versuchsaufbau mit Vorwiderstand haben wir bereits erörtert, deshalb widmen wir uns nun einem etwas veränderten Beispiel, um das Verhalten von Dioden näher zu beleuchten.

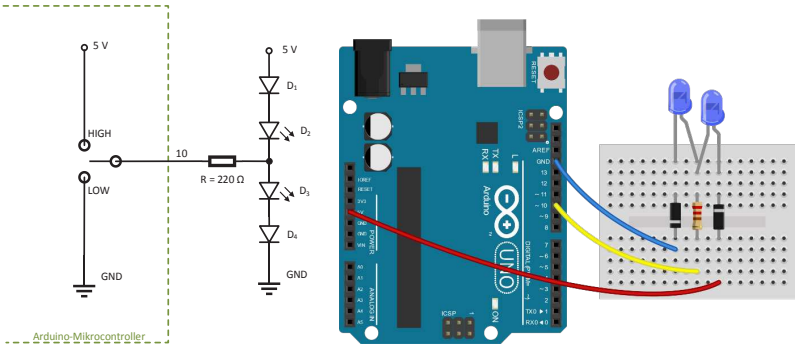


Abb. 7.1 In diesem Beispiel werden zwei blaue Leuchtdioden und zwei Siliziumdioden in Durchlassrichtung in Reihe geschaltet.

Diese Schaltung wirkt zunächst etwas seltsam. Scheinbar kann der Strom doch einfach direkt über alle 4 Dioden fließen, da sie in Durch-

lassrichtung eingebaut wurden. Doch wir müssen zum einen bedenken, dass sich die Spannung aufgrund der Reihenschaltung auf alle 4 Dioden aufteilt, zum anderen besitzt jede Diode aufgrund ihrer Kennlinie eine gewisse Flussspannung (siehe Kapitel 3.1.2.4), unterhalb der sie fast keinen Stromfluss zulässt. Bei den Silizium-Dioden liegt diese Spannung bei etwa 0,7 V, bei den blauen LEDs jedoch bei rund 3 V. Um überhaupt einen relevanten Stromfluss durch die Dioden zu erzeugen, müsste die Betriebsspannung also sogar 7 Volt übersteigen. Bei den hier anliegenden 5 V passiert demnach zunächst gar nichts.

Nun kommt der Arduino ins Spiel. Steht sein Ausgang auf `HIGH`, so fließt der Strom von der Spannungsquelle durch den Ausgangspin über den Widerstand durch die unteren beiden Dioden. In Summe haben sie eine Flussspannung von rund 3,7 Volt, somit reichen die 5 V Betriebsspannung aus, um sie zum Durchlass zu bringen. Der Widerstand begrenzt dabei die Stromstärke. Der Strom lässt die LED entsprechend leuchten. Durch die oberen beiden Dioden fließt weiterhin gar kein Strom. Steht der Pin auf `LOW`, verhält es sich umgekehrt: Der Strom fließt von der Spannungsquelle durch die oberen beiden Dioden durch den Widerstand in den Arduino-Pin, welcher direkt mit Masse verbunden ist. Nun fließt durch die unteren beiden Dioden kein Strom.

Zum Test der Funktion eignet sich der Sketch aus Kapitel 5.4, mit welchem die Pulsweitenmodulation demonstriert wurde. Er lässt beide LEDs im Wechsel sanft auf- und abblenden – und das mit nur einem Ausgangspin! Natürlich werden die LEDs auch hier wieder in Wahrheit sehr schnell ein- und ausgeschaltet, aber unser Auge kann es schlicht nicht wahrnehmen. Stattdessen sehen wir einen gleitenden Übergang.

Eigentlich wäre es ja naheliegender, die LEDs mit einer gleitenden Spannung zwischen 0 V und 5 V zu dimmen. Probieren wir es aus, indem wir statt des Arduino-Ausgangspins ein Potentiometer verwenden:

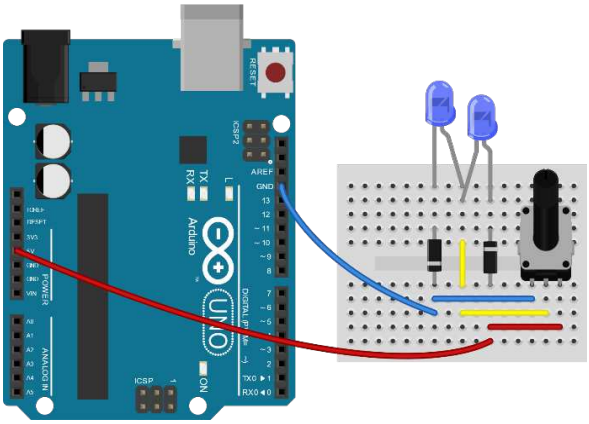


Abb. 7.2 Nun ersetzt das Potentiometer den Ausgang des Arduino

Beim Test stellt sich jedoch heraus, dass sich die LEDs damit deutlich schlechter regeln lassen. An einem Endanschlag des Potis ist eine der beiden LEDs erwartungsgemäß erleuchtet. Dreht man jedoch weiter, erlischt sie lange bevor die andere zu leuchten beginnt. Dieses Verhalten liegt in der bereits angesprochenen nichtlinearen (abgeknickten) Kennlinie begründet. Die Helligkeit der LED lässt sich nur in einem kleinen Spannungsbereich regeln – liegt die Spannung darunter, leuchtet sie gar nicht, liegt sie zu hoch, wird sie zerstört.

Aus diesem Grund ist die Pulsweitenmodulation bei der Dimmung von LEDs sogar ein deutlich vorteilhafteres Verfahren gegenüber der Spannungsregelung. Dies gilt allerdings nur, wenn die Frequenz des PWM-Signals groß genug ist, damit unser Auge kein Flackern wahrnehmen kann. Sicher kennen Sie unangenehme Flimmereffekte an manchen dimmbaren LED-Beleuchtungsinstallationen. Ursache sind sehr häufig suboptimale PWM-Dimmer.

Es sei noch erwähnt, dass alternativ auch Konstantstromquellen zur Speisung von LEDs verwendet werden können. Dies kommt ebenfalls häufig in der Beleuchtungstechnik zum Einsatz. Derartige Energiequellen erzeugen einen konstanten Stromfluss, unabhängig von der dafür benötigten Spannung (abgesehen von Maximalgrenzen). Da die Hellig-

keit einer LED von dem durch sie fließenden Strom abhängt, garantiert dies eine konstante Helligkeit. Werden mehrere LEDs in Reihe geschaltet, erhöht die Konstantstromquelle automatisch die Spannung, um den gewünschten Stromfluss beizubehalten.

7.2 RGB-LED

Statt einer einzelnen LED können natürlich auch an jeden weiteren Arduino-Ausgang LEDs angeschlossen werden. Eine besondere Variante stellen dabei sogenannte RGB-LEDs dar. Sie vereinen je eine rote, grüne und blaue LED in einem Gehäuse, wodurch beliebige Farben „gemischt" werden können. Die Anoden (+) jeder LED sind dabei einzeln herausgeführt. Als Kathode (-) dient ein gemeinsamer Kontakt.

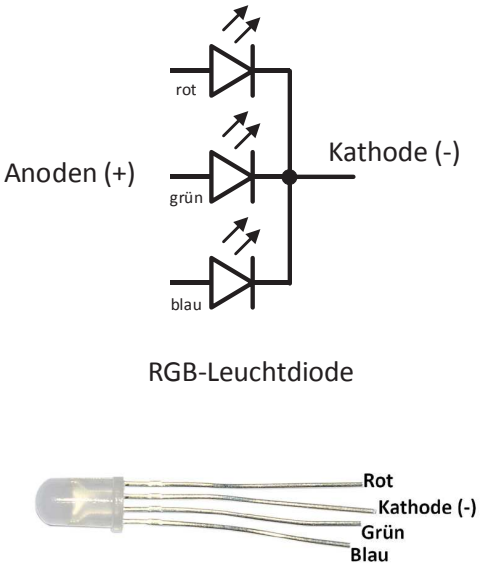


Abb. 7.3 Eine RBG-LED besteht aus drei separaten Leuchtdioden in einem gemeinsamen Gehäuse

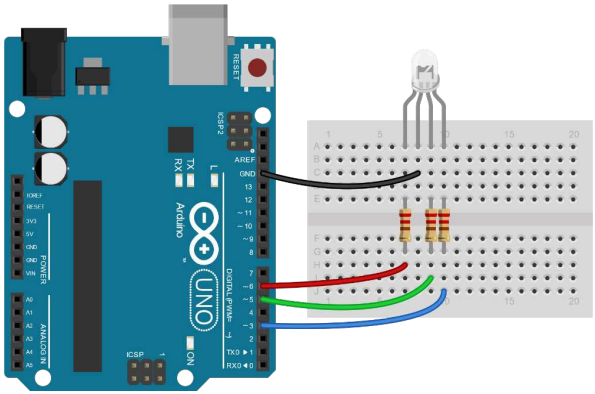


Abb. 7.4 Da es sich im Prinzip um 3 unabhängige LEDs handelt, benötigt jede einen Vorwiderstand von 220 Ohm

Als Ausgangspins wurden bewusst Pins gewählt, welche die Pulsweitenmodulation unterstützen. Der folgende Sketch lässt die RGB-LED zur Demonstration im Sekundentakt in einer anderen (zufälligen) Farbe leuchten.

```
1  void setup() {
2    pinMode(3, OUTPUT);
3    pinMode(5, OUTPUT);
4    pinMode(6, OUTPUT);
5  }
6
7  void loop() {
8    analogWrite(3, random(256));                    // (1)
9    analogWrite(5, random(256));
10   analogWrite(6, random(256));
11   delay(1000);
12 }
```

1. Neu ist hier lediglich die Funktion `random()`. Sie gibt als Rückgabewert eine Zufallszahl[1] im Bereich von 0 bis zu der im Argument

1 Genau genommen handelt es sich um eine Pseudozufallszahl, da der Arduino keine echten Zufälle erzeugen kann. Dies äußert sich darin, dass sich die Reihenfolge der Zufallszahlen nach jedem Einschalten identisch ist. Probieren Sie es aus – merken Sie sich die

übergebenen Zahl (welche selbst nicht mehr zum Bereich gehört). Somit wird als Helligkeit für jede Farbe eine Zahl von 0 bis 255 „ausgewürfelt".

7.3 7-Segment-Anzeige

Abb.7.5 typische 7-Segment-Anzeige

Diese Ziffernanzeige kennen Sie sicherlich aus verschiedenen Anwendungen – vom Radiowecker bis zum Hoteltresor. Der Name rührt von der Zusammensetzung aus 7 Balken beziehungsweise Segmenten. Unter Berücksichtigung des ebenfalls vorhandenen Punktes ist manchmal auch von der 8-Segment-Anzeige die Rede, es geht jedoch immer um das gleiche Bauteil. Wie schon bei der RGB-LED wurden auch hier mehrere LEDs in einem Gehäuse zusammengefasst und teilen sich die Kathode. Da der Arduino genügend Ausgangspins besitzt, könnte man einfach jede LED (über einen Vorwiderstand) direkt ansteuern. Dies führt allerdings zu Problemen, wenn noch zusätzliche Komponenten mit mehreren Pins (zum Beispiel mehrere Taster) genutzt werden sollen.

---

ersten drei Farben nach dem Einschalten und drücken Sie den Reset-Knopf auf dem Board. Sie werden danach wieder die identischen Farben sehen. Für unsere Zwecke ist dies jedoch nicht von Belang.

Es gibt daher auch eine elegantere Möglichkeit der Ansteuerung: Schieberegister. Der 74HC595-Baustein ist ein solches. Die Daten können bitweise hintereinander in den Baustein „geschoben“ werden. Die jeweils letzten 8 Bit gibt er dann über 8 Pins aus.

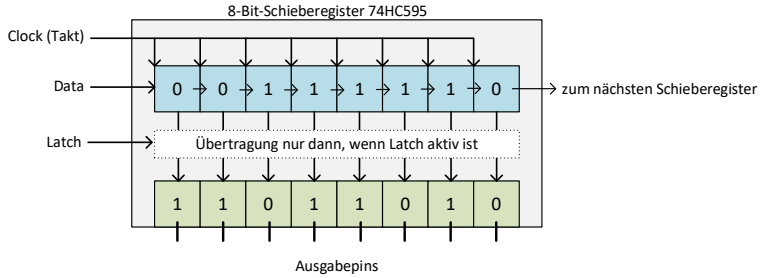
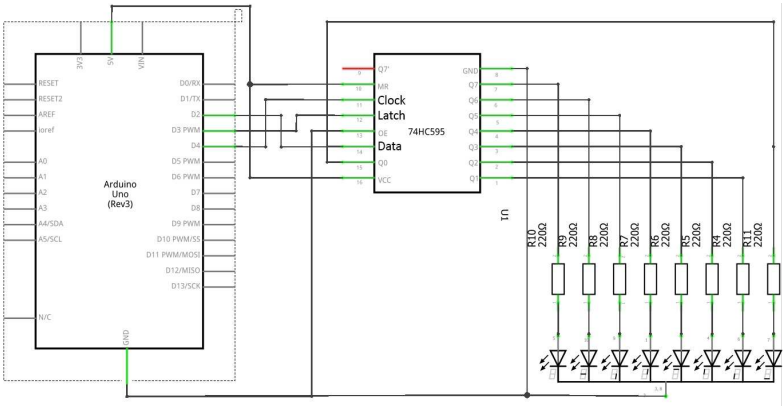


Abb.7.6 Schieberegister sind praktische Helfer bei der Datenausgabe

Es reichen drei Pins des Arduino, um das Schieberegister zu steuern. Der Data-Pin überträgt nacheinander die eigentlichen Bitwerte, wobei die Clock-Leitung den Takt dafür angibt. Wurden alle Bits übertragen, sorgt ein Impuls auf der Latch-Leitung dafür, dass das Bitmuster auf die Ausgangspins der Schieberegisters geleitet wird.



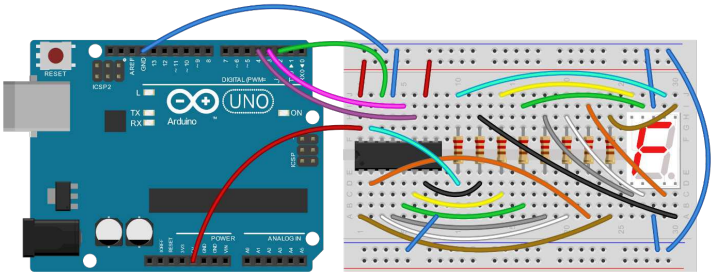


Abb. 7.7 Da jedes LED-Element einen Vorwiderstand von 220 Ohm benötigt, ist diese Schaltung etwas umfangreicher.

Folgender Sketch zeigt fortlaufend die Ziffern 0 bis 9 auf der Anzeige:

```
#define LATCHPIN 3                                              // (1)
#define CLOCKPIN 4
#define DATAPIN 2

byte sieben_seg_ziffern[10] = {                                 // (2)
  B11111100,  // = 0
  B01100000,  // = 1
  B11011010,  // = 2
  B11110010,  // = 3
  B01100110,  // = 4
  B10110110,  // = 5
  B10111110,  // = 6
  B11100000,  // = 7
  B11111110,  // = 8
  B11100110   // = 9
};

void setup() {
  pinMode(LATCHPIN, OUTPUT);                                    // (3)
  pinMode(CLOCKPIN, OUTPUT);
  pinMode(DATAPIN, OUTPUT);
}

void loop() {
  for(byte Zahl = 0; Zahl < 10; Zahl++) {                       // (4)
    sevenSegWrite(Zahl);
    delay(1000);
  }
}

```

```
31 void sevenSegWrite(byte ziffer) {                          // (5)
32   digitalWrite(LATCHPIN, LOW);                            // (6)
33   shiftOut(DATAPIN, CLOCKPIN, LSBFIRST,
34   sieben_seg_ziffern[ziffer]);                            // (7)
35   digitalWrite(LATCHPIN, HIGH);                           // (8)
36 }
```

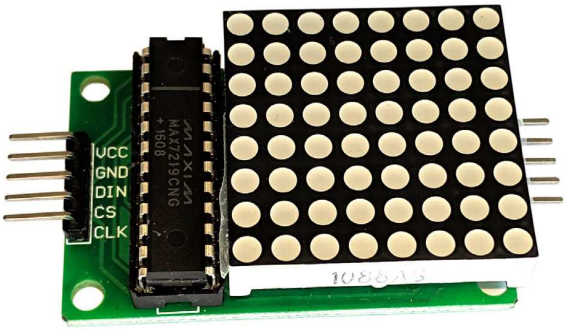
1. Zugunsten der Lesbarkeit verwenden wir Präprozessorkonstanten für die mit dem Schieberegister verbundenen Pins.
2. Wir definieren ein Variablenfeld vom Typ `byte` mit 10 Elementen. Darin wird für jede darstellbare Ziffer ein Bitmuster hinterlegt, welches letztlich darüber entscheidet, welche Segmente leuchten und welche nicht. Experimentieren Sie gern ein wenig damit herum, verändern einige Bits und beobachten das Resultat nach erneutem Hochladen.
3. Die Datenleitungen zum Schieberegister müssen natürlich als `OUTPUT` konfiguriert werden.
4. In der Hauptschleife wird lediglich die Variable `Zahl` wiederkehrend im Sekundentakt von 0 bis 9 gezählt und an die Funktion `sevenSegWrite()` übergeben.
5. In der Funktionsdeklaration wird nun der eigentliche Datentransfer an das Schieberegister durchgeführt.
6. Das Senken des Pegels am Latch-Pin führt dazu, dass die Ausgangspins des Schieberegisters „eingefroren“ werden, so stört das „Einschieben“ der neuen Bits nicht die aktuelle Anzeige.
7. Die Funktion `shiftOut()` ist speziell für Schieberegister vorgesehen. Man übergibt ihr zunächst die Nummern von Daten- und Taktpin, dann die Information, in welcher Reihenfolge die Bits übertragen werden sollen (hier: *LSB first – Least Significant Bit first*, niedrigwertigstes Bit zuerst) und als letztes Argument schließlich noch das entsprechende Bitmuster. Dazu rufen wir die `byte`-Variable ab, welche zu Beginn des Sketches für die entsprechende Ziffer im Array hinterlegt wurde.

8. Das abschließende Heben des Pegels am Latch-Pin bringt das übertragene Bitmuster schließlich zur Anzeige.

Ein großer Vorteil dieser Schieberegister ist ihre Kaskadierbarkeit. Die eingeschobenen Bits können dabei einfach über beliebig viele Register weiter durchgereicht werden, wie in *Abbildung 5.22* angedeutet. Somit kann man auch ein Display mit mehreren Ziffern durch nur drei Arduino-Pins steuern.

7.4 LED-Matrix

Als nächstes wollen wir uns mit einer LED-Matrix auseinandersetzen, wie sie zum Beispiel als Etagenanzeige in manchen Aufzügen verwendet wird. Dabei werden viele einzelne LEDs zu einem Gitter zusammengefasst und die Anoden und Kathoden zeilen- beziehungsweise spaltenweise verbunden.



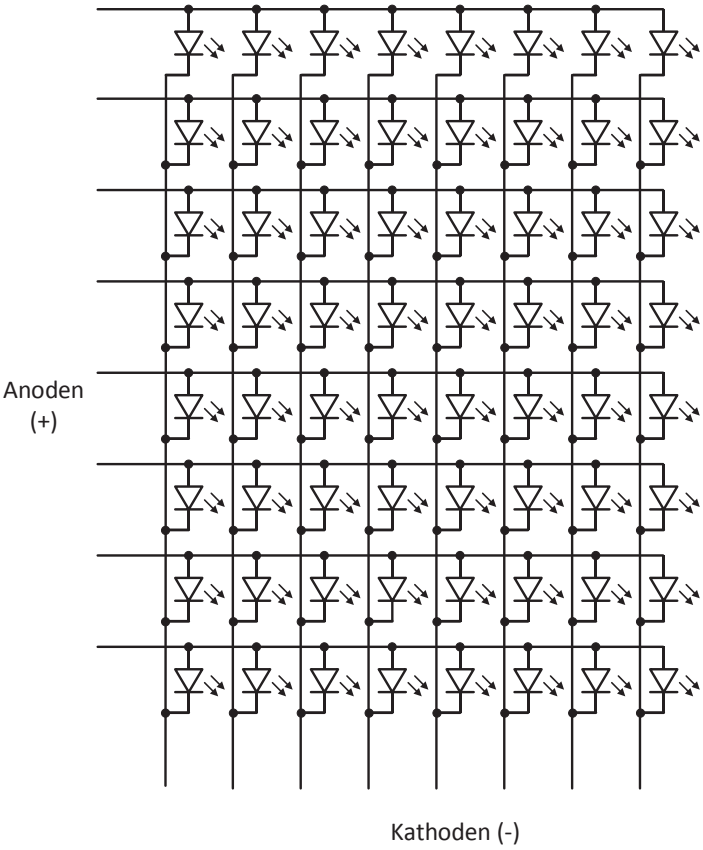


Abb. 7.8 In Matrix-Anzeigen sind Anoden und Kathoden zeilen- und spaltenweise zusammengefasst

Man kann nun jedes beliebige Pixel einzeln zum Leuchten bringen, indem man an die Anode und Kathode der entsprechenden Zeile/Spalte eine Spannung anlegt. Die hier dargestellte Pinzahl übersteigt nun bereits die direkten Anschlussmöglichkeiten unseres Arduino. Doch auch hierfür gibt es Standard-Controller, die uns viel Arbeit abnehmen. Im folgenden Beispiel wollen wir einen MAX7219-Controller mit zugehöriger 8x8-Pixel-Matrix verwenden. Beides ist zusammen als bereits vormontierte Platine erhältlich.

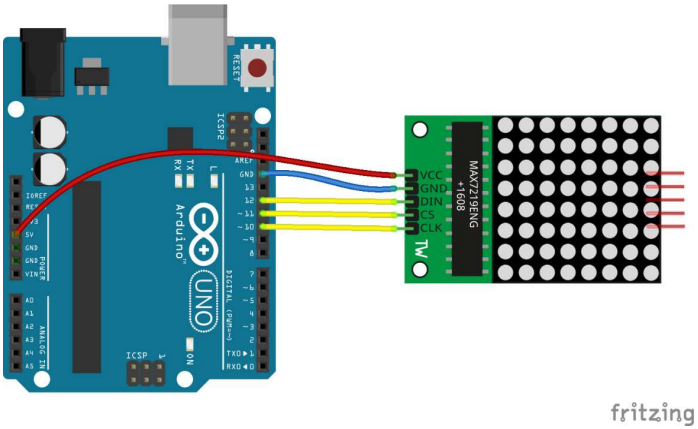


Abb. 7.9 Für die Matrix genügen drei Datenleitungen

Die Ansteuerung des Controllers erfolgt über eine SPI-ähnliche Verbindung, für welche es wieder eine Bibliothek gibt, die wir allerdings zunächst aus dem Internet laden müssen. Wie bereits in Kapitel 5.5.4 gezeigt, können Sie dies direkt über die Arduino IDE erledigen. Suchen Sie dazu im Bibliotheksverwalter nach „*LedControl*“:

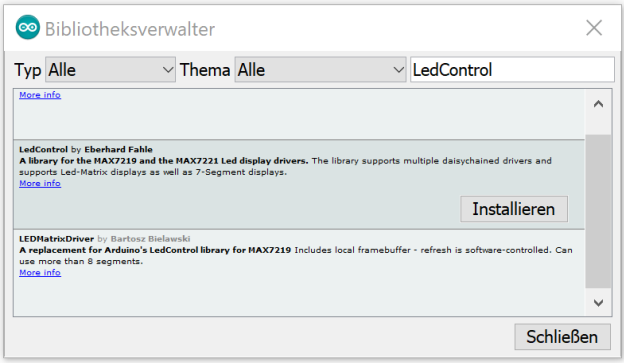


Abb. 7.10 Die erforderliche Bibliothek LedControl kann aus dem Internet heruntergeladen werden

Schauen wir uns nun also einen Sketch an, der die LED-Matrix ansteuert. Nach dem Hochladen werden Sie bemerken, dass in ständiger Wiederholung nacheinander der Schriftzug „A-r-d-u-i-n-o“ auf der LED-Matrix erscheint.

```
#include "LedControl.h"                                    // (1)

#define DATA_IN 12
#define CS 11
#define CLK 10

LedControl Anzeige = LedControl(DATA_IN, CLK, CS, 1);      // (2)

void setup() {
  Anzeige.shutdown(0,false);                               // (3)
  Anzeige.setIntensity(0,8);                               // (4)
  Anzeige.clearDisplay(0);                                 // (5)
}

void loop() {
  Anzeige.setRow(0,0,B00000000); // A                      // (6)
  Anzeige.setRow(0,1,B00111100);
  Anzeige.setRow(0,2,B01000010);
  Anzeige.setRow(0,3,B01111110);
  Anzeige.setRow(0,4,B01000010);
  Anzeige.setRow(0,5,B01000010);
  Anzeige.setRow(0,6,B01000010);
  Anzeige.setRow(0,7,B00000000);
  delay(500);
  Anzeige.setRow(0,1,B00000000); // r
  Anzeige.setRow(0,2,B00000000);
  Anzeige.setRow(0,3,B00111110);
  Anzeige.setRow(0,4,B01000000);
  Anzeige.setRow(0,5,B01000000);
  Anzeige.setRow(0,6,B01000000);
  delay(500);
  Anzeige.setRow(0,1,B00000010); // d
  Anzeige.setRow(0,2,B00000010);
  Anzeige.setRow(0,3,B00111110);
  Anzeige.setRow(0,4,B01000010);
  Anzeige.setRow(0,5,B01000010);
  Anzeige.setRow(0,6,B00111100);
  delay(500);
  Anzeige.setRow(0,1,B00000000); // u
  Anzeige.setRow(0,2,B00000000);
  Anzeige.setRow(0,3,B01000010);
```

```
42    Anzeige.setRow(0,4,B01000010);
43    Anzeige.setRow(0,5,B01000010);
44    Anzeige.setRow(0,6,B01111100);
45    delay(500);
46    Anzeige.setRow(0,1,B00001000); // i
47    Anzeige.setRow(0,2,B00000000);
48    Anzeige.setRow(0,3,B00001000);
49    Anzeige.setRow(0,4,B00001000);
50    Anzeige.setRow(0,5,B00001000);
51    Anzeige.setRow(0,6,B00011100);
52    delay(500);
53    Anzeige.setRow(0,1,B00000000); // n
54    Anzeige.setRow(0,2,B00000000);
55    Anzeige.setRow(0,3,B01111100);
56    Anzeige.setRow(0,4,B01000010);
57    Anzeige.setRow(0,5,B01000010);
58    Anzeige.setRow(0,6,B01000010);
59    delay(500);
60    Anzeige.setRow(0,1,B00000000); // o
61    Anzeige.setRow(0,2,B00000000);
62    Anzeige.setRow(0,3,B00111100);
63    Anzeige.setRow(0,4,B01000010);
64    Anzeige.setRow(0,5,B01000010);
65    Anzeige.setRow(0,6,B00111100);
66    delay(500);
67    Anzeige.clearDisplay(0);
68    delay(500);
69  }
```

1. Hier wird die soeben heruntergeladene Bibliothek zur Ansteuerung des LED-Controllers eingebunden.

2. Die eingebundene Bibliothek wurde objektbasiert programmiert (siehe Kapitel 4.12) und stellt die Klasse `LedControl` zur Verfügung. Wir definieren hier eine Instanz dieser Klasse mit dem Namen `Anzeige`. Dieses Objekt repräsentiert nun unsere LED-Matrix. Wir initialisieren es mit der sogenannten Konstruktorfunktion `LedControl()`. An sie werden Nummern der verwendeten Anschlusspins sowie die Anzahl der LED-Matrizen (man könnte sie kaskadieren) übergeben.

3. Die Funktion `Anzeige.shutdown()` mit dem Argument 0 bewirkt einen Reset des LED-Controllers, was zu Programmstart sinnvoll ist, um eventuelle alte Daten im Speicher zu beseitigen.

4. Hier wird die Helligkeit auf einen mittleren Wert eingestellt. Erlaubt sind Werte von 0 bis 15.
5. Diese Funktion löscht die Anzeige. Das dient in diesem Fall dem „Aufräumen“ vor Programmstart. Auch wenn sie in vielen denkbaren Szenarien an dieser Stelle gar nicht nötig wäre, zeugt sie von einem guten Programmstil.
6. Die Funktion `setRow()` gibt nun ein Bitmuster auf eine bestimmte LED-Zeile aus. Als Argumente werden zunächst die laufende Nummer der LED-Matrix (beginnend bei 0, in unserem Falle also immer 0, da wir nur eine Matrix nutzen), dann die betreffende Zeilennummer (beginnend von oben bei 0) und abschließend die eigentlichen Daten übergeben. Die Funktion sendet die Daten sofort an den Matrixcontroller, dieser gibt sie sofort aus. Auf diese Weise werden nun nacheinander alle Buchstaben übertragen, jeweils mit einer halben Sekunde Pause dazwischen. Wenn Sie sich die Bitmuster im Sketch genau anschauen, können Sie auch darin bereits die Buchstaben erkennen.

Die Bibliothek stellt noch weitere Funktionen bereit, der Umfang würde jedoch den Rahmen einer Einführung sprengen. Es sei daher, wie bei allen Bibliotheken, auf die verfügbare Online-Referenz verwiesen.

7.5 LCD

Flüssigkristallanzeigen (*Liquid Crystal Display* – LCD) begegnen uns im Alltag ebenfalls seit Jahrzehnten. In ihrer ursprünglichen einfarbigen Ausführung kennt sie jeder von Armbanduhren, Taschenrechnern und allerlei anderen Geräten. Im Gegensatz zu den LEDs leuchten sie nicht selbst, sondern bestehen aus einem mehrschichtigen Glasplättchen, welches durch einen dünnen Flüssigkeitsfilm in Kombination mit auf dem Glas befindlichen Polarisationsfiltern Licht entweder blockieren oder hindurchlassen kann. Zur Steuerung verwendet man ein elektrisches Feld. Um auch in dunklen Umgebungen Lesbarkeit zu gewähr-

leisten, können derartige Displays mit einer Hintergrundlichtquelle (zum Beispiel LED) ausgestattet sein.

Im folgenden Beispiel wollen wir ein Standard-LCD verwenden, welches 2 Textzeilen zu je 16 Zeichen darstellen kann. Als de-facto Industriestandard für die Ansteuerung solcher alphanumerischer Displays hat sich bereits vor Jahrzehnten der von Hitachi entwickelte Controller HD44780 durchgesetzt, daher werden diese LCDs meist schon zusammen mit diesem Controller ausgeliefert.

Abb. 7.11 Dieses Standard-LCD ist bereits fest auf einer Platine mit dem HD44780 vormontiert. Deutlich erkennbar sind die Anschlüsse für die zahlreichen Signalleitungen.

Nachteilig ist allerdings, dass zu seiner Ansteuerung mindestens 7 Datenpins (plus Spannungsversorgung) notwendig sind. Auch wenn die Ansteuerung per Arduino damit prinzipiell möglich wäre, wollen wir das Projekt etwas vereinfachen. Zu diesem Zweck sind im Handel spezielle Backpacks (“Rucksäcke“) erhältlich, die auf die Rückseite der LCD-Platine gelötet werden können und die Ansteuerung des HD44780 übernehmen. Zusätzlich verfügen sie über ein Potentiometer, mit welchem der Kontrast des LCD geregelt werden kann – auch diese Aufgabe müssten wir sonst gesondert realisieren. Das Backpack selbst kommuniziert komfortabel über den I^2C-Bus mit dem Arduino.

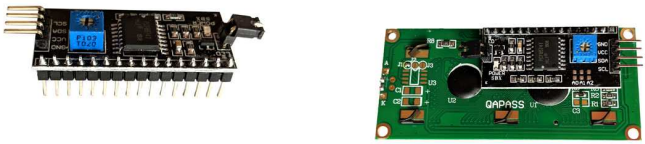


Abb. 7.12 Backpack für HD44780-LCDs - links einzeln, rechts auf die Rückseite der LCD-Platine montiert

Auf der Platine des Backpacks sind drei unbenutzte Lötstellen zu erkennen, markiert mit A0, A1 und A2. Sie führen zu drei Eingangspins des auf dem Board befindlichen Mikrochips und dienen der Beeinflussung der I²C-Adresse.

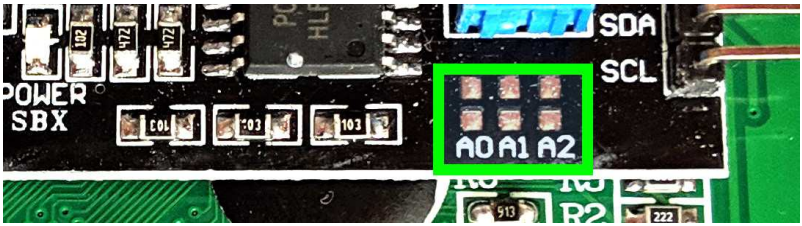


Abb. 7.13 Die unbenutzten Lötpads (grün markiert) dienen der optionalen Beeinflussung der I²C-Adresse. Die oberen drei Kontaktflächen sind mit dem jeweiligen Eingang verbunden, die unteren drei Flächen haben Massepotential. Links daneben erkennt man die drei zugehörigen Pull-Up-Widerstände in sehr kleiner Bauform (sogenannte SMD-Komponenten).

Links neben den Lötstellen sind deutlich die drei Pull-Up-Widerstände zu erkennen, mit denen die Eingangspins auf `HIGH` gehalten werden (siehe Kapitel 5.2). Um einen bestimmten Pin auf LOW zu setzen muss man die entsprechenden zwei Lötpads (im obigen Bild jeweils übereinander gelegen) mit einem Stückchen aufgelöteten Draht überbrücken. Sein Potential liegt dann auf Masse-Niveau (`LOW`). Somit kann man zwischen 8 verschiedenen I²C-Adressen wählen und dadurch mehrere LCDs an einem I²C-Bus nutzen, ohne dass Adressen doppelt verwendet werden ("kollidieren").

I²C-Adresse	A0	A1	A2
32	LOW	LOW	LOW
33	HIGH	LOW	LOW
34	LOW	HIGH	LOW
35	HIGH	HIGH	LOW
36	LOW	LOW	HIGH
37	HIGH	LOW	HIGH
38	LOW	HIGH	HIGH
39 (Standard)	HIGH	HIGH	HIGH

Tabelle 7.1 Adresskonfigurationen des I²C-Backpacks. Die Standardadresse ist 39, da bei unveränderten Lötpads A0, A1 und A2 über Pull-Up-Widerstände auf HIGH gesetzt werden.

Wir benutzen im Beispiel nur ein LCD und können daher die Standardadresse 39 beibehalten. Testen wir es nun mit unserem Arduino:

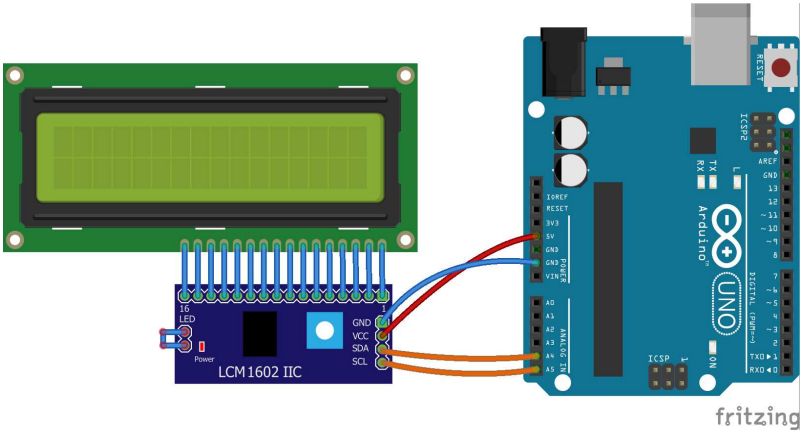


Abb. 7.14 Die Verdrahtung ist dank I²C-Backpack recht übersichtlich

Zur einfacheren Ansteuerung des Backpacks installieren wir die Bibliothek „LiquidCrystal_I2C.h“.

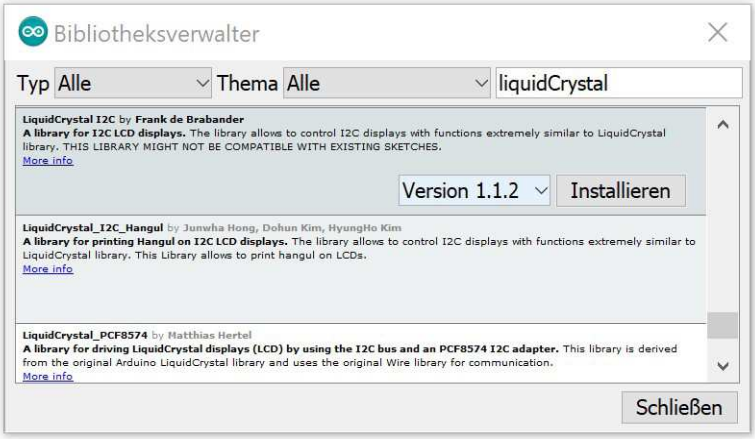


Abb. 7.15 Die Bibliothek „LiquidCrystal“ stellt Funktionen zur Ansteuerung des Displays über das Backpack bereit

Der folgende Sketch zeigt zunächst für 5 Sekunden einen Grußtext und danach eine im Sekundentakt hochzählende Zahl. Sollte bei Ihnen zunächst nichts im Display erscheinen, drehen Sie mit einem kleinen Schraubenzieher vorsichtig am Potentiometer des Backpacks, um den LCD-Kontrast zu verändern.

Abb. 7.16 Das Display zeigt unsere erste selbstprogrammierte Grußbotschaft

```
#include "Wire.h"
#include "LiquidCrystal_I2C.h"                                  // (1)

LiquidCrystal_I2C LCD(39, 16, 2);                               // (2)

void setup()
{
  LCD.init();                                                   // (3)
  LCD.clear();                                                  // (4)
  LCD.backlight();                                              // (5)
  LCD.setCursor(2, 0);                                          // (6)
  LCD.print("Hallo, Welt!");                                    // (7)
  LCD.setCursor(1, 1);
  LCD.print("Hallo, Arduino!");
  delay(5000);
  LCD.clear();                                                  // (8)
}

void loop()
{
  LCD.setCursor(2, 0);
  LCD.print(millis()/1000);                                     // (9)
  delay(1000);
}
```

1. Die Wire.h-Bibliothek zur Nutzung der I²C-Schnittstelle kennen wir bereits, hier hilft uns nun zusätzlich die LiquidCrystal_ I2C.h-Bibliothek bei der Kommunikation mit dem LCD-Backpack.

2. Da auch diese Bibliothek objektbasiert arbeitet, erzeugen wir ein Objekt mit dem Namen `LCD` aus der Klasse `LiquidCrystal_I2C` – es repräsentiert nun unser Display. Bei der Initialisierung erwartet die Bibliothek drei Argumente: Die I²C-Adresse, die Zeichenzahl pro Zeile und die Zeilenzahl.

3. Im `setup()` wird das LCD zunächst per `init()` gestartet und

4. der Displayinhalt gelöscht (`clear()`) sowie

5. die Hintergrundbeleuchtung eingeschaltet. Zum Ausschalten verwendet man übrigens die Funktion `LCD.noBacklight()`, ebenfalls ohne Argumente.

6. Die Textausgabe auf der Anzeige erfolgt über einen Cursor – so wie in einem Textverarbeitungsprogramm. Dieser wird über die Num-

mer der Spalte und Zeile positioniert, wobei das Zählen stets bei 0 beginnt. Wir setzen der Cursor hier also an die dritte Stelle (2) in der ersten Zeile (0).

7. Die eigentliche Ausgabe des Textes ähnelt nun stark der bereits bekannten Funktion `Serial.print()` zur Ausgabe auf dem Computermonitor. Als Argument kann einfach der gewünschte Text übergeben werden.
8. Nach einer Pause von 5 Sekunden wird der Displayinhalt wieder gelöscht.
9. In der Hauptschleife des Programmes wird nun der Cursor immer wieder an die gleiche Stelle gesetzt und die Anzahl der vergangenen Sekunden seit Programmstart ausgegeben. Dazu wird der Rückgabewert der Funktion `millis()` (also die Zahl der vergangenen Millisekunden seit Programmstart) ganzzahlig durch 1000 geteilt.

7.6 OLED-Display

Seit etwa 15 Jahren werden LCDs in vielen Anwendungsbereich von OLED-Displays (*organic LED-Displays*) abgelöst. In diesen können die Pixel, ähnlich wie bei einer herkömmlichen LED, selbst Licht erzeugen – es ist also keine Hintergrundbeleuchtung mehr notwendig. Das sorgt für schärfere Kontraste und bessere Lesbarkeit.

Auch hier gibt es einen Controller, der sich als quasi-Standard durchgesetzt hat: Der SSD1306. Im Handel sind sehr preiswert vorgefertigte Module mit dem Controller und einem kleinen OLED-Display erhältlich. Die Ansteuerung erfolgt wieder komfortabel per I^2C. Im folgenden Beispiel verwenden wir ein solches Display mit 128 Pixeln Breite und 64 Pixeln Höhe. Der Hersteller *Adafruit* stellt eine leistungsstarke Bibliothek dafür bereit.

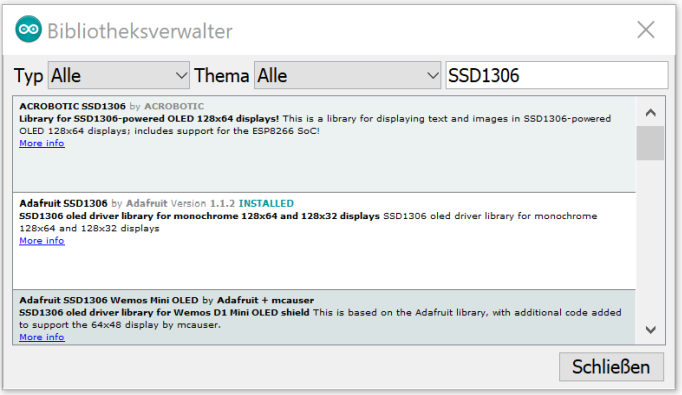


Abb. 7.17 Die Bibliothek „Adafruit SSD1306“ kümmert sich um die Steuerung des OLED-Controllers

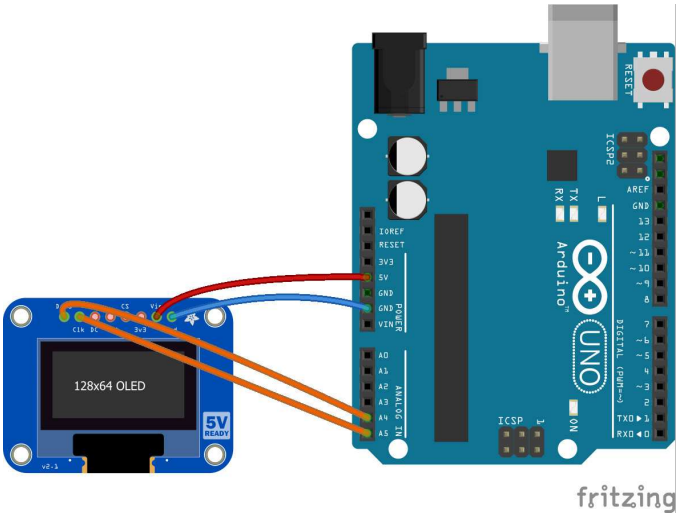


Abb. 7.18 Die Verdrahtung ist simpel, da für den I²C-Bus zwei Datenleitungen genügen. Dabei wird A4 mit dem Anschluss DAT beziehungsweise SDA verbunden, A5 führt das Taktsignal CLK beziehungsweise SCL

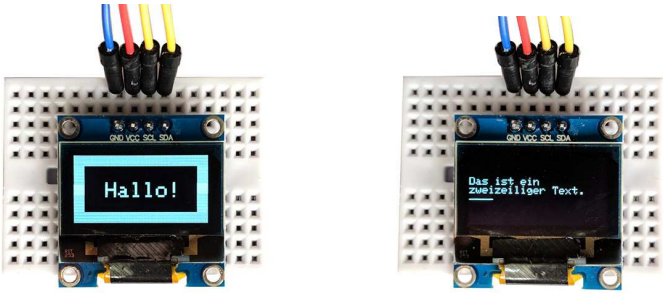


Abb. 7.19 Das OLED-Display ist deutlich kontraststärker als das LCD, da es keine Hintergrundbeleuchtung benötigt

Das Display gibt im Wechsel einen eingerahmten Gruß „Hallo!“ sowie einen kleinen Beispieltext mit Animation aus, wenn wir folgenden Sketch benutzen:

```
#include "Wire.h"
#include "Adafruit_SSD1306.h"                                   // (1)

Adafruit_SSD1306 display(4);                                    // (2)

void setup()
{
  display.begin(SSD1306_SWITCHCAPVCC, 60);                      // (3)
}

void loop()
{
  display.clearDisplay();                                       // (4)

  display.fillRect(0, 0, 127, 63, WHITE);                       // (5)
  display.fillRect(10, 10, 107, 43, BLACK);                     // (6)

  display.setTextSize(2);                                       // (7)
  display.setTextColor(WHITE);                                  // (8)
  display.setCursor(30,25);                                     // (9)
  display.print("Hallo!");                                     // (10)
  display.display();                                            // (11)
  delay(5000);

```

```
25    display.clearDisplay();
26    display.setTextSize(1);
27    display.setCursor(0,20);
28    display.println("Das ist ein");                          // (12)
29    display.println("zweizeiliger Text.");
30    display.display();
31
32    for(byte i = 0; i < 128; i++)                           // (13)
33    {
34      display.drawLine(0,40,i,40, WHITE);                   // (14)
35      display.display();
36      delay(30);
37    }
38  }
```

1. Wir benötigen die bekannte I²C-Bibliothek Wire.h sowie die eben heruntergeladene Bibliothek Adafruit_SSD1306.h
2. Wir erstellen das Objekt `display` aus der Klasse `Adafruit_SSD1306`, welches im Folgenden unser Display repräsentiert. Die Bibliothek erwartet als Argument einen Pin welcher als Reset für das OLED-Display fungiert. Durch den Anschluss per I²C brauchen wir diesen aber gar nicht. Dennoch muss zwingend ein Argument angegeben werden, wir nutzen willkürlich die 4.
3. Mit der Funktion `begin()` wird das Display initialisiert und gestartet. Als erstes Argument ist der Typ anzugeben, dies ist üblicherweise SSD1306_SWITCHCAPVCC. (Die Bezeichnung dieser Konstanten wird von der Bibliothek vorgegeben.) Das zweite Argument ist die I²C-Adresse. Der hier verwendete Displaytyp nutzt standardmäßig die Adresse 60, diese kann aber wieder per Lötbrücke geringfügig verändert werden.
4. In der Hauptprogrammschleife wird der Displayinhalt nun zunächst gelöscht,
5. bevor mittels des Befehls `fillRect()` ein ausgefülltes weißes Rechteck gezeichnet wird. Als Argumente sind zunächst die X- und Y-Position (jeweils beginnend oben links mit 0) sowie die Breite und Höhe in Pixeln anzugeben. Das letzte Argument repräsentiert die Farbe, für welche die Bibliothek bereits Konstantennamen angelegt

hat. Bei einem einfarbigen Display wie in unserem Fall gibt es natürlich nur zwei Möglichkeiten: `BLACK` und `WHITE`.

6. Der weiße Rahmen wird erzeugt, indem in das weiße Rechteck ein etwas kleineres schwarzes Rechteck gezeichnet wird, welches dieses also überdeckt.
7. In den nächsten Zeilen werden die Schriftgröße (wobei 1 für die kleinste Schrift steht) und
8. Schriftfarbe gewählt.
9. In gleicher Weise wie beim LCD wird nun der Textcursor gesetzt, allerdings erfolgt die Positionsangabe diesmal in Pixeln, hier also das 31. Pixel von links und das 26. Pixel von oben.
10. Die Textausgabe übernimmt wieder eine Funktion mit dem Namen `print()`.
11. Alle bisherigen Befehle haben den Displayinhalt nur in einen Zwischenspeicher geschrieben. Erst die Funktion `display()` überträgt ihn auf das eigentliche OLED-Display.
12. Nach erneutem Löschen wird ein diesmal etwas kleinerer Text auf das Display geschrieben. Analog zur Nutzung bei der Ausgabe über die serielle Schnittstelle gibt es auch hier wieder eine Funktion `println()`, welche nach der Ausgabe einen Zeilenumbruch vornimmt.
13. Eine `for`-Schleife sorgt nun für eine kleine Animation:
14. Der Befehl `drawLine()` zeichnet eine Linie von den ersten beiden übergebenen Koordinaten (x und y) zu den folgenden beiden Koordinaten. Das fünfte Argument ist wiederum die Farbe. Da diese Funktion bei jedem Schleifendurchlauf aufgerufen wird, und die Variable `i` dabei immer ansteigt, wird die Linie immer länger.

Die Bibliothek bietet noch zahlreiche weitere Möglichkeiten. Da sie, wie die meisten anderen Bibliotheken auch, eigene Beispielsketche mitbringt, können Sie unkompliziert etwas herumexperimentieren. Klicken Sie einfach auf *Datei -> Beispiele -> Adafruit SSD1306 -> ssd1306_128x64_i2c*.

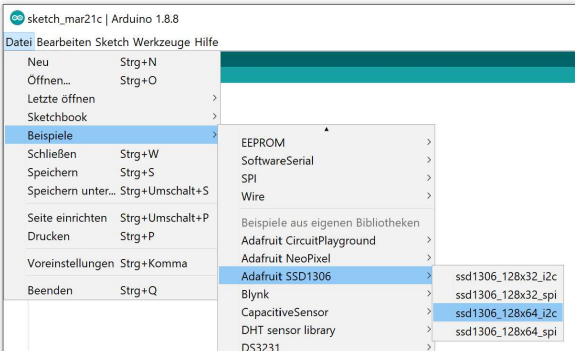


Abb. 7.20 Viele Bibliotheken bringen auch gleich Beispiele zum Testen mit

7.7 Adressierbare LEDs

Eine weitere interessante Möglichkeit zur Datenausgabe, aber auch für Kunstprojekte, sind adressierbare LEDs. Auch hier gibt es diverse Arten, wir widmen uns beispielhaft den LEDs vom Typ WS2812B, welche sich in der Bastlergemeinde großer Beliebtheit erfreuen.

Abb. 7.21 Beispiele für adressierbare LEDs - als Ring, Matrix und einzeln. Es sind noch zahlreiche weitere Ausführungen erhältlich.

Es gibt sie in verschiedensten Formen zu kaufen: einzeln, in Gitterform auf einer Platine vormontiert, als flexibler Klebestreifen, als Kreis und

sogar als Weihnachtsbaum-Lichterkette. Der riesige Vorteil dieser RGB-LEDs liegt darin, dass sie einzeln ansteuerbar sind – obwohl sie auf den ersten Blick in einer Art Reihenschaltung verdrahtet wurden und es nur eine Leitung für ein Steuersignal gibt.

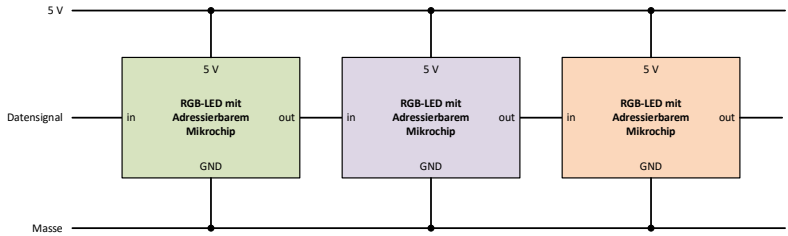


Abb. 7.22 Adressierbare LEDs können beliebig kaskadiert werden und sind dennoch einzeln ansteuerbar

Der Trick liegt darin, dass über die Steuerleitung ein serielles Datensignal (ähnlich der seriellen Schnittstelle, jedoch mit etwas abweichender Darstellung der einzelnen Bits) gesendet wird. Es besteht aus einem Byte (8 Bit) für die Helligkeit des grünen Farbanteils, einem Byte für den roten Anteil und abschließend einem Byte für den Helligkeitswert der blauen LED. Aus diesen drei Werten zwischen 0 und 255 können wieder beliebige Farben gemischt werden, genau wie bei der RGB-LED in Kapitel 7.2.

In der WS2812B-LED ist neben der eigentlichen Leuchtdiode auch ein kleiner Mikrochip, der diese 24 Bit empfängt und auswertet. Er verfügt neben dem Eingangspin auch über einen Ausgangspin, welcher mit der nächsten adressierbaren LED verbunden ist. Während des Empfangs der oben beschriebenen 3 Byte für den gewünschten Farbwert wird an diesem Ausgang jedoch kein Signal ausgegeben. Erst wenn daraufhin noch weitere Bits folgen, werden diese durch den Chip ignoriert und einfach weitergeleitet. Gleichzeitig achtet der Chip weiterhin auf das Datensignal – gibt es darin eine Pause von mehr als 50 Mikrosekunden, wird die Übertragung als beendet angesehen und die zuletzt empfangenen Helligkeitswerte werden auf der RGB-LED ausgegeben. Diese Farbe wird beibehalten, bis ein neues Signal empfangen wurde.

Um nun also alle adressierbaren LEDs mit einer individuellen Farbe zu versorgen, sendet man einfach ein serielles Signal mit den jeweiligen Farbwerten direkt nacheinander in der Reihenfolge ihrer Verdrahtung.

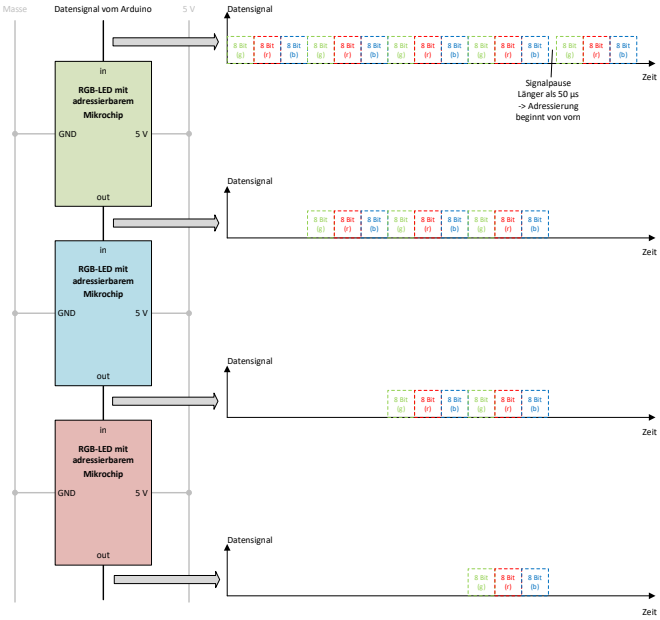


Abb. 7.23 Das Datensignal verändert sich nach jedem LED-Modul, da die ersten 3 Byte stets „geschluckt“ werden. Somit wird eine Adressierung entlang des Verdrahtungsweges möglich.

Durch dies Kaskadierung sind nahezu beliebig große Displays möglich. Die Übertragung von 24 Bits dauert nur 30 Mikrosekunden. In einer Sekunde können somit rund 33.000 LEDs mit Daten bespielt werden – oder man lässt 330 LEDs bis zu 100-mal pro Sekunde die Farbe wechseln. LED-Anzahl und Bildfrequenz begrenzen sich also gegenseitig, in den meisten praktischen Anwendungen wird aber keine dieser Grenzen auch nur annähernd relevant.

Viel wichtiger ist es, die Stromversorgung im Auge zu behalten. Da jede der RGB-LEDs bei der Anzeige von hellem Weiß rund 15 mA benötigt,

sollten Sie für eine 8x8-Matrix mit 64 LEDs sicherstellen, dass die verwendete Stromversorgung (zum Beispiel USB-Port des Computers) 1 Ampere Strom liefern kann. Bei noch größeren Anordnungen ist es ratsam, eine externe Spannungsversorgung zu verwenden (Näheres dazu in Kapitel 12.3). Für das folgende Beispiel mit einer vorgefertigten 8x8-Matrix aus 64 adressierbaren LEDs genügt gerade noch die USB-Versorgung.

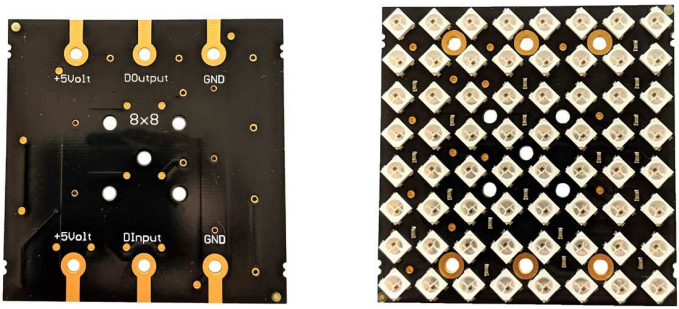


Abb. 7.24 Die adressierbaren LEDs in dieser Matrix sind zeilenweise im Zickzack verbunden. Der Eingang der ersten und der Ausgang der letzten LED sind als Lötanschlüsse ausgeführt.

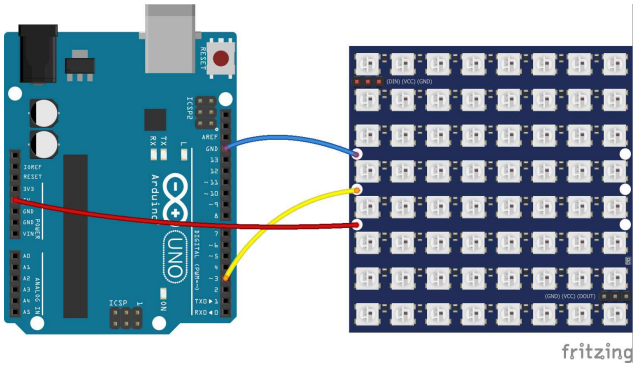


Abb. 7.25 Die Verdrahtung ist entsprechend einfach, als Datenpin eignet sich jeder Ausgangspin

Wir nutzen zur Ansteuerung die Bibliothek FastLED.h.

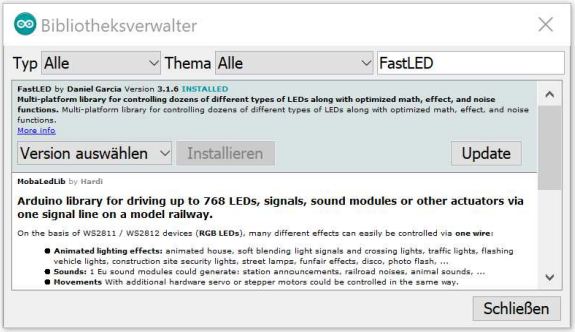


Abb. 7.26 Die Bibliothek FastLED von Daniel Garcia bietet zahlreiche Funktionen für adressierbare Leuchtdioden.

Der folgende Beispielsketch zieht einen Lichtschweif über die LEDs, welcher allmählich die Farbe wechselt.

```
#include "FastLED.h"

#define NUM_LEDS 64                                          // (1)
#define DATA_PIN 3

CRGB leds[NUM_LEDS];                                         // (2)
byte Farbe = 0;
byte Position = 0;

void setup() {
  FastLED.addLeds<WS2812B, DATA_PIN, GRB>(leds, NUM_LEDS); // (3)
}

void loop() {
  leds[Position++] = CHSV(Farbe++, 255, 128);               // (4)
  FastLED.show();                                            // (5)
  delay(50);
  fadeToBlackBy(leds, NUM_LEDS, 20);                         // (6)
  if(Position > NUM_LEDS-1)
    Position = 0;
}
```

1. Die Gesamtzahl der verbundenen LEDs sowie die Nummer des verwendeten Ausgangspins werden zur besseren Lesbarkeit als Präprozessor-Konstanten definiert. Dadurch lässt sich der Sketch auch später einfacher anpassen, wenn zum Beispiel eine größere LED-Matrix verwendet wird.

2. Wir erzeugen ein Array vom Objekt-Datentyp `CRGB`, dieser wird von der Bibliothek zur Verfügung gestellt. Dieser Datentyp kann RGB-Farbwerte speichern. Das Feld `leds` repräsentiert die gesamte LED-Matrix mit jedem einzelnen Farbwert.

3. Die Funktion `addLeds()` teilt der Bibliothek mit, dass wir adressierbare LEDs verwenden möchten, und zwar vom Typ WS2812B, am Pin 3, mit der Farbreihenfolge grün-rot-blau. Als Argument in den Klammern wird noch der Name des Arrays für die Farbwerte sowie die Anzahl seiner Elemente übergeben. Die etwas merkwürdige Notation mit den spitzen Klammern rührt daher, dass die Funktion innerhalb der Bibliothek in der Form eines sogenannten Templates angelegt ist. Dies macht die Bibliothek flexibler für verschiedene Anwendungen. Eine genaue Betrachtung würde hier den Rahmen jedoch sprengen, verstehen Sie es hier einfach als Bestandteil des Funktionsnamens.

4. In der Hauptschleife wird nun einem Element des Arrays `leds[]` mit Hilfe der Funktion `CSHV()` eine konkrete Farbe zugewiesen. Diese Funktion erwartet als ersten Parameter eine Zahl zwischen 0 und 255, welche die Farbe (wie auf einem Farbrad) repräsentiert. Das zweite Argument ist die Sättigung, das Dritte die Helligkeit. Es handelt sich immer um `byte`-Werte. Dabei werden sowohl die Position als auch der Farbe bei jedem Aufruf um 1 erhöht. Die Farbe fängt dabei nach 255 Schritten von allein wieder bei 0 an, da es sich um einen `byte`-Wert handelt. Die Position stellen wir über eine `if`-Bedingung am Ende der Schleife selbst wieder an den Anfang, wenn alle LEDs durchlaufen sind. Natürlich wäre auch eine `for`-Schleife passend. Alternativ könnte man auch über die ähnliche Funktion `CRGB()` die einzelnen Helligkeitswerte für Rot, Grün und Blau angeben. Da wir

aber kontinuierlich durch alle Farben wechseln möchten, ist die obige Vorgehensweise hier vorteilhafter.

5. Die Funktion `show()` schickt nun die im `leds[]`-Array hinterlegten Farbwerte über den Ausgangspin an die adressierbaren LEDs.
6. Diese Funktion wird ebenfalls von der Bibliothek bereitgestellt. Sie reduziert alle in `leds[]` hinterlegten Helligkeitswerte um ein gewisses Maß, welches hier willkürlich mit 20 gewählt wurde. Das sorgt für ein allmähliches Verlöschen bei jedem weiteren Durchlauf und ist damit für den Schweif-Effekt verantwortlich.

Auch hier bringt die Bibliothek wieder zahlreiche Beispiele mit. Eine andere Bibliothek zur Steuerung adressierbarer LEDs heißt Adafruit_NeoPixel.h – sie bietet weitere Funktionen, die für kunstvolle LED-Projekte interessant sein könnten. Experimentieren Sie doch ein wenig mit den Beispielen, wenn Sie noch mehr Eindrücke von den Möglichkeiten dieser Technik erlangen möchten.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 8

Praxisprojekt: Stoppuhr mit OLED-Display

8.1 Idee

Als nächstes praktisches Beispiel programmieren wir eine Stoppuhr mit OLED-Display. Um die Bedienung möglichst einfach zu halten, soll ein einziger Taster genügen. Der erste Druck startet die Zeitmessung, der zweite stoppt die Uhr und das Ergebnis wird auf Tausendstelsekunden genau angezeigt.

Für die Messung können wir uns die bereits bekannte Funktion `millis()` zu Nutze machen. Ihr Rückgabewert ist die Anzahl der vergangenen Millisekunden (Tausendstelsekunden) seit Programmstart. Diesen Wert können wir wie einen Zeitstempel benutzen. Speichern wir den Rückgabewert zum Zeitpunkt A und zum Zeitpunkt B, so können wir durch Differenzbildung die zwischen A und B vergangene Zeit in Millisekunden errechnen. Dieser Wert lässt sich entsprechend aufbereiten (in Stunden, Minuten und Sekunden umrechnen) und ausgeben.

8.2 Versuchsaufbau

Der sehr einfache Versuchsaufbau erübrigt einen separaten Stromlaufplan. Wir benötigen:

- 1 kleines Breadboard
- 1 Arduino UNO
- 1 OLED-Display (SSD1306, 128x64 Pixel, I^2C-Anschluss)
- 1 Taster

- Verbindungsdrähte

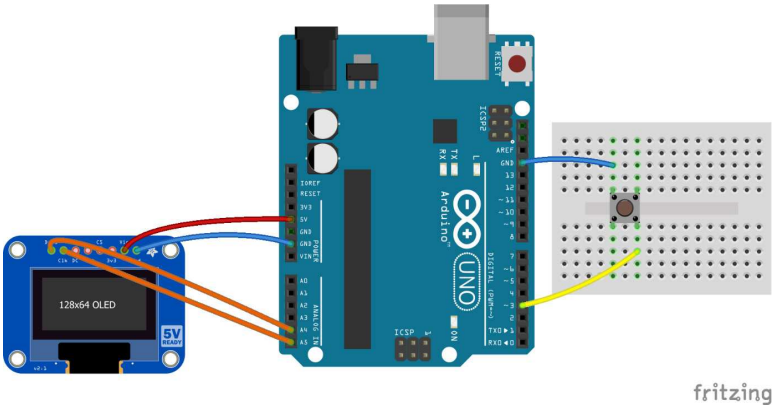


Abb. 8.1 Alternativ kann das Display auch mit auf das Breadboard gesteckt werden

8.3 Programmcode

Im folgenden Sketch sind kurze Kommentare wieder direkt im Programmcode eingefügt:

```
#define TASTE 3
#define MILLISPROSTUNDE 3600000      // Konstanten zur einfacheren
                                     // Umrechnung
#define MILLISPROMINUTE 60000        // von Millisekunden in
                                     // andere Einheiten

#include "Wire.h"
#include "Adafruit_SSD1306.h"        // Bibliothek für das
                                     // OLED-Display

Adafruit_SSD1306 display(1);

long Timer = 0;
                              // nachfolgend: Hilfsvariablen
                              // zur Zeitmessung
long Startzeit = 0;           // Zeitpunkt A
long Stoppzeit = 0;           // Zeitpunkt B
long Zeitdifferenz = 0;       // abgelaufene Zeit in Millisekunden
```

```
19
20 byte Stunden; byte Minuten; byte Sekunden; int Tausendstel;
21
22 boolean aktiv = false;      // Zeitmessung aktiv (true) oder
23                             // angehalten (false)
24
25 void setup()
26 {
27   pinMode(TASTE, INPUT_PULLUP);
28
29   display.begin(SSD1306_SWITCHCAPVCC, 60);  // Display-
30                                             // Initialisierung
31
32   display.clearDisplay();
33   display.setTextColor(WHITE);              // Farbe weiß
34                                             // auswählen
35 }
36
37 void loop()
38 {
39   if(millis() > Timer)                                    // (1)
40   {
41     if(aktiv)                                             // (2)
42       Zeitdifferenz = millis()-Startzeit;
43     else
44       Zeitdifferenz = Stoppzeit-Startzeit;
45
46     Stunden = Zeitdifferenz / MILLISPROSTUNDE;            // (3)
47     Minuten = (Zeitdifferenz % MILLISPROSTUNDE) / MILLISPROMINUTE;
48     Sekunden = (Zeitdifferenz % MILLISPROMINUTE) / 1000;
49     Tausendstel = Zeitdifferenz % 1000;
50
51     display.clearDisplay();
52     display.setTextSize(2);
53     display.setCursor(17,2);
54     display.println("Stoppuhr");              // Überschrift
55     display.drawLine(17,20,  110,20, WHITE);  // Unterstreichung
56
57     display.setTextSize(1);
58     display.setCursor(40,30);
59
60     if(aktiv)
61       display.println("aktiv!");
62     else
63       display.println("Resultat:");
64
65     display.setCursor(30,50);
66     if(Stunden < 10)                                      // (4)
```

```
67        display.print("0");
68      display.print(Stunden);
69      display.print(":");
70      if(Minuten < 10)
71        display.print("0");
72      display.print(Minuten);
73      display.print(":");
74      if(Sekunden < 10)
75        display.print("0");
76      display.print(Sekunden);
77
78      if(!aktiv)              // Tausendstel nur anzeigen, wenn
79      {                       // Zeitnahme angehalten wurde.
80        display.print(".");
81        if(Tausendstel < 100)
82          display.print("0");
83        if(Tausendstel < 10)
84          display.print("0");
85        display.print(Tausendstel);
86      }
87      display.display();
88
89      Timer = millis() + 1000;  // nächste Displayaktualisierung
90                                // in 1 Sek.
91    }
92
93    if(!digitalRead(TASTE))     // Wird gerade die Taste gedrückt?
94    {
95      if(aktiv)
96      {
97        Stoppzeit = millis();
98        aktiv = false;
99        Timer = 0;
100     }
101     else
102     {
103       Startzeit = millis();
104       aktiv = true;
105       Timer = 0;
106     }
107
108     while(!digitalRead(TASTE));     // warten, bis Taste wieder
109                                     // losgelassen wurde
110   }
111
112 }
```

1. Wie schon in Kapitel 6.4 nutzen wir zur Verzögerung auch hier eine Variable als `Timer`, statt per `delay()` das Programm anzuhalten. Die nachfolgenden Anweisungen sorgen dafür, dass das Display aktualisiert wird. Es genügt, wenn dies einmal pro Sekunde geschieht. Würde es bei jedem `loop()`-Durchlauf erneut aktualisiert, sähe man ein deutliches Flackern. Die `Timer`-Variable wird daher immer am Ende der Displayaktualisierung auf einen Zahlenwert `millis()+1000` gesetzt. Die Funktion `millis()` allein erreicht diesen Wert erst eine Sekunde später, so dass der `if`-Block bei allen weiteren Programmdurchläufen innerhalb der nächsten Sekunde übersprungen wird.
2. `Zeitdifferenz` ist die Zeit, welche angezeigt wird. Bei aktiver (laufender) Stoppuhr ist es die Differenz zwischen der aktuellen Zeit und der `Startzeit`. Wurde die Zeitmessung bereits angehalten, ist es die Differenz zwischen `Stoppzeit` und `Startzeit`.
3. Die ermittelte Zeitdifferenz wird durch ganzzahlige Divisionen und Modulodivisionen (Divisionsrest, siehe Kapitel 4.11) in Stunden, Minuten, Sekunden und Tausendstel zerlegt. Darauf folgt die Textausgabe auf dem Display.
4. An mehreren Stellen im Sketch werden führende Nullen ausgegeben, wenn der entsprechende Wert kleiner als 10 ist. Die übliche Schreibweise einer Zeit ist schließlich 10:04:09 und nicht 10:4:9.

Es sei noch erwähnt, dass dieser Sketch nur deshalb zuverlässig funktioniert, weil der Mikrocontroller die `loop()`-Schleife mehrere Zehntausend Male pro Sekunde durchläuft und somit auch quasi ständig der Zustand der Taste per `if(!digitalRead(TASTE))` abgefragt wird. Enthielte das Programm Befehle, welche die Abarbeitung deutlich verzögern (zum Beispiel `delay()` oder umfangreiche Bildschirmausgaben per `Serial.print()`) könnte es sein, dass dadurch ein Tastendruck nicht erkannt wird. In der professionellen Programmierung behilft man sich in so einem Fall mit Interrupts, welche das Programm auf Hardware-Ebene unterbrechen können.

8.4 Resultat

Unsere selbstgebaute Stoppuhr ist fertig. Die Ausgabe auf dem Display sieht beispielhaft so aus:

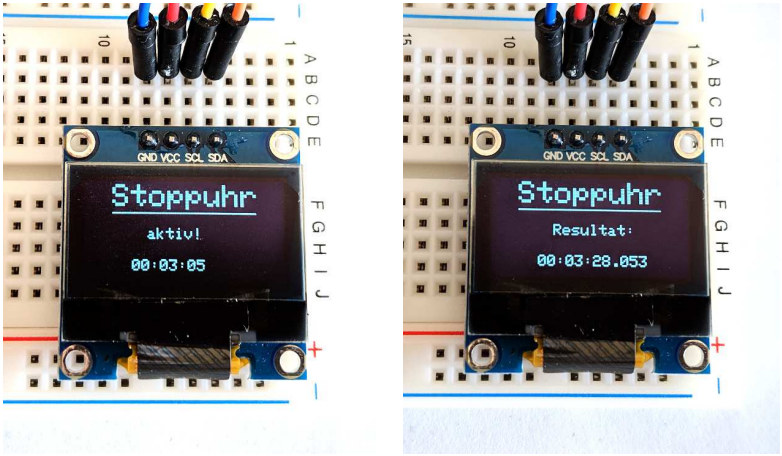


Abb. 8.2 Nach dem Ende der Zeitnahme werden auch die Nachkommastellen angezeigt

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 9

Sensoren und Eingabegeräte

Um sinnvoll mit der Umwelt zu interagieren, sind neben den Möglichkeiten zur Datenausgabe natürlich auch Eingabekomponenten notwendig. Beim Blick auf die digitalen und analogen Eingänge haben wir bereits Taster und Potentiometer kennengelernt. In gleicher Weise können natürlich auch Türkontakte, Neigungsschalter und vieles mehr abgefragt werden. Blicken wir nun auf einige weitere Möglichkeiten der Datenerfassung.

9.1 Folientastatur

Eine preiswerte und schnell implementierbare Möglichkeit, Nutzereingaben zu erfassen, sind Folientastaturen. Sie funktionieren wie Taster, die gitterartig verschaltet sind – auf ähnliche Weise wie die LED-Matrix in Kapitel 7.4.

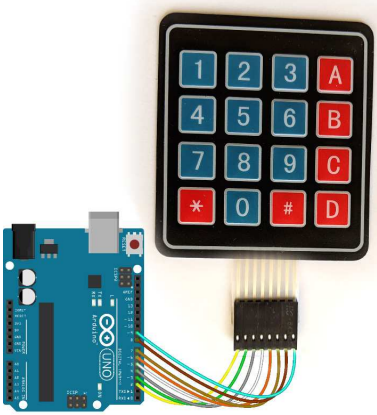


Abb. 9.1 Folientastaturen kommen oft zur Eingabe von Zahlenwerten zum Einsatz

Wir nutzen für das folgende Beispiel ein Standardmodell und greifen wieder auf die serielle Übertragung zum Computer zurück, um die Daten auszugeben. Dabei hilft uns die Bibliothek Keypad.h.

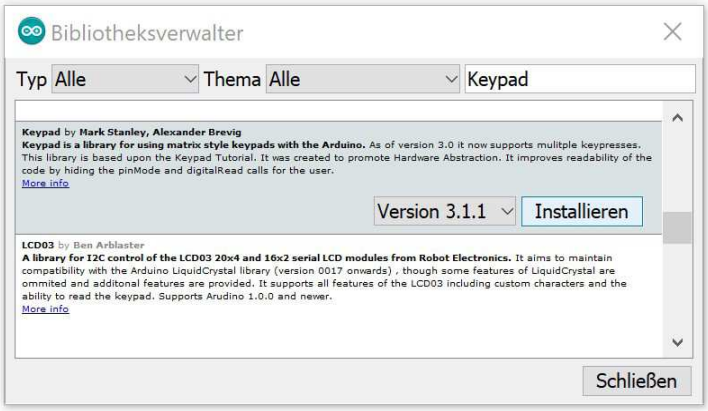


Abb. 9.2 Die Bibliothek Keypad übernimmt für uns das Auslesen der Tasten

```
#include "Keypad.h"

#define ZEILENZAHL 4
#define SPALTENZAHL 4

char Tasten[ZEILENZAHL][SPALTENZAHL] = {                    // (1)
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
byte ZeilenPins[ZEILENZAHL] = {2,3,4,5};                    // (2)
byte SpaltenPins[ZEILENZAHL] = {6,7,8,9};

Keypad Tastatur = Keypad(makeKeymap(Tasten), ZeilenPins,
SpaltenPins, ZEILENZAHL, SPALTENZAHL);                      // (3)

void setup(){
  Serial.begin(9600);                                       // (4)
}

```

```
22 void loop(){
23   char Eingabe = Tastatur.getKey();                          // (5)
24
25   if(Eingabe){                                               // (6)
26     Serial.print("Taste wurde gedrückt: ");
27     Serial.println(Eingabe);
28   }
29 }
```

1. Wir definieren ein 2-dimensionales Array mit dem Namen `Tasten` vom Typ `char`, in dem wir die Zuordnung der Tasten hinterlegen.
2. Wir definieren weitere Arrays, welche die Pinnummern der verwendeten Anschlüsse angeben.
3. Wir erzeugen das Objekt `Tastatur` aus der Klasse `Keypad`, wobei wir gemäß der Dokumentation der `Keypad`-Bibliothek die bereits erwähnten Arrays sowie die Anzahl der Zeilen und Spalten übergeben. Das Zuordnungsfeld `Tasten` wird zuvor mittels der Funktion `makeKeymap()` in ein anderes Format gewandelt, dies ergibt sich ebenfalls aus der Dokumentation der Bibliothek.
4. Zur Ausgabe nutzen wir einfacherweise wieder die bereits bekannte serielle Schnittstelle.
5. In der Hauptschleife wird nun die Funktion `getKey()` aufgerufen. Wurde eine Taste gedrückt, gibt sie den Wert zurück, welcher dieser Taste im Array `Tasten` zugeordnet wurde. Ansonsten ist der Rückgabewert 0.
6. Die verkürzte Bedingung dieser `if`-Struktur kann wie `(Eingabe != 0)` verstanden werden. In diesem Beispiel ist besonders hervorzuheben, dass damit nicht die „0“ aus dem Tastenfeld gemeint ist, sondern der Zahlenwert 0 – welcher in unserem Fall symbolisiert, dass überhaupt keine Taste gedrückt wurde. Möchte man einen Vergleich mit der Ziffer „0“ als Symbol durchführen, so ist sie in Hochkommata zu setzen. Der Ausdruck `(Eingabe == '0')` würde also prüfen, ob die „0“ auf dem Tastenfeld gedrückt wurde. In Wirklichkeit wird dann auch hier die „0“, wie jedes Zeichen vom Tastenfeld, gemäß Zeichentabelle im Hintergrund in eine völlig andere Zahl umgewandelt, welche als Variable vom Typ `char` abgespeichert werden kann.

Die Ausgabe dieses Programmes auf dem seriellen Monitor hat demnach folgendes Aussehen:

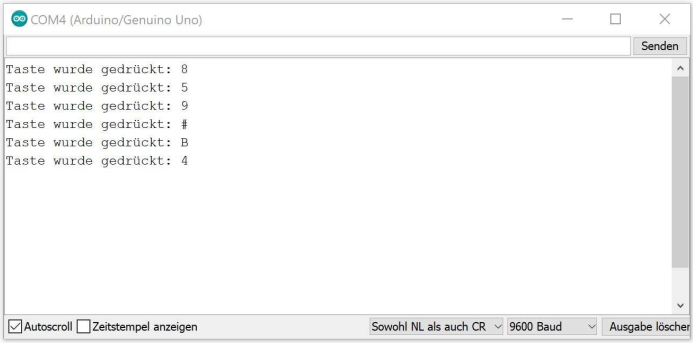


Abb. 9.3 beispielhafte Bildschirmausgabe

9.2 IR-Sensor/Fernbedienung

Einen komfortableren Weg der Dateneingabe stellt natürlich eine Infrarotfernbedienung dar. Zur Datenübertragung wird ein binäres Signal auf ein Infrarotsignal aufmoduliert, welches mit einer Frequenz von 36 kHz bis 40 kHz schwingt.

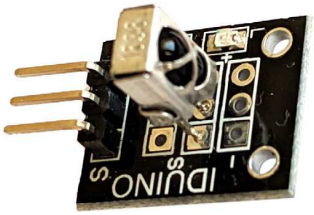


Abb. 9.4 Die komplette Elektronik dieses Infrarot-Empfängermoduls befindet sich in dem umgitterten schwarzen Baustein; dieser wird auch einzeln angeboten – die Platine erleichtert lediglich die Montage. Das abgebildete Modul ist unter der Bezeichnung KY-022 erhältlich.

Zu diesem Zweck gibt es günstige IR-Empfangsmodule, welche bereits die Filterung nach dieser Frequenz sowie eine Vorverstärkung des Signals übernehmen. Der Arduino muss dann nur noch die ankommenden Binärdaten auswerten, auch dafür gibt es natürlich wieder eine passende Bibliothek. Wir greifen im folgenden Fall auf IRremote.h zurück und verwenden ein Modul vom Typ KY-022.

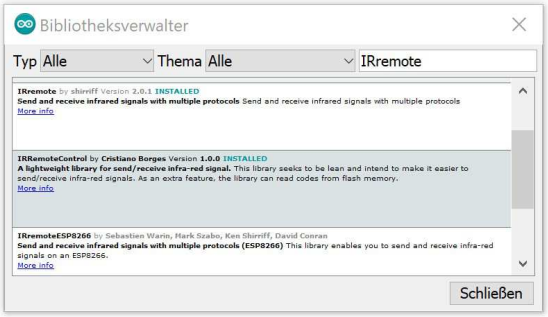


Abb. 9.5 Es gibt mehrere Bibliotheken für den Empfang von Infrarot-Signalen, wir entscheiden uns hier für IRemote

Die Verbindung ist sehr simpel, zum Test taugt jede handelsübliche Infrarotfernbedienung.

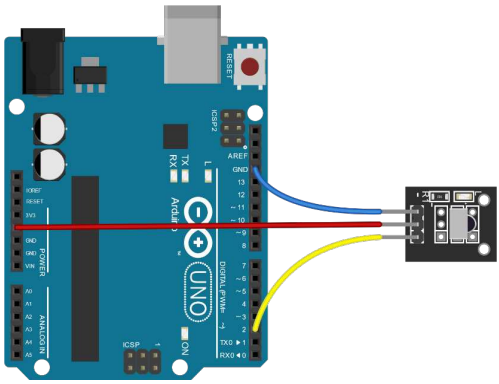


Abb. 9.6 Der Baustein kann direkt angeschlossen werden

```
#include "IRremote.h"

IRrecv Empfaenger(2);                                    // (1)
decode_results Daten;                                    // (2)

void setup()
{
  Serial.begin(9600);
  Empfaenger.enableIRIn();                               // (3)
}

void loop() {
  if(Empfaenger.decode(&Daten)) {                        // (4)
    Serial.print("Empfangene Daten: ");
    Serial.println(Daten.value, HEX);                    // (5)
    Empfaenger.resume();
  }
}
```

1. Wir erzeugen das Objekt `Empfaenger` aus der Klasse `IRrecv`, es repräsentiert unseren Empfänger. Als Argument übergeben wir die Nummer des Pins, an den er angeschlossen ist.

2. Das Objekt `Daten` aus der Klasse `decode_results` wird die empfangenen Daten aufnehmen.

3. Die Funktion `enableIRIn()` stellt die Empfangsbereitschaft her. Im Hintergrund wird dabei durch die Bibliothek ein Interrupt definiert. Dies ist hardwareseitig beim Arduino UNO nur für die digitalen Pins 2 und 3 möglich, daher sollte der Empfänger an einen dieser Pins angeschlossen werden. Nutzt man dafür einen anderen Pin, ist der Empfang möglicherweise unzuverlässig oder gar nicht möglich.

4. Hier wird geprüft, ob das Objekt `Daten` dekodierbare Informationen enthält. Ist dies nicht der Fall, gibt die Funktion `decode()` den Wert 0 zurück und die Bedingung ist nicht erfüllt. In jedem anderen Fall werden die Daten nun ausgegeben.

5. Die Darstellung solcher Binärdaten erfolgt üblicherweise hexadezimal, es könnte aber auch eine andere Darstellung gewählt werden. Lässt man das zweite Argument (`HEX`) weg, erfolgt die Anzeige im Dezimalsystem.

Das Resultat sieht in etwa so aus:

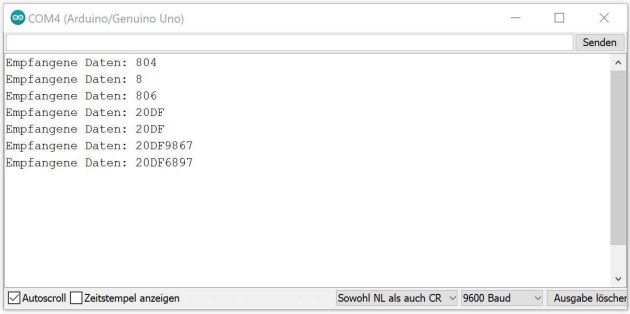


Abb. 9.7 beispielhafte Ausgabe der empfangenen Daten

Die angezeigten Werte können sich dabei je nach Typ der verwendeten Fernbedienung stark unterscheiden, da es dutzende verschiedene Codetabellen gibt, mit denen die Daten zugeordnet werden. Möchten Sie eine bestimmte Fernbedienung nutzen, können Sie die entsprechenden Codes am besten durch Ausprobieren herausfinden, denn Dokumentationen sucht man bei vielen Herstellern leider vergebens.

Wollen Sie Projekte mit einer neuen Fernbedienung ausstatten, können Sie auf günstige Universalfernbedienungen zurückgreifen. Diese lassen sich auf verschiedene Gerätecodes programmieren. Durch etwas Herumprobieren finden Sie schnell einen Code, welcher sich einfach im Arduino verarbeiten lässt. Etwas teurere Modelle können auch die Signale von einer anderen Fernbedienung „lernen“, indem sie das Signal der anderen Fernbedienung empfangen. Damit können Sie ein Projekt ohne Umprogrammieren vervielfältigen, selbst wenn die ursprüngliche Fernbedienung nicht mehr erhältlich ist.

Die verwendete Bibliothek nutzt einen internen Hardware-Timer des Arduino. Diese Timer erzeugen definierte Zeitsignale (wie ein Metronom) und werden für verschiedene Anwendungen genutzt, beispiels-

weise greifen die Funktion `millis()` und auch die PWM-Schaltungen darauf zurück. Es gibt drei dieser Timer, jedem von ihnen sind je zwei PWM-Ausgänge zugeordnet. Die IRremote.h verwendet Timer 2 und konfiguriert ihn geringfügig um. Dadurch darf die PWM-Funktion an den Pins 3 und 11 nicht mehr genutzt werden – sonst wird der Empfang von Infrarot-Befehlen nicht funktionieren.

Es sei noch die Möglichkeit erwähnt, mit einem Arduino selbst eine Fernbedienung zu bauen, in dem Falle benötigt man dann eine IR-LED zum Senden sowie natürlich passende Bibliotheken. Wir wollen zugunsten der anderen Beispiele hier jedoch nicht näher darauf eingehen, das Prinzip sollte durch die bisherigen Ausführungen klargeworden sein.

9.3 Fotowiderstand

Ein sehr simpler Weg, um auf optische Umgebungsreize zu reagieren, sind Fotowiderstände. Sie verhalten sich wie ein normaler ohmscher Widerstand, verändern ihren Widerstandswert jedoch abhängig vom Umgebungslicht. Dabei sinkt der Widerstand üblicherweise mit steigendem Lichteinfall.

Abb. 9.8 Charakteristisch für Fotowiderstände ist eine auffällige Fläche, welche lichtempfindlich reagiert

In der folgenden Schaltung nutzen wir den Fotowiderstand als Bestandteil eines Spannungsteilers und lesen die resultierende Spannung an einem analogen Pin ein.

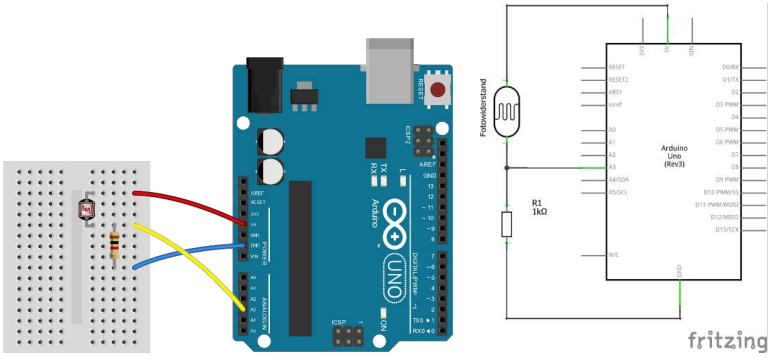


Abb. 9.9 Die Schaltung entspricht einem Spannungsteiler, allerdings mit einem konstanten und einem lichtempfindlichen Widerstand

Zur Demonstration genügt der folgende Sketch, welchen Sie mit Ihren bisherigen Kenntnissen ohne weitere Erklärung nachvollziehen können:


```
1  void setup()
2  {
3    Serial.begin(9600);
4  }
5
6  void loop()
7  {
8    Serial.print("Eingelesener Analogwert: ");
9    Serial.println(analogRead(3));
10   delay(1000);
11 }
```

Die Ausgabe am seriellen Monitor sieht erwartungsgemäß so aus, natürlich mit jeweils anderen Zahlenwerten:

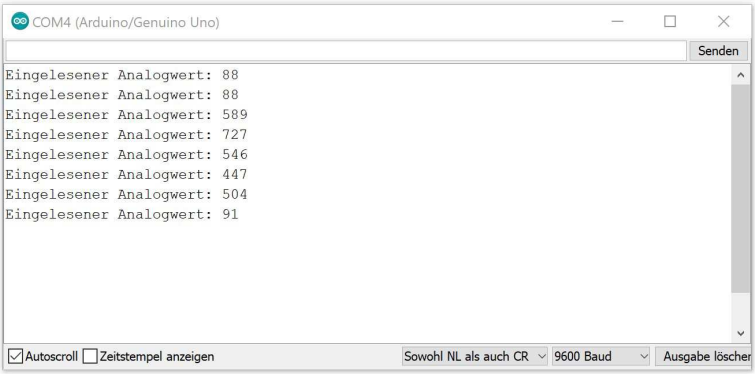


Abb. 9.10 Der eingelesene Analogwert verändert sich in Abhängigkeit des Lichteinfalls

9.4 Bewegungsmelder

Bewegungsmelder sind jedem aus dem Alltag bekannt. Es handelt sich dabei überwiegend um sogenannte PIR[1]-Sensoren. Dazu wird Infrarotstrahlung (Wärmestrahlung) der Umgebung aufgefangen und auf eine Sensorfläche geleitet, welche genau genommen nur auf Temperaturänderungen reagiert. Eine vorgelagerte Kunststoffkalotte dient der Bündelung des Infrarotlichtes aus einem weiten Winkelbereich und gibt dem Sensor das typische Aussehen.

Aufgrund des Messprinzips per Infrarotstrahlung eignet sich ein solches Modul besonders gut zur Detektion von Menschen und Tieren, denn deren Körperwärme strahlt Infrarotlicht ab, welches von der Kalotte auf den Sensor geleitet wird. Da die Umgebung in der Regel deutlich kühler ist, registriert der Sensor dies als örtlich begrenzte Temperaturänderung und löst daraufhin ein Signal aus.

1 *Pyroelectric Infrared Sensor* – Infrarot-Sensor auf Basis der Pyroelektrizität. Dies ist ein Effekt spezieller Halbleiterkristalle, welche eine Temperaturänderung in eine Spannung umwandeln können.

Gegenstände heben sich in ihrer Temperatur meist nicht von der Umgebung ab, sie lösen daher den Sensor nicht aus – selbst wenn sie sich bewegen (wie zum Beispiel im Wind wehende Äste). Dies gilt jedoch nicht unbegrenzt, so können sehr helle Lichtquellen im Überwachungsbereich des Sensors den Betrieb dennoch stören.

