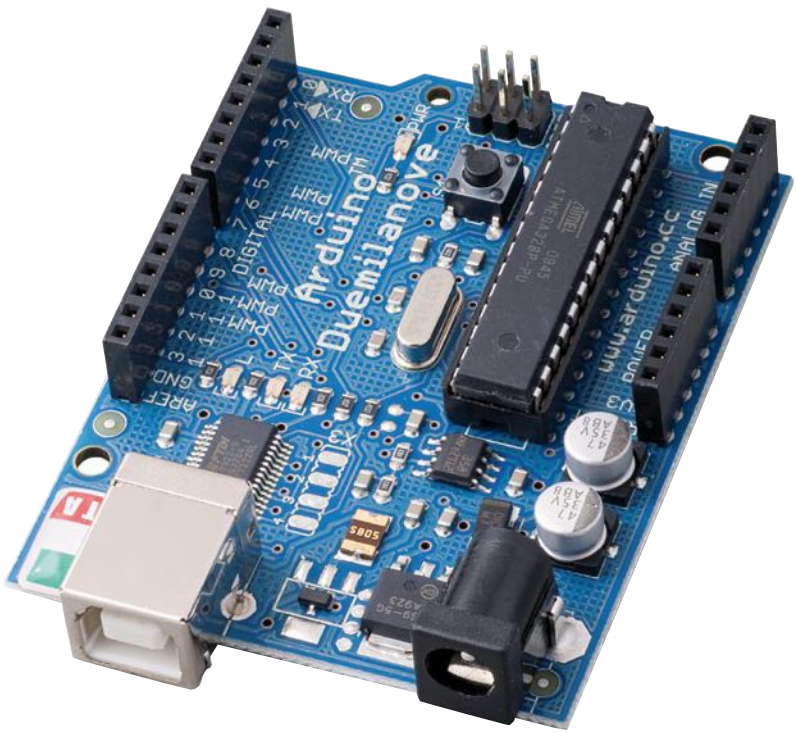
2. Auflage

Manuel Odendahl,
Julian Finn & Alex Wenger

Arduino

Physical Computing für Bastler, Designer & Geeks

o'reillys basics



- Microcontroller-Programmierung für alle
- Rapid Prototyping
- Mit kompletter Programmiersprachenreferenz

O'REILLY®

2. Auflage

Arduino – Physical Computing für Bastler, Designer und Geeks

Manuel Odendahl, Julian Finn & Alex Wenger

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Taipei · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

1. Auflage 2009
2. Auflage 2010

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Lektorat: Volker Bombien, Köln
Korrektorat: Kathrin Jurgenowski, Köln
Satz: III-satz, Husby
Umschlaggestaltung: Michael Oreal, Köln
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-89721-995-3

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Einleitung

In diesem Kapitel:

Vielleicht stehen Sie gerade im Laden und überlegen sich, ob Sie dieses Buch kaufen möchten. Vielleicht haben Sie es auch schon erworben, wofür wir uns natürlich recht herzlich bedanken. In jedem Fall aber halten Sie gerade das erste deutschsprachige Buch zum Thema Arduino in den Händen. Arduino ist die Verbindung aus einem Board, das einen Mikrocontroller beherbergt, mit einer Programmiersprache, die es sehr leicht macht, diesem kleinen Prozessor auch Befehle zu erteilen. Zusammen bilden die beiden die Basis für sogenanntes Physical Computing, die Verbindung der realen, physischen Welt mit der virtuellen Welt der Einsen und Nullen.

Der Arduino eignet sich hervorragend für alle möglichen Projekte, denn er ist günstig zu erwerben und einfach und vor allem schnell zu programmieren. So wird die Zeit von der fixen Idee bis zur Umsetzung möglichst kurz gehalten. Das bewahrt viele gute Projekte vor dem Papierkorb, in dem sie unweigerlich landen, wenn ihre Schöpfer vor allzu großen Problemen stehen.