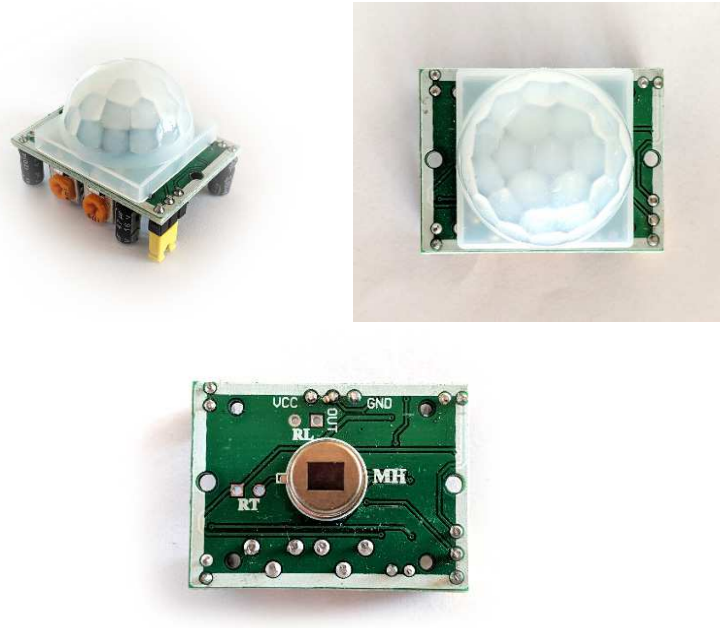


Abb. 9.11 Bewegungsmelder sind an der typischen weißen Kalotte erkennbar. Der eigentliche PIR-Sensor liegt darunter.

Ein günstiges Modell für den Heimanwenderbereich trägt die Bezeichnung HC-SR501. Dieses Modul verfügt über einen digitalen (binären) Ausgangspin, dessen Verhalten über einen Jumper (steckbaren Kontakt) auf der Platine gewählt werden kann. Im Ruhezustand ist das Ausgabesignal LOW.

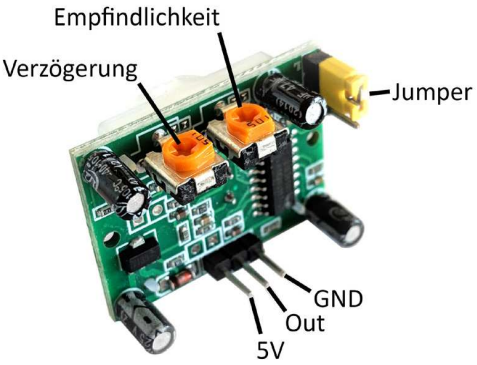


Abb. 9.12 Zwei Potentiometer und ein Jumper dienen der Konfiguration des HC-SR501-Moduls

In Standardeinstellung (Jumper am Platinenrand) springt das Signal bei der Erkennung einer Bewegung auf HIGH und bleibt für eine bestimmte Verzögerungszeit auf diesem Level. Danach fällt es zurück auf LOW und verbleibt weitere 2,5 Sekunden auf LOW. In dieser Zeit reagiert der Sensor auch nicht auf weitere Bewegungen. Nach dieser Zeit beginnt die Messung erneut.

In der alternativen Einstellung springt das Signal bei Auslösung des Melders auf HIGH. Registriert der Sensor nun weitere Bewegungen, bleibt das Signal HIGH. Erst wenn keine Bewegung mehr erkannt wird, beginnt die Verzögerungszeit. Danach fällt das Signal wieder auf LOW.

Die Verzögerungszeit selbst kann über ein Potentiometer geregelt werden, daneben befindet sich ebenfalls ein Verstellwiderstand für die Empfindlichkeit. Mit den Werkseinstellungen sollten sich bei einem Test üblicherweise schon brauchbare Ergebnisse erzielen lassen.

Wir wollen einen kleinen Versuchsaufbau mit dem Bewegungsmelder realisieren:

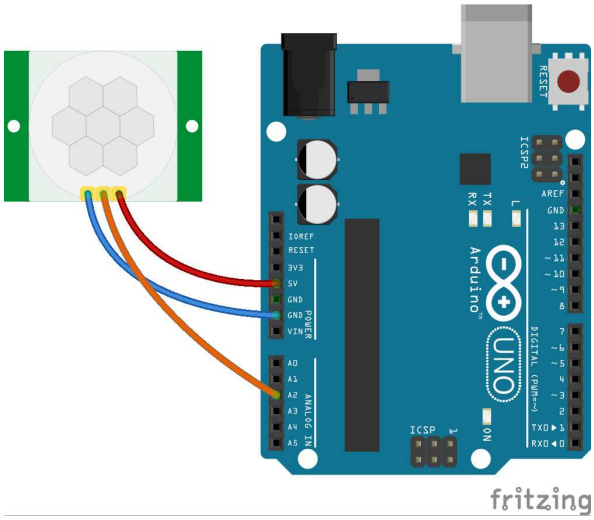


Abb. 9.13 Für das Signal eignet sich jeder Eingangspin. Hier wurde aufgrund der Kabellänge willkürlich ein Analogpin verwendet, der ja auch binär ausgelesen werden kann.

Die Onboard-LED des Arduino soll uns den aktuellen Status des Sensorsignals zeigen, außerdem soll bei jeder neuen Bewegungserkennung am seriellen Monitor eine Meldung mit dem aktuellen Zeitpunkt ausgegeben werden. Der Jumper des Moduls wird dazu idealerweise auf die alternative Position gesetzt, so dass bei einer anhaltenden Bewegung nur ein einziges durchgängiges HIGH-Signal entsteht.

```
#define SENSOR 16     // Analogpin A2 kann digital als Pin 16
                      // angesprochen werden

boolean letzterZustand = false;

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  Serial.begin(9600);
  Serial.println("Initialisiere Sensor...");
  delay(60000);
  Serial.println("Fertig.");
}

```

```
14  void loop() {
15    if(digitalRead(SENSOR))             // Wenn Sensorausgang
16                                        // HIGH-Pegel hat, dann
17    {
18      digitalWrite(LED_BUILTIN, HIGH);  // schalte die LED ein.
19
20      if(letzterZustand == false)       // nur wenn erstmalig HIGH
21                                        // anliegt,
22      {                                 // Textmeldung ausgeben:
23        Serial.print("Bewegung erkannt! Zeitpunkt: ");
24        Serial.print(millis()/1000);
25        Serial.println(" Sekunden nach dem Einschalten.");
26        letzterZustand = true;
27      }
28    }
29    else
30    {
31      digitalWrite(LED_BUILTIN, LOW);
32      letzterZustand = false;
33    }
34  }
```

In der `setup()`-Routine ist eine Pause von 60 Sekunden vorgesehen. Der Grund liegt in der Dokumentation des Sensor-Herstellers, hier wird nach dem Einschalten der Spannungsversorgung eine Initialisierungszeit von einer Minute angegeben, in welcher der Sensor keine verlässlichen Messungen durchführt.

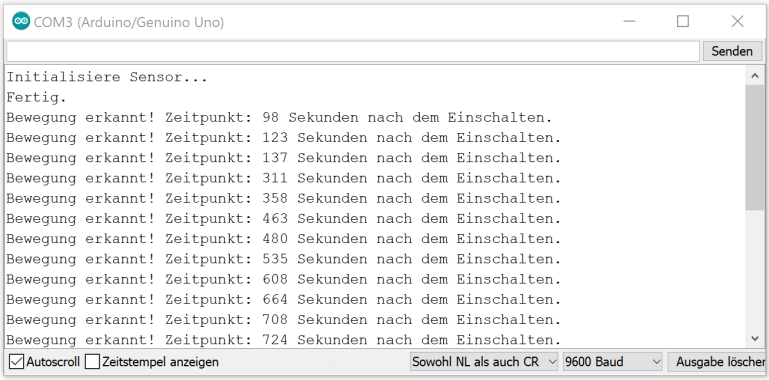


Abb. 9.14 beispielhafte Bildschirmausgabe

Die Bildschirmausgabe gibt den Zeitpunkt nur relativ zum Einschaltmoment an. Für eine reale Zeitangabe könnte man den Versuch mit einem Echtzeitmodul kombinieren, welches in Kapitel 9.11 betrachtet wird.

9.5 Bodenfeuchte-Sensor

Ebenso große Beliebtheit wie das Basteln im Hobbykeller genießen auch die Gartenpflege und die Aufzucht von Zimmerpflanzen. Der Gedanke liegt nahe, die Gebiete zu kombinieren, um zum Beispiel ein automatisches Bewässerungssystem zu entwickeln.

Ein Temperatur- und Luftfeuchtesensor (siehe Kapitel 9.6) kann dafür bereits wichtige Daten liefern. Eine weitere relevante Größe ist die Bodenfeuchte, welche ebenfalls durch Sensoren messbar ist. Ältere Modelle wendeten das Prinzip der Widerstandsmessung an: Ein feuchter Boden leitet den Strom besser als ein trockener. Die dafür im Kontakt mit dem Boden stehenden Elektroden (Metallflächen) korrodieren allerdings schnell. Außerdem entstehen bei der Messung aufgrund der Elektrolyse[2] (in geringem Maße) Metallionen im Boden, die für Menschen giftig sind.

Neuere Modelle arbeiten daher kapazitiv. Sie erinnern sich sicher an den Aufbau eines Kondensators (Kapitel 3.1.2.2), er besteht aus zwei Metallflächen und einem dazwischenliegenden Isolator (Dielektrikum). Die Kapazität des Kondensators wird unter anderem auch durch das Material des Dielektrikums bestimmt. Beim kapazitiven Bodenfeuchte-Sensor werden zwei Metallflächen (mit einer dünnen Lackschicht als Korrosionsschutz) in den Boden eingebracht, der Boden selbst dient als Dielektrikum. Eine Elektronik misst die Kapazität dieses „Bodenkondensators". Daraus lassen sich Rückschlüsse auf die Feuchte ziehen.

2 Wird elektrischer Strom durch eine leitfähige Flüssigkeit geleitet, bilden sich an den Elektroden unter bestimmten umständen Ionen, welche sich Ablagern oder in die Flüssigkeit gelangen. Dies nennt man Elektrolyse.

Abb. 9.15 kapazitiver Bodenfeuchte-Sensor: Die Metallflächen (links) sind mit einem schwarzen Korrosionsschutz bedeckt und daher kaum erkennbar. Die Elektronik rechts darf nicht in den Boden eingebracht werden – oder muss entsprechend isoliert sein.

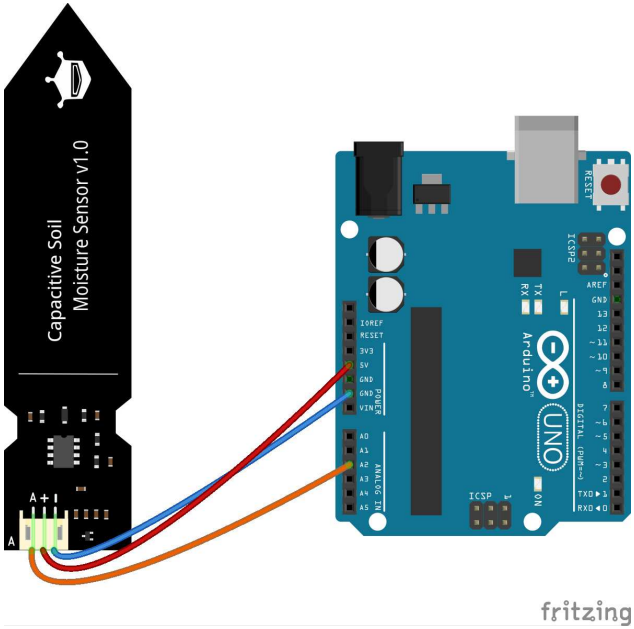


Abb. 9.16 Der Sensorwert wird als analoge Spannung zwischen 0 V und 5 V ausgegeben

Das Modul wird mit der Betriebsspannung und Masse verbunden und liefert den Messwert als analoge Spannung. Daher kann für einen Test

einfach der Sketch aus Kapitel 9.3 verwendet werden, er gibt die eingelesenen Analogwerte am seriellen Monitor aus.

Es ist zu beachten, dass der Sensorwert nicht nur vom Wassergehalt, sondern auch von der Zusammensetzung des Bodens abhängt. Deshalb sollte bei einer konkreten Anwendung zunächst eine Kalibrierung erfolgen, indem man an der späteren Messstelle je einen Messwert bei sehr feuchtem und sehr trockenem Boden aufzeichnet. Innerhalb dieser Grenzen werden sich dann auch spätere Messwerte bewegen. Für eine Bewässerungsanlage empfiehlt sich zudem eine Hysterese, näheres dazu finden Sie im Zusammenhang mit dem Relais-Modul in Kapitel 11.1.

9.6 Temperatursensor

Als Nächstes möchten wir die Temperatur messen. Es gibt zu diesem Zweck temperaturempfindliche Widerstände, welche wir genau wie den Fotowiderstand nutzen könnten. Allerdings ergibt sich durch diese analoge Messmethode der Nachteil, dass Toleranzen des zweiten Widerstandes sowie die Leitungswiderstände (bei längeren Wegen) unser Messergebnis verfälschen können. Aus diesem Grunde strebt man üblicherweise an, den analogen Messwert sehr nah am Sensor zu digitalisieren.

Das Modul DHT22 arbeitet genau nach diesem Prinzip. Es verfügt über einen Temperatur- und Feuchtesensor, dessen Werte direkt im Modul durch einen Mikrochip digitalisiert werden. Das Auslesen erfolgt digital über ein spezielles serielles Protokoll. Dazu schickt der Mikrocontroller einen Abfrageimpuls an das Sensormodul, welches dann mit den Messwerten antwortet.

Diese Arbeit nimmt uns natürlich wieder eine Bibliothek ab. Am besten eignet sich die „DHT sensor library“.

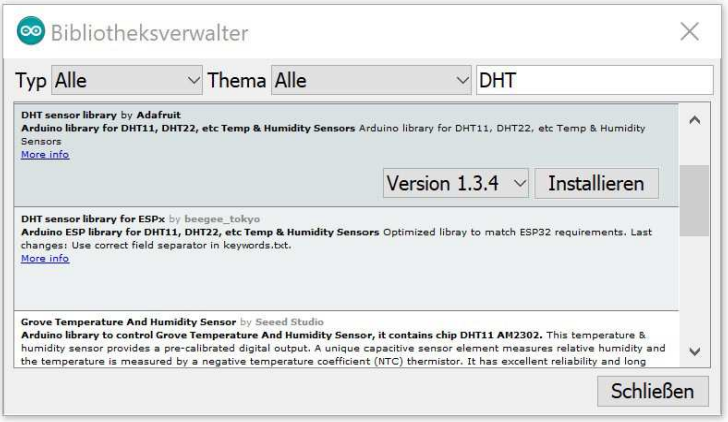


Abb. 9.17 Nutzen Sie die DHT-Bibliothek

Allerdings sollten Sie zusätzlich noch die „Adafruit Unified Sensor Driver Library“[3] herunterladen und über den Befehl *Sketch -> Bibliothek einbinden -> .ZIP-Bibliothek hinzufügen* installieren, sonst erhalten Sie eine Fehlermeldung beim Kompilieren.

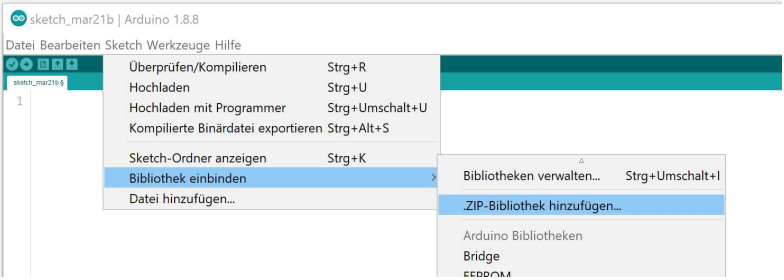


Abb. 9.18 Über den Menübefehl lassen sich auch Bibliotheken aus anderen Quellen einbinden

Die Verbindung ist wieder sehr einfach:

3 https://github.com/adafruit/Adafruit_Sensor

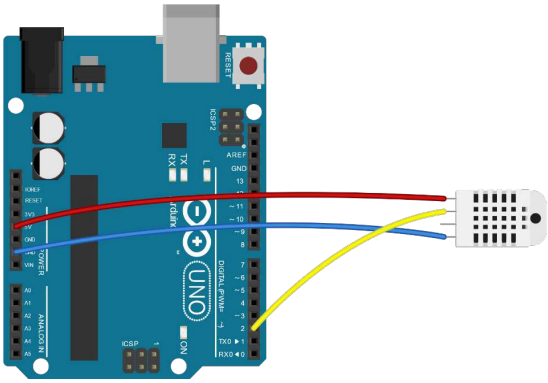


Abb. 9.19 Der Sensor kann direkt mit dem Board verbunden werden. Es gibt auch Varianten, bei denen der Sensor auf einer kleinen Platine aufgelötet ist, die Anschlüsse sind jedoch identisch.


```
#include "DHT.h"

DHT dht(2, DHT22);                                          // (1)

void setup() {
  Serial.begin(9600);
  dht.begin();                                              // (2)
}

void loop() {

  float Temp = dht.readTemperature();                       // (3)
  float Feuchte = dht.readHumidity();                       // (4)
  float HI = dht.computeHeatIndex(Temp, Feuchte, false);    // (5)

  Serial.print("Temperatur: ");
  Serial.print(Temp);
  Serial.print("°C  Luftfeuchtigkeit: ");
  Serial.print(Feuchte);
  Serial.print("%  gefühlte Temp.: ");
  Serial.print(HI);
  Serial.println("°C ");

  delay(5000);
}

```

1. Das Objekt `dht` repräsentiert unseren Sensor, wir übergeben zur Erstellung den Anschlusspin sowie den Typ des Sensors. (Es gibt zum Beispiel auch Sensoren des Typs DHT11, dieser besitzt eine geringere Genauigkeit.)
2. Unser Sensor wird, wie bei vielen Modulen üblich, über einen Befehl initialisiert.
3. Die Funktion `readTemperature()` liest die Temperatur in Grad Celsius aus und gibt sie als Fließkommawert zurück.
4. Die Funktion `readHumidity()` liest die Luftfeuchte aus und gibt sie als Fließkommawert zurück.
5. Die ebenfalls von der Bibliothek bereitgestellte Funktion `computeHeatIndex()` berechnet aus den Argumenten Temperatur und Feuchte den sogenannten *Heat Index*, in Deutschland bekannt als *gefühlte Temperatur*. Das dritte Argument gibt an, ob in Fahrenheit gerechnet werden soll – ist es `false`, wird Celsius benutzt.

Der Sketch gibt erwartungsgemäß im 5-Sekunden-Takt die aktuellen Werte aus:

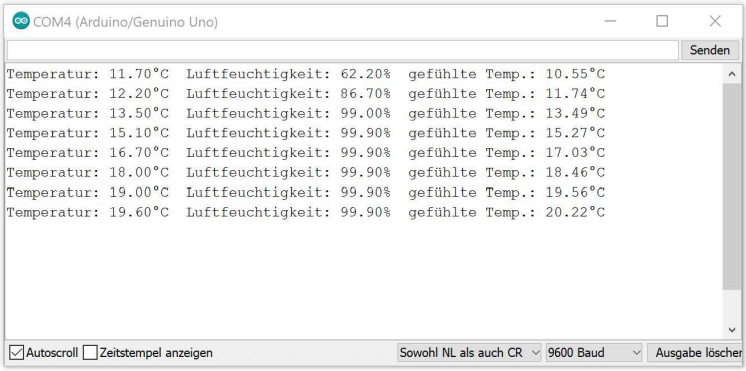


Abb. 9.20 beispielhafte Monitorausgabe

9.7 Ultraschall-Abstandssensor

Besonders für Projekte mit beweglichen Teilen sind Abstandsmessungen interessant. Zu diesem Zweck werden in manchen Anwendungsbereichen Ultraschallsensoren genutzt, Sie kennen sie als Einparkhilfe beim PKW oder als Füllstandsensoren. Auch für Mikrocontroller sind günstige Ultraschallmodule erhältlich.

Abb. 9.21 Auf dem HC-SR04-Modul sind die Ultraschall-Wandler deutlich als große Zylinder erkennbar

Im Beispiel verwenden wir ein HC-SR04-Modul. Es verfügt bereits über eine Auswerteelektronik, welche die Signalverarbeitung vereinfacht. Soll eine Abstandsmessung erfolgen, erhält es vom Mikrocontroller über den Trigger-Anschluss ein Signal. Daraufhin sendet es automatisch acht für Menschen unhörbare Ultraschallimpulse (40 kHz) ab und prüft, ob ein Echo zurückkommt. Ist dies der Fall, sendet das Modul auf dem Echo-Anschluss ein Signal der gleichen zeitlichen Länge, die der Schallimpuls für seinen Weg benötigt hat.

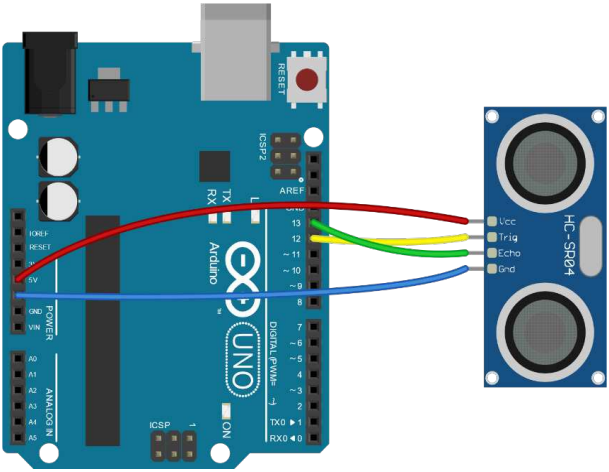


Abb. 9.22 Das Modul kann mit entsprechenden Verbindungskabeln wieder direkt mit dem Arduino verbunden werden

Vor der Verwendung suchen wir wieder eine passende Bibliothek und laden anschließend den Sketch hoch.

Abb. 9.23 Die Bibliothek HCSR04 von Martin Sosic eignet sich gut für unser Testprojekt

```
#define ECHOPIN 13
#define TRIGPIN 12

#include "HCSR04.h"
UltraSonicDistanceSensor Abstandssensor(TRIGPIN,ECHOPIN);  // (1)

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

void loop () {
    Serial.print("Abstand: ");
    Serial.print(Abstandssensor.measureDistanceCm());      // (2)
    Serial.println(" cm");
    delay(500);
}
```

1. Das Objekt `Abstandssensor` vom Typ `UltraSonicDistanceSensor` repräsentiert den Ultraschallsensor. Zur Initialisierung werden nacheinander die Pins für das Trigger- und Echosignal übergeben. Zur besseren Lesbarkeit wurden diese zu Beginn des Sketches als Konstanten definiert.
2. Die Funktion `measureDistanceCm()` gibt den Abstand in Zentimeter als Fließkommawert zurück, wir können ihn direkt an die Ausgabefunktion weitergeben.

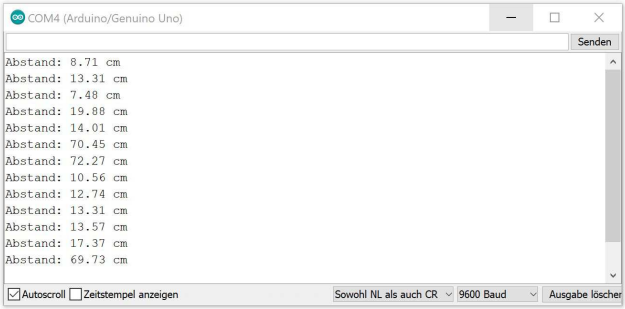


Abb. 9.24 beispielhafte Monitorausgabe

Da das Echosignal über Pin 13 empfangen wird, können Sie auch an der Onboard-LED beobachten, wie sich der Impuls deutlich verkürzt, wenn

man den Abstand verringert. Dauert das Echo zu lang oder wird überhaupt keines empfangen, wird `-1` ausgegeben, um den Fehler zu signalisieren.

9.8 Hall-Sensor

Um die Position eines Rotors oder anderer beweglicher Teile zu erkennen, gibt es neben den naheliegenden mechanischen Tastern auch noch eine elegantere Möglichkeit: Hall-Sensoren. Der namensgebende Hall-Effekt erzeugt eine quer gerichtete Spannung, wenn ein stromdurchflossener Leiter von einem externen Magnetfeld durchsetzt wird. Auf dieser Grundlage wurden empfindliche Magnetfeld-Sensoren in halbleiterbauweise konstruiert. Werden bewegliche Teile an entsprechender Stelle mit kleinen Dauermagneten ausgestattet, kann durch Hall-Sensoren sogar über Abstände von mehr als einem Zentimeter hinweg zuverlässig deren Näherung erkannt werden. Anders als bei mechanischen Tastern erfolgt diese Detektion dabei völlig geräuschlos und verschleißfrei.

Im Gegensatz zu Spulen, mit denen man aufgrund des Induktionseffektes sich ändernde Magnetfelder erkennen kann, sind Hall-Sensoren in der Lage, stationäre (also konstante) Magnetfelder zu messen. Manche Telefone nutzen diese Technik zum Beispiel auch, um zu erkennen, ob der Hörer aufgelegt wurde.

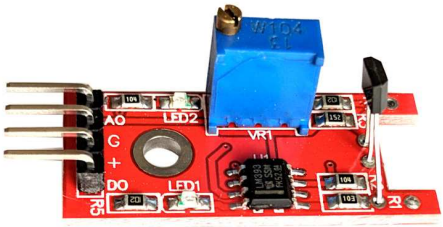


Abb. 9.25 Das Hall-Sensor-Modul KY-024 – Der abstehende kleine schwarze 3-Pin-Baustein auf der rechten Seite ist der eigentliche Hall-Sensor. Der blaue Quader ist ein Wendelpotentiometer, mit dem die Empfindlichkeit reguliert werden kann.

Das von uns verwendete Modul KY-024 verfügt neben dem eigentlichen Sensor noch über ein Wendelpotentiometer[4] zur Regulierung der Empfindlichkeit. Im Normalfall sollte der Auslieferungszustand aber bereits brauchbare Ergebnisse liefern. Die am Ausgang A0 ausgegebene Spannung liegt zwischen 0 und 5 V und repräsentiert das detektierte Magnetfeld. Im feldfreien Zustand liegt diese Spannung nahe 2,5 V. Je nach Richtung der Feldlinien (Nordpol/Südpol) sinkt oder steigt sie entsprechend.

Zusätzlich verfügt das Modul über einen Komparator (also eine Vergleichsschaltung), welcher die vom Sensor ausgegebene analoge Spannung mit einer Referenzspannung von 2,5 V (halbe Betriebsspannung) vergleicht. Liegt die Sensorspannung darunter, wird der digitale Ausgang D0 auf HIGH geschaltet und LED2 leuchtet, sonst liegt der Ausgang auf LOW und LED2 ist dunkel. LED1 leuchtet dauerhaft, um die vorhandene Betriebsspannung anzuzeigen.

Im folgenden Versuch wollen wir die Ausgabesignale des Sensormoduls erfassen und auf dem Bildschirm anzeigen. Zum Test ist ein kleiner Dauermagnet erforderlich.

4 Im Gegensatz zu normalen Potentiometern, deren Achse aufgrund des mechanischen Anschlags nie eine komplette Umdrehung erlaubt, erfolgt die Verstellung bei Wendelpotis über Schrauben- oder Schneckengetriebe, so dass eine feinere Einstellung und mehrere Umdrehungen möglich sind. Dennoch gibt es auch hier Endanschläge.

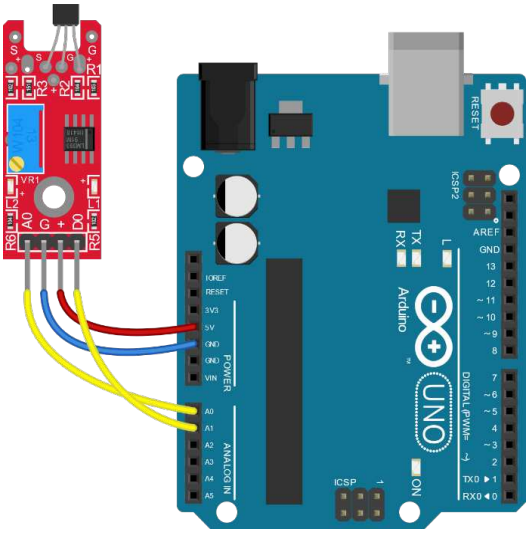


Abb. 9.26 Zur Erfassung der Signale nutzen wir diesmal zwei analoge Pins, auch wenn Do ein digitales (binäres) Signal ist – denn die analogen Pins können schließlich auch als digitale Eingänge verwendet werden

Da der Sensor lediglich einen analogen Spannungswert und einen digitalen Binärwert ausgibt, ist keine Bibliothek notwendig.

```
#define ANALOGPIN 0
#define DIGITALPIN 15                                          // (1)

int SensorWert;
int minWert = 1023;                                            // (2)
int maxWert = 0;

void setup()
{
  Serial.begin(9600);
  pinMode(LED_BUILTIN,OUTPUT);
}
void loop()
{
  SensorWert = analogRead(ANALOGPIN);                          // (3)

```

```
17    if(SensorWert > maxWert)                                    // (4)
18      maxWert = SensorWert;
19
20    if(SensorWert < minWert)                                    // (5)
21      minWert = SensorWert;
22
23    Serial.print("Aktueller Messwert: ");
24    Serial.print(SensorWert);
25    Serial.print(", Maximal-/Minimalwert: ");
26    Serial.print(maxWert);
27    Serial.print(" / ");
28    Serial.println(minWert);
29
30    digitalWrite(LED_BUILTIN,digitalRead(DIGITALPIN));          // (6)
31
32    delay(1000);
33  }
```

1. Für die Erfassung des binären Signals vom Do-Ausgang des Sensormoduls wollen wir aufgrund des einfacheren Kabelweges den Eingangspin A1 verwenden, dieser kann ja auch digital genutzt werden. Er wird dazu über Pinnummer `15` angesprochen, siehe Kapitel 5.3 – alternativ wäre hier auch die Bezeichnung `A1` möglich.

2. Wir deklarieren je eine Variable für den Minimal- und Maximalwert. Ihre anfänglichen Werte entsprechen dem kleinstmöglichen Maximalwert und dem größtmöglichen Minimalwert. Würden wir sie stattdessen auf den mittleren Wert 512 setzen, zeigt beispielsweise der Minimalwert auch dann immer 512, selbst wenn alle gemessenen Werte weit über 700 lägen.

3. Wir lesen den Sensorwert als analogen Spannungswert ein, der Wertebereich reicht bei analogen Pins von 0 bis 1023. Dementsprechend wird der Normalwert ohne Magnetfeld in der Nähe von 512 liegen. Geringfügige Abweichungen sind aufgrund von Bauteiltoleranzen möglich.

4. Liegt der aktuell gemessene Sensorwert über dem gespeicherten Maximalwert, wird dieser entsprechend erhöht.

5. Der Minimalwert wird ebenfalls auf gleiche Weise untersucht.

6. Das binäre Signal des Sensormoduls wird einfach zur Onboard-LED weitergeleitet. Allerdings erfolgt auch diese Weiterleitung, wie die gesamte Signalverarbeitung in diesem Sketch, nur einmal pro Sekunde aufgrund des `delay()`-Befehls. Dieser Zyklus kann beliebig verkürzt werden – hier dient er nur dazu, die Ausgabe am seriellen Monitor nicht zu „überfluten".

Wie erwartet gibt der Sketch nach dem Hochladen sekündlich die aktuellen Messwerte auf dem seriellen Monitor aus. Zum Zurücksetzen der Maximal- und Minimalwerte kann die Reset-Taste des Arduinos genutzt werden.

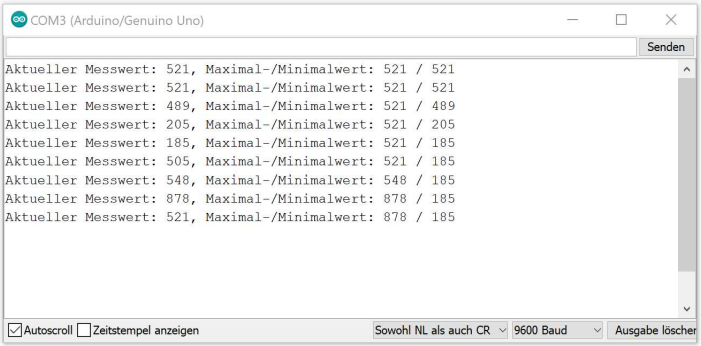


Abb. 9.27 beispielhafte Bildschirmausgabe

Möchte man beispielsweise die Nullposition eines rotierenden Teils finden, genügt es, diesen Rotor an einer Stelle mit einem kleinen Dauermagneten auszustatten, welcher bei jeder Drehung einen feststehenden Hall-Sensor im Abstand von einigen Millimetern überstreift. Durch Auswertung des Maximal- beziehungsweise Minimalwertes dieses Sensors kann die Nullposition zuverlässig ermittelt werden. In so einem Fall würde man den Sensorwert natürlich nicht nur einmal pro Sekunde, sondern deutlich öfter einlesen. Das analoge Signal setzt hierbei keine Limitierungen.

In der fortgeschrittenen Programmierung könnte man auch das binäre Signal des Do-Ausgangs verwenden, um einen Interrupt am Ar-

duino auszulösen und damit zum Beispiel die Umdrehungen eines Rotors zu zählen. Auch die Stoppuhr aus Kapitel 8 ließe sich damit steuern, dann sollte der Eingangspin genau genommen als `INPUT` statt `INPUT_PULLUP` konfiguriert werden, da kein Pull-Up-Widerstand benötigt wird.

9.9 Beschleunigungssensor

Eine weitere Datenquelle für Informationen über die Umgebung sind Beschleunigungssensoren. Mit ihrer Hilfe kann die räumliche Lage ermittelt werden und sie erlauben Rückschlüsse auf Bewegungen.

Wir experimentieren beispielhaft mit einem MPU-6050-Modul. Es besitzt zusätzlich zum Beschleunigungssensor auch ein Gyroskop, also einen Drehratensensor. Alle Werte lassen sich komfortabel per I^2C-Bus auslesen. Baugleiche Module existieren unter der Bezeichnung GY-521.

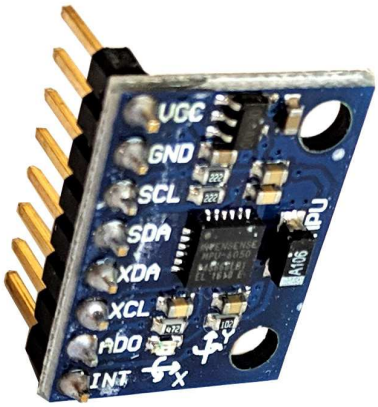


Abb. 9.28 Das Modul ist nur so groß wie ein Daumennagel

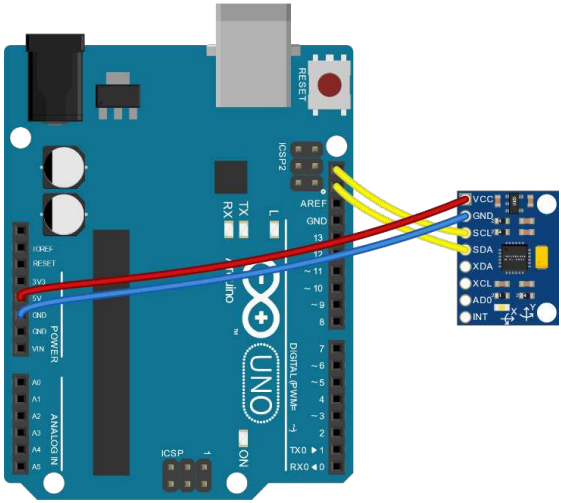


Abb. 9.29 Statt der analogen Pins A4 und A5 können wir für die I²C-Verbindung auch die seitlich beschrifteten SDA- und SCL-Pins auf der gegenüberliegenden Seite nutzen

Dazu verwenden wir die Bibliothek MPU6050_tockn.h.

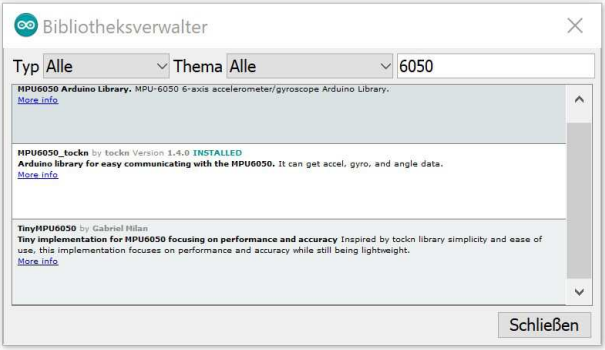


Abb. 9.30 Auch hier gibt es mehrere Bibliotheken, die Ressource mit dem Namen MPU6050_tockn hat sich im Test als probat erwiesen

Zur Demonstration eignet sich direkt der von der Bibliothek mitgelieferte Beispielsketch, welcher aufgrund seines Umfanges hier nur in Ausschnitten erörtert werden soll. Der größte Teil des Sketches beschäftigt sich mit der Messwertausgabe und ist selbsterklärend.

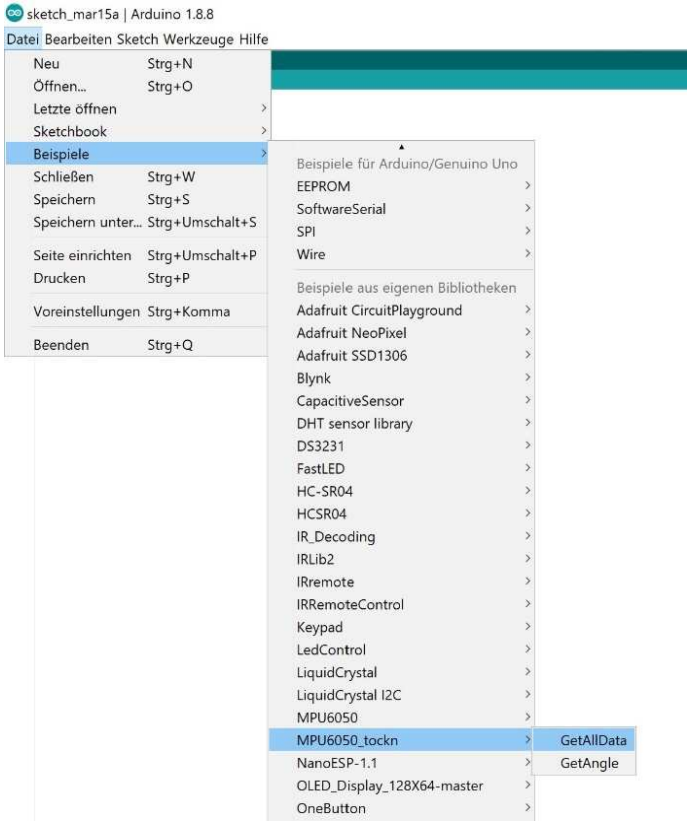


Abb. 9.31 Wir nutzen den Beispielsketch „GetAllData“, den die soeben installierte Bibliothek bereitstellt.

```
1  #include "MPU6050_tockn.h"
2  #include "Wire.h"
3
4  MPU6050 mpu6050(Wire);                                    // (1)
5
```

```
6  long timer = 0;
7
8  void setup() {
9    Serial.begin(9600);
10   Wire.begin();
11   mpu6050.begin();
12   mpu6050.calcGyroOffsets(true);                          // (2)
13 }
14
15 void loop() {
16   mpu6050.update();                                       // (3)
17
18   if(millis() - timer > 1000){                            // (4)
19
20     Serial.println("=======================================");
21     Serial.print("temp : ");
22     Serial.println(mpu6050.getTemp());                    // (5)
23     Serial.print("accX : ");Serial.print(mpu6050.getAccX());
24   [...]
25     Serial.println("=======================================");
26     timer = millis();                                     // (6)
27   }
28 }
```

1. Das Objekt `mpu6050` repräsentiert unseren Sensor. Als Argument wird das `Wire`-Objekt übergeben, daran erkennt die Bibliothek, dass I^2C zur Kommunikation verwendet werden soll. Falls Sie sich wundern, warum keine Adresse angegeben wird: Das MPU-6050-Modul hat eine feste Adresse, die nicht verändert werden kann – die Bibliothek kennt sie bereits.

2. Der Sensor wird nach seiner Initialisierung zunächst kalibriert, dafür sollte er sich in Ruhelage befinden. Dieser Vorgang ist üblich bei Beschleunigungssensoren.

3. Die Funktion `update()` führt den eigentlichen Messvorgang durch. Vergisst man ihren Aufruf, erhält man stets nur die gleichen Werte.

4. Wir hatten die Verzögerung in bisherigen Beispielen immer per `delay()` realisiert. Hier wird ein anderer Weg gewählt: Bei jedem Durchlauf der Hauptschleife wird geprüft, ob die Differenz zwischen den aktuell abgelaufenen Millisekunden seit Programmstart

(`millis()`) und einer `timer`-Variable größer als 1000 (also 1 Sekunde) ist. Falls ja, werden

5. diverse Messdaten ausgegeben und
6. die `timer`-Variable auf den aktuellen Wert gesetzt, so dass in den Schleifendurchläufen während der nächsten 1000 Millisekunden die `if`-Bedingung wieder `false` ist.

Die beschriebene Vorgehensweise ohne `delay()`-Befehl hat bei umfangreicheren Sketches den Vorteil, dass das Programm während der Wartezeit andere Aufgaben erledigen kann und nicht am `delay()`-Befehl „festhängt“.

Die Ausgabe des Beispielsketches sieht etwa so aus:

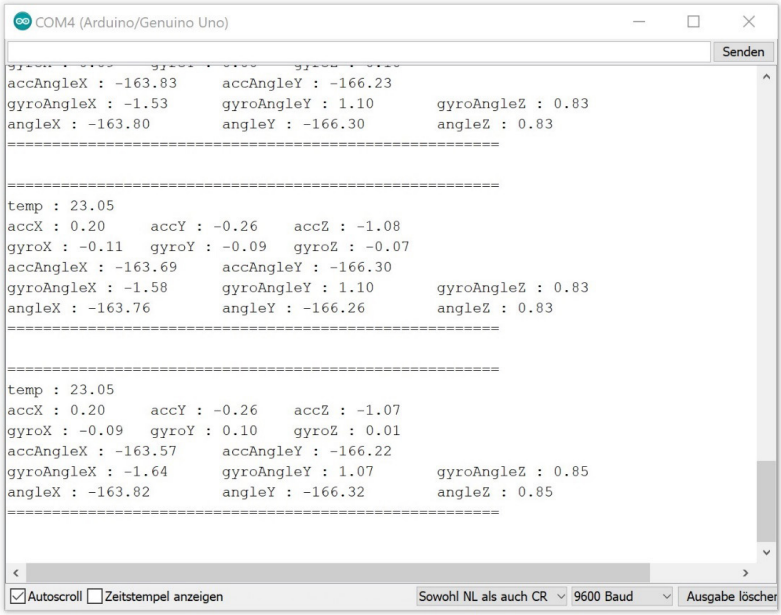


Abb. 9.32 Der Sensor ermittelt umfangreiche Daten

Neben der Temperatur werden für alle 3 Achsen die Beschleunigungswerte sowie Winkellagen und -beschleunigungen angegeben. Bewegen

Sie den Sensor und beobachten Sie die Veränderung der Werte. Je nach Anwendung werden in konkreten Projekten natürlich immer nur die benötigten Informationen ausgewertet. So lassen sich zum Beispiel Lageerkennungen für Displays (wie beim Smartphone) oder sogar selbstbalancierende Roboter entwickeln.

9.10 Kompass

Ebenfalls ist in jedem Smartphone mittlerweile ein Kompass-Sensor zu finden. Wir testen am Beispiel des GY-271-Moduls eine mögliche Implementierung in einem Arduino-Projekt. Das Modul ähnelt sehr dem eben betrachteten MPU-6050, jedoch ist die Funktion eine komplett andere.

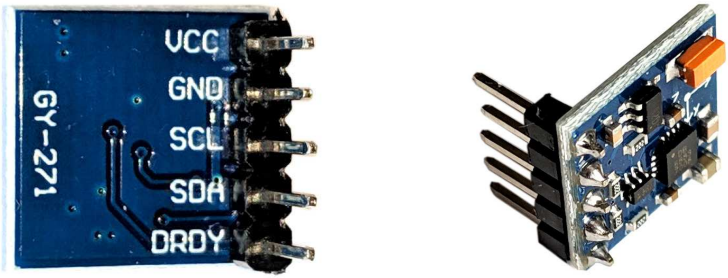


Abb. 9.33 Das Modul ist noch kleiner als das des Beschleunigungssensors. Der Pin DRDY (Data Ready) wurde vom Hersteller als Statussignal vorgesehen, wird aber im Normalfall nicht genutzt.

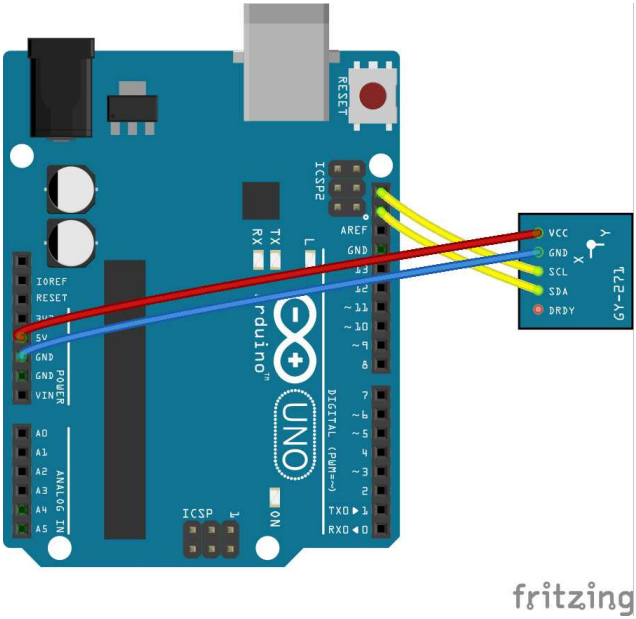


Abb. 9.34 Die Anschlussweise ist identisch zum Beschleunigungssensor. Da beide Module unterschiedliche (fest voreingestellte) I²C-Adressen haben, könnte man sie auch gemeinsam anschließen.

Auch diese Sensorplatine verfügt über einen I²C-Anschluss und lässt sich daher wieder sehr einfach verbinden. Allerdings ist die benötigte Bibliothek leider nicht direkt über die Arduino IDE verfügbar, Sie können sie jedoch stattdessen einfach als .zip-Datei von Github[5] herunterladen und über den entsprechenden Menübefehl installieren (wie bereits in *Abb. 9.16* gezeigt).

Unser Beispielsketch liest wieder die Messwerte aus und zeigt sie auf dem Computer an.

5 https://github.com/mechasolution/Mecha_QMC5883L

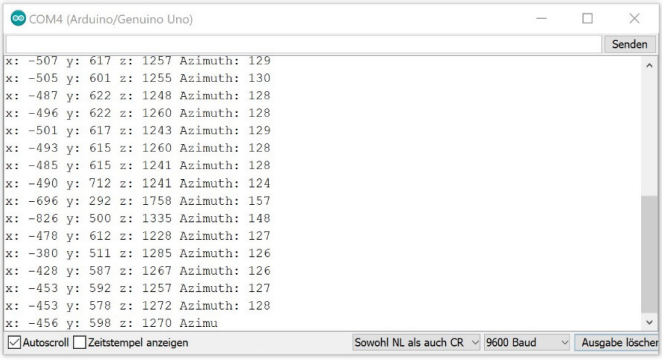


Abb. 9.35 beispielhafte Monitorausgabe der magnetischen Feldstärken aller 3 Achsen sowie des errechneten Azimuths (Winkel bezogen auf Norden)

```
#include "Wire.h"
#include "MechaQMC5883.h"

MechaQMC5883 Kompass;                                    // (1)
int x, y, z;                                             // (2)
int azimuth;

void setup() {
  Wire.begin();
  Serial.begin(9600);
  Kompass.init();                                        // (3)
}

void loop() {
  Kompass.read(&x, &y, &z, &azimuth);                    // (4)
  Serial.print("x: ");
  Serial.print(x);                                       // (5)
  Serial.print(" y: ");
  Serial.print(y);
  Serial.print(" z: ");
  Serial.print(z);
  Serial.print(" Azimuth: ");
  Serial.print(azimuth);
  Serial.println();
  delay(500);
}
```

1. Wie üblich bei Bibliotheken, erzeugen wir ein Objekt – diesmal aus der Klasse `MechaQMC5883`.
2. Diese Bibliothek erfordert zusätzlich, dass wir Variablen für die Messergebnisse anlegen.
3. Auch Initialisierungsfunktionen sind uns schon bekannt, in dieser Bibliothek heißt sie `init()`.
4. Die `read()`-Funktion arbeitet hier etwas anders als die Funktionen, die wir bereits kennengelernt haben. Statt das Ergebnis als Rückgabewert zu liefern oder im Objekt abzuspeichern, erwartet sie Zeiger auf die Speicheradressen der gewünschten Variablen. Sie schreibt die Ergebnisse dann direkt an die entsprechende Stelle im Arbeitsspeicher und weist sie dadurch den Variablen zu.
5. Entsprechend müssen nachfolgend nur noch die Werte der Variablen ausgegeben werden.

Die Werte für `x`, `y` und `z` sind dabei ein Maß für die gemessenen Magnetfelder in den entsprechenden Raumrichtungen. Der Azimuth ist der daraus resultierende Winkel gegenüber der Nordrichtung, also die Stellung einer gedachten Kompassnadel. Wie bei jedem Kompass ist bei der Auswertung der Daten Vorsicht geboten, sie können leicht durch magnetische Störfelder beeinflusst werden. Bringen Sie einen Dauermagneten in die Nähe des Sensors und testen Sie es selbst.

9.11 Echtzeitmodul

Eine frühe Anwendung von Mikroprozessoren findet sich in Digitaluhren. Die dafür notwendige Zeitmessung ist mit dem Arduino allein nur umständlich möglich. Zwar gibt es einige Funktionen, welche Zeitwerte bereitstellen (beispielsweise gibt `millis()` die Zahl der vergangenen Millisekunden seit Programmstart zurück), jedoch werden diese mit jedem Neustart wieder zurückgesetzt. Ohne Spannungsversorgung ist eine weitere Zeitmessung nicht möglich, da es keine Batterie auf der Platine gibt. Wie bei einem alten Radiowecker müsste man also nach jedem Neustart die Uhrzeit erneut einstellen.

Abhilfe schafft eine Echtzeituhr (oft auch bezeichnet als RTC – *Real Time Clock*) als Zusatzmodul. Das Modell ZS-042 (mit dem Chip DS3231) erfreut sich hierbei großer Beliebtheit. Die Platine ist mit einer Knopfzelle ausgestattet, welche das Weiterlaufen der quarzbasierten Zeitmessung auch ohne Betriebsspannung sicherstellt. Sie misst zudem minütlich die Temperatur und korrigiert rechnerisch den thermischen Einfluss auf den internen Schwingquarz. Daraus ergibt sich eine monatliche Zeitabweichung von unter 10 Sekunden, was für Hobby-Anwendungen mehr als ausreichend sein sollte. Der Anschluss ist wieder komfortabel per I^2C ausgeführt.

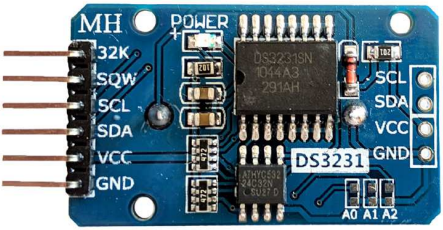


Abb. 9.36 Die Knopfzelle befindet sich auf der Rückseite des Moduls

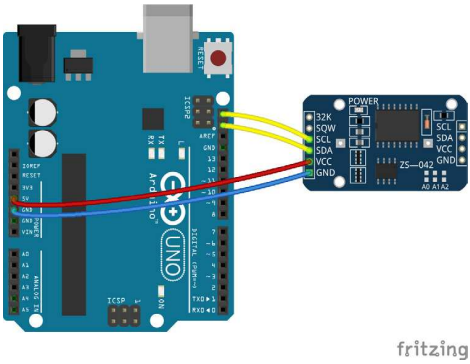


Abb. 9.37 Der Datenaustausch erfolgt, wie bei vielen digitalen Modulen, per I^2C. An den zusätzlichen Pins 32K und SQW kann optional eine 32 Kilohertz-Schwingung sowie ein Weck-Signal abgegriffen werden.

Zur Kommunikation mit der Platine gibt es eine Vielzahl von Bibliotheken, wir nutzen im Folgenden die DS3231.h von Andrew Wickert:

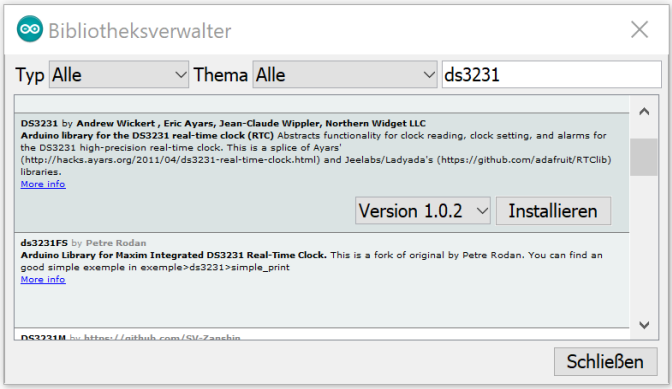


Abb. 9.38 Andrew Wickerts Bibliothek

Der folgende Sketch demonstriert den Datenaustausch mit dem Modul und erzeugt diese Anzeige am Bildschirm.

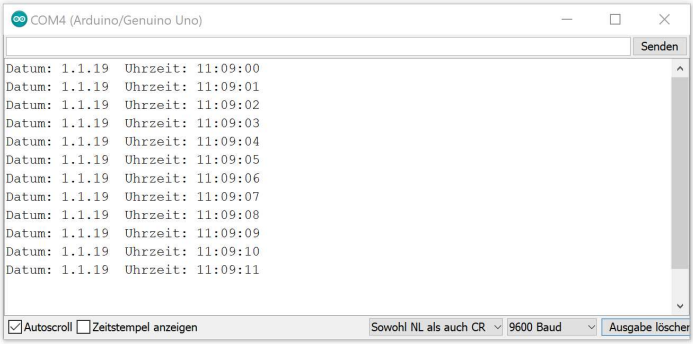


Abb. 9.39 beispielhafte Bildschirmausgabe

```
1  #include "DS3231.h"
2
3  DS3231 Uhr;                                          // (1)
```

```
4
5  boolean Dummy;                                          // (2)
6
7  void setup()
8  {
9    Serial.begin(9600);
10   Wire.begin();                                         // (3)
11
12   Uhr.setDate(1);                                       // (4)
13   Uhr.setMonth(1);
14   Uhr.setYear(19);
15   Uhr.setHour(11);
16   Uhr.setMinute(9);
17   Uhr.setSecond(0);
18 }
19
20 void loop()
21 {
22   Serial.print("Datum: ");
23   Serial.print(Uhr.getDate());                          // (5)
24   Serial.print(".");
25   Serial.print(Uhr.getMonth(Dummy));                    // (6)
26   Serial.print(".");
27   Serial.print(Uhr.getYear());
28
29   Serial.print("  Uhrzeit: ");
30   Serial.print(Uhr.getHour(Dummy,Dummy));               // (7)
31   Serial.print(":");
32   if(Uhr.getMinute() < 10)                              // (8)
33     Serial.print("0");
34   Serial.print(Uhr.getMinute());
35   Serial.print(":");
36   if(Uhr.getSecond() < 10)
37     Serial.print("0");
38   Serial.println(Uhr.getSecond());
39
40   delay(1000);
41 }
```

1. Wir erstellen das Objekt `Uhr` als Instanz der Klasse `DS3231`. Es repräsentiert nun unser Echtzeitmodul.

2. Diese Variable werden wir nur aufgrund einer Gegebenheit dieser Bibliothek brauchen (siehe Punkt (6)). Da sie keine praktische Funktion hat, passt der Name `Dummy`.

3. Wir starten die I²C-Verbindung als Master. Es war nicht notwendig, die Wire.h-Bibliothek einzubinden, da dies bereits innerhalb der Bibliothek DS3231.h geschieht. Dort ist ebenfalls bereits die Adresse des RTC-Moduls hinterlegt.

4. Da das Modul im Auslieferungszustand oft noch nicht gestellt wurde, stellen wir hier einmalig eine Uhrzeit ein, in diesem Fall den 1. Januar 2019 um 11:09 Uhr. Dafür stellt das Objekt verschiedene Funktionen bereit, welche jeweils mit `set` beginnen und deren Namen selbsterklärend sind.

   Nach dem ersten Hochladen des Sketches können Sie diese Befehle auskommentieren oder löschen. (Eine Möglichkeit der Zeiteinstellung durch den Nutzer wird im nachfolgenden Praxisprojekt dargestellt.)

5. Analog zum Einstellen von Datum und Uhrzeit stellt das Objekt auch Funktionen bereit, um die Daten auszulesen. Sie beginnen jeweils mit `get` und liefern als Rückgabewert die jeweilige Zahl. In der Regel erwarten diese Funktionen keine Argumente, außer:

6. Die Funktion `getMonth()` erwartet ein Argument von Typ `boolean`. Der Hintergrund: Beim Abruf des Monats (welcher nur 4 Bit benötigt), sendet das RTC-Modul ein zusätzliches Bit, welches angibt, ob das Datum vor oder nach dem Jahr 2000 liegt. Übergibt man der Funktion eine boolesche Variable, schreibt die Funktion dieses Bit an deren Speicheradresse. Für heutige Anwendungen ist dies obsolet. Da die Bibliothek aber einst so deklariert wurde, müssten wir hier eine boolesche Variable übergeben, sonst erhalten wir eine Compiler-Fehlermeldung. Die Variable `Dummy` hat also keinen weiteren Nutzen für uns, wir brauchen sie lediglich, um den formalen Anforderungen dieser Funktion gerecht zu werden.

7. Die Funktion `getHour()` erwartet sogar zwei boolesche Argumente, hier geht es um das in Amerika verbreitete 12-Stunden-Format mit *AM* und *PM*. Da diese Information für uns keine Relevanz besitzt, übergeben wir zweimal den `Dummy`, um die formalen Anforderungen zu erfüllen.

8. Bei der Anzeige der Minuten und Sekunden müssen wir die führende Null gegebenenfalls selbst erzeugen, ansonsten erhielten wir beispielsweise statt 11:09:08 die ungewöhnliche Zeitanzeige 11:9:8.

Andere Bibliotheken bieten im Zusammenhang mit dem DS3231-Modul auch die Möglichkeit, dessen Temperatursensor auszulesen, einen Alarm zu programmieren oder den auf der Platine befindlichen Speicher zu benutzen. Bei Interesse finden Sie entsprechende Informationen in der Dokumentation der jeweiligen Bibliothek.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 10

Praxisprojekt: LCD-Uhr mit Thermometer

10.1 Idee

In einem weiteren praktischen Beispiel wollen wir uns eine Digitaluhr bauen, welche auf einem zweizeiligen LCD das Datum, die aktuelle Uhrzeit sowie Temperatur und Luftfeuchte angezeigt. Über zwei Bedienknöpfe soll es möglich sein, die Uhr zu stellen sowie die Minimal- und Maximalwerte für Temperatur und Luftfeuchtigkeit abzulesen.

Für die Bedienung planen wir folgendes Schema:

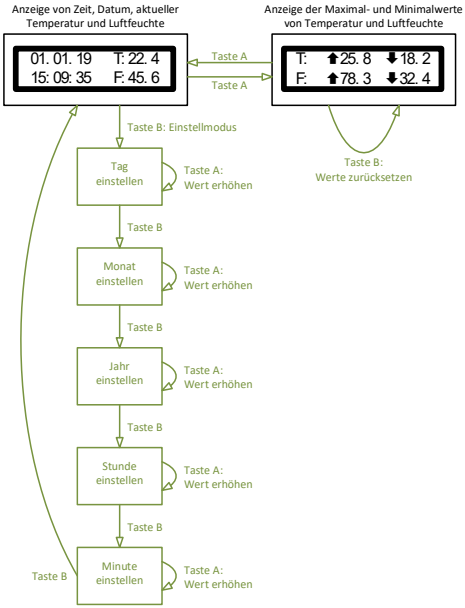


Abb. 10.1 geplante Menüabfolge

Als Temperatur- und Feuchtesensor benutzen wir einen DHT22, die Zeitmessung erledigt ein Echtzeitmodul vom Typ DS3231, als Display eignet sich ein LCD mit dem LCM1602-Backpack – alle diese Komponenten haben wir bereits kennengelernt.

10.2 Stromlaufplan

Da sowohl das LCD-Backpack als auch das Echtzeitmodul per I^2C angesteuert werden, können wir ihre Anschlüsse einfach parallelschalten:

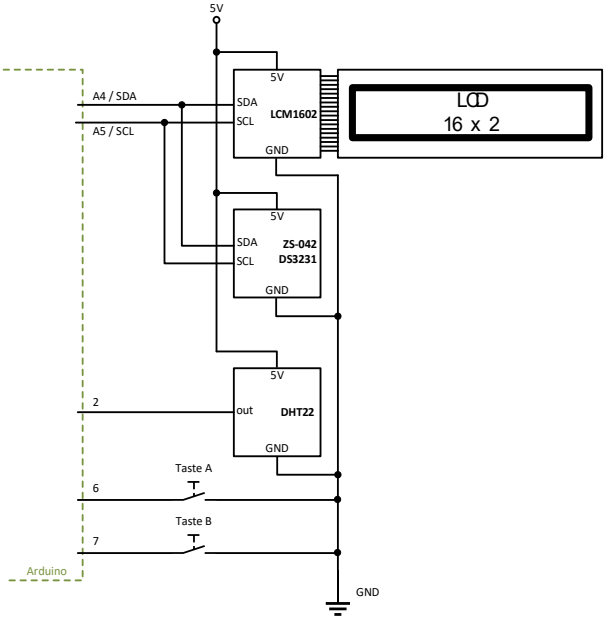


Abb. 10.2 Der Stromlaufplan zeigt die Verdrahtung der drei Module

10.3 Versuchsaufbau

Wir benötigen:

- 1 kleines Breadboard

- 1 Arduino UNO
- 1 LCD (16x2 Zeichen) mit LCM1602-Backpack
- 1 Echtzeitmodul ZS-042/DS3231
- 1 Temperatur- und Feuchtesensor DHT22
- 2 Taster
- Verbindungsdrähte

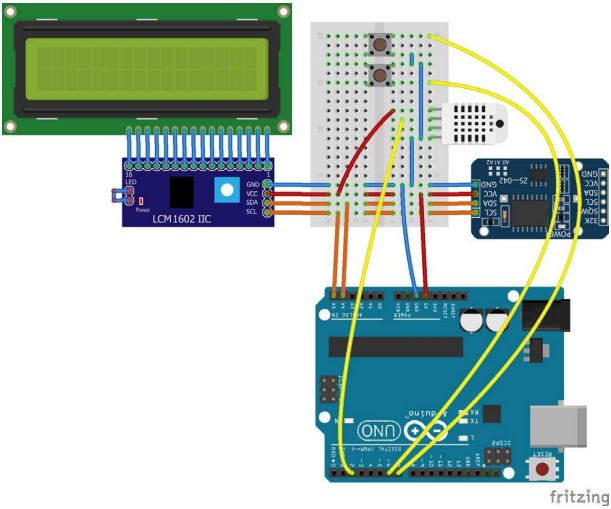


Abb. 10.3 Um Fehler zu vermeiden, sollte man die Verdrahtung anhand des Stromlaufplans vornehmen

10.4 Programmcode

Bei diesem Projekt ist der Sketch natürlich etwas umfangreicher:

```
#define TASTE_A 6
#define TASTE_B 7

#include "LiquidCrystal_I2C.h"      // Bibliothek für das LCD
#include "DS3231.h"                 // Bibliothek für das
```

```
                                    // Echtzeitmodul
#include "DHT.h"                    // Bibliothek für den Sensor

LiquidCrystal_I2C LCD(39,16,2);     // Objekt LCD repräsentiert
                                    // das Display

DS3231 Uhr;                         // Objekt Uhr repräsentiert
                                    // das Zeitmodul
DHT dhtSensor(2, DHT22);            // Objekt dhtSensor
                                    // repräsentiert den Sensor

boolean Dummy = false;              // Dummy-Variable für
                                    // Zeitmodul
long Timer = 0;
float Temp; float maxTemp = -100.0; float minTemp = 100.0;
float Feuchte;  float maxFeuchte = 0.0; float minFeuchte = 100.0;

boolean zeigeMinMax = false;        // Betriebsmodus (Zeitanzeige
                                    // / Min-Max-Anzeige)

byte maxPfeil[8] = {B00100,                              // (1)
                    B01110,
                    B11111,
                    B01110,
                    B01110,
                    B01110,
                    B01110,
                    B00000};
byte minPfeil[8] = {B00000,
                    B01110,
                    B01110,
                    B01110,
                    B01110,
                    B11111,
                    B01110,
                    B00100};

void setup()
{
  pinMode(TASTE_A, INPUT_PULLUP);
  pinMode(TASTE_B, INPUT_PULLUP);

  LCD.init();
  LCD.createChar(0, maxPfeil);                           // (2)
  LCD.createChar(1, minPfeil);
  LCD.clear();
  LCD.backlight();
  LCD.setCursor(2,0);
```

```
54
55    dhtSensor.begin();
56  }
57
58  void loop()
59  {
60    if(millis() > Timer)                                        // (3)
61    {
62      Temp = dhtSensor.readTemperature();      // Temperatur und
63      Feuchte = dhtSensor.readHumidity();      // Feuchte auslesen
64
65      if(Temp > maxTemp)                       // Hier werden die
66                                               // Sensorwerte
67        maxTemp = Temp;                        // mit den
68                                               // gespeicherten
69      if(Temp < minTemp)                       // Extremwerten
70                                               // verglichen.
71        minTemp = Temp;                        // Bei Bedarf werden
72                                               // die Extremwerte
73      if(Feuchte > maxFeuchte)                 // angepasst.
74        maxFeuchte = Feuchte;
75      if(Feuchte < minFeuchte)
76        minFeuchte = Feuchte;
77
78      if(zeigeMinMax)                  // Wenn der Betriebsmodus auf
79      {                                // Extremwertanzeige steht,
80        LCD.setCursor(0,0);            // werden die entsprechenden
81        LCD.print("T: ");              // Minimal- und Maximalwerte
82        LCD.print((char)0);            // ausgegeben.
83        LCD.print(maxTemp,1);
84        LCD.setCursor(10,0);
85        LCD.print((char)1);
86        LCD.print(minTemp,1);
87
88        LCD.setCursor(0,1);
89        LCD.print("F: ");
90        LCD.print((char)0);
91        LCD.print(maxFeuchte,1);
92        LCD.setCursor(10,1);
93        LCD.print((char)1);
94        LCD.print(minFeuchte,1);
95      }
96      else                                     // Sonst werden
97                                               // Datum, Uhrzeit
98      {                                        // und die aktuellen
99                                               // Sensorwerte
100       LCD.setCursor(0,0);                    // angezeigt.
101
```

```
102       if(Uhr.getDate() < 10)              // Die Zeitformatierung
103                                           // erfordert
104         LCD.print("0");                   // führende Nullen bei
105                                           // Werten
106       LCD.print(Uhr.getDate());           // kleiner als 10
107
108       LCD.print(".");
109
110       if(Uhr.getMonth(Dummy) < 10)
111         LCD.print("0");
112       LCD.print(Uhr.getMonth(Dummy));       // Das Dummy-Argument
113                                             // resultiert aus
114                                             // einer formalen
115                                             // Anforderung der
116                                             // RTC-Bibliothek;
117                                             // siehe Kapitel 9.11
118       LCD.print(".");
119
120       if(Uhr.getYear() < 10)
121         LCD.print("0");
122       LCD.print(Uhr.getYear());
123
124       LCD.print("  T:");
125       LCD.print(Temp,1);
126
127       LCD.setCursor(0,1);                   // Beginn der zweiten
128                                             // Zeile
129
130       if(Uhr.getHour(Dummy, Dummy) < 10)
131         LCD.print("0");
132       LCD.print(Uhr.getHour(Dummy, Dummy));
133
134       LCD.print(":");
135
136       if(Uhr.getMinute() < 10)
137         LCD.print("0");
138       LCD.print(Uhr.getMinute());
139
140       LCD.print(":");
141
142       if(Uhr.getSecond() < 10)
143         LCD.print("0");
144       LCD.print(Uhr.getSecond());
145
146       LCD.print("  F:");
147       LCD.print(Feuchte,1);
148     }
```

```
149
150     Timer = millis() + 1000;        // nächste Aktualisierung
151                                     // in 1 Sekunde
152   }
153
154   if(!digitalRead(TASTE_B))         // Wird Taste B gedrückt?
155   {
156     while(!digitalRead(TASTE_B));   // warten, bis Taste wieder
157                                     // losgelassen wird
158     if(zeigeMinMax)                 // Wenn Extremwertmodus aktiv
159                                     // ist,
160     {                               // dann jetzt die
161                                     // gespeicherten
162       maxTemp = -100.0; minTemp = 100.0;    // Werte
163                                             // zurücksetzen.
164       maxFeuchte = 0.0; minFeuchte = 100.0;
165       LCD.clear();
166       Timer = 0;
167     }
168     else
169     {                                                       // (4)
170       Uhr.setDate(Einstellen(Uhr.getDate(), 1, 31, 0, 0));
171       Uhr.setMonth(Einstellen(Uhr.getMonth(Dummy), 1, 12, 3, 0));
172       Uhr.setYear(Einstellen(Uhr.getYear(), 19, 29, 6, 0));
173       Uhr.setHour(Einstellen(Uhr.getHour(Dummy, Dummy),
174       0, 23, 0, 1));
175       Uhr.setMinute(Einstellen(Uhr.getMinute(), 0, 59, 3, 1));
176       Uhr.setSecond(0);             // Sekundeneinstellung ist
177                                     // nicht sinnvoll
178     }
179   }
180
181   if(!digitalRead(TASTE_A)) // Falls während des normalen
182                             // Betriebs
183   {                         // die Taste A gedrückt wird:
184
185    while(!digitalRead(TASTE_A));    // warten, bis A wieder
186                                     // losgelassen wird
187     zeigeMinMax = !zeigeMinMax;     // Betriebsmodus umschalten
188     LCD.clear();                    // LCD leeren
189     Timer = 0;                      // sofort Display
190                                     // aktualisieren
191   }
192 }
193
194 byte Einstellen(byte aktuellerWert, byte minimalWert, byte
195 maximalWert, byte posX, byte posY)
196 {
```

```
197   while(digitalRead(TASTE_B))                              // (5)
198   {
199     LCD.setCursor(posX, posY);      // Cursor an die übergebene
200                                     // Position setzen
201
202     if(millis()%1000 > 800)                                  // (6)
203     {
204       LCD.print("__");              // Anzeige von Unterstrichen
205     }                               // für 200ms
206     else
207     {
208       if(aktuellerWert < 10)        // Anzeige des aktuellen
209                                     // Wertes
210         LCD.print("0");             // für 800ms
211       LCD.print(aktuellerWert);
212     }
213
214     if(!digitalRead(TASTE_A))               // Falls Taste A
215                                             // gedrückt wird:
216     {
217       aktuellerWert++;                      // Wert erhöhen
218       if(aktuellerWert > maximalWert)       // Wenn Maximum
219                                             // überschritten,
220         aktuellerWert = minimalWert;        // wieder von vorn
221                                             // beginnen
222
223       while(!digitalRead(TASTE_A));         // warten, bis Taste
224                                             // A losgelassen wird
225     }
226   }
227
228   while(!digitalRead(TASTE_B));             // warten, bis Taste
229                                             // B losgelassen wird
230
231   LCD.setCursor(posX, posY);                // aktuellen Wert
232                                             // ausgeben
233   if(aktuellerWert < 10)                    // (um korrekte
234                                             // Anzeige
235                                             // sicherzustellen,
236     LCD.print("0");                         // falls gerade
237                                             // Unterstriche
238   LCD.print(aktuellerWert);                 // angezeigt wurde)
239
240   return aktuellerWert;                     // eingestellten Wert
241                                             // zurückgeben
242 }
```

1. Die Bibliothek LiquidCrystal_I2C.h erlaubt es, bis zu 8 benutzerdefinierte Symbole für das Display anzulegen. Wir machen davon Gebrauch, da wir den Maximalwert durch einen Pfeil nach oben und den Minimalwert durch einen Pfeil nach unten symbolisieren möchten. Beide Symbole sind nicht im Standard-Zeichensatz enthalten. Um selbst ein Symbol (5 Pixel Breite, 8 Pixel Höhe) anzulegen, erzeugen wir ein Array vom Typ `byte`, welches für jede Pixelreihe ein Byte enthält. 5 Bit dieses Bytes repräsentieren jeweils 5 Pixel nebeneinander. Die übrigen (höchstwertigen) 3 Bit werden nicht ausgewertet. Auf diese Weise lassen sich alle Pixel des gewünschten Symbols einzeln festlegen. Schaut man sich die Deklaration dieses Arrays genau an, kann man im Muster der Einsen und Nullen den Pfeil nach oben erkennen. Das Resultat ist auf dem Displayfoto im nächsten Abschnitt zu sehen.

2. Die Funktion `createChar()` der LiquidCrystal_I2C-Bibliothek erzeugt nun aus dem übergebenen Array ein benutzerdefiniertes Symbol und speichert es auf Platz 0. Es gibt insgesamt 8 Speicherplätze für benutzerdefinierte Symbole (0 bis 7).

3. Die Zeitverzögerung per `Timer`-Variable kennen Sie bereits. Auch in diesem Beispiel genügt es, das Display einmal pro Sekunde zu aktualisieren.

4. Wird die Taste B gedrückt, während die Uhr im Zeitanzeige-Modus ist, beginnt dadurch die Einstellung der Zeit. Nacheinander können die einzelnen Datumswerte eingestellt werden (siehe *Abb. 10.1*). Um dies zu erreichen, wird für jeden Wert die entsprechende `set`-Funktion des `Uhr`-Objektes aufgerufen. Diese Funktion erwartet als Argument einen `byte`-Wert. Hier kommt unsere selbsterstelle Funktion `Einstellen()` ins Spiel. Sie übernimmt die Nutzerabfrage und liefert die eingestellte Zahl als Rückgabewert. Als Argumente benötigt sie den aktuellen Wert, das Minimum, das Maximum sowie die Anzeigeposition auf dem Display (Spalten- und Zeilennummer, wie bei `setCursor()`).

5. Innerhalb der Funktion `Einstellen()` sorgt diese Schleife dafür, dass der aktuell einstellbare Wert blinkt (genau genommen wechselt er zwischen Zahlenanzeige und Unterstrichen hin und her).

Wird währenddessen die Taste A gedrückt, wird der Zahlenwert entsprechend erhöht. Ein Druck auf die Taste B beendet die Schleife (und damit die Einstellung des Wertes).

6. Das Blinken wird dadurch erreicht, dass der Ausdruck `(millis()%1000 > 800)` jeweils für 800 Millisekunden `false` ergibt und dann für 200 Millisekunden `true`. Das Prozentzeichen steht für eine Modulodivision, also für den Rest bei ganzzahliger Division durch 1000 – das heißt in diesem Fall ganz simpel: Das Ergebnis sind die letzten drei Ziffern.

Es ist noch zu beachten, dass dieses Programm keine Plausibilitätsprüfung des Datums vornimmt. Es ist also beispielsweise möglich, den 31. Februar einzustellen. Bei Bedarf könnte man dafür eine zusätzliche Prüfung vor dem Verlassen des Einstellmodus einfügen.

10.5 Resultat

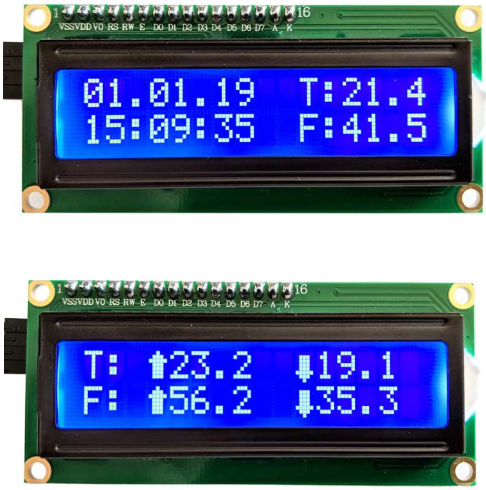


Abb. 10.4 beispielhafte Displayanzeige im Zeitmodus (links) und Extremwertmodus (rechts)

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 11
Aktoren

Nachdem wir verschiedene Möglichkeiten der Ein- und Ausgabe von Daten betrachtet haben, sollen nun einige Aktoren in den Fokus rücken. Mit ihnen kann der Arduino durch bewegliche Komponenten aktiv seine Umwelt manipulieren.

11.1 Relais

Sollen größere Lasten (beispielsweise Geräte an Netzspannung) geschaltet werden, bieten sich Relais an. Sie stellen eine Abhängigkeit zwischen zwei Stromkreisen her, welche nur auf einer mechanischen Schaltverbindung basiert. Dadurch wird eine galvanische Trennung beider Stromkreise erreicht. Insbesondere wenn beide Kreise durch fremdartige Spannungsquellen gespeist werden, ist dies vorteilhaft. Obwohl es damit auch möglich ist, im Laststromkreis Geräte unter Netzspannung zu schalten, sollten Sie dies nur durchführen, wenn Sie die erforderliche Qualifikation besitzen. Wir werden uns auch in diesem Versuch wieder auf ungefährliche 5 Volt beschränken.

Die Funktionsweise von Relais haben wir bereits in Kapitel 3.1.2.6 kennengelernt. Dabei wurde auch klar, dass zur Ansteuerung üblicherweise noch ein Transistor und eine Freilaufdiode notwendig sind. Das Relais-Modul KY-019 stellt all dies schon bereit.

Abb. 11.1 Das Relais-Modul KY-019 kann direkt vom Arduino angesteuert werden

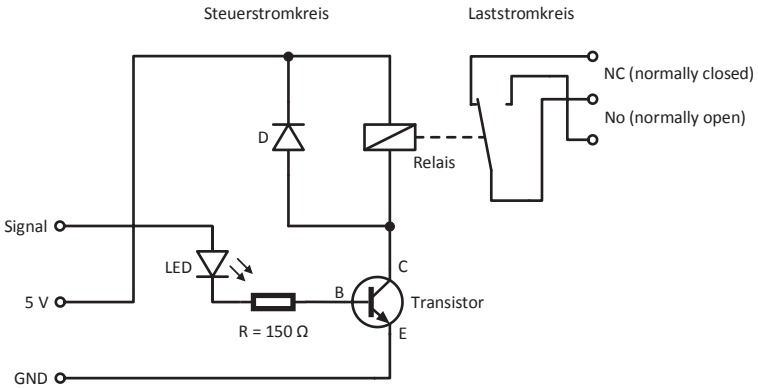


Abb. 11.2 Der Stromlaufplan des Moduls entspricht dem Vorschlag aus Kapitel 3.1.2.6. Es wurde lediglich eine LED ergänzt und der Laststromkreis erweitert.

Der Stromlaufplan des Steuerstromkreises wurde bereits diskutiert. Im Laststromkreis schaltet das Relais zwischen zwei Kontakten hin- und her. Der NC-Kontakt ist im Ausgangszustand bereits geschlossen, der NO-Kontakt ist offen. Wird das Relais angesteuert, wechseln diese in den jeweils umgekehrten Zustand. So könnte man zum Beispiel einen Wechselblinker realisieren.

Die Ansteuerung ist trivial: Der Signal-Pin kann direkt mit einem Ausgangspin des Arduino verbunden werden. Ist dieser LOW, sperrt der Transistor und das Relais bleibt im Ausgangszustand. Schaltet der Pin auf HIGH, öffnet der Transistor den Kollektor-Eingang und das Relais schaltet um. Die LED im Signalweg dient lediglich als Indikator. Als Signal genügt bei diesem Modul eine Spannung ab 3 Volt.

Im folgenden kleinen Beispiel wollen wir das Relais benutzen, um ein Lämpchen eines anderen Stromkreises einzuschalten, wenn das Umgebungslicht einen bestimmten Wert unterschreitet. Der Fotowiderstand wird dazu genau wie in Kapitel 9.3 verwendet.

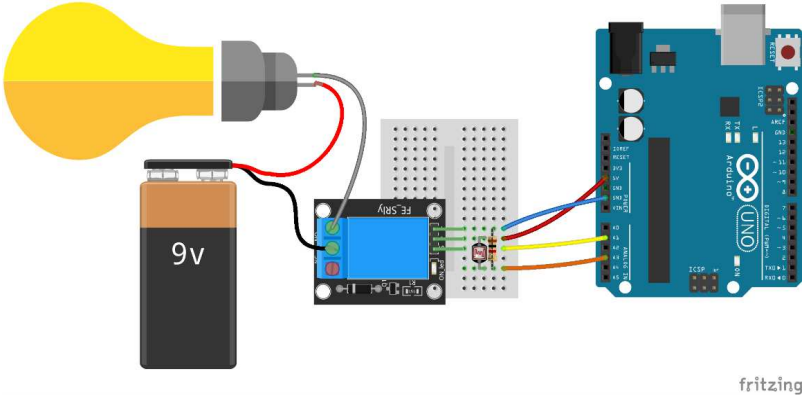


Abb. 11.3 Den Laststromkreis (links) können Sie für den Test auch weglassen: Die Funktion des Relais ist deutlich hörbar, zudem dient die LED des Moduls als Indikator.

Allerdings müssen wir hierbei auch eine mögliche Rückwirkung beachten: Scheint das Licht des Lämpchens auf den Fotowiderstand, könnte dies wiederum das Ausschalten bewirken, da der gemessene Helligkeitswert dann zu hoch ist. Im Resultat würde die Schaltung bei geringem Umgebungslicht ein ungewolltes Blinken erzeugen. Um dem entgegenzuwirken, sollte die Lichtmessung möglichst abseits des Lämpchens geschehen. Außerdem können wir den Sketch so gestalten, dass Ein- und Ausschalten bei unterschiedlichen Schwellenwerten er-

folgen. Man nennt dieses Prinzip *Hysterese*[1]. Es reduziert die Gefahr ungewollter Schaltvorgänge.

```
#define SENSOR 3
#define RELAIS 15

boolean Relaiszustand = false;

void setup() {
  pinMode(RELAIS, OUTPUT);
}

void loop() {
  if(analogRead(SENSOR) < 400)
    Relaiszustand = true;

  if(analogRead(SENSOR) > 600)
    Relaiszustand = false;

  digitalWrite(RELAIS, Relaiszustand);
  delay(1000);
}
```

Der analoge Eingang A1 wird aufgrund des kürzeren Kabelweges hier als digitaler Ausgang benutzt, seine Pinnummer ist daher 15. Die Schwellenwerte wurden willkürlich gewählt. Sie können natürlich experimentell auch andere Werte testen. Im Kapitel 11.6 werden wir dieses Beispiel aufgreifen und den Wert dynamisch anpassen.

1 *Hysterese* bedeutet wörtlich *Nachwirkung*. In unserem konkreten Fall heißt das, dass für den Schaltzustand des Relais nicht nur der aktuelle Messwert, sondern auch die Vergangenheit relevant ist. Liegt der Messwert beispielsweise bei 500, so könnte das Relais eingeschaltet (wenn der Wert vorher weit darunter lag) oder ausgeschaltet sein (wenn vergangene Werte weit darüber lagen). Man nutzt Hysterese in zahlreichen weiteren Anwendungen, zum Beispiel beim Bimetall-Thermostat oder bei Füllstandsensoren.

11.2 Gleichstrommotor

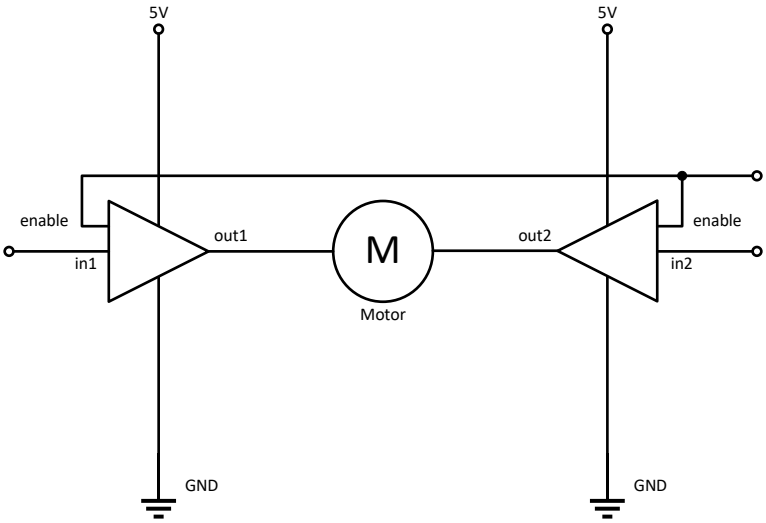
Die einfachste Art von Hobby-Motoren sind Gleichstrommotoren. Sie kennen sie aus batteriebetriebenen Ventilatoren oder Computer-Lüftern.

Abb. 11.4 ein Gleichstrommotor mit Ventilator-Aufsatz

Der Strom fließt in einem solchen Motor durch eine drehbare Spule, welche sich im Feld eines Dauermagneten befindet. So entsteht eine Kraft auf die Spule, welche in einer Drehung resultiert. Um eine konstante Rotationsbewegung zu erhalten, wird die Stromrichtung in der Spule über Schleifkontakte nach jeder halben Umdrehung umgepolt.

Ein Gleichstrommotor benötigt eine relativ hohe Stromstärke: Für Bastel-Motoren sind Werte zwischen 0,1 und 0,5 Ampere üblich. Daher ist die Ansteuerung nicht direkt durch einen Arduino-Ausgangspin möglich. Man verwendet stattdessen eine Transistorschaltung, wie wir sie bereits beim Relais kennengelernt haben. Wir könnten einen Gleichstrommotor also einfach statt des Relais in eine Schaltung wie in *Abb. 3.19* beziehungsweise *Abb. 11.2* einbauen. Per Pulsweitenmodulation ließe sich dann auch die Leistung regulieren. Allerdings lässt sich mit dieser Schaltung nicht die Drehrichtung des Motors ändern, denn dafür wäre eine Umpolung nötig.

Für den Richtungswechsel müssen wir demnach beide Pole des Motors einzeln ansteuern können. Wären Gleichstrommotoren nicht auf vergleichsweise hohe Stromstärken angewiesen, könnten wir sie einfach an zwei Ausgangspins des Arduino anschließen, einen davon auf `HIGH` schalten und den anderen auf `LOW`. Für einen Richtungswechsel müssten nur beide Ausgänge entsprechend umgeschaltet werden. Die Ausgangspins sind jedoch auf eine maximale Stromstärke von 20 Milliampere begrenzt. Es gibt aber spezielle Transistorschaltungen, welche das Signal eines Ausgangspins verstärken können, indem sie höhere Stromstärken bereitstellen. In der folgenden Abbildung sind sie als Dreiecke dargestellt. Wird ihr Eingang mit Masse verbunden (LOW) schalten sie ihren Ausgang ebenfalls auf Masse; ist der Eingang HIGH, wird auch der Ausgang mit der Betriebsspannung verbunden. Im Beispiel ist ein Gleichstrommotor mit beiden Anschlüssen an einen solchen sogenannten Treiber angeschlossen. Durch wechselseitiges Anlegen von HIGH beziehungsweise LOW an gegenüberliegenden Eingängen lässt sich die Motordrehrichtung entsprechend Steuern. Aufgrund der Ähnlichkeit zum Buchstaben H wird diese Schaltung auch als *H-Brücke* bezeichnet.



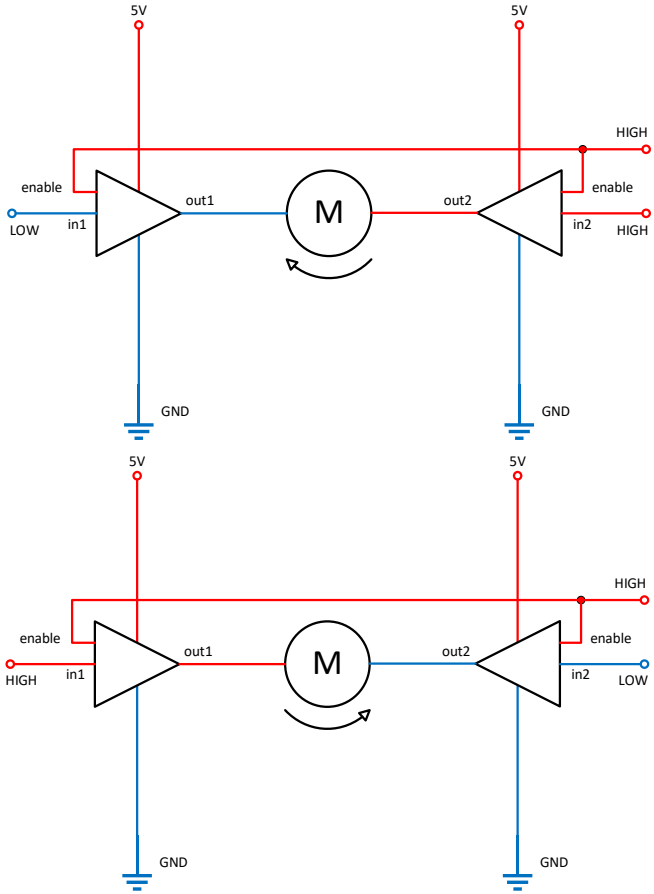


Abb. 11.5 Schaltung einer H-Brücke sowie Signalzustände für Links- und Rechtslauf

Die Treiber im Schaltbild haben noch einen zusätzlichen Eingang, welcher mit *enable* gekennzeichnet ist. Soll der Motor laufen, muss dort ein HIGH-Signal anliegen. Setzt man ihn auf LOW, wird der Ausgang der Treiber komplett abgeschaltet, er verhält sich dann hochohmig (wie ein Eingangspin am Arduino). Man kann somit über die Signale an den Eingangspins (in1 und in2) die Drehrichtung festlegen

und dann die Leistung regulieren, indem man am Enable-Pin ein PWM-Signal anlegt.

H-Brücken sind als integrierte Schaltkreise erhältlich, sie haben also ein ähnliches Gehäuse wie der Mikrocontroller auf unserer Arduino-Platine. Ein gebräuchliches Modell ist der L293D, er stellt 4 separate Treiber bereit – somit kann man zwei unabhängige Motoren ansteuern. Verzichtet man auf den Laufrichtungswechsel, lassen sich sogar 4 Motoren anschließen (indem man nur noch einen Draht des Motors mit dem Treiber verbindet und den anderen mit der Betriebsspannung oder Masse). Ein weiterer Vorteil des L293D sind die integrierten Freilaufdioden, welche die Treiber vor Spannungsspitzen aufgrund der Induktivität (siehe Abschnitt 3.1.2.6) schützen.

Im folgenden Beispiel wollen wir einen Gleichstrommotor über ein Potentiometer steuern. Am Rechtsanschlag des Potis soll der Motor mit Maximalgeschwindigkeit laufen. In Mittelstellung soll er Stillstehen und am linken Anschlag mit Maximalgeschwindigkeit in die Gegenrichtung drehen.