Der Arduino wurde im italienischen Ivrea entwickelt, um Designstudenten möglichst leicht zu bedienende Werkzeuge an die Hand zu geben, und hat seit 2005 einen fast beispiellosen Siegeszug angetreten. Rund um den Globus finden sich nun an Kunsthochschulen Seminare zur Programmierung, zum Löten und über physikalische Grundlagen, begünstigt durch die einfache Verfügbarkeit dieser Technologie. Mehr über die Geschichte des Arduino finden Sie in Kapitel 1.

Aber vorweg – um was geht es hier eigentlich genau? Arduino ist zum einen eine Plattform, die aus einem Mikrocontroller besteht, der ähnlich wie ein PC eigenständig Berechnungen durchführen

kann, aber auch leicht mit vielen Sensoren zur Interaktion mit der Umwelt verbunden werden kann. Arduino ist aber auch die dazugehörige Entwicklungsumgebung, und – nicht zu vergessen – das Framework, das viele komplizierte Details der hardwaregestützten Entwicklung einfacher macht. So kann mit einem einzelnen Knopfdruck der Programmcode sowohl kompiliert als auch direkt über USB in den Arduino geladen werden.

All das erlaubt die schnelle Entwicklung von Prototypen; aber dadurch, dass alle Teile auf bestehenden Standards wie C++ und verbreiteten Mikrocontrollern basieren, kann es später auch losgelöst vom Arduino-Konzept weiterverwendet werden.

Für wen ist dieses Buch gedacht?

Sie haben die Einleitung gelesen und fragen sich nun, ob Sie zu den »Bastlern, Designern und Geeks« gehören, die der Buchtitel anspricht? Nun, wahrscheinlich schon. Dieses Buch versucht, seinen Leserinnen und Lesern die Welt des Physical Computing und damit auch der Programmierung anhand von praktischen Beispielen näherzubringen.

Sie alle haben gemeinsam, dass sie fortan Neues entwickeln wollen, entweder in der realen Welt als Gegenstand oder virtuell auf dem Rechner als Daten und Software. Hier fügen sich mit Arduino zwei Welten zusammen, die eigentlich nie richtig voneinander getrennt waren, und doch ist das Überqueren dieser Grenze für viele eine hohe Hürde. Der Programmierer scheut den Lötkolben und löst Probleme lieber auf seine Weise, wogegen der Bastler lieber noch drei weitere Bausätze verwendet, als dass er sich mit der Programmierung eines Computers oder Mikrocontrollers auseinandersetzt.

Bastlern, die bisher vielleicht nur in der physischen Welt an ihren Projekten gearbeitet haben, werden dadurch neue Möglichkeiten eröffnet: Sie können die im Buch aufgezeigten Ideen verfolgen und dabei lernen, wie der Arduino zu ihrem Hobby viele neue Aspekte beisteuern kann.

Als Designer und Medienkünstler werden Sie bislang möglicherweise ähnliche Probleme gehabt haben: Die Inspiration war da, bloß an der Umsetzung hat es gehapert. Mit dem Arduino können einfache Prototypen schnell umgesetzt werden, sodass man sich nicht mit allzu komplizierter Technik aufhalten muss. Man kommt auch nicht mehr in die Situation, dass Informatiker einem gern hel-

fen würden, aber einfach nicht so recht verstehen, was man ihnen eigentlich sagen will.

Auch wenn Sie schon Erfahrung mit einer Programmiersprache haben, sollten Sie nun das Buch nicht beiseite legen: Es ist so aufgebaut, dass die Programmierung beiläufig erklärt wird, während die einzelnen Projekte zusammengesetzt werden. Selbst als Geek, dessen natürliche Umgebung meist aus einem Codedschungel besteht, wird Ihnen die Lektüre also kaum langweilig werden. Besonders wenn Sie zwar im Schlaf Software schreiben können, dafür aber noch unerfahren in Sachen Lötkolben, Sensoren und Widerstände sind, wird Ihnen dieses Buch bieten, was Sie suchen. All das erlaubt das schnelle Erstellen von Prototypen, sodass nicht die langwierige, umständliche Umsetzung, sondern die Idee im Vordergrund steht.

All das heißt natürlich auch, dass dieses Buch sich vor allem an Einsteiger richtet. Trotzdem geht es z. B. in Kapitel 11, das sich mit der Anbindung von Arduino an die Welt der Musik beschäftigt, auch ein wenig mehr in die Tiefe. Weiterhin bietet der Anhang eine vollständige Referenz der Programmiersprache, sodass Sie diese Seiten auch im weiteren Verlauf Ihrer Physical-Computing-Karriere auf dem Schreibtisch behalten sollten, um jederzeit nachschlagen zu können.

Der Projektcharakter ist bewusst nach den Prinzipien des Rapid Prototyping gestaltet, um die Fantasie der Leser nicht unnötigerweise einzuengen. Wir freuen uns natürlich über Projektbeschreibungen und Bilder der daraus entstandenen Arbeiten, die Sie uns gern über Twitter (@arduinobuch) zuschicken können.

Hilfreich ist es, wenn Sie zum Buch auch ein Arduino-Board erwerben, falls Sie noch keines besitzen. In Kapitel 1 finden Sie Hinweise dazu, wo Sie den hier verwendeten Arduino Duemilanove beziehen können. Weiterhin wird für die im Buch beschriebenen Workshops eine Reihe von Bauteilen benötigt. Wir empfehlen, dass Sie die Liste der benötigten Teile zu Anfang durchlesen und entsprechend einkaufen. Zudem wäre es hilfreich, wenn Sie einen Lötkolben, ausreichend Lötzinn und Entlötlitze sowie ein Steckbrett besitzen.

Aufbau dieses Buchs

Die Kapitel dieses Buchs lassen sich einteilen in einzelne Workshops und Erläuterungen der physikalischen Aspekte der Entwicklung von Arduino-Projekten. Nach der Einleitung und Einführung

in Arduino sowie der Installation der Programmierumgebung werden zunächst die physikalischen Grundlagen erklärt, und wir versuchen, Ihnen die Angst vor dem Basteln zu nehmen. Dazu gehören zum Beispiel richtiges Löten und die Verwendung von Steckbrettern sowie etwas Hilfe bei der Fehlersuche. Ab Kapitel 3 werden dann einzelne Projekte beschrieben, wobei Kapitel 7 als Einschub die gängigsten Sensoren und Aktoren beschreibt. Zu jedem Themenkomplex beschreibt das Buch zudem weitere Projekte, zu denen man im Internet Anleitungen finden kann, sowie Ideen dazu, was man mit dem erlernten Wissen noch alles anstellen kann.

Komplette Neueinsteiger in die Programmierung werden schon in Kapitel 3 in die Grundbegriffe der Programmierung eingeführt und sollten dieses Kapitel gründlich durcharbeiten. Ab Kapitel 5 werden auch weiterführende Aspekte der Programmierung eingesetzt, die ebenfalls recht ausführlich erklärt ist. Sollten Sie zunächst an einer Stelle kapitulieren, empfehlen wir dennoch, weiterzulesen. Sie werden mit Sicherheit später noch einmal darauf zurückkommen können.

In Kapitel 1 wird zunächst die Geschichte des Arduino-Projekts erläutert. Anschließend wird der Frage nachgegangen, was der Benutzer eigentlich vor sich hat, wenn er ein Arduino-Board gekauft hat. Das Kapitel endet mit der Installation und Erläuterung der Arduino-Software und dem Anschluss des Boards an einen PC.

Kapitel 2 unternimmt einen Ausflug in die Welt des Physical Computing. Zunächst werden der Begriff geklärt und eine Reihe verschiedener Projekte vorgestellt. Im Folgenden erläutern wir die physikalischen Grundlagen wie etwa den Zusammenhang zwischen Strom, Spannung und Widerstand. Weitere Grundlagen umfassen etwa den Aufbau von Schaltkreisen und wichtigen Bauteilen wie Schalter oder Netzteil. Den Abschluss machen ausführliche Anleitungen zum Basteln selbst: Lötkolben und Lötzinn werden ebenso beschrieben wie die Fehlersuche mit Multimeter oder Oszilloskop.