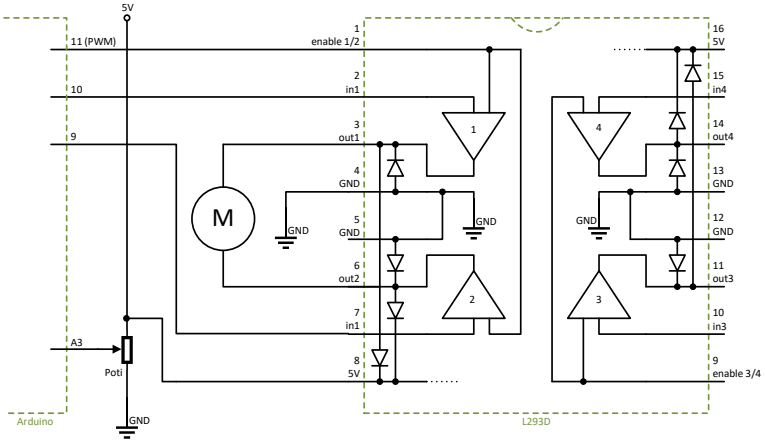


Abb. 11.6 Wir nutzen nur zwei der vier integrierten Treiber des L293D

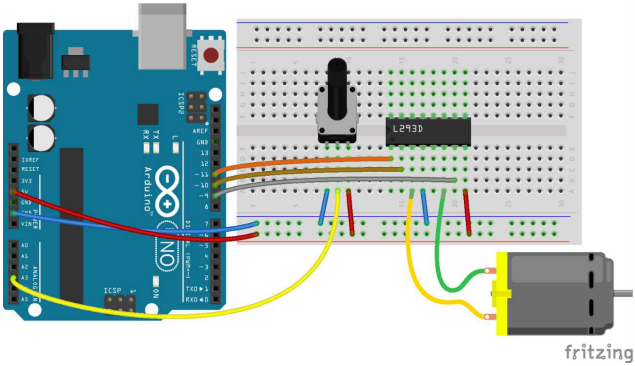


Abb. 11.7 Bei integrierten Schaltkreisen ist auf die richtige Ausrichtung zu achten, dazu ist an einer Seite eine kleine Kerbe angebracht (hier links neben dem Schriftzug „L293D“). Ab dieser Kerbe zählt man die Pins entgegen dem Uhrzeigersinn, beginnend mit 1. Die entsprechenden Pinnummern finden Sie auch im Stromlaufplan.

Der Sketch unterscheidet nun je nach Stellung des Potentiometers zwischen Vorwärts- und Rückwärtslauf und reguliert die Motorleistung per PWM-Signal:


```
#define POTI 3
#define M_PWM 11
#define M_IN1 10
#define M_IN2 9

int PotiWert = 0;
byte MotorLeistung = 0;
boolean rueckwaerts = false;

void setup() {
  pinMode(M_PWM, OUTPUT);
  pinMode(M_IN1, OUTPUT);
  pinMode(M_IN1, OUTPUT);
}

void loop() {
  PotiWert = analogRead(POTI);

  if(PotiWert >= 512)               // 512...1023 -> vorwärts
  {
```

```
    rueckwaerts = false;
    MotorLeistung = (PotiWert-512) / 2;     // ergibt Zahl
                                            // zwischen 0 und 255
  }
  else
  {                                         // 0...511 ->
                                            // rückwärts
    rueckwaerts = true;
    MotorLeistung = (511-PotiWert) / 2;     // ergibt Zahl
                                            // zwischen 0 und 255
  }

  if(rueckwaerts)
  {                                         // Treiber-Eingänge
                                            // setzen
    digitalWrite(M_IN1, HIGH);              // für Rückwärtslauf
    digitalWrite(M_IN2, LOW);
  }
  else
  {
    digitalWrite(M_IN1, LOW);               // bzw. Vorwärtslauf
    digitalWrite(M_IN2, HIGH);
  }

  analogWrite(M_PWM, MotorLeistung);        // Leistung per PWM
                                            // am enable-Pin
                                            // des L293D
                                            // regulieren
}
```

Beim Test werden Sie feststellen, dass besonders im mittleren Leistungsbereich ein leises Fiepen zu hören ist. Dies liegt daran, dass die Frequenz der Pulsweitenmodulation des Arduino im für Menschen hörbaren Bereich liegt. Solange sie nur als Spannung vorliegt (wie bei der gedimmten LED), können wir sie nicht wahrnehmen. Den Motor versetzt sie jedoch in Schwingungen, welche sich an die Luft übertragen und somit hörbar werden. Vielleicht haben Sie ein ähnliches Geräusch auch schon einmal bei anfahrenden Straßenbahnen oder S-Bahnen gehört. Es handelt sich um den gleichen Effekt.

11.3 Servomotor

In manchen Anwendungen ist es notwendig, die Position einer Drehachse exakt einzustellen. Denken Sie beispielsweise an die Lenkung bei

einem Fahrzeug oder die Stellung der Tragflächen bei einem Modellflugzeug. Dies sind typische Einsatzgebiete für Servomotoren. Sie bieten die Möglichkeit, mit ihrer Achse einen bestimmten Winkel exakt anzufahren.

Der Begriff Servomotor steht dabei für den Verbund einer Ansteuerungs- und Antriebseinheit. Im Inneren befindet sich ein Gleichstrommotor, welcher über ein Getriebe die Ausgangswelle antreibt. Diese ist mit einem Winkelsensor verbunden. Eine Steuer-Elektronik vergleicht den Ausgabewert des Winkel-Sensors mit dem von außen vorgegebenen Sollwert und steuert den Motor entsprechend, um die gewünschte Position zu erreichen. Vorteil dieser Feedback-Technik ist, dass die Position zuverlässig angefahren und gehalten wird, auch wenn dabei Störungen von außen (zum Beispiel Windkräfte beim Modellflug) wirken, die einen normalen Gleichstrommotor aus seiner Position drehen würden.

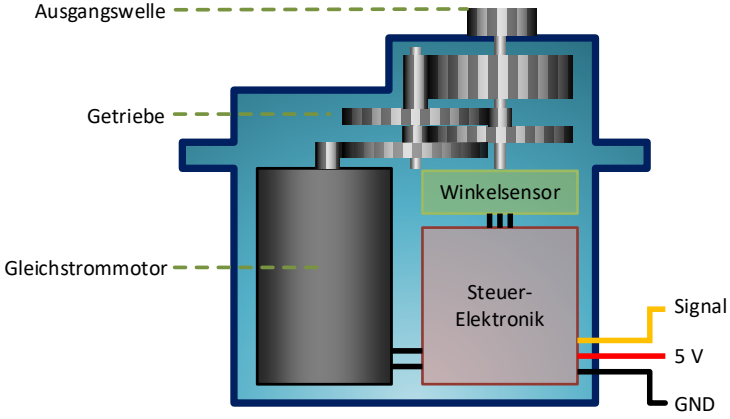


Abb. 11.8 Aufbau des im Modellbau beliebten Servomotor-Modells SG90

Da als Winkelsensor meist einfache Potentiometer verwendet werden, sind viele Servomotoren auf einen Wirkbereich von deutlich unter 360 Grad beschränkt, sie können also keine vollständigen Umdrehungen

ausführen. Wir verwenden im Folgenden das Modell SG90, welches Drehbewegungen im Bereich von 180 Grad ermöglicht.

Der SG90 erwartet ein (für Servos typisches) spezielles pulsweitenmoduliertes Signal, mit dem der Drehwinkel vorgegeben wird. Es besteht aus Impulsen von 1 bis 2 Millisekunden Länge, welche alle 20 Millisekunden ((also 50-mal pro Sekunde, das heißt mit einer festgelegten Frequenz von 50 Hertz) wiederholt werden. Die Länge der Impulse bestimmt den anzufahrenden Winkel, wobei ein Impuls von einer Millisekunde für *0 Grad* steht und zwei Millisekunden *180 Grad* symbolisieren. Es sind auch beliebige Zwischenwerte möglich.

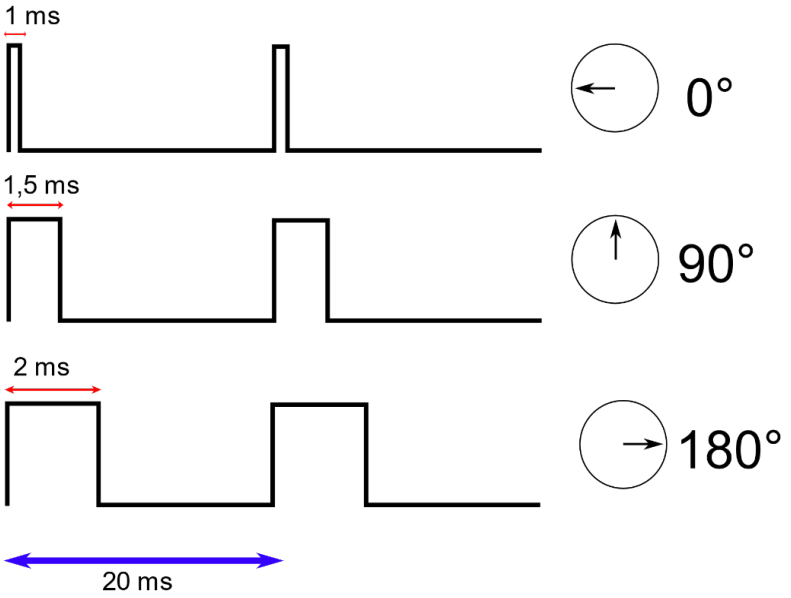


Abb. 11.9 Der SG90 wird über eine Form der Pulsweitenmodulation gesteuert

Für einen kleinen Test wollen wir den analog eingelesenen Helligkeitswert des Fotosensors ausgeben, indem wir den Servomotor wie ein analoges Zeigerinstrument benutzen.

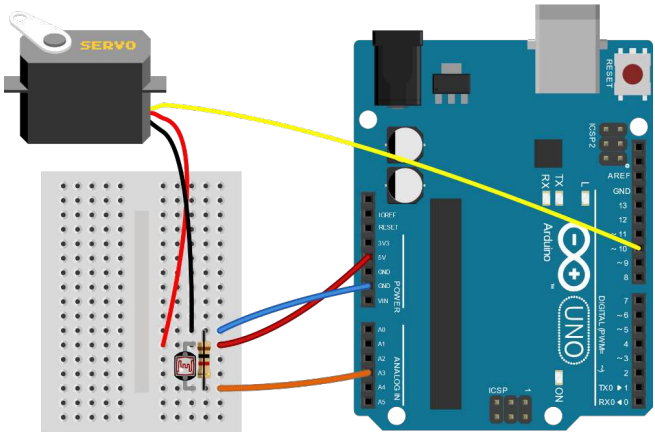


Abb. 11.10 Der Servo soll als Zeigerinstrument für die gemessene Helligkeit dienen

Da die im Arduino bereits hardwaremäßig vorgesehene Pulsweitenmodulation eine deutlich kürzere Impulsfolge hat, können wir diese nicht direkt zur Ansteuerung verwenden. Wir nutzen stattdessen die Bibliothek Servo.h, welche in der Arduino IDE bereits vorinstalliert ist.


```
#define SERVOPIN 10  // digitaler Ausgang für Servo
#define SENSORPIN 3  // analoger Eingang für Fotowiderstand

#include "Servo.h"

Servo Servomotor;    // erstellt Objekt "Servomotor" aus der
                     // Klasse "Servo"
int Sensorwert;
byte Servowinkel;

void setup() {
  Servomotor.attach(SERVOPIN);    // initialisiert die Verbindung
}                                 // zum Servomotor

void loop() {
  Sensorwert = analogRead(SENSORPIN);
  Servowinkel = map(Sensorwert, 0, 1023, 0, 180);          // (1)
  Servomotor.write(Servowinkel);                           // (2)
  delay(50);
}
```

1. Die Funktion `map()` haben Sie bisher noch nicht kennengelernt. Sie dient dazu, einen Zahlenwert aus einem bestimmten Wertebereich in einen anderen Wertebereich umzurechnen (und gibt ihn dann als Rückgabewert aus). Im hiesigen Fall soll sie die Zahl `Sensorwert` aus dem Bereich von 0 bis 1023 in den Bereich von 0 bis 180 umrechnen. Sicher erinnern Sie sich: Beginnend ab Kapitel 5.5.2 haben wir mehrmals einen eingelesenen Analogwert (0 ... 1023) durch 4 geteilt, um ihn an einem PWM-Pin (0 ... 255) ausgeben zu können. Wir hätten stattdessen auch schreiben können `map(Variable, 0, 1023, 0, 255)` – da aber der neue Wertebereich genau ein Viertel des alten Bereiches abbildete, war die Division durch 4 die simplere Lösung. In unserem jetzigen Fall wäre eine Rechnung über den klassischen Dreisatz nötig – hier bietet die Funktion `map()` die komfortablere Lösung.
2. Hier erfolgt schließlich die Ausgabe des Signals an den Servo mittels der Bibliotheksfunktion `write()`. Zulässig sind dabei Werte zwischen 0 und 180, welche direkt den angestrebten Winkel repräsentieren.

Nach dem Hochladen des Programms können Sie beobachten, wie der Servo bei Helligkeitsänderungen am Fotowiderstand reagiert und die Helligkeit wie ein Zeigerinstrument anzeigt.

11.4 Schrittmotor

Beide bisher betrachteten Antriebe verwendeten Gleichstrommotoren zur Erzeugung der Bewegung. Diese bestehen aus einer drehbar gelagerten Spule, welche sich im Feld eines feststehenden Dauermagneten befindet. Legt man eine Spannung an, richtet sich die Spule an diesem Magnetfeld aus. Durch einen Schleifkontakt wird dabei die Polarität der Spule gewechselt, so dass sie sich nun entgegengesetzt ausrichtet. Dieser Vorgang wiederholt sich ständig und führt dadurch zur Rotation.

Nachteilig ist beim Gleichstrommotor, dass sich sein Drehwinkel nicht genau steuern lässt. Der Servomotor hingegen kann keine kompletten Umdrehungen ausführen. Manche Anwendungen erfordern jedoch

beides: Zielgenaue Positionierung, auch über mehrere Umdrehungen hinweg. Beispiele finden sich in der Antriebsmechanik von Druckern, Plottern und zahlreichen weiteren Geräten.

In solchen Fällen kommen Schrittmotoren zur Anwendung. Im Gegensatz zum Gleichstrommotor rotiert hierbei der Dauermagnet, während die Spulen ruhen. Eine schleifringgesteuerte Umpolung findet nicht statt. Somit verharrt der Motor in einer bestimmten Position, bis das Magnetfeld durch Änderung des Stromflusses wechselt. Dabei kommen meist mehrere Spulen zum Einsatz, wodurch eine genauere Positionierung und bessere Laufruhe erreicht werden kann.

Es gibt Schrittmotoren in zahlreichen Ausführungen, wir betrachten beispielhaft die Funktionsweise des im Hobbybereich häufig anzutreffenden Modells 28BJY.

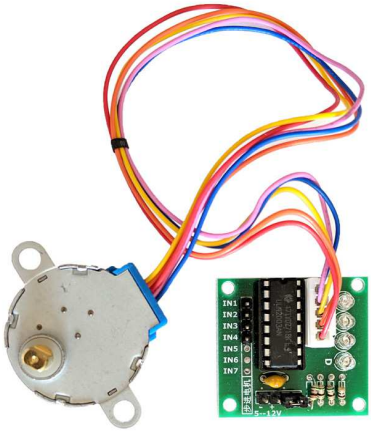


Abb. 11.11 Schrittmotor 28BYJ mit mitgelieferter ULN2003A-Treiberplatine

Er verfügt über zwei Spulen, welche rechtwinklig zueinander angeordnet sind. Man könnte nun den vom Gleichstrommotor bekannten L293D-Schaltkreis nutzen, um beide Spulen in beliebigen Stromrichtungen anzusteuern, man spricht dann auch von einer *bipolaren*

Arbeitsweise. Der 28BYJ bietet allerdings auch die Möglichkeit, ihn *unipolar* zu betreiben, dies wollen wir im Folgenden betrachten. Der Vorteil der unipolaren Beschaltung liegt darin, dass man keine H-Brücke benötigt, weil keine Umpolung stattfindet.

Die folgende Grafik zeigt beispielhaft die Beschaltung des Schrittmotors in den nacheinander erfolgenden Phasen einer kompletten Umdrehung. Die mittleren Anschlüsse beider Spulen sind in der unipolaren Beschaltung stets mit der Betriebsspannung verbunden. Je nach Phase wird ein bestimmter Abgriff mit Masse verbunden. Der dann fließende Strom erzeugt ein magnetisches Feld und der Rotor strebt danach, sich daran auszurichten. Sicher erinnern Sie sich noch an die Regel „Gegensätze ziehen sich an" – so wird der Nordpol (rot) des Rotors also vom Südpol (grün) des Elektromagneten angezogen und umgekehrt.

Die Richtungsumkehr des Magnetfeldes erfolgt dadurch, dass der Strom vom Mittellabgriff aus in eine andere Richtung fließt (vergleiche Phase 1 und 3).

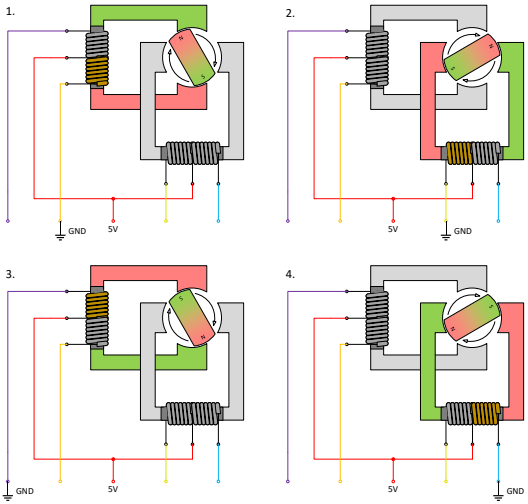


Abb. 11.12 Phasen einer Rotation am 28BYJ bei unipolarer Beschaltung. In bipolarer Arbeitsweise würde der rote Anschluss nicht genutzt.

Es ist auch möglich, beide Spulen gleichzeitig anzusteuern. Dann erfolgt eine Rotation durch das gleichzeitige Wirken von Phase 1 und 2, gefolgt von 2 und 3, dann 3 und 4 und schließlich 4 und 1. In diesem Fall verdoppelt sich die aufgenommene Stromstärke, aber auch das Drehmoment steigt, der Motor hat also mehr Kraft. Für den Rückwärtslauf werden die Phasen entsprechend in umgekehrter Reihenfolge durchlaufen.

Die vier Abgriffe, welche abwechselnd mit Masse verbunden werden, könnten wir jeweils durch die gleiche Schaltung ansteuern wie das Relais in *Abb. 3.19*. Dabei dient ein Transistor als „Ventil" – je nach Steuersignal sperrt er den Stromfluss (wie bei einem Abgriff, an den nichts angeschlossen ist) oder er schaltet durch und stellt damit die Masse-Verbindung her.

Da eine solche Schaltung recht häufig gebraucht wird, gibt es sie auch in Form eines integrierten Schaltkreises: Der ULN2003A fasst genau diese Schaltung gleich in siebenfacher Ausführung in einem Gehäuse zusammen. Sein Verhalten ist entsprechend simpel: Liegt am Eingang eine Spannung an, wird der entsprechende Ausgang auf Masse durchgeschaltet, fehlt die Eingangsspannung, sperrt der Ausgang. Neben den Transistoren sind auch die notwendigen Basiswiderstände und Freilaufdioden (siehe Kapitel 3.1.2.6) bereits in diesem IC (*Integrated Circuit* – integrierter Schaltkreis) enthalten.

Unser Schrittmotor wurde mit einem Treibermodul geliefert, welches diesen integrierten Schaltkreis verwendet. Da wir nur vier Anschlüsse benötigen, bleiben drei unbenutzt. Zudem verfügt die Platine über vier LEDs, welche als Indikator dienen, ob ein bestimmter Pin gerade aktiv ist.

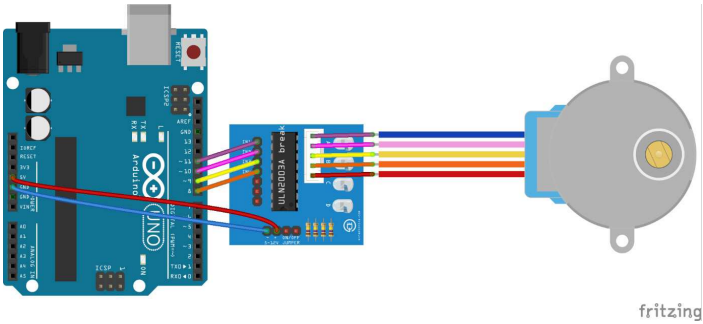


Abb. 11.13 Der 28BYJ-Schrittmotor mit Treiberplatine, welche sich durch beliebige Ausgangspins des Arduino ansteuern lässt

Da die Ansteuerung lediglich aus zyklischem Umschalten der Ausgabepins besteht, könnten wir dies auch einfach über `digitalWrite()`-Befehle abhandeln. Wir werfen dennoch einen Blick auf die bereits vorinstallierte Bibliothek Stepper.h, da sie das Arbeiten mit dem Schrittmotor etwas komfortabler macht.

Im Voraus sei noch erwähnt, dass der 28BYJ über ein integriertes Getriebe verfügt. Die Motorwelle wird also intern zunächst über mehrere Zahnräder geführt, welche die Drehzahl reduzieren, aber das Drehmoment (und damit die „Kraft“ des Motors) erhöhen. Diese sogenannte Untersetzung bedingt, dass 2048 Schritte (Phasen) durchlaufen werden müssen, bis die Ausgangswelle eine komplette Umdrehung ausgeführt hat. Je nach Motortyp unterscheidet sich dieser Wert.

```
1  #define SCHRITTEPROUMDREHUNG 2048
2
3  #include "Stepper.h"
4
5  Stepper Schrittmotor(SCHRITTEPROUMDREHUNG, 8, 10, 9, 11);   // (1)
6
7  void setup() {                                               // (2)
8  }
9
10 void loop() {
```

```
11    Schrittmotor.setSpeed(5);                  // Geschwindigkeit
12                                               // in Umdrehungen
13                                               // pro Minute
14    Schrittmotor.step(SCHRITTEPROUMDREHUNG);   // komplette Drehung
15                                               // nach rechts
16    delay(500);
17
18    Schrittmotor.setSpeed(1);                    // sehr langsame
19    Schrittmotor.step(-SCHRITTEPROUMDREHUNG/4);  // Vierteldrehung
20                                                 // nach links
21    delay(500);
22  }
```

1. Das Objekt `Schrittmotor` als Instanz der Klasse `Stepper` repräsentiert unseren Motor. Bei der Erstellung des Objektes werden 5 Argumente übergeben: Die Anzahl der Schritte, bis eine vollständige Umdrehung erfolgt, sowie die 4 Anschlusspins. Dabei gehören die ersten beiden zu einer Spule und die letzten beiden zur anderen, daraus ergibt sich hier die ungewohnte Reihenfolge.
2. Die Initialisierung führt die Klasse `Stepper` bereits bei der Erstellung des Objektes durch, dazu gehört auch die Pinkonfiguration per `pinMode()`. Wir haben in diesem Fall also in der `setup()`-Routine tatsächlich nichts mehr zu tun.

Erwartungsgemäß führt der Beispielsketch zunächst eine komplette Umdrehung und dann eine langsame Vierteldrehung in die Gegenrichtung aus. Die Positionen werden dabei völlig exakt angefahren, selbst wenn man die Welle mit der Hand leicht hemmt. Dies ist der große Vorteil von Schrittmotoren.

Allerdings kann es problematisch sein, dass im Gegensatz zum Servomotor keine absolute Positionsbestimmung möglich ist. Wir können mit obigem Programm zwar die Welle um exakt eine Vierteldrehung bewegen, jedoch nur relativ zu ihrer Ausgangsposition – wir können hingegen nicht die Position „oben“ beziehungsweise „12 Uhr“ anfahren. Soll dies ermöglicht werden, ist über einen Sensor die Position der Welle zu erfassen. Es genügt jedoch, dies einmalig zu tun. So könnte man bei Systemstart die Welle eine komplette Drehung ausführen lassen und gleichzeitig über einen Hall-Sensor einen magnetisch markier-

ten Punkt auf einer angesteckten Scheibe erkennen. Daraus ergibt sich dann eine eindeutige momentane Position, von der aus alle weiteren Stellungen angefahren werden können.

11.5 Elektromagnet

Obwohl Motoren zweifellos die wichtigste Rolle spielen, wenn es um steuerbare Bewegungen geht, soll dennoch auch ihr „kleiner Bruder", der Elektromagnet, nicht unerwähnt bleiben. In Kapitel 3.1.2.6 haben wir bereits die Spule kennengelernt. Formt man den von ihr umwickelten Kern so, dass ein offener Ring entsteht, welcher durch einen Eisenkörper geschlossen werden kann, erhält man einen kraftvollen Elektromagneten. Die Anziehungskraft (Reluktanzkraft) liegt darin begründet, dass ferromagnetische Stoffe[2] stets in Richtung hoher Feldstärken ("in das Magnetfeld hinein") gezogen werden.

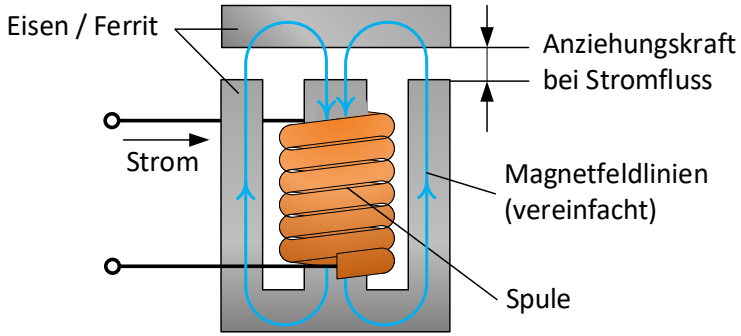


Abb. 11.14 Funktionsprinzip eines Elektromagneten: Das Magnetfeld einer Spule wird in einem unterbrochenen Eisenkörper so geleitet, dass an der Lücke große Anziehungskräfte entstehen.

2 Ferromagnetische Stoffe sind Metalle, welche in besonders starker Wechselwirkung mit magnetischen Feldern stehen. Dazu gehören beispielsweise Eisen, Nickel und Cobalt – nicht jedoch Kupfer oder Aluminium.

Diese Kraft wird umso größer, je höher die Stromstärke und je kleiner der Abstand ist. Außerdem sollte die angezogene Metallplatte mindestens die gleiche Stärke haben wie der Rest des Aufbaus. Wäre das angezogene Plättchen in der oberen Abbildung beispielsweise nur halb so dick, böte es im Vergleich zum restlichen Metallkörper deutlich weniger Querschnittsfläche. Diese Fläche ist jedoch relevant für den magnetischen Fluss, welcher direkt mit der Anziehungskraft zusammenhängt. Außerdem gilt es zu beachten, dass Spulen mitunter hohe Stromstärken erfordern, welche die Ausgangspins von Mikrocontrollern schnell überlasten können. Sie werden daher immer über Transistoren angesteuert, wie wir dies auch bereits beim Relais (welches auch eine Spule enthält) betrachtet haben.

Für Hobbyanwendungen sind fertig aufgebaute Module erhältlich, wie zum Beispiel das Elektromagnet-Modul des Herstellers *Keystudio*.

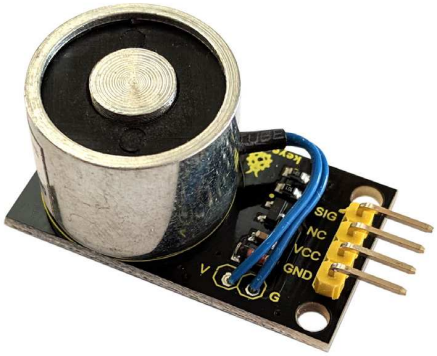


Abb. 11.15 Das Elektromagnet-Modul enthält bereits einen Transistor zur Ansteuerung der Spule, auch die Freilaufdiode ist vorhanden

Die Schaltung auf dem Modul entspricht fast exakt der Beschaltung des Relais-Moduls aus *Abb. 11.2*. Lediglich die Leuchtdiode zur Indikation fehlt. Die verbaute Spule hat einen Widerstand von 10 Ohm. Das heißt, bei 5 Volt Betriebsspannung fließt ein Strom von 0,5 Ampere durch sie. Das wäre bei weitem zu viel für den direkten Anschluss an

einen Ausgangspin (maximal 0,02 Ampere). Verwendet man mehrere dieser Elektromagnete (oder zusätzlich noch andere leistungsintensive Komponenten), ist eine externe Stromversorgung bereitzustellen, zum Beispiel über ein Netzteil. Üblicherweise sind USB-Anschlüsse nur auf 0,5 Ampere ausgelegt, können oft aber bis zu 1 Ampere liefern. Danach erfolgt eine automatische Abschaltung, auf die man sich jedoch nicht verlassen sollte. Versagt sie, droht bei zu hoher Stromstärke ein dauerhafter Schaden am verwendeten USB-Port.

Bei der Nutzung mehrerer separater Spannungsquellen sollte man darauf achten, dass das gesamte Projekt eine gemeinsame Masse-Verbindung benötigt, jedoch die Betriebsspannungen einzelner Abschnitte nicht miteinander verbunden werden sollten. Das folgende Schema zeigt die Verdrahtung für die Nutzung einer separaten Batterie, während der Arduino weiter per USB oder Netzteil mit Strom versorgt wird. Entfällt die Batterie, muss der VCC-Pin des Moduls mit dem 5V-Pin des Arduino verbunden werden.

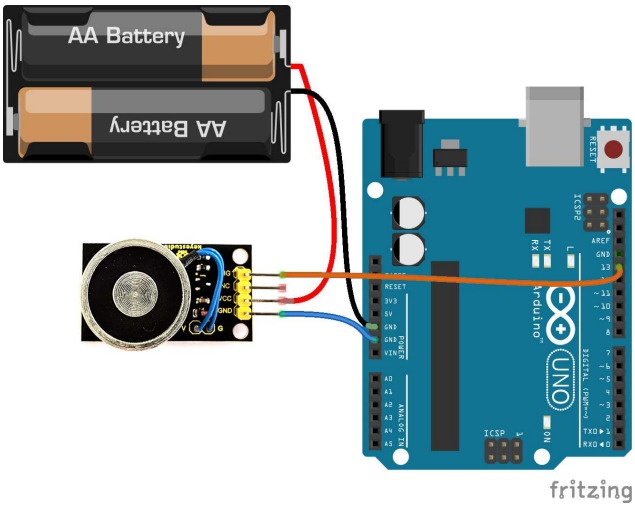


Abb. 11.16 Der Elektromagnet wird genau gleich angesteuert wie das Relais aus Kapitel 11.1, hier allerdings beispielhaft mit einer separaten Spannungsquelle

Als Test-Sketch eignet sich sehr simpel das Blink-Beispiel der Arduino-IDE (Kapitel 4.2). Die blinkende Onboard-LED zeigt dabei, ob der ebenfalls an Pin 13 angeschlossene Elektromagnet gerade aktiv ist oder nicht. Als Testobjekte eignen sich Schlüssel, Besteck und Ähnliches. Natürlich werden nur ferromagnetische Metalle angezogen, wie beispielsweise Eisen oder Stahl. Bei Kupfer oder Aluminium entsteht hingegen keine Anziehungskraft.

Eine typische Anwendung ist das Geschlossenhalten einer Klappe, welche bei abgeschaltetem Magnetfeld aufspringt – oder das Auslösen einer gespannten Feder, welche von einem Haken-Mechanismus zurückgehalten wird. Natürlich sind auch hier wieder der Kreativität keine Grenzen gesetzt.

11.6 Summer

Neben den bereits betrachteten mechanischen Möglichkeiten, auf die Umwelt einzuwirken, kann man sich auch der Akustik bedienen und beispielsweise mit einem Summer (Buzzer) auf sich aufmerksam machen. Sie kennen typische Buzzer-Geräusche sicher von Codeschlössern, Mikrowellen, Armbanduhren und allerlei anderen Geräten mit Nutzerinteraktion.

Das Grundprinzip ist simpel: Eine schwingfähig gelagerte Membran wird elektrisch in Bewegung versetzt. Diese Schwingungen gibt die Membran weiter an die Luft, wodurch sie hörbar werden. Es handelt sich also um das gleiche Prinzip wie bei einem Lautsprecher. Einziger Unterschied: Bei einem Lautsprecher wird die Bewegung elektromagnetisch erzeugt (durch eine Spule), bei einem Buzzer elektrostatisch (durch ein elektrisches Feld) oder durch den Piezzo-Effekt (Kristalle, welche sich unter elektrischer Spannung verformen). Dadurch ermöglichen Buzzer eine deutlich kleinere Bauform, jedoch verzerren sie viele Frequenzbereiche und sind daher nicht für Musikwiedergabe geeignet.

Es gibt zwei unterschiedliche Arten von Buzzern:

- Passive Buzzer müssen mit einer modulierten Spannung in einer hörbaren Frequenz angesteuert werden, um ein akustisches Signal abzugeben. Es eignet sich beispielsweise die Arduino-Pulsweitenmodulation bei einem Tastverhältnis von 50% (Ausgabewert 128).
- Aktive Buzzer besitzen hingegen eine integrierte Elektronik, welche selbst eine bestimmte Frequenz erzeugt. Sie werden daher einfach mit konstanter Betriebsspannung angesteuert.

Abb. 11.17 Aktive und passive Buzzer sind kaum zu unterscheiden. In diesem Fall ist der Summer mit der grünen Platine an der Rückseite der passive.

Als Beispiel wollen wir uns mit dem bereits bekannten Fotowiderstand eine kleine Alarmanlage bauen. Dabei messen wir alle 500 Millisekunden den Spannungswert am Fotowiderstand. Gibt es abrupte Änderungen, wird ein kurzer Alarmton ausgegeben. Als Kriterium dafür ermitteln wir die Differenz zwischen den letzten beiden Messwerten.

Überschreitet sie den Wert 10, führt dies zum Alarm. Zusätzlich werden die Messwerte zur Anschaulichkeit auf dem seriellen Monitor ausgegeben.

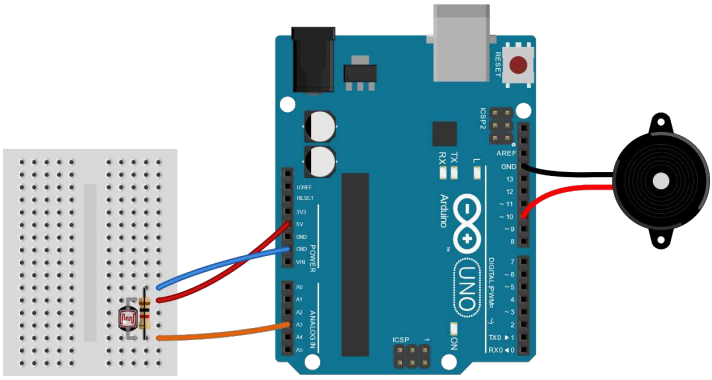


Abb. 11.18 Die Verdrahtung ist übersichtlich; hier wurde für den aktiven Summer Pin 10 gewählt.

Der nachfolgende Sketch bietet sowohl die Möglichkeit, einen aktiven Summer (an Pin 10) als auch einen passiven (an Pin 11) zu verwenden.

```
#define SENSOR 3
#define AVTIVERBUZZER 10     // wird bei Alarm mit 5V beschaltet
#define PASSIVERBUZZER 11    // wird bei Alarm mit PWM beschaltet

int letzterWert = 0;
int aktuellerWert = 0;

void setup() {
  pinMode(AVTIVERBUZZER, OUTPUT);
  pinMode(PASSIVERBUZZER, OUTPUT);
  digitalWrite(AVTIVERBUZZER, LOW); // Bei Programmstart sollen
  analogWrite(PASSIVERBUZZER, 0);   // die Summer ausgeschaltet
                                    // sein.
  Serial.begin(9600);
  letzterWert = analogRead(SENSOR);
}

void loop() {
  aktuellerWert = analogRead(SENSOR);

```

```
21    Serial.print("aktueller Wert: ");
22    Serial.println(aktuellerWert);
23
24    if(abs(aktuellerWert - letzterWert) > 10)                // (1)
25    {
26      Serial.println("ALARM");
27      digitalWrite(AVTIVERBUZZER, HIGH);      // Im Alarmfall wird
28                                              // ein
29      analogWrite(PASSIVERBUZZER, 128);       // Piepton ausgegeben
30      delay(500);                             // und nach 500
31                                              // Millisekunden
32      digitalWrite(AVTIVERBUZZER, LOW);       // wieder
33                                              // deaktiviert.
34      analogWrite(PASSIVERBUZZER, 0);
35      aktuellerWert = analogRead(SENSOR);
36    }
37    letzterWert = aktuellerWert;
38    delay(500);
39  }
```

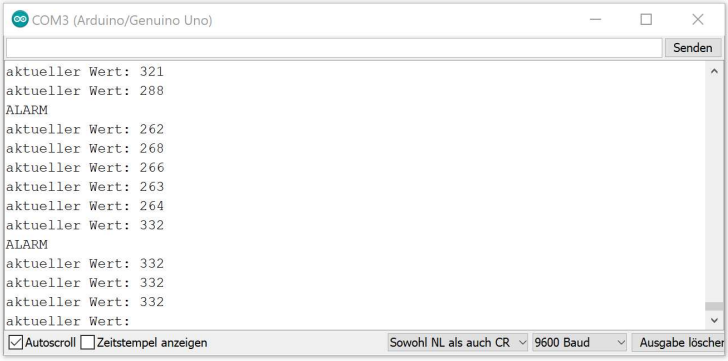


Abb. 11.19 Unterscheiden sich die letzten beiden Werte um mehr als 10, wird der Alarm ausgelöst

1. Das einzig Neue für Sie ist in diesem Sketch die Funktion `abs()`. Sie liefert den Absolutwert einer Zahl, entfernt also ein eventuelles negatives Vorzeichen. Schließlich könnte das Ergebnis von `(aktuellerWert - letzterWert)` beispielsweise auch -34 sein. Trotz des großen Betrages würde dann kein Alarm ausgelöst, denn -34 < 10.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 12

Praxisprojekt: fernsteuerbares Auto

12.1 Idee

Mit unseren bisherigen Kenntnissen sollte es nun schon möglich sein, ein fernsteuerbares Auto zu bauen. Im Internet sind Bausätze erhältlich, welche bereits eine Kunststoff-Grundplatte sowie mehrere Getriebemotoren und Räder bieten. Je nach Umfang enthalten sie auch noch weitere Elemente und einen Aufbauvorschlag. Im folgenden Projekt wollen wir das Modell selbst gestalten und greifen dafür nur auf grundlegende Komponenten aus einem Bausatz zurück. Es sind natürlich auch beliebige Abwandlungen denkbar, wenn Sie stattdessen andere Teile verwenden möchten.

Abb. 12.1 Der RoboCar-Bausatz aus dem Internet lässt viel Spielraum für Kreativität. Wir benötigen nur die Grundplatte und zwei Motoren mit zugehörigen Rädern.

Unser kleines Mobil soll per Infrarotfernbedienung steuerbar sein, außerdem soll es drohende Kollisionen in Vorausrichtung per Ultraschall-Abstandsmesser selbständig erkennen und verhindern.

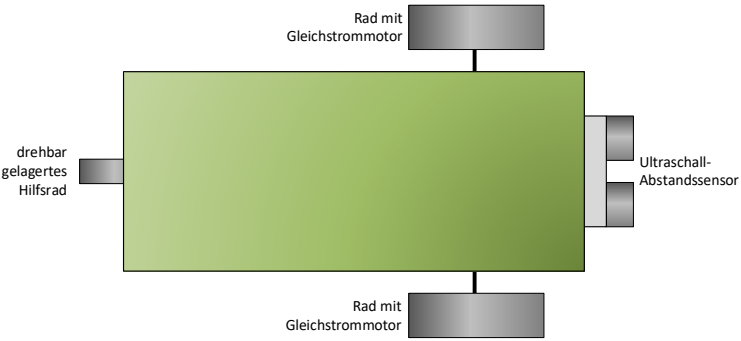


Abb. 12.2 Skizze des geplanten Models

Als Fernbedienung wollen wir eine handelsübliche TV-Fernbedienung verwenden. Die Zahlentasten bieten dabei eine gute Möglichkeit für leicht einprägsame Steuerbefehle:

1: ↰ Linkskurve vorwärts	2: ↑ vorwärts geradeaus	3: ↱ Rechtskurve vorwärts
4: ↺ linksherum drehen	5: ■ stehenbleiben	6: ↻ rechtsherum drehen
7: ↲ Linkskurve rückwärts	8: ↓ rückwärts	9: ↳ Rechtskurve rückwärts

Tabelle 12.1 geplantes Steuerungsschema

Um die geplante Funktionalität zu realisieren, benötigen wir also zwei Gleichstrommotoren mit L293D-Treiber sowie einen IR-Empfänger und einen Ultraschall-Sensor.

12.2 Stromlaufplan

Der Stromlaufplan wird dadurch etwas umfangreicher:

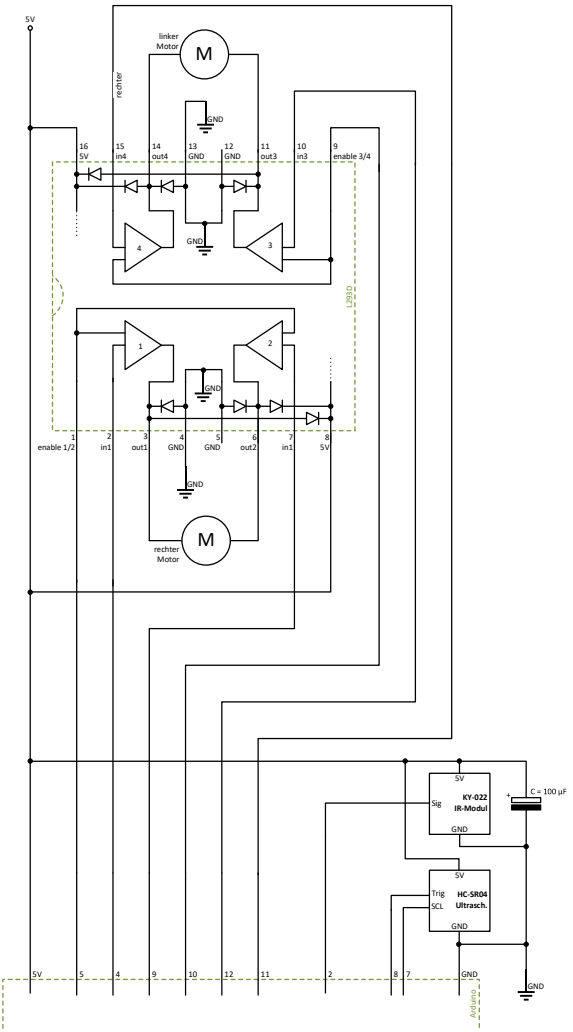


Abb. 12.3 Da für jeden Motor drei Pins des Arduino benötigt werden, ist der Stromlaufplan etwas umfassender. Dennoch sind alle Verbindungen bereits aus vorangegangenen Kapiteln bekannt. Erstmals dient hier der Arduino nicht selbst (via USB) als Spannungsquelle, sondern erhält die Betriebsspannung von außen.

Parallel zur Betriebsspannung wurde ein Kondensator vorgesehen. Er soll kurzfristige Spannungsschwankungen „abfedern“. Dies ist notwendig, da wir beide Motoren per Pulsweitenmodulation ansteuern werden und somit eine relativ große Last pulsierend an- und abschalten. Die Batterie als Spannungsquelle mit relativ hohem Innenwiderstand (siehe Kapitel 3.1.2.1) kann dies nicht ausgleichen, somit würde auf der Betriebsspannung ein pulsierendes Signal messbar sein. Dies stört jedoch sowohl den Arduino-Mikrocontroller als auch das Infrarot-Empfangsmodul und führt zu unerwünschten Ergebnissen. Der Kondensator gleicht diese Schwankungen wie ein Stoßdämpfer beim Auto aus (siehe Kapitel 3.1.2.2).

12.3 Versuchsaufbau

Als Träger können wir uns aus der Kunststoffplatte und zwei Motoren des RoboCar-Sets ein Grundgerüst bauen. Beide Motoren sind einzeln ansteuerbar, somit kann die Lenkung direkt durch unterschiedliche Drehzahlen der Antriebsmotoren realisiert werden und ein zusätzliches lenkbares Rad ist nicht notwendig. Ein frei drehbar gelagertes Hilfsrad soll lediglich das Umkippen verhindern.

Abb. 12.4 Das RoboCar-Set dient als Grundgerüst für unser Auto

Da es nun auf Mobilität ankommt, müssen wir zusätzlich eine Batterie mitführen. Um Platz zu sparen, werden wir ein Spannungsversorgungs-Modul des Herstellers YwRobot verwenden. Es bie-

tet die Möglichkeit, eine 9-Volt-Blockbatterie anzuschließen und regelt die Spannung auf 5 Volt herunter. Eine weitere Platzeinsparung können wir dadurch erreichen, statt des Arduino UNO einen Arduino Nano zu verwenden. Er bietet die gleiche Funktionalität auf einer wesentlich kleineren Platine, welche direkt auf das Breadboard gesteckt werden kann. Die Pin-Nummern stimmen mit denen seines großen Bruders überein. Allerdings müssen wir daran denken, in der Arduino IDE *Werkzeuge -> Board -> Arduino Nano* auszuwählen.

Wir benötigen demnach für das Projekt:

- 1 RoboCar-Grundgerüst mit 2 Getriebemotoren und Rädern
- 1 großes Breadboard
- 1 kleines Breadboard
- 1 Arduino Nano
- 1 L293D-Motortreiber-IC
- 1 Spannungsversorgungs-Modul mit 9V-Batterie
- 1 Ultraschall-Sensor HC-SR04
- 1 Infrarot-Empfänger KY-022
- 1 Infrarot-Fernbedienung (hier beispielhaft „Vivanco UR 2“)
- 1 Elektrolyt-Kondensator (100 µF)
- 2 Taster
- Verbindungsdrähte

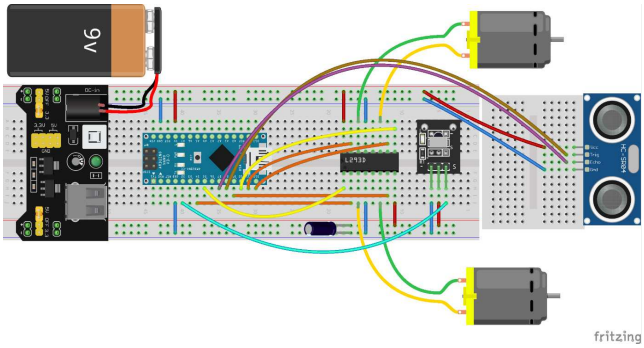


Abb. 12.5 Die Verdrahtung ist recht umfangreich; zur Fehlervermeidung sollte man sich dabei am Stromlaufplan orientieren.

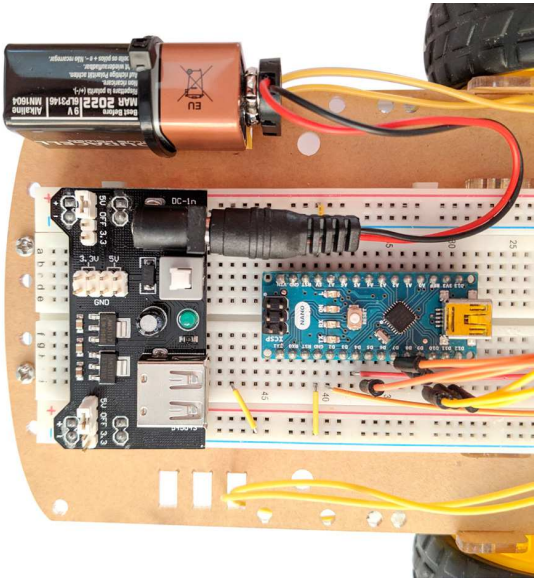


Abb. 12.6 Das Spannungsversorgungs-Modul kann auf das Breadboard gesteckt werden und versorgt dessen obere und untere Querleisten direkt mit der Betriebsspannung. Wichtig ist, dass die beiden weißen Jumper zur Spannungswahl auf dem Modul auf "5V" stehen.

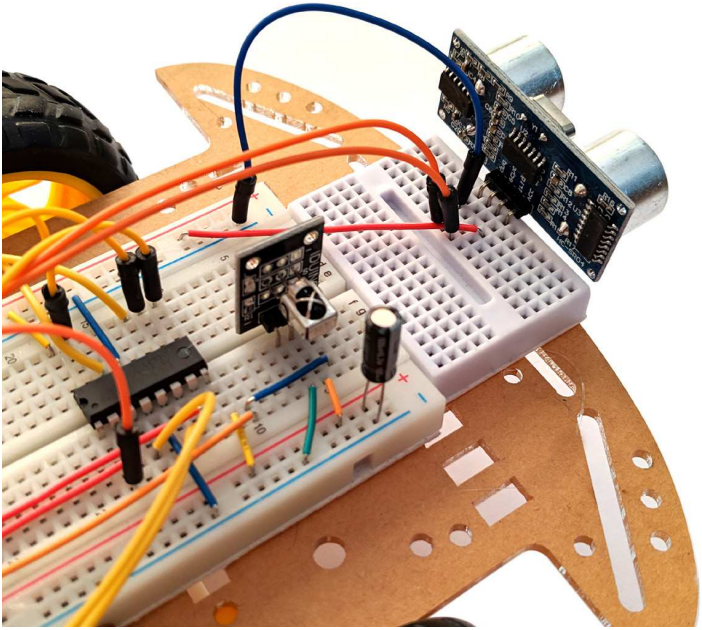


Abb. 12.7 Der Ultraschall-Sensor muss natürlich in Vorausrichtung angebracht werden. Das große Breadboard bietet jedoch keine Steckmöglichkeit in diese Richtung. Daher nehmen wir ein kleines Breadboard zur Hilfe.

Im Beispielaufbau wurden beide Breakboards mit Heißkleber auf der Grundplatte befestigt. Verwendet man dabei nur geringe Mengen Kleber, kann man ihn bei Bedarf rückstandslos entfernen.

Als Fernbedienung kommt hier beispielhaft eine „Vivanco UR 2"-Universalfernbedienung zum Einsatz.

Abb. 12.8 Es eignet sich nahezu jede beliebige TV-Fernbedienung. Ist eine Neuanschaffung nötig, sind Universalfernbedienungen oft die preisgünstigste Variante.

12.4 Programmcode

Bevor wir den Programmcode entwickeln, müssen wir zunächst herausfinden, welche Signale unsere Fernbedienung sendet. Im Beispiel verwenden wir die genannte Universalfernbedienung mit dem willkürlich gewählten Gerätecode „1-2-3“.

Üblicherweise senden Infrarotfernbedienungen für jeden Tastendruck ein bestimmtes Bitmuster, welches wir als Zahl interpretieren können. Oft werden bestimmte Bits davon noch mit einer weiteren Bedeutung

belegt, zum Beispiel um einen wiederholten Tastendruck von einer gedrückt gehaltenen Taste zu unterscheiden.

Um dies näher zu untersuchen, laden wir den Sketch aus Kapitel 9.2 auf den Arduino und testen unsere Fernbedienung:

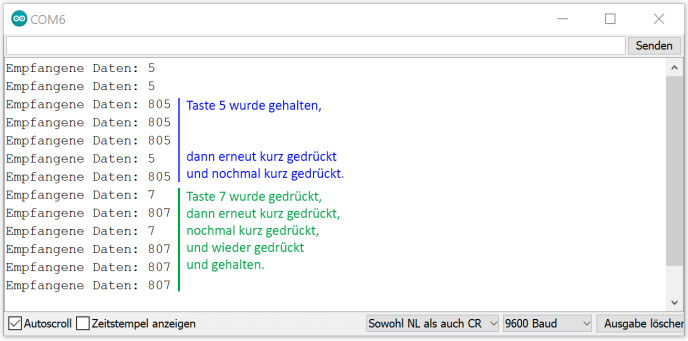


Abb. 12.9 Ergebnis eines Fernbedienungstests mit den Tasten 5 und 7

Im abgebildeten Beispiel wurden die Tasten 5 und 7 getestet, und zwar sowohl bei wiederholtem Drücken als auch bei Halten der Taste. An der (hexadezimalen) Bildschirmausgabe ist erkennbar: Offenbar können die niedrigwertigsten Bits direkt als Zahl interpretiert werden, welche der gedrückten Ziffer entspricht. Zusätzlich werden in manchen Fällen noch weitere Bits übertragen, welche den Hexadezimalwert 0x800 bilden. Beim manuellen Versuch wird deutlich: Lässt man eine Taste los und drückt sie erneut, wird abwechselnd der Wert 0x800 weggelassen oder addiert. Hält man eine Taste gedrückt, passiert dies nicht. Unterscheidet sich also ein empfangenes Signal vom zuletzt empfangenen Wert um genau diese 0x800, handelt es sich um einen neuen Tastendruck. Gibt es diesen Unterschied nicht, wurde die Taste gedrückt gehalten. Dies wollen wir dann nicht als neuen Befehl werten.

Je nach verwendeter Fernbedienung unterscheiden sich diese Empfangssignale. Wichtig ist, dass man eine Möglichkeit findet, einer Taste einen bestimmten Empfangswert zuzuordnen und dass man das Festhalten von Tasten von einem erneuten Tastendruck unterscheiden kann.

Da wir diese Unterscheidung nun beherrschen, können wir sogar verschiedene Leistungsstufen festlegen. So soll unser Auto bei einem Druck auf Taste 2 zunächst mit geringer Geschwindigkeit losfahren und erst beim zweiten Druck auf die Taste in die höchste Leistungsstufe schalten.

Alle diese Vorbetrachtungen fließen nun mit in den Programmcode ein:

```
#define MR_A    4    // Pins für die Motorsteuerung
#define MR_B    9    // über den L293D-Motortreiber
#define MR_PWM  5    // MR – rechter Motor
#define ML_A   11    // ML – linker Motor
#define ML_B   12    // A/B – Eingänge der Treiber
#define ML_PWM 10    // PWM-Pins zur Leistungssteuerung
                     // (enable des Treibers)

#define IR_PIN  2    // Pin des IR-Empfängers

#define ECHOPIN 7    // Pins des Ultraschallmoduls
#define TRIGPIN 8

#define ML 0         // symbolisiert linken Motor
#define MR 1         // symbolisiert rechten Motor
#define VOR 0        // symbolisiert Vorwärtslauf
#define RUECK 1      // symbolisiert Rückwärtslauf
#define STUFE1 128   // Leistungsstufen 1 und 2
#define STUFE2 255

#include "IRremote.h"
#include "HCSR04.h"

IRrecv Empfaenger(IR_PIN);
decode_results Daten;
UltraSonicDistanceSensor Abstandssensor(TRIGPIN, ECHOPIN);

float Abstand;                       // Ergebnis der
                                     // Ultraschall-Messung
int letzterInfrarotbefehl = 0;
int Eingabebefehl = 0;               // Benutzereingabe
byte aktuelleBetriebsart = 5;        // Betriebsart (aktueller
                                     // Fahrmodus)

byte aktuelleLeistungR = 0;
byte ZielleistungR = 0;
byte aktuelleLeistungL = 0;
```

```
byte ZielleistungL = 0;
long AusweichTimer = 0;

void setup()
{
  pinMode(MR_A, OUTPUT); pinMode(MR_B, OUTPUT);
  pinMode(MR_PWM, OUTPUT);
  pinMode(ML_A, OUTPUT); pinMode(ML_B, OUTPUT);
  pinMode(ML_PWM, OUTPUT);
  Empfaenger.enableIRIn();
}

void loop()
{

  if (Empfaenger.decode(&Daten))              // Wenn IR-Signal
                                              // empfangen wurde:
  {
     if(Daten.value != letzterInfrarotbefehl) // Nur wenn
                                              // empfangenes
                                              // Signal
     {                                        // vom vorherigen
                                              // abweicht
                                              // (denn sonst
                                              // handelt es sich
                                              // nicht um neuen
                                              // Tastendruck):
       Eingabebefehl = Daten.value % 0x800;   // Eingabe auslesen
                                              // und
                                              // niedrigwertigste
                                              // Bits extrahieren
       letzterInfrarotbefehl = Daten.value;
     }
     Empfaenger.resume();
  }

  switch(Eingabebefehl)                       // Eingabe
                                              // verarbeiten
  {
    case 1:                                                  // (1)
      aktuelleLeistungR = aktuelleLeistungL = 255;
      ZielleistungR = STUFE2;
      ZielleistungL = STUFE1;
      aktuelleBetriebsart = 1;
    break;

    case 2:                                                  // (2)
      if(aktuelleBetriebsart == 2)
```

```
86          ZielleistungR = ZielleistungL = STUFE2;
87        else
88        {
89          aktuelleLeistungR = aktuelleLeistungL = 255;
90          ZielleistungR = ZielleistungL = STUFE1;
91        }
92        aktuelleBetriebsart = 2;
93      break;
94
95      case 3:
96        aktuelleLeistungR = aktuelleLeistungL = 255;
97        ZielleistungR = STUFE1;
98        ZielleistungL = STUFE2;
99        aktuelleBetriebsart = 3;
100     break;
101
102     case 4:
103       aktuelleLeistungR = aktuelleLeistungL = 255;
104       ZielleistungR = ZielleistungL = STUFE1;
105       aktuelleBetriebsart = 4;
106     break;
107
108     case 5:
109       aktuelleBetriebsart = 5;
110     break;
111
112     case 6:
113       aktuelleLeistungR = aktuelleLeistungL = 255;
114       ZielleistungR = ZielleistungL = STUFE1;
115       aktuelleBetriebsart = 6;
116     break;
117
118     case 7:
119       aktuelleLeistungR = aktuelleLeistungL = 255;
120       ZielleistungR = STUFE2;
121       ZielleistungL = STUFE1;
122       aktuelleBetriebsart = 7;
123     break;
124
125     case 8:
126       if(aktuelleBetriebsart == 8)
127         ZielleistungR = ZielleistungL = STUFE2;
128       else
129       {
130         ZielleistungR = ZielleistungL = STUFE1;
131         aktuelleLeistungR = aktuelleLeistungL = 255;
132       }
133       aktuelleBetriebsart = 8;
```

```
134     break;
135
136     case 9:
137       aktuelleLeistungR = aktuelleLeistungL = 255;
138       ZielleistungR = STUFE1;
139       ZielleistungL = STUFE2;
140       aktuelleBetriebsart = 9;
141     break;
142   }
143
144   Eingabebefehl = 0;                                          // (3)
145
146   switch(aktuelleBetriebsart)                                 // (4)
147   {
148     case 1: case 2: case 3: // Vorausfahrt gerade oder mit Kurve
149       Richtung(MR, VOR);
150       Richtung(ML, VOR);
151       analogWrite(MR_PWM, aktuelleLeistungR);
152       analogWrite(ML_PWM, aktuelleLeistungL);
153
154       Abstand = Abstandssensor.measureDistanceCm();           // (5)
155       if(Abstand < 5.0)
156       {
157         if(random(2))                                         // (6)
158         {
159           aktuelleBetriebsart = 7;
160           ZielleistungR = STUFE2;
161           ZielleistungL = STUFE1;
162         }
163         else
164         {
165           aktuelleBetriebsart = 9;
166           ZielleistungR = STUFE1;
167           ZielleistungL = STUFE2;
168         }
169
170         aktuelleLeistungR = aktuelleLeistungL = 255;
171         AusweichTimer = millis() + 2000;                      // (7)
172       }
173     break;
174     case 4:                 // linkssherum drehen
175       Richtung(MR, VOR);
176       Richtung(ML, RUECK);
177       analogWrite(MR_PWM, aktuelleLeistungR);
178       analogWrite(ML_PWM, aktuelleLeistungL);
179     break;
180     case 5:                           // stillstehen
181       ZielleistungR = ZielleistungL = 0;
```

```
182       analogWrite(MR_PWM, aktuelleLeistungR);
183       analogWrite(ML_PWM, aktuelleLeistungL);
184     break;
185     case 6:                                  // rechtsherum drehen
186       Richtung(MR, RUECK);
187       Richtung(ML, VOR);
188       analogWrite(MR_PWM, aktuelleLeistungR);
189       analogWrite(ML_PWM, aktuelleLeistungL);
190     break;
191     case 7: case 8: case 9: // Rückwärtsfahrt gerade oder mit
192                             // Kurve
193       Richtung(MR, RUECK);
194       Richtung(ML, RUECK);
195       analogWrite(MR_PWM, aktuelleLeistungR);
196       analogWrite(ML_PWM, aktuelleLeistungL);
197     break;
198   }
199
200   if(aktuelleLeistungL < ZielleistungL)     // fließende
201                                             // Angleichung
202     aktuelleLeistungL++;                    // von aktueller
203                                             // Leistung
204   if(aktuelleLeistungL > ZielleistungL)     // und Zielleistung
205                                             // durch
206     aktuelleLeistungL--;                    // Annäherung um
207                                             // einen
208                                             // Schritt pro
209                                             // Durchlauf
210   if(aktuelleLeistungR < ZielleistungR)     // der Hauptschleife
211     aktuelleLeistungR++;
212   if(aktuelleLeistungR > ZielleistungR)
213     aktuelleLeistungR--;
214
215   if(millis() > AusweichTimer && AusweichTimer != 0)       // (8)
216   {
217     aktuelleBetriebsart = 2;
218     ZielleistungR = ZielleistungL = STUFE1;
219     aktuelleLeistungR =  aktuelleLeistungL = 255;
220     AusweichTimer = 0;
221   }
222
223   delay(1);
224 }
225
226 void Richtung(byte Motor, byte Richtung)                   // (9)
227 {
228   byte Pin_a = (Motor == ML) ? ML_A : MR_A;               // (10)
229   byte Pin_b = (Motor == ML) ? ML_B : MR_B;
```

```
230
231   if(Richtung == VOR)
232   {
233     digitalWrite(Pin_a, HIGH);
234     digitalWrite(Pin_b, LOW);
235   }
236   else
237   {
238     digitalWrite(Pin_a, LOW);
239     digitalWrite(Pin_b, HIGH);
240   }
241 }
```

1. In dieser Switch-Anweisung wird für jeden Tastenbefehl ein bestimmtes Verhalten abgearbeitet. Dabei wird für jeden der beiden Motoren eine `Zielleistung` (Sollleistung) festgelegt. Im Falle des Tastenbefehls [1] muss sich der rechte Motor mit höherer Leistung drehen als der linke, um die Kurvenfahrt zu realisieren. Zusätzlich haben wir für jeden Motor eine Variable namens `aktuelleLeistung` vorgesehen. Dahinter steckt der Gedanke, dass wir sowohl die Möglichkeit haben wollen, die aktuelle Leistung des Motors abrupt zu ändern (beispielsweise, um vor einem Hindernis zu stoppen) als auch ein sanftes Anfahren zu ermöglichen, indem wir lediglich eine `Zielleistung` vorgeben und die `aktuelleLeistung` in den folgenden Millisekunden langsam daran angeglichen wird.

   An dieser konkreten Stelle wird allerdings sogar ein abruptes Anfahren provoziert, da der linke Motor sonst aufgrund der verminderten Leistung eventuell nicht die Haftreibung aus dem Stand überwindet und somit komplett stehenbliebe.

2. Im Voraus-Modus wird zusätzlich geprüft, ob vor der Benutzereingabe auch schon dieser Modus bestand. Wenn ja, wird in die höchste Geschwindigkeit geschaltet, wenn nein, wird zunächst die geringe Geschwindigkeit gewählt. Zur Überwindung der Haftreibung erfolgt dann allerdings auch, wie bei (1) beschrieben, ein kurzer Impuls mit voller Leistung, die `Zielleistung` liegt jedoch darunter. Die weiteren Zahleneingaben werden auf ähnliche Weise verarbeitet.

3. Die Hilfsvariable `Eingabebefehl` wird hier zurückgesetzt, um sicherzustellen, dass die vorherige `switch`-Verzweigung tatsächlich nur einmal pro Tastendruck abgearbeitet wird.

4. Im Gegensatz dazu wird diese `switch`-Verzweigung nun bei jedem Durchlauf der Hauptschleife relevant, denn ihre Entscheidungsvariable `aktuelleBetriebsart` wird nicht zurückgesetzt. Sie entspricht meist dem letzten Tastenbefehl, kann aber auch vom Programm selbst geändert werden – beispielsweise, um Kollisionen zu vermeiden.

5. Während einer Vorausfahrt wird regelmäßig der Abstand gemessen. Unterschreitet er 5 cm, so wird

6. zufällig (`random(2)` liefert mit je 50% Wahrscheinlichkeit eine `1` oder eine `0` beziehungsweise `true` oder `false`) entweder nach hinten links [7] oder hinten rechts [9] zurückgefahren.

7. Zusätzlich wird eine Variable als Zeitmesser (für `2` Sekunden) gesetzt,

8. welche außerhalb dieser `switch`-Verzweigung geprüft wird. Ist die Zeit davon abgelaufen, wird diese Bedingung `true` und es findet wieder eine Umschaltung in eine langsame Vorausfahrt statt.

9. Diese selbstdefinierte Funktion soll lediglich die Richtungsumschaltung erleichtern, indem sie die Pins für die Motortreiber jeweils für den Vor- oder Rückwärtslauf konfiguriert.

10. Diese Notation einer Variablenzuweisung kennen Sie noch nicht. Sie verkürzt lediglich eine `if`-Verzweigung. Ist die Bedingung vor dem Fragezeichen `true`, wird der Variablen `Pin_a` der Wert `ML_A` zugewiesen, sonst `MR_A`. Man könnte für diese Zeile also auch schreiben:

```
if(Motor == ML)
  Pin_a = ML_A;
else
  Pin_b = MA_B;
```

Gäbe es innerhalb der `if`-Verzweigung noch weitere Befehle, wäre die Verkürzung nicht möglich.

12.5 Resultat

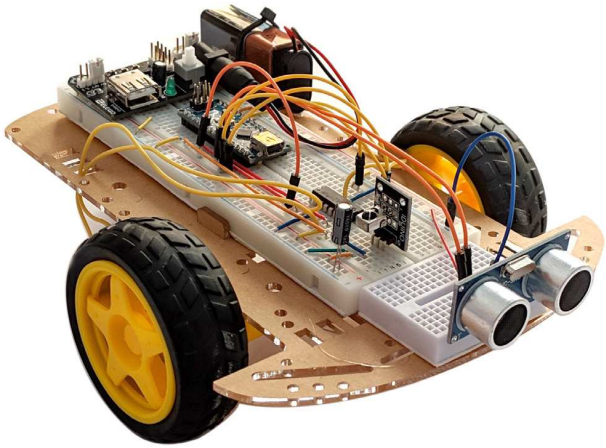


Abb. 12.10 das fertige Mini-Auto

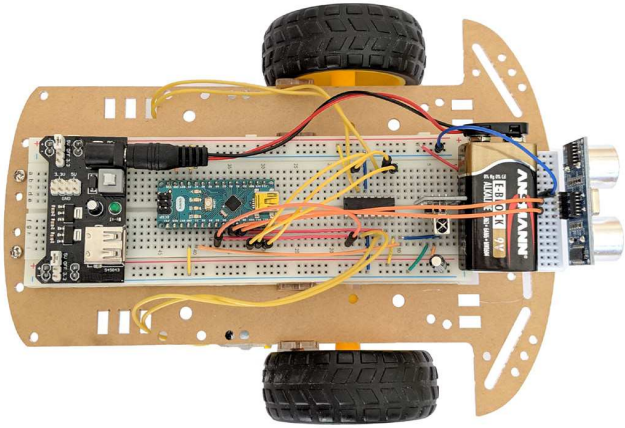


Abb. 12.11 Befestigt man die Batterie weiter vorn, wirkt sich das günstig auf die Gewichtsverteilung aus

Beim Test stellt sich das kleine Mobil als durchaus wendig heraus. Es gibt zahlreiche Stellschrauben, so kann man durch Variieren des Wertes für die `STUFE1` und der Timer-Dauer für das Wendemanöver das Verhalten beeinflussen. Außerdem fällt auf, dass auch im Modus [2] und [8] der Fahrweg nicht ganz gerade ist. Grund sind Fertigungstoleranzen der Gleichstrommotoren. Man könnte dies durch die Nutzung von Schrittmotoren beheben. Alternativ ist es auch möglich, über einen Sensor die tatsächliche Drehbewegung der einzelnen Motoren zu erfassen und die jeweilige Leistung entsprechend dynamisch anzupassen.

Es wäre auch denkbar, an die Frontseite mehrere Fotowiderstände zu montieren und das Mobil automatisch dorthin fahren zu lassen, wo das hellste Licht gemessen wird. Nutzen Sie das RoboCar gern für eigene Experimente, die Möglichkeiten sind fast unbegrenzt!

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 13
Datenverarbeitung

Unsere bisherigen Beispiele zielten stets darauf ab, erfasste Daten sofort zu verarbeiten und gegebenenfalls gleich wieder auszugeben. In manchen Fällen kann es aber notwendig sein, Informationen permanent zu speichern – beispielsweise, wenn es um Nutzereinstellungen geht oder Sensormesswerte über einen längeren Zeitraum erfasst werden sollen. Zudem kann es nötig sein, die erfassten Daten von einem Gerät mit mehr Rechenleistung (zum Beispiel einem PC) auswerten zu lassen, da Geschwindigkeit und Speicherplatz des Arduino limitiert sind.

13.1 Permanente Speicher

In der Anfangszeit der Computertechnik war das dauerhafte Speichern von Daten eine Herausforderung. Zwar hatte man bereits Schaltungen für das Zwischenspeichern von Binärdaten entwickelt, jedoch waren diese auf ständige Spannungszufuhr angewiesen. Ohne angeschlossene Energieversorgung gingen die Daten verloren. Das gleiche Verhalten zeigt auch der Arbeitsspeicher unseres Arduino.

Parallel zur Entwicklung der ersten Festplatten, welche das permanente Speichern von größeren Datenmengen auf rotierenden Magnetscheiben ermöglichten, entstanden EPROMs (*Erasable Programmable Read-Only-Memory* – löschbarer programmierbarer Nur-Lese-Speicher). Diese Bausteine hatten zwar ein geringeres Speichervolumen als Festplatten, konnten dafür aber platzsparend direkt auf der Platine des Rechners verbaut werden. Auch ohne Spannungsversorgung blieben die Daten erhalten. Der Nachteil: Die Module waren zwar elektronisch beschreibbar, jedoch mussten sie zum Löschen umständlich von außen mit UV-Licht bestrahlt werden.

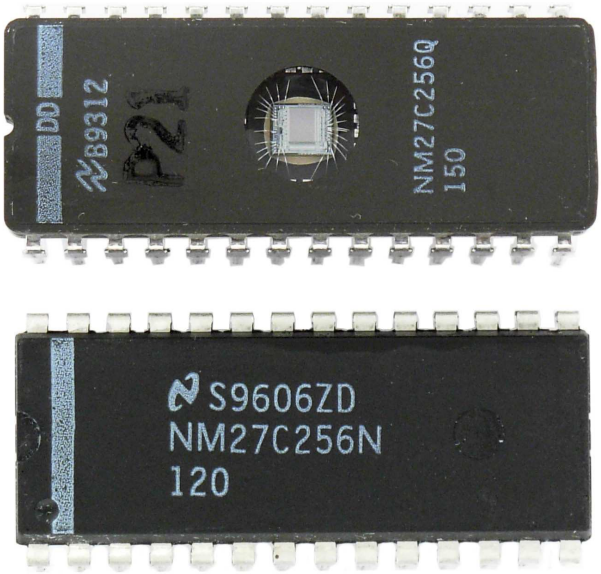


Abb. 13.1 EPROM-Baustein; das Sichtfenster dient der Bestrahlung mit UV-Licht von außen während des Löschvorgangs

Später entwickelten sich daraus die EEPROMs (*Electrically Erasable Programmable Read-Only-Memory*), nun konnte also endlich auch die Löschung von Daten elektronisch erfolgen. EEPROMs sind mittlerweile problemlos in andere Chips integrierbar, daher werden sie häufig auch in Mikrocontroller eingebaut und dort zur Speicherung von Einstellungen verwendet.

13.1.1 Interner EEPROM

Der Arduino-Mikrocontroller bietet einen internen EEPROM mit einer Größe von 1024 Byte. Im folgenden Beispiel wollen wir ihn nutzen, um die vom Bediener eingestellte Helligkeit einer dimmbaren LED permanent (also auch bei Spannungsverlust) zu speichern.

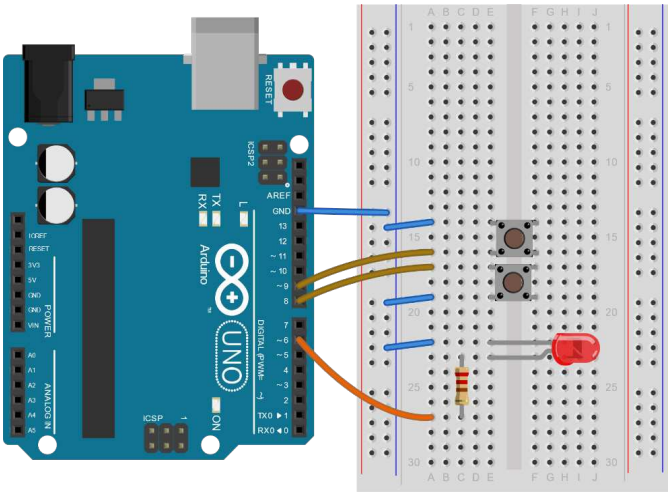


Abb. 13.2 Die Verdrahtung ist einfach, drei digitale Pins sind mit zwei Tastern und einer LED verbunden. Der Vorwiderstand der LED beträgt, wie in allen Beispielen dieses Buches, 330 Ohm.

Dazu programmieren wir eine Steuerung, welche sich über zwei Taster bedienen lässt. Ein Taster erhöht die Helligkeit, der andere verringert sie. Wird die Versorgungsspannung getrennt und später wieder verbunden, soll die zuletzt eingestellte Helligkeit wieder aufgenommen werden.

Zur Nutzung des internen EEPROMs benötigen wir die Bibliothek EEPROM.h, sie ist bereits vorinstalliert.

```
#define KNOPF1 9
#define KNOPF2 8
#define LED 6

#include "EEPROM.h"

byte Helligkeit;
long Timer;

void setup() {
  pinMode(KNOPF1,INPUT_PULLUP);
```

```
12    pinMode(KNOPF2,INPUT_PULLUP);
13    pinMode(LED,OUTPUT);
14
15    Helligkeit = EEPROM.read(0);                              // (1)
16    analogWrite(LED, Helligkeit);
17  }
18
19  void loop() {
20    if(!digitalRead(KNOPF1))
21    {
22      if(Helligkeit < 255)
23        Helligkeit++;
24      analogWrite(LED, Helligkeit);
25
26      Timer = millis() + 1000;
27      while(!digitalRead(KNOPF1))                             // (2)
28      {
29        if(millis() > Timer)
30        {
31          if(Helligkeit < 255)
32            Helligkeit++;
33          analogWrite(LED, Helligkeit);
34          delay(10);
35        }
36      }
37      EEPROM.write(0,Helligkeit);                             // (3)
38    }
39
40    if(!digitalRead(KNOPF2))                                  // (4)
41    {
42      if(Helligkeit > 0)
43        Helligkeit--;
44      analogWrite(LED, Helligkeit);
45
46      Timer = millis() + 1000;
47      while(!digitalRead(KNOPF2))
48      {
49        if(millis() > Timer)
50        {
51          if(Helligkeit > 0)
52            Helligkeit--;
53          analogWrite(LED, Helligkeit);
54          delay(10);
55        }
56      }
57      EEPROM.write(0,Helligkeit);
58    }
59  }
```

1. Die Bibliothek erstellt automatisch das Objekt `EEPROM`. Mittels seiner Funktion `read()` können wir ein bestimmtes Byte aus dem EEPROM auslesen, das Argument ist die gewünschte Speicheradresse. Insgesamt gibt es 1024 Bytes, demnach muss das Argument im Bereich von 0 bis 1023 liegen. Die Rückgabewert ist immer vom Typ `byte` (beziehungsweise `char`).

2. Hier gehen wir bei der Abfrage der Eingabe noch einen Schritt weiter als in früheren Beispielen. Während wir bisher an dieser Stelle immer eine `while`-Schleife mit leerem Rumpf nutzten, um auf das Loslassen der Taste zu warten, wollen wir dem Nutzer nun ermöglichen, durch längeres Drücken den Wert schneller zu verändern. Deshalb setzen wir kurz vor der Schleife eine `Timer`-Variable auf einen Wert, der eine Sekunde in der Zukunft liegt. Wird dieser Zeitpunkt erreicht, während die Schleife noch läuft, wird die Variable `Helligkeit` kontinuierlich bis zum Maximum (255) erhöht.

   Der Nutzer kann nun also nach Belieben durch einen einzelnen Tastendruck die Helligkeit um einen einzelnen Schritt erhöhen oder durch Halten der Taste den Wert schnell verändern.

3. Nachdem der Helligkeitswert verändert wurde, wird er in den EEPROM geschrieben, um beim nächsten Programmstart abrufbar zu sein. Die dafür vorgesehene Funktion `write()` erwartet als Argumente die gewünschte Speicheradresse (0 ... 1023) und den zu speichernden Wert (als 8-Bit-Variable, also `byte` oder `char`).

4. Für die Verringerung der Helligkeit wurde der gerade diskutierte Programmteil dupliziert und entsprechend angepasst.

   Eine gewisse Limitierung ergibt sich dadurch, dass die Funktion `write()` nur 8-Bit-Variablen verarbeiten kann. Sollen größere Daten (beispielsweise `int` oder Zeichenketten) gespeichert werden, muss man sie entsprechend auf mehrere 8-Bit-Werte aufteilen. So könnte man eine 16-Bit-Variable wie folgt teilen und später wieder zusammenfügen:

```
1 ...
2 int a = 12345;
3 EEPROM.write(0, a / 256);                    // ergibt die 8
4                                              // höchstwertigen Bit
```

```
5  EEPROM.write(1, a % 256);                    // ergibt die 8
6                                               // niedrigstwertigen
7                                               // Bit
8  ...
9  a = EEPROM.read(0)*256 + EEPROM.read(1);     // auslesen und
10                                              // zusammenfügen
11 ...
```

13.1.2 Externer EEPROM

Natürlich ist es auch möglich, externe EEPROM-Module zu nutzen, wenn der interne Speicher nicht ausreicht. Als Beispiel eignet sich das DS3231-Echtzeituhrmodul, welches Sie bereits in Kapitel 9.11 kennengelernt haben. Neben dem DS3231-Zeitmesser enthält es auch noch einen AT24C32-Chip, welcher 4096 Byte EEPROM-Speicher zur Verfügung stellt. Beide teilen sich die Spannungsversorgung und den I²C-Bus, sind ansonsten aber völlig unabhängig voneinander. Der Verbund rührt daher, dass dieses Modul ursprünglich für die Anwendung als Datenlogger entwickelt wurde.

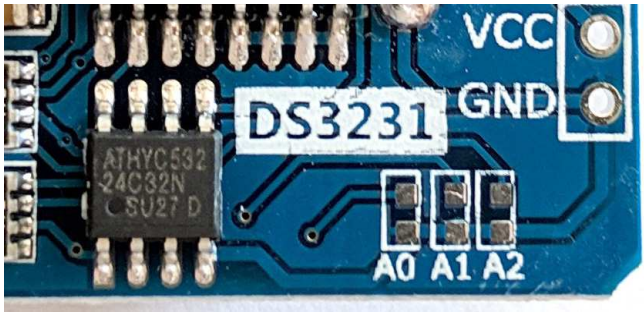


Abb. 13.3 Dieses Foto ist eine Ausschnittvergrößerung des Echtzeitmoduls von Abbildung 9.34. Der 24C32-Speicherchip ist deutlich kleiner als der benachbarte Zeitmesser. Erkennbar sind auch die Lötpads zur Anpassung der I²C-Adresse.

Durch die verschiedenen I²C-Adressen lassen sich beide Chips getrennt ansteuern. Zusätzlich bietet der 24C32-EEPROM-Chip die Möglichkeit, die Busadresse durch das Überbrücken von Lötpads zu verändern. Man geht dazu genau wie beim LCD-Backpack aus Kapitel 7.5 vor. Somit lassen sich bis zu acht derartige Speicher an einem I²C-Bus betreiben.

I²C-Adresse	A0	A1	A2
80	LOW	LOW	LOW
81	HIGH	LOW	LOW
82	LOW	HIGH	LOW
83	HIGH	HIGH	LOW
84	LOW	LOW	HIGH
85	HIGH	LOW	HIGH
86	LOW	HIGH	HIGH
87 (Standard)	HIGH	HIGH	HIGH

Tabelle 13.1 Durch das Überbrücken der Lötpads kann man am entsprechenden Eingang des Chips einen LOW-Pegel erzeugen. Ohne Brücke wird der Pegel per Pull-Up-Widerstand auf HIGH gezogen. Daher ist die Standardadresse bei unveränderten Lötpads 87.

Wir wollen nun unser vorangegangenes Beispiel noch einmal mit dem auf dem Echtzeituhrmodul enthaltenen AT24C32 nachstellen.

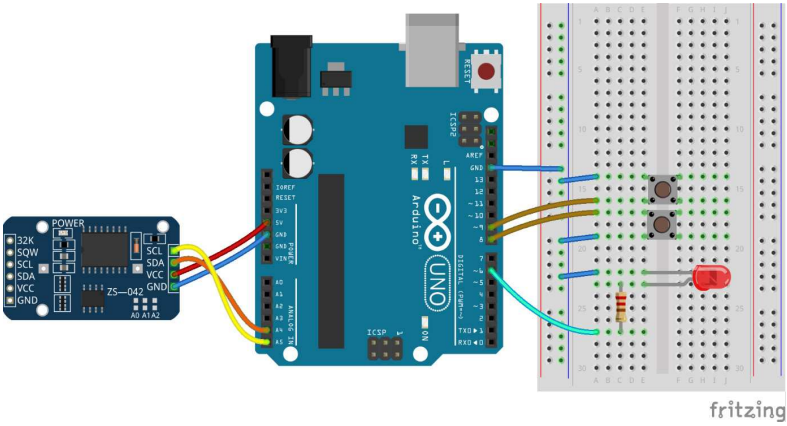


Abb. 13.4 Der Versuchsaufbau wird um das Echtzeitmodul erweitert, um dessen EEPROM-Chip zu nutzen

Zur Ansteuerung installieren wir die Bibliothek uEEPROMlib.h:

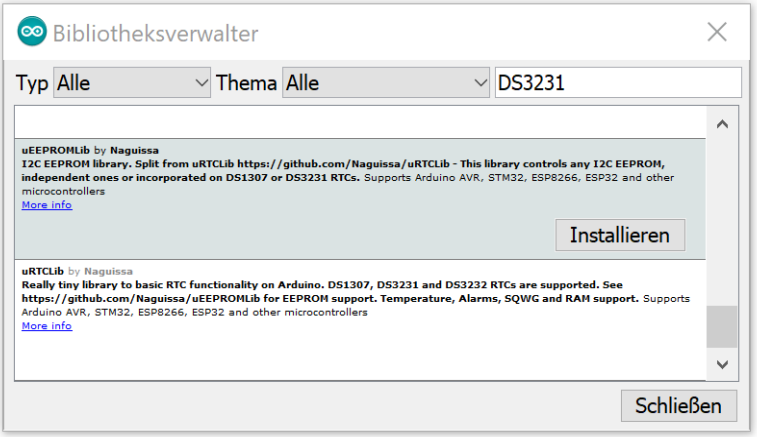


Abb. 13.5 Die Bibliothek uEEPROMLib ist wie üblich über den Bibliotheksverwalter installierbar

Der Sketch entspricht im Wesentlichen dem aus dem vorangegangenen Beispiel, daher sind nachfolgend nur die Unterschiede kommentiert.

```
#define KNOPF1 9
#define KNOPF2 8
#define LED 6

#include "uEEPROMLib.h"             // Bibliothek für den AT24C32

uEEPROMLib externerSpeicher(87);                              // (1)

byte Helligkeit;
long Timer;

void setup() {
  pinMode(KNOPF1,INPUT_PULLUP);
  pinMode(KNOPF2,INPUT_PULLUP);
  pinMode(LED,OUTPUT);

  Wire.begin();                                               // (2)
  Helligkeit = externerSpeicher.eeprom_read(0);               // (3)

  analogWrite(LED, Helligkeit);
```

```
}

void loop() {
  if(!digitalRead(KNOPF1))
  {
    if(Helligkeit < 255)
      Helligkeit++;
    analogWrite(LED, Helligkeit);

    Timer = millis() + 1000;
    while(!digitalRead(KNOPF1))
    {
      if(millis() > Timer)
      {
        if(Helligkeit < 255)
          Helligkeit++;
        analogWrite(LED, Helligkeit);
        delay(10);
      }
    }
    externerSpeicher.eeprom_write(0,Helligkeit);           // (4)
  }

  if(!digitalRead(KNOPF2))
  {
    if(Helligkeit > 0)
      Helligkeit--;
    analogWrite(LED, Helligkeit);

    Timer = millis() + 1000;
    while(!digitalRead(KNOPF2))
    {
      if(millis() > Timer)
      {
        if(Helligkeit > 0)
          Helligkeit--;
        analogWrite(LED, Helligkeit);
        delay(10);
      }
    }
    externerSpeicher.eeprom_write(0,Helligkeit);
  }
}
```

1. Das Objekt `externerSpeicher` als Instanz der Klasse `uEEPROMLib` repräsentiert unser Modul. Hierbei muss als Argument die I²C-Adresse übergeben werden, in unserem Fall 87.
2. Die I²C-Verbindung wird als Master gestartet. Die Wire.h-Bibliothek mussten wir nicht einbinden, da dies bereits innerhalb der uEEPROMLib.h geschieht. Es wäre aber auch kein Fehler gewesen, dies doppelt zu tun, denn der Compiler ignoriert es dann.
3. Das Auslesen erfolgt jetzt lediglich über einen anderen Funktionsnamen, aber in gleicher Weise: Die Funktion `eeprom_read()` unseres Objektes liest ein Byte Daten von der als Argument übergebenen Speicheradresse (bei diesem Chip im Bereich von 0 bis 4095).
4. Für das Schreiben gilt dies ebenso, die Funktion heißt nun `eeprom_write()` und erwartet die Speicheradresse und das zu schreibende Byte.

Auch hier kann jede Speicheradresse nur 8 Bit speichern. Die Bibliothek stellt allerdings Funktionen bereit, mit denen größere Daten einfach aufgeteilt werden können. Bei Interesse finden Sie weitere Informationen, wie bei jeder Bibliothek, in der zugehörigen Dokumentation.

13.1.3 SD-Karte

Möchten Sie noch umfangreichere Daten speichern, beispielsweise Messwerte über einen längeren Zeitraum, so können Sie auf SD-Karten zurückgreifen. Die aus der Computerwelt bekannten Datenträger gibt es mit verschiedenen Speichervolumina von üblicherweise mehreren Gigabyte. Dieser enorme Platz reicht für nahezu alle denkbaren Anwendungen in unserer Arduino-Welt.

Zur Verbindung sind sowohl spezielle Adapter-Module als auch Shields, also Aufsteckplatinen, erhältlich.

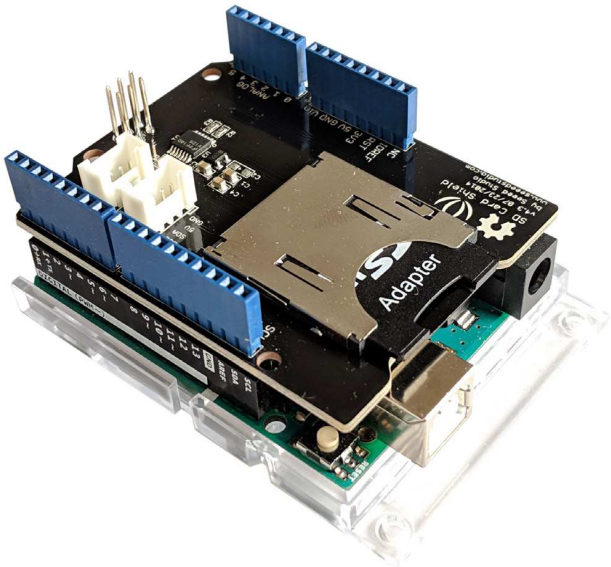


Abb. 13.6 Arduino UNO mit aufgestecktem SD-Card-Shield. An den blauen Buchsen sind weiterhin alle Pins des Arduino abgreifbar.

Unabhängig ob per separatem Modul oder Shield erfolgt die Verbindung üblicherweise per SPI. Bei Shields ist die sonst frei wählbare Chip-Select-Leitung meist mit Pin 4 verbunden.

Wir wollen nun zum Test einen Datenlogger bauen, welcher alle 15 Sekunden die Temperatur und Luftfeuchte misst und diese Informationen zusammen mit dem aktuellen Zeitstempel in eine Textdatei auf einer SD-Karte schreibt. Als Zeitmodul greifen wir wieder auf die DS3231-Echtzeituhr zurück, als Sensor dient uns ein DHT22.

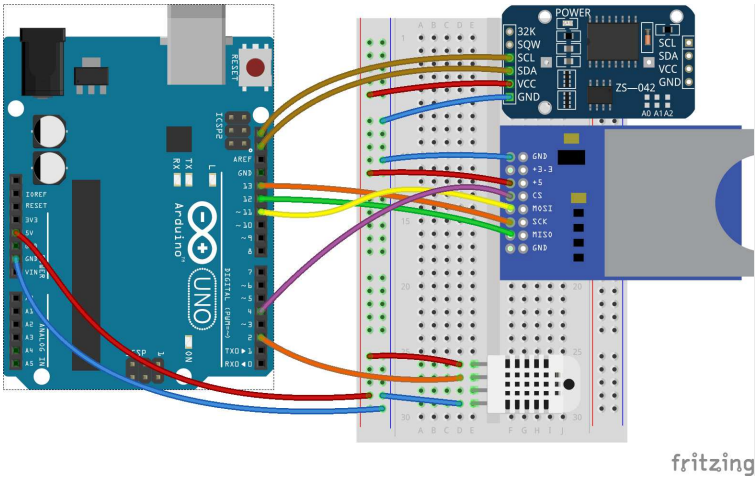


Abb. 13.7 Das Echtzeituhrmodul wird per I^2C, der SD-Karten-Adapter per SPI und der Temperatursensor an einen normalen Digitalpin angeschlossen. Wird stattdessen ein Shield genutzt, entfällt die Verbindung zum SD-Kartenadapter; die restlichen Leitungen bleiben gleich.

Als SD-Karte eignet sich jedes handelsübliche Modell, allerdings sollte sie das Dateisystem FAT16 oder FAT32 aufweisen. Andere Dateisysteme werden nicht unterstützt. Im Zweifel können Sie die Karte formatieren, indem Sie sie an einem Computer (Windows) per Rechtsklick auf die Karte die Option „Formatieren“ wählen. Dabei werden jedoch alle auf der Karte befindlichen Daten gelöscht!

Es sei noch erwähnt, dass Sie den Arduino-Adapter nicht direkt als SD-Kartenleser für Ihren Computer verwenden können. Um die auf der SD-Karte befindlichen Daten einzusehen, benötigen Sie also gegebenenfalls ein Kartenlesegerät für den Computer, falls Ihr Computer keinen integrierten SD-Kartenleser besitzt.