Kapitel 3 beinhaltet den ersten Workshop des Buches. Ziel des Workshops ist es, eine Lampe mit LEDs zu bauen und sie mit dem Arduino zu betreiben. Dazu gehören die ersten Schritte in der Programmiersprache sowie der Aufbau von einfachen Schaltungen mit LED, Widerstand und später Schaltern oder Tastern. Am Ende steht der Code für eine programmierbare RGB-Lampe, die fast alle Farben des sichtbaren Spektrums darstellen kann, sowie ein Ausblick auf weitere Möglichkeiten mit Arduino und Licht.

Kapitel 4 führt den Workshop aus Kapitel 3 fort und erklärt, wie an den Arduino mit dem sogenannten Multiplexing auch Matrix-Displays mit 7x5-LEDs angeschlossen werden können. Zudem wird mit dem Interrupt eine etwas weiter fortgeschrittene Technologie erklärt. Der Workshop schließt mit einer Gehirnwellenmaschine, die durch pulsierende LEDs für Entspannung im Kopf sorgen soll.

Kapitel 5 erlöst den Arduino aus seiner Einsamkeit und zeigt verschiedene Möglichkeiten der Kommunikation mit dem Rechner und darüber hinaus. Zunächst wird auf die einfache Verbindung über den USB-Port eingegangen und auf die Kommunikation mit der seriellen Konsole der Arduino-Programmierumgebung, die man auch für die Fehlersuche nutzen kann. Anschließend wird das Programm Gobetwino erklärt, das ebenfalls unkompliziert die Kommunikation zwischen Arduino und PC ermöglicht. Das Kapitel schließt mit der Anbindung an Processing, eine Programmierumgebung, die dem Arduino als Vorbild gedient hat. Sie ermöglicht es, einfach und schnell grafische Anwendungen zu entwickeln, die mit dem Arduino gesteuert werden.

Kapitel 6 führt das fünfte Kapitel fort und erläutert die Verwendung eines Ethernet-Shields. Dieser Aufsatz für das Arduino-Board erlaubt den Anschluss eines Netzwerkkabels, sodass der Arduino auch mit dem Internet verbunden werden kann. Zunächst wird ein einfacher Webserver implementiert, mit dem man den Status des Arduino im Browser anzeigen lassen kann. Dann folgt ein kleines Projekt, das über Twitter meldet, ob das Licht im Raum an oder aus ist – Physical Computing in Reinform! Zum Abschluss erklärt das Kapitel den Datenspeicher, auf den der Arduino zurückgreifen kann und beschreibt, wie sich dort einzelne Daten ablegen und wieder auslesen lassen.

Kapitel 7 bildet einen Einschub zwischen die verschiedenen Workshops des Buches. Nun, da einige Grundlagen erläutert sind, werden die gängigsten Sensoren und Aktoren beschrieben, die man mit dem Arduino verwenden kann, und einige Projekte und Einsatzmöglichkeiten geschildert. Zudem liefert das Kapitel ein Beispiel dafür, wie man einen kapazitiven Näherungssensor recht einfach selbst bauen kann. Das Kapitel schließt mit einem kleinen Würfel-Projekt, das auf einen kleinen Schalter aufbaut, der zufällig öffnet und schließt.

Kapitel 8 vernetzt den Arduino noch weiter mit seiner Umgebung. In diesem Kapitel wird eine Lichtsteuerung per DMX-Bus erklärt. Anschließend wird der Arduino verwendet, um herkömmliche

Funksteckdosen aus dem Baumarkt anzusteuern. So kann man das gesamte Haus ohne Probleme automatisieren; ob Lichter eingeschaltet oder der Fernseher aus, bleibt ganz dem Bastler überlassen!

In Kapitel 9 wird ein kleiner Ausflug in die Welt der Kleidung unternommen. Das Prinzip des »Wearable Computing« wird erklärt und anhand eines kleinen Beispiels eingeführt. Mit dem erläuterten Projekt sollte es problemlos möglich sein, eine beliebiges Kleidungsstück zu einer iPod-Steuerung umzufunktionieren. Nie mehr lästiges Herumfummeln, nur um die Musik an- oder auszumachen!

Kapitel 10 handelt von der Steuerung von Musik mit dem Arduino. Dazu gehören verschiedene Musikprogramme auf dem PC, die etwa über Midi angesteuert werden. Zudem geht das Kapitel auf Möglichkeiten ein, mit Aktoren direkt auf Musikinstrumente einzuwirken und sie so zu bespielen.

In Kapitel 11 wird dann der Arduino selbst zur Erzeugung von Klängen verwendet. Zunächst geht das Kapitel dabei auf die elektronische Klangerzeugung im Allgemeinen ein, bevor anhand einiger Beispiele erläutert wird, wie man dem Arduino diese Klänge entlocken kann. Den Abschluss des Kapitels macht ein Workshop, in dem ein Theremin, ein elektronisches Musikinstrument, selbst gebaut wird, dessen Töne sich ohne Berührung mit den Händen steuern lassen.

Der Anhang ist in diesem Buch vergleichsweise lang. Er enthält neben einer vollständigen Referenz zur Programmiersprache auch eine Übersicht über die wichtigsten Arduino-Bibliotheken, Boards und Shields. Damit dient das Buch auch als Nachschlagewerk beim Entwickeln eigener Projekte.

Typografische Konventionen

Die folgenden typografischen Konventionen werden in diesem Buch verwendet:

Kursiv
: Wird für URLs, die Namen von Verzeichnissen und Dateien, Optionen, Menüs und zur Hervorhebung verwendet.

`Nichtproportionalschrift`
: Wird für Codebeispiele, den Inhalt von Dateien und sowie für die Namen von Variablen, Befehlen und anderen Codeabschnitten verwendet.

Verwendung der Codebeispiele

Dieses Buch soll Ihnen bei der Umsetzung Ihrer Projekte helfen. Es ist allgemein erlaubt, den Code aus diesem Buch in Ihren Programmen und Dokumentationen weiterzuverwenden. Dafür ist es nicht notwendig, uns um Erlaubnis zu fragen, es sei denn, es handelt sich um eine größere Menge Code. So ist es beim Schreiben eines Programms, das einige Codeschnipsel dieses Buchs verwendet, nicht nötig, sich mit uns in Verbindung zu setzen, beim Verkauf oder Vertrieb einer CD-ROM mit Beispielen aus O'Reilly-Büchern dagegen schon. Das Beantworten einer Frage durch das Zitat von Beispielcode erfordert keine Erlaubnis. Verwenden Sie einen erheblichen Teil des Beispielcodes aus diesem Buch in Ihrer Dokumentation, ist unsere Erlaubnis dagegen nötig.

Eine Quellenangabe ist zwar erwünscht, aber nicht obligatorisch. Dazu gehört in der Regel die Erwähnung von Titel, Autor, Verlag und ISBN, zum Beispiel: »Arduino – Physical Computing für Bastler, Designer und Geeks« von Manuel Odendahl, Julian Finn & Alex Wenger. Copyright 2010 O'Reilly Verlag, ISBN 978-3-89721-995-3.«

Falls Sie sich nicht sicher sind, ob die Nutzung der Codebeispiele außerhalb der hier erteilten Erlaubnis liegt, nehmen Sie bitte unter der Adresse kommentar@oreilly.de Kontakt mit uns auf.

Die Codebeispiele zu diesem Buch

Zu diesem Buch gibt es einen Twitter-Account namens

@arduinobuch unter *http://twitter.com/arduinobuch*.

Dort werden interessante Links zu Arduino-Projekten genauso angegeben wie Errata oder Hinweise zum Buch.

Die Codebeispiele zum Buch und weitere Informationen finden Sie auch auf der Website des O'Reilly Verlags unter *http://www.oreilly.de/catalog/micprogger*.

Die Arduino-Welt

Arduino wird zwar von einer italienischen Firma vertrieben, die Layoutdaten für das Board sind aber ebenso wie die Programmierumgebung unter einer Open-Source-Lizenz verfügbar. Die Weiterentwicklung wird also auch von einer Gemeinschaft unterstützt,

die hilft, beides konstant zu verbessern. Zudem gibt es eine riesige und stets wachsende Menge an Websites, die Arduino-Projekte dokumentieren oder Anleitungen zum Basteln geben. Hier im Buch sind viele dieser Projekte erwähnt, zusätzlich wollen wir einen kleinen Überblick darüber geben, welche Ressourcen und Websites im Zusammenhang mit dem Arduino interessant sind.