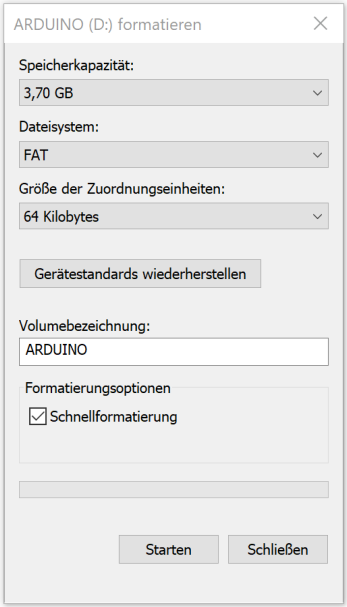


Abb. 13.8 Beispielhafte Einstellungen für das Formatieren: Wichtig ist das Dateisystem „FAT“ oder „FAT32“, die anderen Werte können je nach Größe der SD-Karte abweichen.

Die zum Ansteuern des SD-Moduls benötigte Bibliothek ist bereits in der Arduino IDE vorinstalliert, somit können wir uns direkt dem Sketch widmen. Er erfasst in 15-Sekunden-Abständen die Sensordaten, erzeugt daraus zusammen mit dem Zeitstempel einen Datensatz, welchen er in die Datei „temp_log.txt“ auf der SD-Karte schreibt und zusätzlich am seriellen Monitor ausgibt.

```
#define SPI_CS 4              // ChipSelect-Pin der SPI-Verbindung
#define SENSORPIN 2

#include "SPI.h"
#include "SD.h"
#include "DHT.h"
#include "DS3231.h"

```

```
DHT dht(SENSORPIN, DHT22);
DS3231 Uhr;

float Temp;
float Feuchte;
boolean Dummy = false;      // Wird für die DS3231-Bibliothek
                            // benötigt,
                            // siehe Kaptitel 9.11
void setup() {
  dht.begin();
  Wire.begin();

  Serial.begin(9600);
  Serial.println("SD-Karte initialisieren...");

  if (!SD.begin(SPI_CS))                                    // (1)
  {
    Serial.println("Karte nicht lesbar!");
    while (1);              // Programm anhalten
  }
  Serial.println("erfolgreich.");
}

void loop() {

  Temp = dht.readTemperature();
  Feuchte = dht.readHumidity();

  String Datensatz = "# " +                                 // (2)
    String(Uhr.getDate(),2) + "." +
    String(Uhr.getMonth(Dummy)) + "." +
    Uhr.getYear() + ",  " +
    Uhr.getHour(Dummy, Dummy) + ":";

  if(Uhr.getMinute() < 10)
    Datensatz = Datensatz + "0";

  Datensatz = Datensatz + Uhr.getMinute() + ":";

  if(Uhr.getSecond() < 10)
    Datensatz = Datensatz + "0";

  Datensatz = Datensatz + Uhr.getSecond() +
    " # Temperatur: " + String(Temp) +
    " # Feuchte: " + String(Feuchte) + " #";

  Serial.println(Datensatz);                                // (3)

```

```
57    File Zieldatei = SD.open("temp_log.txt", FILE_WRITE);     // (4)
58    if(Zieldatei)                                             // (5)
59    {
60      Zieldatei.println(Datensatz);                           // (6)
61      Zieldatei.close();                                      // (7)
62
63    }
64    else {
65      Serial.println("Datensatz konnte nicht geschrieben werden!");
66    }
67
68    delay(15000);
69  }
```

1. Die SD.h-Bibliothek stellt das Objekt `SD` bereit, welches über die Funktion `begin()` eine Verbindung aufbaut. Übergeben wird der ChipSelect-Pin, somit könnten auch mehrere Kartenleser am gleichen SPI-Bus betrieben werden. Die Funktion testet, ob eine SD-Karte vorhanden ist. Ist dies nicht der Fall, oder gibt es einen anderen Fehler, wird `false` zurückgegeben.
2. Über die nächsten Zeilen hinweg wird der Datensatz als Zeichenkette zusammengefügt.
3. Der Datensatz ist komplett und wird am seriellen Monitor ausgegeben.
4. Das Objekt `Zieldatei` vom Typ `File` repräsentiert die Textdatei, in welche wir die Daten schreiben wollen. Sie wird per `open()` geöffnet. Dateinamen dürfen hier nur acht Zeichen lang sein (plus drei Zeichen für die Namenserweiterung).
5. Schlug das Öffnen der Datei fehl, ist das Objekt leer und damit `false`, ansonsten `true`.
6. Das Schreiben in die Datei erfolgt ähnlich wie beim seriellen Monitor.
7. Wichtig ist jedoch, die Datei wieder zu schließen, sofern man nicht sofort noch weitere Daten schreiben möchte. Tut man es nicht, können Daten verlorengehen, wenn vor dem `close()`-Befehl die Betriebsspannung getrennt wird oder ein Reset erfolgt.

In der Datei „temp_log.txt“ werden neue Daten immer an das Ende angefügt. Ihr Inhalt sieht genau so aus wie die Ausgabe am seriellen Monitor, lediglich die Statusmeldungen in den ersten beiden Zeilen fehlen.

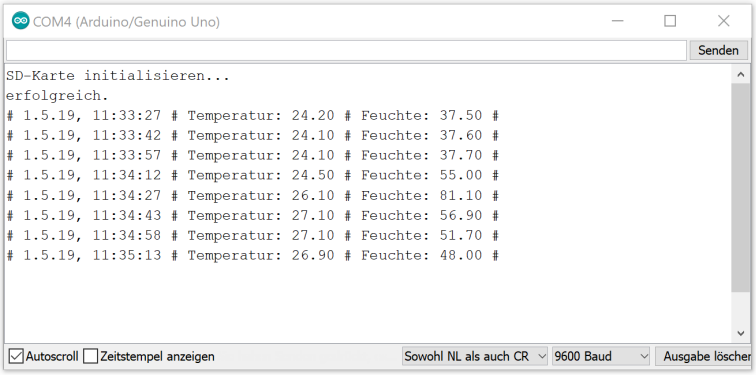


Abb. 13.9 beispielhafte Bildschirmausgabe

Natürlich ist es auch denkbar, die Daten einfach durch Kommata oder Semikolons zu trennen, um die Datei dann als CSV (*Comma-Separated Values*) in eine Datenverarbeitungssoftware wie beispielsweise Microsoft Excel zu importieren. Über diesen Weg sind auch grafische Auswertungen und Analysen möglich.

13.2 Processing

Bei einigen Projekten ist es sinnvoll, einen PC mit einzubeziehen, zum Beispiel, um dem Nutzer eine grafische Benutzeroberfläche zu bieten oder umfangreiche Berechnungen auszuführen. Aufgrund der standardisierten seriellen Schnittstelle am Arduino ist prinzipiell die Kommunikation mit jeglicher Software möglich, welche auf den seriellen Anschluss des Computers zugreifen kann.

Wir wollen unseren Blick nun im speziellen auf die *Processing*-Plattform richten. Sie wurde etwa zeitgleich mit der Arduino-Platt-

form entwickelt und richtet sich vorwiegend an Künstler, Gestalter und Programmieranfänger, denn sie erlaubt die Erstellung grafischer Nutzeroberflächen mit vergleichsweise wenig Aufwand. Ein besonderer Vorteil für uns ist, dass die Arduino IDE einst von der Processing IDE abgeleitet wurde, daher werden wir uns schnell zurechtfinden.

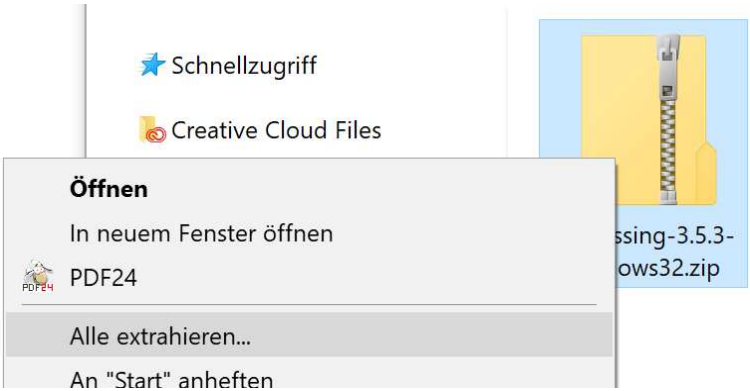
13.2.1 Einrichtung

Bevor wir starten, muss die Software von der offiziellen Projektseite

https://processing.org/download/

heruntergeladen werden. Es stehen Versionen für alle gängigen Betriebssysteme zur Verfügung, im Folgenden orientieren wir uns an der Windows-Version. Da es sich bei Processing ebenfalls um ein freies Software-Projekt handelt, haben Sie im Zusammenhang mit dem Download wieder die Möglichkeit, Geld zu spenden.

Im Gegensatz zur Arduino IDE ist bei Processing keine Installation notwendig. Die heruntergeladene Datei ist ein zip-Archiv, welches lediglich extrahiert werden muss – ein einfacher Rechtsklick bietet diese Option an. Der Zielordner ist frei wählbar.



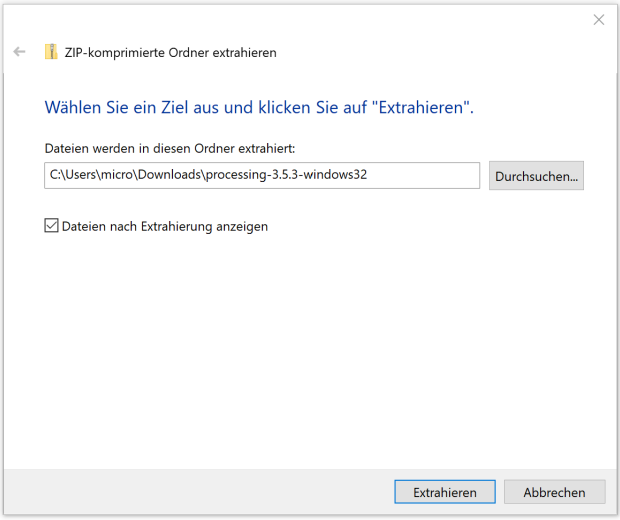


Abb. 13.10 Die heruntergeladene Datei kann per Rechtsklick extrahiert werden

Nachdem die Extrahierung abgeschlossen ist, befindet sich am Zielort der Ordner mit allen zur Plattform gehörenden Dateien. Ein Doppelklick auf „processing.exe“ führ zum Programmstart.

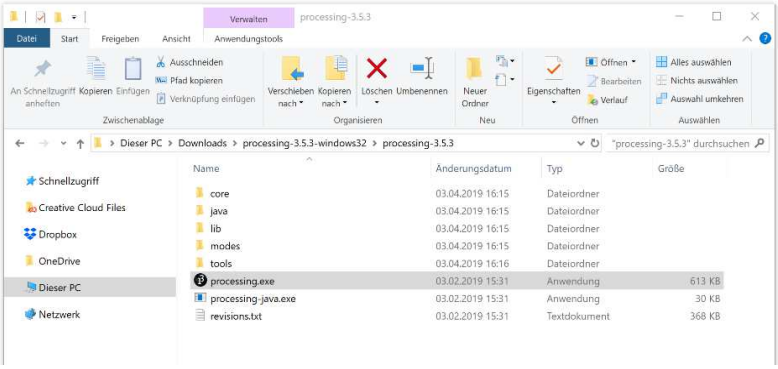


Abb. 13.11 Die Datei processing.exe startet das Programm

Um die Software später bei Bedarf wieder zu entfernen, genügt es, den soeben extrahierten Ordner zu löschen.

13.2.2 Prinzip

Nach dem Programmstart werden Sie die deutliche Ähnlichkeit zur Arduino IDE erkennen. Der Programmcode wird wieder als *Sketch* bezeichnet und befindet sich in der Mitte, oben links ermöglicht ein angedeutetes *Play*-Symbol den Programmstart (und ersetzt dadurch das von Arduino bekannte *Hochladen*). Am unteren Rand befindet sich die schwarz hinterlegte Konsole für eventuelle Status- oder Fehlermeldungen.

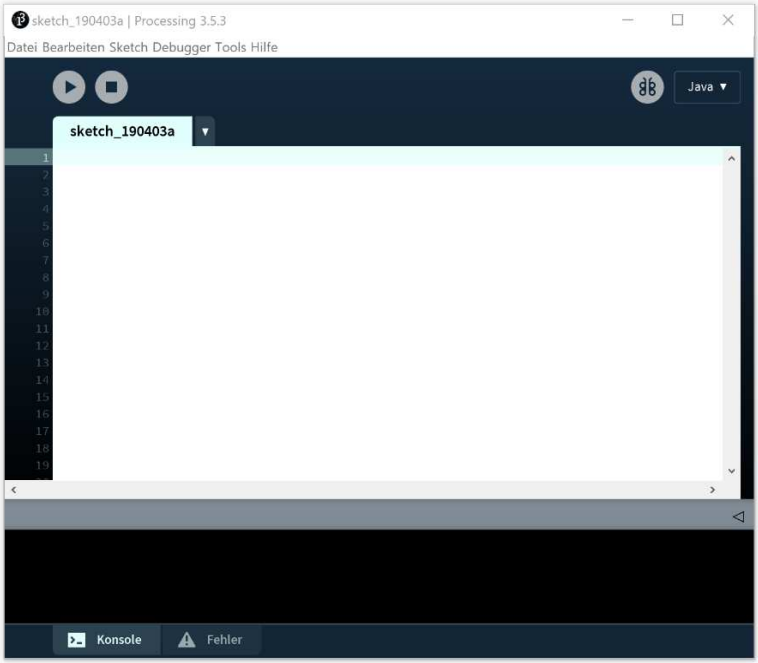


Abb. 13.12 Die Programmoberfläche ähnelt der Arduino IDE

Um einen ersten Eindruck von den Möglichkeiten der Plattform zu erhalten, lohnt es sich, einen Blick in die mitgelieferten Beispiele zu werfen. Die folgenden drei wurden stellvertretend ausgewählt. Sie können sie selbst testen, indem Sie über *Datei -> Beispiele* den entsprechenden Sketch laden und dann die Play-Schaltfläche (direkt unter dem Menü *Bearbeiten*) aktivieren.

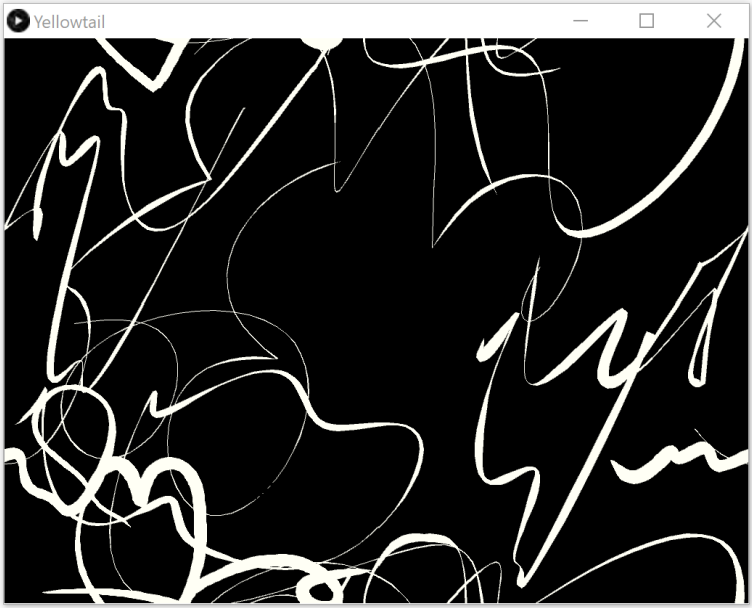


Abb. 13.13 Das Beispiel „Demos -> Graphics -> Yellowtail“ reagiert dynamisch auf Zeichnungen mit der Maus und wandelt sie in kleine animierte Kunstwerke um

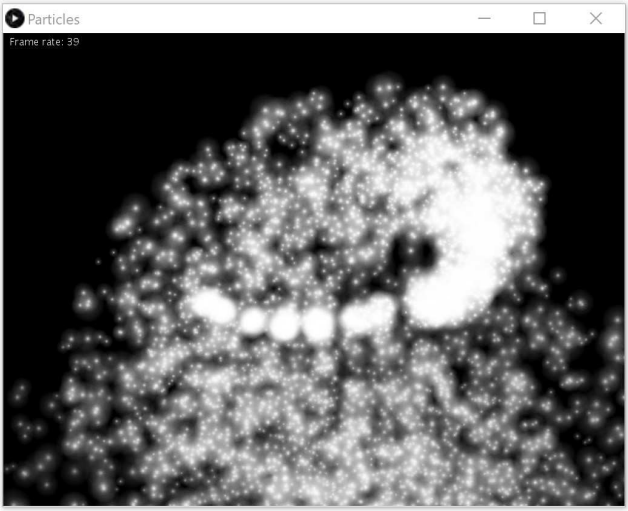


Abb. 13.14 Das Beispiel „Demos -> Graphics -> Particles“ verwandelt den Mauszeiger in eine Wunderkerze

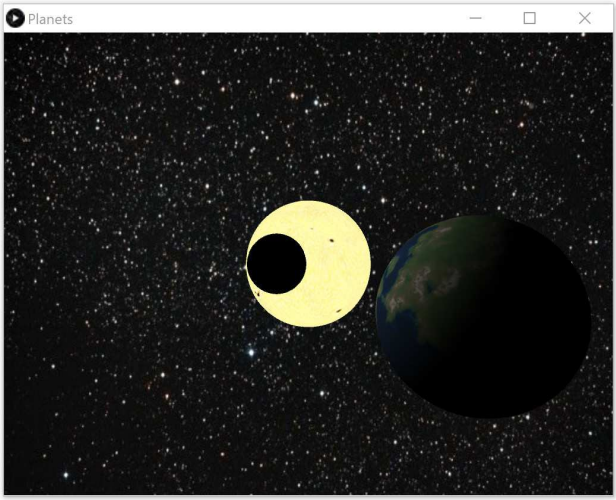


Abb. 13.15 Das Beispiel „Demos -> Graphics -> Planets“ simuliert einfache Planetenbewegungen

Es wird schnell offensichtlich, dass bei dieser Plattform die grafische Interaktion mit dem Nutzer im Vordergrund steht. Wird ein Sketch gestartet, öffnet sich ein neues Fenster, in welchem das betreffende Programm agiert. Wir können uns dies zu Nutze machen, um beispielsweise Schaltflächen, Regler und Anzeigen für unsere Arduino-Projekte zu gestalten.

13.2.3 Beispiel

Eine umfassende Einführung in die Programmierung mit Processing würde ein separates Buch füllen. Dennoch lohnt es sich, zumindest einige grundlegende Befehle kennenzulernen. Ist diese Basis gelegt, werden Sie bei Bedarf schnell in der Lage sein, sich weitere Kenntnisse anzueignen.

Der erste Arduino-Sketch, den wir kennenlernten, trug den Titel *Blink* und hatte einzig die Funktion, eine LED blinken zu lassen. Wir wollen in Anlehnung daran auch bei Processing mit einem Blink-Sketch beginnen. Er wird ein Rechteck auf dem Bildschirm erscheinen lassen, welches abwechselnd rot und grün blinkt.

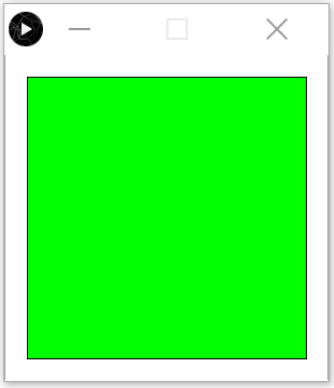


Abb. 13.16 Unser erster Processing-Sketch wird dieses Rechteck rot-grün-blinken lassen

Erstellen Sie dazu den folgenden Sketch in der Processing IDE:

```
void setup()
{
  size(300,300);                                    // (1)
  background(255);                                  // (2)
  stroke(0);                                        // (3)
}

void draw()
{
  if(millis() % 1000 > 500)                         // (4)
    fill(255,0,0);                                  // (5)
  else
    fill(0,255,0);                                  // (6)

  rect(20,20,260,260);                              // (7)
}
```

Auch hier ist wieder sofort die Ähnlichkeit zur Arduino-Plattform erkennbar. Es gibt eine `setup()`-Funktion, welche nur einmal bei Programmstart durchlaufen wird und eine weitere Funktion, welche sich ständig wiederholt. Allerdings heißt diese nun `draw()` statt `loop()`. Der veränderte Name weist lediglich darauf hin, dass diese Funktion ständig die Benutzeroberfläche neu „zeichnet“. Die uns noch unbekannten Funktionen sind entsprechend auch hauptsächlich grafischer Natur:

1. Die Funktion `size()` legt die Größe des Fensters unserer Benutzeroberfläche fest. Die Argumente bestimmen die Breite und Höhe in Pixeln.

2. Die Funktion `background()` legt die Hintergrundfarbe fest. Wie alle Funktionen, die eine Farbangabe benötigen, erwartet sie drei Zahlen für die Helligkeitswerte von rot, grün und blau. Es handelt sich dabei immer um 8-Bit-Werte, also im Bereich von 0 bis 255. Sind alle drei Werte gleich, kann diese Helligkeit auch einfach als einziges Argument übergeben werden. Hier wird also die Hintergrundfarbe weiß gewählt (rot, grün und blau jeweils auf voller Helligkeit).

3. Die Funktion `stroke()` wählt die Farbe für die Umrandung von gezeichneten Objekten. Standardmäßig haben alle Objekte einen Rand von 1 Pixel. Hier wird schwarz dafür gewählt.
4. Diese Bedingung kennen Sie bereits aus Kapitel 10.4. Sie ist im ständigen Wechsel für eine halbe Sekunde `true` und für die folgende halbe Sekunde `false`, sie erzeugt damit das Blinken.
5. Die Funktion `fill()` wählt die Füllfarbe für Zeichenobjekte vor. Wohlgemerkt: Genau wie bei der Funktion `stroke()` wird dadurch noch nichts gezeichnet. Es wird nur eine Farbe ausgewählt, welche dann bei kommenden Zeichenbefehlen zum Tragen kommt. Die Argumente sind in der Reihenfolge (rot, grün, blau) notiert. Dementsprechend wird hier die Farbe Rot vorgewählt, wenn die `if`-Bedingung erfüllt ist,
6. sonst fällt die Farbwahl auf die Farbe Grün.
7. Hier nun der eigentliche Zeichnungsbefehl. Die Funktion `rect()` zeichnet ein Rechteck (*Rectangle*) mit der jeweils vorgewählten Füll- und Randfarbe. Argumente sind die X- und Y-Position der oberen linken Ecke sowie die Breite und Höhe in Pixel. In diesem Fall wird also ein 260 Pixel hohes und 260 Pixel breites Rechteck gezeichnet, welches vom linken Rand 20 Pixel entfernt ist und zum oberen Rand ebenfalls 20 Pixel Abstand hält. Die Zählweise der Positionsangaben haben Sie bereits beim OLED-Display kennengelernt, man beginnt stets oben links, die Zählung fängt bei 0 an.

Nutzen Sie den Sketch gern, um mit den Positions- und Farbwerten zu experimentieren und das Ergebnis zu beobachten. Nach jeder Änderung im Sketch müssen Sie erneut auf die Play-Schaltfläche klicken, um die Auswirkung zu sehen.

13.2.4 Kommunikation mit dem Arduino

An einem weiteren Beispiel wollen wir die Kommunikation zwischen Processing und dem Arduino untersuchen. Wir wollen Sensorwerte

und Tasteneingaben vom Arduino am Computer grafisch ausgeben und umgekehrt eine LED über die Benutzeroberfläche ein- und ausschalten. Ein geeigneter Beispielaufbau ergibt sich, indem wir den EEPROM-Versuch um einen lichtempfindlichen Widerstand erweitern:

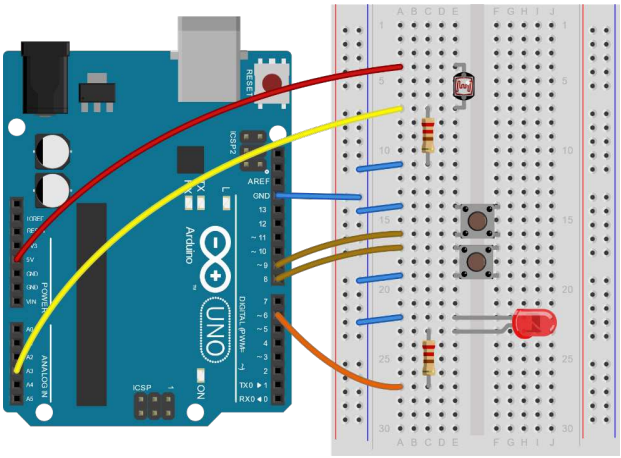


Abb. 13.17 Der Versuchsaufbau aus Kapitel 13.1.1 wird um den Fotowiderstand erweitert. Er bildet einen Spannungsteiler zusammen mit einem 1-Kiloohm-Widerstand.

An Pin A3 liegt nun also ein analoger Helligkeitsmesswert an, Pins 8 und 9 sind als Pull-Up-Eingänge mit Tastern verbunden und über Pin 6 kann eine LED angesteuert werden. All dies wollen wir nun in Processing abbilden.

Bevor wir die Programme für beide Plattformen schreiben, müssen wir uns Gedanken machen, wie die Kommunikation zwischen beiden laufen soll. Die serielle Verbindung ist relativ unkompliziert in der Lage, einzelne Zeichen und Zeichenketten zu übertragen. Zur Meldung der Tasterzustände und der Helligkeit an den PC kann der Arduino also eine Zeichenkette mit diesen Informationen senden. Für die Datenverarbeitung ist es immer vorteilhaft, wenn solche Datenpakete neben einer festgelegten Form auch eine feste Länge haben. Für unser Beispiel würde sich zum Beispiel folgende Form eignen:

```
A100736E
```

Dabei legen wir (willkürlich) fest, dass ein Datenpaket immer mit dem Buchstaben „A“ beginnt, um den Anfang zu markieren. Es folgen die eigentlichen Daten als Ziffern, und zwar

- an erster Stelle der Status des ersten Tasters (1 oder 0),
- an zweiter Stelle Status des zweiten Tasters (1 oder 0),
- danach vier Ziffern für den eingelesenen Helligkeitswert (von 0 bis 1023), und zwar immer vierstellig, also gegebenenfalls mit führenden Nullen.

Den Abschluss bildet der Buchstabe „E“ als Endmarkierung. Somit sind die Datenpakete immer exakt 8 Zeichen lang.

In die Gegenrichtung wird lediglich ein Signal benötigt, welches die LED ein- beziehungsweise ausschaltet. Dafür genügt es, ein einzelnes Zeichen zu übertragen. Wir legen willkürlich fest: „e“ bedeutet *LED ein* und „a“ schaltet die *LED aus*.

Nun können wir bereits unseren Arduino-Sketch schreiben:

```
#define TASTER1 9
#define TASTER2 8
#define LEDPIN 6

boolean Knopf1gedrueckt = false;
boolean Knopf2gedrueckt = false;
int Sensorwert = 0;
char Empfangsdaten;

char Datenpaket[] = "A000000E";

void setup() {
  Serial.begin(9600);
  pinMode(TASTER1, INPUT_PULLUP);
  pinMode(TASTER2, INPUT_PULLUP);
  pinMode(6, OUTPUT);
}

```

```
19 void loop()
20 {
21   Knopf1gedrueckt = !digitalRead(TASTER1);
22   Knopf2gedrueckt = !digitalRead(TASTER2);
23   Sensorwert = analogRead(3);
24
25   sprintf(Datenpaket,"A%1d%1d%04dE", Knopf1gedrueckt,
26   Knopf2gedrueckt,Sensorwert);                          // (1)
27   Serial.println(Datenpaket);                           // (2)
28
29   delay(50);
30
31   if(Serial.available())                                // (3)
32   {
33     char Empfangsdaten = Serial.read();
34
35     if(Empfangsdaten == 'e')
36       digitalWrite(LEDPIN, HIGH);
37
38     if(Empfangsdaten == 'a')
39       digitalWrite(LEDPIN, LOW);
40    }
41 }
```

1. Die Funktion `sprintf()` kennen Sie noch nicht. Sie dient dazu, eine Zeichenkette in einem bestimmten Format zu erstellen. Als Argumente erwartet sie als erstes die Variable (`char`-Array), in welcher das Ergebnis gespeichert werden soll. Das zweite Argument ist ein Muster, also quasi die Formvorgabe für die Zeichenkette. In diesem Muster sind bestimmte Formatierungszeichen enthalten. `%1d` steht beispielsweise dafür, dass an dieser Stelle eine Dezimalzahl mit einer Ziffer stehen soll. %04d symbolisiert eine vierstellige Dezimalzahl mit führenden Nullen. Als weitere Argumente sind die entsprechenden Zahlenwerte in der Reihenfolge ihres Vorkommens im Muster anzugeben. Nach Ausführung dieser Funktion enthält unsere Zeichenkette also die entsprechenden Werte und hat die gewünschte Form, zum Beispiel `A110298E` wenn beide Taster gedrückt sind und der Helligkeitswert bei 298 liegt.

2. Die Übertragung erfolgt genau wie unsere bisherigen Datenausgaben am PC. Im Unterschied läuft dann auf dem PC allerdings nicht der serielle Monitor, sondern die Processing IDE. Dennoch ist es natürlich möglich, die Datenpakete auch am seriellen Monitor auszugeben, zum Beispiel zur Fehlersuche. Allerdings kann immer nur ein Programm auf den seriellen Anschluss zugreifen, das andere erhält beim gleichzeitigen Versuch eine Fehlermeldung.
3. Hier erfolgt der Datenempfang, die Funktionsweise haben Sie bereits in Kapitel 4.8 kennengelernt.

Damit ist der Arduino-Programmcode komplett. Aus der Sicht der Mikrocontrollers handelt es sich also um eine normale Kommunikation über die serielle Schnittstelle, wie man sie auch für die Datenausgabe am seriellen Monitor verwenden würde.

In Bezug auf den Processing-Sketch lohnt es sich, zunächst über die Gestaltung der Nutzeroberfläche nachzudenken und anhand dessen den Programmcode zu entwickeln. Wir legen (wieder willkürlich) fest:

- Der Status der Knöpfe soll durch zwei Rechtecke symbolisiert werden. Bei gedrücktem Knopf erscheinen sie grün, sonst rot.
- Der Helligkeitswert soll als Zahl dargestellt und als Balken visualisiert werden.
- Es gibt je eine Schaltfläche, um die LED ein- und auszuschalten.

Nachfolgend sei zur Anschaulichkeit bereits das Resultat des Autors vorgegriffen, bei der realen Softwareentwicklung hat man zu diesem Zeitpunkt üblicherweise zumindest eine Skizze angefertigt.

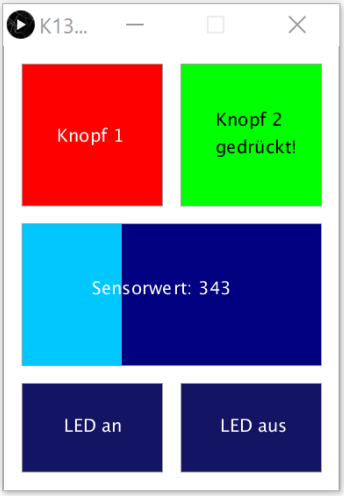


Abb. 13.18 die Bedienoberfläche unseres Projektes

In diesem Beispiel erfolgt die Visualisierung des Sensorwertes durch einen beweglichen Balken. Je größer der Messwert, desto größer wird der hellblaue Anteil des mittleren Balkens. Die unteren beiden Rechtecke können mit der Maus angeklickt werden, um die LED entsprechend zu schalten. Bewegt man die Maus über diese Schaltflächen, ändern sie geringfügig ihre Farbe, um auf die Klickmöglichkeit hinzuweisen.

Betrachten wir den zugehörigen Processing-Sketch:

```
1  import processing.serial.*;                                    // (1)
2
3  Serial seriellerPort;
4  String Datenpaket;
5
6  int Knopf1 = 0;      // Auch "kleine" Variablen werden
7                       // hier üblicherweise
8  int Knopf2 = 0;      // mit großen Datentypen
9                       // ausgestattet, da am PC
10 int Sensorwert = 0; // keine Notwendigkeit herrscht, Speicher
11                      // zu sparen.
12                      // Für Anfänger hat dies den Vorteil, dass
```

```
                    // viele
                    // Typumwandlungen als Fehlerquelle entfallen
                    // können.
void setup()
{
  size(380,500);    // Fenstergröße (380 Pixel breit, 500 Pixel
                    // hoch)
  background(255);  // weiße Hintergrundfarbe
  stroke(160);      // graue Umrandungen
  textSize(20);     // Textgröße 20

  printArray(Serial.list());
  String PortName = Serial.list()[0];                        // (2)
  seriellerPort = new Serial(this, PortName, 9600);          // (3)

  seriellerPort.bufferUntil('\n');                           // (4)
}

void draw()
{
  if(Datenpaket != null) {// Wurde ein Datenpakt empfangen?
    if (Datenpaket.length() == 10                            // (5)
         && Datenpaket.charAt(0) == 'A'
         && Datenpaket.charAt(7) == 'E')
    {

      Knopf1 = int(Datenpaket.substring(1,2));               // (6)
      Knopf2 = int(Datenpaket.substring(2,3));

      Sensorwert = int(Datenpaket.substring(3,7));

    // Knopf-Flächen zeichnen:

      if (Knopf1 == 1)
      {                              // falls Knopf 1 gedrückt ist,
        fill(0,255,0);               // grün vorwählen
        rect(20,20,160,160);         // Rechteck zeichnen

        fill(0,0,0);                 // schwarz vorwählen
        text("Knopf 1\ngedrückt!",  60,90);                  // (7)
      }
      else
      {                       // falls Knopf 1 nicht gedrückt ist,
        fill(255,0,0);        // rot vorwählen
        rect(20,20,160,160); // Rechteck zeichnen

        fill(255,255,255);           // weiß vorwählen
        text("Knopf 1",  60,108);
```

```
61        }
62
63        if (Knopf2 == 1)
64        {
65          fill(0,255,0);
66          rect(200,20,160,160);
67
68          fill(0,0,0);
69          text("Knopf 2\ngedrückt!",  240,90);
70        }
71        else
72        {
73          fill(255,0,0);
74          rect(200,20,160,160);
75
76          fill(255,255,255);
77          text("Knopf 2",  240,108);
78        }
79
80      // Sensorbalken zeichnen:
81
82        fill(0,0,128);          // dunkelblauen
83        rect(20,200,340,160); // Hintergrund für den Sensorbalken
84                                // zeichnen
85
86        fill(0,200,255);        // hellblau vorwählen
87        rect(20,200,map(Sensorwert,0,1023,0,340),160);
88                                // Balken zeichnen; Balkenbreite
89                                // entsprechend des Sensorwertes,
90                                // umgerechnet in den Bereich von 0
91                                // bis 340
92                                // Pixel.
93
94        fill(255);              // weiß vorwählen
95        text("Sensorwert: " + Sensorwert, 100,280);
96                                // Sensorwert auf Balken schreiben
97      }
98    }
99
100   // LED-Schaltflächen zeichnen:
101
102   if(MausueberEinschaltknopf())     // Ist der Mauszeiger gerade
103                                     // auf dem Knopf?
104     fill(150,150,200);              // wenn ja: hellere Farbe
105                                     // vorwählen
106   else
107     fill(20,20,100);                // wenn nein: dunklere Farbe
108                                     // vorwählen
```

```
109   rect(20,380,160,100);              // Einschaltfläche zeichnen
110
111   if(MausueberAusschaltknopf())
112     fill(150,150,200);
113   else
114     fill(20,20,100);
115   rect(200,380,160,100);             // Ausschaltfläche zeichnen
116
117   fill(255);
118   text("LED an",  68,435);           // LED_Knöpfe beschriften
119   text("LED aus",  245,435);
120
121 }                                    // Hier endet die draw()-
122                                      // Schleife,
123                                      // es folgen
124                                      // Funktionsdefinitionen.
125
126
127 void serialEvent(Serial seriellerPort)                      // (8)
128 {
129   Datenpaket = seriellerPort.readString();
130 }
131
132 boolean MausueberEinschaltknopf()                           // (9)
133 {
134   if(mouseX >= 20 && mouseX <= 180 && mouseY >= 380 &&
135   mouseY <= 480)
136     return true;
137   else
138     return false;
139 }
140
141 boolean MausueberAusschaltknopf()
142 {
143   if(mouseX >= 200 && mouseX <= 360 && mouseY >= 380 &&
144   mouseY <= 480)
145     return true;
146   else
147     return false;
148 }
149
150 void mousePressed()                                         // (10)
151 {
152   if(MausueberEinschaltknopf())
153     seriellerPort.write('e');
154   if(MausueberAusschaltknopf())
155     seriellerPort.write('a');
156 }
```

1. Auch bei Processing gibt es Bibliotheken, die Schreibweise weicht hier leicht von der Arduino-Welt ab. Die vorinstallierte Bibliothek processing.serial.* benötigen wir für die serielle Schnittstelle.

2. In der Arduino IDE gibt es die Möglichkeit, im Menü und im seriellen Monitor den gewünschten seriellen Anschluss auszuwählen. In der Regel ist dies nicht notwendig, da es bei den meisten aktuellen Computern nur einen gibt. Bei Processing gibt es diese Einstellmöglichkeit in der Software nicht, stattdessen muss der gewünschte Port im Sketch ausgewählt werden. Man könnte also statt `PortName` auch direkt „`COM3`“ schreiben, nur leider ändert sich die Nummer der Ports unter gewissen Umständen, und dann würde der Sketch nicht mehr funktionieren.

   Um dies zu vermeiden, wird hier aus der Liste alle verfügbarer COM-Ports der erste Eintrag ausgewählt. Sollte dies nicht funktionieren (weil Sie eventuell mehrere serielle Anschlüsse haben), ändern Sie testweise den Index von 0 auf 1 oder noch höher. Zur einfacheren Fehlersuche gibt der in der Zeile darüber befindliche Befehl die Liste der verfügbaren COM-Ports in der Konsole (im schwarzen Bereich unten am Bildschirm) aus.

3. Die serielle Verbindung wird hergestellt, indem ein neues Objekt (`seriellerPort`) aus der Klasse `Serial` erstellt wird. Dabei werden als Argumente der Name des seriellen Anschlusses und die Geschwindigkeit übergeben. Das Schlüsselwort `this` rührt von der hier verwendeten objektorientierten Programmierung her und stellt lediglich einen Bezug dar.

4. Diese Funktion definiert, woran der Abschluss eines Datenpaketes erkannt werden soll. Da wir am Arduino die Daten per `println()`, also mit Zeilenumbruch, senden, können wir daran das Ende erkennen. Der Zeilenumbruch selbst ist als Steuerzeichen nicht sichtbar, man kann ihn jedoch durch die Zeichenkombination `\n` repräsentieren.

5. Wurde ein Datenpaket empfangen, wird zunächst getestet, ob es den formalen Vereinbarungen entspricht. (Wenn dies nicht der Fall ist, wurde die Übertragung vermutlich gestört.)

Es wird getestet, ob die Länge exakt 10 Zeichen (8 Zeichen Datenpaket + 2 Zeichen Zeilenumbruch) beträgt und ob Anfang und Ende wie vereinbart aus „A“ beziehungsweise „E“ bestehen.

6. Sind die formalen Anforderungen erfüllt, werden die Daten aus der Zeichenkette extrahiert. Die Funktion `substring()` hilft dabei, indem sie einen bestimmten Teil aus einer Zeichenkette herausschneidet. Das erste Argument gibt die Anfangsstelle an, das zweite Argument markiert das Ende. Dabei gilt: Die Anfangsstelle wird mit einbezogen, die Endstelle jedoch nicht. Im konkreten Fall `(1,2)` bedeutet das: „Ab Stelle 1 bis vor Stelle 2“ – also nur Stelle 1 und damit nur das zweite Zeichen der Zeichenkette, denn die Zählung beginnt auch hier bei „Stelle 0“.

   Die übergeordnete Funktion `int()` wandelt den Rückgabewert (Typ `char`) wie gewünscht in eine `int`-Variable um.

7. Die Funktion `text()` schreibt eine Zeichenkette in der vorgewählten Farbe und Schriftgröße an die Stelle, welche in den Argumenten übergeben wurde. Auch hier markiert `\n` einen Zeilenumbruch.

8. Dies ist eine Eigenheit von Processing, diese Funktion wird automatisch aufgerufen, wenn das in (4) vereinbarte Zeichen empfangen wird. Wir übertragen in dem Falle die Daten aus dem Empfangs-Zwischenspeicher in die Variable `Datenpaket`, welche dann weiterverarbeitet werden kann.

9. Diese selbstdefinierte Funktion prüft lediglich, ob sich der Mauszeiger (dessen Position immer über die Systemvariablen `mouseX` und `mouseY` abrufbar ist) gerade über dem LED-Einschaltknopf befindet und gibt entsprechend `true` oder `false` zurück.

10. Die Funktion `mousePressed()` ist ein festgelegter Begriff, sie wird immer dann aufgerufen, wenn eine Maustaste gedrückt wird. Damit ähnelt sie einer Interrupt-Funktion in der Arduino-Welt. Wir prüfen innerhalb der Funktion nun, ob die Maus gerade auf einer Schaltfläche steht. Befindet sie sich beispielsweise gerade auf dem LED-Einschalteknopf, senden wir das vereinbarte Zeichen „e“ an den Arduino. Für den Ausschalteknopf senden wir stattdessen das „a“. Befindet sich die Maus gerade über gar keiner Schaltfläche, passiert nichts.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 14

Praxis-Projekt: LED-Matrix mittels Processing steuern

14.1 Idee

In einem weiteren praktischen Beispiel wollen wir Processing nutzen, um die bereits betrachteten adressierbaren LEDs aus Kapitel 7.7 komfortabel ansteuern zu können. Damit wären Anwendungen als Laufschrift oder dynamisches Hinweisschild denkbar.

Ziel ist eine Nutzeroberfläche, welche es erlaubt, einzelne Pixel einer 8x8-RGB-Matrix beliebig einzufärben. Zudem soll ein Lauftext hinterlegt sein. Als Zusatz kann die Matrix mit Zufallsdaten gefüllt werden oder ein Schachbrettmuster anzeigen. Dabei soll zu jeder Zeit die Bildschirmanzeige am PC mit dem tatsächlichen Status der LEDs übereinstimmen.

14.2 Konzeption

In die Vorabüberlegungen ist einzubeziehen, welche Daten vom PC an den Arduino übertragen werden müssen und welches Format dafür geeignet ist. Rufen wir uns dazu nochmal den Aufbau der Matrix ins Gedächtnis. Sie besteht aus 64 adressierbaren LED-Modulen vom Typ WS2812B. Entsprechend der Zickzack-Verbindung ihrer Datenleitung ergeben sich auch die Adressen.

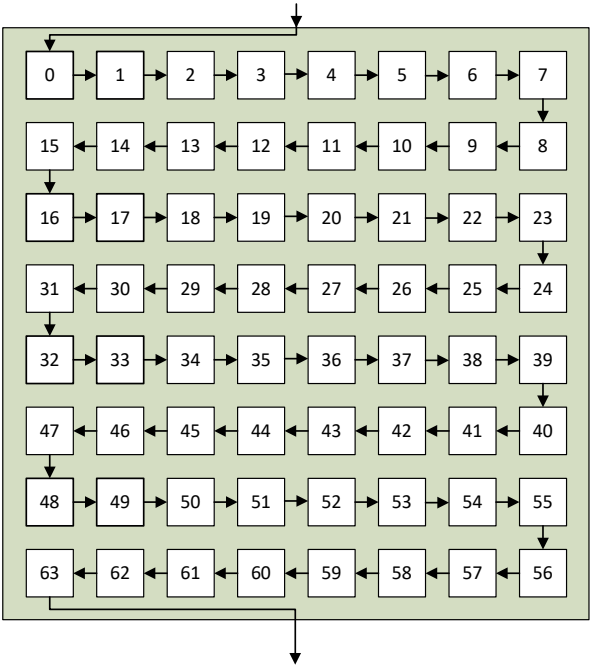


Abb. 14.1 Die adressierbaren WS2812B-LEDs der Matrix sind im Zickzack-Muster verbunden, entsprechend ergeben sich ihre Adressen

Soll nun ein einzelnes Pixel umgefärbt werden, genügt es, dessen Position sowie die neuen Farbwerte für rot, grün und blau zu übertragen. Die Position kann dabei schon vom Computer in die entsprechende Adresse gemäß obigem Schema umgerechnet werden. Man könnte also in Anlehnung an das vorangegangene Beispiel ein Datenpaket nach dem Muster

```
1 AaarrrgggbbbE
```

festlegen, wobei „aa“ die zweistellige Adresse und die anderen Kleinbuchstaben jeweils die dreistelligen Farbwerte repräsentieren. Der Arduino erhält diesen Datensatz, fügt ihn in seinen Arbeitsspeicher ein und schickt die einzelnen Farbwerte erneut an alle LEDs. (Aufgrund des

Kommunikationsprotokolls der WS2812B-Baureihe muss der Arduino immer mindestens auch alle vor der zu ändernden LED liegenden Pixel erneut mit Daten versorgen, siehe Kapitel 7.7.)

Hat sich nun der komplette Bildinhalt geändert und muss somit für jedes Pixel ein Datensatz an den Arduino geschickt werden, ist es nicht sinnvoll, nach jedem Datenpaket wieder alle LEDs zu aktualisieren – denn das benötigt eine gewisse Zeit, in welcher der Arduino keine weiteren Pakete empfangen kann. Stattdessen wäre es besser, wenn diese Aktualisierung erst nach dem Empfang des letzten Datenpaketes vom PC erfolgt.

Zu diesem Zweck vereinbaren wir, dass es neben dem „E“ als Abschluss noch eine zweite Möglichkeit des Paketendes gibt: „N“, es bedeutet, dass unmittelbar noch weitere Pakete folgen und der Arduino mit der Aktualisierung der LEDs noch warten soll.

Eine Übertragung mit dem Datensatz

```
1 A42128020255N
```

würde also dem Pixel mit der Adresse 42 den Farbwert 128 (rot), 20 (grün), 255 (blau) zuweisen und dafür sorgen, dass der Arduino gleich wieder auf Empfangsbereitschaft geht, ohne die Daten an die LEDs weiterzuleiten. Processing muss entsprechend sicherstellen, dass das letzte Datenpaket eines Vorgangs mit einem „E“ abgeschlossen wird, damit das Pixelmuster an die LEDs übertragen wird.

Die Oberfläche am PC soll intuitiv bedienbar sein, indem man über Farbregler oder Schaltflächen einen Farbton anwählt und dann die gewünschten Pixel auf einer nachgebildeten Matrix anklickt. Separate Schaltflächen ermöglichen das Löschen/Zurücksetzen, die Aktivierung eines Lauftextes, das Erzeugen von Zufallsdaten sowie die Anzeige eines Schachbrettmusters. Eine Skizze dient anfangs zur Orientierung:

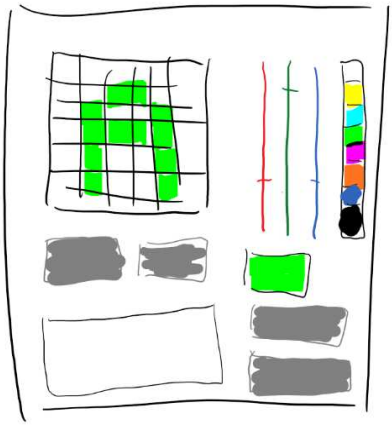


Abb. 14.2 Skizze der geplanten Benutzeroberfläche

Das große Rasterfeld im linken oberen Bereich der Bedienoberfläche soll die LED-Matrix repräsentieren. Rechts daneben dienen drei Schieberegler zur Anwahl eines Farbtons, das Rechteck darunter zeigt die gewählte Farbe als Vorschau an. Alternativ können direkt Farben aus den Kästchen rechts daneben angeklickt werden. Weitere Schaltflächen sind grau hinterlegt, ein Textfeld unten links gibt Informationen aus, welche die Funktion besser verdeutlichen und während der Entwicklung die Fehlersuche erleichtern.

14.3 Versuchsaufbau

Wir benötigen:

- 1 Arduino UNO
- 1 WS2812B-Matrix mit 64 Elementen
- Verbindungsdrähte

Die Verdrahtung ist einfach, sie entspricht dem Beispiel aus Kapitel 7.7:

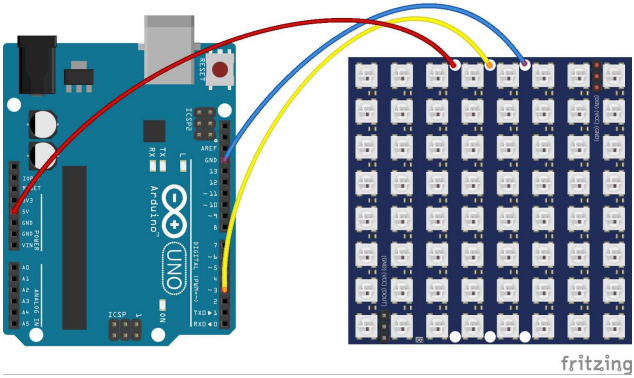


Abb. 14.3 Da das Datensignal bei WS2812B-LEDs nur einen Draht benötigt, ist der Versuchsaufbau sehr einfach.

14.4 Programmcode (Arduino)

Widmen wir uns zunächst dem Arduino-Sketch. Er muss die Datenpakete über die serielle Schnittstelle empfangen und verarbeiten, außerdem sendet er die entsprechenden Daten an die adressierbaren LEDs. Deren Ansteuerung wurde bereits in Kapitel 7.7 erläutert und wird nachfolgend nicht mehr thematisiert.

```
#include "FastLED.h"

#define NUM_LEDS 64
#define DATA_PIN 3
#define PAKETLAENGE 15       // Paketlänge: 13 Zeichen + 2 Zeichen
                             // Zeilenumbruch

CRGB leds[NUM_LEDS];

char Datenpaket[PAKETLAENGE];  // ...dient dem Zwischenspeichern
                               // des empfangenen Datenpaketes.

void setup() {
 Serial.begin(57600);                                        // (1)
 FastLED.addLeds<WS2812B, DATA_PIN, GRB>(leds, NUM_LEDS);
}

```

```
18 void loop() {
19
20   if(empfangen())                                          // (2)
21     auswerten();                                           // (3)
22
23 }
24
25
26 boolean empfangen()
27 {
28  static byte Zaehler = 0;                                  // (4)
29  char Endmarkierung = '\n';
30  char Empfangsbyte;
31
32  while (Serial.available() > 0)      // solange noch Daten
33                                      // im Empfangs-
34  {                                   // speicher sind,
35    Empfangsbyte = Serial.read();     // werden diese Byte
36                                      // für Byte ausgelesen
37
38    if(Empfangsbyte != Endmarkierung) // Ist der Datensatz zu
39                                      // Ende?
40    {                                 // falls nein:
41      Datenpaket[Zaehler] = Empfangsbyte; // empfangenes Byte
42                                          // im Datensatz speichern
43      Zaehler++;
44      if (Zaehler >= PAKETLAENGE)       // Sicherstellen, dass
45                                        // Zaehler nicht
46        Zaehler = PAKETLAENGE - 1;      // den Maximalwert
47                                        // ueberschreitet.
48    }
49    else
50    {
51      Datenpaket[Zaehler] = '\0';                            // (5)
52      Zaehler = 0;
53      return true;                                           // (6)
54    }
55  }
56  return false;
57 }
58
59 void auswerten()
60 {
61   byte LEDnr, r, g, b = 0;
62                                                             // (7)
63   if(Datenpaket[0] == 'A' && (Datenpaket[12] == 'E' ||
64   Datenpaket[12] == 'N' ))
```

```
65    {
66      LEDnr = (Datenpaket[1]-'0') * 10 + (Datenpaket[2]-'0');// (8)
67      r = (Datenpaket[3]-'0') * 100 + (Datenpaket[4]-'0') * 10 +
68      (Datenpaket[5]-'0');
69      g = (Datenpaket[6]-'0') * 100 + (Datenpaket[7]-'0') * 10 +
70      (Datenpaket[8]-'0');
71      b = (Datenpaket[9]-'0') * 100 + (Datenpaket[10]-'0') * 10
72        + (Datenpaket[11]-'0');
73
74      leds[LEDnr] = CRGB(r, g, b);                          // (9)
75      if(Datenpaket[12] == 'E')                            // (10)
76        FastLED.show();
77    }
78  }
```

1. Die Geschwindigkeit der seriellen Verbindung wurde hier höher gewählt, da deutlich mehr Daten zu übertragen sind. Die bisher übliche geringe Geschwindigkeit (9600) würde zu Wartezeiten führen, wenn der komplette Matrixinhalt übertragen wird. Die auf den ersten Blick willkürlich gewählte Zahl entspricht einer von mehreren möglichen, fest definierten Geschwindigkeitsstufen (300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 oder 115200).

2. In der (sehr kurzen) Hauptschleife wird mittels der selbstdefinierten Funktion `empfangen()` geprüft, ob ein neues Datenpaket empfangen wurde. Ist dies der Fall, schreibt sie dieses in die Zeichenkette `Datenpaket` und gibt `true` zurück, sonst `false`.

3. Wurde ein Paket empfangen, wird es mittels der ebenfalls selbstdefinierten Funktion `auswerten()` verarbeitet.

4. Innerhalb der `empfangen()`-Funktion wird eine `Zaehler`-Variable verwendet, welche die aktuelle Position im Datensatz angibt. Mittels des Schlüsselwortes `static` wird dem Compiler signalisiert, dass diese Variable auch nach dem Verlassen dieser Funktion weiter bestehen und beim nächsten Aufruf noch ihren alten Wert aufweisen soll. Ohne dieses Schlüsselwort würde diese Variable bei jedem Aufruf dieser Funktion auf 0 gesetzt.

5. Ist der Empfang zu Ende, wird die Zeichenkette `Datensatz` mit einem Nullbyte (darstellbar als `\0`) formal korrekt abgeschlossen.

In unserer Anwendung wäre das gar nicht nötig, es gehört aber zu einem guten Programmierstil. Würde man diese Zeichenkette ohne Abschluss per `print()` oder ähnlicher Anweisung ausgeben, könnten das Programm hängenbleiben, da es vergeblich das Ende der Zeichenkette sucht.

6. Das Schlüsselwort `return` markiert den Rückgabewert und sorgt gleichzeitig dafür, dass die Funktion sofort verlassen wird. Die weiter unten stehende Anweisung `return false` kommt also in diesem Fall gar nicht zum Zuge.

7. Die Funktion `auswerten()` prüft zunächst, ob unsere formalen Bestimmungen für ein Datenpaket (Anfang mit „A“, Ende mit „E“ oder „N“) eingehalten wurden. Ist dies nicht der Fall, gab es vermutlich eine Übertragungsstörung und der Datensatz wird ignoriert.

8. Anschließend werden die einzelnen Ziffern extrahiert und in entsprechenden Variablen zwischengespeichert. Die Notation der Form `(Datenpaket[1]-'0')` rührt daher, dass das Array `Datenpaket` den Variablentyp `char` hat. Eine empfangene Ziffer „3“ steht also dort nicht in ihrer Bedeutung als Zahl, sondern wird als Zeichen nach der ASCII-Codetabelle[1] durch eine andere Zahl repräsentiert. So steht hinter einer „3“ der Zahlenwert 51. Um nun den eigentlichen Zahlenwert der Ziffer zu erhalten, kann man einfach den ASCII-Code der Ziffer „0“ (48) subtrahieren.

9. Die empfangenen Farbwerte werden in das Array `leds` übertragen, welches im Arbeitsspeicher alle Pixelwerte der Matrix bereithält.

10. Gemäß unserer Konzeption erfolgt die Übertragung der Daten an die LEDs nur, wenn der Abschluss des Datenpaketes durch ein „E“ erfolgte.

Sie können die Funktion dieses Sketches testen, indem Sie nach dem Hochladen den seriellen Monitor öffnen, die Geschwindigkeit auf 57600 stellen und dann in das Eingabefeld beispielsweise A280002550000E

1 siehe Anhang

eintippen. Nach der Bestätigung mit Enter sollte LED Nr. 28 (etwa in der Mitte der Matrix) in hellem Grün leuchten.

14.5 Programmcode (Processing)

Als nächstes gilt es, den Sketch für die Processing-Software zu entwickeln. Eine günstige Vorgehensweise ist es, zunächst die grafisches-Nutzeroberfläche zu gestalten und sie erst danach mit Funktionalität auszustatten.

Abgeleitet von unserer Skizze (*Abb. 14.2*) könnte so folgender Sketch entstehen. Er enthält nur grafische Anweisungen, welche wir bereits kennengelernt haben:

```
1  void setup() {
2    size(450,560);
3    background(255);
4    textSize(20);
5  }
6
7  void draw()
8  {
9    // Fläche für LED-Matrix
10   stroke(50); fill(150);
11   rect(20, 20, 240, 240);
12
13   // Farbauswahlkästchen am rechten Rand
14   stroke(0);
15   fill(255);                  // weiß
16   rect(380, 20, 29, 29);
17   fill(255,255,0);            // gelb
18   rect(380, 50, 29, 29);
19   fill(0,255,255);            // cyan
20   rect(380, 80, 29, 29);
21   fill(0,255,0);              // grün
22   rect(380, 110, 29, 29);
23   fill(255,0,255);            // pink
24   rect(380, 140, 29, 29);
25   fill(255,0,0);              // rot
26   rect(380, 170, 29, 29);
27   fill(0,0,255);              // blau
28   rect(380, 200, 29, 29);
29   fill(00);                   // schwarz
```

```
30    rect(380, 230, 29, 29);
31
32    // roter Farbregler
33    fill(255,0,0); stroke(255);
34    rect(300,20,5,256);
35    rect(294,200,16,4);
36
37    // grüner Farbregler
38    fill(0,255,0);
39    rect(320,20,5,256);
40    rect(314,200,16,4);
41
42    // blauer Farbregle
43    fill(0,0,255);
44    rect(340,20,5,256);
45    rect(334,200,16,4);
46
47    // Farbvorschaufeld
48    fill(100, 150, 200); stroke(0);
49    rect(290,300,64,50);
50
51    // Textanzeige des aktuellen Farbwertes (vor weißem
52    Hintergrund)
53    fill(255); stroke(255);
54    rect(360,290,70,70);
55    fill(0);
56    text("r: ",  360,312);
57    text("g: ",  360,332);
58    text("b: ",  360,352);
59
60    // Raum für Statusinfos (Text vor weißem Hintergrund)
61    fill(255); stroke(0);
62    rect(20,380,240,85);
63    fill(0);
64    text("Mauszeigerposition:",  25,400);
65    text("Zeile: ",  50,420);
66    text("Spalte: ",  50,440);
67    text("LEDnr: ",  50,460);
68
69    fill(255); rect(20,465,240,75);
70    fill(0);
71    text("letztes Datenpaket:", 25,490);
72
73    // Lösch-Button
74    fill(128);
75    stroke(64);
76    rect(20,300,100,50);
77    fill(0);
```

```
78    text("löschen",  30,330);
79
80    // Zufall-Button
81    fill(128);
82    stroke(64);
83    rect(160,300,100,50);
84    fill(0);
85    text("Zufall",  180,330);
86
87    // Lauftext-Button
88    fill(128);
89    stroke(64);
90    rect(290,380,140,65);
91    fill(0);
92    text("Lauftext",  320,420);
93
94    // Schachbrett-Button
95    fill(128);
96    stroke(64);
97    rect(290,475,140,65);
98    fill(0);
99    text("Schachbrett",  305,515);
100 }
```

Dieser Sketch erzeugt folgende Bildschirmausgabe:

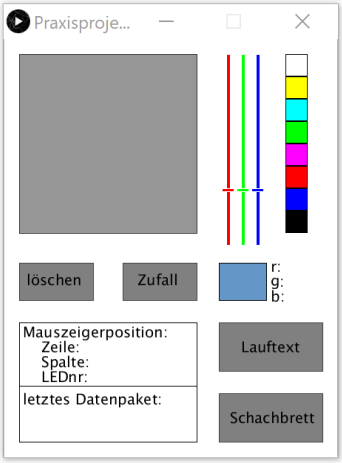


Abb. 14.4 Das Anwendungsfenster verfügt hier noch über keinerlei Funktionalität

Die konkreten Zahlenwerte für die Positionsangaben im Sketch kann man entweder durch mehrmaliges Ausprobieren und Optimieren ermitteln oder man zeichnet die Skizze in einer Grafiksoftware und liest die entsprechenden Positionen dort ab.

Ausgehend von obigem Sketch wurden nun die Funktionalitäten ergänzt. Viele Anweisungen kennen Sie bereits aus dem vorangegangenen Processing-Beispiel, daher wird nachfolgend nur Neues erläutert:

```
import processing.serial.*;

Serial seriellerPort;

int LEDwerte[][][];      // deklariert Matrix zur Speicherung der
                         // Farbwerte;
                         // im setup() muss diese Matrix
                         // zusätzlich
                         // dimensioniert werden.

int Zeile;               // Matrix-Zeile der Mausposition
int Spalte;              // Matrix-Spalte der Mausposition
int LEDnr;               // LED-Nr der Mausposition
int ScrollPosition = 0;  // Hilfsvariablen für Lauftext
long LauftextTimer = 0;

// Variablen für aktuell vom Nutzer vorgewählte Farbe:
int rot = 255; int gruen = 255; int blau = 255;

void setup() {
  size(450,560);
  background(255);
  stroke(160);
  fill(50);
  textSize(20);

  // Größe der Matrix: 8 Pixel Breite x 8 Pixel Höhe x
  // 3 Farbwerte
  // Notation erzeugt neue Matrix des angegebenen Typs.
  LEDwerte = new int[8][8][3];

  // zu Beginn alle Werte der Matrix auf 0 setzen:
  for(byte x = 0; x < 8; x++)
  {
      for(byte y = 0; y < 8; y++)
      {
```

```
        LEDwerte[x][y][0] = 0;
        LEDwerte[x][y][1] = 0;
        LEDwerte[x][y][2] = 0;
      }
  }

  printArray(Serial.list());
  String portName = Serial.list()[0];  // gegebenenfalls ändern,
                                       // wenn
                                       // anderer Port genutzt
                                       // wird
  seriellerPort = new Serial(this, portName, 57600);

  delay(500);               // kurz warten, bis serieller Port
                            // aktiviert wurde

  allesloeschen();  // alle LEDs auf schwarz setzen und an Ardui
                    // no senden

  // Farbauswahlkästchen am rechten Rand zeichnen
  // (einmalig, diese werden nie überschrieben)
  stroke(0);
  fill(255);                // weiß
  rect(380, 20, 29, 29);
  fill(255,255,0);          // gelb
  rect(380, 50, 29, 29);
  fill(0,255,255);          // cyan
  rect(380, 80, 29, 29);
  fill(0,255,0);            // grün
  rect(380, 110, 29, 29);
  fill(255,0,255);          // pink
  rect(380, 140, 29, 29);
  fill(255,0,0);            // rot
  rect(380, 170, 29, 29);
  fill(0,0,255);            // blau
  rect(380, 200, 29, 29);
  fill(00);                 // schwarz
  rect(380, 230, 29, 29);
}


void draw()
{
  if(mouseX >= 20 && mouseX < 260 && mouseY >= 20 &&
  mouseY < 260)
  {
                            // Falls sich die Maus auf
```

```
                                    // der LED-Matrix befindet:
  Spalte = (mouseX-20) / 30;        // aktuelle Spalte berechnen
  Zeile =  (mouseY-20) / 30;        // aktuelle Zeile berechnen
  LEDnr = XYtoNR(Spalte, Zeile);    // und daraus LED-Nr
                                    // berechnen
  if(mousePressed)                  // Wird gerade eine Taste
                                    // gedrückt gehalten?
  {
    if(mouseButton == LEFT)         // linke Taste: Feld
                                    // einfärben
      faerbe(Spalte, Zeile, rot, gruen, blau);
    else                            // rechte Taste: Feld
                                    // schwärzen
      faerbe(Spalte, Zeile, 0, 0, 0);
  }
}
else
{                                   // Falls sich die Maus nicht
                                    // auf der Matrix befindet,
  Zeile = Spalte = LEDnr = -1;      // werden die zugehörigen
                                    // Variablen
}                                   // nicht benötigt.


if(mousePressed)  // bei gedrückter Maustaste
{                 // prüfen, ob sie über einem Farbregler steht
                  // und gegebenenfalls Farbwert übernehmen
  if(mouseX >= 294 && mouseX <= 310 && mouseY >= 20 &&
  mouseY <= 275)
    rot = 255-(mouseY-20);
  if(mouseX >= 314 && mouseX <= 330 && mouseY >= 20 &&
  mouseY <= 275)
    gruen = 255-(mouseY-20);
  if(mouseX >= 334 && mouseX <= 350 && mouseY >= 20 &&
  mouseY <= 275)
    blau = 255-(mouseY-20);
}


// Farbauswahlregler mit weißem Rechteck überdecken
fill(255); stroke(255);
rect(293,15,60,280);

// und nun neu zeichnen:
fill(255,0,0);
rect(300,20,5,256);
rect(294,274-rot,16,4);

```

```
132   fill(0,255,0);
133   rect(320,20,5,256);
134   rect(314,274-gruen,16,4);
135
136   fill(0,0,255);
137   rect(340,20,5,256);
138   rect(334,274-blau,16,4);
139   // Farbvorschaufeld zeichnen
140   fill(rot, gruen, blau); stroke(0);
141   rect(290,300,64,50);
142
143   // Farbwertanzeige mit weißem Rechteck überdecken
144   fill(255); stroke(255);
145   rect(360,290,70,70);
146   fill(0);
147   text("r: " + rot, 360,312);   // Farbwerte anzeigen
148   text("g: " + gruen, 360,332); // zur Kontrolle für den
149   text("b: " + blau, 360,352);  // Nutzer
150
151   // Berechnete LED-Position anzeigen
152   fill(255); stroke(0);
153   rect(20,380,240,85);
154   fill(0);
155   text("Mauszeigerposition:",  25,400);
156   text("Zeile: " + Zeile, 50,420);
157   text("Spalte: " + Spalte, 50,440);
158   text("LEDnr: " + LEDnr, 50,460);
159
160   // Schaltflächen zeichnen:
161   if(MausueberLoeschButton())  // befindet sich die Maus auf der
162                                // Schaltfläche?
163     fill(160);                 // wenn ja: hellere
164                                // Hintergrundfarbe nutzen
165   else
166     fill(128);
167   stroke(64);
168   rect(20,300,100,50);
169   fill(0);
170   text("löschen",  30,330);
171
172   if(MausueberZufallButton())
173     fill(160);
174   else
175     fill(128);
176   stroke(64);
177   rect(160,300,100,50);
178   fill(0);
179   text("Zufall",  180,330);
```

```
180
181
182   if(MausueberLTButton())
183     fill(160);
184   else
185     fill(128);
186   stroke(64);
187   rect(290,380,140,65);
188   fill(0);
189   text("Lauftext",  320,420);
190
191   if(MausueberSchachbrettButton())
192     fill(160);
193   else
194     fill(128);
195   stroke(64);
196   rect(290,475,140,65);
197   fill(0);
198   text("Schachbrett",  305,515);
199
200
201   // Falls Lauftextmodus aktiv ist, die entsprechende Funktion
202   // nach Ablauf ihres Timers aufrufen:
203   if(LauftextTimer > 0 && millis() > LauftextTimer)
204     Lauftext();
205 }
206
207
208 // Wenn die Maustaste gedrückt wurde, wird die folgende
209 // Funktion automatisch einmalig aufgerufen:
210
211 void mousePressed()
212 {
213   // Lauftext anhalten, falls aktiv:
214   LauftextTimer = 0;
215
216   // Prüfen, ob die Maus gerade auf einem Farbauswahlkästchen
217   // steht:
218   if(mouseX >= 380 && mouseX < 409 && mouseY >= 20 && mouseY < 50)
219   {  // weiß
220     rot = 255; gruen = 255; blau = 255;
221   }
222   if(mouseX >= 380 && mouseX < 409 && mouseY >= 50 && mouseY < 80)
223   {  // gelb
224     rot = 255; gruen = 255; blau = 0;
225   }
226   if(mouseX >= 380 && mouseX < 409 && mouseY >= 80 && mouseY < 110)
227   {  // cyan
```

```
228       rot = 0; gruen = 255; blau = 255;
229     }
230     if(mouseX >= 380 && mouseX < 409 && mouseY >= 110 &&
231     mouseY < 140)
232     {  // grün
233       rot = 0; gruen = 255; blau = 0;
234     }
235     if(mouseX >= 380 && mouseX < 409 && mouseY >= 140 &&
236     mouseY < 170)
237     {  // magenta
238       rot = 255; gruen = 0; blau = 255;
239     }
240     if(mouseX >= 380 && mouseX < 409 && mouseY >= 170 &&
241     mouseY < 200)
242     {  // rot
243       rot = 255; gruen = 0; blau = 0;
244     }
245     if(mouseX >= 380 && mouseX < 409 && mouseY >= 200 &&
246     mouseY < 230)
247     {  // blau
248       rot = 0; gruen = 0; blau = 255;
249     }
250     if(mouseX >= 380 && mouseX < 409 && mouseY >= 230 &&
251     mouseY < 260)
252     {  // schwarz
253       rot = 0; gruen = 0; blau = 0;
254     }
255
256
257     // prüfen, ob die Maus auf einer Schaltfläche steht:
258
259     if(MausueberLoeschButton())
260       allesloeschen();
261
262     if(MausueberZufallButton())
263       Zufallsbild();
264
265     if(MausueberSchachbrettButton())
266       Schachbrett();
267
268     if(MausueberLTButton())
269       LauftextTimer = millis();
270   }
271
272
273   // Die folgenden Funktionen testen lediglich, ob sich die Maus
274   // gerade auf einer bestimmten Schaltfläche befindet
275   // und geben entsprechend true oder false zurück.
```

```
// Die Zahlenwerte wurden aus den Koordinaten der
// Zeichnungsbefehle
// abgeleitet.

boolean MausueberLoeschButton()
{
  if(mouseX >= 20 && mouseX <= 120 && mouseY >= 300 &&
  mouseY <= 350)
    return true;
  else
    return false;
}

boolean MausueberZufallButton()
{
  if(mouseX >= 160 && mouseX <= 260 && mouseY >= 300 &&
  mouseY <= 350)
    return true;
  else
    return false;
}

boolean MausueberLTButton()
{
  if(mouseX >= 290 && mouseX <= 430 && mouseY >= 380 &&
  mouseY <= 445)
    return true;
  else
    return false;
}

boolean MausueberSchachbrettButton()
{
  if(mouseX >= 290 && mouseX <= 430 && mouseY >= 475 &&
  mouseY <= 540)
    return true;
  else
    return false;
}


// Die folgende Funktion berechnet aus einer gegebenen
// Spalten- (x)
// und Zeilennummer (y) die Adresse einer LED
// entsprechend des Layouts der 8x8-LED-Matrix
int XYtoNR(int x, int y)
{
  if(y % 2 == 1)        // ungerade Zeile?
```

```
    return y * 8 + (7-x);
  else
    return y * 8 + x;
}


// Die folgende Funktion färbt das Matrix-Pixel
// an der übergebenen Position mit den übergebenen
// Farbwerten und sendet diese Information direkt
// an den Arduino.

void faerbe(int x, int y, int r, int g, int b)
{
  fill(r,g,b); stroke(0);
  rect(20+x*30, 20+y*30, 29, 29);
  LEDwerte[x][y][0] = r;
  LEDwerte[x][y][1] = g;
  LEDwerte[x][y][2] = b;
  sende_aktuelle_LED();
  delay(10);
}


// Die folgende Funktion sendet nur die Information über
// das aktuelle
// Pixel an den Arduino.

void sende_aktuelle_LED()
{
  String Datenpaket;
  Datenpaket = "A" +nf(LEDnr,2) + nf(rot,3) + nf(gruen,3) +
  nf(blau,3) + "E\n";
      // nf() formatiert die übergebene Zahl auf die gewünschte
      // Stellenanzahl, gegebenenfalls mit führenden Nullen.

  // Statusfeld mit weißem Rechteck überdecken und neu schreiben:
  fill(255); rect(20,465,240,75);
  fill(0);
  text("letztes Datenpaket:\n"+Datenpaket, 25,490);
  print(Datenpaket);    // Daten zur Veranschaulichung auch in
                        // der Konsole ausgeben

  seriellerPort.write(Datenpaket);
}


// Die folgende Funktion sendet alle 64 Datensätze
// an den Arduino.
```

```
372
373 void sende_alle_LEDs()
374 {
375   String Datenpaket = "";
376   String Endzeichen = "N\n";
377
378   for(byte x = 0; x < 8; x++)
379   {
380       for(byte y = 0; y < 8; y++)
381       {
382         if(x == 7 && y == 7)        // letztes Datenpaket?
383           Endzeichen = "E\n";       // falls ja: Endzeichen "E"
384                                     // statt "N"
385         Datenpaket = "A"
386                     + nf(XYtoNR(x,y),2)
387                     + nf(LEDwerte[x][y][0],3)
388                     + nf(LEDwerte[x][y][1],3)
389                     + nf(LEDwerte[x][y][2],3)
390                     + Endzeichen;
391
392         print(Datenpaket);      // Datenpaket zur Kontrolle auch
393                                 // auf der Konsole ausgeben
394         seriellerPort.write(Datenpaket);
395       }
396   }
397
398   // Statusfeld mit weißem Rechteck überdecken und neu schreiben:
399   fill(255); rect(20,465,240,75);
400   fill(0);
401   text("letztes Datenpaket:\n"+Datenpaket, 25,490);
402
403 }
404
405
406 void allesloeschen()
407 {
408   // schwärzt die Matrix-Fläche:
409   stroke(0);fill(0);
410   rect(20,20,240,240);
411
412   for(byte j = 0; j < 8; j++)
413   {
414       for(byte k = 0; k < 8; k++)
415       {
416         for(byte l = 0; l < 3; l++)
417           LEDwerte[j][k][l] = 0;
418       }
419   }
```

```
  sende_alle_LEDs();
  rot = gruen = blau = 255;  // weiß vorwählen
  LauftextTimer = 0;         // Lauftext anhalten, falls aktiv
}


// Die folgende Funktion erzeugt einen grauen
// Lauftext mit dem Schriftzug "Arduino".

void Lauftext()
{
  int Zeiger = 0;
  int Helligkeit = 0;
  String Vorlage[] = {
    " XXX                                              ",
    "X   X            X          X                     ",
    "X   X            X                                ",
    "XXXXX  X XX   XXXX  X   X  XX   XXXX    XXX      ",
    "X   X  XX    X   X  X   X   X   X   X  X   X     ",
    "X   X  X     X   X  X   X   X   X   X  X   X     ",
    "X   X  X      XXXX   XXXX  XXX  X   X   XXX      ",
    "                                                  "};

  // Matrixfläche schwärzen
  stroke(0);fill(0);
  rect(20,20,240,240);


  // alle Pixel durchlaufen, um aktuelles Scroll-Bild zu
  // erzeugen:
  for(byte j = 0; j < 8; j++)
  {
      for(byte k = 0; k < 8; k++)
      {
        Zeiger = j + ScrollPosition;
        if(Zeiger > 46)
          Zeiger -= 47;

        // Befindet sich in der Vorlage an entsprechender
        // Stelle ein X?
        if(Vorlage[k].charAt(Zeiger) == 'X')
          Helligkeit = 128;    // Falls ja, Helligkeit zuweisen
        else
          Helligkeit = 0;      // sonst schwarz.

        // Alle drei Farbwerte setzen:
        for(byte l = 0; l < 3; l++)
          LEDwerte[j][k][l] = Helligkeit;

        // Pixel in die Matrix zeichnen
        fill(Helligkeit);
        rect(20+j*30, 20+k*30, 29, 29);
      }
```

```
  }

  sende_alle_LEDs();

  ScrollPosition++;  // Scrollposition weiterrücken
  if(ScrollPosition > 47)
    ScrollPosition = 0;

  // nächste Aktualisierung in 200 Millisekunden
  LauftextTimer = millis() + 200;
}

void Zufallsbild()
{
  stroke(0);
  for(byte j = 0; j < 8; j++)
  {
      for(byte k = 0; k < 8; k++)
      {
        for(byte l = 0; l < 3; l++)
          LEDwerte[j][k][l] = (int)random(256);

        fill(LEDwerte[j][k][0],LEDwerte[j][k][1],LEDwerte[j][k][2]);
        rect(20+j*30, 20+k*30, 29, 29);
      }
  }
  sende_alle_LEDs();
}


void Schachbrett()
{
  stroke(0);
  for(byte j = 0; j < 8; j++)
  {
      for(byte k = 0; k < 8; k++)
      {
        for(byte l = 0; l < 3; l++)
          LEDwerte[j][k][l] = (XYtoNR(j,k)%2) * 255;
                    // jede zweite LED weiß,
                    // die anderen schwarz

        fill(LEDwerte[j][k][0]);
        rect(20+j*30, 20+k*30, 29, 29);
      }
  }
  sende_alle_LEDs();
}
```

Sicher fragen Sie sich, worin der Unterschied zwischen der Variablen `mousePressed` und der Funktion `mousePressed()` liegt. Letztere ist

ein Ereignis, sie wird beim Drücken einer Maustaste nur einmalig durchlaufen. Die Variable hingegen ist ein Zustand, sie kann ständig und beliebig oft abgefragt werden. Daher ordnen wir beispielsweise das Erzeugen des Zufallsmusters der Funktion `mousePressed()` zu, schließlich wollen wir dieses nur einmal erzeugen, auch wenn die Taste länger gedrückt gehalten wird. Das Verschieben eines Reglers realisieren wir hingegen besser über die Abfrage von `mousePressed`, denn dadurch können wir so lange reagieren, wie die Maustaste gedrückt wird.

Des Weiteren haben Sie bemerkt, dass an einigen Stellen die bisherige Anzeige von einem weißen Rechteck überdeckt und dann neu gezeichnet wird. Dies ist immer dann notwendig, wenn sich Inhalte ändern – zum Beispiel die Position der Regler am Farbwähler. Würde man das weiße Rechteck weglassen, wären auch alle früheren Positionen des Reglers noch darunter sichtbar.

14.6 Resultat

Beim direkten Blick auf die LED-Matrix werden Sie aufgrund der vielen kleinen punktförmigen Lichtquellen eine starke Blendwirkung feststellen.

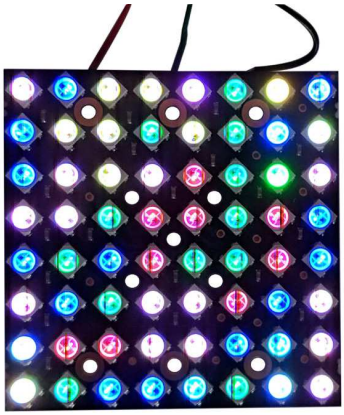
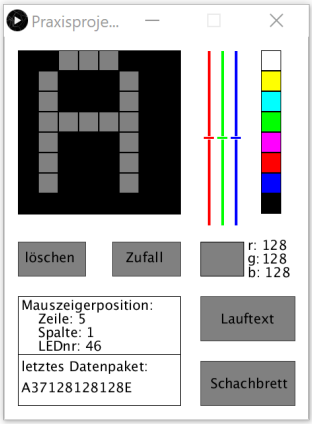
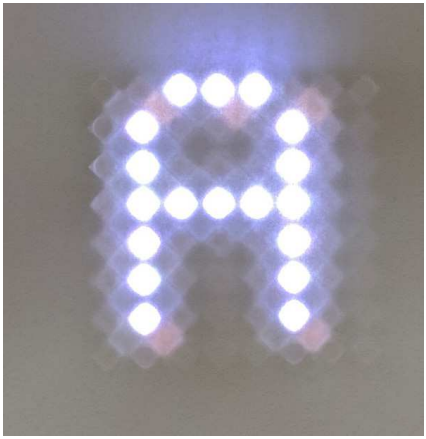
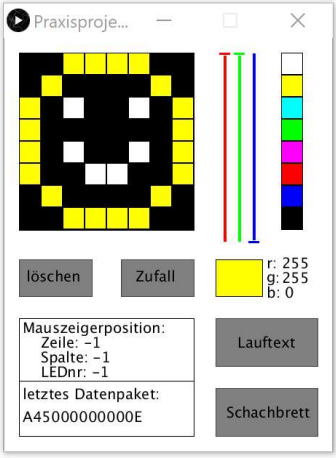


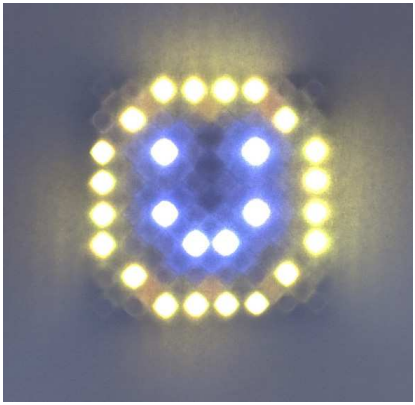
Abb. 14.5 Die einzelnen Lichtpunkte der LED-Matrix entwickeln schnell eine Blendwirkung

Es empfiehlt sich daher, einen Diffusor vor der Matrix anzubringen. Testweise genügt dafür ein Blatt Papier, wie in den folgenden Beispielen. Für einen dauerhaften Einsatz könnte man entsprechend diffuse Kunststoffe oder Milchglas verwenden.









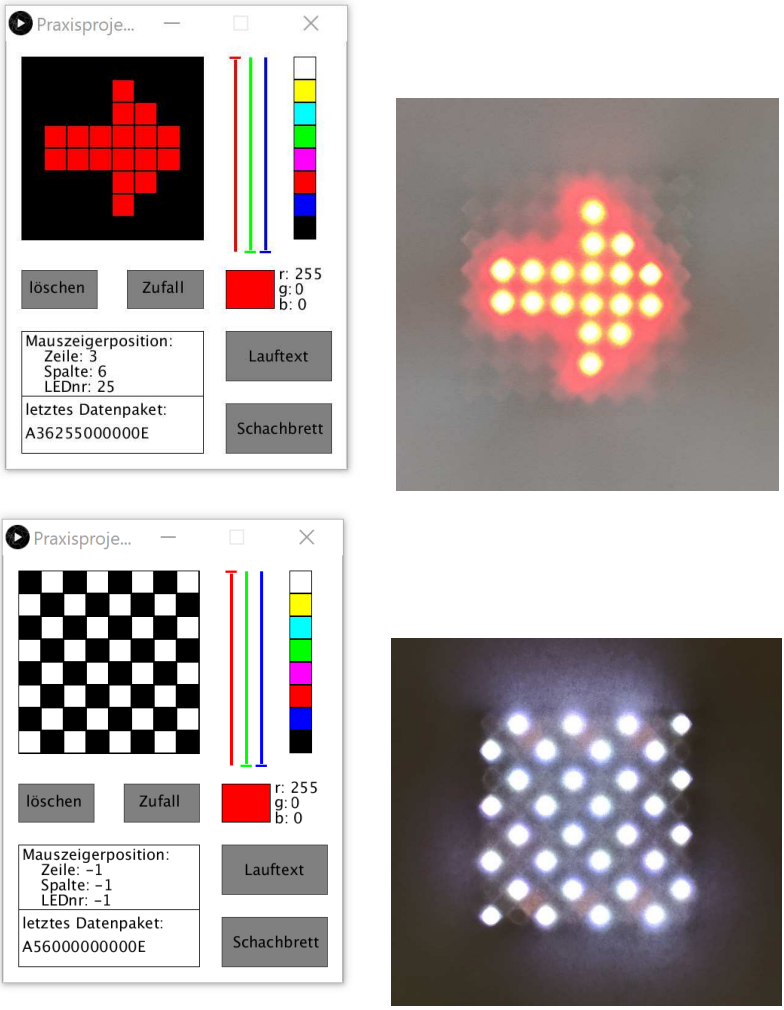


Abb. 14.6

Die Processing-Plattform bietet sogar die Möglichkeit, derartige Projekte zu exportieren, so dass sie auch auf anderen Computern ausgeführt werden können. Ein vorheriges Herunterladen der Processing-Software ist dafür auf dem anderen Gerät nicht notwendig.

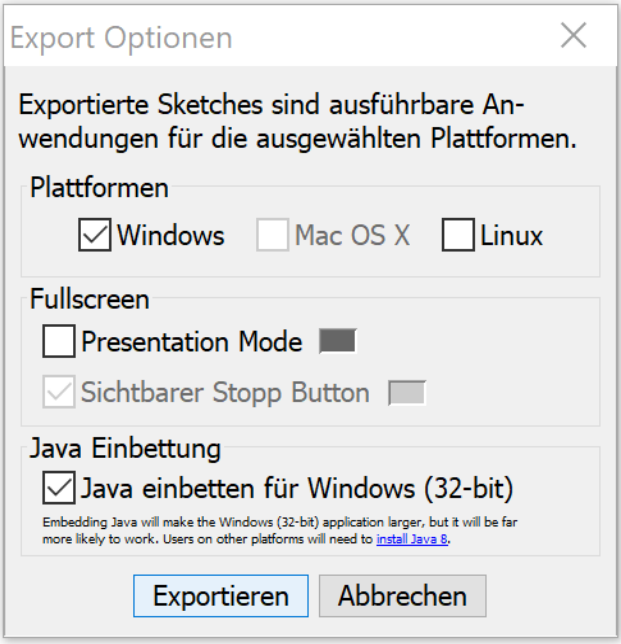


Abb. 14.7 Das Export-Menü öffnen Sie über Datei -> Exportieren

Sie sollten jedoch die Java-Einbettung aktivieren, denn dann können Sie sicher sein, dass das Programm genau so funktioniert wie auf Ihrem Computer. Andernfalls könnte eine abweichende oder fehlende Java-Plattform auf dem fremden System die Ausführung behindern.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 15
Objektorientierte Programmierung

Bereits in Kapitel 4.12 haben wir einen ersten Blick auf die objektorientierte Programmierweise geworfen. Die dortige kurze Einführung diente jedoch lediglich dem besseren Verständnis der späteren Programmbeispiele mit Bibliotheken, welche oft mit Objekten arbeiten.

Mittlerweile sind Ihre Programmierkenntnisse deutlich fortgeschritten und die Projekte werden schnell komplexer, so dass es sich lohnt, in diesem Kapitel nun auch selbst die Grundlagen der objektorientierten Programmierung (oft kurz als OOP bezeichnet) zu erlernen. Abschließend werden wir sogar unsere erste eigene Bibliothek erstellen.

15.1 Gründe für OOP

Grundlegend gilt: Alles, was mit objektorientierter Programmierung realisierbar ist, kann alternativ auch auf herkömmliche Weise programmiert werden, und umgekehrt. Wieso hat sich also dieser zusätzliche Programmierstil entwickelt?

Wie bereits in Kapitel 4.12 dargestellt, besteht die Grundidee der OOP darin, Strukturen im Quellcode näher an der Realität anzulehnen. Das steigert die Übersichtlichkeit, vermeidet Fehler und sorgt dafür, dass Sie geschriebenen Programmcode viel einfacher für spätere Projekte wiederverwenden können.

Nehmen wir dazu zunächst ein Beispiel aus der realen Welt, jenseits der Programmierung: Stellen Sie sich vor, Sie möchten für Ihre Kinder eine kleine Spielzeug-Rallye bauen. Im einfachsten Fall werden Sie die klein-

en Autos einzeln basteln. Jedes mit einem kleinen Motor, einem Antrieb, einer Steuerung und natürlich darf auch die Beleuchtung nicht fehlen. Das ist viel Aufwand, aber bei einer Kleinserie dennoch irgendwie zu bewältigen.

Wenn Ihnen allerdings klar wird, dass Sie womöglich als Erweiterung oder für Freunde später noch weitere Spielzeugautos bauen wollen, lohnt es sich, über Rationalisierungen nachzudenken. Vielleicht entwerfen Sie eine Bastelvorlage mit einer Stanzschablone, einem fertigen Verdrahtungsplan und einer genauen Bauanleitung. Natürlich müssen Sie dafür zusätzliche Zeit investieren – aber die späteren „Neubauten" gelingen Ihnen dadurch viel schneller und mit weniger Fehlern.

Eine ganz ähnliche Idee verfolgt die objektorientierte Programmierung: Strukturen, die häufig wiederkehrend vorkommen, werden abstrakt als sogenannte *Klassen* definiert (wie die Bauanleitung) und können dann genutzt werden, um in einem konkreten Programm *Objekte* (also Spielzeugautos) zu erzeugen. Diese Objekte (auch *Instanzen* genannt) können verschiedene Eigenschaften besitzen und sogar Funktionen beinhalten. Die Rallye-Autos haben schließlich auch verschiedene Farben und können diverse Aktionen ausführen.

15.2 Klassen und Objekte

Allgemein gilt: Eine *Klasse* ist eine Vorlage, aus der mehrere *Instanzen* dieser Klasse (*Objekte*) erzeugt werden können. Im Quellcode werden dazu nicht die Objekte selbst, sondern nur die Klassen gleichartiger Objekte definiert.

Um diesen elementaren Zusammenhang besser zu verstehen, betrachten wir ein weiteres Beispiel, diesmal aus der Welt des virtuellen Geldes: Für ein Brettspiel mit Spielgeldkonten möchten wir eine kleine Kontoverwaltungs-Software kreieren. Hier könnte es eine Klasse `SpielKonto` geben, deren Instanzen Konten der einzelnen Mitspieler sind. Im Quelltext werden diese Objekte dann ganz ähnlich erstellt wie eine Variablendefinition:

```
...
SpielKonto KontoA;
SpielKonto KontoB;
SpielKonto KontoC;
...
```

Ob bei dieser Erstellung bereits Werte (Argumente) übergeben werden können oder müssen, hängt von der Klassendefinition ab. In diesem ersten Beispiel wollen wir darauf noch verzichten.

Im späteren Spielverlauf könnten wir diese Konten nun mittels Funktionen, welche wir in der Klassendefinition anlegen, manipulieren (man nenn derartige Funktionen, welche den inneren Zustand von Objekten verändern, auch *Methoden*) oder ihre Eigenschaften auslesen:

```
...
KontoA.zahleAus(200);
KontoB.zahleEin(200);
KontoC.verzinse();
Serial.print("Kontostand Spieler C:");
Serial.println(KontoC.gibKontostand());
...
```

15.3 Klassendefinition

Bevor Objekte als Instanzen einer Klasse erstellt werden können, muss die Klasse definiert werden. Dies erfolgt vorzugsweise zu Beginn eines Sketches, eingeleitet durch das Schlüsselwort `class`. Danach folgt der gewünschte Name der Klasse sowie die eigentliche Klassendefinition in geschweiften Klammern, analog zu der bereits bekannten Definition von Funktionen.


```
class Beispielklasse {
    private:
      long Variable1;
      byte FunktionA(void);
      byte Variable2;
      ...
    public:
      byte FunktionB(int);
      void FunktionC(int);
    ...
};
...
```

Innerhalb dieser Klammern wird nun aufgelistet, aus welchen Variablen (oder auch Konstanten) und Funktionen ein derartiger „Objekt-Prototyp“ besteht. Hierbei liegt das Augenmerk auf dem zu reservierenden Speicher, daher sind stets alle Datentypen (von Variablen und Funktionen inklusive deren Argumenten und Rückgabewerten) anzugeben. Konkrete Funktionsdefinitionen, welche etwas mehr Platz im Quelltext benötigen, können später auch noch außerhalb der Klassendefinition platziert werden – aber nur, wenn sie innerhalb der geschweiften Klammern bereits angekündigt wurden. Aus diesem Grund folgen auf die Funktionsnamen in obigem Beispiel keine geschweiften Klammern – wir unterstellen hier, dass die Definition später erfolgt.

Als Besonderheit einer Klassendefinition fällt die Zweiteilung ins Auge: Es gibt stets einen `private`- und einen `public`-Bereich. Wie die Bezeichnungen schon vermuten lassen, handelt es sich bei ersterem um Interna. Das heißt, jede Instanz dieser Klasse wird diese Werte und Funktionen ins sich tragen – es kann aber nicht von außen direkt darauf zugegriffen werden. Anders ist dies beim `public`-Bereich – diese Variablen und Funktionen stehen quasi als Schnittstelle mit der Außenwelt zur Verfügung.