Das Arduino-Projekt

Die Marke »Arduino« sowie die Rechte an den Schaltplänen liegen bei der Firma tinker.it, die aus den ersten Arbeiten am Arduino hervorgegangen ist. Die Firma behält sich allerdings nur das Markenrecht vor, alle anderen Daten können also frei verwendet werden, sofern die Ergebnisse nicht unter dem Namen Arduino veröffentlicht werden. Neben dem Arduino hat sich tinker.it vor allem auf Consulting und Training rund ums Physical Computing und den Arduino spezialisiert.

Das Make Magazine

Das Make Magazine erscheint vierteljährlich beim O'Reilly Verlag in den USA und ist auf Projekte zum Selbstbasteln spezialisiert. Jede Ausgabe kommt mit einer ganzen Reihe an Anleitungen, die Schritt für Schritt nachgebaut werden können. Unter *http://www.makezine.com/* gibt es zudem eine umfangreiche Website, die unter anderem auch ein Archiv für Arduino-Projekte beherbergt. Für weitere Inspiration und Anleitungen sei Ihnen *http://blog.makezine.com/archive/arduino/* empfohlen. Dort finden Sie eine lange Liste von Ideen, mit denen man sich stundenlang beschäftigen kann.

Weiterführende Quellen

Neben den schon genannten Webseiten gibt es eine ganze Reihe von Ressourcen rund um das Thema Arduino. Für den Erwerb von Boards sei zusätzlich der entsprechende Absatz in Anhang A empfohlen. Hier wollen wir einige interessante Websites vorstellen. Diese Liste ist natürlich nicht einmal annähernd vollständig.

- *Freeduino.org* ist eine Community-Seite, die Projekte und Möglichkeiten rund um Arduino und seine Abwandlung Freeduino sammelt. Die Seite verfügt über einen Index von Anleitungen rund um Sensoren und Aktoren, die mit dem Arduino verbunden werden können, und ist deshalb besonders nütz-

lich, wenn man auf der Suche nach Informationen über ein bestimmtes Bauteil ist.

- *Instructables.com* ist ein Portal im Netz, das Anleitungen für allerlei Projekte zur Verfügung stellt. Auch zum Thema Arduino und Physical Computing wird man hier fündig, ebenso gibt es einige Bastelanleitungen für selbstgebaute Sensoren oder Aktoren.
- *Ladyada (http://www.ladyada.net)* ist eine Website einer ehemaligen MIT-Studentin, die eine umfangreiche Ressourcensammlung zum Thema Arduino zusammengetragen hat. Hier finden sich viele Ideen, aber auch Angebote für verschiedene Arduino-Shields.
- Die Webseite *http://mediamatic.net* beschäftigt sich mit dem künstlerischen und designerischen Aspekt von Physical Computing. Insbesondere ist die Website empfehlenswert, weil sie eine umfangreiche Ressourcensammlung zum Thema Wearable Computing beherbergt, also Computer, die in Kleidung eingearbeitet sind. Diese Unterseite ist unter http://www.mediamatic.net/page/12648/nl zu finden.
- Unter *http://www.talk2myshirt.com* findet sich eine herausragende Ressource rund um das Thema Wearable Computing. Ein Blog beschreibt immer wieder neue und interessante Projekte, während eine Community zu fast allen Problemen eine Lösung findet.

Danksagungen

Wir möchten hiermit noch einer Reihe von Menschen danken, ohne die dieses Buch nicht möglich gewesen wäre. Zunächst möchten wir unserem herausragenden Lektor Volker Bombien danken, mit dem dieses Projekt weit mehr als eine normale Geschäftsbeziehung war. Weiterer Dank gilt Jens Ohlig für das großartige Einleitungskapitel sowie allen unseren Freunden im Entropia e.V., besonders Cupe, Flowhase, Bugbabe, Neingeist, Syb, nanoc und Hannes.

Zudem möchten wir in dieser zweiten Auflage besonders all jenen danken, die uns wertvolles Feedback geschickt haben.

Julian Finn dankt desweiteren besonders Steffi sowie Silvan Horbert, Roman Alexis Anastasini, Jayoung Bang und Martin Feldkamp für Inspiration und Instruktion.

Manuel Odendahl möchte seinen Dank aussprechen an Julia Tziridis, Hans Hübner, Philip Baljeu, Martin Hirsch und Fabienne Serriere.

Besonders bedanken möchte sich Alex Wenger bei seiner Frau und seinen Kindern, die immer Verständnis für die viele Arbeit hatten.

Sicherlich ein ganzes Stück schlechter wäre dieses Buch ohne die KorrekturleserInnen Prisca Fey, Benedikt Achatz, Frank Bierlein, Greta Louise Hoffmann, Lukas Fütterer und David Loscher, die uns mit hilfreicher Kritik aus Sicht unserer »Zielgruppe« unterstützt haben.

Natürlich wäre das Buch auch nicht möglich ohne die fantastische Arduino-Community und insbesondere die Hauptentwickler Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, David Mellis und Nicholas Zambetti.

Ein ganz spezieller Dank geht an unseren Fotografen Jonas Zilius, dem wir das Fotostudio mehrere Tage belagert haben, und an Yunjun Lee, der für uns all diese fantastischen Zeichnungen erstellt hat.

Dedication from David Cuartielles

Thinking, burning, redesigning

There is little things in life where it is actually allowed to fail. First time I got to understand this, was at the end of my graduate studies back in the 90's. A professor explained to me it didn't matter my thesis could render somehow unsuccessful because I would at least demonstrate the path of my choice was not the right one. It wasn't a relief, but he claimed it should at least help others figuring out the right way. In my defense I should say the project turned out right, but it could also have failed and I would have gotten my degree.

Trial and error, in my opinion one of the most important aspects of the scientific method, doesn't apply to its full extent when one introduces the cost factor. One might consider not risking his precious smartphone to science when studying gravity by throwing it from a certain height and measure the fall using the internal accelerometer. There could be a lot of beneficial outcomes from such an experiment, but are you crazy enough to try?

The act of acquiring the right tool to do something is almost a religious one. Imagine you are about to learn astronomy, you will be

interested in getting a telescope to look at the stars. You might not go for the »gold edition« one in the first place. People usually get something that works, and if they like it and get hooked, then they make a move for something better - and probably more expensive.

When it comes to learn about crafting electronic devices like you want to feel the thrill of controlling a light bulb, starting up the engine of your car remotely, or time up your fireworks next New Year's Eve by means of digital logics ... Arduino offers a door to easily access this world. Get yourself a board, download the software, check out the examples and in a couple of sessions you will be controlling motors, launching rockets, and reading data from a GPS.

We made Arduino a tool for learning. It is made to be easy to use for beginners, but also powerful for those that want to go deeper in the topic. However, we wanted it to be affordable. We like people to try things without the fear of breaking them. Your Arduino will most likely not let you down, but if it did ... it wouldn't be a big deal. It is cheap to get a new one, but even cheaper to fix it. Since it is open source you can always consider the possibility of creating your own version of it, our website will tell you how.

I hope this book will encourage you to try things out. Its authors have put a lot of effort in transmitting one of the Arduino Philosophy Fundamentals: make it work while having fun!

May 2010
David Cuartielles ist Mitentwickler des Arduino-Boards

KAPITEL 1

Von Königen und Kondensatoren

In diesem Kapitel:

- Die Geschichte des Arduino-Projekts
- Der Arduino, das unbekannte Gerät
- Arduino-Projekte: eine kleine Vorstellung
- Hardware
- Die Arduino-Entwicklungsumgebung