In der professionellen Programmierung ist man bestrebt, üblicherweise alle Variablen als `private` zu definieren und deren äußeren Zugriff lediglich indirekt über speziell dafür angelegte Funktionen zu erlauben, man spricht in diesem Kontext auch von *Schnittstellenfunktionen*. Innerhalb dieser Funktionen kann dann geprüft werden, ob der Zugriff überhaupt berechtigt ist und ob eine beabsichtigte Veränderung einer Variablen überhaupt plausibel ist.

Verständlicher wird dies, wenn wir uns wieder unserer Spielgeld-Bank widmen. Eine Klassendefinition für die Spielerkonten könnte wie folgt lauten:

```
1  class SpielKonto {
2   private:
3     long Kontostand;                                    // (1)
4     float Zinssatz;
5   public:
```

```
6     long gibKontostand(void) {return Kontostand;};          // (2)
7     boolean zahleEin(long);                                 // (3)
8     boolean zahleAus(long);
9     boolean verzinse(void)
10      {Kontostand+=Kontostand*Zinssatz/100.0; return true;};// (4)
11    boolean setzeZinssatz(float);
12    SpielKonto(void);                                       // (5)
13 };
14 ...
```

Hier wird die wichtigste Eigenschaft eines Kontos, der Kontostand, ebenfalls bewusst als `private`-Variable geschützt. Stünde sie im `public`-Bereich, so könnte der Wert einfach beispielsweise als `KontoA.Kontostand` ausgelesen und manipuliert werden. Allerdings gäbe es dann auch keinen Schutz vor unbeabsichtigten Änderungen. Um genau dies zu verhindern, werden in unserem Beispiel mehrere `public`-Funktionen vorgesehen (deren Definitionen hier teilweise noch nicht sichtbar sind), welche geregelten Zugriff auf den Kontostand erlauben.

(1) Hier wird der für die internen Variablen nötige Speicherplatz reserviert. Eine sofortige Anfangswertzuweisung ist zwar möglich, aber eher unüblich. Wir werden zu diesem Zweck stattdessen in Kürze die Konstruktorfunktion kennenlernen. Für den Kontostand sollen uns ganzzahlige Beträge genügen.

(2) Diese Funktion wollen wir vorsehen, um später den Kontostand auslesen zu können. Da sie lediglich auf die interne Variable Kontostand zugreifen und diese als Rückgabewert liefern soll, ist ihre Definition sehr kurz. In diesem Fall schreibt man sie üblicherweise direkt in die Klassendefinition.

(3) In den Funktionen für Ein- und Auszahlungen wollen wir Maximalbeträge prüfen. Sie haben daher geringfügig mehr Quelltext. Zur besseren Übersichtlichkeit kündigen wir sie hier nur an und definieren sie etwas weiter unten im Sketch (im nächsten Codeauszug). Von ihren Argumenten müssen wir nur den Datentyp, nicht jedoch den Namen ankündigen.

(4) Auch die Funktion zur Verzinsung ist kurz genug, um direkt notiert zu werden.

(5) Diese Funktion ist eine Besonderheit, sie wird auch *Konstruktor* genannt. Sie trägt stets den exakten Namen der Klasse und hat grundsätzlich keinerlei Rückgabewert (auch nicht `void`). Werden Instanzen der Klasse erstellt (wenn die einzelnen Konten angelegt werden), wird die Konstruktor-Funktion sofort nach der Erstellung jedes Objekts für dieses Objekt einmalig automatisch aufgerufen. Wir können uns dies zu Nutze machen, um beispielsweise den internen Variablen entsprechende Anfangswerte zuzuweisen.

Es sei erwähnt, dass es in Analogie dazu auch die Möglichkeit einer Destruktor-Funktion gibt. Ihr Name beginnt mit einer Tilde (~), ansonsten gelten die gleichen Regeln wie für die Konstruktor-Funktion. Allerdings kann sie keinerlei Argumente annehmen. Sie wird einmalig durchlaufen, wenn ein Objekt aufhört zu existieren (dies kann durch explizite Anweisungen geschehen). Für die Arduino-Welt sind Destruktor-Funktionen jedoch nahezu irrelevant, da hier Objekte üblicherweise während der kompletten Programmlaufzeit existieren.

Schauen wir uns nun noch den weiteren Sketch mit den Funktionsdefinitionen an.

```
...
boolean SpielKonto::zahleEin(long Betrag)
{
  if(Betrag > 0 && Betrag <= 10000)        // Plausibilität prüfen
  {
    Kontostand+= Betrag;
    return true;                    // Bei Erfolg: true zurückgeben
  }
  else
    return false;                    // Fehler durch false-Rückgabe
}                                                  // kennzeichnen

boolean SpielKonto::zahleAus(long Betrag)
{
  if(Kontostand >= Betrag && Betrag <= 10000)
  {
```

```
17     Kontostand-= Betrag;
18     return true;
19    }
20    else
21     return false;
22  }
23
24  boolean SpielKonto::setzeZinssatz(float neuerSatz)
25  {
26     if(neuerSatz >= 0 && neuerSatz <= 10)
27     {
28       Zinssatz = neuerSatz;
29       return true;
30     }
31     else
32       return false;
33  }
34  SpielKonto::SpielKonto(void)                          // Konstruktor
35  {
36     Kontostand = 5000;
37     Zinssatz = 2.0;
38  }
39  ...
```

Mit Ihren bisherigen Kenntnissen von Funktionen ist es problemlos möglich, diese Beispiele nachzuvollziehen. Hauptsächlich wird hier geprüft, ob übergebene Werte plausibel und gewünschte Aktionen überhaupt möglich sind. Bei der Auszahlung muss der Kontostand beispielsweise größer als der Auszahlbetrag sein, bei den Zinssätzen sind keine negativen Werte möglich und so weiter.

Zu einem guten Programmierstil gehört es, bei Funktionen stets einen Rückgabewert vorzusehen – im einfachsten Falle `true`, wenn die Anweisung normal abgearbeitet werden konnte; beziehungsweise `false`, wenn es ein Problem (zum Beispiel ungenügende Kontodeckung) gab.

Ungewohnt dürfte allerdings der hier erstmals verwendete Gültigkeitsoperator (::) sein. Seine Verwendung rührt daher, dass wir die betreffenden Funktionen außerhalb der Klasse definieren und dem Compiler mitteilen müssen, dass sie eigentlich in die entsprechende Klassendefinition gehören.

Womöglich fragen Sie sich, warum dafür nicht der bereits bekannte Punkt als Trennung von Objektname und zugehörigem Element verwendet wird. Hier müssen wir sehr genau auf die Abgrenzung achten. In obigem Beispiel handelt es sich um eine *Klassen*definition, wir schreiben also gerade den „Bauplan“ für unsere späteren Objekte. `SpielKonto` ist somit der Name einer Klasse. Bei einem späteren Zugriff wie beispielsweise `KontoA.zahleAus(100)` handelt es sich um das konkrete Objekt `KontoA`, welches also nur eine *Instanz* der Klasse `SpielKonto` ist. Nur in diesem Fall, also beim Zugriff auf ein konkretes *Objekt*, verwendet man den Punkt.

Unsere Klassendefinition ist nun bereits komplett. Mit einigen weiteren Codezeilen lässt sich die simple Spielgeld-Bank testen:

```
...
SpielKonto KontoA;
SpielKonto KontoB;
SpielKonto KontoC;

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

void loop()
{
  Serial.print("Kontostand A: ");
  Serial.println(KontoA.gibKontostand());
  Serial.print("Kontostand B: ");
  Serial.println(KontoB.gibKontostand());
  Serial.print("Kontostand C: ");
  Serial.println(KontoC.gibKontostand());
  Serial.println("----------------------------");

  if(KontoA.zahleAus(1000))  // wenn KontoA genügend Deckung hat,
    KontoB.zahleEin(1000);   // werden 1000 Geldeinheiten
                             // transferiert

  KontoA.verzinse();
  KontoB.verzinse();
  KontoC.verzinse();

  delay(10000);
}
```

Wenn wir diesen Codeausschnitt zusammen mit den beiden vorangegangenen zu einem Sketch zusammenfügen und hochladen, können wir bereits unsere erste selbsterstellte Klasse am seriellen Monitor in Aktion sehen. Von Konto A werden so lange 1000 Geldeinheiten auf Konto B gebucht, bis die Deckung von A nicht mehr ausreicht. Zudem werden alle Konten bei jedem Durchlauf verzinst und der Kontostawird ausgegeben:

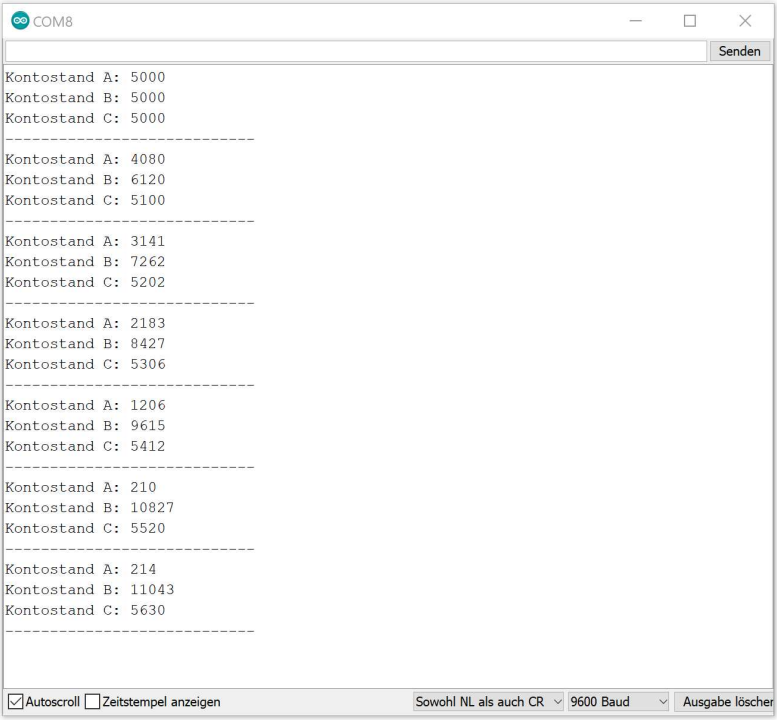


Abb. 15.1 beispielhafte Ausgabe am seriellen Monitor

An diesem einfachen Beispiel ist erkennbar, warum interne Variablen üblicherweise als `private` definiert werden: Über die speziell darauf zugeschnittenen Funktionen ist ein eleganter, zweckbezogener Zugriff auf die Werte möglich und es ist sehr unwahrscheinlich, versehentlich

beispielsweise einen Kontostand zu überschreiben. Wären die Werte hingegen `public`, so könnte man problemlos in der Hauptschleife auf `KontoA.Kontostand` zugreifen und ihn einfacher auslesen, allerdings auch ungehindert überschreiben oder versehentlich auf 0 setzen. Je größer ein Projekt wird (und je mehr möglicherweise auch fremde Programmteile von anderen Autoren zusammenwirken), umso stärker machen sich die Vorteile „abgeschotteter" Klassendefinitionen (mit möglichst wenig Möglichkeiten, von außen in unbeabsichtigter Weise auf innere Zustände einzuwirken) bemerkbar.

15.4 Beispiel: Schrittmotorsteuerung

Wir wollen die objektorientierte Programmierung nun auch an einem konkreten Anwendungsfall aus der Arduino-Welt nachvollziehen. Bitte blättern Sie einmal kurz zurück in das Kapitel 11.4. Wir haben dort den Schrittmotor kennengelernt und für seine Ansteuerung die Bibliothek Stepper.h genutzt. Im Quelltext erzeugten wir dafür das Objekt `Schrittmotor` als Instanz der Klasse `Stepper`, welche (für uns unsichtbar) innerhalb der Bibliothek definiert wurde.

```
1 ...
2 Stepper Schrittmotor(SCHRITTEPROUMDREHUNG, 8, 10, 9, 11);
3 ...
```

Im Hintergrund hat der Compiler an dieser Stelle also gemäß der für uns unsichtbaren „Bauanleitung" (Klassendefinition) `Stepper` ein Objekt mit dem Namen `Schrittmotor` erstellt. Innerhalb der Klassendefinition wurde dem Compiler auch mitgeteilt, wie er die übergebenen Argumente (in diesem Fall ein Getriebekennwert und die Anschlusspins) verarbeiten soll. Hier kommt wieder die bereits angesprochene Konstruktor-Funktion ins Spiel: Sie erhält die bei der Objekterstellung übergebenen Argumente und kann sie entsprechend verarbeiten.

Die Klasse Stepper.h sieht auch weitere Funktionen vor, die dann den erstellen Objekten zugehören – genau so, wie wir es eben auch beim Konto-Beispiel kennengelernt haben. Die folgenden beiden Anweisun-

gen rufen jeweils Funktionen des Objektes auf, um interne Variablen zu verändern oder eine Aktion auszulösen:

```
1 ...
2 Schrittmotor.setSpeed(5);
3 Schrittmotor.step(SCHRITTEPROUMDREHUNG);
4 ...
```

Im Beispiel dient die erste Funktion der Anpassung der objektinternen Geschwindigkeits-Variablen. Die zweite lässt den repräsentierten Schrittmotor eine komplette Umdrehung ausführen.

Da die angesprochene Stepper-Bibliothek nur grundlegende Funktionen bietet und es beispielsweise nicht ermöglicht, zwei Motoren zeitgleich zu bewegen, wollen wir uns nun unsere eigene Schrittmotor-Klasse für den bereits bekannten Motortyp 28BYJ erstellen.

In Kapitel 11.4 wurde erklärt, dass für die Ansteuerung 4 Pins notwendig sind, welche einfach zyklisch hintereinander ein- und ausgeschaltet werden. Für die beste Kraftwirkung des Motors nutzt man in jeder der 4 Phasen zwei Pins auf HIGH und zwei Pins auf LOW, wodurch stets beide Spulen des Motors von einem Strom durchflossen werden.

Phase	Pin A	Pin B	Pin C	Pin D
0	HIGH	HIGH	LOW	LOW
1	LOW	HIGH	HIGH	LOW
2	LOW	LOW	HIGH	HIGH
3	HIGH	LOW	LOW	HIGH

Tabelle 15.1 Signalphasen am verwendeten Schrittmotor bei unipolarer Beschaltung

Eine Rotation der Motorwelle kann dadurch erreicht werden, indem einfach alle 4 Phasen wiederholt hintereinander durchlaufen werden. Die Abfolge 0-1-2-3-0-1-2-3-0-1-2-... bewirkt dabei die Bewegung in eine Richtung; die Abfolge 3-2-1-0-3-2-1-0-3-... entsprechend in die Gegenrichtung.

Um die aktuelle Motorposition im Programm abzubilden, nutzen wir die `long`-Variable `aktuellePosition`, welche einfach die durchlaufenen Phasen immer weiter zählt – so wie ein altes Bandzählwerk bei einer Musikkassette. In obiger Tabelle wurde die prinzipiell willkürliche Zuordnung so gewählt, dass die Phasenzählung bei 0 beginnt, wodurch die Berechnung der aktuellen Phase aus einer beliebigen Motorposition einfach durch Modulodivision mit der Zahl 4 (`aktuellePosition%4`) geschehen kann. Der Divisionsrest ergibt also direkt unsere aktuelle Phasenziffer.

Bei Motoren vom Typ 28BYJ müssen wir bedenken, dass ein Durchlaufen aller 4 Phasen zwar für eine komplette Rotation des internen Motors sorgt, allerdings ist dessen Welle über ein festes Getriebe mit der außen sichtbaren Welle verbunden. Daher ist für eine komplette Umdrehung der äußeren Welle das Durchlaufen von 2048 Phasen notwendig. Auch dies soll unsere Klasse berücksichtigen.

Wir möchten zudem sicherstellen, dass mehrere angeschlossene Motoren gleichzeitig bewegt werden können und die Programmabarbeitung nicht anhält, wenn sich ein Motor bewegt (denn dies ist ein großer Nachteil der bisher verwendeten `Stepper.h`-Bibliothek). Um das sicherzustellen, müssen wir die Verwendung von `delay()`-Anweisungen komplett vermeiden und stattdessen mit Hilfsvariablen als Timer arbeiten, wie wir es bereits in mehreren Beispielen angewendet haben. Ziel soll es sein, dass die `loop()`-Schleife trotz „parallel“ laufender Schrittmotorsteuerung dennoch tausende Male pro Sekunde durchlaufen wird und nicht ins Stocken gerät.

Schauen wir uns im Folgenden eine Klassendefinition an, mit welcher wir dies bewerkstelligen können:

```
class Schrittmotor {                                          // (1)
  private:
    byte Modus;                                               // (2)
    unsigned long StartZeit; long StartPosition; long
    ZielPosition;
    long aktuellePosition; int aktuelleGeschwindigkeit; int
    maxGeschwindigkeit;
    long SchritteProUmdrehung;
    byte p1, p2, p3, p4;                          // Anschlusspins
    void Ausgabe(char);
    long Timer;

  public:
    Schrittmotor(byte, byte, byte, byte);                     // (3)
    void bewegePosition(long);
    void bewegeGeschwindigkeit(int);
    void Schleife(void);
    byte gibStatus(void) {return Modus;};                     // (4)
    long gibPosition(void) {return aktuellePosition;};
    long setze0(void) {aktuellePosition = 0; return
    true;};                                                   // (5)
    void Stopp(void) {Modus = 0; return true;};
};

Schrittmotor::Schrittmotor(byte a, byte b, byte c, byte d) // (6)
{
  p1 = a<20 ? a : 1;                                          // (7)
  p2 = b<20 ? b : 2;
  p3 = c<20 ? c : 3;
  p4 = d<20 ? d : 4;
  Modus = 0;                                                  // (8)
  StartPosition = 0L; StartZeit = 0L;
  aktuellePosition = 0L; aktuelleGeschwindigkeit = 0L;
  ZielPosition = 0L; Timer = 0L;
  SchritteProUmdrehung = 2048;
  maxGeschwindigkeit = 15;
  pinMode(p1, OUTPUT); pinMode(p2, OUTPUT);                   // (9)
  pinMode(p3, OUTPUT); pinMode(p4, OUTPUT);
  Ausgabe(4);                                                // (10)
}

void Schrittmotor::Ausgabe(char Phase)                       // (11)
{
 switch(Phase)
  {
    case 0:
      digitalWrite(p1,HIGH); digitalWrite(p2,HIGH);
      digitalWrite(p3,LOW); digitalWrite(p4,LOW);
```

```
49      break;
50      case 1: case -3:                                        // (12)
51        digitalWrite(p1,LOW); digitalWrite(p2,HIGH);
52        digitalWrite(p3,HIGH); digitalWrite(p4,LOW);
53      break;
54      case 2: case -2:
55        digitalWrite(p1,LOW); digitalWrite(p2,LOW);
56        digitalWrite(p3,HIGH); digitalWrite(p4,HIGH);
57      break;
58      case 3: case -1:
59        digitalWrite(p1,HIGH); digitalWrite(p2,LOW);
60        digitalWrite(p3,LOW); digitalWrite(p4,HIGH);
61      break;
62      case 4:
63        digitalWrite(p1,LOW); digitalWrite(p2,LOW);
64        digitalWrite(p3,LOW); digitalWrite(p4,LOW);
65      break;
66    }
67    return true;
68  }
69
70  void Schrittmotor::bewegeGeschwindigkeit
71  (int Geschwindigkeit)                                       // (13)
72  {
73    if(abs(Geschwindigkeit) > 0 && abs(Geschwindigkeit)
74    <= maxGeschwindigkeit)
75    {
76      Modus = 1;
77      aktuelleGeschwindigkeit = Geschwindigkeit;
78      StartPosition = aktuellePosition;
79      StartZeit = millis();
80      return true;
81    }
82    else
83      return false;
84  }
85
86  void Schrittmotor::bewegePosition(long Position)            // (14)
87  {
88    Modus = 2;
89    ZielPosition = Position;
90  }
91
92  void Schrittmotor::Schleife(void)                           // (15)
93  {
94    if(!Modus)                                                // (16)
95      Ausgabe(4);
96    if(Modus == 1)                                            // (17)
```

```
97    {
98      long Zeitdifferenz = millis() - StartZeit;
99      aktuellePosition = StartPosition
100       + (Zeitdifferenz * SchritteProUmdrehung
101       * aktuelleGeschwindigkeit / 60000L);
102     Ausgabe(aktuellePosition % 4);                          // (18)
103   }
104   if(Modus == 2 && millis() > Timer)                        // (19)
105   {
106     if(ZielPosition > aktuellePosition)
107       aktuellePosition++;
108     if(ZielPosition < aktuellePosition)
109       aktuellePosition--;
110     if(ZielPosition == aktuellePosition)
111       Modus = 0;
112     Ausgabe(aktuellePosition%4);                           // (20)
113     Timer = millis() + 1;                                  // (21)
114   }
115 }
116
117 ...
```

(1) In der eigentlichen Definition unserer Klasse `Schrittmotor` benennen wir zunächst die Variablen und Funktionen, welche wir verwenden werden. Ihre konkreten Bedeutungen werden im späteren Kontext ersichtlich, daher sei hier auf detaillierte Kommentare verzichtet.

(2) Die Variable `Modus` soll die aktuelle Betriebsart repräsentieren:

0 – Stillstand

1 – Bewegung mit konstanter Geschwindigkeit

2 – Bewegung zu bestimmter Zielposition

(3) Die hier angekündigte Konstruktor-Funktion wird mit 4 Argumenten (vom Typ `byte`) angeführt. Somit können wir direkt bei der Erstellung eines Schrittmotor-Objekts die Nummern der 4 Anschlusspins übergeben. Die `Stepper.h`-Bibliothek nutzte das gleiche Verfahren, allerdings mit 5 Argumenten.

(4) Funktionen mit sehr kurzen Deklarationen können wir komplett in der Klassendefinition notieren. So dienen `gibSta-`

`tus()` und `gibPosition()` lediglich dem Auslesen interner Variablen und fallen entsprechend knapp aus.

(5) Auch `setze0()` (zum optionalen Zurücksetzen unseres „Bandlaufwerks“ auf 0) und `Stopp()` (zum sofortigen Anhalten des Motors) benötigen nur eine Codezeile.

(6) Die Konstruktor-Funktion, welche also bei der Erstellung einer Instanz automatisch einmal durchlaufen wird, verarbeitet zunächst die übergebenen Nummern der Anschluss-Pins.

(7) Dabei prüft sie die Plausibilität (zusammen mit den Analog-Pins sind beim Arduino UNO nur Pin-Ziffern von 0 bis 19 möglich) und weist andernfalls eine Standardziffer zu. Die Speicherung der Pinnummern erfolgt in den internen Variablen `p1` bis `p4`.

(8) Zudem werden die Anfangswerte für alle internen Variablen gesetzt. Die Ergänzung „L“ hinter einer Zahl weist den Compiler explizit an, diese Zahl in das Format einer long-Variable zu bringen. Das spart etwas Platz im Programmcode und vermeidet Fehler, da keine zusätzliche Typumwandlung nötig ist. Unser Sketch würde allerdings auch ohne diesen Zusatz funktionieren; dennoch gehört er zu einem guten Programmierstil.

(9) Wir führen bei der Erstellung einer Instanz auch gleich die Konfiguration der Ausgangspins durch – das erspart uns später zusätzliche Arbeit.

(10) Auf die Funktion `Ausgabe()` blicken wir gleich noch genauer; hier bedeutet dieser Befehl nur: Alle 4 Ausgabepins auf `LOW`.

(11) Die interne Funktion `Ausgabe()` soll unsere Schnittstelle zum eigentlichen Motor bilden, indem sie gemäß der im Argument übergebenen Phase entsprechend unserer Tabelle die Anschlusspins auf `HIGH` oder `LOW` schaltet. Zusätzlich sehen wir eine Phase 4 vor, in der einfach alle Pins auf `LOW` geschaltet werden um Strom zu sparen während der Motor stillsteht.

(12) Das übergebene Argument `Phase` kann auch negativ sein, da sich unser „Zählwerk“ `aktuellePosition` bei Rückwärtslauf des Motors in Richtung negativer Zahlen bewegt. Die bereits angesprochene Modulodivision ergibt dann negative Werte. Entsprechend wurden zu den Phasen 1, 2 und 3 noch die jeweiligen negativen Äquivalente als zusätzliche `case`-Optionen angefügt. Die auf den ersten Blick merkwürdige Zuordnung (1 und -3 etc.) ergibt sich daraus, dass auch beim Über- beziehungsweise Unterschreiten der 0 ein zyklisches Weiterlaufen der Phasen nötig ist. Würde man 1 und -1 einander zuordnen, ergäbe sich beim Überstreifen des Nullpunkts eine Richtungsumkehr des Motors.

(13) Die Funktion `bewegeGeschwindigkeit()` dient als Schnittstelle mit der Außenwelt, um den „Befehl“ zu empfangen, den Motor mit einer konstanten Geschwindigkeit (im Argument übergeben, im Maßstab „Umdrehungen pro Minute“) zu bewegen. Eine negative Geschwindigkeit bewirkt den Rückwärtslauf. Innerhalb dieser Funktion wird zunächst die Plausibilität der Anweisung geprüft: Ist der Betrag der Geschwindigkeit größer als 0 und kleiner als die maximal zulässige Geschwindigkeit? Bei zulässiger Geschwindigkeit werden anschließend die internen Variablen entsprechend angepasst. Eine konkrete Motoransteuerung findet innerhalb dieser Funktion jedoch nicht statt; dazu kommen wir weiter unten.

(14) Auch dies ist eine Schnittstellenfunktion. Wird sie aufgerufen, kann dem Motor-Objekt damit die Anweisung erteilt werden, eine konkrete (als Argument übergebene) Zielposition schnellstmöglich anzufahren.

(15) Hier nun die eigentlich „handelnde“ Funktion: Wir definieren eine Funktion `Schleife()` und legen (für uns selbst) fest, dass wir diese Funktion für jede Instanz unserer Klasse innerhalb der späteren `loop()`-Schleife unsere Sketches einmal auf-

rufen. Wir müssen also später daran denken, dies auch so zu implementieren.
Somit stellen wir sicher, dass die Anweisungen der Schleife()-Funktion mehrere Tausend Male pro Sekunde abgearbeitet werden, nämlich bei jedem loop()-Durchlauf. Dies können wir nun nutzen, um die eigentliche Motorsteuerung zu realisieren. Dabei werden in Abhängigkeit der aktuellen Betriebsart (Modus) verschiedene Befehle ausgeführt.

(16) Im Modus 0 (Stillstand) werden lediglich alle Ausgangspins auf `LOW` geschaltet.

(17) Im Modus 1 (konstante Geschwindigkeit) wird bei jedem Durchlauf der Wert für `aktuellePosition` neu berechnet. Dafür herangezogen wird die vergangene Zeit (`ZeitDifferenz`), die `StartPosition` sowie die angestrebte `aktuelleGeschwindigkeit` und die vom Getriebe abhängige Zahl der `SchritteProUmdrehung`. Die Division durch 60000 ergibt sich durch die Angabe der Geschwindigkeit in Umdrehungen pro Minute, da die `ZeitDifferenz` in Millisekunden berechnet wird. (Eine Minute hat 60.000 Millisekunden.)

(18) Der berechnete Positionswert wird per Modulodivision durch 4 in die aktuelle Schrittmotor-Phase umgerechnet und ausgegeben.

(19) Im Modus 2 soll eine vorgegebene Position schnellstmöglich angefahren werden. Dazu wird unsere Zählvariable `aktuellePosition` mit der `ZielPosition` verglichen und gegebenenfalls erhöht beziehungsweise verringert. Ist das Ziel erreicht, wird der Modus in den Stillstand geändert.

(20) Der Schrittmotor folgt also auch in dieser Betriebsart der Zählvariablen `aktuellePosition`, deren zugehörige Motorphase über die Modulodivision berechnet wird. Wir müssen allerdings sicherstellen, dass sich die Zählvariable nicht zu schnell ändert, damit der Motor mechanisch noch folgen kann.

(21) Dafür nutzen wir das bereits bekannte Verfahren mit einer `Timer`-Variable. In diesem Fall genügt bereits ein Abstand von nur einer Millisekunde. So wird sichergestellt, dass die Zählvariable nicht öfter als einmal pro Millisekunde hoch- beziehungsweise runtergezählt werden kann.

Würden wir auf diesen Timer verzichten, würde die Zählvariable bei jedem Schleifendurchlauf (Dutzende Male pro Millisekunde) hochgezählt und der Motor würde mechanisch nicht mehr folgen. Im Modus 1 haben wir diese Gefahr übrigens durch das Prüfen der Geschwindigkeitsvorgabe gegen eine Maximalgeschwindigkeit (siehe Punkt 13) ausgeschlossen.

Damit endet bereits unsere Klassendefinition. Ergänzen wir den Sketch um ein paar Codezeilen, können wir unsere selbsterstelle Schrittmotor-Klasse direkt ausprobieren:

```
...

Schrittmotor Zeiger1(9,10,11,12);                          // (22)
Schrittmotor Zeiger2(4,5,6,7);

void setup()
{
  Serial.begin(9600);                                      // (23)
}

void loop()
{
  Zeiger1.Schleife();                                      // (24)
  Zeiger2.Schleife();

  if(Serial.available() > 0)                               // (25)
  {
    char Eingabe = Serial.read();
    switch(Eingabe)
    {
      case 'r':     // Rechtslauf Zeiger1 mit 10 U/min
        Zeiger1.bewegeGeschwindigkeit(10);
      break;
      case 'R':     // Rechtslauf zeiger2 mit 10 U/min
        Zeiger2.bewegeGeschwindigkeit(10);
      break;
      case 'l':     // Linkslauf Zeiger1 mit 10 U/min
        Zeiger1.bewegeGeschwindigkeit(-10);
      break;
      case 'L':     // Linkslauf Zeiger2 mit 10 U/min
        Zeiger2.bewegeGeschwindigkeit(-10);
      break;
      case 's':     // beide Motoren stoppen
        Zeiger1.Stopp();
        Zeiger2.Stopp();
      break;
      case 't':     // beide Motoren sehr langsam
                    // entgegengesetzt drehen
        Zeiger1.bewegeGeschwindigkeit(1);
        Zeiger2.bewegeGeschwindigkeit(-1);
      break;
      case '0':     // Nullposition anfahren
        Zeiger1.bewegePosition(0);
        Zeiger2.bewegePosition(0);
      break;

      case 'p':     // Ausgabe der aktuellen Position
```

```
48          Serial.print("aktuelle Position Zeiger 1: ");
49          Serial.println(Zeiger1.gibPosition());
50          Serial.print("aktuelle Position Zeiger 2: ");
51          Serial.println(Zeiger2.gibPosition());
52        break;
53      }
54    }
55  }
```

(22) Wir wollen gleich zwei Schrittmotoren steuern und erstellen daher die beiden Objekte `Zeiger1` und `Zeiger2` aus unserer Klasse `Schrittmotor`. Unsere Konstruktor-Funktion erwartet dabei die 4 Anschlusspins als Argumente.

(23) Der serielle Monitor soll uns zum Test auch als Eingabegerät für Befehle dienen.

(24) Gemäß der Vereinbarung mit uns selbst (siehe 15) wollen wir die Funktion `Schleife()` für jedes betreffende Objekt einmal pro `loop()`-Durchlauf aufrufen.

(25) Wir nutzen die Tastatureingabe am seriellen Monitor, um unsere Motorsteuerung zu testen. Dabei wurden einzelnen Buchstaben diverse Beispielaktionen zugeordnet. Natürlich sind beliebige Anpassungen und Erweiterungen denkbar.

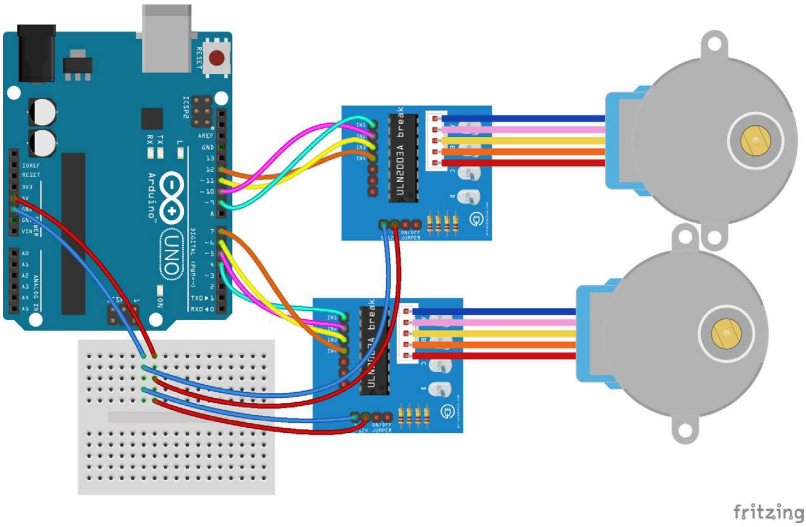


Abb. 15. 2 Beispielaufbau zum Test mit zwei Schrittmotoren

Wenn Sie den Sketch testen, werden Sie feststellen, dass sich beide Motoren mühelos unabhängig voneinander steuern lassen. Zudem können im `loop()` auch noch weitere Aktionen ausgeführt werden (zum Beispiel Sensoreingaben auswerten), da unsere Motor-Objekte den Programmablauf nicht behindern. Einzige Bedingung: Die `Schleife()`-Funktionen der Objekte müssen regelmäßig (bei hohen Motorgeschwindigkeiten Hunderte Male pro Sekunde) aufgerufen werden. Das heißt also, die restlichen Anweisungen im `loop()`-Bereich sollten den Ablauf nicht verzögern. Insbesondere `delay()`-Anweisungen müssen daher vermieden werden.

Es sei erwähnt, dass die in der Klasse definierten Funktionen hier als Anregungen zu verstehen sind – daher haben wir innerhalb dieses Sketches nicht jede dieser Optionen auch benötigt beziehungsweise benutzt (zum Beispiel die Funktion `setze0()`). Bei einer Klassendefinition ist es immer von Vorteil, wenn man bereits an zukünftige Anwendungen denkt und eine möglichst flexible Nutzung ermöglich. Als Weiterentwicklung wäre es daher beispielsweise auch denkbar, die

maximale Geschwindigkeit über eine Schnittstellenfunktion anpassen zu können oder zu prüfen, ob im Modus 1 die `Schleife()` häufig genug aufgerufen wird, so dass keine Schritte übersprungen werden. Dem Erfindergeist sind also auch hier keine Grenzen gesetzt – mit dem zusätzlichen Vorteil, dass die Klasse und die eigentlichen Anwendungsprogramme getrennt voneinander entwickelt werden können. Solange sich nichts Wesentliches an den Schnittstellenfunktionen ändert, wird das Zusammenspiel auch weiterhin funktionieren.

15.5 Bibliotheken erstellen

Nach der erfolgreichen Entwicklung unserer eigenen Schrittmotoren-Klasse liegt der Gedanke nahe, daraus eine Bibliothek zu erstellen, so dass künftige Projekte einfacher davon profitieren können und nicht jedes Mal der komplette Code der obigen Klassendefinition in einen Sketch kopiert werden muss. Dafür müssen wir unsere Klassendefinition auf zwei Textdateien aufteilen und dabei bestimmte Namenskonventionen einhalten. Im Unterordner *\libraries* des Arduino-Verzeichnisses legen wir einen neuen Ordner an, dem wir den gewünschten Namen unserer Bibliothek geben. Dieser darf mit dem Klassennamen übereinstimmen, es ist jedoch nicht zwingend notwendig – Bibliotheken können schließlich auch mehrere Klassendefinitionen enthalten. Wir übernehmen einfach den Klassennamen und nennen den Ordner `Schrittmotor`.

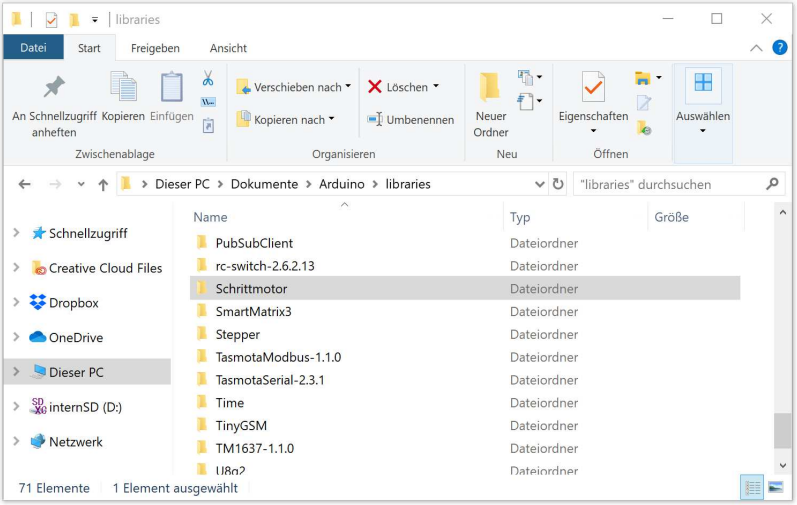


Abb. 15.3 Erstellen Sie einen neuen Ordner im Unterverzeichnis „Arduino\libraries\“

In diesem Ordner legen wir nun die beiden Textdateien *Schrittmotor.h* und *Schrittmotor.cpp* an. Der Dateiname muss also mit dem Bibliotheksnamen übereinstimmen, die beiden Namenserweiterungen sind festgelegt. Nutzen Sie zum Erstellen und Bearbeiten der Dateien einen einfachen Texteditor wie beispielsweise die bei Windows mitgelieferte Software „Editor“. Die Arduino IDE unterstütz uns hier leider nicht. Auch umfangreiche Textverarbeitungsprogramme wie Word sind nicht geeignet, da sie zusätzlich zum Text noch Formatierungsinformationen abspeichern, welche der Compiler als Fehler werten würde.

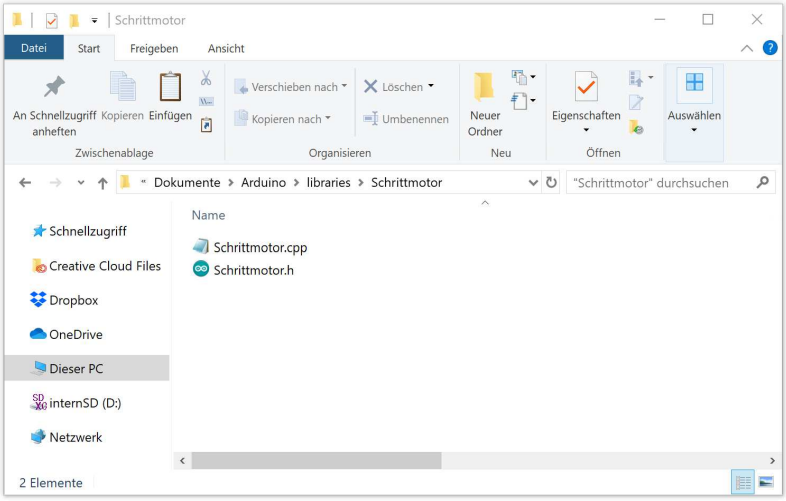


Abb. 15.4 Zwei Textdateien mit den Endungen .h und .cpp sind die Grundlage jeder Bibliothek

In die Datei *Schrittmotor.h* (h steht für *Header*, in Anlehnung an einen Briefkopf) kopieren wir nun die Klassendefinition, jedoch ohne die Funktionsdefinitionen, welche wir außerhalb notiert hatten. Zusätzlich ergänzen wir noch drei Zeilen, welche sich an den Präprozessor richten:

```
1  #ifndef Schrittmotor_h                                    // (1)
2  #define Schrittmotor_h                                    // (2)
3
4  class Schrittmotor {
5    private:
6      byte Modus;
7      unsigned long StartZeit; long StartPosition; long
8      ZielPosition;
9      long aktuellePosition; int aktuelleGeschwindigkeit;
10     int maxGeschwindigkeit;
11     long SchritteProUmdrehung;
12     byte p1, p2, p3, p4;
13     void Ausgabe(char);
14     long Timer;
15
16   public:
17     Schrittmotor(byte, byte, byte, byte);
```

```
18      void bewegePosition(long);
19      void bewegeGeschwindigkeit(int);
20      void Schleife(void);
21      byte gibStatus(void) {return Modus;};
22      long gibPosition(void) {return aktuellePosition;};
23      long setze0(void) {aktuellePosition = 0; return true;};
24      void Stopp(void) {Modus = 0; return true;};
25  };
26
27  #endif                                                    // (3)
```

(1) Die Raute am Zeilenanfang kennzeichnet, dass eine Anweisung an den Präprozessor folgt. In diesem Fall prüft die Anweisung `ifndef`, ob die nachfolgend genannte Konstante (für die wir willkürlich den Dateinamen ohne Punkt übernommen haben), definiert wurde. Ist dies der Fall, also die Konstante noch unbekannt, werden die nachfolgenden Anweisungen (also die Klassendefinition) an den Compiler übergeben. Sonst wird der komplette Block bis zur Anweisung `#endif` übersprungen.

(2) Hier wird die Konstante, deren Existenz wir gerade geprüft haben, definiert. Ihr wird kein Wert zugewiesen, da er für uns nicht relevant ist. Für obige `ifndef`-Anweisung ist nur interessant, ob diese Konstante definiert wurde oder nicht.

(3) `endif` markiert das Ende der an den Präprozessor gerichteten `if`-Anweisung, es ist also analog zu einer schließenden geschweiften Klammer zu verstehen.

Diese auf den ersten Blick etwas seltsame Konstruktion aus Präprozessor-Anweisungen soll lediglich sicherstellen, dass es nicht zu Problemen kommt, wenn ein Anwender die Bibliothek versehentlich mehrfach in seinen Sketch einbindet. In solch einem Fall wird bei wiederholtem Aufruf durch den Präprozessor erkannt, dass die Konstante `Schrittmotor_h` schon existiert, was bedeutet, dass die Klassendefinition bereits erfolgt ist. In diesem Fall überspringt er einfach die erneute Definition. Somit wird eine wiederholte Klassendefinition mit selbem Namen verhindert – andernfalls wäre das Resultat eine Compiler-Fehlermeldung.

In die Datei *Schrittmotor.cpp* (cpp für CPlusPlus, also unsere Programmiersprache C++) kopieren wir nun die Funktionsdefinitionen, welche wir außerhalb der Klassendefinition notiert hatten. Auch hier fügen wir am Anfang zwei Präprozessor-Anweisungen hinzu.

```
#include "Arduino.h"                                          // (1)
#include "Schrittmotor.h"                                     // (2)

Schrittmotor::Schrittmotor(byte a, byte b, byte c, byte d)
{
  p1 = a<14 ? a : 1;
  p2 = b<14 ? b : 2;
  p3 = b<14 ? c : 3;
  p4 = b<14 ? d : 4;
  Modus = 0;
  StartPosition = 0L; StartZeit = 0L;
  aktuellePosition = 0L; aktuelleGeschwindigkeit = 0L;
  ZielPosition = 0L; Timer = 0L;
  SchritteProUmdrehung = 2048;
  maxGeschwindigkeit = 15;
  pinMode(p1, OUTPUT); pinMode(p2, OUTPUT);
  pinMode(p3, OUTPUT); pinMode(p4, OUTPUT);
  Ausgabe(4);
}

void Schrittmotor::Ausgabe(char Phase)
{
 switch(Phase)
  { // Phasen 0-3 stellen die einzelnen Rotationsschritte dar.
    case 0:
      digitalWrite(p1,HIGH); digitalWrite(p2,HIGH);
      digitalWrite(p3,LOW); digitalWrite(p4,LOW);
    break;
    case 1: case -3:
      digitalWrite(p1,LOW); digitalWrite(p2,HIGH);
      digitalWrite(p3,HIGH); digitalWrite(p4,LOW);
    break;
    case 2: case -2:
      digitalWrite(p1,LOW); digitalWrite(p2,LOW);
      digitalWrite(p3,HIGH); digitalWrite(p4,HIGH);
    break;
    case 3: case -1:
      digitalWrite(p1,HIGH); digitalWrite(p2,LOW);
      digitalWrite(p3,LOW); digitalWrite(p4,HIGH);
    break;
    case 4:
```

```
42        digitalWrite(p1,LOW); digitalWrite(p2,LOW);
43        digitalWrite(p3,LOW); digitalWrite(p4,LOW);
44      break;
45    }
46    return true;
47  }
48
49  void Schrittmotor::bewegeGeschwindigkeit(int Geschwindigkeit)
50  {
51    if(abs(Geschwindigkeit) > 0 && abs(Geschwindigkeit)
52    <= maxGeschwindigkeit)
53    {
54      Modus = 1;
55      aktuelleGeschwindigkeit = Geschwindigkeit;
56      StartPosition = aktuellePosition;
57      StartZeit = millis();
58      return true;
59    }
60    else
61      return false;
62  }
63
64  void Schrittmotor::bewegePosition(long Position)
65  {
66    Modus = 2;
67    ZielPosition = Position;
68  }
69
70  void Schrittmotor::Schleife(void)
71  {
72    if(!Modus)
73      Ausgabe(4);
74    if(Modus == 1)
75    {
76      long Zeitdifferenz = millis() - StartZeit;
77      aktuellePosition = StartPosition
78        + (Zeitdifferenz * SchritteProUmdrehung
79        * aktuelleGeschwindigkeit / 60000L);
80      Ausgabe(aktuellePosition % 4);
81    }
82    if(Modus == 2 && millis() > Timer)
83    {
84      if(ZielPosition > aktuellePosition)
85        aktuellePosition++;
86      if(ZielPosition < aktuellePosition)
87        aktuellePosition--;
88      if(ZielPosition == aktuellePosition)
```

```
89        Modus = 0;
90      Ausgabe(aktuellePosition%4);
91      Timer = millis() + 1;
92    }
93  }
```

(1) Die Präprozessor-Anweisungen kennen Sie bereits gut. Es wird also die Bibliothek Arduino.h eingebunden, welche viele grundlegende Arduino-Funktionen (`delay()`, `millis()` und vieles mehr) bereitstellt. In unsere bisherigen Sketche mussten wir diese Bibliothek nicht einbinden, weil die Arduino IDE diese Ergänzung bei jedem Hochladen automatisch vornimmt. Bei Bibliotheken müssen wir uns jedoch selbst darum kümmern.

(2) Die .cpp-Datei einer Bibliothek muss auch stets die eigene Header-Datei inkludieren.

Bei den Funktionsdefinitionen gibt es keinerlei Änderungen gegenüber unserem bisherigen Sketch. Wenn Sie beide Dateien gespeichert haben, ist die Bibliothek nutzbar. Starten Sie dazu die Arduino IDE neu (denn nur beim Start werden die vorhandenen Bibliotheken geprüft) und testen Sie den Sketch aus dem vorigen Abschnitt, indem Sie den kompletten oberen Teil einfach durch das Einbinden der Bibliothek ersetzen.

```
1  #include "Schrittmotor.h"
2
3  Schrittmotor Zeiger1(9,10,11,12);
4  Schrittmotor Zeiger2(4,5,6,7);
5
6  void setup()
7  {
8    Serial.begin(9600);
9  }
10
11 void loop()
12 {
13   Zeiger1.Schleife();
14 ...
```

Im `setup()` und `loop()` ergeben sich dadurch keinerlei Änderungen, weshalb hier auf die erneute Auflistung verzichtet werden soll.

Es gibt die Möglichkeit, im Ordner der Bibliothek noch weitere Dateien anzulegen, um künftigen Nutzern zusätzliche Informationen bereitzustellen. Da diese Ergänzungen optional sind, wollen wir hier nur kurz darauf eingehen:

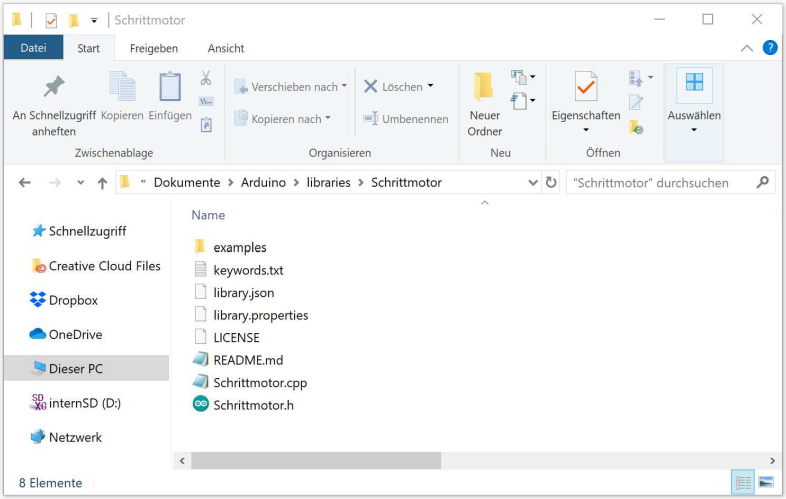


Abb. 15.5 Ergänzende Informationen können in zusätzlichen Dateien bereitgestellt werden

Die Datei `keywords.txt` teilt der Arduino IDE mit, welche Begriffe sie im Sketch farblich hervorheben soll. Dies geschieht recht simpel, indem den Begriffen der Wert KEYWORD1 (üblicherweise für Klassennamen, orange hervorgehoben) oder KEYWORD2 (üblicherweise für Funktionen/Methoden, braun hervorgehoben) zugeordnet wird:

```
1  # Schlüsselwort für Klassennamen (werden orange hervorgehoben):
2  KEYWORD1
3  Schrittmotor                KEYWORD1
4
```

```
# Schlüsselwörter für Funktionen/Methoden: KEYWORD2
gibStatus                   KEYWORD2
gibPosition                 KEYWORD2
bewegeGeschwindigkeit       KEYWORD2
bewegePosition              KEYWORD2
setze0                      KEYWORD2
Stopp                       KEYWORD2
Schleife                    KEYWORD2
```

Die Datei *library.properties* enthält Daten, welche durch den Bibliotheksmanager der Arduino IDE verarbeitet werden[1]. So wird beispielsweise eine Kategorie zugeordnet und eine kurze Erklärung des Zwecks der Bibliothek angegeben. Üblicherweise verfasst man derartige Angaben in Englisch.

```
name=Schrittmotor
version=1.0.0
author=Danny Schreiter
maintainer=Danny Schreiter
sentence=Stepper motor class to be used with 28BYJ stepper motors
paragraph=This library is part of Danny Schreiters Arduino
Kompendium book and shall show an example of how to create
libraries on your own.
category=Device Control
url=http://www.bmu-verlag.de
architectures=*
includes=Schrittmotor.h
```

Die Datei library.json enthält ähnliche Daten, jedoch im etwas abweichenden JSON-Datenformat, welches von anderen Entwicklungsplattformen bevorzugt ausgelesen wird:

```
{
  "name": "Schrittmotor",
  "version": "1.0.0",
  "description": "This library is part of Danny Schreiters
  Arduino Kompendium book and shows an example of how to
  create libraries on your own. It can be used to control
  28BYJ stepper motors.",
```

1 Weitere Informationen über das vom Library Manager benötigte Datenformat: https://github.com/arduino/Arduino/wiki/Arduino-IDE-1.5:-Library-specification

```
  "keywords": "28BYJ, stepper, Arduino ",
  "authors":
  {
    "name":  "Danny Schreiter",
    "url":   "https://www.bmu-verlag.de",
    "maintainer": true
  },
  "repository":
  {
    "type": "git",
    "url": "https://github.com/beispiel"
  },
  "homepage": "https://www.bmu-verlag.de",
  "examples": "examples/*/*/*.ino"
}
```

Die weiteren Dateien `LICENSE` und `README.md` sind Textdateien ohne spezielles Datenformat. Hier können Sie Lizenzhinweise sowie eine Benutzungsanleitung der Bibliothek hinterlegen. Um bei so vielen Dateien nicht den Überblick zu verlieren, dürfen die elementaren .h- und .cpp-Dateien auch in einem Unterordner *\src* (für Source, also Quelltext) abgelegt werden.

Außerdem darf ein Unterordner *\examples* angelegt werden, welcher Beispielsketche enthält. Ein Beispielsketch mit dem Namen `Test1` muss dabei im Unterordner *\examples\Test1* unter dem Dateinamen `Test1.ino` abgelegt werden, um von der Arduino IDE korrekt gefunden und im Menü *Datei -> Beispiele* angezeigt zu werden.

Um die Bibliothek mit anderen zu teilen, können Sie einfach den kompletten Ordner *Schrittmotor* in eine .zip-Datei komprimieren und diese veröffentlichen. Die Datei kann dann von jedem Nutzer über die Arduino IDE installiert werden, wie wir es bereits in Kapitel 2.7 kennengelernt haben.[2]

2 Natürlich steht Ihnen auch die hier als Beispiel vorgestellte Schrittmotor-Bibliothek, ebenso wie alle anderen Codebeispiele, auf www.bmu-verlag.de/arduino-kompendium als Download zur Verfügung

Professionelle Programmierer nutzen zur Weitergabe von Quellcode meist Plattformen wie GitHub. Daraus entstehen Vorteile wie eine sauber dokumentierte Versionsverwaltung und einfachere Zusammenarbeit mit anderen Entwicklern. Wenn Sie Ihre eigene Bibliothek auf GitHub veröffentlichen und dabei bestimmte Konventionen in Bezug auf die Meta-Daten einhalten[3], wird sie sogar automatisch zur Liste des Bibliotheksverwalters der Arduino IDE hinzugefügt und kann so von anderen Nutzern viel einfacher gefunden und installiert werden.

15.6 Weitere Möglichkeiten

Nun sind Sie mit dem Grundprinzip der objektorientierten Programmierung vertraut. Die Möglichkeiten der OOP sind enorm, im Rahmen dieses Einsteigerbuches lässt sich jedoch nur ein Bruchteil davon abbilden. Sollten Sie tiefergehendes Interesse haben, lohnt sich die Lektüre eines entsprechenden Fachbuches. Insbesondere zur Programmiersprache C++ ist viel Literatur verfügbar. Da die in den Arduino-Sketchen verwendete Sprache von C++ abgeleitet ist, lassen sich viele Sachverhalte direkt übertragen. So ist es beispielsweise durch *Vererbung* möglich, Familien von Klassen zu bilden, welche ausgewählte Variablen und Funktionen teilen. Durch sogenanntes Überladen verleihen Sie Ihrer Klasse Funktionen, die flexibel auf verschiedene Variablentypen in den Argumenten reagieren.

Es lässt sich sogar festlegen, wie Instanzen einer Klasse sich verhalten sollen, wenn sie als Bestandteil einer mathematischen Berechnung vorkommen. Eine sehr mächtige Klasse, welche sich vieler dieser Möglichkeiten bedient, kennen Sie bereits: `String`. Mit Objekten dieser Klasse lässt sich die Anweisung `StringA + StringB` mühelos implementieren, obwohl streng genommen gar keine mathematische Addition von Zeichenketten möglich ist. Dank der weitsichtigen Klassendefinition erkennt der Compiler jedoch, dass für String-Objekte ebenfalls eine Vorgehensweise zur Addition hinterlegt ist, welche er anwendet und die Zeichenketten dementsprechend korrekt aneinanderfügt. Dieses Beispiel ist nur eines von vielen, welches die Vorzüge der OOP insbesondere bei großen Projekten erahnen lässt.

3 nähere Informationen: https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ

Kapitel 16
Arduino & Internet

16.1 Grundlagen

Das Internet verändert unseren Alltag beständig, und natürlich macht es auch vor der Bastler-Szene nicht halt. *IoT – Internet of Things* heißt das Stichwort, welches derzeit in aller Munde ist. Der Begriff *Internet der Dinge* meint dabei, dass die Vernetzung nicht mehr nur auf klassische Computer beschränkt ist, sondern bald alle Geräte, egal ob Rasenmäher oder Schreibtischlampe, miteinander verbunden sein können. Man kann sich durchaus praktische Anwendungen vorstellen: Wie wäre es, wenn Ihr Dachfenster automatisch geschlossen werden kann, sobald der Regensensor im Garten schlechtes Wetter meldet? Oder durch Tippen auf das Smartphone alle Wohnzimmerlampen sofort in die Krimi-Lichtstimmung gebracht werden können? Die Möglichkeiten scheinen unbegrenzt, auch wenn sich als Kehrseite dadurch Sicherheitsrisiken ergeben können, falls man allzu unbedacht an das Thema herangeht.

Die folgenden Abschnitte sollen Ihnen daher eine Einführung in die Grundlagen der Vernetzung geben, bevor wir unseren Arduino auf die Datenautobahn schicken.

16.1.1 Das Internet Protocol

Sicherlich haben Sie schon einige Erfahrungen im Internet gesammelt. Sie wissen daher, dass es ein weltweiter Verbund aus unzähligen Computern ist. Viele davon stehen in Privathaushalten, andere wiederum in großen Rechenzentren. Das Besondere daran ist, dass die Vernetzung untereinander nicht wie beim alten Telefonnetz durch fest aufgebaute Verbindungen (“Leitungen“) erfolgt, sondern nur Datenpakete ausgetauscht werden – so wie beim Briefversand. Die Größe eines Daten-

paketes liegt dabei zwischen einigen Dutzend Bytes bis hin zu einigen Kilobytes. Sind größere Datenmengen zu übertragen, werden sie entsprechend aufgeteilt.

Die Struktur des Rechnerverbundes ähnelt unserem Straßennetz: Mehrere Häuser (Computer) werden zu einer Straße (Lokalnetz) zusammengefasst. Die Straße ist mit einer oder mehreren anderen Straßen (weiteren Netzen) verbunden. Gemeinsam bilden sie eine Stadt, ein Land und schließlich einen Kontinent. Der Aufbau des Internets weist also eine gewisse Hierarchie auf, wobei es insbesondere auf höherer Ebene zahlreiche Querverbindungen gibt. Um von einer Stadt zu einer anderen zu gelangen, gibt es ja auch meist mehrere Wege über die Autobahn – wenn auch gegebenenfalls mit Umweg. Dadurch ist sichergestellt, dass selbst bei Ausfällen von einzelnen Internet-Knotenpunkten der Rest des Netzes weiterhin funktionieren kann.

Um nun einen konkreten Rechner im Internet zu finden, gibt das *Internet Protocol* jedem Teilnehmer eine Adresse, die sogenannte *IP*-Adresse. Diese besteht aus 4 Byte, welche als von Punkten getrennte Dezimalzahlen notiert werden, zum Beispiel *95.90.172.144*. Man kann dies wie eine Postadresse verstehen, sie ist von links nach rechts geordnet. *95* könnte also das Land repräsentieren, *90* die Stadt, *172* die Straße und *144* die Hausnummer. In der Realität erfolgt die Einteilung dieser sogenannten Subnetze allerdings nicht genau in 8-Bit-Blöcken (also an den Punkten), sondern die gesamten 32 Bit werden (je nach Netzstruktur) unterschiedlich eingeteilt. So könnten beispielsweise auch nur die ersten 5 Bit für die höchste Hierarchieebene stehen, dann 2 Bit für die folgende Ebene, dann wieder 7 Bit für die nächste Ebene und so weiter. An welcher Stelle genau die Trennung zum übergeordneten Netz liegt, legt die *Subnetzmaske* fest. Sie sieht aus wie eine IP-Adresse, funktioniert aber lediglich wie eine Schablone. Die Subnetzmaske 255.255.0.0 enthält beispielsweise die Information, dass die ersten 16 Bit zu höheren Hierarchieebenen gehören (denn diese Bits haben jeweils den Wert „1“) und die hinteren 16 Bit dem Lokalnetz zur Verfügung stehen (denn sie sind alle „0“).

Als „Adresszusatz“ dienen die sogenannten *Ports*. Diese 16-Bit-Werte können mit einem Doppelpunkt an die IP angehängt werden, um zu bestimmen, für welche Software auf dem Rechner das Datenpaket bestimmt ist. Man kann den Port also als Name des Adressaten verstehen. Wenn Sie an Ihrem Computer gleichzeitig Internet-Radio hören und chatten, muss schließlich sichergestellt sein, dass die entsprechenden Datenpakete auch bei der jeweils korrekten Anwendung landen. Ein üblicher Port für einen Homepage-Server wäre beispielsweise Port 80, in der Adressnotation also *95.90.172.144:80*.

Die Verbindung der einzelnen Rechner erfolgt in der Regel über sogenannte Switches, welche man sich vereinfacht als „Verteiler“ vorstellen kann. Sie kümmern sich um den Datenverkehr in einem Lokalnetz, also auf einer Hierarchieebene. Router haben zusätzlich die Fähigkeit, mehrere Netze zu verbinden. Sie prüfen für jedes Datenpaket anhand der Zieladresse und der Subnetzmaske, ob das Paket für das lokale Netz oder für ein anderes Netz bestimmt ist und leiten es entsprechend weiter.

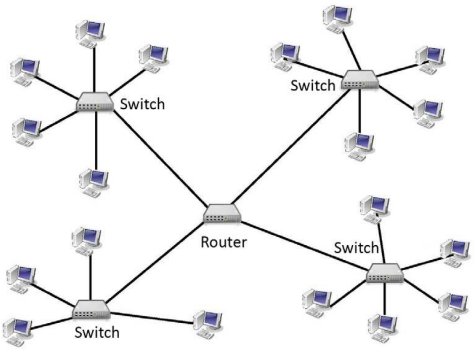


Abb. 16.1 Router können mehrere untergeordnete Netze miteinander verbinden

Möglicherweise kennen Sie dies von Ihrem Router am heimischen Internet-Anschluss. Viele aktuelle Geräte besitzen mehrere LAN-Buchsen. Dort angeschlossene Geräte bilden ein lokales Heim-Netz. Spe-

ziell für solche Zwecke wurde der IP-Adressbereich 192.168.x.x reserviert. So könnte Ihr PC beispielsweise eine lokale IP-Adresse der Form 192.168.0.5 haben. Die Subnetzmaske lautet üblicherweise 255.255.255.0 – das bedeutet, dass sich die Geräte im lokalen Netz nur im letzten Byte der IP-Adresse unterscheiden. Schickt der PC nun ein Datenpaket an *95.90.172.144*, erkennt der Router anhand der Subnetzmaske 255.255.255.0, dass dieses Paket nicht für das lokale Netz bestimmt ist und leitet es daher an die höhere Hierarchieebene (Internet) weiter. Dort kümmern sich weitere Router um die Weiterleitung.

Allerdings birgt dieses Verfahren ein Problem: Schickt die Gegenseite die Antwort an 192.168.0.5, würde sie nie ankommen, denn in fast jedem Heimnetz gibt es diese Adresse – sie ist also nicht eindeutig. Um diesen Nachteil zu umgehen, nutzen heimische Router üblicherweise die sogenannte Netzwerkadressübersetzung (*Network Address Translation – NAT*). Bevor der Router das Paket in das Internet weiterleitet, ersetzt er die Absenderadresse (und -portnummer) durch seine öffentliche Internet-IP. Außerdem generiert er eine Portnummer und trägt diesen Vorgang in eine interne Tabelle ein. Erhält er eine Antwort, kann er anhand der Portnummer nachvollziehen, für welchen Teilnehmer des lokalen Netzes das Paket bestimmt ist, und die entsprechend umgekehrte Adressänderung vornehmen.

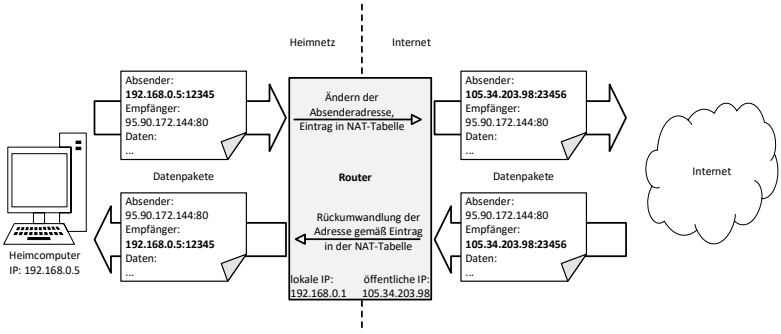


Abb. 16.2 Durch die Network Address Translation ermöglicht der Router den Heimnetzteilnehmern den Internetzugriff und schützt gleichzeitig das Heimnetz vor nicht angefordertem Datenverkehr

Dieses Verfahren hat den Nebeneffekt, dass der Router Daten von außen (Internet) nur dann ins Heimnetz weiterleitet, wenn sie vorher von einem Teilnehmer des Heimnetzes angefragt wurden. Ungefragte Kontaktaufnahmen werden vom Router blockiert, sofern er nicht anders konfiguriert wurde. Dies trägt zur Sicherheit des Heimnetzes bei.

16.1.2 Server-Client-Beziehung

Die meisten Dienste im Internet nutzen eine bestimmte Rollenaufteilung. Dabei gibt es einen oder mehrere Server (Dienstanbieter) sowie einen oder mehrere Clients (Kunden). Ein typisches Beispiel sind Websites: Sie werden auf einem Server bereitgehalten und auf Anfrage eines Clients an diesen übertragen. Wichtig ist dabei, dass die Begriffe Server und Client genau genommen nur für die Software stehen, welche die entsprechende Aufgabe erledigt, nicht jedoch für den kompletten Rechner. So ist es problemlos möglich, dass auf einem Computer mehrere Server und Clients gleichzeitig parallel laufen – beispielsweise wenn Sie Dateien im lokalen Netzwerk freigeben (Server) und im Hintergrund ein Download aus dem Internet läuft (Client). Die Zuordnung der Datenpakete erfolgt, wie bereits erwähnt, anhand der Portnummern.

16.1.3 DHCP

Die Vergabe der IP-Adressen kann auf zwei Arten erfolgen: Bei der ersten Variante existiert ein geplantes Schema, nach dem jedem Gerät manuell eine bestimmte Adresse zugewiesen wird. Diese sogenannte statische IP wird dann in der Software-Konfiguration festgelegt. Dieses Verfahren ist in professionellen Umgebungen wie Unternehmensnetzwerken üblich, in denen es nur selten zu strukturellen Änderungen kommt.

Die zweite Variante nutzt einen DHCP-Server (*Dynamic Host Configuration Protocol*), also eine Software, welche sich automatisch um die Ver-

gabe von IP-Adressen in einem bestimmten Adressbereich kümmert. Üblicherweise läuft auf jedem heimischen Internet-Router ein solcher DHCP-Server. Wird ein neues Gerät dem Netzwerk (egal ob per Kabel oder WLAN) hinzugefügt, fragt es über ein spezielles Protokoll die Vergabe einer IP-Adresse an. Der DHCP-Server teilt diese Adresse zu und führt im Hintergrund eine Tabelle mit den bereits vergebenen Adressen, um Dopplungen zu vermeiden.

Diese Adresse kann sich auch ändern, beispielsweise wenn ein Gerät längere Zeit nicht am Netzwerk angeschlossen war und dann erneut eine IP-Vergabe anfragt, denn möglicherweise hat der DHCP-Server die mittlerweile abgelaufene IP in der Zwischenzeit einem anderen Gerät zugewiesen. Man spricht daher auch von dynamischen (veränderlichen) IPs.

16.1.4 MAC-Adressen

Die gerade erläuterten IP-Adressen sind also in gewissem Rahmen variabel, sie können sich ändern – so wie Ihre Postanschrift bei einem Umzug oder der Neuvergabe von Hausnummern. Aus der Sicht eines Switches oder Routers ist es jedoch nötig, eine eindeutige Kennung der angeschlossenen Geräte zu erhalten. Zu diesem Zweck hat jedes Gerät eine eindeutige MAC[1]-Adresse bestehend aus 6 Byte. Üblicherweise werden diese hexadezimal und durch Doppelpunkt getrennt angegeben. Eine typische MAC-Adresse wäre also `00:80:42:ae:fd:7e`.

Die MAC-Adresse ist nur bei der Verbindung mit dem direkt nächsten Kommunikationspartner relevant (zum Beispiel PC <-> Router) und wird daher bei jeder Weiterleitung durch die eigene MAC-Adresse überschrieben.

1 *Media Access Control* – Medienzugriffssteuerung

16.1.5 Das World Wide Web

Das World Wide Web hat dem Internet zum Ende des letzten Jahrtausends zum Durchbruch verholfen und wird noch bis heute begrifflich oft mit dem Internet gleichgesetzt. Dabei ist es genau genommen nur ein Teil dessen: Als WWW versteht man die unglaublich große Zahl an untereinander verlinkten Websites (ursprünglich *Hypertext-Dokumente* genannt), welche über Webbrowser abrufbar sind. Dienste wie Email, Skype oder Videostreaming gehören also zum Internet, genau genommen aber nicht zum WWW, da sie spezielle Software und Übertragungsprotokolle erfordern. Mittlerweile sind die Grenzen jedoch fließend.

Der Abruf einer Website aus dem World Wide Web folgt dabei streng dem Client-Server-Modell. Die Verständigung ist im *Hypertext Transfer Protocol* (HTTP) geregelt: Der Client (Ihr Webbrowser) schickt eine Anfrage (einen Request) an einen Server und teilt ihm mit, welche Seite er benötigt. Dieser Request könnte wie folgt aussehen:

```
1 GET /infotext.html HTTP/1.1
2 Host: www.beispiel.net
```

Der Client möchte also die Seite (Datei) infotext.html vom Server www.beispiel.net erhalten. (Die Adresse wird anhand einer Art online-Telefonbuch[2] in eine IP übersetzt.) Dabei soll die Kommunikation nach dem Protokoll *HTTP Version 1.1* erfolgen.

Die Antwort des Servers könnte wie folgt aussehen:

```
1 HTTP/1.1 200 OK
2 Server: Apache/1.3.29 (Unix) PHP/4.3.4
3 Content-Length: 123456
4 Content-Language: de
5 Connection: close
6 Content-Type: text/html
7
```

2 *Domain Name System (DNS)* – Register, welches Internetadressen (*Domains*) eine IP zuordnet

```
[Seitenhinhalt von infotext.html]
...
```

Nach dem Protokollnamen sendet der Server einen im Protokoll festgelegten Statuscode, in diesem Fall `200` – also „alles OK“. Den berüchtigten Code `404` kennen Sie womöglich aus eigener Erfahrung beim Surfen: Er bedeutet, dass die angeforderte Datei nicht gefunden wurde.

In der zweiten Zeile stellt sich der Server kurz vor, indem er seine Softwareversion angibt. Als nächstes folgen Länge (in Byte) und Sprache des Dateiinhaltes. Mit `Connection: close` weist der Server darauf hin, dass er nach der Übertragung der Datei die Verbindung beenden wird. Abschließend nennt er noch den Typ des Dateiinhaltes. Im Internet ist das üblicherweise eine Textdatei mit speziellen Formatierungsbefehlen, die *Hypertext Markup Language* (HTML).

Eine sehr einfache HTML-Datei lässt sich auch ohne Webbrowser lesen:

```
<html>
 <head>
  <title>Hier steht der Titel</title>
 </head>
 <body>
  <p>Inhalt der Seite</p>
 </body>
</html>
```

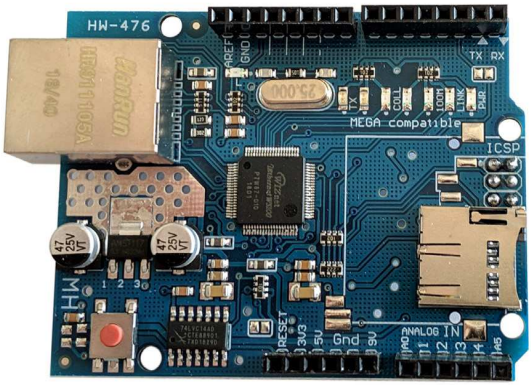
Diese Datei zeigt den Schriftzug „Inhalt der Seite“ in einem weißen leeren Fenster an, welches den Fenstertitel „Hier steht der Titel“ trägt. Die HTML-Befehle, sogenannte Tags, werden von spitzen Klammern eingeschlossen. Beginnt der Tag mit einem Slash (/), schließt er einen bestimmten Bereich ab. So steht beispielsweise der eigentliche Seiteninhalt stets im `body`-Bereich, während allgemeine Informationen (Titel, Angaben über den Autor, ...) üblicherweise im `head`-Bereich notiert werden. Die Einrückungen sind wieder optional, sie werden (genau wie auch Zeilenwechsel) vom Browser ignoriert.

Für das Verständnis der folgenden Abschnitte reichen diese knappen Grundlagen aus. Bei weitergehendem Interesse an HTML sei die Online-Dokumentation SELFHTML[3] empfohlen.

16.2 IoT-Webserver

16.2.1 Arduino mit Ethernet-Shield

Wir wollen nun unseren Arduino nutzen, um einen Webserver bereitzustellen, welcher auf einer sehr einfach gehaltenen HTML-Seite über einen Sensorwert informiert und zudem die Möglichkeit bietet, eine LED ein- und auszuschalten. Hardwareseitig wollen wir dies zunächst über ein Ethernet-Shield lösen, welches per Netzwerkkabel an Ihrem Heim-Router angeschlossen werden kann. Am gebräuchlichsten sind Ethernet-Shields mit dem Chip W5100 des Herstellers *WIZnet*.



3 https://selfhtml.org/

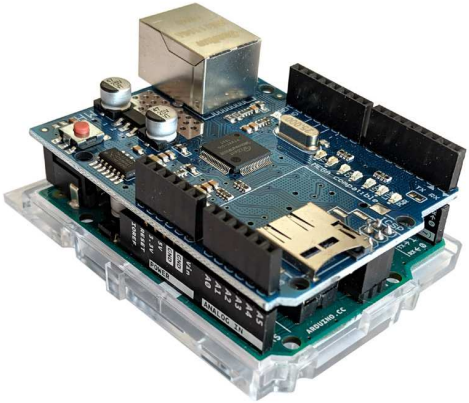


Abb. 16.3 Ethernet-Shield – links einzeln, rechts Arduino UNO mit aufgestecktem Shield. Es kann optional auch als SD-Karten-Leser genutzt werden.

Zum Test schließen wir eine LED und ein Potentiometer an, wie bereits in früheren Projekten.

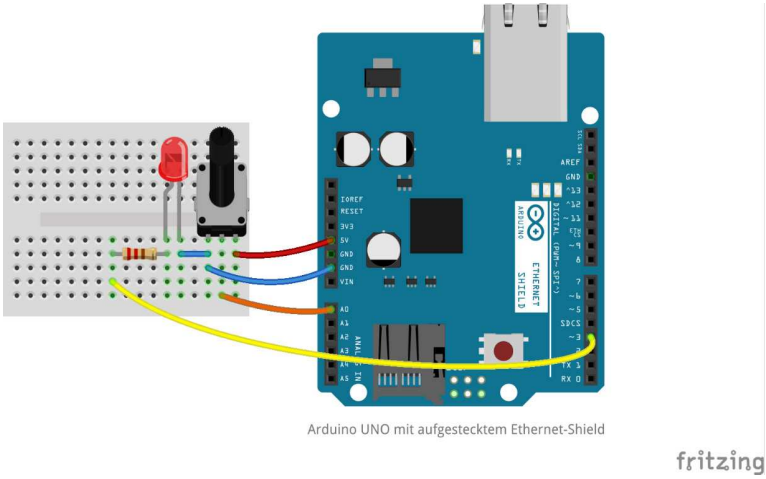


Abb. 16.4 Unter dem Ethernet-Shield befindet sich der hier nicht sichtbare Arduino UNO

Das Ethernet-Shield muss entsprechend per LAN-Kabel mit Ihrem Router verbunden werden. Alternativ können Sie es auch direkt mit einem PC verbinden – in diesem Fall sollten Sie im Sketch an später genannter Stelle eine statische IP vergeben, zum Beispiel „192.168.0.18“, da der PC meist keinen DHCP-Server besitzt.

Ziel ist es, auf einer simpel gestalteten HTML-Seite den Analogwert zu zeigen und per Klick auf einen Link die LED ein- oder ausschalten zu können. Dies wird realisiert, indem der Einschalte-Link auf die gleiche Adresse der Website führt, allerdings erweitert um „?ein“ – der Ausschaltelink endet entsprechend auf „?aus“. Die Schreibweise mit Fragezeichen ist bei Web-Anfragen üblich, wenn zusätzlich zum eigentlichen Server- oder Dateinamen noch weitere Daten (beispielsweise Formulardaten) übertragen werden sollen. Im folgenden Beispiel mit der Server-Adresse „192.168.0.18“ führt der Link „LED Einschalten“ also auf das Ziel „192.168.0.18/?ein“. Unser Arduino-Webserver kann diese Anfrage dann auswerten und entsprechend reagieren.

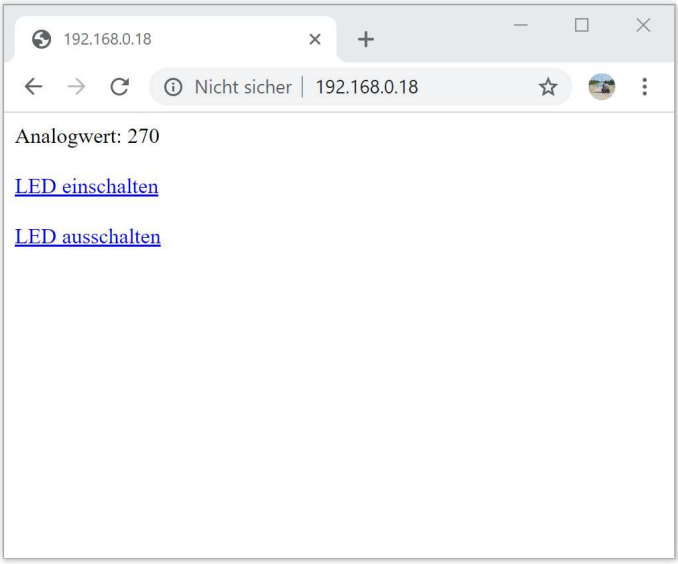


Abb. 16.5 So soll das Resultat im Webbrowser aussehen

Der zugehörige HTML-Code[4] lautet wie folgt:

```
<html>
 Analogwert: 270<br><br>
 <a href='?ein'>LED einschalten</a>
 <br><br>
 <a href='?aus'>LED ausschalten</a>
</html>
```

Die
-Tags sorgen für Zeilenumbrüche (break), die a-Tags erzeugen die anklickbaren Links (Verweise), deren Ziel über die href-Angabe (Hyper-Reference) definiert wird. Ist das Ziel der eigene Server, kann seine Adresse auch weggelassen werden, wovon wir hier Gebrauch machen.

Wir benötigen also ein Programm, welches diesen HTML-Code erzeugen kann und zudem auf Anfragen zum Ein- und Ausschalten entsprechend reagiert. Die Bibliothek Ethernet.h ist bereits vorinstalliert, somit können wir direkt den folgenden Sketch hochladen:

```
#define LEDPIN 3
#define ANALOGPIN A0

#include "SPI.h"                                          // (1)
#include "Ethernet.h"

byte MAC[] = {0x00, 0x08, 0xDC, 0x12, 0x34, 0x56};        // (2)

EthernetServer Server(80);                                // (3)
EthernetClient Client;
String Request;

void setup()
{
  pinMode(LEDPIN, OUTPUT);
```

4 Zur Vereinfachung wurde hier auf Head- und Body-Teile verzichtet. Damit genügt das HTML-Dokument zwar nicht professionellen Webdesign-Standards, es wird aber dennoch von allen Webbrowsern korrekt angezeigt. Bei Interesse an den HTML-Standards sei auf die im vorherigen Kapitel beschriebene SELHTML-Dokumentation verwiesen.

```
  Serial.begin(115200);                                    // (4)
  delay(100);
  Serial.println("Eigene IP-Adresse: ");
  Ethernet.begin(MAC);                                     // (5)
  delay(3000);
  Serial.println(Ethernet.localIP());                      // (6)

  Server.begin();                                          // (7)
}

void loop()
{
  Client = Server.available();                             // (8)

  if (Client)
  {
    Serial.println("Neuer Client");
    boolean leereZeile = true;

    while (Client.connected())                             // (9)
    {
      if (Client.available())                              // (10)
      {
        char c = Client.read();
        if (Request.length() < 100)                        // (11)
          Request += c;

        if (c == '\n' && leereZeile)                       // (12)
        {                                                  // (13)
          Serial.print("Request von Client: ");
          Serial.println(Request);

          if(Request.indexOf("ein")>0)                     // (14)
            digitalWrite(LEDPIN,HIGH);                     // (15)

          if(Request.indexOf("aus")>0)                     // (16)
            digitalWrite(LEDPIN,LOW);

          Client.println("HTTP/1.1 200 OK");               // (17)
          Client.println("Content-Type: text/html");
          Client.println("Connection: close");             // (18)
          Client.println("Refresh: 2");                    // (19)
          Client.println();                                // (20)
          Client.println("<html>");                        // (21)
          Client.print("Analogwert: ");
          Client.print(analogRead(ANALOGPIN));
          Client.println("<br><br>");
```

```
63            Client.println("<a href='?ein'>LED einschalten</a>");
64            Client.println("<br><br>");
65            Client.println("<a href='?aus'>LED ausschalten</a>");
66            Client.println("</html>");
67
68            Request = "";                                    // (22)
69            break;                                           // (23)
70          }
71
72          if (c == '\n')                                     // (24)
73            leereZeile = true;
74          else if (c != '\r')
75              leereZeile = false;
76        }
77      }
78
79      delay(1);                                              // (25)
80      Client.stop();
81      Serial.println("Verbindung mit Client beendet.");
82      Serial.println("");
83    }
84
85    Ethernet.maintain();                                     // (26)
86  }
```

1. Die Verbindung zum Shield erfolgt per SPI, daher ist diese Bibliothek hier ebenfalls nötig.

2. Im Gegensatz zu den meisten anderen Netzwerkgeräten hat der W5100-Chip keine vom Hersteller festgelegte MAC-Adresse. Da für die Verbindung aber zwingend eine solche benötigt wird, legen wir manuell eine Adresse fest, hier also 00:08:DC:12:34:56. Dabei dürfen wir nicht völlig willkürlich vorgehen: Die ersten drei Byte machen den Hersteller kenntlich und sind über Tabellen im Internet ersichtlich. 00:08:DC steht für WIZnet, den Hersteller des Shields. Würden wir hier willkürlich einen anderen Wert eintragen, könnte es sein, dass wir versehentlich einen Adressbereich nutzen, welcher keinem Hersteller zugeordnet ist. Dies führt bei manchen Switches/Hubs zur Ablehnung der Verbindung. Die letzten drei Byte wurden tatsächlich willkürlich gewählt.

3. Aus Klassen der Bibliothek werden Objekte für den Server und den Client erstellt. Der Server erwartet als Argument die Nummer des Ports, unter welcher er Anfragen entgegennehmen soll. Bei unverschlüsselten Websites (http) ist dies stets Port 80.

4. Da die Datenmenge im seriellen Monitor hier und in den folgenden Beispielen etwas ansteigt, wählen wir eine höhere Geschwindigkeit, welche dann auch entsprechend im seriellen Monitor eingestellt werden muss. In diesem und den folgenden Beispielen dient der serielle Monitor nur der besseren Nachvollziehbarkeit und könnte theoretisch auch komplett entfallen.

5. Die Funktion `Ethernet.begin()` startet die Netzwerkverbindung, wenn ihr die MAC-Adresse übergeben wird. Optional kann als zweites Argument eine statische IP übergeben werden. Dies ist nötig, wenn Sie das Modul direkt mit einem anderen PC verbinden. Fehlt dieses Argument, bezieht das Modul seine IP per DHCP.

6. Die (eventuell per DHCP bezogene) IP wird angezeigt.

7. Die Funktion `Server.begin()` startet den eigentlichen Webserver. Ab jetzt achtet das Modul auf Anfragen, welche per Datenpaket an die eigene Adresse, versehen mit der Portnummer 80, gesendet wurden.

8. Liegt eine Anfrage vor, so gibt `Server.available()` eine Referenz (quasi eine Vorgangsnummer) auf den anfragenden Client zurück. Dadurch können IP und Port des Clients später eindeutig zugeordnet werden, wenn wir ihm eine Antwort senden wollen. Liegt keine Anfrage vor, wird `false` zurückgegeben.

9. Solange die Verbindung zum Client besteht,

10. und seine Anfrage (in Form einer Zeichenkette) noch nicht komplett ausgelesen wurde,

11. wird diese nun Byte für Byte einzeln per `Client.read()` ausgelesen und dem String-Objekt `Request` angehängt. Die angefragte Adresse steht immer zu Beginn der Anfrage, danach folgen noch Daten über das Betriebssystem und den verwendeten Browser, wofür wir

uns nicht interessieren. Um Speicherplatz zu sparen, begrenzen wir die Zeichenkette daher auf 100 Byte.

12. Das Ende der Anfrage erkennen wir an einer leeren Zeile, welche von Clients immer als Abschluss gesendet wird. „\n“ steht dabei für das (nicht anzeigbare) Zeichen, welches einen Zeilenumbruch einleitet. An der Hilfsvariable `leereZeile` erkennen wir, ob vorher bereits Zeichen in dieser Zeile enthalten waren (`false`) oder sie tatsächlich leer war (`true`).

13. Ist die Anfrage tatsächlich zu Ende, beginnen wir mit der Verarbeitung und geben anschließend die Antwort aus.

14. Die Funktion `indexOf()` eines String-Objektes sucht in diesem Objekt nach der übergebenen Zeichenkette. Der Rückgabewert ist die Fundstelle oder `-1`, wenn sie nicht gefunden wurde. Wir prüfen also, ob der Request (inklusive angefragter Adresse) das Wort „ein“ enthält.

15. Falls ja, schalten wir die LED ein.

16. Das Ausschalten der LED erfolgt in gleicher Weise.

17. Wir senden unsere Antwort an den Webbrowser. Sie folgt dem Schema, welches wir im vorangegangenen Kapitel kennengelernt haben.

18. Nach der Übertragung wird die Verbindung beendet. Dies ist das übliche Vorgehen bei Webservern.

19. Wir weisen den Client an, die Seite automatisch alle 2 Sekunden neu zu laden, um den Analogwert stets zu aktualisieren.

20. Eine Leerzeile markiert das Ende der Kopfdaten und den Beginn des eigentlichen Inhalts des angeforderten Dokuments.

21. Das in den folgenden Zeilen übertragene HTML-Dokument haben Sie oben bereits gesehen.

22. Nach dem Ende der Übertragung wird der String `Request` geleert, um einen neuen Request aufnehmen zu können.

23. Die Anweisung `break` kennen Sie bereits aus der `switch`-Anweisung. Sie führt zum sofortigen Verlassen der übergeordneten Schleife oder `switch`-Verzweigung. In diesem Fall führt sie also zum Abbruch der `while`-Schleife. Der Programmablauf wird dann an der mit (25) markierten Zeile fortgesetzt.

24. Diese verschachtelte `if`-Bedingung führt dazu, dass die Hilfsvariable `leereZeile` zum Beginn einer neuen Zeile ("\n") stets auf `true` gesetzt wird. Das Auftreten eines nicht zum Zeilenumbruch ("\n" oder auch „\r") gehörenden Zeichens führt hingegen dazu, dass diese den Wert `false` erhält.

25. Nach einer kurzen Pause (weil das Ethernet-Modul nach `print()`-Befehlen in manchen Versionen etwas träge reagiert) wird die Verbindung beendet. Dieser Befehl führt erst zum eigentlichen Senden des Datenpaketes an den Client – vorher werden die Daten im Modul zwischengespeichert.

26. Wenn die IP per DHCP bezogen wurden, ist ein regelmäßiger Aufruf der Funktion `Ethernet.maintain()` nötig. Dadurch tritt das Ethernet-Modul mit dem DHCP-Server in Kontakt, um die Gültigkeitsdauer der erhaltenen IP-Adresse zu verlängern. Unterlässt man dies, läuft die IP irgendwann (üblicherweise nach einigen Stunden) ab und kann dann vom DHCP-Server an ein anderes Gerät vergeben werden.

    Es ist problemlos möglich, diese Funktion einfach bei jedem `loop()`-Durchlauf aufzurufen. Das Ethernet-Modul führt nur dann wirklich eine Aktion aus, wenn die IP tatsächlich kurz vor dem Ablauf ihrer Gültigkeit steht. Ansonsten wird der Aufruf einfach ignoriert.

Um das Beispiel selbst zu testen, öffnen Sie nach dem Hochladen den seriellen Monitor, um die per DHCP zugeteilte IP zu erfahren.

Abb. 16.6 Der serielle Monitor informiert über die per DHCP bezogenen IP-Adresse

Diese IP können Sie einfach in die Adresszeile Ihres Webbrowsers eingeben. Daraufhin öffnet sich die besagte Website. Parallel können Sie den Request am seriellen Monitor verfolgen. Gleiches gilt, wenn Sie einen der LED-Links anklicken:

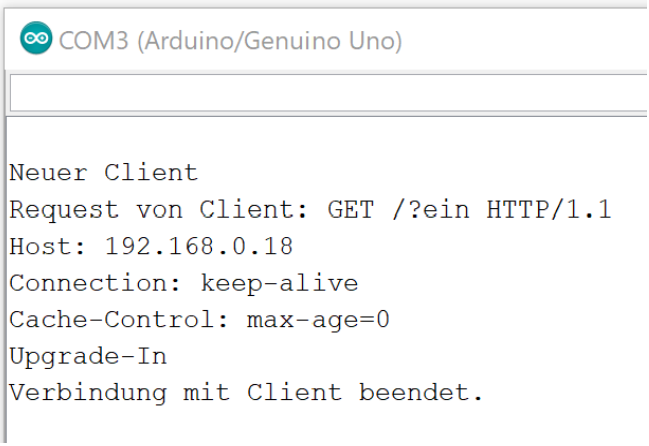


Abb. 16.7 Im Browser wurde der Link zum Einschalten der LED ("?ein") angeklickt und hat diesen Request ausgelöst. Aufgelistet werden nur die ersten 100 Zeichen.

16.2.2 Verbindung über WLAN

Die drahtgebundene Vernetzung ist in der Welt des *Internet of Things* natürlich die Ausnahme, üblicherweise werden Verbindungen per WLAN hergestellt. Zu diesem Zweck wurden für den Arduino Wi-Fi[5]-Shields entwickelt, welche sich ähnlich wie das eben vorgestellte Ethernet-Shield nutzen lassen. Allerdings ist die Ansteuerung oft etwas mühsam: Meist wird die serielle Schnittstelle genutzt, weshalb über einen Schalter auf dem Modul diese Verbindung getrennt werden muss, wenn ein Sketch vom Computer auf den Arduino geladen wird, da dies ja ebenfalls über den seriellen Anschluss geschieht. Die Nutzung des seriellen Monitors ist daher während des Betriebs nicht möglich. Zwar existieren Ausweichlösungen (so kann beispielsweise durch eine spezielle Bibliothek auch eine serielle Verbindung über andere digitale Pins hergestellt werden), jedoch bringen diese stets Nachteile bezüglich der Reaktionszeit und des Speicherplatzes.

Aus diesem Grund hat sich ein anderer Trend entwickelt: Die auf den WiFi-Shields verbauten Chips selbst rücken in den Fokus der Bastler. Es handelt sich hierbei auch um Mikrocontroller, welche (aufgrund ihrer späteren Entwicklung) sogar deutlich leistungsfähiger sind als der auf dem Arduino verwendete ATmega328. Wir wollen daher in den folgenden Abschnitten einen Blick auf die am häufigsten verwendeten Modelle werfen. Um Sie zu testen, benötigen Sie ein lokales WLAN-Netz mit DHCP-Server. Dies ist bei Heim-Internet-Routern der Standardfall.

16.2.2.1 ESP32

Der jüngste „Star" unter den WLAN-fähigen Mikrocontrollern ist der ESP32 der chinesischen Firma *espressif*. Auf einer nur daumennagelgro-

5 Im englischen Sprachraum ist die Bezeichnung *WiFi* beziehungsweise *Wi-Fi* (ein vom Marketing ersonnener Kunstbegriff in Anlehnung an den Qualitätsstandard *Hi-Fi* für hochwertige Audiotechnik) weit gebräuchlicher als die vor allem im Deutschen genutzte Abkürzung *WLAN* (*Wireless Local Area Network* – drahtloses lokales Netzwerk).

ßen Platine bietet der Chip 520 Kilobyte Arbeitsspeicher und (je nach Ausführung) mindestens 4 Megabyte Programmspeicher. Er arbeitet bei einem Takt von bis zu 240 Megahertz.

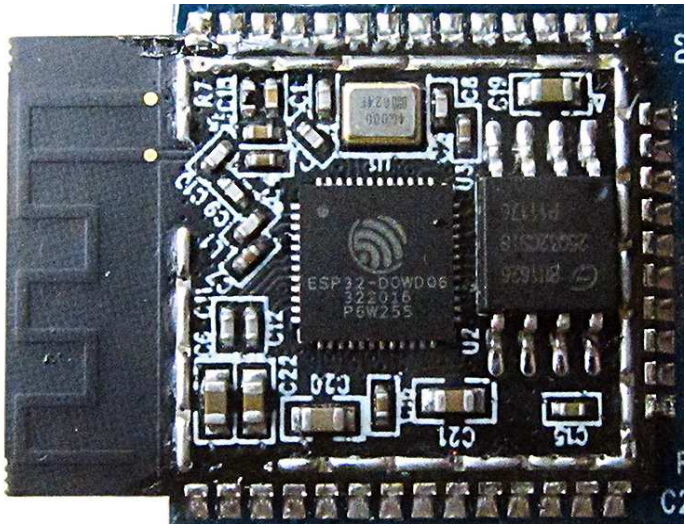


Abb. 16.8 ESP32-Modul mit geöffneter Abdeckung. Im linken Bereich ist die Antenne als schwarz bedruckte Zickzack-Leiterbahn deutlich erkennbar.

Die Chip-Platine selbst verfügt über 38 Kontaktflächen, welche aufgrund der kleinen Abmessungen nur schwer abgreifbar sind. Ähnlich wie beim Arduino gibt es allerdings auch hier Module, welche den ESP32 auf einer größeren Platine aufnehmen, die Anschlüsse an normale Pins führen und einen USB-Anschluss bereitstellen – so wie es der Arduino UNO mit dem ATmega328 vormacht. Leider steht hier jedoch keine zentrale Organisation über dem Projekt, daher unterscheiden sich die Modelle verschiedener Hersteller stark voneinander. Wir orientieren uns im Folgenden am weit verbreiteten „ESP32 Development Module“.

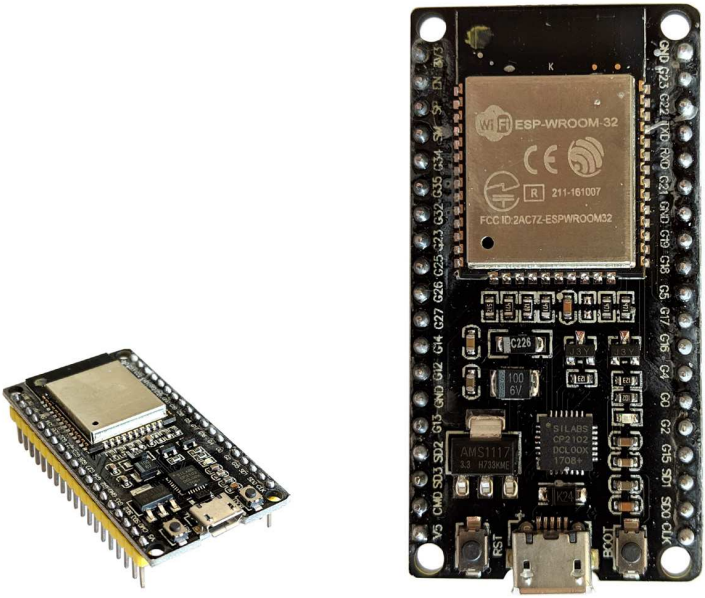


Abb. 16.9 Das ESP32 Development Module führt alle GPIO-Pins (Ein- und Ausgangspins) als breadboardkompatible Steckkontakte aus. Zudem ist ein USB-Anschluss (mit USB-zu seriell-Wandler) für die einfache Programmierung per Computer vorhanden. Die USB-Spannung wird durch einen Spannungsregler auf die benötigten 3,3 Volt reduziert.

Mit Blick auf das Anschlussschema fällt auf, dass es zahlreiche Ein- und Ausgangspins (*GPIO – General Purpose Input/Output*) gibt, wobei einige von ihnen auch weitere Funktionen (wie beispielsweise analoge Eingänge oder I^2C-Schnittstellen) übernehmen können. Zudem muss unbedingt beachtet werden, dass die gesamte Platine mit einer Spannung von 3,3 Volt arbeitet. Es gibt zwar einen Pin, an dem die über USB bezogenen 5 Volt anliegen, jedoch sollte diese Spannung nicht

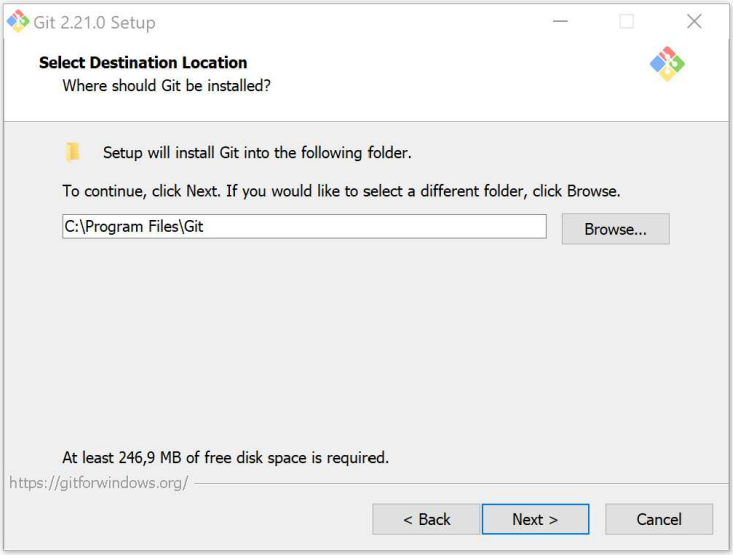
direkt auf die Eingangspins gelegt werden, um den ESP32 nicht zu beschädigen.

Ein Vorteil ist, dass sich der Chip über die Arduino IDE programmieren lässt, wenn zuvor ein entsprechender Boardverwalter und Compiler heruntergeladen wird. Dies wollen wir nun durchführen, um das bereits am Ethernet-Shield betrachtete Webserver-Beispiel auch mit dem ESP32 zu realisieren.

Zunächst müssen Sie Git-Scm, einen kostenlosen Client für die Versionskontrollsoftware Git, herunterladen und installieren:

https://git-scm.com/downloads

Starten Sie dazu die heruntergeladene Datei. Es erscheint ein Installationsprogramm, welches zahlreiche Einstellmöglichkeiten bietet. Lassen Sie alle Einstellungen auf ihren Standardwerten, wenn Sie sich nicht sicher sind.



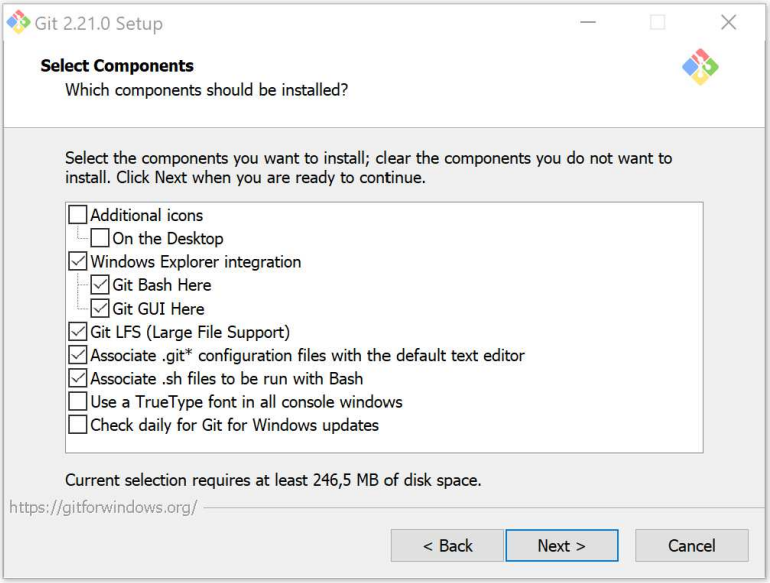


Abb. 16.10 Die Installationssoftware bietet zahlreiche Konfigurationsmöglichkeiten, die nur für professionelle Anwender relevant sind

Nach der Installation starten Sie das Programm Git-Gui aus Ihrem Startmenü und wählen Sie den Punkt „Clone Existing Repository“

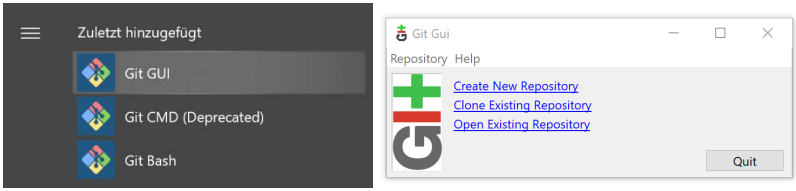


Abb. 16.11 Am einfachsten finden Sie die Software bei Windows 10 in der Rubrik „zuletzt hinzugefügt“

Im daraufhin geöffneten Dialogfeld tragen Sie als Quellort (*Source Location*) folgendes ein:

```
1 https://github.com/espressif/arduino-esp32.git
```

Der Zielort (*Target Location*) ist ein Ordner, den Sie erst anlegen müssen. Öffnen Sie dazu Ihren Windows-Ordner „Dokumente“ und darin den Ordner „Arduino“, legen Sie nun den Unterordner „hardware“ und darin einen Unterordner „espressif“ an. Wählen Sie diesen Pfad über die Browse-Schaltfläche der Git-Gui-Software aus und hängen Sie manuell „/esp32“ an. Diese Unterordner darf noch nicht existieren, sonst erhalten Sie eine Fehlermeldung. Die Software erstellt den Ordner automatisch. An der Einstellung *Clone Type* brauchen Sie nichts zu ändern.

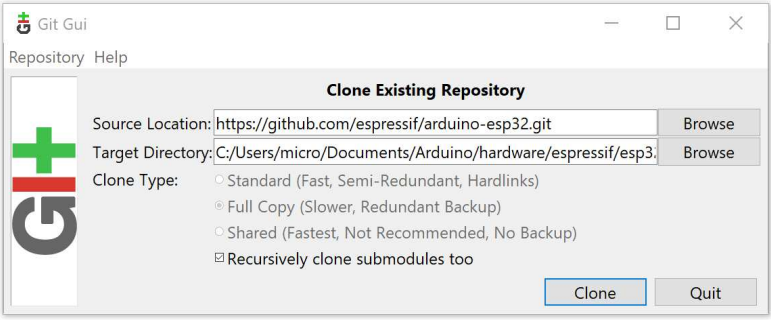


Abb. 16.12 Im Dialogfeld müssen die Felder für die Quelle und das Ziel der Daten angegeben werden. Der Zielordner kann (abhängig vom Namen Ihres Benutzerkontos) geringfügig anders lauten.

Ein Klick auf die *Clone*-Schaltfläche startet das Herunterladen:

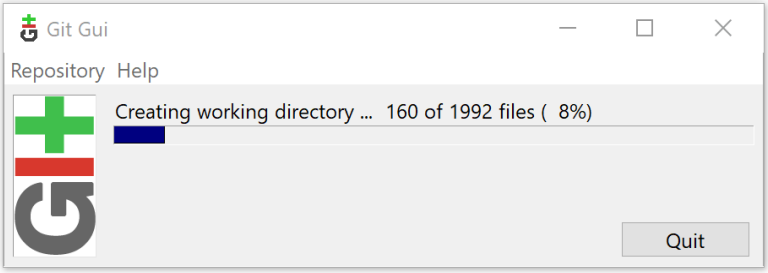


Abb. 16.13 Der Download kann einige Minuten in Anspruch nehmen

Nach Abschluss des Downloads öffnen Sie den Unterordner „tools“ des Zielordners.

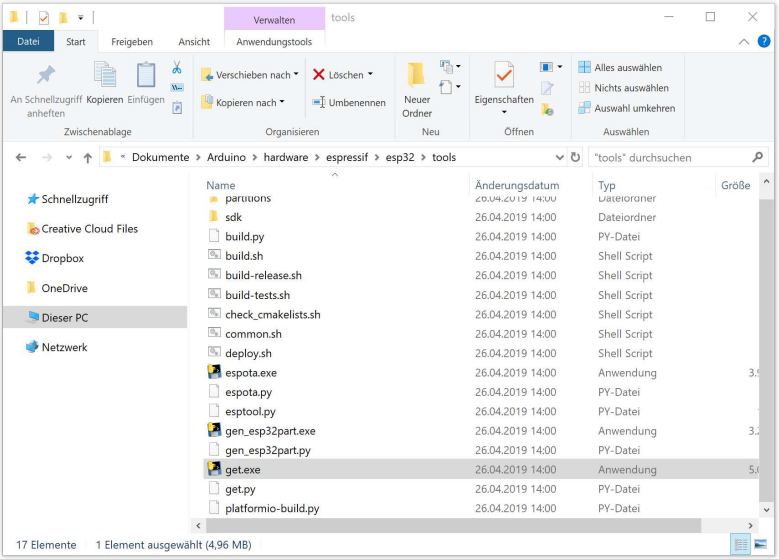


Abb. 16.14 Die ausführbare Datei get.exe lädt den ESP32-Compiler herunter

Darin befindet sich die Datei get.exe. Starten Sie diese, um den Compiler herunterzuladen. Es öffnet sich ein Textfenster, welches über den

Status des Downloads informiert. Nach Abschluss dieses Downloads ist die Installation komplett. Falls die Arduino IDE im Hintergrund geöffnet war, müssen Sie sie einmalig neu starten.

Danach werden im Menü *Werkzeuge -> Board* der Arduino IDE zahlreiche neue Boards angeboten. In unserem Fall wählen wir „ESP32 Dev Module“.

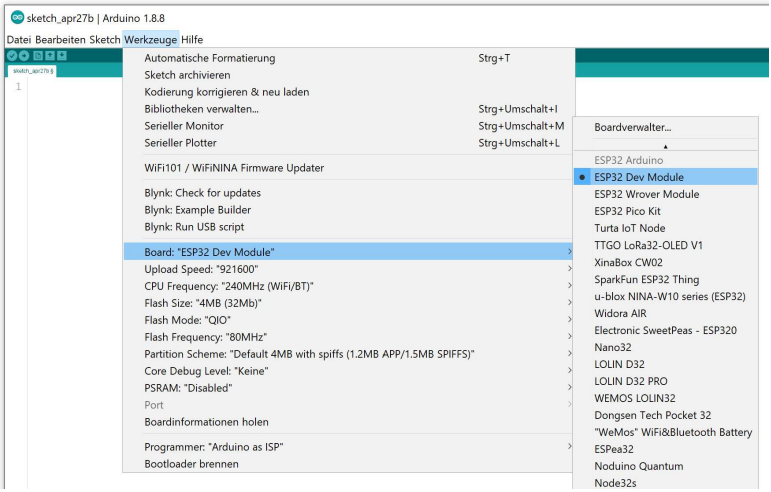


Abb. 16.15 Die Auswahl an Boards mit dem ESP32-Chip ist sehr groß

Nachdem das Modul mit dem Computer verbunden wurde, sollte auch unter dem Menüpunkt *Port* ein entsprechender COM-Port anwählbar sein, wie wir es bereits vom Arduino gewohnt sind.

Um den Webserver-Versuch nachzustellen, verbinden wir das Modul wieder mit dem Potentiometer und der LED. Da diesmal die Spannung nur 3,3 Volt beträgt, kann der Vorwiderstand der LED von bisher 330 Ohm auf 220 Ohm abgesenkt werden. Aber auch ohne diese Änderung ist der Versuch problemlos durchführbar, die LED erreicht lediglich nicht die volle Helligkeit.

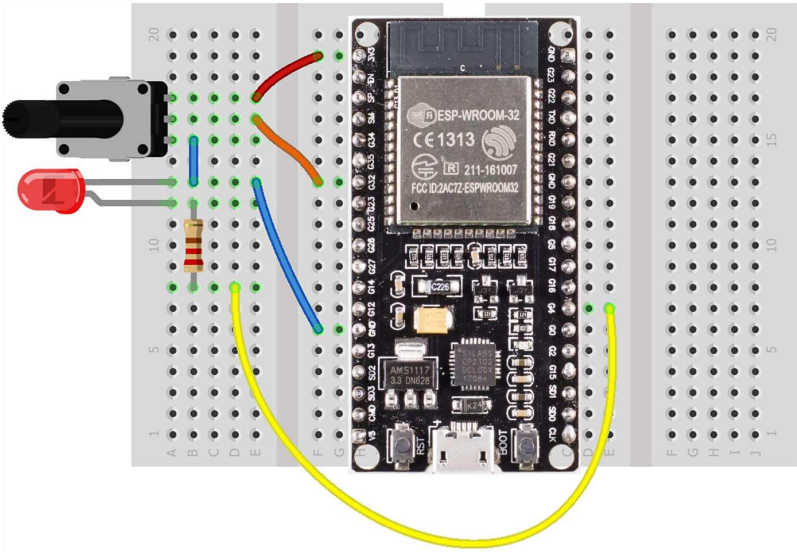


Abb. 16.16 Das Modul ist zu groß für ein einzelnes Breadboard. Durch die Nutzung zweier kleiner Steckbretter kann man sich behelfen.

Im Sketch sind nur geringfügige Anpassungen notwendig, das Meiste kann einfach übernommen werden. Daher werden nachfolgend nur die Unterschiede kommentiert:

```
#define LEDPIN 4                              // = GPIO 4     // (1)
#define ANALOGPIN ADC1_CHANNEL_4              // = GPIO 32

#define NETZWERKNAME "MeinWLAN"                            // (2)
#define PASSWORT "M.e.i.n)P.a-s/s-wo{rt"

#include "WiFi.h"                                          // (3)
#include "driver/adc.h"                                    // (4)

WiFiServer Server(80);                                     // (5)
WiFiClient Client;
String Request;

void setup()
{
  pinMode(LEDPIN, OUTPUT);
```

```
  adc1_config_width(ADC_WIDTH_BIT_10);                        // (6)
  adc1_config_channel_atten(ANALOGPIN,ADC_ATTEN_DB_11);       // (7)

  Serial.begin(115200);
  delay(100);
  Serial.println();
  Serial.print("Verbinde mit: ");
  Serial.println(NETZWERKNAME);

  WiFi.begin(NETZWERKNAME, PASSWORT);                         // (8)

  while (WiFi.status() != WL_CONNECTED)                       // (9)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Erfolgreich. Eigene IP-Adresse: ");
  Serial.println(WiFi.localIP());

  Server.begin();
}

void loop()
{
  Client = Server.available();

  if (Client)
  {
    Serial.println("Neuer Client");
    boolean leereZeile = true;

    while (Client.connected())
    {
      if (Client.available())
      {
        char c = Client.read();
        if (Request.length() < 100)
          Request += c;

        if (c == '\n' && leereZeile)
        {
          Serial.print("Request von Client: ");
          Serial.println(Request);
          if(Request.indexOf("ein")>0)
            digitalWrite(LEDPIN,HIGH);

```

```
64            if(Request.indexOf("aus")>0)
65              digitalWrite(LEDPIN,LOW);
66
67            Client.println("HTTP/1.1 200 OK");
68            Client.println("Content-Type: text/html");
69            Client.println("Connection: close");
70            Client.println("Refresh: 2");
71            Client.println();
72            Client.println("<html>");
73            Client.print("Analogwert: ");
74            Client.print(adc1_get_raw(ANALOGPIN));          // (10)
75            Client.println("<br><br>");
76            Client.println("<a href='?ein'>LED einschalten</a>");
77            Client.println("<br><br>");
78            Client.println("<a href='?aus'>LED ausschalten</a>");
79            Client.println("</html>");
80
81            Request = "";
82            break;
83          }
84
85          if (c == '\n')
86            leereZeile = true;
87          else if (c != '\r')
88              leereZeile = false;
89        }
90      }
91
92      delay(1);
93      Client.stop();
94      Serial.println("Verbindung mit Client beendet.");
95      Serial.println("");
96    }                                                       // (11)
97  }
```

1. Als Ausgang genutzte Pins werden beim ESP über ihre GPIO-Nummer angesprochen. Analoge Eingänge hingegen werden über den Analog-Digital-Wandler (ADC – *Analogue/Digital Converter*), an den sie angeschlossen sind, unterschieden. Auskunft dazu gibt die Online-Dokumentation des jeweils genutzten Moduls.

2. Diese Daten müssen verständlicherweise durch die Daten Ihres WLAN-Netzes ersetzt werden.

3. Die für die Vernetzung verantwortliche Bibliothek heißt nun WiFi.h, sie wurde bereits durch die Git-Software installiert.
4. Abweichend vom Arduino muss hier eine spezielle Bibliothek eingebunden werden, um die analogen Eingänge (ADC) nutzen zu können.
5. Genau wie beim Ethernet-Shield werden auch hier wieder Objekte für den Server und den Client angelegt, lediglich die Klassennamen unterscheiden sich.
6. Die folgenden beiden Funktionen stammen aus der adc.h-Bibliothek und legen Einstellungen für den analogen Eingang fest. Wir wählen eine Auflösung von 10 Bit, also den üblichen Bereich von 0 bis 1023. Maximal wären 12 Bit möglich.
7. Hier wird eine Signalverstärkung gewählt, welche dafür sorgt, dass unser Messbereich genau von 0 bis 3,3 Volt reicht.
8. Im Gegensatz zum Ethernet-Shield erwartet die Funktion `begin()` nun keine MAC-Adresse (diese ist fest vorgegeben), sondern den Namen und das Passwort Ihres WLAN-Netzwerks.
9. Da der Aufbau einer drahtlosen Netzwerkverbindung etwas Zeit beanspruchen kann, wird durch diese `while`-Schleife der weitere Programmablauf so lange verzögert, bis die Verbindung steht.
10. Wie bereits erwähnt, weicht die Schreibweise der Analogpin-Funktionen beim ESP32 aufgrund der separaten Bibliothek etwas ab. Die Handhabung ist aber gleich, die Funktion `adc1_get_raw()` liefert als Rückgabe den eingelesenen Messwert.
11. Die beim Ethernet-Shield notwendige Funktion `maintain()` kann hier entfallen, da sich der ESP32 im Hintergrund selbst um die Aufrechterhaltung der Verbindung und die Gültigkeit der IP kümmert.

Nach dem Hochladen kann wieder über den seriellen Monitor die IP in Erfahrung gebracht werden.

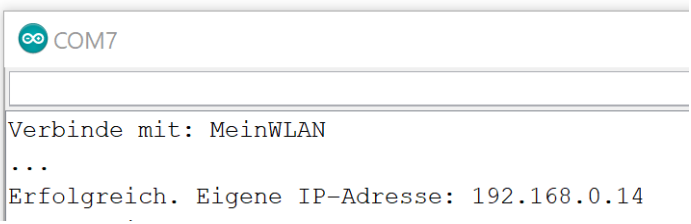


Abb. 16.17 Der DHCP-Server des Routers hat dem ESP32 wie erwartet eine andere IP gegeben

Über den Webbrowser lässt sich nun auch dieser Versuch testen. Er hat die exakt gleiche Funktionalität wie beim Ethernet-Shield, jedoch erfolgt die Datenübertragung nun drahtlos per WLAN.

16.2.2.2 ESP8266

Eine weitere Alternative stellt der ESP8266-Mikrocontroller dar, welcher ebenfalls vom Hersteller *espressif* stammt. Er ist der Vorgänger des ESP32 und deshalb etwas leistungsschwächer, übertrifft den ATmega328 aber immer noch um Längen. Da er schon länger am Markt ist als sein Nachfolger, gibt es eine deutlich größere Entwickler-Community. So ist es viel leichter, eine bestimmtes Codebeispiel für den ESP8266 zu finden als für den ESP32. Die meisten erhältlichen WiFi-Shields für den Arduino verwenden ebenfalls den ESP8266.

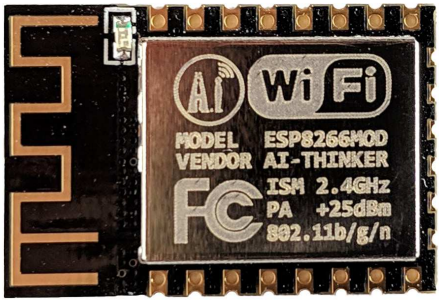


Abb. 16.18 Das einzelne ESP8266-Modul ist geringfügig kleiner als beim ESP32. Auffällig ist wieder die Antenne links.

Auch bei diesem Chip sind die Anschlüsse in seiner minimal-Bauform nicht besonders komfortabel erreichbar. Daher gibt es ebenfalls wieder unzählige Entwickler-Boards, welche das Experimentieren mit den Anschlüssen deutlich erleichtern. Wir beziehen uns nachfolgend auf das weit verbreitete „NodeMCU WiFi Development Board“.

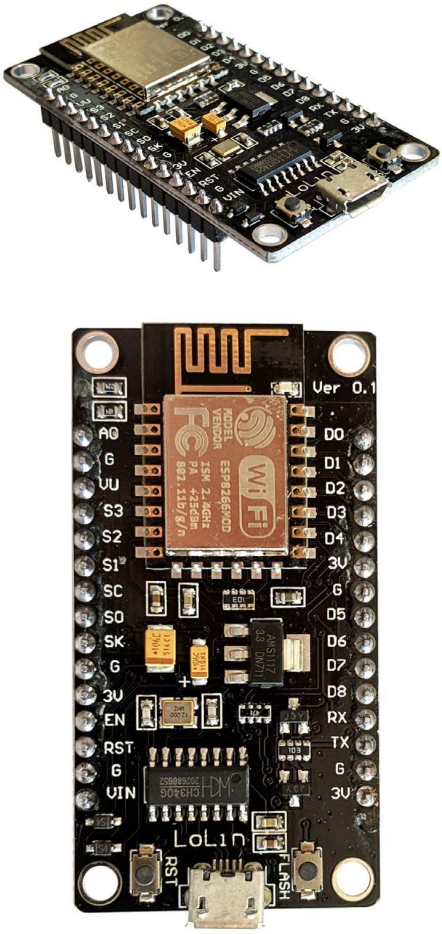


Abb. 16.19 Auch das NodeMCU-Board ist kompatibel zu Breadboards und kann per USB angesteuert werden

Bevor wir es verwenden können, müssen wir wieder die Arduino IDE auf den neuen Mikrocontroller vorbereiten. Diesmal geht das jedoch etwas einfacher.

Öffnen Sie in der Arduino IDE das Menü *Datei -> Voreinstellungen*.

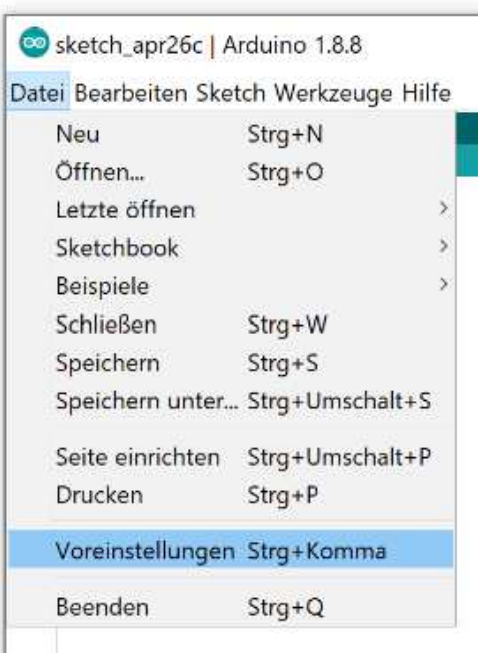


Abb. 16.20 Öffnen Sie das Einstellungsmenü

Im Dialogfeld tragen Sie bei „Zusätzliche Boardverwalter-URLs“ Folgendes ein:

```
1 http://arduino.esp8266.com/stable/package_esp8266com_index.json
```

Bestätigen Sie mit der *OK*-Schaltfläche.

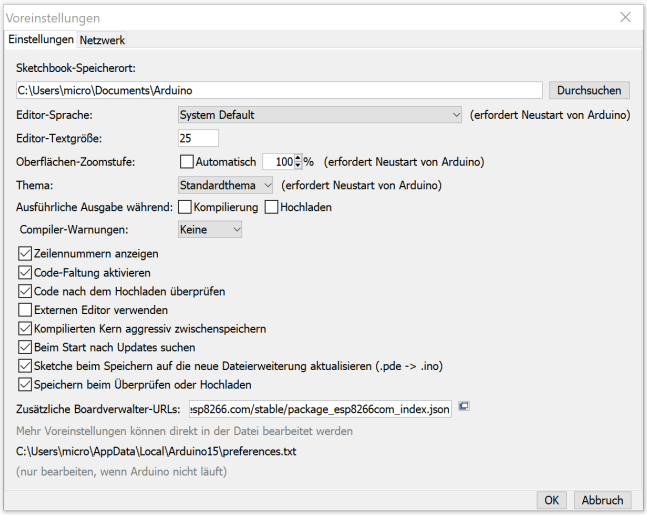


Abb. 16.21 Tragen Sie die Boardverwalter-URL ein. Die anderen Einstellungen bleiben unverändert.

Rufen Sie anschließend über *Werkzeuge -> Board -> Boardverwalter* das Fenster zum Installieren neuer Boards auf.

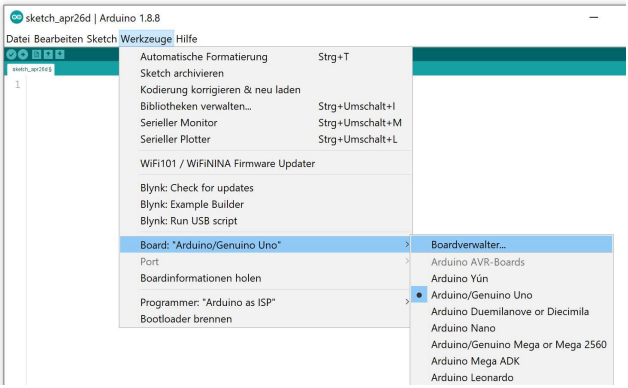


Abb. 16.22 Ähnlich wie bei den Bibliotheken bietet die Arduino IDE auch eine Option zum Verwalten der installierten Boards

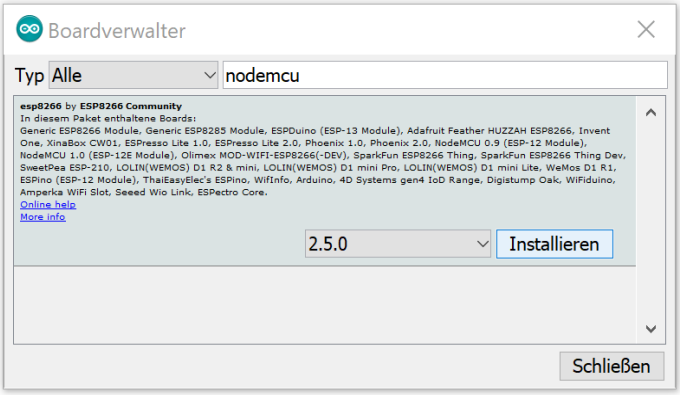


Abb. 16.23 Als alternativen Suchbegriff können Sie auch ESP8266 benutzen

Suchen Sie dort nach NodeMCU und installieren Sie das Paket *esp8266* von der *ESP8266-Community*. Danach steht Ihnen unter *Werkzeuge -> Board* eine breite Auswahl an ESP8266-Boards zur Verfügung.

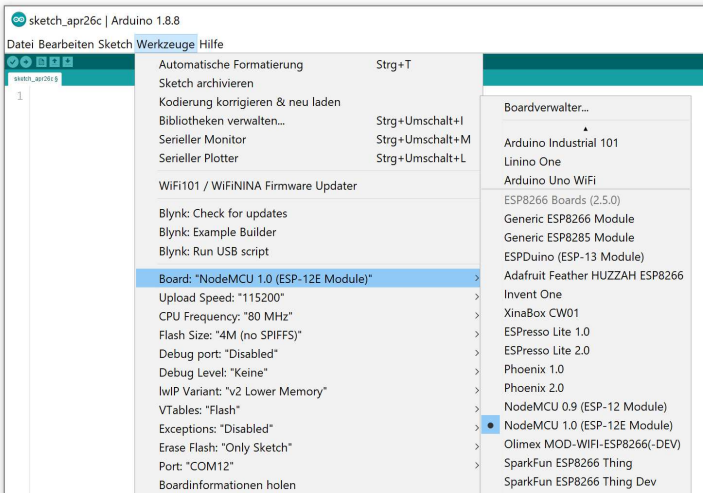


Abb. 16.24 Die neu installierten Boards befinden sich im Abschnitt „ESP8266 Boards“

Wir wählen in unserem Beispiel „NodeMCU 1.0“ (sowie einen passenden Port, wenn das Board angeschlossen ist) und wollen nun ebenfalls den Versuch mit Potentiometer und LED durchführen. Im Versuchsaufbau ändert sich wieder nur der Mikrocontroller:

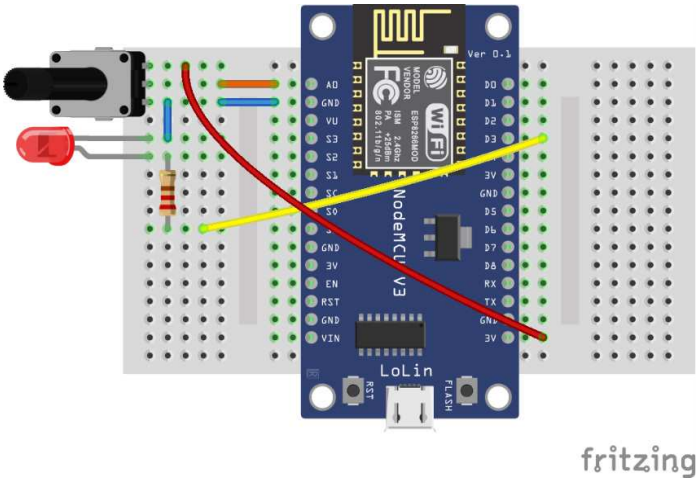


Abb. 16.25 Auch hier wurden zwei kleine Breadboards nebeneinandergelegt

Auch im Sketch gibt es nur geringfügige Änderungen:

```
#define LEDPIN D3                                                    // (1)
#define ANALOGPIN A0

#define NETZWERKNAME "MeinWLAN"
#define PASSWORT "M.e.i.n)P.a-s/s-wo{rt"

#include "ESP8266WiFi.h"                                             // (2)

WiFiServer Server(80);
WiFiClient Client;
String Request;

void setup()
{
  pinMode(LEDPIN, OUTPUT);
  Serial.begin(115200);
```

```
17    delay(100);
18    Serial.println();
19    Serial.print("Verbinde mit: ");
20    Serial.println(NETZWERKNAME);
21
22    WiFi.begin(NETZWERKNAME, PASSWORT);
23
24    while (WiFi.status() != WL_CONNECTED)
25    {
26      delay(500);
27      Serial.print(".");
28    }
29    Serial.println("");
30    Serial.print("Erfolgreich. Eigene IP-Adresse: ");
31    Serial.println(WiFi.localIP());
32
33    Server.begin();
34  }
35
36  void loop()
37  {
38    Client = Server.available();
39
40    if (Client)
41    {
42      Serial.println("Neuer Client");
43      boolean leereZeile = true;
44
45      while (Client.connected())
46      {
47        if (Client.available())
48        {
49          char c = Client.read();
50          if (Request.length() < 100)
51            Request += c;
52
53          if (c == '\n' && leereZeile)
54          {
55            Serial.print("Request von Client: ");
56            Serial.println(Request);
57            if(Request.indexOf("ein")>0)
58              digitalWrite(LEDPIN,HIGH);
59
60            if(Request.indexOf("aus")>0)
61              digitalWrite(LEDPIN,LOW);
62
63            Client.println("HTTP/1.1 200 OK");
```

```
64            Client.println("Content-Type: text/html");
65            Client.println("Connection: close");
66            Client.println("Refresh: 2");
67            Client.println();
68            Client.println("<html>");
69            Client.print("Analogwert: ");
70            Client.print(analogRead(ANALOGPIN));              // (3)
71            Client.println("<br><br>");
72            Client.println("<a href='?ein'>LED einschalten</a>");
73            Client.println("<br><br>");
74            Client.println("<a href='?aus'>LED ausschalten</a>");
75            Client.println("</html>");
76
77            Request = "";
78            break;
79          }
80
81          if (c == '\n')
82            leereZeile = true;
83          else if (c != '\r')
84            leereZeile = false;
85        }
86      }
87
88      delay(1);
89      Client.stop();
90      Serial.println("Verbindung mit Client beendet.");
91      Serial.println("");
92    }
93  }
```

1. Die Namen der Pins entsprechen am NodeMCU exakt dem Aufdruck, daher werden digitale Pins auch mit vorangestelltem D angegeben.
2. Die benötigte Bibliothek heißt ESP8266WiFi.h und wurde bereits über den Boardverwalter installiert.
3. Im Gegensatz zum ESP32 kann die Abfrage des analogen Eingangs hier genau wie beim Arduino erfolgen.

Nach dem Hochladen kann der Sketch wieder getestet werden. Das Verhalten ist wie erwartet identisch zu den vorherigen Abschnitten.

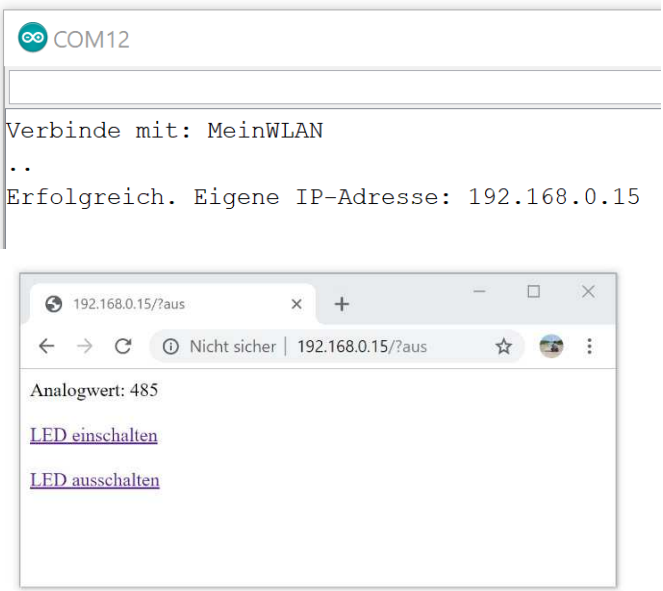


Abb. 16.26 Die am seriellen Monitor abgelesene IP öffnet im Browser wieder die vom Modul bereitgestellte Website

Abschließend sei noch erwähnt, dass der Begriff NodeMCU nicht nur für den Typ des Entwicklerboards steht. Es bezeichnet vielmehr eine komplette Entwicklungsplattform, welche, ähnlich wie die Arduino-Plattform, um den ESP8266 herum entstanden ist. In diesem Umfeld wurde auch ein freies Betriebssystem mit dem Titel NodeMCU entwickelt, welches auf den ESP8266 geladen werden kann. Damit ist es möglich, Programme in der Skriptsprache *Lua* direkt auf dem Mikrocontroller auszuführen, ohne sie vorher von einem Compiler übersetzen zu lassen. Ziel der Entwickler war ein möglichst flexibler Aufbau von Drahtlosnetzwerken, deren Module auch *over the air*, also drahtlos per Fernzugriff, umprogrammiert werden können.[6]

6 Weitere Informationen finden Sie bei Interesse auf der Projektseite: www.nodemcu.com

16.2.2.3 D1mini

Das D1mini ist kein separater Mikrocontroller, sondern ein weiteres Entwicklerboard auf Basis des ESP8266. Es wurde unter dem Gesichtspunkt eines minimalen Platzbedarfs entwickelt, daher sind nicht alle Anschlüsse des ESP8266 nach außen geführt. Für die meisten Anwendungen sollten die vorhandenen 9 digitalen Pins (plus 1 analoger Eingang) jedoch ausreichend sein.

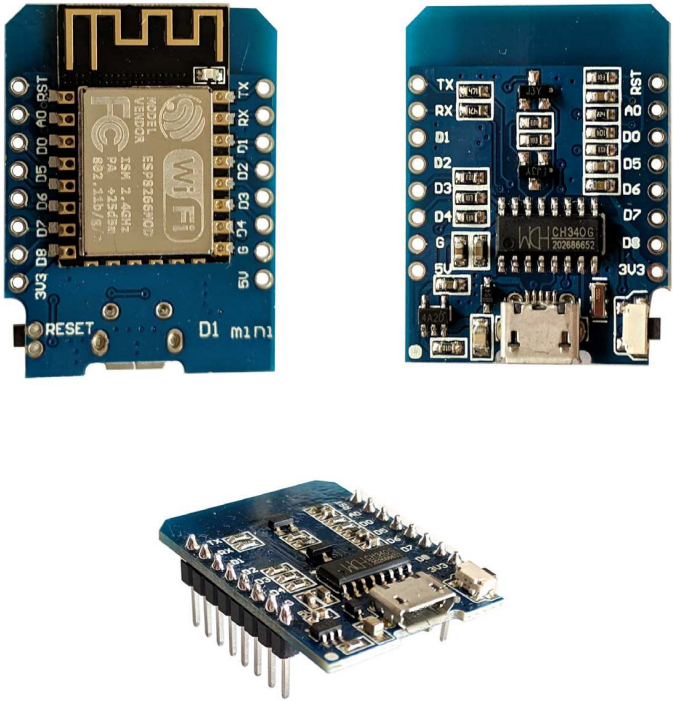


Abb. 16.27 Das D1mini-Board wird mit separaten Pin- und Buchsenleisten geliefert, welche je nach Bedarf (und Platzanforderungen) selbst angelötet werden können. Ein USB-Anschluss ist ebenfalls vorhanden.

Auch mit diesem Board lässt sich unser Versuch nachstellen. In der Arduino IDE sind gegenüber dem NodeMCU keinerlei Änderungen nötig, da volle Kompatibilität besteht. Somit kann der Sketch aus dem vorangegangenen Kapitel auch direkt auf das D1mini geladen werden.

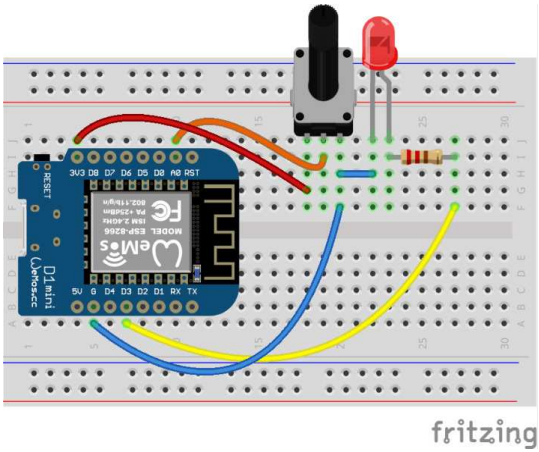


Abb. 16.28 Mit angelöteten Pinleisten kann der D1mini gut auf ein kleines Breadboard gesteckt werden

16.2.2.4 Vergleich zwischen Arduino und ESP

	ATmega328	ESP8266	ESP32
Anzahl der Prozessorkerne	1	1	2
Taktfrequenz	16 MHz	240 MHz	160 MHz
Programmspeicher	32 Kilobyte	4 Megabyte	4 Megabyte
Arbeitsspeicher	2 Kilobyte	160 Kilobyte	520 Kilobyte
Ein- und Ausgangspins	237	17	36

7 Beim Arduino sind es nur 20, mit einer speziellen Konfiguration des Bootloaders können jedoch auch die Schwingquartz-Pins und der Reset-Pin als Ein- und Ausgänge genutzt werden, daher ergibt sich die Maximalzahl von 23.

Analoge Eingänge	6	1	16
Betriebsspannung	5 Volt	3,3 Volt	3,3 Volt
WLAN	nein	ja	ja
Bluetooth	nein	nein	ja
PWM-Kanäle	6	8	16

Tabelle 16.1 Vergleich der verwendeten Mikrocontroller

Ein Vergleich der Mikrocontroller zeigt die deutlichen Leistungsunterschiede auch in Bezug auf den Arduino-Chip ATmega328. Dennoch hat jeder seine Vorzüge: Für den ATmega328 spricht der enorm günstige Preis (unter 2 Euro pro Stück) sowie seine große Community. Für fast jede Aufgabe gibt es im Internet Bibliotheken oder Codebeispiele, welche sofort auf die Arduino-Boards übertragbar sind. Es existieren umfangreiche Dokumentationen, da der Hersteller zahlreiche technische Informationen über den Chip veröffentlicht hat.

Der ESP32 hat als jüngstes Modell noch eine eher geringe Verbreitung. Außerdem ist damit zu rechnen, dass die für ihn bereitgestellten Compiler und Konfigurationsdateien noch nicht frei von Fehlern sind und bestimmte Funktionalitäten noch nicht gut implementiert wurden. (Daher war das Auslesen des analogen Eingangswertes etwas umständlicher.) Dies wird sich in Zukunft möglicherweise noch ändern.

Der ESP8266 vereint aktuell die Vorteile von beiden: Sein Compiler ist ausgereift und er verfügt über eine rege Fangemeinde. Daher ist er für viele Entwickler die erste Wahl, wenn es um die Umsetzung von IoT-Projekten geht. Dennoch erfordert er vom Anwender etwas mehr Erfahrung als der ATmega328, da die Fülle seiner Fähigkeiten auch verwirren kann.

16.2.2.5 Pegelanpassung

Den Umstand der unterschiedlichen Betriebsspannungen zwischen ATmega328-Mikrocontrollern und der ESP-Familie hatten wir bisher nur kurz angesprochen. Möchte man ihre Signalpins untereinander

verbinden (beispielsweise für den seriellen Datenaustausch), ist unter Umständen eine Pegelanpassung vorzunehmen.

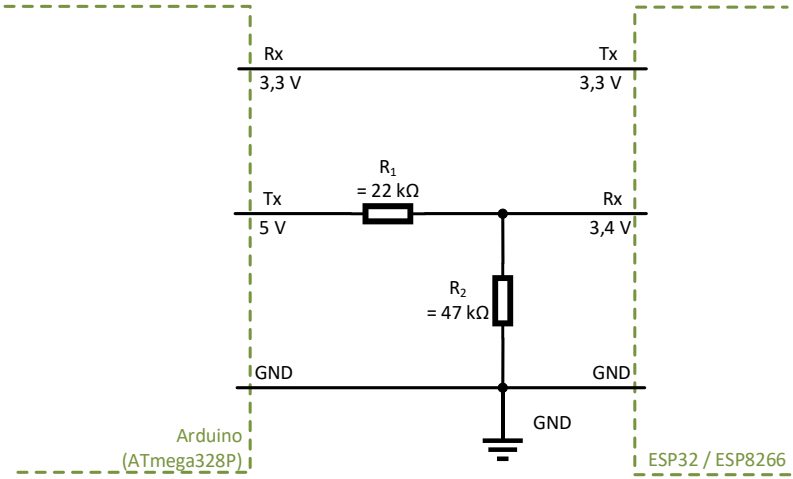


Abb. 16.29 Bei einer seriellen Verbindung ist ein Spannungsteiler für die Pegelanpassung ausreichend

Für das Herabsetzen eines Signals von 5 Volt kann ein einfacher Spannungsteiler (siehe Kapitel 3.1.2.1) genutzt werden. Mit Widerstandswerten von beispielsweise 22 kΩ und 47 kΩ ergibt sich eine Spannung von 3,4 Volt, welche im Toleranzbereich des 3,3-Volt-Chips liegt.

In umgekehrter Richtung ist im einfachsten Fall gar keine Anpassung nötig, da der ATmega328 Spannungen über 3,0 Volt bereits als HIGH-Signal wertet. Gleiches gilt für viele der in diesem Buch vorgestellten externen Module.

Wird von der Gegenseite zwingend eine Signalspannung von 5 Volt erwartet, kann auch einfach der in Kapitel 11.2 vorgestellte Motortreiber L293D zur Pegelanpassung verwendet werden. Er schaltet bereits ab einem Eingangspegel von 2,3 Volt seinen Ausgang auf HIGH (5 Volt).

16.3 MQTT

Die im vorangegangenen Kapitel vorgestellte Webserver-Anwendung lässt erahnen, dass für komplexere Aufgaben neben der Mikrocontroller-Programmierung auch fortgeschrittene Kenntnisse in der Vernetzung und Webseitenprogrammierung nötig sein können. Um dies zu umgehen und sowohl Einsteigern als auch Profis eine flexible und plattformunabhängige Möglichkeit des Datenaustausches zwischen IoT-Geräten und Computern zu ermöglichen, wurde das offene MQTT[8]-Protokoll entwickelt.

16.3.1 Prinzip

Die Idee dahinter ähnelt der einer Pinnwand, an der Familienmitglieder Informationen austauschen. Diese Wand sei geteilt in verschiedene Themenbereiche, welche auch weitere Unterteilungen zulassen. Jedes Familienmitglied kann nun kleine Zettel mit einer Information anheften, beispielsweise „Badewannenfüllstand: halbvoll“ im Themenbereich „Bad“. Zudem kann auch jedes Familienmitglied die von anderen angehefteten Informationen lesen, wobei es sich vielleicht nur für bestimmte Bereiche interessiert.

Die Aufgabe der Pinnwand übernimmt bei MQTT ein Server, welcher auch als MQTT-Broker bezeichnet wird. Er empfängt von allen Teilnehmern (Clients) die Zettelchen (Messages), welche er gemäß ihren Themenbereichen (Topics) sortiert. Anhand von zuvor vereinbarten Abonnements (Subscriptions) weiß der Broker, welcher Client sich für ein bestimmtes Topic interessiert. Erhält der Broker zu diesem Topic eine neue Nachricht, leitet er diese Information sofort an den betreffenden Client weiter.

8 Message Queuing Telemetry Transport – etwa „Datenübertragung mittels Nachrichten-Warteschlangen“

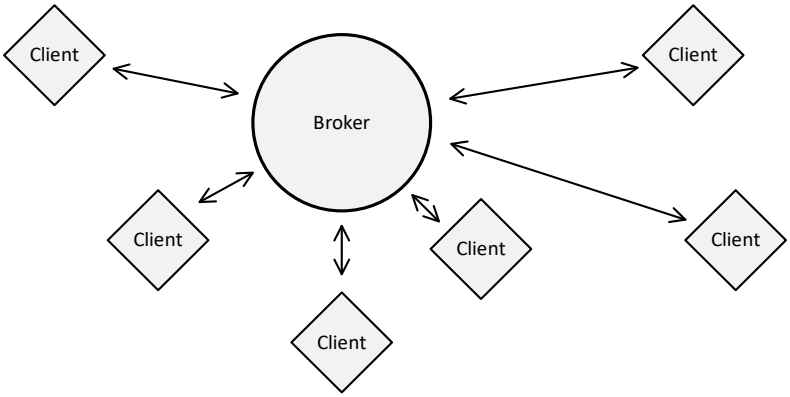


Abb. 16.30 Bei MQTT gibt es keine Direktverbindungen der Clients untereinander. Jeglicher Datenaustausch läuft über den Broker.

Beispiele für Messages von IoT-Geräten in einem Haushalt:

- Topic: „Wohnung/Badezimmer/Temperatur“ Inhalt: „23.5 Grad“
- Topic: „Wohnung/Badezimmer/Luftfeuchte“ Inhalt: „37 %“
- Topic: „Wohnung/Wohnzimmer/Deckenlampe“ Inhalt: „aus“
- Topic: „Wohnung/Wohnzimmer/Temperatur“ Inhalt: „21.8 Grad“

Für den Broker sind sowohl Topic als auch der Inhalt der Message einfache Zeichenketten, er führt keinerlei Verarbeitung durch. Er leitet lediglich die Messages anhand ihrer Topics an bestimmte Subscriber (Abonennten) weiter. Der abonnierende Client hat dafür zuvor den Broker über seine gewünschte Subscription informiert, wobei auch „Joker“ (sogenannte Wildcards) möglich sind. Eine Raute steht dabei für „alle Topics dieser Ebene sowie alle untergeordneten Ebenen“, ein Plus steht hingegen nur für „alle Topics auf dieser Ebene“. Das bedeutet im obigen Beispiel:

- Ein Client mit einer Subscription von „Wohnung/Badezimmer/Temperatur“ erhält nur die erste Message.

- Ein Client mit einer Subscription von „Wohnung/Badezimmer/#“ erhält die ersten beiden Messages.
- Ein Client mit einer Subscription von „#“ erhält alle Messages.
- Ein Client mit einer Subscription von „Wohnung/+/Temperatur“ erhält die erste und die vierte Message
- Ein Client mit einer Subscription von „Wohnung/Bad/Temperatur“ erhält überhaupt keine Message (da die Schreibweise nicht übereinstimmt). Er erhält jedoch auch keine Fehlermeldung vom Broker, welche ihn auf die falsche Schreibweise hinweisen könnte.

Jeder Client kann beliebig viele Subscriptions anlegen und bei Bedarf auch wieder löschen. Ebenso darf jeder Client Messages zu jedem beliebigen Topic absenden (publishen). Es ist dem Programmierer überlassen, in Abhängigkeit von der jeweiligen Anwendung eine sinnvolle Hierarchie der Topics aufzustellen und die entsprechenden Schreibweisen konsequent beizubehalten.

Normale Messages werden nach ihrer Weiterleitung an eventuelle Subscriber vom Broker sofort wieder „vergessen“. Um dies zu verhindern, können Nachrichten auch mit einer Retain-Markierung versehen werden. Der Broker speichert diese dann so lange, bis er zum betreffenden Topic eine neue Retain-Message erhält. Meldet sich in der Zwischenzeit ein Client für ein neues Abonnement zu diesem Topic an, erhält er zu Beginn direkt die letzte Retain-Message dieses Topics vom Broker. Dieses Vorgehen wäre im obigen Beispiel bei der dritten Message (“Deckenlampe aus“) sinnvoll. Vielleicht abonniert der Fernseher dieses Topic, weil er das Umgebungslicht ermitteln will – somit erfährt er sofort den letzten Status der Deckenlampe, auch wenn es schon länger kein Update dazu gab, weil sich nichts geändert hat.

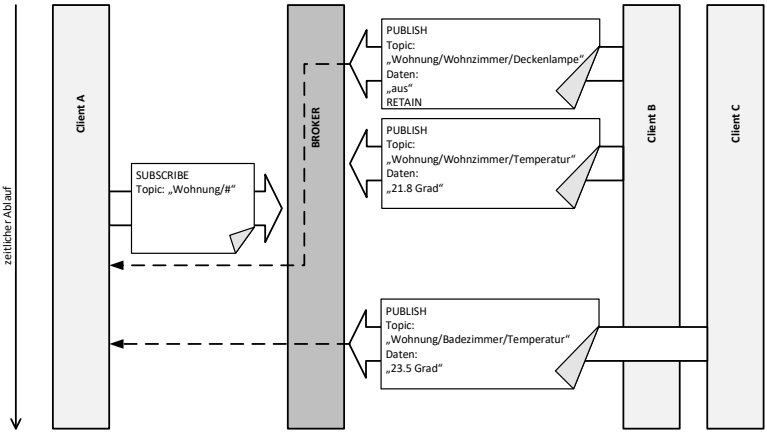


Abb. 16.31 schematische Darstellung des zeitlichen Ablaufs eines MQTT-Datenaustauschs. Client A erhält nach seiner Subscription die letzte Statusnachricht der Deckenlampe, da diese per RETAIN markiert wurde. Die Wohnzimmertemperatur erhält er jedoch vorerst nicht, hier muss er warten bis Client B die Temperatur das nächste Mal publisht.

Des Weiteren ist es möglich, Messages mit einer QoS[9]-Markierung auszustatten. Da bei der Übertragung über ein Netzwerk ja auch Datenpakete verloren gehen können, kann dadurch die Verlässlichkeit gesteigert werden. In der Standardeinstellung „QoS 0“ wird eine Message nur einmal gesendet, ohne irgendeine Rückmeldung zu erwarten. Mit der Einstellung „QoS 1“ wird eine Message so oft wiederholt gesendet, bis die Gegenstelle (Broker oder Client) den Empfang quittiert. Dadurch ist es allerdings möglich, dass Nachrichten mehrfach ankommen. In der höchsten Stufe „QoS 2“ wird sichergestellt, dass die Nachricht genau einmal von der Gegenstelle empfangen wurde. Für die meisten Anwendungen im Heim-Bereich ist die niedrigste Stufe (0) völlig ausreichend. Die Steuerung sicherheitsrelevanter Anlagen (Türschlösser, Garagentore etc.), welche eine höhere Dienstgüte erfordert, sollte ohnehin nur

9 *QoS – Quality of Service*, zu Deutsch „Dienstgüte“ meint in diesem Zusammenhang die Verlässlichkeit der Datenübertragung.

von Profis realisiert werden, da hierfür auch weitergehende Betrachtungen (Verschlüsselung, Schutz vor Fremdzugriff) zwingend nötig sind.

Wir wollen nun das IoT-Beispiel mit Potentiometer und LED auf die MQTT-Plattform übertragen und damit eine deutlich ansprechendere Visualisierung im Browser gestalten.

16.3.2 Software

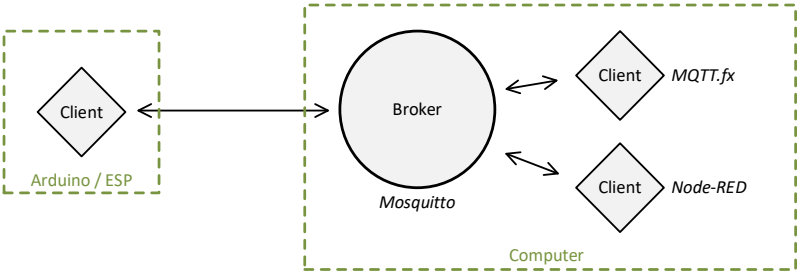


Abb. 16.32 Der Computer wird in unserem Versuch zugleich Broker und Client sein. Wir nutzen dafür die Software „mosquitto“, „MQTT.fx“ und „Node-RED“.

Wir werden sowohl den Broker als auch zwei Clients auf dem PC laufen lassen. Einer der Clients dient uns lediglich der Anzeige der MQTT-Messages, um die Vorgänge besser nachvollziehen zu können. Der andere Client stellt uns eine Browser-Benutzeroberfläche zur Verfügung, mit welcher sich der eingelesene Analogwert komfortabel auswerten und die LED steuern lässt.

Zunächst müssen dafür einige (kostenlose) Programme installiert werden. Den Beginn macht der freie MQTT-Broker *Mosquitto*. Laden Sie die Installationsdatei von der Projektseite herunter:

http://mosquitto.org/download/

Für Windows erhalten Sie eine ausführbare exe-Datei, welche direkt die Installation startet. Die Standardeinstellungen müssen nicht geändert werden.

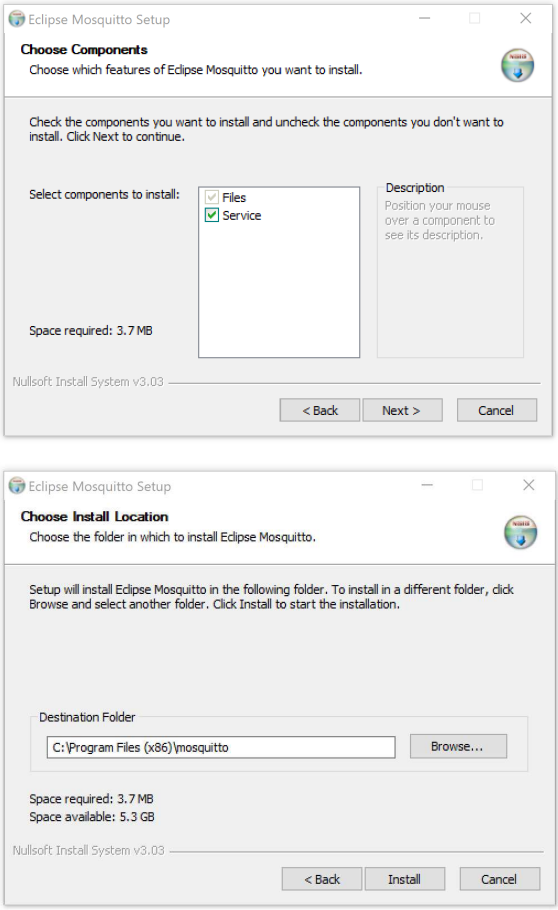


Abb. 16.33 Die Mosquitto-Software belegt nur knapp 4 Megabyte Festplattenspeicher

Damit ist die Installation des Brokers bereits abgeschlossen. Um selbst einen Einblick in den MQTT-Datenverkehr zu bekommen, installieren

wir als Nächstes den freien Desktop-Client *MQTT.fx*. Laden Sie auch hier die aktuelle Version von der Projektseite herunter:

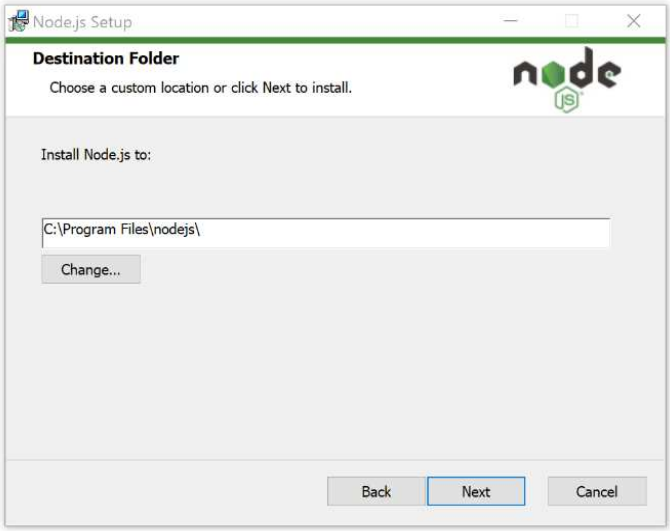
https://mqttfx.jensd.de/index.php/download

Nach der Auswahl des Ordners *Windows* erhalten Sie wiederum eine exe-Datei, welche durch Doppelklick den Installationsvorgang startet.

Nach Abschluss dieser Installation wollen wir zusätzlich noch den Client *Node-RED* herunterladen, welchen wir für die grafische Benutzeroberfläche verwenden können. Diese Software setzt wiederum die Plattform *Node.js* voraus. Laden Sie den zugehörigen Installer von folgender Website:

https://nodejs.org/en/

Die heruntergeladene msi-Datei lässt sich unter Windows ausführen und führt durch die Installation. Das Ändern von Einstellungen ist auch hier nicht notwendig.



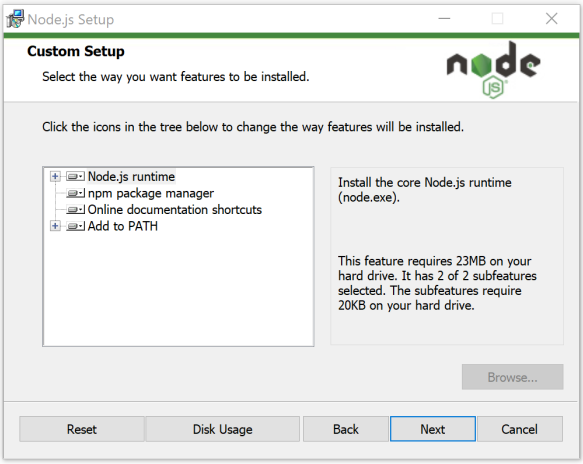


Abb. 16.34 Auch bei der Installation von Node.js können alle Voreinstellungen beibehalten werden

Nach dem Abschluss der Installation der Node.js-Plattform muss nun noch die Node-RED-Anwendung installiert werden. Öffnen Sie dazu die Windows-Eingabeaufforderung, indem Sie im Suchfeld „cmd“ (in älteren Versionen „Ausführen...“) eintippen.

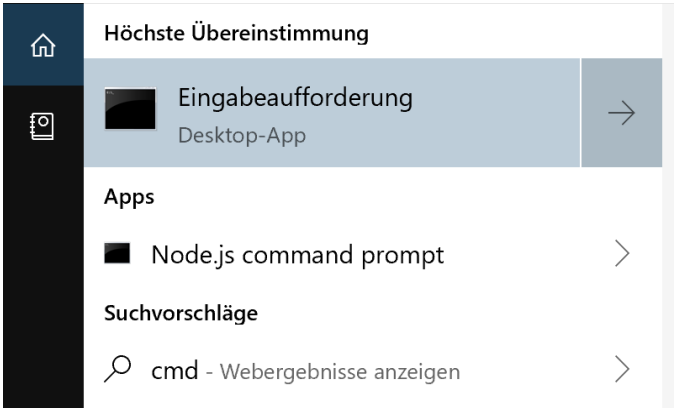


Abb. 16.35 Die Suche nach „cmd“ führt Sie zur Eingabeaufforderung (englisch „command prompt“)

Es öffnet sich ein Fenster. Geben Sie zunächst Folgendes ein:

```
node --version && npm -version
```

und drücken Sie die Enter-Taste, um die korrekte Installation zu überprüfen. Es muss daraufhin die Versionsnummer von Node.js und des zugehörigen Node Paket Managers (NPM) angezeigt werden. Erscheint stattdessen eine Fehlermeldung, wiederholen Sie die Installation.

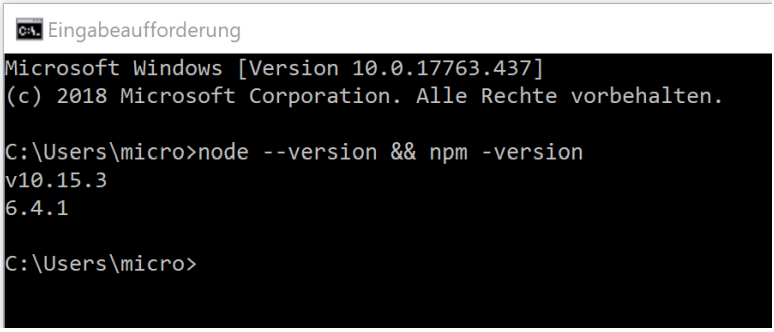


Abb. 16.36 Der Begriff „micro“ steht für den Benutzerkontonamen und kann bei Ihnen anders lauten

Als nächsten Schritt starten Sie die Installation von Node-RED mit dem Befehl

```
npm install -g --unsafe-perm node-red
```

gefolgt von der Enter-Taste. Der Node Paket Manager lädt daraufhin Node-RED herunter und installiert es.

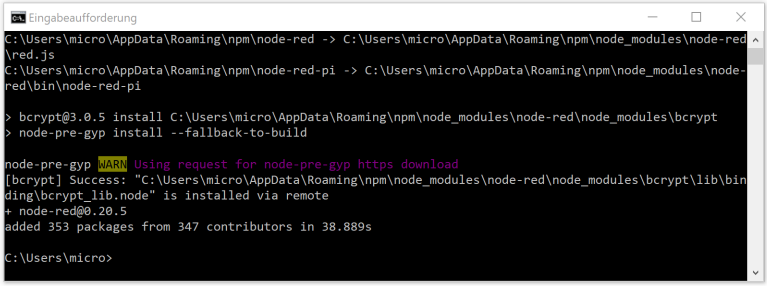


Abb. 16.37 Während der Installation werden diverse Meldungen angezeigt. Enden diese wie hier gezeigt, war die Installation erfolgreich.

Der Computer ist startklar für unser erstes MQTT-Projekt. Nun müssen wir natürlich auch noch die Mikrocontroller-Programmierung anpassen.

16.3.3 Sketch

Laden Sie zunächst die benötigte Bibliothek PubSubClient.h über den Bibliotheksverwalter der Arduino IDE herunter.

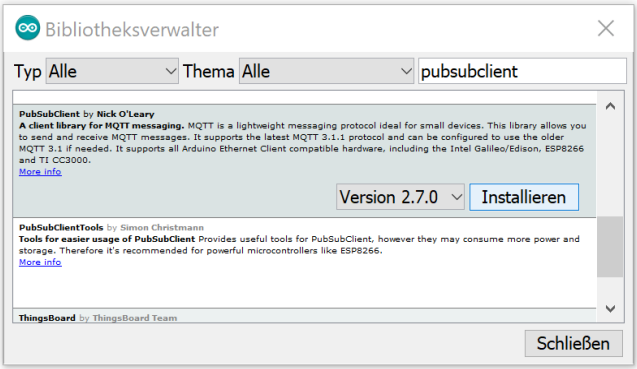


Abb. 16.38 Wir verwenden die MQTT-Bibliothek „PubSubClient“ von Nick O'Leary

Bevor wir den Sketch anpassen, müssen wir noch die IP unseres MQTT-Brokers (in diesem Fall also des Computers) herausfinden. Öffnen Sie dazu noch einmal die Windows-Eingabeaufforderung und tippen Sie den Befehl `ipconfig` ein. Nach dem Druck auf Enter informiert der Computer über die aktuellen Netzwerkverbindungen.

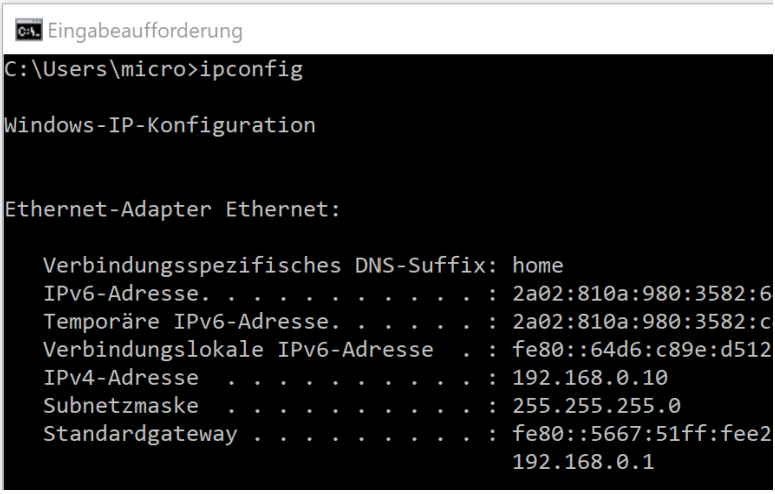


Abb. 16.39 Ignorieren Sie Angaben zu IPv6 – dieser Nachfolger des althergebrachten IPv4-Standards hat in Heimnetzen noch keine Relevanz erlangt

In diesem Beispiel lautet sie also 192.168.0.10 ("IPv4-Adresse"). Nun können wir den Sketch an die MQTT-Plattform anpassen. Die Zielstellung und den Versuchsaufbau übernehmen wir einfach aus dem vorangegangenen Kapitel: Wir wollen den analog eingelesenen Sensorwert (Potentiometer) ausgeben und eine Fernsteuerung der LED ermöglichen. Dabei erweitern wir die Funktion diesmal um die LED-Dimmung per PWM.

Blicken wir dabei zunächst auf den Sketch für den Arduino mit Ethernet-Shield. Die Sketche für den ESP8266 und ESP32 werden danach ebenfalls angeführt, sie unterscheiden sich nur geringfügig.

```
#define LEDPIN 3
#define ANALOGPIN A0
#define MQTT_BROKER "192.168.0.10"

#include "SPI.h"
#include "Ethernet.h"
#include "PubSubClient.h"                                    // (1)

byte MAC[] = {0x00, 0x08, 0xDC, 0x12, 0x34, 0x56};

EthernetClient ethClient;
PubSubClient mqttClient(ethClient);                          // (2)

long Timer = 0;
char Zwischenspeicher[50];

void setup()
{
  pinMode(LEDPIN, OUTPUT);
  Serial.begin(115200);
  delay(100);
  Serial.println("Eigene IP-Adresse: ");
  Ethernet.begin(MAC);
  delay(3000);
  Serial.println(Ethernet.localIP());

  mqttClient.setServer(MQTT_BROKER, 1883);                   // (3)
}

void loop()
{
  if (!mqttClient.connected())                               // (4)
    reconnect();

  mqttClient.loop();                                         // (5)

  if(millis() > Timer)                                       // (6)
  {
    String Analogwert = String(analogRead(A0));
    Analogwert.toCharArray(Zwischenspeicher, 20);            // (7)
    mqttClient.publish("Arduino/Analogwert",
    Zwischenspeicher);                                       // (8)

    Serial.println("Wert published: " + Analogwert);
    Timer = millis() + 5000;
  }
  Ethernet.maintain();
}
```

```
49 void subscribeReceive(char* Topic, byte* Nutzdaten,
50 unsigned int Laenge)                                    // (9)
51 {
52   Serial.println("### Empfangen ###");
53   Serial.print("Topic: ");
54   Serial.println(Topic);                                // (10)
55
56   Serial.print("Nutzdaten: ");
57   for(int i = 0; i < Laenge; i ++)
58     Serial.print(char(Nutzdaten[i]));
59   Serial.println("");
60
61   byte Helligkeit = atoi(Nutzdaten);                    // (11)
62   Serial.print("Helligkeit: "); Serial.println(Helligkeit);
63   analogWrite(LEDPIN, Helligkeit);
64
65   Serial.println("-----------------");
66 }
67
68 void reconnect()                                       // (12)
69 {
70   while(!mqttClient.connected())                        // (13)
71   {
72     Serial.print("Verbinde mit MQTT.");
73
74     if(!mqttClient.connect("EthernetClient"))           // (14)
75     {
76       Serial.print("Fehler: ");
77       Serial.print(mqttClient.state());
78       Serial.println(" Neuer Versuch in 5 Sekunden.");
79       delay(5000);
80     }
81   }
82   mqttClient.setCallback(subscribeReceive);             // (15)
83   mqttClient.subscribe("Arduino/LED");                  // (16)
84 }
```

1. Naheliegenderweise wird nun die soeben heruntergeladene MQTT-Bibliothek benötigt.
2. Das Objekt `mqttClient` repräsentiert nun unseren Client.
3. Die Funktion `setServer()` legt fest, mit welchem Broker sich der Client verbinden soll. Da wir den auf dem Computer installierten Mosquitto-Server nutzen wollen, geben wir dessen IP an. Der Port 1883 ist der Standard-Port für MQTT-Broker.

4. Zu Beginn jedes Durchlaufes der Hauptschleife prüfen wir, ob die Verbindung zum Broker noch besteht. Falls nicht, führen wir die weiter unten definierte Funktion `reconnect()` aus. Nach dem Programmstart wird über diesen Weg auch erstmalig die Verbindung hergestellt.
5. Die PubSubClient.h-Bibliothek erfordert den Aufruf der `loop()`-Funktion des Client-Objektes bei jedem Durchlauf der Hauptschleife. Dabei wird beispielsweise geprüft, ob der Broker neue Messages (aufgrund von Subscriptions) an den Client gesendet hat.
6. Diese `if`-Bedingung wird alle 5 Sekunden aktiv, um den aktuellen Analogwert unseres „Sensors“ (Poti) an den Broker zu senden.
7. Die Bibliothek benötigt den zu sendenden Wert als `char`-Array, daher kopieren wir das String-Objekt `Analogwert` mittels der Objektfunktion `toCharArray()` in das `char`-Array `Zwischenspeicher`.
8. Das Veröffentlichen erfolgt über die Funktion `publish()`, welche als Argumente lediglich das Topic und den Nachrichteninhalt erwartet. Optional kann noch die Retain-Markierung gesetzt werden, dann ist als drittes Argument `true` zu übergeben.
9. Die hier deklarierte Funktion wird von der Bibliothek aufgerufen, wenn eine Message vom Broker (aufgrund einer Subscription) empfangen wurde.
10. Das Topic wird in diesem Beispiel nicht näher ausgewertet, sondern nur zur Kontrolle am seriellen Monitor ausgegeben. Wenn nur ein Topic abonniert wurde, sendet der Broker ohnehin keine Nachrichten anderer Topics.
11. Der empfangene Nachrichteninhalt (Nutzdaten) liegt als Zeichenkette vor, die Funktion `atoi()` (ASCII to Integer) wandelt sie in eine Ganzzahl um, welche wir zur Ansteuerung der LED per PWM benötigen.
12. Diese Funktion dient dem erstmaligen oder wiederholten Herstellen der Verbindung zum MQTT-Broker.

13. Solange die Statusabfrage über die Funktion `connected() false` zurückgibt,
14. wird mittels der Funktion `connect()` die Herstellung einer Verbindung versucht. Das übergebene Argument ist der Name des Objektes, welches die Netzwerkverbindung repräsentiert.
15. Ist die Verbindung hergestellt, wird die Funktion `subscribeReceive()` als „Ansprechpartner“ für eingehende Messages an die Bibliothek übergeben
16. und das Topic „Arduino/LED“ abonniert.

Für den ESP8266 (NodeMCU, D1mini) sind nur geringfügige Anpassungen notwendig:

```
#define LEDPIN D3
#define ANALOGPIN A0

#define NETZWERKNAME "MeinWLAN"
#define PASSWORT "M.e.i.n)P.a-s/s-wo{rt"
#define MQTT_BROKER "192.168.0.10"

#include "ESP8266WiFi.h"
#include "PubSubClient.h"

WiFiClient espClient;
PubSubClient mqttClient(espClient);

long Timer = 0;
char Zwischenspeicher[50];

void setup()
{
  pinMode(LEDPIN, OUTPUT);
  Serial.begin(115200);
  delay(100);
  Serial.println();
  Serial.print("Verbinde mit WLAN: ");
  Serial.println(NETZWERKNAME);

  WiFi.begin(NETZWERKNAME, PASSWORT);

  while (WiFi.status() != WL_CONNECTED)                    // (1)
```

```
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Erfolgreich. Eigene IP-Adresse: ");
  Serial.println(WiFi.localIP());

  mqttClient.setServer(MQTT_BROKER, 1883);
}

void loop()
{
  if (!mqttClient.connected())
    reconnect();

  mqttClient.loop();

  if(millis() > Timer)
  {
    String Analogwert = String(analogRead(A0));
    Analogwert.toCharArray(Zwischenspeicher, 20);
    mqttClient.publish("Arduino/Analogwert",  Zwischenspeicher);

    Serial.println("Wert published: " + Analogwert);
    Timer = millis() + 5000;
  }
}

void subscribeReceive(char* Topic, byte* Nutzdaten,
unsigned int Laenge)
{
  Serial.println("### Empfangen ###");
  Serial.print("Topic: ");
  Serial.println(Topic);

  Serial.print("Nutzdaten: ");
  for(int i = 0; i < Laenge; i ++)
    Serial.print(char(Nutzdaten[i]));
  Serial.println("");

  byte Helligkeit = (Nutzdaten[0]-'0')*100
                  + (Nutzdaten[1]-'0')*10
                  + (Nutzdaten[2]-'0');                    // (2)
  Serial.print("Helligkeit: "); Serial.println(Helligkeit);
  analogWrite(LEDPIN, Helligkeit);

```

```
76    Serial.println("-----------------");
77  }
78
79  void reconnect()
80  {
81    while (!mqttClient.connected())
82    {
83      Serial.print("Verbinde mit MQTT.");
84
85      if (!mqttClient.connect("ESP8266Client"))
86      {
87        Serial.print("Fehler: ");
88        Serial.print(mqttClient.state());
89        Serial.println(" Neuer Versuch in 5 Sekunden.");
90        delay(5000);
91      }
92    }
93    mqttClient.setCallback(subscribeReceive);
94    mqttClient.subscribe("Arduino/LED");
95  }
```

1. Wie bereits beim Webserver-Beispiel wird hier so lange gewartet, bis die WLAN-Verbindung steht.
2. Leider kann die im ESP8266-Compiler implementierte `atoi()`-Funktion keine Zeichenketten umwandeln, welche als `byte`-Array (statt `char`-Array) vorliegen. Die PubSubClient.h-Bibliothek gibt dies jedoch so vor. Kürzester Ausweg ist hier die manuelle Umwandlung der Zeichenkette, wie wir es bereits beim Processing-Praxisprojekt (Kapitel 14.4) durchgeführt hatten.

Für den ESP32 ergeben sich zusätzliche Anpassungen dadurch, dass der aktuelle Compiler leider noch nicht das einfache Einlesen von Analogpins und die einfache PWM-Ausgabe unterstützt:

```
1  #define LEDPIN 4                              // = GPIO 4
2  #define ANALOGPIN ADC1_CHANNEL_4              // = GPIO 32
3
4  #define PASSWORT "M.e.i.n)P.a-s/s-wo{rt"
5  #define MQTT_BROKER "192.168.0.10"
6  #define MQTT_BROKER "192.168.0.10"
7
8  #include "WiFi.h"
```

```
#include "driver/adc.h"                                          // (1)
#include "PubSubClient.h"

WiFiClient espClient;
PubSubClient mqttClient(espClient);

long Timer = 0;
char Zwischenspeicher[50];

void setup()
{
  pinMode(LEDPIN, OUTPUT);
  Serial.begin(115200);
  delay(100);
  Serial.println();
  Serial.print("Verbinde mit WLAN: ");
  Serial.println(NETZWERKNAME);

  WiFi.begin(NETZWERKNAME, PASSWORT);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Erfolgreich. Eigene IP-Adresse: ");
  Serial.println(WiFi.localIP());

  mqttClient.setServer(MQTT_BROKER, 1883);

  adc1_config_width(ADC_WIDTH_BIT_10);                           // (2)
  adc1_config_channel_atten(ANALOGPIN,ADC_ATTEN_DB_11);

  ledcSetup(0, 5000, 8);                                         // (3)
  ledcAttachPin(LEDPIN, 0);                                      // (4)
}

void loop()
{
  if (!mqttClient.connected())
    reconnect();

  mqttClient.loop();

  if(millis() > Timer)
  {
    String Analogwert = String(adc1_get_raw(ANALOGPIN));
```

```
    Analogwert.toCharArray(Zwischenspeicher, 20);
    mqttClient.publish("Arduino/Analogwert",  Zwischenspeicher);

    Serial.println("Wert published: " + Analogwert);
    Timer = millis() + 5000;
  }
}

void subscribeReceive(char* Topic, byte* Nutzdaten,
unsigned int Laenge)
{
  Serial.println("### Empfangen ###");
  Serial.print("Topic: ");
  Serial.println(Topic);

  Serial.print("Nutzdaten: ");
  for(int i = 0; i < Laenge; i ++)
    Serial.print(char(Nutzdaten[i]));
  Serial.println("");

  byte Helligkeit = (Nutzdaten[0]-'0')*100
                  + (Nutzdaten[1]-'0')*10
                  + (Nutzdaten[2]-'0');

  Serial.print("Helligkeit: "); Serial.println(Helligkeit);
  ledcWrite(PWM_CHANNEL, Helligkeit);                    // (5)

  Serial.println("");
  Serial.println("-----------------");
}

void reconnect()
{
  while (!mqttClient.connected())
  {
    Serial.print("Verbinde mit MQTT.");

    if (!mqttClient.connect("ESP8266Client"))
    {
      Serial.print("Fehler: ");
      Serial.print(mqttClient.state());
      Serial.println(" Neuer Versuch in 5 Sekunden.");
      delay(5000);
    }
  }
  mqttClient.setCallback(subscribeReceive);
  mqttClient.subscribe("Arduino/LED");
}
```

1. Die zusätzliche Bibliothek für den Analog-Digital-Wandler kennen wir bereits vom Webserver-Beispiel,
2. genau wie auch die zugehörigen Konfigurationsbefehle.
3. Zur Nutzung der Pulsweitenmodulation stehen 16 Kanäle (vom Hersteller als *LED-Controller* bezeichnet) zur Verfügung. Dafür muss über die Funktion `ledcSetup()` einem Kanal (0 ... 15, erstes Argument) eine PWM-Frequenz (zweites Argument) sowie eine Auflösung (hier: 8 Bit, drittes Argument) zugewiesen werden.
4. Anschließend wird dieser PWM-Kanal einem Ausgangspin zugewiesen.
5. Die Funktion `ledcWrite()` entspricht in diesem Kontext der bekannten Funktion `analogWrite()`.

Nach dem Hochladen des Sketches wollen wir nun mit dem einfachen Client MQTT.fx am Computer testen, ob unser Projekt funktioniert. Unabhängig davon, ob Sie diesen Versuch am Arduino mit Ethernet-Shield oder mit dem ESP32 beziehungsweise ESP8266 durchführen, sind die folgenden Schritte nun wieder gleich.

16.3.4 Starten des Brokers

Starten Sie zunächst den bereits installierten MQTT-Broker Mosquitto, indem Sie den Installationsordner (C:\Programme\mosquitto) öffnen und darin die Datei mosquitto.exe starten. Es öffnet sich ein leeres Fenster der Eingabeaufforderung.

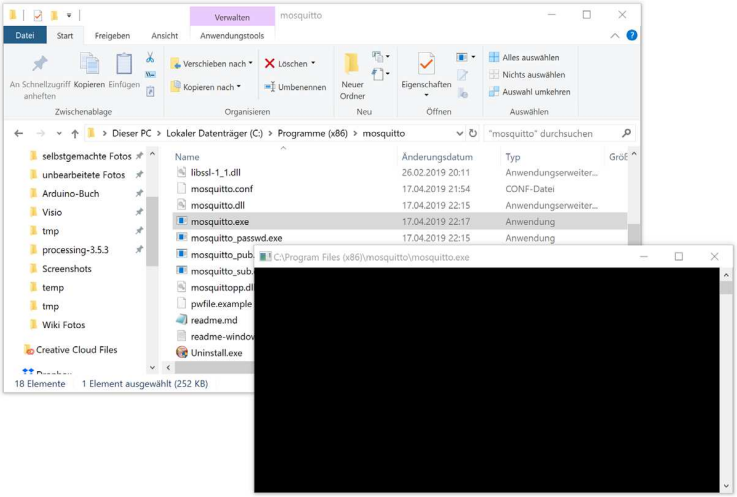


Abb. 16.40 Der MQTT-Broker muss manuell über eine Datei in seinem Installationsverzeichnis gestartet werden

Das ist alles, was wir vom Broker sehen – er läuft ansonsten komplett im Hintergrund. Sie können dieses Fenster minimieren, sollten es aber nicht schließen.

Nun öffnen Sie die Client-Software MQTT.fx über den Eintrag im Startmenü:

Abb. 16.41 Am schnellsten finden Sie das Programm in der Rubrik „Zuletzt hinzugefügt“

Auch diesem Client müssen wir mitteilen, mit welchem Broker er sich verbinden soll – selbst wenn sich dieser, wie in unserem Fall, auf demselben Computer befindet. Der Broker wird über die Auswahlliste oben links ausgewählt. Beim ersten Start müssen wir einen neuen Eintrag anlegen, da unser Mosquitto-Broker noch nicht hinterlegt ist. Klicken Sie dazu auf das Zahnrad und übernehmen Sie die Einstellungen aus der folgenden Abbildung:

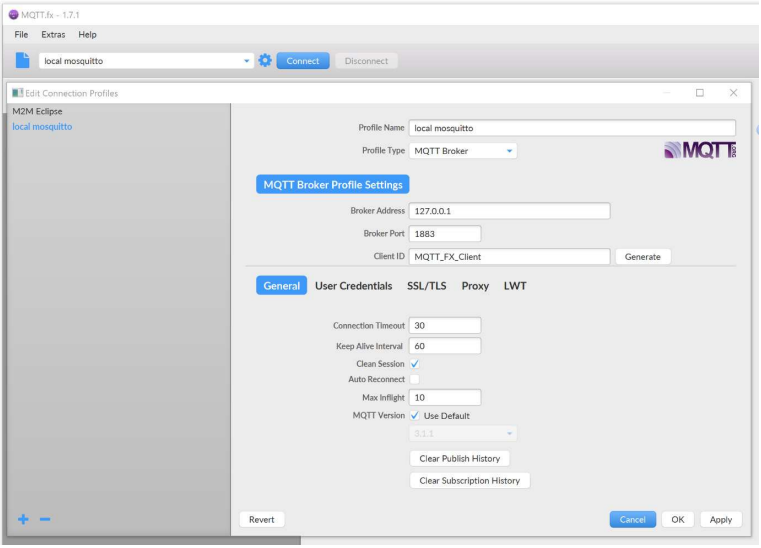


Abb. 16.42 Über das Dialogfeld können Sie einen neuen Broker anlegen

Wichtig sind hierbei der Server (127.0.0.1, dies ist eine spezielle „symbolische“ IP-Adresse, welche immer den eigenen Computer meint) sowie der MQTT-typische Port 1883. Die übrigen Einstellungen können mit ihren Standardwerten übernommen werden. Bestätigen Sie die Eintragungen mit OK.

Nun kann der gerade angelegte Broker ausgewählt werden, die Schaltfläche *Connect* stellt dann eine Verbindung her. Die erfolgreiche Verbindung wird durch einen grünen Punkt oben rechts angezeigt.

Klicken Sie nun auf *Subscribe* und abonnieren Sie alle Topics, indem Sie eine Raute (#) in das Eingabefeld tippen und die Schaltfläche daneben anklicken.

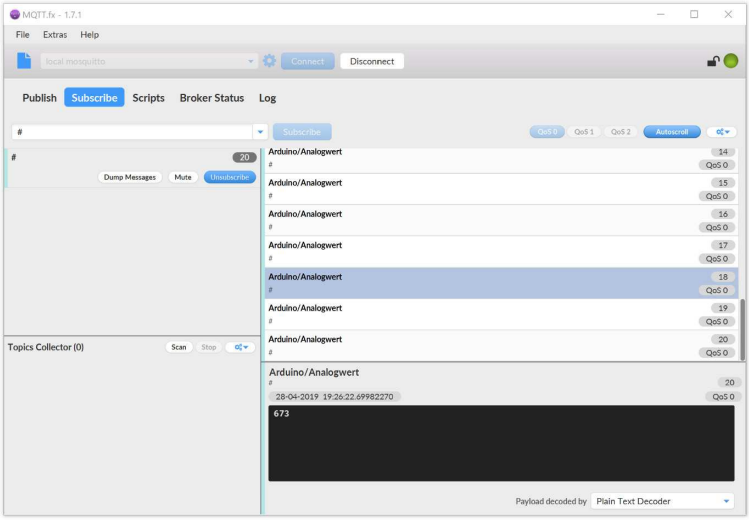


Abb. 16.43 Wir nutzen die Raute, um uns alle Topics senden zu lassen. Im linken Feld kann dieses Abonnement auch pausiert (Mute) oder storniert (Unsubscribe) werden.

Nun sollten Sie feststellen, dass der Arduino (oder ESP) im 5-Sekunden-Takt Messages mit dem aktuellen Analogwert sendet. Ist dies nicht der Fall, starten Sie den Mikrocontroller zunächst neu und prüfen Sie bei weiteren Problemen, ob das hinterlegte WLAN-Passwort korrekt ist und die richtige IP des Brokers (Computer) im Sketch hinterlegt wurde.

Nun wollen wir selbst Daten an den Mikrocontroller senden. Klicken Sie auf den Bereich *Publish*, wählen Sie als Topic „Arduino/LED“[10] und

10 Unabhängig davon, ob der Arduino oder ein ESP-Board verwendet wird, haben wir in allen Sketches den willkürlich gewählten Topic-Namen bei „Arduino“ belassen, um eine einfachere Austauschbarkeit zu gewährleisten.

senden Sie als Nachricht eine Zahl zwischen 0 und 255. Nach dem Klick auf die *Publish*-Schaltfläche sollte die LED den entsprechenden Helligkeitswert annehmen.

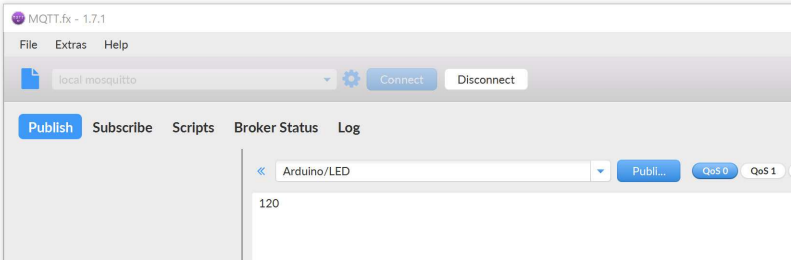


Abb. 16.44 Das Senden (Publishen) von Nachrichten ist ebenfalls sehr einfach

All diese Vorgänge können Sie auch zeitgleich am seriellen Monitor nachvollziehen, denn wir hatten im Sketch ja einige `Serial.print()`-Befehle eingebaut.

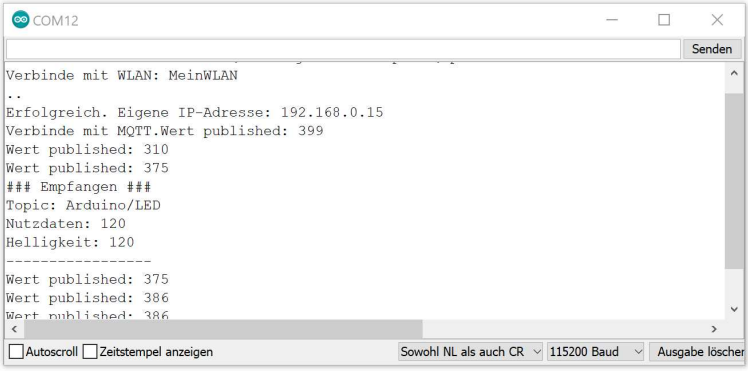


Abb. 16.45 Der serielle Monitor gibt uns Einblick in die Aktivitäten des Mikrocontrollers

Die Kommunikation per MQTT funktioniert also, jedoch ist die Steuerung unseres kleinen IoT-Moduls darüber noch etwas umständlich. Daher wollen wir im folgenden Abschnitt eine Visualisierung erstellen.

16.3.5 Visualisierung mit Node-RED

Die MQTT.fx-Software benötigen wir nun nicht mehr, sie kann geschlossen werden. Jedoch sollte sichergestellt sein, dass der Mosquitto-Broker weiterhin im Hintergrund läuft. Andernfalls öffnen Sie ihn einfach neu.

Öffnen Sie nun ein weiteres Fenster der Windows-Eingabeaufforderung (cmd) und tippen Sie folgendes ein:

```
1 node-red
```

Nach dem Druck auf die Enter-Taste startet der Node-RED-Server. Dieses Fenster muss nun ebenfalls im Hintergrund geöffnet bleiben, kann aber minimiert werden.

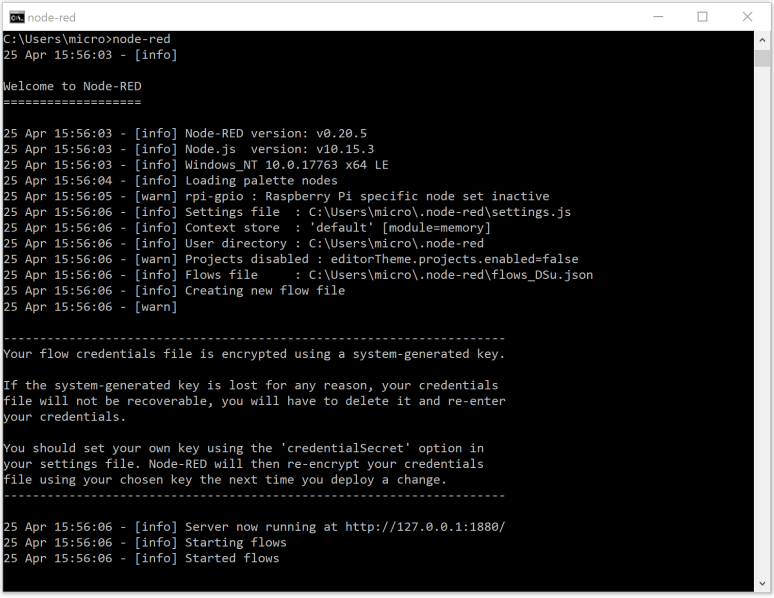


Abb. 16.46 Die Node-RED-Software zeigt beim Start einige Statusmeldungen

Öffnen Sie nun in einem Webbrowser die Adresse

```
1 127.0.0.1:1880
```

Es öffnet sich die Oberfläche von Node-RED (welches standardmäßig Port 1880 nutzt) mit einem leeren *Flow*. Flows repräsentieren das „Programm“ und können analog zu einem Sketch in der Arduino IDE verstanden werden. Allerdings arbeitet Node-RED nicht (vordergründig) mit Programmcode, sondern stellt Zusammenhänge visuell dar.

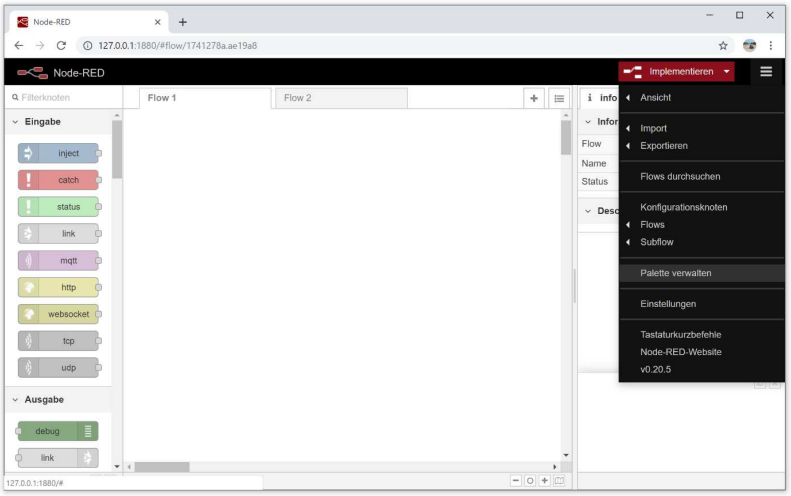


Abb. 16.47 Die Node-RED-Software wird komplett über den Webbrowser gesteuert

Bevor wir dies selbst ausprobieren, müssen wir noch ein Modul (ähnlich einer Bibliothek bei Arduino) installieren. Öffnen Sie dazu mit einem Klick auf die drei Balken oben rechts das Menü und klicken Sie auf den Unterpunkt *Palette verwalten*. Tippen Sie „dashboard“ in das Suchfeld ein und installieren Sie das Modul *node-red-dashboard*.

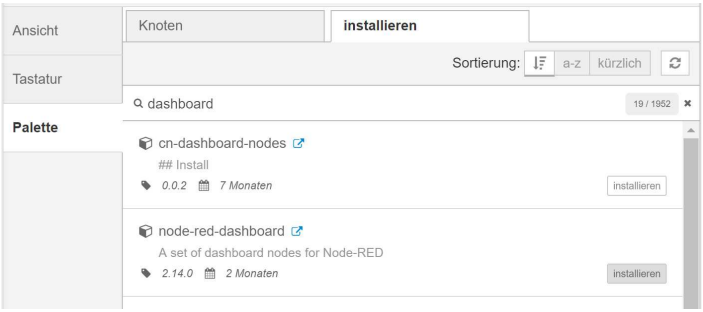


Abb. 16.48 Nach dem Klick auf „installieren“ wird die Palette (das Modul) heruntergeladen und installiert

Nun kann es losgehen. Unser Ziel ist es, eine Website zu erstellen, auf der wir komfortabel die LED-Helligkeit steuern und den eingelesenen Analogwert einsehen können. Zum besseren Verständnis sei hier das Ergebnis schon einmal vorgegriffen:

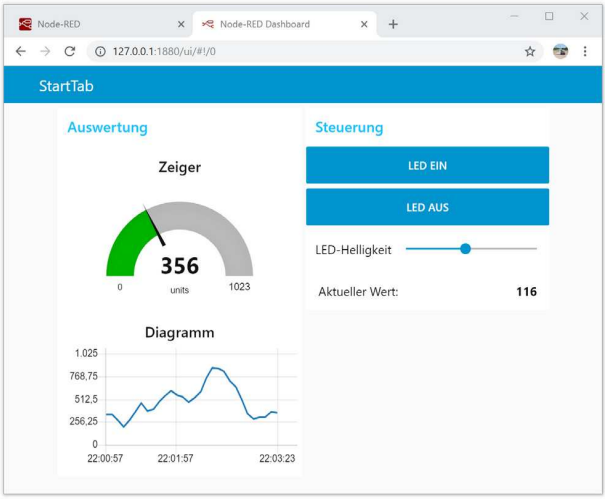


Abb. 16.49 Unser angestrebtes Ergebnis: In dieser Nutzeroberfläche wird links der (von Potentiometer erhaltene) Analogwert dargestellt, rechts besteht die Möglichkeit, die LED zu steuern

Um dies zu erreichen, fertigen Sie den folgenden Flow an, indem Sie zunächst die verwendeten Elemente (“Knoten“) aus der linken Leiste auf die leere Fläche ziehen und anordnen. Achten Sie bei den Knoten auf die korrekten Symbole.

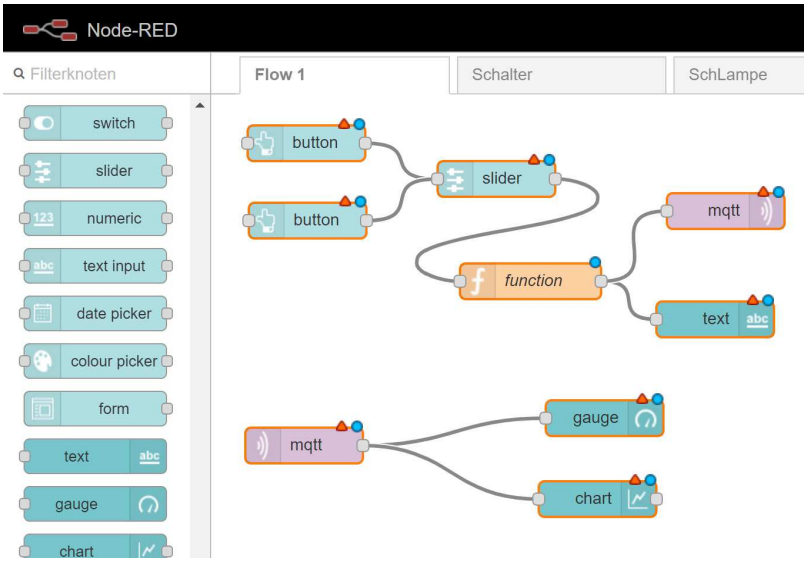


Abb. 16.50 Setzen Sie zunächst die Knoten nach diesem Schema zusammen

Die türkisfarbenen Knoten finden Sie in der Rubrik *dashboard* (2x button, 1x slider, 1x text, 1x gauge, 1x chart). Die beiden pinken Elemente sind die mqtt-Knoten aus den Rubriken *Ausgabe* (oben) und *Eingabe* (unten). Das orangene Element ist der `function`-Knoten aus der Rubrik *Funktion*.

Wenn Sie alle Elemente angeordnet haben, verbinden Sie diese wie in der Abbildung gezeigt. Anschlüsse an der rechten Seite eines Knotens sind Ausgänge, hier können Daten abgegriffen und auf einen Eingang (linke Seite) eines anderen Knotens geleitet werden.

Nun müssen den einzelnen Knoten noch gewisse Einstellungen zugewiesen werden. Wir beginnen mit dem *button* oben links. Er soll für das Einschalten der LED verantwortlich sein, indem er den Wert 255 an den *slider* schickt, welcher diesen über den Funktionsknoten an den MQTT-Sender (pinker Block oben rechts) weiterleitet. Doppelklicken Sie auf den *button*-Knoten oben links und übernehmen Sie folgende Werte:

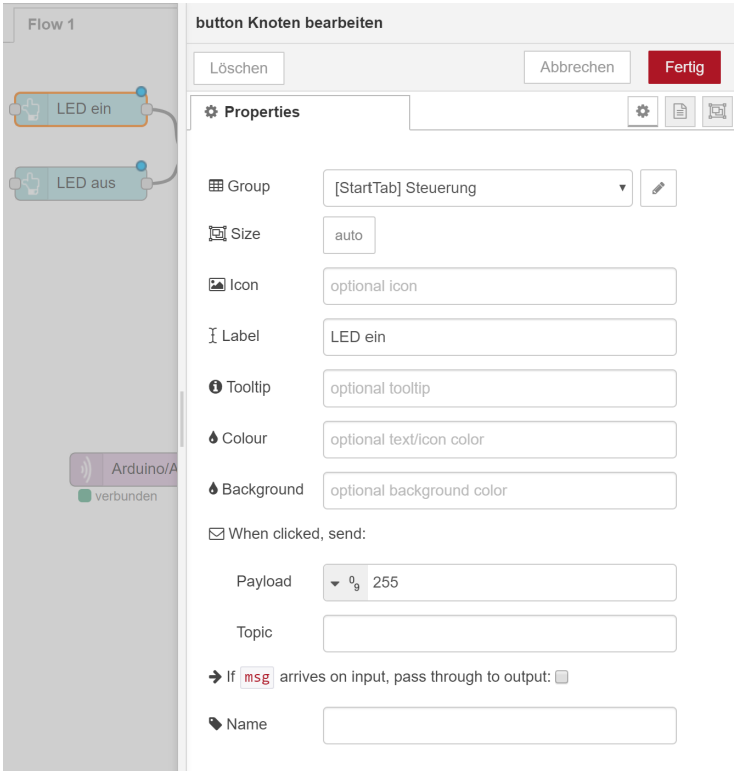


Abb. 16.51 Die meisten Einstellungen sind optional

Die Darstellung der einzelnen Schaltflächen und Elemente auf der späteren Website erfolgt in Gruppen (Groups). Wir wollen die LED-Steuerung von der Auswertung des Analogwertes durch die Zuordnung zu verschiedenen Groups unterscheiden. Legen Sie daher über

das kleine Stift-Symbol neben dem Group-Eingabefeld die Gruppe "Steuerung" an.

Haben Sie auch Label und Payload (Nutzdaten) übernommen, schließen Sie den Dialog mit der *Fertig*-Schaltfläche. Die Parameter des zweiten *buttons* unterscheiden sich nur geringfügig, übertragen Sie auch diese:

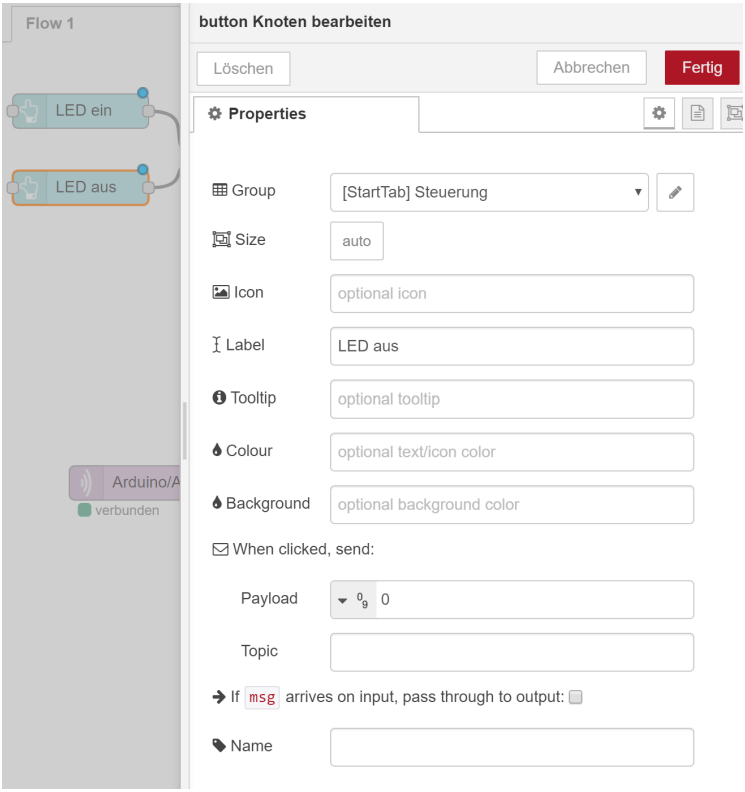


Abb. 16.52 Die Werte des zweiten Buttons unterscheiden sich nur minimal, wesentlich ist vor allem der veränderte Payload-Wert, da die LED über diesen `button` ausgeschaltet (0) werden soll

Weiter geht es mit dem Schieberegler (slider):

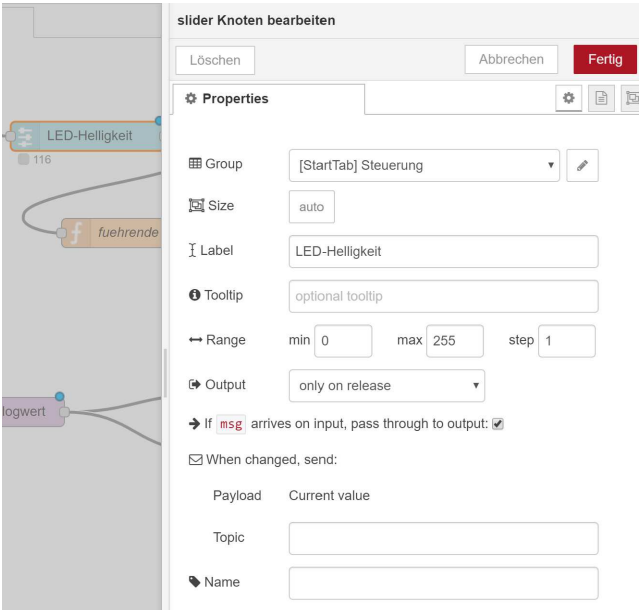


Abb. 16.53 Beim Schieberegler ist die Festlegung des Wertebereiches (Range) wichtig

Eine Besonderheit bildet der *function*-Knoten. Wir benötigen ihn, um dem Client immer einen dreistelligen Helligkeitswert zu senden, gegebenenfalls also mit führenden Nullen. Dies ist nötig, da wir im Sketch für die ESP-Mikrocontroller die Funktion `atoi()` nicht verwenden konnten, sondern die Umwandlung von einer Zeichenkette in einen Zahlenwert manuell vorgenommen haben. Die von uns in diesen Sketchen genutzte Methode funktioniert jedoch nur bei Zahlen mit stets der gleichen Länge.

Daher schicken wir unsere Nutzdaten (den am Regler oder über die Buttons eingestellten Zahlenwert) durch eine JavaScript-Funktion, welche gegebenenfalls führende Nullen anfügt. Die Notation in JavaScript weicht etwas von der uns bekannten Schreibweise in C++ ab, dennoch werden Sie sie nachvollziehen können. Übernehmen Sie einfach das folgende Skript:

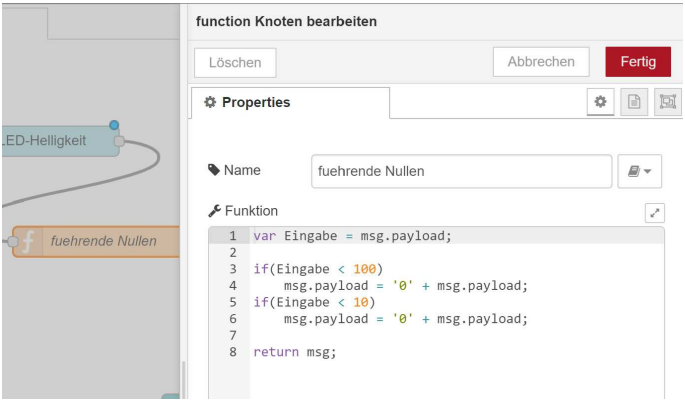


Abb. 16.54 Die Skriptsprache JavaScript ähnelt in ihren Ansätzen den Arduino-Sketches

Als Nächstes konfigurieren wir den *mqtt*-Knoten, welcher den Helligkeitswert schließlich an den Broker publisht – so wie wir es im vorangegangenen Abschnitt manuell über den *MQTT.fx*-Client getan haben.

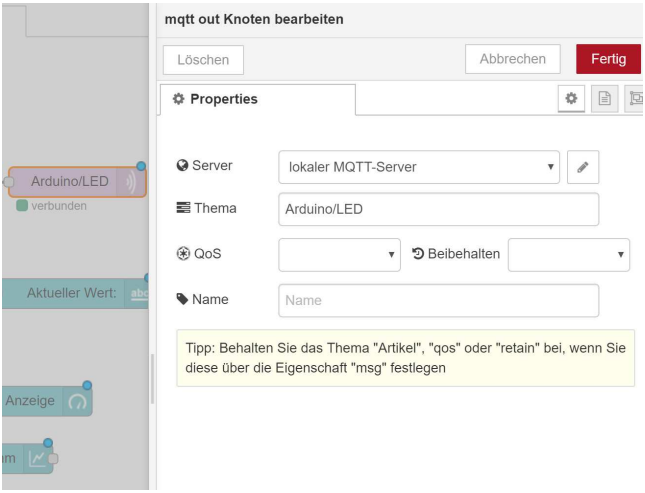


Abb. 16.55 Im *mqtt-out*-Knoten wird auch das Topic festgelegt, unter welchem die an den Knoten übertragenen Nutzdaten veröffentlicht werden sollen

Der Server muss zunächst noch angelegt werden. Klicken Sie dazu auf das Stift-Symbol und tragen Sie folgende Daten ein:

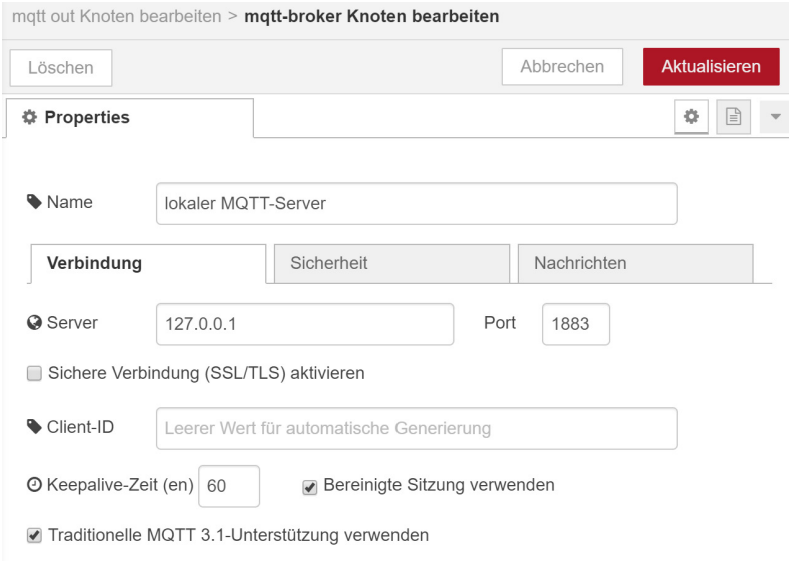


Abb. 16.56 Der Broker liegt auf dem eigenen Computer, daher wird wieder die auf sich selbst zeigende IP 127.0.0.1 verwendet

Parallel soll der eingestellte Helligkeitswert auch noch auf der Website angezeigt werden (zur Kontrolle). Dafür verwenden wir den *text*-Knoten:

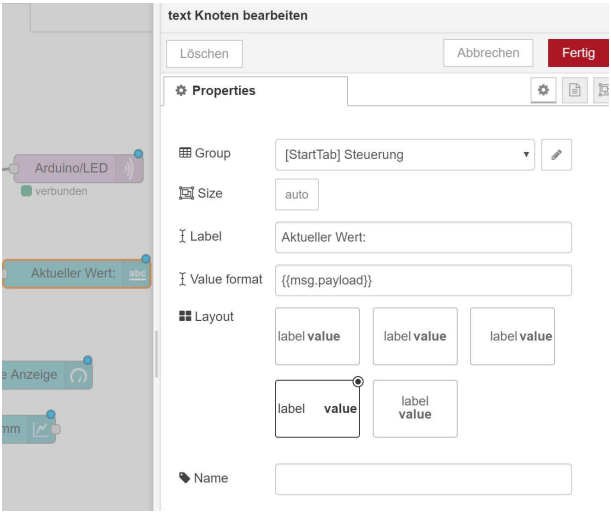


Abb. 16.57 Der *text*-Knoten dient nur der Kontrolle

Zusätzlich zu den bisherigen Knoten gibt es im unteren Teil der Flow-Grafik noch drei weitere Elemente, welche für den Analogwert zuständig sind. Dieser wird von einem *mqtt-in*-Knoten empfangen, er abonniert also das Topic „Arduino/Analogwert“:

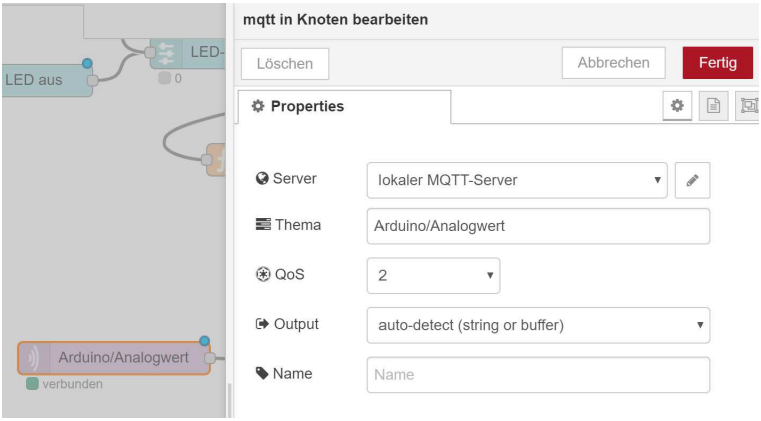


Abb. 16.58 Der vorhin angelegte Server steht nun schon zur Auswahl bereit

Den Empfangswert leitet er weiter an ein analoges Zeigerinstrument (*gauge*-Knoten). Die Umwandlung von einer Zeichenkette in einen Zahlenwert wird bei Node-RED automatisch im Hintergrund erledigt.

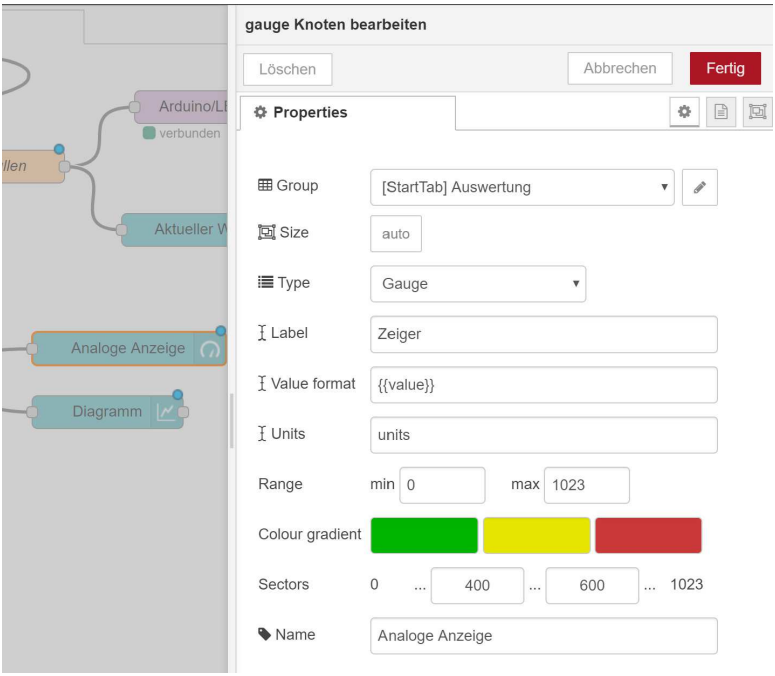


Abb. 16.59 Der *gauge*-Knoten bietet optional die Möglichkeit, bestimmte Wertebereiche einzufärben, um beispielsweise auf kritische Temperaturen hinzuweisen.

Wir nutzen für diesen Knoten eine neu zu erstellende Group „Auswertung". Parallel wird der Wert auch an einen *chart*-Knoten weitergeleitet, welcher eine komfortable Diagrammauswertung der Messdaten ermöglicht.

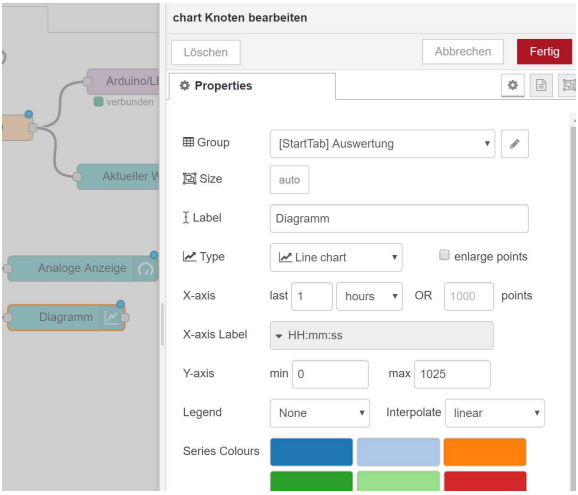


Abb. 16.60 Der *chart*-Knoten bietet zahlreiche Formatierungsmöglichkeiten für die erstellten Diagramme. Wichtig sind vor allem die korrekten Minimal- und Maximalwerte für die Y-Achse.

Haben Sie alle Knoten konfiguriert, klicken Sie oben rechts auf *Implementieren*. Dadurch werden alle Änderungen gespeichert.

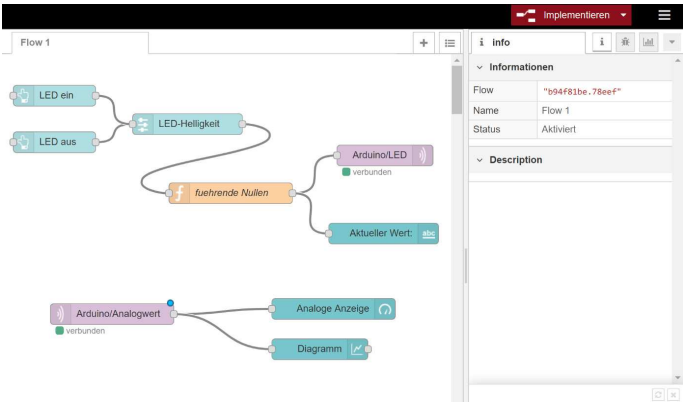


Abb. 16.61 Die Implementieren-Schaltfläche überträgt die Daten in die Server-Software

Um das Ergebnis zu sehen, können Sie in einem neuen Browser-Tab die Adresse

```
1 127.0.0.1:1880/ui
```

(für *User Interface*) aufrufen. Nun sollte die aus *Abb. 16.49* bekannte Nutzeroberfläche angezeigt werden. Eventuell weicht die Reihenfolge der Anordnung ab, dies kann auf Wunsch unter *Dashboard -> Layout* umgestellt werden.

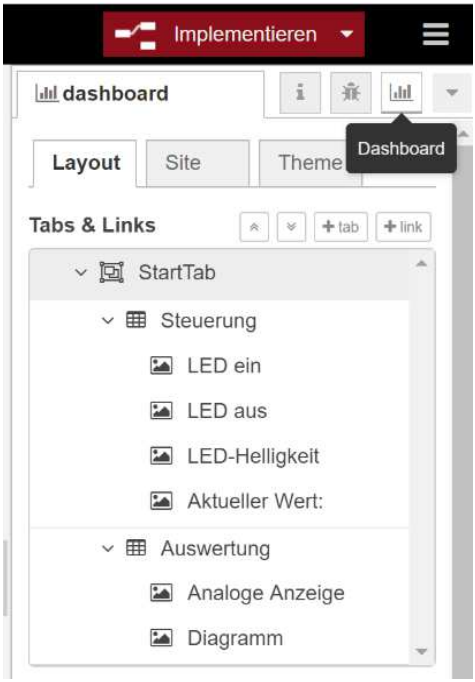


Abb. 16.62 Die Anordnung der Elemente auf der Nutzeroberfläche lässt sich ändern

Sie können diese User-Interface-Website sogar von Ihrem Smartphone aufrufen, wenn es sich per WLAN im selben Netzwerk befindet. Dazu verwenden Sie dann die IP des Computers, in unserem Beispiel also:

```
1 192.168.0.10:1880/ui
```

Auch die gleichzeitige Nutzung der Oberfläche per Smartphone und Computer (oder über weitere Geräte) ist kein Problem.

An diesem einfachen Beispiel können Sie nun sehen, wie mächtig die MQTT-Plattform ist. Es ist problemlos möglich, jede Ecke eines Hauses mit Temperatursensoren auszustatten und die Werte auf einem Tablet zu verfolgen oder die Lichtstimmung im Wohnzimmer durch einen Mikrocontroller in Abhängigkeit von der Außenwetterlage zu ändern.

Die Vorteile von MQTT sind besonders in der Welt des Internet der Dinge sehr günstig:

- Lediglich die Adresse des Brokers muss allen bekannt sein, die einzelnen Clients brauchen über ihre gegenseitigen IPs nicht informiert zu werden.
- Clients können jederzeit während des laufenden Betriebs hinzugefügt oder entfernt werden.
- Dem Programmierer sind viele Freiheiten überlassen, was die Struktur der übertragenen Daten sowie deren Häufigkeit betrifft.
- Der Datenverkehr ist über Clients wie MQTT.fx einfach einsehbar, um Programmierfehler zu finden.

In unserem Beispiel müsste der Computer zu jeder Zeit laufen, wenn eine Datenübertragung stattfinden soll. Bei dauerhaften Projekten wird der MQTT-Broker (und auch die Node-RED-Software samt Node.js-Basis) daher oft auf einen Mini-Computer (beispielsweise Raspberry Pi) ausgelagert. Es ist auch möglich, den Broker auf einem Server im Internet laufen zu lassen. Allerdings sind dafür unbedingt Sicherheitsmaßnahmen gegen unberechtigte Zugriffe (Verschlüsselung, Passwortschutz) zu treffen, welche von Einsteigern oft unterschätzt werden. Auch in Heimnetzen sollten Sie sich darüber im Klaren sein, dass unter Umständen (beispielsweise, wenn sich ein mit Schadsoftware infiziertes Smartphone im Heim-WLAN befindet,) Angreifer den MQTT-Datenverkehr mitlesen können, falls der Server (wie in unserem Beispiel)

keine ausreichende Verschlüsselung nutzt. Und selbst vermeintlich unverfängliche Daten (wie die Raumtemperatur) könnten kriminelle Personen darüber informieren, zu welchen Zeiten Sie nicht zu Hause sind – und so einen Einbruch begünstigen.

Sollten Sie sich weitergehend für dieses Thema interessieren, sei der Leitfaden des heise-Verlages[11] empfohlen, welcher Schritt für Schritt die nötigen Maßnahmen zur Absicherung eines MQTT-Netzes aufzeigt.

11 https://www.heise.de/developer/artikel/Sichere-IoT-Kommunikation-mit-MQTT-Teil-1-Grundlagen-3645209.html

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 17

Arduino Clones, minimaler Arduino

17.1 Clones

Da die Arduino-Plattform als Open-Source-Projekt für jeden offen steht, haben einige Hersteller baugleiche Platinen auf den Markt gebracht. Sie werden gemäß dem englischen Wort für Nachbauten auch als *Clones* bezeichnet.

Prinzipiell eignen sich diese ebenso gut für Hobby-Projekte. Ihr großer Vorteil liegt im vielfach günstigeren Preis, wobei dieser bei Abnahme großer Stückzahlen sogar noch weiter sinken kann. Zudem bieten einige Module zusätzliche Funktionen wie beispielsweise vormontierte Displays oder Drahtlosadapter auf der Platine.

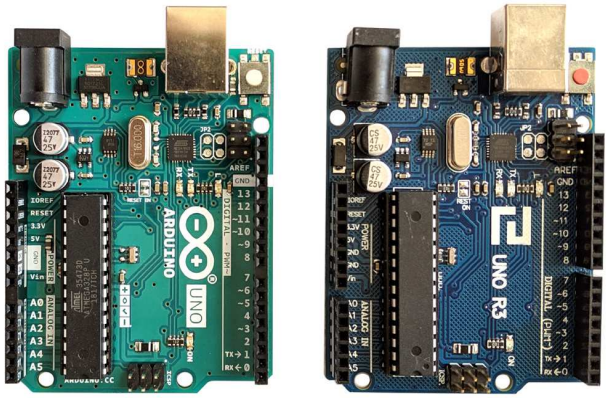


Abb. 17.1 links ein originaler Arduino UNO, rechts ein Nachbau

Es ist jedoch möglich, dass sich die Kopien in Details unterscheiden. So fällt beim folgenden Nachbau des Arduino Nano zunächst die andere

Bauform des ATmega328P-Mikrocontrollers auf – diese hat aber keine Auswirkung auf die Funktion:

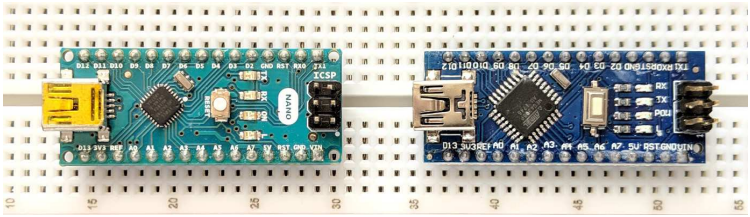


Abb. 17.2 links ein originaler Arduino Nano, rechts ein Nachbau

Ein deutlicherer Unterschied wird ersichtlich, wenn man einen genauen Blick auf die Rückseite wirft: Als USB-zu-seriell-Wandler wird beim Original ein Chip des Herstellers FTDI verwendet – der Nachbau nutzt einen anderen Chip, offensichtlich vom Typ CH340.

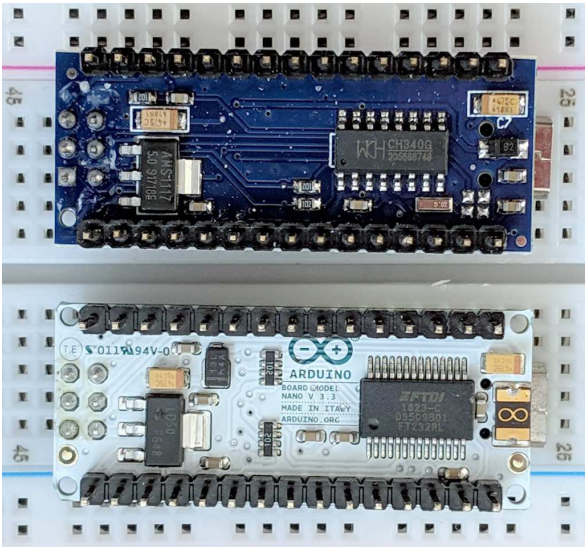


Abb. 17.3 Unten der originale Arduino Nano, oben der Nachbau. Ein relevanter Unterschied besteht in den USB-zu-seriell-Wandlern (jeweils der große IC rechts).

Anders als der Chip des Originals wird der CH340 von den meisten Computern nicht automatisch wie ein serieller Anschluss behandelt. Manche Betriebssysteme ignorieren ihn komplett, andere zeigen einen Hinweis auf nicht erkannte Hardware. Um das Problem zu lösen, kann von Github[1] der benötigte Treiber heruntergeladen und installiert werden.

Besonders bei Direktbestellungen außerhalb der EU ist außerdem zu beachten, dass die europäischen Verbraucherschutzregeln nicht greifen und jeder Kunde als Importeur selbst dafür verantwortlich ist, neben den Zollbestimmungen auch die RoHS-Richtlinien[2] einzuhalten. Günstiger ist es daher, derartige Produkte über deutsche Händler zu beziehen, da hierbei die RoHS-Konformität garantiert wird. Eine Liste mit vorgeschlagenen Bezugsquellen finden Sie im Anhang dieses Buches.

17.2 Minimaler Arduino

Mit Blick auf die Clones liegt die Frage nahe, ob man auch selbst eine Arduino-Platine nachbauen kann. Tatsächlich spricht nichts dagegen. Möchte man ein Projekt aus dem Experimentierstatus in den Praxiseinsatz überführen, bietet diese Taktik sogar Kosten- und Platzvorteile.

Alles, was Sie dafür benötigen, ist der Arduino-Mikrocontroller (ATmega328 oder ATmega328P[3]), welcher im Handel für deutlich unter 5 Euro pro Stück erhältlich ist. Es gibt mehrere Bauformen, für Bastel-Einsteiger eignet sich die große DIP-Bauweise (*Dual Inline Package*) mit zwei

1 https://github.com/HobbyComponents/CH340-Drivers
2 *Restriction of Hazardous Substances* – Beschränkung gefährlicher Stoffe (EU-Richtlinen 2002/95/EG und 2011/65/EU)
3 Der Zusatz P weist lediglich darauf hin, dass die überarbeitete Version des Mikrocontrollers effektivere Stromsparfunktionen (Power-Management) besitzt. In der Anwendung gibt es keine Unterschiede.

Anschlussreihen am besten, da sie sowohl für Breadboards als auch für die Platinenmontage geeignet ist.

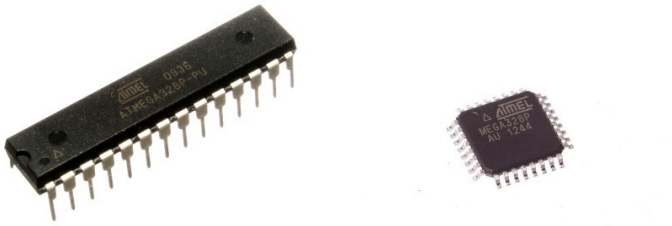


Abb. 17.4 zwei ATmega328P – links in DIP-Bauform, rechts in deutlich kleinerer SMD-Ausführung

In der Serienherstellung werden häufig die kleineren SMD-Bauformen[4] verwendet, welche sich jedoch nur schwer manuell verlöten lassen. Das Innenleben ist unabhängig von der Bauform jeweils gleich.

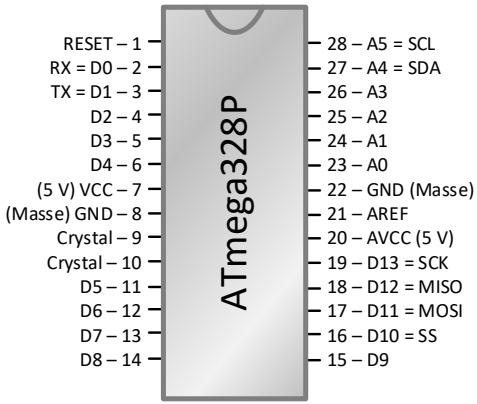


Abb. 17.5 Pinbelegung des ATmega328P in DIP-Bauform. Zur Abgrenzung von den Bauteil-Pinnummern wurde den digitalen Ein- und Ausgangspins ein D vorangestellt.

4 *Surface Mounted Device* – Bei dieser Form werden die Pins nicht durch die Platinen hindurch gesteckt, sondern direkt auf die Oberfläche gelötet.

Der Mikrocontroller verfügt über 28 Pins, von denen die meisten bereits aus unseren Arduino-Versuchen bekannt sind. Die Arduino-Platine verbindet die Ein- und Ausgangspins einfach direkt mit denen des Mikrocontrollers.

Der RESET-Pin funktioniert wie ein digitaler Eingang mit internem Pull-Up-Widerstand. Nur wenn er mit Masse verbunden wird, löst dies einen Neustart aus. Für die Betriebsspannung (VCC – *Voltage* at the *Common Collector*) sind gleich zwei Pins vorgesehen. Zur Vermeidung von gegenseitigen Störungen werden in der Signalverarbeitung traditionell getrennte Betriebsspannungen für digitale (VCC) und analoge Schaltungsteile (AVCC) verwendet. In unseren Anwendungsfällen (ohne hochfrequente oder hochpräzise Signale) ist dies nicht relevant, deshalb sind auch auf der originalen Arduino-Platine einfach beide Pins mit dem gleichen 5-V-Spannungsregler verbunden.

Der AREF-Pin bietet die Möglichkeit, eine abweichende Referenzspannung für die analogen Eingänge zu verwenden, wenn die Betriebsspannung nicht dafür genutzt werden soll (siehe Kapitel 5.3).

Die Bezeichnung der Crystal-Pins weist auf den 16-Megahertz-Quarzkristall hin, welche hier als Taktgeber anzuschließen ist. Wie ein Metronom bestimmt er die Arbeitsgeschwindigkeit des Mikrocontrollers und muss gemäß Spezifikation des Chipherstellers mit zwei 22-Pikofarad-Kondensatoren zur Masse hin stabilisiert werden.

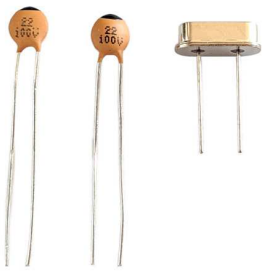


Abb. 17.6 zwei 22-pF-Kondensatoren und ein 16-MHz-Quarzkristall

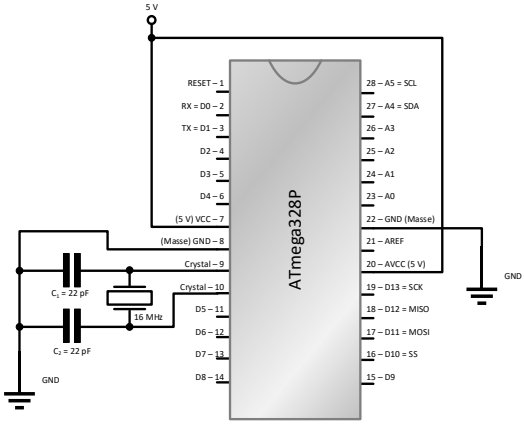


Abb. 17.7 minimale Beschaltung des ATMega328P

Daraus ergibt sich auch schon die minimale Beschaltung de Mikrocontrollers – tatsächlich reichen die Spannungsversorgung sowie der Schwingquarz samt Kondensatoren. Genau genommen könnte dieser Taktgeber sogar entfallen, da der ATmega328P auch über einen internen 8-Megahertz-Taktgeber verfügt. Allerdings muss dafür die Konfiguration des Bootloaders geändert werden. Dieser Eingriff ist nicht über die Arduino IDE möglich und sollte nur von Profis durchgeführt werden, da ein Fehler dazu führen kann, dass der Chip dauerhaft unbenutzbar wird. Wir bleiben also im Folgenden beim externen Takt, welcher zudem nicht nur doppelt so schnell, sondern auch genauer ist.

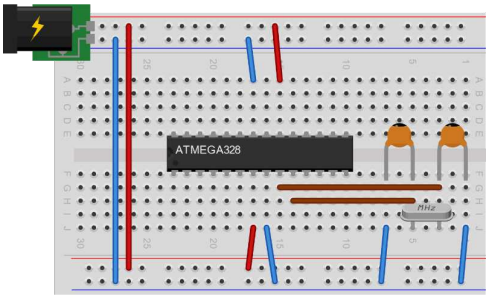


Abb. 17.8 minimale Beschaltung des ATmega328 beziehungsweise ATmega328P

17.3 In-System-Programmer

Die minimale Beschaltung sorgt zwar dafür, dass der Microkontroller läuft – jedoch ist er noch nicht programmiert. Handelt es sich um einen fabrikneuen Chip, müssen wir zudem davon ausgehen, dass auch der Bootloader ("Startprogramm", siehe Kapitel 3.1.3) noch nicht vorhanden ist. Dieser kann ebenfalls über die Arduino IDE auf den Mikrocontroller geladen werden, allerdings nicht mit dem bisherigen Weg über die serielle Schnittstelle.

Stattdessen benötigt man einen ISP (*In-System-Programmer*), manchmal auch ICSP (*In-Circuit-Serial-Programmer*) genannt. Diese Module sind üblicherweise als USB-Adapter (für den 6-Pin-ICSP-Steckplatz) erhältlich und ermöglichen das unmittelbare Schreiben auf den Programmspeicher des Chips. Bei unserer bisher üblichen seriellen Programmierweise hatte uns unbemerkt immer der vorinstallierte Bootloader geholfen.

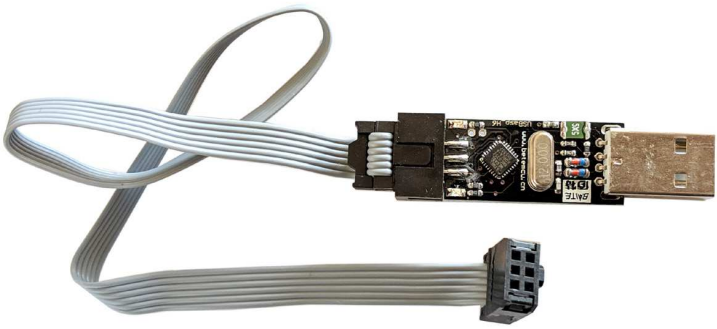


Abb. 17.9 USB-ISP mit dem typischen 6-Pin-Stecker

Verfügt man nicht über einen solchen Adapter, kann man aber auch einfach einen weiteren Arduino benutzen. Im Idealfall besitzt man sogar zwei Arduino-UNO-Platinen: Dann kann man den original-Chip vorübergehend aus einer der Platinen entfernen und durch den fabrikneuen ersetzen:

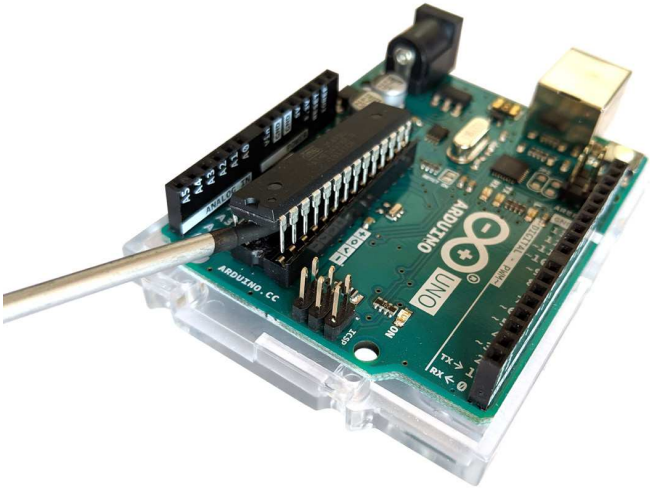


Abb. 17.10 Mit einem Schraubenzieher kann der Mikrocontroller vorsichtig aus seiner Halterung gelöst werden

Anschließend verbindet man beide wie folgt:

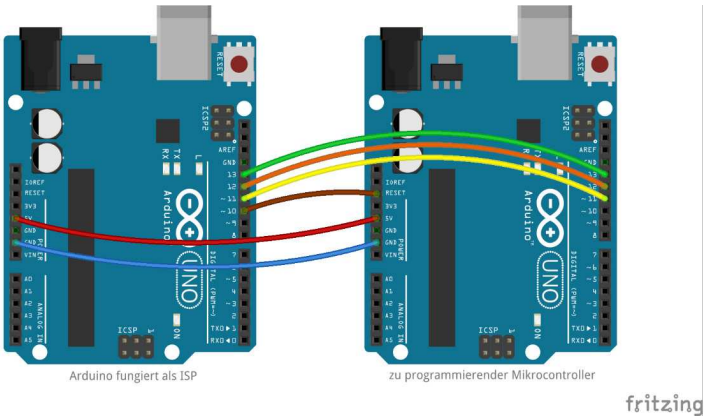


Abb. 17.11 links der (mit dem Computer verbundene) Arduino, welcher als ISP fungiert - rechts die Platine mit dem fabrikneuen Chip, auf welchen nun auch der Bootloader gespielt werden soll

Die Datenübertragung erfolgt über die Pins 11 bis 13. Die Reset-Verbindung ist wichtig, weil das Aufspielen von Daten nur direkt nach einem Neustart möglich ist. Verfügen Sie über nur einen Arduino UNO, kann die Programmierung natürlich auch am Breadboard erfolgen. Die folgende Verdrahtung ist identisch mit der aus *Abb. 17.11*:

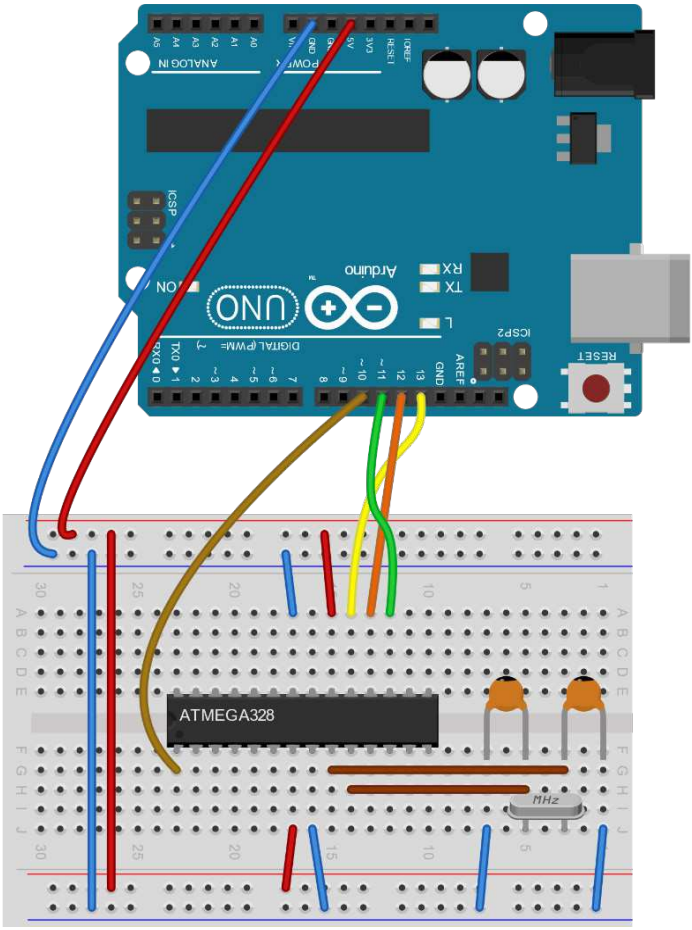


Abb. 17.12 oben der mit dem Computer verbundene Arduino UNO, welcher als ISP fungiert

Mit der Arduino IDE muss nun im ersten Schritt ein spezieller Sketch auf den Arduino geladen werden, welcher als ISP dienen soll. Sie finden diesen unter *Datei -> Beispiele -> 11.ArduinoISP -> ArduinoISP*.

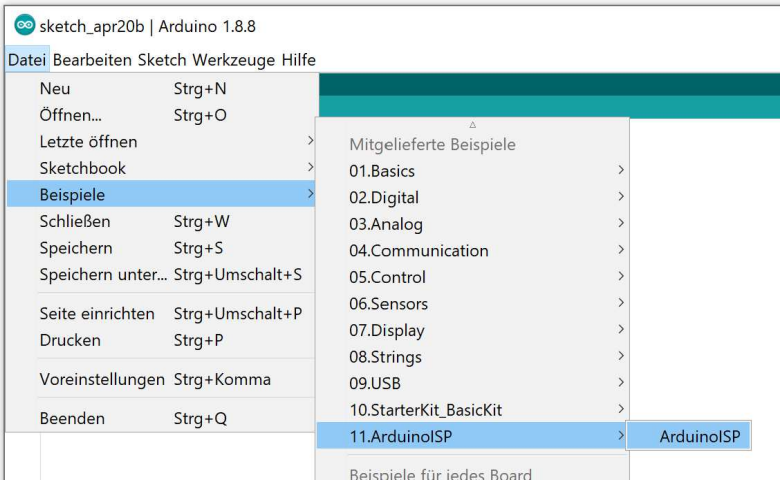


Abb. 17.13 Der für die ISP-Funktion benötigte Sketch befindet sich im Beispiele-Menü

Laden Sie diesen Sketch ohne Änderungen auf den Arduino hoch. Nun ist der Arduino-ISP einsatzbereit. Im nächsten Schritt stellen Sie noch einmal sicher, dass unter *Werkzeuge -> Board: "Arduino/Genuino Uno"* eingestellt und unter *Werkzeuge -> Programmer: "Arduino as ISP"* ausgewählt ist. (Würden Sie stattdessen einen USB-ISP nutzen, müsste dieser hier entsprechend angewählt werden.) Die Wahl des korrekten Boards ist sehr wichtig, da die Software daraus auf einen bestimmten Mikrocontroller-Typ schließt (in unserem Fall ATmega328). Für jeden Typ ist ein zugehöriger Standard-Bootloader hinterlegt. Kommt es hier zu einer Verwechslung, wird womöglich der falsche Bootloader genutzt und der Chip könnte dauerhaft unbrauchbar werden.

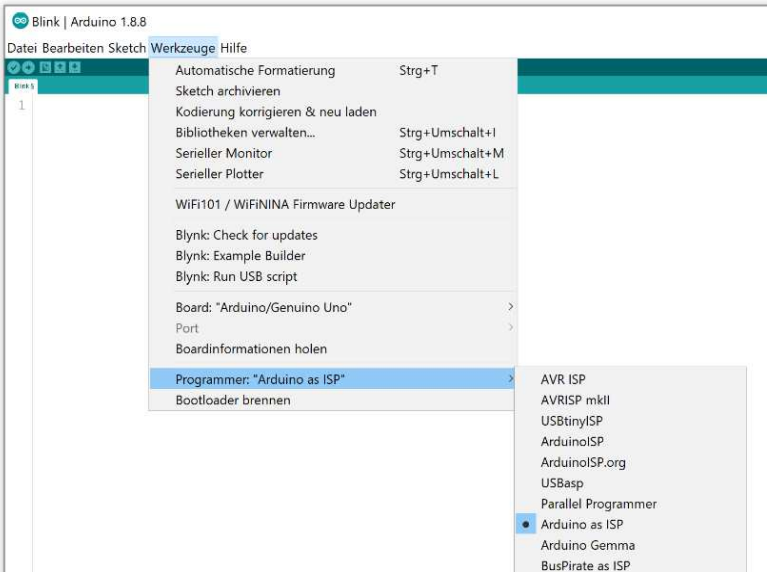


Abb. 17.14 Prüfen Sie die korrekten Einstellungen für „Board“ und „Programmer“

Sind alle Einstellungen korrekt, kann über *Werkzeuge -> Bootloader brennen* der Vorgang gestartet werden. Die Wortwahl „brennen“ hat sich eingebürgert, weil der Bootloader bei älteren Mikrocontrollern nach einmaligem Schreiben nicht mehr geändert werden konnte, er war also quasi „eingebrannt“. Auch heute wird er meist während der gesamten Lebensdauer eines Mikrocontrollers nicht mehr geändert, obwohl es rein technisch möglich ist.

Nun ist der neue Mikrocontroller einsatzbereit. Wenn Sie möchten, können Sie direkt den ersten Sketch hochladen. Nutzen Sie dazu allerdings nicht die normale Hochladen-Funktion, sonst landet das Programm auf dem Hilfs-Arduino, welchen wir als ISP genutzt haben. Klicken Sie stattdessen auf *Sketch -> Hochladen mit Programmer*, um der Arduino IDE mitzuteilen, dass sie den ISP nutzen soll, um das Programm zu übertragen.

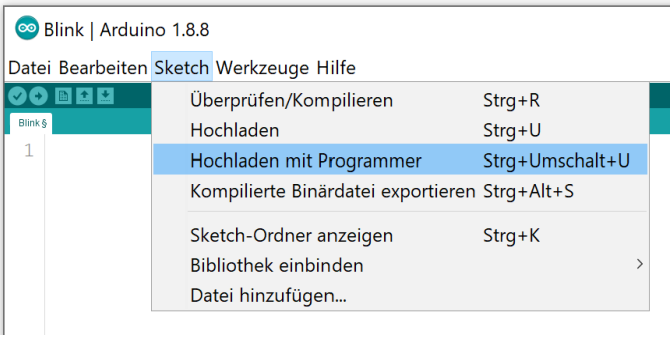


Abb. 17.15 Das Hochladen muss bei der aktuellen Verbindung via Arduino-ISP über diesen Menübefehl erfolgen

Da der Bootloader nun aufgespielt ist, können wir unseren minimal-Arduino künftig auch wie üblich über die serielle Verbindung mit einem neuen Sketch bespielen. Auch dazu sind spezielle USB-Adapter erhältlich. Es gibt aber auch wieder einen Weg ohne Adapter, indem der USB-zu-seriell-Wandler eines Arduino-Boards genutzt wird. Dafür muss allerdings der ATmega328 des entsprechenden Boards komplett entfernt werden, die Fassung muss leer bleiben – sonst fühlt sich dieser Chip ebenfalls angesprochen und reagiert.

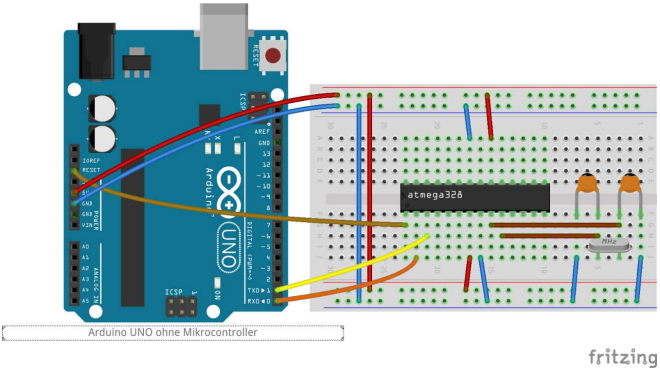


Abb. 17.16 Das Arduino-Board kann als USB-zu-seriell-Wandler verwendet werden, wenn der Mikrocontroller entfernt wurde

Nun können beliebige Programme auf unseren „alleinstehenden“ Mikrocontroller geladen werden. Danach führt er das gewünschte Programm aus, ohne dafür ein Arduino-Board zu benötigen. Die Ein- und Ausgabepins können ohne Einschränkungen genauso genutzt werden wie bei einem originalen Arduino UNO.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 18

Erstellung eigener Platinen

Im vorigen Kapitel haben wir uns angeschaut, wie wir den Platzbedarf und die Kosten reduzieren können, indem wir statt eines kompletten Arduino-Boards nur den eigentlichen Mikrocontroller nutzen. Möchten wir die Schaltung nun dauerhaft einsetzen oder gar in einer kleinen Serie herstellen, sollten wir uns auch eine Alternative für das Breadboard suchen. Zwar bietet das Steckbrett große Vorteile während der Entwicklung einer Schaltung, jedoch können sich beim Transport unbeabsichtigt Verbindungen lösen, zudem verbraucht es meist mehr Platz als nötig.

Der Gedanke liegt nahe, das Projekt auf eine Platine zu übertragen. Im Handel sind zu diesem Zweck Lochrasterplatinen in diversen Größen erhältlich. Einfache Schaltungen können direkt darauf verlötet werden, die Verbindungen zwischen den Komponenten müssen dabei manuell per Draht oder Litze hergestellt werden.

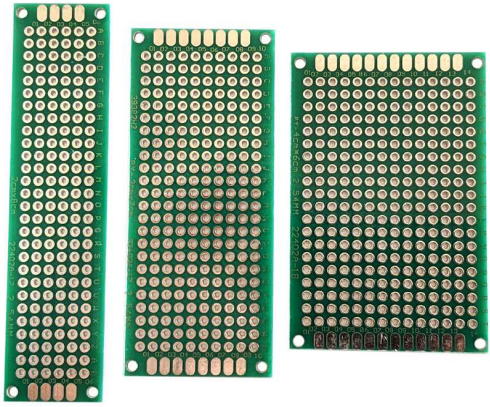


Abb. 18.1 Lochrasterplatinen gibt es in zahlreichen Größen und Varianten

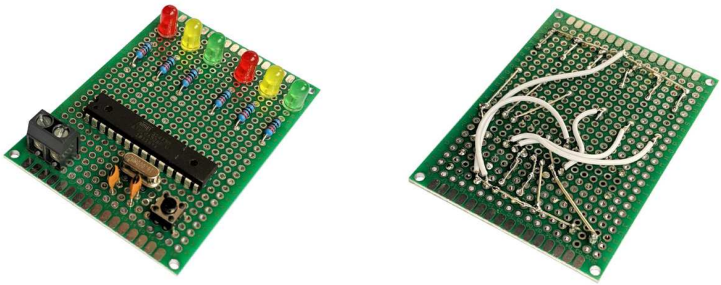


Abb. 18.2 Hier wurde das Ampel-Praxisprojekt auf eine Lochrasterplatine übertragen. Bereits bei einer so einfachen Schaltung werden die Verbindungen auf der Rückseite schnell unübersichtlich und fehleranfällig.

Handelt es sich um komplexere Aufbauten, lohnt es sich, diese mittels einer Software zu planen. Im Folgenden wollen wir uns daher die zwei verbreitetsten Programme ansehen.

18.1 Fritzing

Angehörige der Fachhochschule Potsdam entwickelten die freie Software *Fritzing*, welche sich besonders an Hobby-Entwickler richtet. Ihr großer Vorteil liegt neben der intuitiven Bedienbarkeit auch darin, dass sich Breadboard-Schaltungen planen lassen. Alle Breadboard-Ansichten in diesem Buch wurden ebenfalls mit Fritzing erstellt.

Laden Sie die Software von der offiziellen Projektseite

http://fritzing.org/download/

herunter. Wie üblich bei freier Software haben Sie die Möglichkeit, für das Projekt zu spenden. Die heruntergeladene Datei ist ein zip-Archiv. Genau wie bei Processing in Kapitel 13.2.1 genügt es, diese Datei an ei-

nen Speicherort Ihrer Wahl zu extrahieren. Es ist also keine Installation auf Ihrem Windows-System nötig.

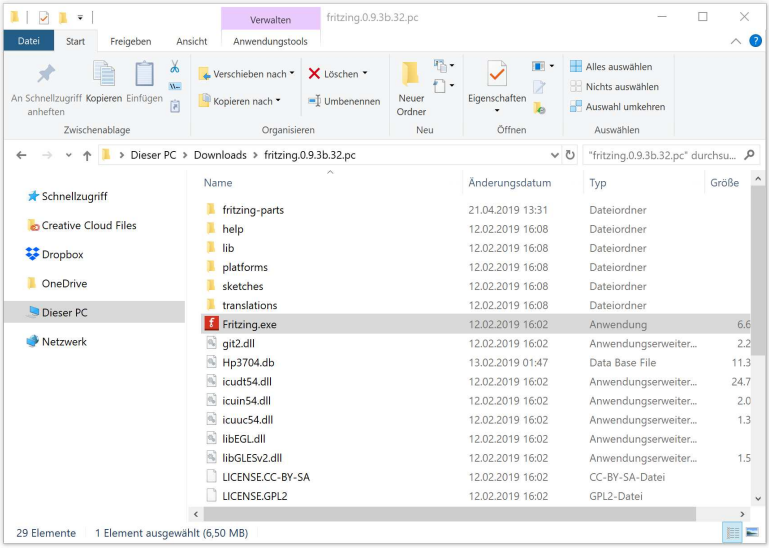


Abb. 18.3 Nach dem Extrahieren kann die Software mit der Datei Fritzing.exe gestartet werden

Im Zielordner finden Sie danach die ausführbare Datei *Fritzing.exe*, sie startet das Programm. Das grundlegende Prinzip besteht darin, dass Sie ein gewünschtes Projekt zunächst grafisch am virtuellen Breadboard zusammenbauen und verdrahten. Im Hintergrund werden parallel ein Stromlaufplan und ein Leiterplatten-Layout vorbereitet. Dieses kann später genutzt werden, um ein PCB (*Printed Circuit Board* – eine professionell hergestellte Platine) anfertigen zu lassen. Im Vergleich zu Lochrasterplatinen ist bei PCBs eine deutlich bessere Raumausnutzung möglich, zudem entfällt der Verdrahtungsaufwand, da die Verbindungen bereits auf der Platine realisiert werden. Daher haben PCBs die manuelle Verdrahtung in der Serienfertigung schon vor Jahrzehnten komplett verdrängt. Anschauungsbeispiele finden Sie in Kapitel 18.3.

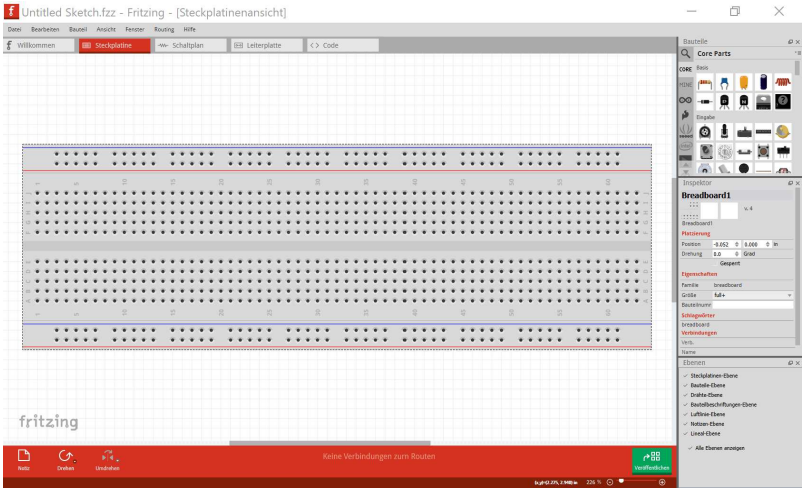


Abb. 18.4 Nach dem Programmstart öffnet sich ein leeres Projekt, mit Klick auf „Steckplatine“ gelangt man in die Breadboard-Ansicht

Auf der rechten Seite finden Sie zahlreiche Bauteile zur Auswahl. Grundlegende elektronische Komponenten befinden sich in der Rubrik *Core*, komplette Arduino-Boards sind über das Arduino-Symbol etwas weiter unten verfügbar. Die Elemente können einfach mit der Maus auf das Breadboard gezogen werden. Ist ein bestimmtes Objekt mit der Maus angewählt, können über den rechts befindlichen *Inspektor*-Bereich weitere Eigenschaften (Widerstandswert, LED-Farbe, Breadboard-Größe und so weiter) angepasst werden. Die Verdrahtung erfolgt ebenfalls intuitiv mit der Maus. Rechtsklicks ermöglichen weitere Anpassungen wie das Drehen von Bauteilen oder die Wahl der Drahtfarbe.

Sollten Sie spezielle Elemente (beispielsweise ein vormontiertes OLED-Display oder einen bestimmten Sensor) vermissen, lohnt sich eine Internet-Suche nach der konkreten Bezeichnung mit der Ergänzung „Fritzing“. Für viele Module und Bauelemente haben andere Entwickler bereits entsprechende Daten erzeugt und stellen sie zur Ver-

fügung. Es handelt sich um Dateien mit der Endung „.fzpz“, sie können einfach über das Menü *Datei -> Öffnen* in die Fritzing-Software importiert werden.

Zur Veranschaulichung wollen wir das Ampel-Praxisprojekt aus Kapitel 6 mit der minimalen Beschaltung eines ATmega328 realisieren. Wir beginnen mit der Schaltung auf dem Breadboard, welche bei Ihnen natürlich auch anders angeordnet sein kann:

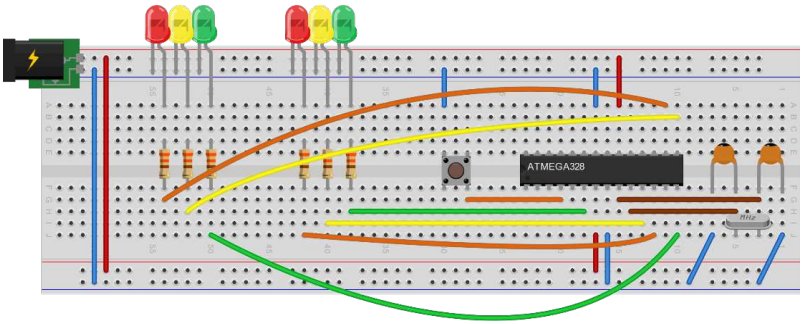


Abb. 18.5 Beispielhafter Aufbau der Schaltung

Idealerweise haben wir diese Schaltung auch bereits einmal real aufgebaut und getestet. Zur Programmierung kann temporär eine Arduino-Platine aushelfen (*Abb. 17.16*).

Als nächster Schritt folgt der Stromlaufplan. Theoretisch kann dieser auch übersprungen werden, allerdings bietet er eine sehr gute Möglichkeit, nochmals alle Verbindungen auf Richtigkeit zu prüfen, da in der Breadboard-Ansicht bei komplexen Projekten schnell etwas verlorengehen kann. Alternativ können Sie auch direkt mit dem Stromlaufplan beginnen, ohne vorher eine Breadboard-Zeichnung zu erstellen.

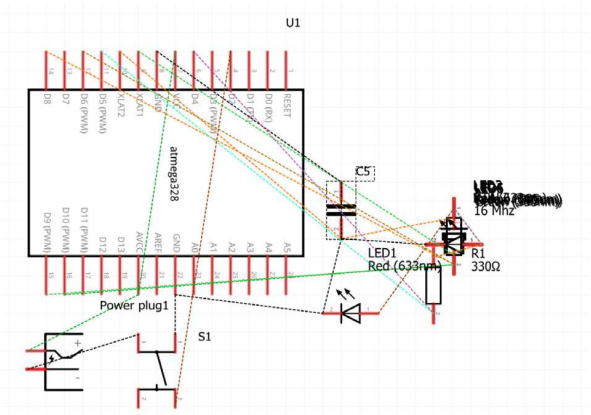


Abb. 18.6 Der Stromlaufplan wirkt zunächst sehr unübersichtlich

Nach dem Klick auf *Schaltplan* erscheint die Ansicht zunächst sehr chaotisch. Ordnen Sie die Bauelemente, indem Sie sie mit der Maus an einen anderen Platz ziehen und gegebenenfalls per Rechtsklick drehen. Die Software hat im Hintergrund zunächst nur alle verwendeten Bauelemente aus dem Breadboard hier eingefügt, ohne auf die Anordnung zu achten.

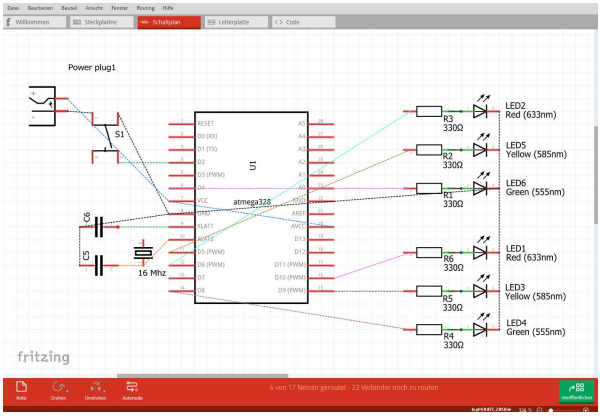


Abb. 18.7 Die Airwires sind deutlich als gestrichelte Linien erkennbar

Nun ist ersichtlich, dass auch die Verbindungen vom Breadboard im Hintergrund vermerkt wurden. Sie sind nun als sogenannte Airwires (Luftdrähte) angedeutet, welche darauf hinweisen sollen, dass an entsprechender Stelle eine Verbindung herzustellen ist. Die eigentlichen Verbindungen im Stromlaufplan können Sie wieder intuitiv per Maus zeichnen. Möchten Sie eine Abzweigung erstellen, halten Sie dazu die Alt-Taste gedrückt.

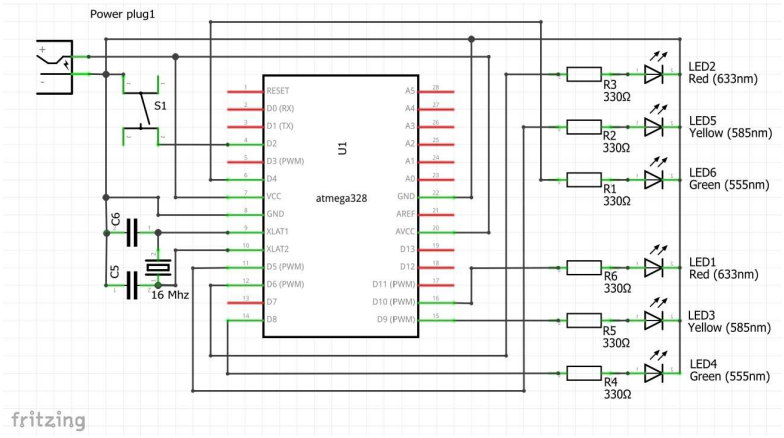


Abb. 18.8 beispielhafter Stromlaufplan

Sind alle Verbindungen hergestellt, können wir daran noch einmal überprüfen, ob die korrekten Anschlüsse gewählt wurden und alle benötigten Pins tatsächlich verbunden sind. Abschließend wechseln wir in die Leiterplatten-Ansicht, um unsere Platine zu gestalten.

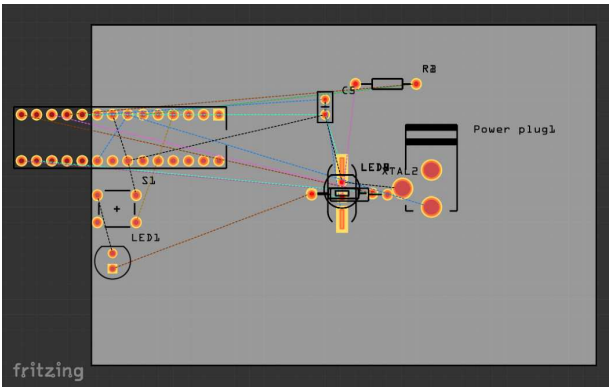


Abb. 18.9 Der Klick auf „Leiterplatte“ wechselt in die Platinen-Ansicht

Auch hier begegnet uns zunächst wieder das Chaos. In gleicher Weise wie beim Stromlaufplan ordnen wir zunächst die Elemente sinnvoll auf der Platine an. Je nach Zweck des Projektes kann es nun nötig sein, auf bestimmte Gehäuseabmessungen zu achten oder aber möglichst platzsparend zu arbeiten.

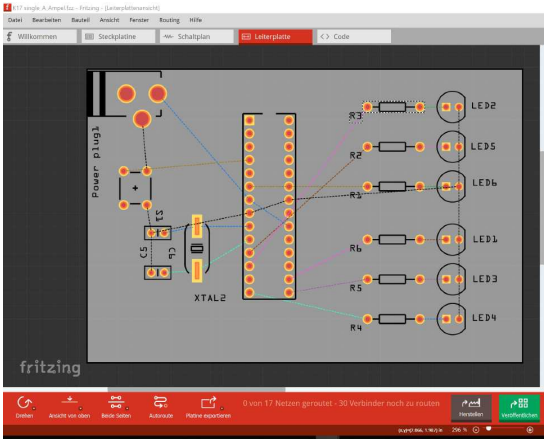


Abb. 18.10 Die vorgesehenen Verbindungen werden wieder durch Airwires markiert. Am unteren Bildrand befindet sich die Autoroute-Schaltfläche.

Airwires weisen wieder auf die zu verbindenden Teile hin. Es sei an dieser Stelle erwähnt, dass die Software über einen sogenannten *Autorouter* verfügt, welcher die Verbindungen automatisch herstellen kann. Allerdings sind die Ergebnisse solcher Algorithmen selten zufriedenstellend, da die Software weder elektrische Vorgaben (hochfrequente Signale sollten möglichst nicht um spitze Ecken geleitet werden und so weiter) noch wirtschaftliche (Vermeidung von zusätzlichen Bohrungen) oder ästhetische Anliegen ausreichend berücksichtigt.

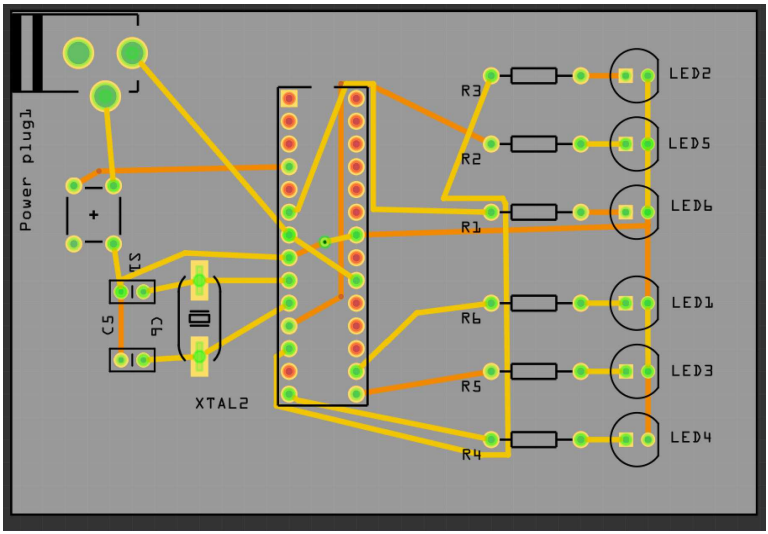


Abb. 18.11 Ergebnis des Autorouters. Gelbe Verbindungen liegen auf der Oberseite, orange auf der Unterseite der Platine. Unter dem Mikrocontroller wäre sogar eine zusätzliche Bohrung nötig.

Besser ist es daher, die Verbindungen manuell per Maus herzustellen, in gleicher Weise wie beim Schaltplan. Leiterbahnen, welche potenziell eine höhere Stromstärke führen (Betriebsspannung und Masse, außerdem beispielsweise auch Motor- oder Spulenansteuerungen) sollten dabei auch direkt breiter angelegt werden. Zu dünne Leiterbahnen stellen einen größeren Widerstand dar und können sich bei hohen Stromstärken erwärmen. Die genaue Berechnung erfolgt unter Beachtung der Kupferstärke und des tolerierbaren Temperaturanstieges.

Als Faustformel können Sie davon ausgehen, dass für 1 Ampere Stromstärke eine Bahnbreite von 1 Millimeter (entspricht rund 40 mil[1]) ausreicht. Die ohnehin nur für 20 Milliampere ausgelegten Ein- und Ausgangspins des Mikrocontrollers können also mit deutlich dünneren Verbindungen ausgestattet werden. Im Zweifel entscheidet man sich natürlich immer für die breitere Variante, sofern genügend Platz zur Verfügung steht.

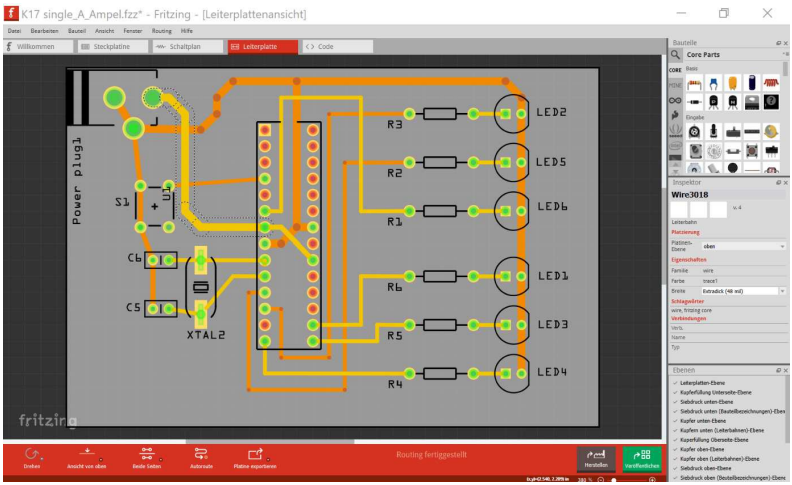


Abb. 18.12 Die Breite und Position (Platinenober- beziehungsweise -unterseite) der Leiterbahnen können im Inspektor-Menü rechts eingestellt werden.

Die fertig geplante Platine kann über das Menü *Datei -> Exportieren* in verschiedenen Formaten ausgegeben werden.

1 *mil* wird als Abkürzung für Milli-Inch, also tausendstel Zoll, verwendet.

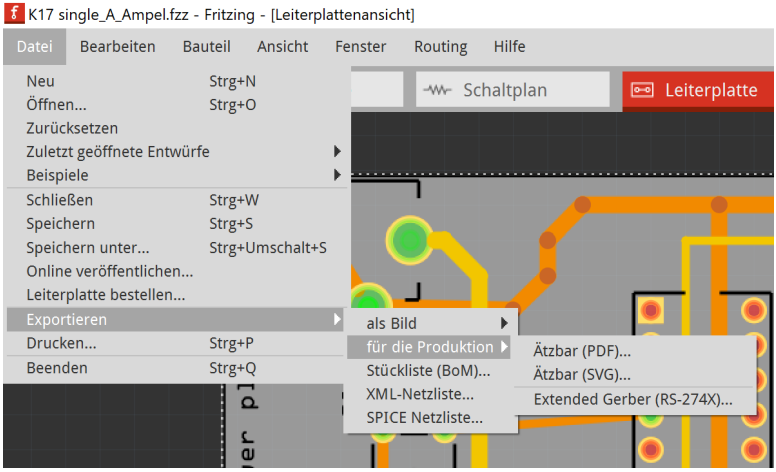


Abb. 18.13 Der Datenexport ist über das Menü Datei möglich

Sicher haben Sie bemerkt, dass Sich im oberen Bereich der Software auch eine *Code*-Schaltfläche befindet. Hier kann optional der zugehörige Arduino-Sketch bearbeitet werden, wenn zuvor ein Compiler angegeben wurde. Allerdings ist diese Möglichkeit weniger komfortabel als die Codebearbeitung in der Arduino IDE, daher soll es hier bei der einfachen Erwähnung bleiben.

18.2 EAGLE

Fortgeschrittene Anwender nutzen statt Fritzing meist *EAGLE* (*Einfach anzuwendender grafischer Layout-Editor*). Diese kommerzielle Software des Herstellers *Autodesk* ist in ihrer kostenfreien *Free*-Version auf eine Leiterplattengröße von 80 cm² beschränkt und erfordert eine Online-Registrierung. Vorteile liegen in der größeren Datenbank an verfügbaren Bauelementen (mit zahlreichen Baugrößen und Variationen) und präziseren Möglichkeiten der Platinengestaltung, nachteilig ist hingegen die weniger nutzerfreundliche Oberfläche. Eine Breadbord-Ansicht gibt es nicht, hier wird das Layout nur anhand des Stromlaufplans entwickelt.

Um EAGLE zu nutzen, laden Sie es von der Herstellerwebsite

https://www.autodesk.de/products/eagle/free-download

herunter. Der Download besteht aus einer ausführbaren Datei. Starten Sie diese Datei, um die Installation durchzuführen. Nach dem Akzeptieren des Lizenzvertrages haben Sie die Möglichkeit, den Installationsort anzupassen und eine Desktopverknüpfung zu erstellen.

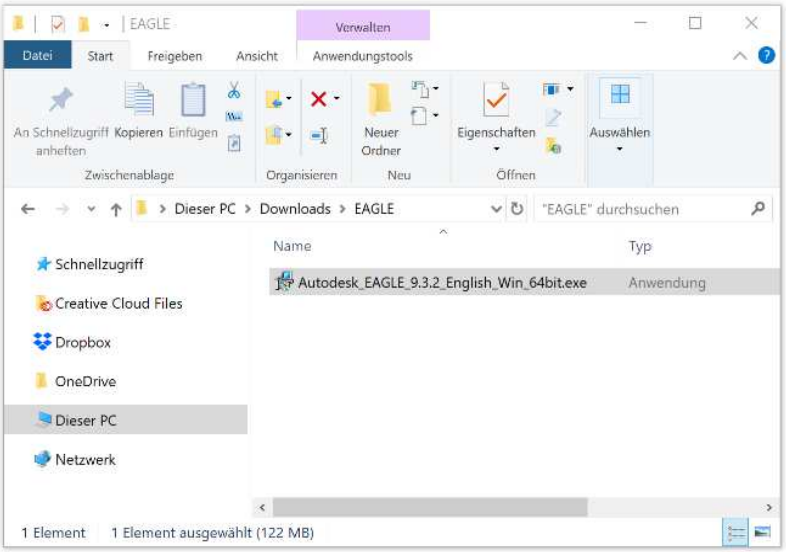


Abb. 18.14 Die ausführbare Datei enthält das Installationsprogramm

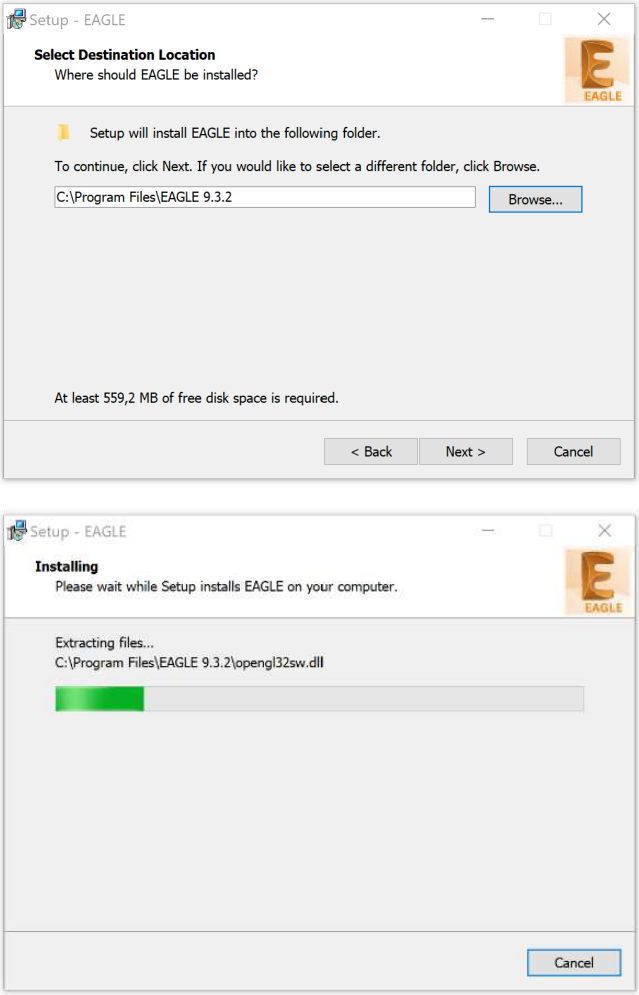


Abb. 18.15 Der Installationsort ist frei wählbar, die Installation läuft automatisch

Öffnen Sie die Software danach über das entsprechende Symbol. Unter Umständen werden Sie aufgefordert, ein kostenloses Benutzerkonto anzulegen, um die Software zu registrieren.

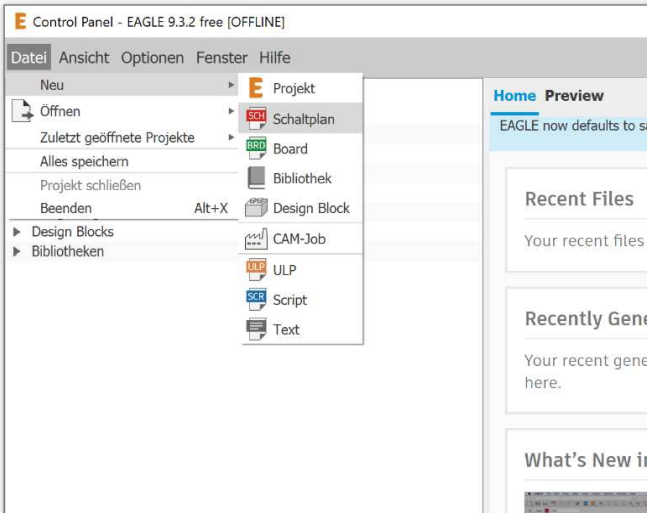


Abb. 18.16 Die Schaltfläche „Projekt“ erstellt nur ein leeres Projekt. Günstiger ist die Schaltfläche „Schaltplan“, welche direkt das Schaltplan-Fenster öffnet.

Mittels *Datei -> Neu -> Schaltplan* können Sie nun das Projekt beginnen.

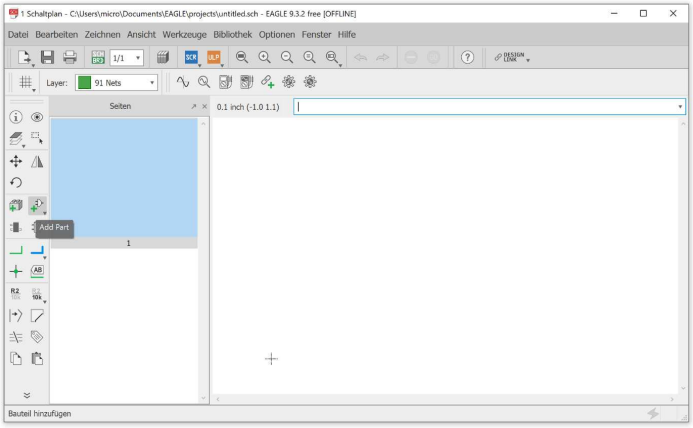


Abb. 18.17 Die Schaltfläche „Add Part“ fügt neue Bauelemente hinzu

Über die Funktion „Add Part“ werden nun die einzelnen Elemente des Stromlaufplans ausgewählt. Es öffnet sich dabei eine sehr umfangreiche Datenbank, welche in Ordner unterteilt ist.

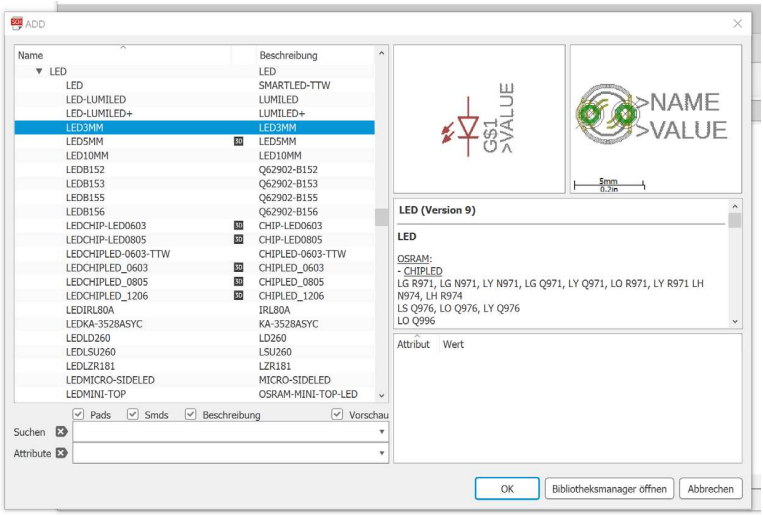


Abb. 18.18 Die Liste der verfügbaren Bauteile ist sehr umfangreich. Die Suchfunktion kann das Finden erleichtern.

Zur Orientierung seien hier einige häufig genutzte Gruppen angeführt:

- *rcl* – Widerstände (R), Kondensatoren (C) und Spulen (L)
- *pinhead* – Steckverbinder
- *led* – Leuchtdioden
- *switch* – Schalter und Taster

Leider ist der ATmega328-Mikrocontroller nicht standardmäßig enthalten. Er kann über die EAGLE-Bibliothek *avr-6*[2] nachinstalliert werden.

2 http://www.ladyada.net/wiki/partfinder/microcontroller

Die Schaltfläche *Bibliotheksmanager öffnen* führt zu einem Fenster, in welchem die heruntergeladene Datei installiert werden kann.

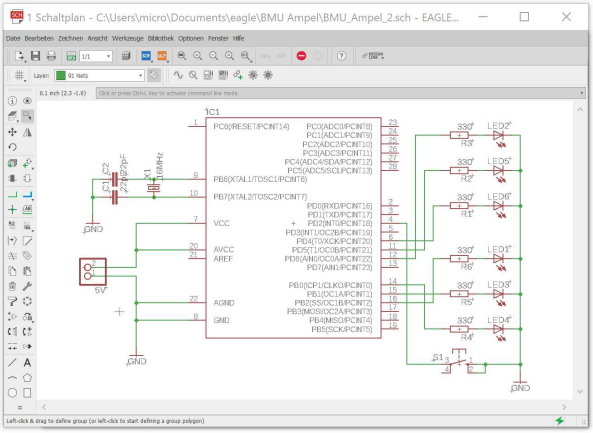


Abb. 18.19 Die Darstellung des Stromlaufplans ähnelt der Darstellung bei Fritzing, allerdings werden die Pins des Mikrocontrollers hier nicht in ihrer realen Anordnung, sondern funktional sortiert gezeigt

Nach etwas Eingewöhnung lässt sich somit auch das Beispiel der Ampel-Schaltung übertragen. Die dafür nötigen Werkzeuge befinden sich in der Leiste links und offenbaren ihre Funktion über eine Hinweiseinblendung, wenn man den Mauszeiger darüber bewegt.

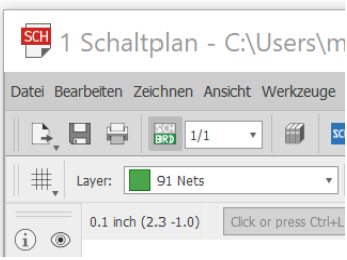


Abb. 18.20 Die unscheinbare Schaltfläche „SCH BRD“ führt zur Platinenansicht

Nach der Fertigstellung des Schaltplans kann über die unscheinbare Schaltfläche „SCH BRD“ (*Schematic/Board* – Stromlaufplan/Platine) zur Platinenansicht gewechselt werden.

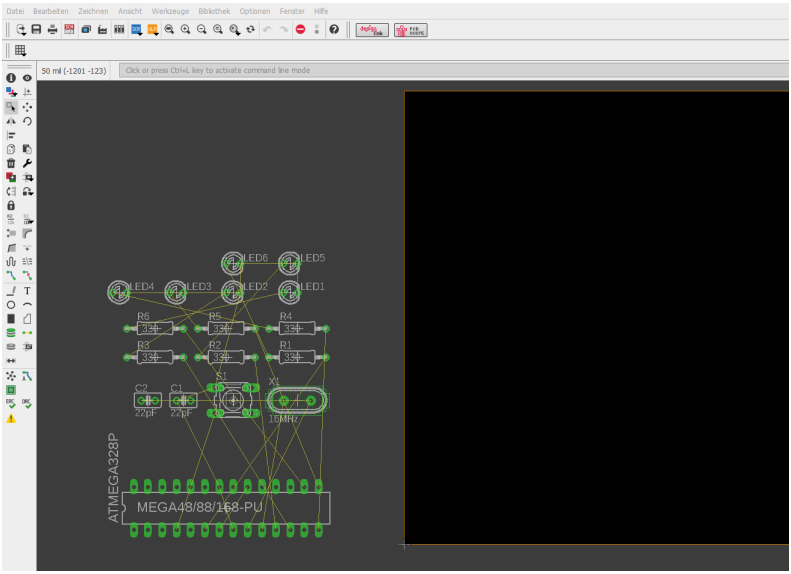


Abb. 18.21 In der Platinenansicht müssen die Bauelemente erneut manuell angeordnet werden

Hier begegnet uns nun ein ähnliches Bild wie bei Fritzing, allerdings weniger chaotisch. Die Bauelemente sind neben der Platine (schwarzes Feld rechts) aufgereiht und müssen per Maus an ihren beabsichtigten Platz gezogen werden. Airwires weisen uns wieder auf die aus dem Stromlaufplan abgeleiteten Verbindungen hin. Sind alle Bauelemente an ihrem Platz, können mit dem Werkzeug *Route* die Verbindungen hergestellt werden.

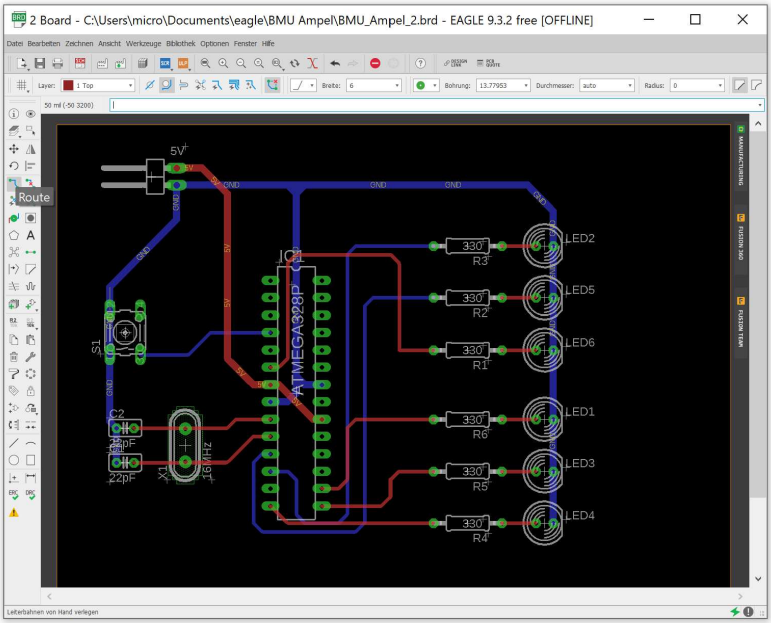


Abb. 18.22 Das Route-Werkzeug erlaubt das Zeichnen der Leiterbahnen. Im oberen Bereich können Breite und Layer (Top (rot) - Platinenoberseite, Down (blau) - Platinenunterseite) ausgewählt werden.

Der hier ebenfalls vorhandene Autorouter sollte aus den bereits im vorigen Kapitel beschriebenen Gründen nur als Notlösung zum Einsatz kommen.

Die zur Platinenherstellung benötigten Daten können über das Menü *Datei -> Generate CAM* erzeugt werden. Das Resultat ist ein zip-Archiv mit allen relevanten Daten.

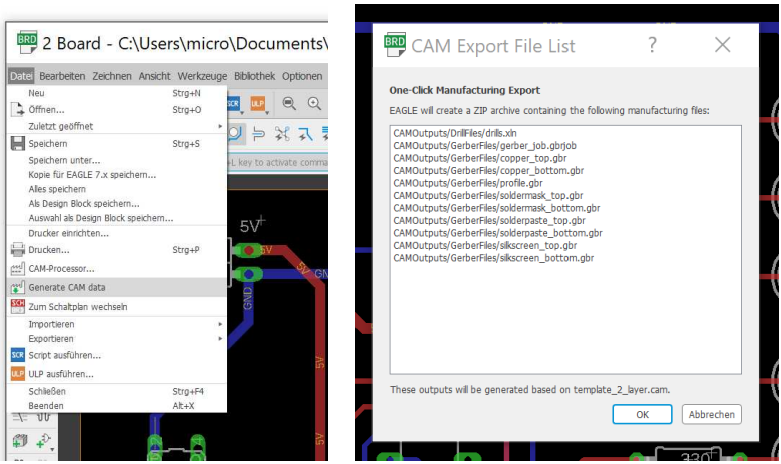


Abb. 18.23 Diese Menüoption stellt die CAM-Daten (Computer Aided Manufacturing, computergestützte Herstellung) bereit

Manche Hersteller akzeptieren auch die EAGLE-Projektdateien, da diese ebenfalls bereits alle relevanten Informationen enthalten.

18.3 Professionelle Platinenherstellung

Im Internet bieten diverse Firmen die Herstellung von Platinen auch als einzelne Prototypen oder in Kleinserien an. Die Kosten für eine Einzelanfertigung im Format 5 cm mal 5 cm liegen dabei zwischen 20 und 50 Euro. Eine spätere Serienfertigung ist entsprechend preiswerter.

Je nach PCB[3]-Anbieter werden zur Herstellung verschiedene Datenformate akzeptiert. Es handelt sich dabei üblicherweise immer um Vektordaten, beispielsweise PDF oder SVG. Oft gebräuchlich ist zudem das noch aus den 1980-er-Jahren stammende Gerber-Format, welches auch von EAGLE genutzt wird. Hierbei handelt es sich um Vektordaten, welche in Textdateien abgelegt sind.

3 *Printed Circuit Board* – gedruckte (industriell hergestellte) Leiterplatte

Unabhängig vom konkreten Dateityp werden immer unterschiedliche Layer, also Schichten, hinterlegt. Diese dienen beim Herstellungsprozess als Schablonen. So gibt es je einen Layer für die Leiterbahnen auf der Ober- und Unterseite, einen Layer für die Bohrungen, einen für die Lötflächen und so weiter. Aufgrund der Vielzahl der Formate kann es vorkommen, dass es dabei zu Verwechslungen kommt. Seriöse Anbieter prüfen Ihre Daten daher vor der Produktion. In manchen Fällen zeigt Ihnen die Bestell-Website die verschiedenen Layer zum Abgleich auch noch einmal an.

Oft werden verschiedene Bestelloptionen angeboten, daher seien nachfolgend übliche Standardwerte genannt, auf die Sie sich im Zweifel beziehen können:

- Material: FR-4 (Kunstharz)
- Dicke: 1,6 Millimenter
- Kleinste Leiterbahnbreite: 6 mil
- Kleinster Leiterbahnabstand: 6 mil
- Kleinste Bohrung: 0,3 mil
- Dicke der Kupferbeschichtung: 35 µm

Das Fritzing-Projekt bietet einen eigenen Herstellungsservice an, welcher direkt aus der Software heraus beauftragt werden kann und daher besonders für Einsteiger komfortabel zu nutzen ist.

Abb. 18.24 Der „Herstellen"-Button in der Platinenansicht der Fritzing-Software leitet direkt zu einem Platinenhersteller weiter

Nachfolgend wird beispielhaft ein etwas komplexeres Projekt des Autors dargestellt. Es handelt sich um eine Schrittmotor-Steuerung auf Basis des ATmega328P. Die Details des Stromlaufplans sind hier nicht relevant, es soll lediglich der Werdegang von der Planung bis hin zur konkreten Platine noch einmal aufgezeigt werden:

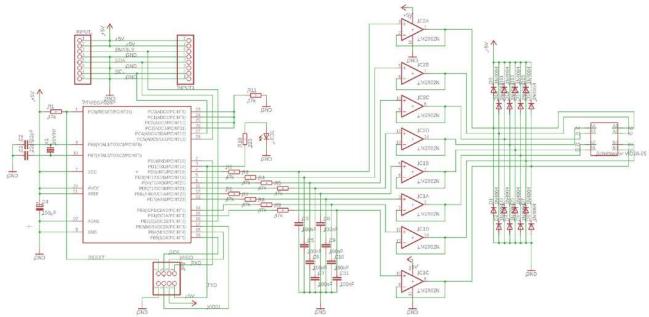


Abb. 18.25 Der Stromlaufplan wurde in EAGLE erstellt. Links der ATmega328P.

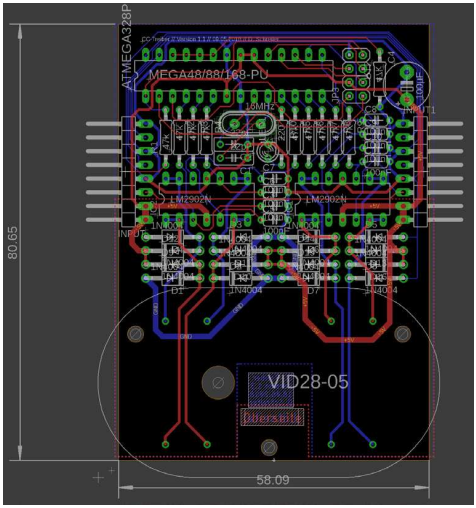


Abb. 18.26 Die Platine wurde unter der Maßgabe bestmöglicher Platzeinsparung gestaltet. Der große Freiraum im unteren Bereich wurde für den zu steuernden Schrittmotor vorgesehen.

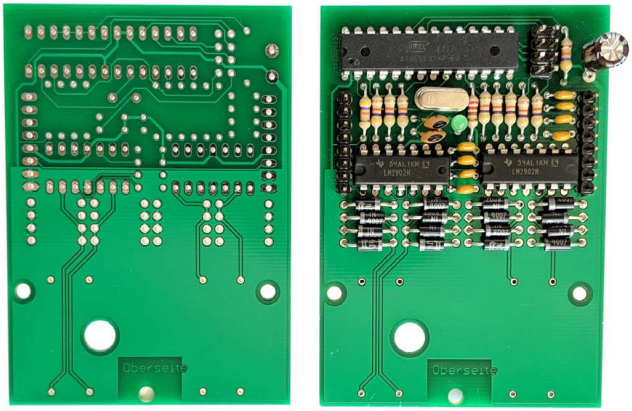


Abb. 18.27 Links die vom Platinenhersteller gelieferte unbestückte Platine, rechts nach der Bestückung (ohne Motor). Deutlich erkennbar ist der ATmega328P am oberen Rand der Platine nebst Schwingquarz und Kondensatoren.

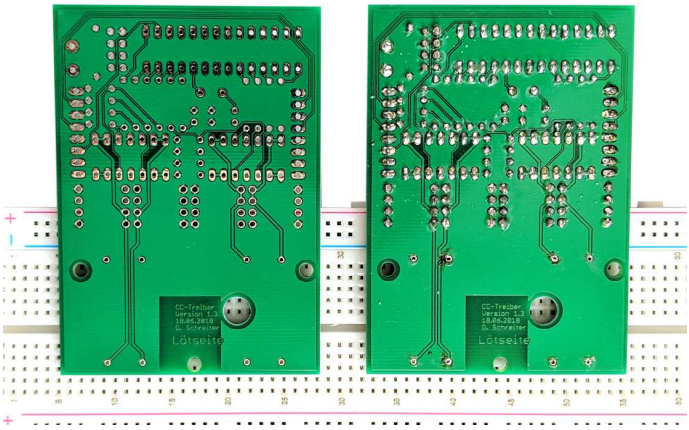


Abb. 18.28 EAGLE unterstützt auch das Ausfüllen einer Fläche, so dient die Unterseite als große Massefläche auch zum Abtransport von Wärme.

Als Platinenhersteller wurde hierbei die Firma MME-Leiterplatten[4] beauftragt, der Preis pro Stück lag bei circa 20 Euro. Bei anderen Herstellern kann der Preis in beide Richtungen abweichen.

In den obigen Beispielfotos ist zu erkennen, dass die komplette Unterseite (sowie ein großer Teil der Oberseite) von einer Metallfläche ausgefüllt ist, welche nur durch die einzelnen Leiterbahnen unterbrochen wird. (Im Gegensatz dazu ist der obere Teil der Oberseite farblich deutlich dunkler, hier existiert diese Fläche nicht.) Dieses große Areal ist mit der Masse verbunden und erspart so einerseits die manuelle Verbindung der Masse-Pins, andererseits schirmt sie die Schaltung auch teilweise gegen mögliche elektromagnetische Störungen von außen (Mobiltelefone, Funkgeräte und so weiter) ab. Ein zusätzlicher Vorteil ist, dass diese große Fläche viel Wärme aufnehmen und abstrahlen kann, denn die verwendeten Motortreiber-Chips nutzen die Masse-Pins auch zum Abtransport von Wärme, welche insbesondere bei der Regelung relativ hoher Stromstärken entsteht. In der EAGLE-Software lässt sich eine solche Fläche in der Board-Ansicht erstellen, indem man mit dem Polygon-Werkzeug ihre Umrisse zeichnet und ihr anschließend per Rechtsklick den Namen *GND* gibt.

Beim Herstellungsprozess kann es aufgrund von Fertigungstoleranzen dazu kommen, dass besonders dünne Leiterbahnen oder dünne Abstände fehlerhaft realisiert werden. Daher bieten professionelle Hersteller üblicherweise einen sogenannten E-Test an. Hierbei wird maschinell geprüft, ob das fertige Werkstück auch tatsächlich alle elektrischen Verbindungen laut Stromlaufplan besitzt und es keine ungewollten Kurzschlüsse gibt. Dieser Test ist besonders bei komplexen Schaltungen oder Serienfertigungen sehr zu empfehlen.

4 https://www.mme-pcb.com/

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 19

Fehlersuche und Programmoptimierung

Ein technisches Hobby wird nicht selten als „Herumexperimentieren" verunglimpft. Dieses Wort ist unpassend, weil es eine gewisse Ziellosigkeit impliziert. Dennoch bleiben auch bei guter Planung Rückschläge nicht aus, und selbst Profi-Programmierer und Elektroniker stehen regelmäßig vor Projekten, die einfach nicht tun, was sie sollen. Doch statt an Problemen zu verzweifeln, kann man sie nutzen, um daraus zu lernen.

Auf den folgenden Seiten werden Sie erfahren, wie man systematisch Fehler eingrenzen kann. Zudem wollen wir einen Blick auf die häufigsten Flüchtigkeitsfehler beim Programmieren werfen und einige Möglichkeiten zur Optimierung von Arduino-Sketches kennenlernen.

19.1 Fehler im Programmcode

Je mehr Codezeilen ein Sketch enthält, umso mehr Gelegenheiten gibt es auch, unbeabsichtigt Fehler einzubauen. Man unterscheidet dabei Syntaxfehler (Fehler in der Schreibweise) von Semantikfehlern (Fehler in der Logik). Da Präprozessor und Compiler den Sketch bei jedem Hochladen auf Plausibilität prüfen, können Ihnen deren Fehlermeldungen wichtige Hinweise auf Syntaxfehler liefern. Achten Sie dazu auf die Meldungen in der schwarzen Statusleiste am unteren Rand der Arduino IDE. Nachfolgend einige Beispiele ausgehend vom bereits bekannten Blink-Sketch:

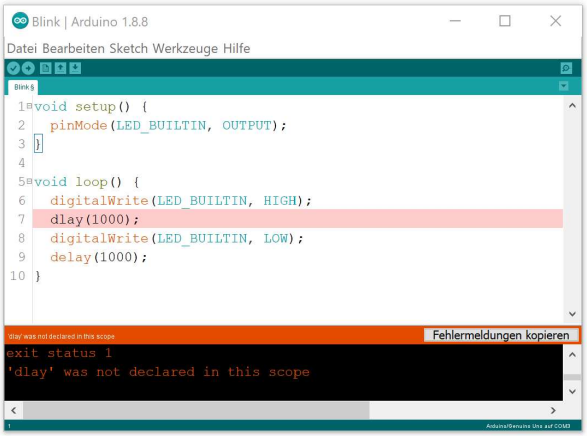


Abb. 19.1 Der korrekte Funktionsname wäre `delay()`

`'...' was not declared in this scope` deutet darauf hin, dass entweder (wie hier) ein Funktionsname falsch geschrieben wurde oder die Schreibweise einer Variablen nicht mit ihrer Schreibweise bei der Deklaration übereinstimmt. Auch die Groß-/Kleinschreibung muss identisch sein.

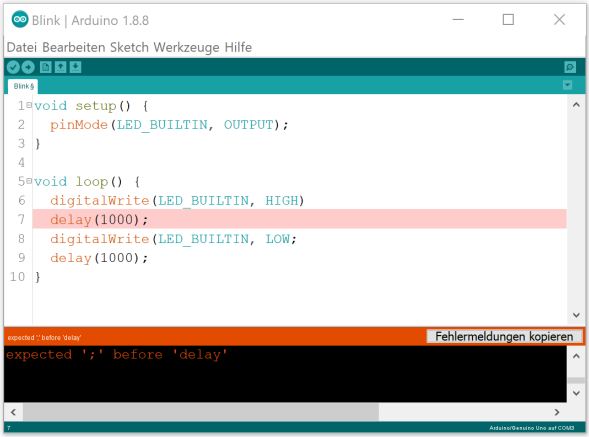


Abb. 19.2 Hier fehlt ein Semikolon

`expected ';' before '...'` weist auf ein fehlendes Semikolon hin. Die Fehlermeldung tritt allerdings oft nicht in der Zeile auf, in welcher das Zeichen vergessen wurde, sondern meist erst eine oder mehrere Zeilen darunter. Das liegt daran, dass sich Befehle prinzipiell über mehrere Zeilen erstrecken dürfen. Für den Compiler wird es erst problematisch, wenn ein neuer Befehl folgt, ohne dass der vorherige korrekt per Semikolon abgeschlossen wurde. Daher markiert die Meldung in diesem Beispiel Zeile 7, obwohl der Fehler eigentlich in Zeile 6 liegt.

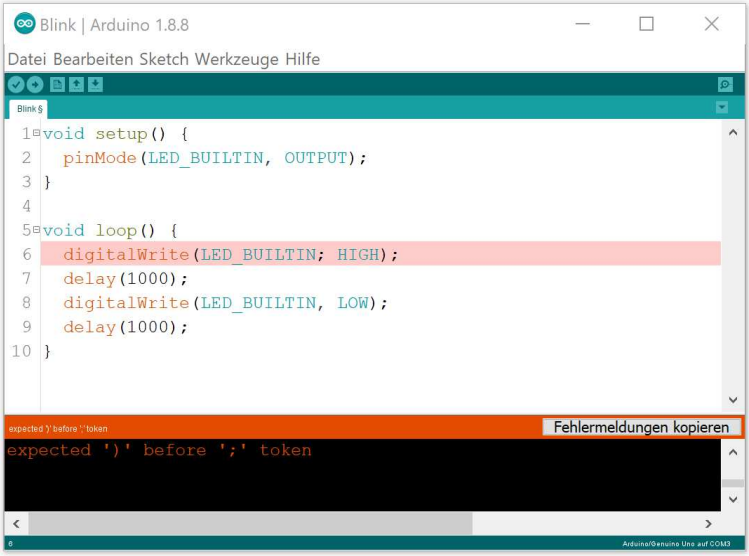


Abb. 19.3 Funktionsargumente werden durch Kommata getrennt

`expected ')' before ';' token` hört sich zunächst an wie eine fehlende Klammer, ist in diesem Beispiel aber ein falsch positioniertes Semikolon. Die Funktionsargumente müssten korrekterweise durch ein Komma getrennt werden, so wie zwei Zeilen darunter. Durch das Semikolon nimmt der Compiler nun an, der Befehl sei bereits nach `LED_BUILTIN` zu Ende – dafür fehlt ihm allerdings noch eine schließende runde Klammer.

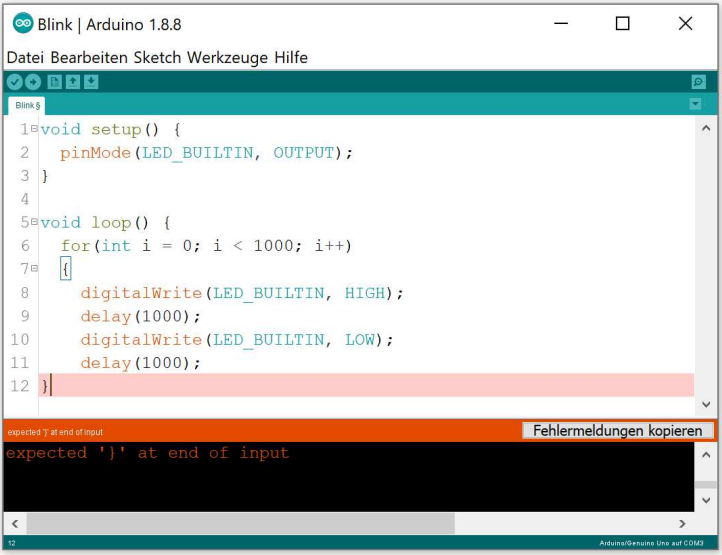


Abb. 19.4 Praktisch: Stellt man den Cursor an eine Klammer, markiert die Arduino IDE automatisch das vermeintliche Gegenstück.

`expected '}' at end of input` weist auf eine fehlende geschweifte Klammer hin. Die Arduino IDE markiert dabei in der Regel das Ende des Sketches, da der Compiler die sich öffnenden und sich schließenden Klammern im gesamten Sketch addiert. Am Ende muss die Anzahl beider gleich sein, sonst erfolgt obige Fehlermeldung. Unabhängig davon kann die korrekte Position der Klammer aber auch viele Zeilen davon entfernt sein. Zur einfacheren Überprüfung bietet die Arduino IDE eine praktische Funktion: Steht der Cursor gerade an einer Klammer, wird das vermeintliche Gegenstück eingerahmt. Entspricht dieses Gegenstück nicht der beabsichtigten Stelle, liegt dazwischen eine fehlende oder überflüssige Klammer. Im obigen Beispiel fehlt die sich schließende Klammer der `for`-Schleife.

An den Beispielen wurde offensichtlich, dass die Fehlermeldungen manchmal etwas irreführend sein können, denn Präprozessor und Compiler lesen den Sketch anders als wir. Sie sind nicht in der Lage,

einen gewünschten Sinn im Programm zu erkennen, sondern interpretieren einfach nur Zeile für Zeile den Quelltext anhand ihrer Programmierung. Insbesondere bei geschweiften Klammern zur Gruppierung von `if`-Blöcken oder Schleifen kann dies dazu führen, dass eine scheinbar unlogische Fehlermeldung erscheint, obwohl der ursächliche Fehler (eine fehlende oder überflüssige geschweifte Klammer) dutzende Zeilen entfernt liegt.

Des Weiteren ist zu bedenken, dass nicht jedes Problem auch eine Fehlermeldung in der Software erzeugt. Liegt trotz eines Fehlers ein formal korrekter Befehl vor (auch wenn er aufgrund des Fehlers womöglich sinnlos sein mag), wird es beim Hochladen keine Meldung geben. Ihr Programm wird jedoch nicht das gewünschte Ergebnis liefern. Eine häufige Ursache sind verwechselte Vergleichsoperatoren. Möchten Sie beispielsweise testen, ob die Variable `Eingabe` den Wert 5 aufweist, würde die folgende falsche Notation keine Fehlermeldung erzeugen:

```
...
  if(Eingabe = 5)
  {
...
```

Es gibt hier formal kein Problem: Innerhalb der Klammern wird der Variablen `Eingabe` der Wert 5 zugewiesen. Diese Anweisung ergibt formal immer `true`, wodurch der darauffolgende `if`-Block in jedem Fall ausgeführt wird. Dies entspricht natürlich nicht Ihrer Absicht – aber das weiß der Compiler nicht. Korrekt wäre die Verwendung des Vergleichsoperators == statt des Zuweisungsoperators =, also wie folgt:

```
...
  if(Eingabe == 5)
  {
...
```


Ähnliches gilt auch für ungewollte Endlosschleifen. Die folgende Schleife wird niemals verlassen, obwohl ihr Kopf auf den ersten Blick vernünftig wirkt:

```
...
  for(byte a = 0; a < 300; a++)
  {
...
```

Der Fehler liegt darin, dass eine `byte`-Variable niemals den Wert von 255 übersteigen wird. Sie wird stattdessen einfach wieder bei 0 beginnen. Korrekt wäre hier also die Nutzung einer `int`-Variable.

An einem zusätzlichen Beispiel wollen wir weitere häufige Fehler betrachten. Dazu schreiben wir ein Programm, welches über den seriellen Monitor eine Tabelle mit einer definierten Anzahl Zeilen und Spalten ausgeben soll. Die Zellen sollen nummeriert sein. Die gewünschte Bildschirmausgabe sieht entsprechend so aus:

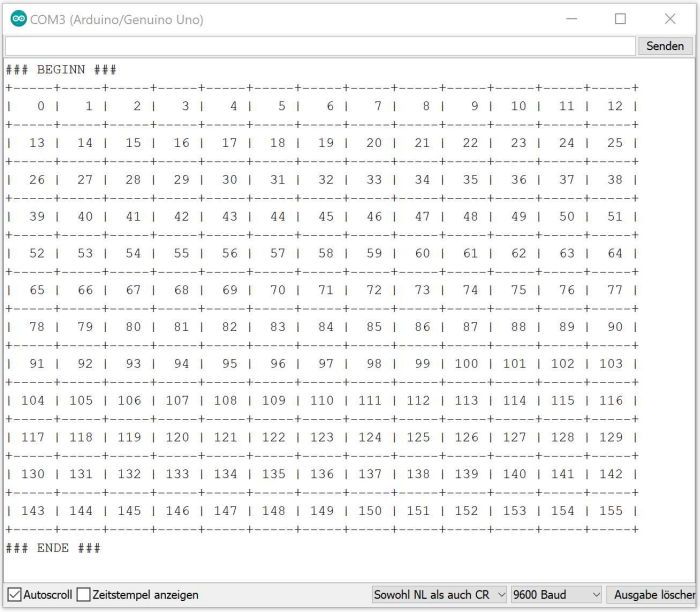


Abb. 19.5 angestrebte Bildschirmausgabe

Zu diesem Zweck wurde der folgende Sketch vorbereitet. Allerdings funktioniert er nicht, denn er enthält 10 Flüchtigkeitsfehler. Einige von

ihnen erzeugen keinerlei Fehlermeldung. Zur Übung können Sie gern testen, ob Sie alle Fehler finden – die Auflösung finden Sie nach dem Sketch anhand der Zeilennummern.

```
#define ZEILENZAHL 12
#define SPALTENZAHL 13;
boolean Zellennummer;

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

void loop() {
   Serial.println('### BEGINN ###');

   for(byte aktuelleZeile = 0; aktuelleZeile <= ZEILENZAHL;
   aktuelleZeile++)
   {
    for(byte aktuelleSpalte = 0; aktuelleSpalte
    < SPALTENZAHL; aktuelleSpalte++);
     Serial.print("+-----");
     Serial.println("+");

     if(aktuelleZeile = ZEILENZAHL)
     {
      Serial.println("### ENDE ###");
     }
     else
     {
       for(byte aktuelleSpalte = 0; aktuelleSpalte
       < SPALTENZAHL, aktuelleSpalte++)
       {
        Zellennummer = aktuelleSpalte + SPALTENZAHL *
        aktuellezeile;
         Serial.print("|");
         if(Zellennummer < 1000)
           Serial.print(" ");
         if(Zellennummer < 100)
           Serial.print(" ")
         if(Zellennummer < 10)
           Serial.print(" ");
         Serial.print(Zellennummer);
         Serial.print(" ");
       }
     Serial.println("|");
       }
```

```
43
44      delay(60000);
45  }
```

In folgenden Zeilen befinden sich Fehler:

2: Bei Anweisungen an den Präprozessor darf kein Semikolon folgen.

3: Der Variablentyp `boolean` ist hier ungeeignet, möglich wäre `byte` oder besser `int`. Dieser Fehler erzeugt beim Hochladen keine Fehlermeldung.

5: Dem `setup()`-Block fehlt die sich öffnende geschweifte Klammer.

10: Die Zeichenkette steht in Hochkommata (`'`) statt in Anführungszeichen (`"`). Hochkommata führen in diesem Fall jedoch zu einer unbeabsichtigten Umwandlung in eine einzelne `char`-Variable. Als Ergebnis wird nur eine Zahl ausgegeben. Auch dieser Fehler erzeugt keine Meldung beim Hochladen.

16: Nach der `for`-Anweisung steht ein unbeabsichtigtes Semikolon. Dies wird vom Compiler als leere Schleife interpretiert. Im Ergebnis gibt es keine Fehlermeldung, aber der eigentliche Schleifenrumpf (Zeile 15) wird fälschlicherweise nur ein einziges Mal ausgeführt.

20: Hier wurde der Zuweisungsoperator = mit dem Vergleichsoperator == verwechselt. Auch dies erzeugt keine Fehlermeldung, jedoch ist die `if`-Bedingung fälschlicherweise immer `true`.

27: Im Kopf einer `for`-Schleife müssen die Teile per Semikolon getrennt werden, hier steht fälschlicherweise ein Komma.

30: Die Variable `aktuellezeile` gibt es nicht. Korrekt wäre `aktuelleZeile`, denn es wird auch Groß- und Kleinschreibung unterschieden.

35: Nach dieser Anweisung fehlt das Semikolon.

43: Die `for`-Schleife hat keine sich schließende Klammer.

Korrekt muss der Sketch also wie folgt lauten:

```
#define ZEILENZAHL 12
#define SPALTENZAHL 13
int Zellennummer;

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

void loop() {
  Serial.println("### BEGINN ###");

  for(byte aktuelleZeile = 0; aktuelleZeile <= ZEILENZAHL;
  aktuelleZeile++)
  {
    for(byte aktuelleSpalte = 0; aktuelleSpalte <
    SPALTENZAHL; aktuelleSpalte++)
      Serial.print("+-----");
    Serial.println("+");

    if(aktuelleZeile == ZEILENZAHL)
    {
      Serial.println("### ENDE ###");
    }
    else
    {
      for(byte aktuelleSpalte = 0; aktuelleSpalte <
      SPALTENZAHL; aktuelleSpalte++)
      {
        Zellennummer = aktuelleSpalte + SPALTENZAHL *
        aktuelleZeile;
        Serial.print("|");
        if(Zellennummer < 1000)
          Serial.print(" ");
        if(Zellennummer < 100)
          Serial.print(" ");
        if(Zellennummer < 10)
          Serial.print(" ");
        Serial.print(Zellennummer);
        Serial.print(" ");
      }
    Serial.println("|");
    }
  }
  delay(60000);
}
```

Diese Auflistung möglicher Fehler ist natürlich bei weitem nicht vollständig. In der fortgeschrittenen Programmierung können beispielsweise die falsche Dimensionierung eines Arrays oder unbedachte Fehler bei der Arbeit mit Adressen von Variablen sogar Programmabstürze verursachen. Der Mikrocontroller führt dann automatisch einen Neustart durch.

19.2 Fehler außerhalb des Programmcodes

Scheint Ihr Sketch frei von Fehlern zu sein und bleibt dennoch das gewünschte Resultat aus, sollten Sie den Blick auf die angeschlossenen Komponenten richten. Prüfen Sie dabei Folgendes:

- Werden alle Komponenten mit Betriebsspannung versorgt? Nutzen Sie gegebenenfalls ein Spannungsmessgerät, um die korrekte Versorgung mit 5 Volt festzustellen.
- Kann die genutzte Spannungsquelle genügend Stromstärke liefern? Ein USB-Anschluss genügt in der Regel nicht, um zeitgleich mehrere Motoren zu betreiben.
- Sind alle Signalkabel korrekt verbunden? Stimmen die Pin-Nummern mit denen im Sketch überein?
- Werden Pins unbeabsichtigt doppelt genutzt? Wenn beispielsweise eine I^2C-Verbindung aktiv ist, dürfen an den Pins A4 und A5 nicht zusätzlich noch analoge Eingangssignale angelegt werden. Durch eine SPI-Verbindung werden die Pins 11, 12 und 13 blockiert, eine serielle Verbindung belegt die Pins 0 und 1.
- In einigen Fällen kann es auch vorkommen, dass Bibliotheken bestimmte Pins beeinflussen, obwohl an ihnen nichts angeschlossen ist. So nutzt beispielsweise die Bibliothek IRremote.h, welche wir in Kapitel 9.2 für den Infrarotempfänger verwendeten, einen internen Timer des Mikrocontrollers, welcher auch für die Pulsweitenmodulation an den Pins 3 und 11 verantwortlich ist. Nutzt man an diesen Pins die PWM, funktioniert die Bibliothek nicht mehr.

Falls ein bestimmtes externes Modul weiterhin nicht funktioniert, sollten Sie diese Komponente möglichst isoliert von anderen testen. Dazu

bieten sich die kleinen Beispielsketche an, welche zur Vorstellung der Module genutzt wurden. So können Sie feststellen, ob die entsprechende Komponente funktioniert.

Lässt sich auch dadurch das Problem nicht beheben, liegt vermutlich ein Fehler im Ansatz (Semantikfehler) vor. Wenn möglich, nutzen Sie den seriellen Monitor und lassen Sie sich an relevanten Stellen im Programm eine kurze Meldung ausgeben. So können Sie zum Beispiel verfolgen, wann von der `setup()`- in die `loop()`-Routine gewechselt wird. Auch der aktuelle Wert von Variablen während der Ausführung kann interessant sein. So können Sie Stück für Stück nachvollziehen, ob der Programablauf Ihrer Planung folgt – oder ob beispielsweise ein `if`-Block wider Erwarten nicht durchlaufen wird, weil die Bedingung falsch formuliert wurde.

19.3 Speicheroptimierung

Der ATmega328P-Mikrocontroller des Arduino bietet 32 Kilobyte Programmspeicher, 2 Kilobyte Arbeitsspeicher (RAM[1]) und 1 Kilobyte EEPROM. In unseren bisherigen Beispielen haben wir diese Grenzen nie auch nur annähernd erreicht. Dennoch gibt es Situationen, in denen diese in heutigen Maßstäben kleinen Speichergrößen eine Limitierung darstellen. Zum Beispiel, wenn Sie Berechnungen mit vielen Messwerten durchführen möchten und daher reichlich Arbeitsspeicher benötigen oder wenn ein grafisches Menü mit Erklärungstexten auf einem OLED-Display ausgegeben werden soll.

Bei jedem Hochladen zeigt der Compiler, wie viel Speicher der aktuelle Sketch (nach dem Übersetzen in den Maschinencode) belegt:

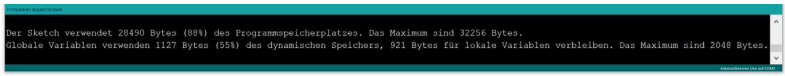


Abb. 19.6 Der Compiler gibt nach dem Übersetzen eine Statusmeldung aus, welche über den auf dem Mikrocontroller benötigten Speicher informiert

1 *Random Access Memory* – Speicher mit beliebigem Zugriff

Er unterscheidet dabei zwischen Programmspeicher und dynamischem Speicher, also Arbeitsspeicher. Der Programmspeicher darf bis zum letzten Byte ausgereizt werden, da in diesem Speicherbereich während der Programmausführung keinerlei Änderungen geschehen. Das Steuerwerk liest lediglich die Befehle daraus und arbeitet sie ab.

Anders verhält es sich beim Arbeitsspeicher. Hier gibt der Compiler lediglich den reservierten Platz für globale Variablen an. Das sind Variablen, welche außerhalb einer Funktion (also auch außerhalb von `setup()` und `loop()`) deklariert wurden. Alle anderen gelten als lokale Variablen, ihr Platzbedarf kann vom Compiler nicht ermittelt werden, da er sich zur Ausführungszeit des Programms ändern kann.

Zur Verdeutlichung blicken wir auf ein Codebeispiel, in welchem die Fakultät[2] einer Zahl durch eine rekursive Funktion berechnet wird. Rekursiv bedeutet dabei, dass die Funktion sich selbst aufruft:

```
1  void setup() {
2    Serial.begin(9600);
3  }
4
5  void loop() {
6    for(int Zahl = 1; Zahl < 14; Zahl++)
7    {
8      Serial.print("Die Fakultät von ");
9      Serial.print(Zahl);
10     Serial.print(" ergibt: ");
11     Serial.println(Fakultaet(Zahl));
12     delay(1000);
13   }
14 }
15
```

2 Die Fakultät einer Zahl bildet man, indem man sie mit allen ihren Vorgängern multipliziert. Diese Berechnung wird im Bereich der Wahrscheinlichkeitsrechnung oft genutzt. Als Symbol nutzt man das Ausrufezeichen. Die Fakultät von 5 wäre also 5! = 5 * 4 * 3 * 2 * 1. Die Fakultät von 8 wäre 8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1. Man könnte daher also auch schreiben: 5! = 5 * 4! beziehungsweise 8! = 8 * 7!, somit lässt sich die Fakultät auch als rekursive Funktion berechnen.

```
16 long Fakultaet(long x)
17 {
18   long Ergebnis = 1;
19   if (x>0)
20     Ergebnis = x * Fakultaet(x-1);
21   return Ergebnis;
22 }
```

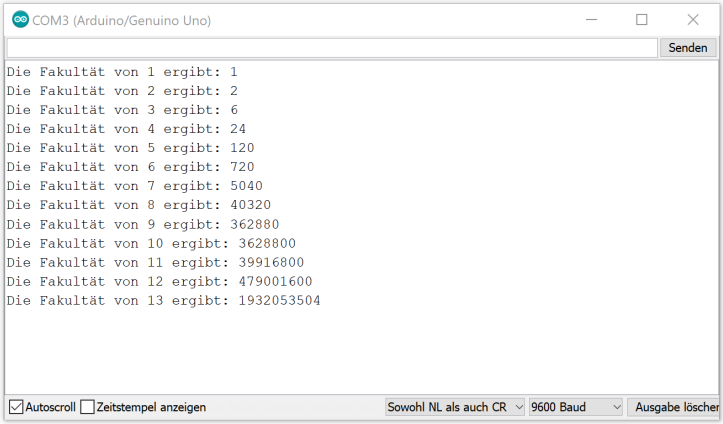


Abb. 19.7 Die Ausgabe am seriellen Monitor stellt die Fakultäten der Zahlen 1 bis 13 dar

Beim Aufruf der Funktion `Fakultaet` müssen im Arbeitsspeicher 4 Byte für die `long`-Variable `Ergebnis` und ebenfalls 4 Byte für die `long`-Variable `x` reserviert werden. Da sich die Funktion selbst aufrufen kann, müssen in so einem Fall weitere 8 Byte bereitgestellt werden – somit steigt der Speicherbedarf durch jede weitere Rekursion. Das Ausmaß dieses Aufwandes kann vom Compiler während der Übersetzung nicht vorhergesehen werden. Er gibt daher in seiner Statusmeldung lediglich an, wie viele Platz für derartige lokale Variablen verbleibt.

Als Faustregel können Sie sich daran orientieren, dass mindestens 50 Prozent des Arbeitsspeichers für lokale Variablen reserviert sein sollte. Natürlich sind Abweichungen möglich: Nutzt Ihr Sketch weder selbstdeklarierte Funktionen noch Bibliotheken und deklarieren Sie konsequent alle Variablen bereits außerhalb der Programmschleifen, so

können Sie die Quote der globalen Variablen gefahrlos bis auf 80 Prozent steigern. Jedoch sollten Sie sich im Klaren sein, dass ein zu geringer Speicherplatz für lokale Variablen (zu denen im Übrigen auch die Zählvariablen von `for`-Schleifen gehören) keinerlei Warn- oder Fehlermeldung erzeugt. Stattdessen wird Ihr Programm unvorhergesehenes Verhalten zeigen, weil Werte im Arbeitsspeicher nicht abgelegt werden können oder ungewollt überschrieben werden.

Nachfolgend betrachten wir einige „Stellschrauben", mit denen man ohne Einschränkung der Funktionalität den Speicherplatzbedarf reduzieren kann.

19.3.1 Reduzierung der Auslastung des Programmspeichers

Um den benötigten Programmspeicher zu reduzieren, empfiehlt es sich, mehrmals wiederkehrende Befehlsgruppen in selbstdefinierte Funktionen auszulagern. Dies wurde beispielsweise im Ampel-Praxisprojekt in Kapitel 6 angewendet. So wird die entsprechende Befehlsgruppe nur ein einziges Mal im Programmspeicher hinterlegt, statt mehrfach vorgehalten zu werden.

Für Befehle, welche direkt hintereinander mehrfach ausgeführt werden, sind natürlich Schleifen die bessere Wahl. Auch in diesem Fall wird die Gruppe nur einmal im Programmspeicher hinterlegt.

Wenn Sie zur Programmentwicklung und Fehlerbeseitigung den seriellen Monitor genutzt haben, ihn aber für die endgültige Funktionalität nicht mehr brauchen, sollten Sie auch die entsprechenden Befehle entfernen (oder per vorangestelltem `//` auskommentieren). Sonst übersetzt der Compiler im Hintergrund auch die zur Ansteuerung der seriellen Schnittstelle nötigen Funktionen und fügt sie dem Maschinencode hinzu – diesen unnötigen Ballast können Sie nach Fertigstellung des Programms über Bord werfen.

Ähnlich verhält es sich mit Bibliotheken, auch hier sollten Sie prüfen, welche Sie wirklich benötigen. Dabei hilft auch schon der Compiler: Er

prüft für jede Bibliotheksfunktion (und auch für jede selbstdefinierte Funktion), ob diese überhaupt verwendet wird. Taucht sie nirgends in Ihrem Sketch auf, wird sie auch nicht mit übersetzt.

Besteht Ihr Programmcode zu weiten Teilen aus Text, Grafiken oder anderen Daten, sollten Sie erwägen, diese in einen externen Speicher auszulagern (siehe Kapitel 13.1). Somit verbleibt auf dem Mikrocontroller mehr Platz für die eigentlichen Programmbefehle.

19.3.2 Reduzierung des benötigten Arbeitsspeichers

Auch zur Verringerung des benötigten Arbeitsspeichers gibt es Möglichkeiten: Prüfen Sie, ob die verwendeten Variablen wirklich ihren Wert ändern, also tatsächlich *variabel* sein müssen. Handelt es sich stattdessen um feste Zahlenwerte, sollten Sie diese als Konstanten definieren, denn dann werden Sie nur im Programmspeicher hinterlegt und verbrauchen nicht den wertvolleren (weil knapperen) RAM.

Wird der Speicher knapp, sollten Sie zudem prüfen, ob Ihre Variablentypen zu groß definiert sind. Nutzen Sie eine Zählvariable, die lediglich von 0 bis 100 läuft, braucht es dafür keinen `int`-Wert, stattdessen genügt `byte`. Kann eine Variable sogar nur zwei Zustände annehmen, genügt der Typ `boolean`, der nur ein einzelnes Bit belegt – alles andere wäre Verschwendung. Insbesondere bei der Nutzung von Arrays kann eine sparsame Wahl des Variablentyps große Speichervorteile bringen. Doch Vorsicht: Die Wahl eines zu kleinen Typs wird keine Fehlermeldung erzeugen, stattdessen treten falsche Ergebnisse auf.

Speichern Sie längere Zeichenketten (zum Beispiel zur Ausgabe auf einem Display), kann dies ebenfalls den Arbeitsspeicher belasten. Zeichenketten, die Sie beispielsweise per `Serial.print()` ausgeben, werden ebenfalls während des Funktionsaufrufes kurzzeitig als lokale Variablen in den RAM geladen. Es gibt jedoch Möglichkeiten, die Nutzung des Programmspeichers zu erzwingen:

```
1 const PROGMEM char Zeichenkette[] = {"Dieser Text steht im
2 Programmspeicher."};
```

```
3  char Zeichen;
4
5  void setup() {
6    Serial.begin(9600);
7  }
8
9  void loop() {
10   for (byte k = 0; k < strlen_P(Zeichenkette); k++)
11   {
12     Zeichen = pgm_read_byte(Zeichenkette + k);
13     Serial.print(Zeichen);
14   }
15   Serial.println();
16
17 Serial.println("Dieser Text wird bei der Ausgabe als lokales
18 Array zwischengespeichert.");
19 Serial.println(F("Dieser Text wird direkt aus dem
20 Programmspeicher ausgegeben."));
21
22   delay(10000);
23 }
```

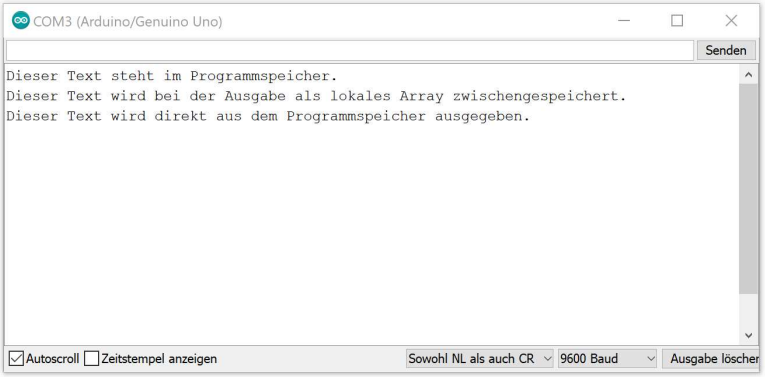


Abb. 19.8 Ausgabe am seriellen Monitor

Durch die Anweisung `const PROGMEM` wird dem Compiler bei einer Deklaration signalisiert, dass die nachfolgende Konstante im Programmspeicher hinterlegt werden soll, in diesem Fall handelt es sich dabei sogar um ein Array. Für den Zugriff benötigt man dann eine spezielle Funktion, in diesem Fall `pgm_read_byte()`. Diese liefert jedoch

immer nur ein einzelnes Byte, deshalb wird über eine Schleife die gesamte Zeichenkette durchlaufen. Die Funktion `strlen_P()` ist ebenfalls speziell auf Programmspeicher-Zugriffe ausgelegt und gibt die Länge der Zeichenkette zurück, wir nutzen sie daher um die Anzahl der Schleifendurchläufe festzulegen.

Das eben beschriebene Vorgehen empfiehlt sich, wenn eine bestimmte Zeichenkette mehrfach Verwendung findet. Wird sie nur einmal benötigt, kann man sie auch einfach als Argument einer Funktion notieren, so wie wir es bisher meistens bei Ausgaben per `Serial.print()` angewendet haben. Sie wird dann auch im Programmspeicher abgelegt. Allerdings muss sie zum Zeitpunkt des Aufrufs der Funktion einmalig komplett in den Arbeitsspeicher geladen werden. Besonders bei langen Zeichenketten kann dies dazu führen, dass dadurch der Platz für lokale Variablen erschöpft wird. Wie bereits erwähnt, erzeugt dies keine Fehlermeldung, sondern unvorhersehbares Programmverhalten. Um dem entgegenzuwirken, kann die Funktion `F()` genutzt werden. Vereinfacht gesagt erzwingt sie, dass das Funktionsargument direkt aus dem Programmspeicher geladen wird, ohne es im RAM zwischen zu speichern.

Zur Klarstellung sei noch erwähnt, dass die Länge von Variablen- oder Funktionsnamen keinerlei Einfluss auf irgendeinen Speicher hat. Der Compiler ersetzt diese Namen ohnehin durch die zugehörigen Speicheradressen. Natürlich haben auch Kommentare keinerlei Speicherauswirkungen auf dem Mikrocontroller, da diese beim Hochladen bereits durch den Präprozessor entfernt werden.

19.4 Zeitoptimierung

Weitere Verbesserungsmöglichkeiten gibt es oft beim zeitlichen Verhalten von Programmen. In allen unseren bisherigen Beispielen arbeitete der Arduino mit seiner Taktfrequenz von 16 Megahertz schnell genug, um keinerlei erkennbare Verzögerung zu erzeugen. Wirklich relevant wird das Zeitverhalten erst, wenn beispielsweise schnellstmöglich auf digitale Signale reagiert werden muss oder ein Roboter auf nur zwei Rädern balancieren soll und daher sehr rasch Lageränderungen aus-

gleichen muss. Auch bei der Erzeugung spezieller Signale, welche im Beispiel der Ansteuerung der adressierbaren LEDs aus Kapitel 7.7 eine Bibliothek übernahm, spielt das Zeitverhalten eine wichtige Rolle.

All dies übersteigt den Einsteiger-Bereich deutlich, daher sei nachfolgend nur ein kurzer Überblick einiger Optimierungsmöglichkeiten gegeben. Sollten Sie an einem Thema tieferes Interesse haben, finden Sie weitere Informationen in der Online-Referenz der Arduino-Plattform.[3]

19.4.1 Verkürzung von Algorithmen

Für jede Aufgabe gibt es unzählige mögliche Realisierungen in einem Sketch. Als Faustregel können Sie davon ausgehen, dass die Variante mit den wenigsten Befehlen/Funktionen auch die schnellste ist. So könnte man die Funktion `Fakultaet()` aus dem vorangegangenen Kapitel weiter „straffen“:

```
...
long Fakultaet(long x)
{
  return x ? x * Fakultaet(x-1) : 1;
}
```

Diese Notation hat exakt die gleiche Funktion wie die vorher verwendete, wird aber um ein Vielfaches schneller ausgeführt und benötigt weniger Arbeitsspeicher. Wesentlich dabei ist, dass die zuvor verwendete Hilfsvariable `Ergebnis` entfallen ist und die `if`-Anweisung durch eine bedingte Wertzuweisung (bekannt aus Kapitel 12.4) ersetzt werden konnte. Es wird allerdings auch offensichtlich, dass der Sketch dadurch besonders für Einsteiger viel schlechter nachvollziehbar ist. Verstehen Sie ihn so: Falls `x` „`true`“ ist (also ungleich 0), wird als Rückgabewert `x * Fakultaet(x-1)` gegeben, sonst `1`. Die enthaltene Rekursion wurde bereits erläutert. Nehmen Sie den originalen Sketch zum Vergleich, so werden Sie feststellen, dass diese Funktion damit genau das gleiche Verhalten zeigt wie das Original.

3 https://www.arduino.cc/reference/de/

Freilich macht sich die Zeitersparnis erst dann überhaupt bemerkbar, wenn diese Funktion tausende Male ausgeführt wird. In unserem simplen Beispiel erfolgt die Berechnung einer Fakultät ohne Optimierung im Bereich von wenigen Dutzend Mikrosekunden (millionstel Sekunden), mit der gezeigten Optimierung ist sie noch etwa dreimal schneller – ein normaler Anwender wird in diesem Fall aber keinen Unterschied bemerken. Würde diese Funktion jedoch genutzt, um komplexe Berechnungen anzustellen (beispielsweise eine Verschlüsselung), wäre der Unterschied sehr wohl spürbar.

Zur Beschleunigung der Programmabarbeitung trägt es ebenfalls bei, wenn nicht mehr benötigte Befehle zur seriellen Ausgabe (welche beispielsweise zur Fehlersuche benutzt wurden), entfernt werden. Das Senden von seriellen Daten benötigt vergleichsweise viel Zeit, in welcher das Programm nichts anderes tun kann. Kann auf die serielle Verbindung nicht verzichtet werden, lässt sich möglicherweise zumindest deren Geschwindigkeit erhöhen.

19.4.2 Vermeidung von Wartebefehlen

In den Praxisprojekten wurde bereits erläutert, wie der vorher oft genutzte `delay()`-Befehl sinnvoll durch `while`-Schleifen mit Timer-Variablen ersetzt werden kann. Dies bietet den großen Vorteil, dass die Wartezeit genutzt werden kann, um andere Aufgaben zu erledigen – zum Beispiel Nutzereingaben verarbeiten oder Zwischenberechnungen anstellen. Generell versucht man in komplexeren Projekten, den `delay()`-Befehl komplett zu vermeiden, da er das Programm „ins Koma“ versetzt, was insbesondere dann problematisch ist, wenn möglichst schnell auf (möglicherweise auch unerwartete) Reize von außen reagiert werden muss. Nutzen Sie Leerlaufzeiten daher lieber, um anliegende Signale zu prüfen oder auf Eingaben zu warten.

19.4.3 Nutzung von Interrupts

Treten in einem Projekt Signale auf, welche sofort bearbeitet werden müssen, werden diese in der fortgeschrittenen Programmierung durch Interrupts (Programmunterbrechungen) verarbeitet. Das Steuerwerk des Mikrocontrollers unterbricht dann sofort die reguläre Programmausführung und handelt zunächst die *Interrupt Service Routine* ab. Dies ist Ihnen in Kapitel 5.5 schon einmal begegnet, als es um den Datenaustausch mit externen Komponenten ging. Interrupts werden aber beispielsweise auch genutzt, um ein Tachosignal sofort zu verarbeiten, wenn die Geschwindigkeit einer rotierenden Welle verzögerungsfrei gemessen werden soll.

Es ergibt sich daraus der Vorteil, dass ein entsprechender Eingang nicht ständig abgefragt werden muss. Nachteilig ist, dass am Arduino UNO nur zwei Pins (2 und 3) Interrupts unterstützen. Ferner erfordert die Verwendung fortgeschrittene Programmierkenntnisse und birgt die Gefahr vieler unbeabsichtigter Fehler.

Andreas Sigismund asigismund@web.de

Downloadhinweis

Alle Programmcodes und Schaltpläne aus diesem Buch stehen kostenfrei zum Download bereit. Dadurch müssen Sie Code nicht abtippen.

Außerdem erhalten Sie die eBook Ausgabe zum Buch im PDF Format kostenlos auf unserer Website:

www.bmu-verlag.de/arduino-kompendium
Downloadcode: siehe Kapitel 20

Kapitel 20
Der Anfang ist getan

Werte Leserin, werter Leser, Sie haben nun das letzte Kapitel dieses Buches erreicht – und ich bedanke mich, dass Sie mir die Treue hielten!

Sie haben nun die Grundlagen der Elektronik und Programmierung kennengelernt, Sie können Mini-Roboter zum Leben erwecken und aus Lochrasterplatinen kleine Kunstwerke gestalten. Auch wenn wir aufgrund der Fülle der Themen vieles nur grundlegend betrachten konnten, hat sich womöglich Ihr Blick für die Zusammenhänge in dieser faszinierenden Welt der *Bits* und *Bytes*, der *Volts* und *Amperes* etwas geweitet – und vielleicht konnte ich Sie sogar etwas neugierig machen auf mehr.

Hat Ihnen ein Teilgebiet besonders Spaß gemacht? Zu allen Themen dieses Buches finden sich im Internet zahlreiche Blogs und Tutorials. Stöbern Sie doch ein wenig und staunen Sie über die Möglichkeiten! Sie werden überrascht sein, wie viel Sie bereits mit Ihren Grundkenntnissen nachvollziehen und verstehen können.

Wir befinden uns mitten in der größten technischen Revolution der Menschheitsgeschichte. Die Digitalisierung ist noch lange nicht abgeschlossen, da steht mit der künstlichen Intelligenz schon der nächste Technologiesprung vor uns. Wer durch technisches Verständnis und Interesse an der Wissenschaft diesen Wandel mitgestalten kann, braucht ihn nicht zu fürchten – sondern kann ihn als große Chance nutzen. Sie haben die ersten Schritte dafür nun getan.

Und was planen Sie als Nächstes? Vielleicht bauen Sie sich eine Heim-Wetterstation, welche Temperatur, Luft- und Bodenfeuchte nicht nur per Display, sondern auch auf dem Smartphone anzeigt? Oder eine indirekte LED-Beleuchtung, die mit automatischer tageszeitabhängiger Farbstimmung den Neid Ihrer Nachbarn erregt?

Lassen Sie Ihrer Kreativität freien Lauf! Ich wünsche Ihnen viel Freude am Experimentieren und zahlreiche erfolgreiche Hobby-Projekte – oder anders ausgedrückt:

Experimentierfreude++;

Ihr Danny Schreiter

Downloadcode für das kostenfreie eBook: q13xvi9zp

Anhang: Verwendete Komponenten / Bezugsquellen

Komponente / Bauelement	Bezugsquelle
Breadboard-Kit (mit Spannungsversorgungsmodul und Steckbrücken)	https://www.amazon.de/dp/B01N4VCYUK
Batterieclip	https://www.amazon.de/dp/B000OH5QT4/
Arduino UNO	https://www.az-delivery.de/collections/arduino-kompatible-boards/products/uno-r3?ls=de
Arduino Nano	https://www.az-delivery.de/collections/arduino-kompatible-boards/products/nano-v3-mit-ch340-arduino-kompatibel?ls=de
Diverse Widerstände	https://www.az-delivery.de/products/az-resistor-kit-525-widerstande?ls=de
LEDs	https://www.conrad.de/de/p/kemo-s036-led-sortiment-rot-gruen-gelb-182225.html
RGB-LED (KY-016)	https://www.az-delivery.de/products/led-rgb-modul?ls=de
7-Segment-Anzeige	https://www.conrad.de/de/p/lite-on-7-segment-anzeige-rot-14-22-mm-2-1-v-ziffernanzahl-1-lshd-5503-1127526.html
LED-Matrix	https://www.az-delivery.de/products/64er-led-matrix-display?ls=de
LCD	https://www.az-delivery.de/products/16x2-lcd-blaues-display?ls=de https://www.conrad.de/de/p/character-16x2-lcd-display-module-1602-black-on-green-5v-i2c-interface-hd44780-802231375.html
OLED-Display (128x64, SSD 1306)	https://www.az-delivery.de/products/0-96zolldisplay?ls=de
Martix aus adressierbaren LEDs (WS2812)	https://www.az-delivery.de/products/u-64-led-panel?ls=de

Komponente / Bauelement	Bezugsquelle
Folientastatur (4x4)	https://www.conrad.de/de/p/folientastatur-tastenfeld-matrix-4-x-4-apem-ac3561ill-1-st-709000.html
Infrarot-Empfänger (KY-022)	https://www.az-delivery.de/products/ir-empfanger-modul?ls=de
Universalfernbedienung (Vivanco UR 2)	https://www.amazon.de/Vivanco-UR-2-Universalfernbedienung-grau-Silber/dp/B000BVX7OY
Fotowiderstand (GL5528)	https://www.az-delivery.de/products/fotowiderstand-photo-resistor-dioden-150v-5mm-ldr5528-gl5528-5528-50pcs?ls=de
Bewegungsmelder	https://www.az-delivery.de/products/bewegungsmelde-modul?ls=de
Bodenfeuchte-Sensor	https://www.az-delivery.de/products/bodenfeuchte-sensor-modul-v1-2?ls=de
Temperatur- und Feuchtesensor (DHT22)	https://www.az-delivery.de/products/dht22-temperatursensor-modul?ls=de
Ultraschall-Abstandssensor (HC-SR04)	https://www.az-delivery.de/products/3er-set-hc-sr04-ultraschallmodule?ls=de
Hall-Sensor (KY-024)	https://www.az-delivery.de/products/hall-sensor-modul?ls=de
Beschleunigungssensor (GY-521)	https://www.az-delivery.de/products/gy-521-6-achsen-gyroskop-und-beschleunigungssensor?ls=de
Kompassmodul (GY-271)	https://www.az-delivery.de/products/gy-271-kompassmodul-kompass-magnet-sensor-fuer-arduino-und-raspberry-pi?ls=de
Echtzeitmodul (DS3231)	https://www.az-delivery.de/products/ds3231-real-time-clock?ls=de
Relais-Modul	https://www.az-delivery.de/collections/raspberry-pi-zubehor/products/relais-modul?ls=de

Komponente / Bauelement	Bezugsquelle
Gleichstrommotor	https://www.amazon.de/DollaTek-10Pcs-Motor-Elektromotor-Spielzeug/dp/B07HBMQ4GF/ https://www.amazon.de/Cylewet-Gear-Schaft-Arduino-clw1037/dp/B06XSJS8W6/
Gleichstrommotor-Treiber (L293D)	https://www.amazon.de/10-L293D-Stepper-Motor-Treiber/dp/B008DBU3S2/
Servomotor (SG90)	https://www.az-delivery.de/products/az-delivery-micro-servo-sg90?ls=de
Schrittmotor (28BYJ) mit Treiberplatine (ULN2003A)	https://www.amazon.de/Neuftech-Schrittmotor-Stepper-28BYJ-48-Treiberplatine/dp/B00NW4X25G/
Elektromagnet	https://www.amazon.de/dp/B07G323C31/
aktiver Summer (KY-012)	https://www.az-delivery.de/products/buzzer-modul-aktiv?ls=de
passiver Summer (KY-006)	https://www.az-delivery.de/products/buzzer-modul-passiv?ls=de
SD-Karten-Modul	https://www.az-delivery.de/products/copy-of-spi-reader-micro-speicherkartenmodul-fur-arduino?ls=de
SD-Karten-Shield	https://www.az-delivery.de/products/datenlogger-modul?ls=de
Motor-Treiber-Shield	https://www.az-delivery.de/products/4-kanal-l293d-motortreiber-shield-schrittmotortreiber?ls=de
Ethernet Shield	https://www.az-delivery.de/products/ethernet-shield-w5100?ls=de
ESP32 (Development Module)	https://www.az-delivery.de/products/esp32-developmentboard?ls=de
ESP8266	https://www.az-delivery.de/products/esp8266-02?ls=de
NodeMCU WiFi Development Board	https://www.amazon.de/dp/B074Q2WM1Y/
D1 Mini	https://www.az-delivery.de/products/d1-mini?ls=de

Komponente / Bauelement	Bezugsquelle
RoboCar-Set	https://www.amazon.de/dp/B07DNX1DX9
ATmega328	https://www.reichelt.de/arduino-atmega328-mit-arduino-bootloader-ard-atmega-328-p230602.html
16-MHz-Quarz	https://www.reichelt.de/standardquarz-grundton-16-000000-mhz-16-0000-hc49u-s-p32852.html
Kondensator 22 pF	https://www.reichelt.de/keramik-kondensator-22p-kerko-22p-p9281.html
Lötkolben mit Zubehör	https://www.amazon.de/dp/B07B3RKZLX/

Anhang: Codereferenz

Nachfolgend sind häufig genutzte Befehle und Funktionen aufgelistet, jedoch ohne Anspruch auf Vollständigkeit. Für eine umfassendere Befehlsreferenz sei auf die Arduino online-Referenz (https://www.arduino.cc/reference/) sowie die Dokumentation der genutzten Bibliotheksdateien verwiesen, welche über entsprechende Suchmaschinen gefunden werden können.

abs()	liefert den Betrag (Absolutwert) einer Zahl zurück
analogRead()	liest einen Analogwert ein
analogWrite()	gibt ein PWM-Signal an einem Ausgangspin aus
atoi()	*ASCII to Integer* – wandelt eine Zeichenkette in eine Zahl um
break	verlässt sofort die übergeordnete Schleife oder switch-Anweisung und springt an deren Ende
byte	Ganzzahl-Variable mit 8 Bit
case	Sprungmarkierung einer switch-Anweisung
char	Ganzzahl-Variable mit 8 Bit inklusive Vorzeichen, für ASCII-Zeichencodierung verwendet
const	definiert die nachfolgende Zuweisung als Konstante (auf Compiler-Ebene)
continue	springt sofort an das Ende der übergeordneten Schleife, verlässt diese jedoch nicht
#define	definiert eine Konstante auf Präprozessor-Ebene
delay()	pausiert das Programm für die übergebene Zahl an Millisekunden
delayMicroseconds()	pausiert das Programm für die übergebene Zahl an Mikrosekunden
digitalRead()	liest ein binäres Signal von einem Eingangspin
digitalWrite()	setzt einen digitalen Ausgangspin auf HIGH oder LOW

EEPROM	Objekt, welches den internen nicht flüchtigen Speicher (EEPROM) repräsentiert
else	alternativer Fall (als Ergänzung einer if-Anweisung)
Ethernet	Objekt, welches die kabelgebundene Netzwerkschnittstelle repräsentiert
float	Fließkommazahl-Variable mit 32 Bit
for	Schleife
#include	veranlasst den Präprozessor, Programmcode aus einer anderen Datei zu importieren
if	bedingte Anweisung
indexOf()	sucht in einem String-Objekt nach der übergebenen Zeichenkette
int, integer	Ganzzahl-Variable mit 16 Bit
loop()	Hauptschleife eines Sketches
map()	rechnet einen Zahlenwert vom Ausgangswertebereich in einen anderen Wertebereich um
micros()	liefert die Zahl der seit Programmstart vergangenen Mikrosekunden
millis()	liefert die Zahl der seit Programmstart vergangenen Millisekunden
pinMode()	konfiguriert das Verhalten eines Anschlusspins als Ein- oder Ausgang
return	beendet eine Funktion und übergibt den (optionalen) Rückgabewert
Serial.begin()	startet eine serielle Datenverbindung mit der übergebenen Geschwindigkeit
Serial.print()	gibt das Argument über die serielle Verbindung aus
Serial.println()	gibt das Argument über die serielle Verbindung aus und fügt einen Zeilenumbruch an
Serial.read()	liest Daten vom seriellen Eingang
setup()	Initialisierungsteil eines Sketches, wird nur einmal durchlaufen
sprintf()	formatiert Zahlen in einer Zeichenkette und schreibt das Ergebnis in ein char-Array

static	markiert eine Variable innerhalb einer Funktion als statisch, sie behält ihren Wert auch für weitere Aufrufe dieser Funktion
switch()	Verzweigung mit mehreren möglichen Pfaden
SPI	Objekt, welches die SPI-Schnittstelle repräsentiert
toCharArray()	wandelt ein String-Objekt in eine Zeichenkette vom Typ char-Array
void	deklariert eine Funktion ohne Rückgabewert
volatile	Schlüsselwort, um Variablen zu kennzeichnen, welche auch von ISRs verändert werden können
while	Schleife, deren Wiederholung an eine Bedingung geknüpft ist
Wire	Objekt, welches die I^2C-Schnittstelle repräsentiert
//	Kommentar
/* .... */	mehrzeiliger Kommentar

Operatoren werden in folgender Prioritätenreihenfolge (beginnend bei 1) abgearbeitet:

Priorität	Operator	Bedeutung
1	::	Gültigkeitsbereichswahl (objektorientierte Programmierung)
2	++ -- () [] . ->	Post-Inkrement (rechts neben dem Variablennamen) Post-Dekrement (rechts neben dem Variablennamen) Funktionsaufruf Array-Referenzierung Element-Auswahl (objektorientierte Programmierung) Element-Auswahl per Zeiger (objektorientierte Programmierung)

3	++ -- + - ! ~ (Typ) * & sizeof new delete	Prä-Inkrement (links neben dem Variablennamen) Prä-Dekrement (links neben dem Variablennamen) Vorzeichen Vorzeichen Negation Negation (bitweise) Typumwandlung Dereferenzierung Adress-Operator Größenbestimmung dynamische Speicherbelegung (objektorientierte Programmierung) dynamische Speicherbelegung (objektorientierte Programmierung)
4	.* ->*	Zeiger (objektorientierte Programmierung) Zeiger (objektorientierte Programmierung)
5	* / %	Multiplikation Division Modulo-Division
6	+ -	Addition Subtraktion
7	<< >>	bitweise nach links verschieben bitweise nach rechts verschieben
8	< > <= >=	kleiner als größer als kleiner gleich größer gleich
9	== !=	gleich ungleich
10	&	bitweise Und
11	^	bitweise Exklusiv-Oder
12	|	bitweise Oder
13	&&	logisches Und
14	||	logisches Oder

15	? :	Bedingte Zuweisung
	=	Zuweisung
	+=	mit Addition kombinierte Zuweisung
	-=	mit Subtraktion kombinierte Zuweisung
	*=	mit Multiplikation kombinierte Zuweisung
	/=	mit Division kombinierte Zuweisung
	%=	mit Modulo-Division kombinierte Zuweisung
	<<=	mit Bitverschiebung kombinierte Zuweisung
	>>=	mit Bitverschiebung kombinierte Zuweisung
	&=	mit Und kombinierte Zuweisung
	^=	mit Exklusiv-Oder kombinierte Zuweisung
	|=	mit Oder kombinierte Zuweisung
16	,	Aufzählung von Argumenten

Anhang: Bildquellen

Bild/Grafik	Quelle und Lizenz
Abbildung 2.2	MakeMagazinDE (https://commons.wikimedia.org/wiki/File:Arduino_nano.jpg), https://creativecommons.org/licenses/by-sa/4.0/legalcode
Abbildung 2.3	David Mellis (https://commons.wikimedia.org/wiki/File:Arduino_Mini.jpg), „Arduino Mini“, https://creativecommons.org/licenses/by/2.0/legalcode
Abbildung 2.4	Geek3 (https://commons.wikimedia.org/wiki/File:Arduino_Micro.jpg), „Arduino Micro“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 2.5	Dsimic (https://commons.wikimedia.org/wiki/File:Arduino_MEGA_2560_R3,_front_side.jpg), https://creativecommons.org/licenses/by-sa/4.0/legalcode
Abbildung 2.6	David Mellis (https://commons.wikimedia.org/wiki/File:LilyPad_Arduino.jpg), „LilyPad Arduino“, https://creativecommons.org/licenses/by/2.0/legalcode
Abbildung 2.7	Antonireykern (https://commons.wikimedia.org/wiki/File:Gemma-arduino-compatible-wearable-microcontroller-1.jpg), https://creativecommons.org/licenses/by-sa/4.0/legalcode
Abbildung 3.4	Afrank99 (https://commons.wikimedia.org/wiki/File:3_Resistors.jpg), „3 Resistors“, https://creativecommons.org/licenses/by-sa/2.5/legalcode
Abbildung 3.6	Iainf (https://commons.wikimedia.org/wiki/File:Potentiometer.jpg), „Potentiometer“, https://creativecommons.org/licenses/by-sa/3.0/legalcode und Pemu (https://commons.wikimedia.org/wiki/File:PCB_variable_resistors.jpg), „PCB variable resistors“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 3.10	Cepheiden (https://commons.wikimedia.org/wiki/File:Plate_Capacitor_DE.svg), „Plate Capacitor DE“, https://creativecommons.org/licenses/by-sa/3.0/legalcode

Bild/Grafik	Quelle und Lizenz
Abbildung 3.11 (3 Fotos)	Elcap (https://commons.wikimedia.org/wiki/File:MLCC-Scheiben-Kerkos-P1090142c.jpg und https://commons.wikimedia.org/wiki/File:Wiki-Folkos-P1090317-1.jpg und https://commons.wikimedia.org/wiki/File:Wiki-Ta-und-Al-Elkos-P1090329-1.jpg), „MLCC-Scheiben-Kerkos-P1090142c“, „Wiki-Folkos-P1090317-1“, „Wiki-Ta-und-Al-Elkos-P1090329-1“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 3.14 (Aufreihung von LEDs)	Afrank99 (https://commons.wikimedia.org/wiki/File:Verschiedene_LEDs.jpg), „Verschiedene LEDs“, https://creativecommons.org/licenses/by-sa/2.0/legalcode
Abbildung 3.17	Honina (https://commons.wikimedia.org/wiki/File:Relais.JPG), „Relais“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 3.29	Johan~commonswiki (https://commons.wikimedia.org/wiki/File:Solder.jpg), „Solder“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 3.30 (rechtes Bild)	MyName/Coronium (https://commons.wikimedia.org/wiki/File:Cold_solder_joint2.jpg), „Cold solder joint2“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 3.37	André Karwath aka Aka (https://commons.wikimedia.org/wiki/File:Digital_Multimeter_Aka.jpg), „Digital Multimeter Aka“, https://creativecommons.org/licenses/by-sa/2.5/legalcode
Abbildung 3.38	smial (https://de.wikipedia.org/wiki/Oszilloskop#/media/File:Digitaloszilloskop_IMGP1971_WP.jpg), „Digitales Oszilloskop“, http://artlibre.org/licence/lal/de/
Abbildung 5.13	Chris828 (https://commons.wikimedia.org/wiki/File:RS-232_timing.svg), https://creativecommons.org/licenses/by-sa/4.0/legalcode

Bild/Grafik	Quelle und Lizenz
Abbildung 5.16 (2 Grafiken)	Cburnett (https://commons.wikimedia.org/wiki/File:SPI_three_slaves_daisy_chained.svg und https://commons.wikimedia.org/wiki/File:SPI_three_slaves.svg), „SPI three slaves daisy chained“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 11.9	Racso (https://commons.wikimedia.org/wiki/File:TiempsServo.svg), https://creativecommons.org/licenses/by-sa/4.0/legalcode
Abbildung 13.1	Quickfix (https://commons.wikimedia.org/wiki/File:National_NM27C256.jpg), „National NM27C256“, https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 16.1	Deadlyhappen (https://de.wikipedia.org/wiki/Datei:Stern-Stern-Netz.pdf (verändert)), https://creativecommons.org/licenses/by-sa/3.0/legalcode
Abbildung 16.8	Brian Krent (https://commons.wikimedia.org/wiki/File:Espressif_ESP-WROOM-32_Wi-Fi_&_Bluetooth_Module.jpg), https://creativecommons.org/licenses/by-sa/4.0/legalcode
Abbildung 17.4 (linkes Foto)	oomlout (https://commons.wikimedia.org/wiki/File:ATMEGA328P-PU.jpg), „ATMEGA328P-PU“, https://creativecommons.org/licenses/by-sa/2.0/legalcode
Abbildung 17.4 (rechtes Foto)	oomlout (https://commons.wikimedia.org/wiki/File:ICIC-TQ32-X-K328-01_(16421989932).jpg), „ICIC-TQ32-X-K328-01 (16421989932)“, https://creativecommons.org/licenses/by-sa/2.0/legalcode

Hier nicht genannte Abbildungen wurden vom Autor selbst angefertigt oder sind gemeinfrei. Alle schematischen Breadboard-Ansichten stammen aus der Software „Fritzing“ (www.fritzing.org).

Anhang: Softwareverzeichnis

Die nachfolgende Tabelle gibt einen Überblick über die verwendete Software und deren Bezugsquellen. Auf den angegebenen Webseiten finden Sie auch Möglichkeiten, für die freien Projekte zu spenden.

Software	Zweck	Entwickler / Quelle
Arduino IDE	Codebearbeitung, Compiler	Arduino www.arduino.cc/
Fritzing	Projektplanung, Platinenerstellung	Friends-of-Fritzing e.V. www.fritzing.org/
EAGLE	Platinenerstellung	CadSoft Computer GmbH www.autodesk.de/products/eagle/free-download
Processing	PC-Interaktion	Ben Fry und Casey Reas www.processing.org/

Verwendete Arduino-Bibliotheken:

Bibliothek	Zweck	Autor / Quelle
Wire.h	I²C-Kommunikation	Arduino / vorinstalliert
SPI.h	SPI- Kommunikation	Arduino / vorinstalliert
RCSwitch.h	433-MHz-Funkmodul	sui77 / Bibliotheksverwalter
LedControl.h	LED-Matrix	Eberhard Fahle / Bibliotheksverwalter
LiquidCrystal_I2C.h	LCD	Frank de Brabander / Bibliotheksverwalter
Adafruit_SSD1306.h	OLED-Display	Adafruit / Bibliotheksverwalter
FastLED.h	adressierbare LEDs	Daniel Garcia / Bibliotheksverwalter
Keypad.h	Folientastatur	Mark Stanley, Alexander Brevig / Bibliotheksverwalter

IRremote.h	IR-Fernbedienung	shirriff / Bibliotheksverwalter
DHT.h	Temperatursensor	Adafruit / github.com/adafruit/Adafruit_Sensor
HCSR04.h	Ultraschallsensor	Martin Sosic / Bibliotheksverwalter
MPU6050_tockn.h	Beschleunigungssensor	tockn / Bibliotheksverwalter
MechaQMC5883.h	Kompass	mechasolution / github.com/mechasolution/Mecha_QMC5883L
DS3231.h	Echtzeitmodul	Andrew Wickert / Bibliotheksverwalter
Servo.h	Servomotor	Michael Margolis / vorinstalliert
Stepper.h	Schrittmotor	Arduino / vorinstalliert
EEPROM.h	interner EEPROM	Arduino / vorinstalliert
uEEPROMlib.h	externer EEPROM	Naguissa / Bibliotheksverwalter
SD.h	SD-Shield	Arduino, SparkFun / vorinstalliert
Ethernet.h	Ethernet-Shield	Arduino / vorinstalliert
PubSubClient.h	MQTT-Client	Nick O'Leary / Bibliotheksverwalter

Anhang: Index

Symbole

24C32 320
74HC595 181

A

Abisolierzange 64
abs() 294
Adresse 106, 108
Adressierbare LEDs 200
Aktor 269
Algorithmus 11
Ampel 162
Ampere 30
Analoger Eingang 130
analogRead 132
analogWrite 136
Anode 45
Antenne 155
Anzeigeelement 175
Array 103
ATmega328 ..... 52, 495
atoi 466

B

Backpack 191
Basis 47
Baudrate 140
Bedingung 79
Beschleunigungssensor 243
Betrag (mathematisch) 294
Bewegungsmelder 224
Bibliothek 27
Biegehilfe 66
Bildschirmausgabe 100
Binärsystem 84
Blink 71
Boardverwalter 431
Bodenfeuchte-Sensor 229
Bootloader 53
Breadboard 57
break 172
browser 416
Bus 142
Buzzer 292
byte 87

C

case 172
char 87
Client 414
Compiler 72
const 546

D

Datenreihe 103
Datenübertragung 139
Datenverarbeitung 315
Datenyp 83
Debugging 531
define 78
delay 76
delayMicroseconds 558
Destruktor 382
Dezimalsystem 83
DHCP 414
DHT11/22 231
Dielektrikum 42, 229

Dienstgüte 456
digital 121, 125
digitalRead....129
digitalWrite....76
dimmen....134
diode....45
Diode....43, 175
draw....337
Drehratensensor 243
Drehregler 132
Dritte Hand 66
Dual-Inline-Package 52
Dualzahl....84

E

EAGLE 517
Echosignal 237
Echtzeitmodul....251
EEPROM 316
Elektromagnet....288
else 79
Emitter 47
Entlötsaugpumpe....63
ESP32 428
ESP8266 440
Ethernet 418

F

Farad....42
Fehlersuche....531
Feld (Array)....103
Feldeffekt-Transistor 47
Fernbedienung....218
Fernbedienungscodes 304
Fließkommazahl....89
float....87
Flüssigkristallanzeige 189
Folientastatur 215
for 94
Formatieren....327
Fotowiderstand 222
Freilaufdiode 51
Fritzing....508
führende Null....483
Funktion 111

G

Ganzzahlvariable 87
Gemma 19
Geschichte 11
Git....433
Gleichstrommotor....273
GND (Ground)....32
GPIO 54, 430
Gyroskop 243

H

Hall-Sensor....238
Hardware 30, 52
Hauptschleife 77
H-Brücke 274
HD44780 191
Heimnetz 413
HIGH 76
Hochladen 71
HTML 416
HTTP 416

I

I²C....141
IC (Integrated Circuit)....285
IDE 21
if-Anweisung....80
Impulslänge 280
include....144
indexOf....425
Infrarotempfänger 218

Input 125
Installation 22
int 87
Integer 87
Internet 410
Internet of Things 410
Interrupt 146, 159, 550
Interrupt Service Routine. 147, 550
Inversion 149
I/O-Pins 54, 430
IOREF 57
IP-Adresse 411
ISR 147, 550

K

Kapazität 42
kapazitiver Sensor 230
Kathode 43
Kennlinie 44
Klasse 378
Kollektor 47
Kommentar 74
Konstante 78
Konstantstromquelle 177
Konstruktor 382
Kontrollstruktur 79

L

L293D 276
Last 49
Laststromkreis 269
Laufrichtung 276
LCD 189
LED, Leuchtdiode 45, 175
Leistung 33
Library 27
LillyPad 18
Lochrasterplatine 64
Logik 82
loop 70
löten 59
Lötpad 191
LOW 76
Luftfeuchte 231

M

MAC- 415
Magnetfeld 288
Maschinencode 72
Masse 32
Master-Slave 142
Matrix 184
Methode 406
micros 559
Mikrocontroller 52
millis 80
MISO 148
Modulodivision 114
MOSI 148
Mosquitto 457
Motor 273, 278, 282

N

Nano 16
NAT 413
NC 270
Network Address Translation 413
Netzwerk 410
NO 270
Noname-Hersteller 493
NPN 47

O

Objektorientierung 115
Ohm 36
Ohmsches Gesetz 36
Oktalsystem 85

OLED-Display 195
Onboard-LED 75
Open Source 13
Output 121

P

Parallelschaltung 35
Parksensor 235
Pegel 139, 451
Pegelanpassung 451
Periodendauer 134
Piezzo-Effekt 291
pinMode 75
PIR-Sensor 225
Platine 507, 527
Plattenkondensator 40
PNP 47
Potential 31
Potentiometer 38
Präfix 85
Präprozessor 73
Processing 330
Programm 69
Programmspeicher 53
Prozessor 52
Pseudo-Zufallszahl 179
Pull-Up-Widerstand 127
Pulsweitenmodulation 134

Q

QoS 456
Quarz 54

R

random 179
Raspberry Pi 14
R-C-Glied 138
Rechenwerk 53
Reihenschaltung 35
Relais 49, 269
Reset-Pin 56
return 115
RGB 180, 201
Richtungsumkehr 284
RoboCar 296
Router 412
Routine 147, 550
RS-232 140
RTC 252
Rückgabewert 80
Rx 99

S

Schaltplan (Stromlaufplan) 32
Schieberegister 181
Schleife 93
Schrittmotor 287
SCL 143
SDA 143
SD-Karte 324
Seitenschneider 65
Sensor 215
Serial 100
Serieller Anschluss 97, 139
serieller Monitor 98
Servo 278
setup 70
Shield 19, 324, 418
Sieben-Segment-Anzeige 180
Silizium-Diode 44
Sketch 71
SMD 191, 496
Spannung 30
Spannungsglättung 41
Spannungspegel 139
Spannungsquelle 32
Spannungsteiler 37

Speicher 53, 107
SPI 148
sprintf 341
static 356
Steckbrett 57
Stepper 287
Steuerwerk 53
Stromkreis 32
Stromstärke 30
Summer 291
switch 172

T

Taktfrequenz 72, 450
Tastatureingabe 102
Taste 171
Taster 32, 128
Tastverhältnis 135
Thermometer 231, 258
Tilde 135
Timer 171, 221
toCharArray 466
Transistor 46
Treiber 274
Tx 99

U

UART 139
Überlauf 88
übersetzen 73
Uhr 251
ULN2003A 285
Ultraschallsensor 235
Universalfernbedienung 303
Unterprogramm 111
USB 99
USB-zu-seriell-Wandler 55

V

Variable 83
Ventilator 273
Vergleichsoperator 81
Versorgungsspannung 33, 497
Verzweigung 79, 172
Vielfachmessgerät 67
void 113
volatile 146
Volt 31
Voltmeter 67
Vorwiderstand 45

W

Watt 33
Wendelpotentiometer 238
Wertebereich 87
while 96
Widerstand 35
WiFi-Shield 428
Winkelsensor 279
Wire 141

Z

Zahlensystem 83
Zähler 94
Zeiger 108
ZIP 29
Zufallszahl 179
Zuweisung 91

Andreas Sigismund asigismund@web.de

Eine umfassende Anleitung zum Arduino!

Die Arduino-Plattform bietet Bastlern die Möglichkeit auch ohne umfassende Elektronik- und Programmiervorkenntnisse eigene intelligente Mikrocontrollerprojekte umzusetzen. Mit diesem Buch lernen Sie praxisnah die notwendigen Grundlagen im Bereich des Programmierens und der Elektronik, um bald eigene spannende Projekte mit dem Arduino basteln zu können. Neben allen relevanten Zubehörteilen wie Sensoren, Aktoren, Displays und Shields werden auch fortgeschrittene Themen wie moderne MQTT Smart Home Systeme, Arduino-Roboter und die Datenverarbeitung und Steuerung über Processing-Programme am PC behandelt. So können Sie die Möglichkeiten des Arduino voll ausnutzen!

Ihr neues Buch im Überblick:

Inhalte

- Grundlagen der Elektronik verständlich erklärt
- Alle wichtigen Sensoren, Aktoren, Displays, Shields und Zubehörteile umfassend vorgestellt
- Programmiergrundlagen in C++
- Arduino-Programmierung über die Arduino IDE
- Datenverarbeitung und Steuerung über Processing-Programme am PC
- Arduino mit dem Internet verbinden, MQTT-Internetanwendungen
- Erstellung eigener Platinen, um Projekte für den professionellen Einsatz zu entwickeln
- Effizientes Debugging, um Fehler schnell zu beheben

Vorteile

- Solides Hintergrundwissen für eigene Projekte durch Erläuterung aller Elektronik- und Programmiergrundlagen
- Einfache, praxisnahe Erklärungen tragen zum schnellen Verständnis bei
- Einsatzbeispiele helfen, das Gelernte anzuwenden und sichern den nachhaltigen Lernerfolg
- Umfangreiche Praxisprojekte wie Arduino-Roboter, Smart Home Anwendungen und Arduino-Wecker dienen als Vorlagen für eigene Projekte
- Alle Schaltpläne und Quellcodes kostenfrei zum Download verfügbar

Mit diesem Buch erhalten Sie eine umfassende Einführung zum Arduino, um eigene Projekte mit dem Arduino realisieren zu können!

Über den Autor

Seit dem Abschluss seines Ingenieurstudiums realisierte Danny Schreiter zahlreiche Elektronik- und Software-Projekte für professionelle Auftraggeber und befreundete Hobby-Bastler. Er blickt mittlerweile auf mehr als 15 Jahre Erfahrung im Bereich der Mikroelektronik und Programmierung zurück. Neben seinem Job als Nachrichtentechnik-Ingenieur gibt er als Dozent seine Begeisterung an Studenten weiter.

{BMU VERLAG}

9 783966 450171