Die ersten Investitionen sind getätigt. Das Buch hier und ein Arduino-Board sind gekauft, und nun liegt dieses wundersame Gebilde aus Schaltkreisen, Chips und Pins auf dem Tisch. Wo kommt es her, wo wird es hingehen? Die folgenden Kapitel sollen darin Einblick geben. Von der Geschichte des Arduino-Projektes geht es zur Beschreibung des in diesem Buch verwendeten Boards, des Arduino Duemilanove. Wenn Sie noch keines besitzen, können Sie in diesem Kapitel auch Bezugsquellen nachschlagen. Anschließend werden die Grundzüge des Physical Computing und der dafür notwendigen Physik erklärt, um in den darauf folgenden Kapiteln schließlich mit kleinen Workshops zu beginnen, die die Programmiersprache und einfache Schaltungen erklären. Wer diese Workshops durcharbeitet, sollte anschließend in der Lage sein, größere Projekte mit Arduino durchzuführen. Ob dabei bunte Lampen gebastelt werden, eine Pflanze über Twitter meldet, wenn sie Wasser benötigt, oder der Arduino dazu verwendet wird, Musik zu machen – dieses Buch sollte nicht nur Einsteigern genügend Möglichkeiten bieten, die Welt des Physical Computing kennenzulernen.

Die Geschichte des Arduino-Projekts

Im Jahre 1002 ließ sich Arduino, Markgraf im oberitalienischen Ivrea, nach dem Tod von Kaiser Otto III zum König von Italien wählen. Er hatte sich damit eine Marktnische gesucht, die es vorher nicht gegeben hatte: Italien war bis dahin von den römisch-deutschen Königen beherrscht worden. An die Macht gekommen war Arduino durch unrechtmäßige Aneignung von Kirchengut, was ihm nach der Ermordung des Bischofs Petrus von Vercelli 997 die Exkommunikation einbrachte, seinen Weg zu mehr Einfluss aber

nicht aufhielt. Sein Geschäftsmodell erwies sich als letztendlich nicht tragfähig: Nachdem er erheblichen militärischen Widerstand von Adel und Klerus erfahren hatte, verzichtete er auf den Thron und starb in einem Kloster. Italien bekam nach ihm erst in der Neuzeit wieder einen König.

Das Arduino-Projekt hat seinen Namen nicht direkt von dem machthungrigen, aber letztendlich erfolglosen italienischen König geerbt. Ein freundlicherer Namenspatron wäre sicher der Geograph Giovanni Arduino (1714–1795), der Vater der italienischen Geowissenschaft, nach dem auch der Meeresrücken Dorsum Arduino auf dem Mond benannt ist. Noch näherliegend ist aber eine ganz andere Erklärung für den Namen: »Arduino« war der Name einer Studentenkneipe (die wiederum nach König Arduino benannt ist) in der Nähe des Interaction Design Institute Ivrea (IDII) in Italien, einer ehemaligen Hochschule für Gestaltung.

Ganz wie der König aus dem 11. Jahrhundert hat das Elektronikprojekt Arduino aber etwas ganz Neues versucht, was es in dieser Form bisher nicht gab. Allerdings ist dem Projekt bis jetzt größerer Erfolg beschieden, und statt Machtgier und Intrigen ist das Leitprinzip die Offenheit.

Die Gestaltung von Interaktion wurde am IDII von 2001 bis 2005 gelehrt. Unter Interaction Design versteht man diejenige Fachrichtung der Gestaltung, die sich mit Verhaltensweisen von Produkten und Systemen beschäftigt, mit denen ein Benutzer interagieren kann. Das IDII hatte nur genau einen Kurs und einen Magisterabschluss in Interaction Design im Angebot. Obwohl die Schule nach vier Jahren mit einer anderen Akademie zusammenlegt wurde, nachdem die ursprüngliche Förderung ausgelaufen war, und heute nicht mehr als eigenständige Institution existiert, hatte sie einen Einfluss, der noch heute spürbar ist. Neben dem Arduino entstanden hier wichtige Beiträge zu der Grafikprogrammierplattform »Processing« vom MIT Media Lab und ein viel gelesenes Standardwerk zum Interaction Design.

Im Winter 2005 hatte Massimo Banzi, Dozent für Gestaltung am IDII, ein Gespräch mit David Cuartielles, einem Mikrochip-Ingenieur aus Spanien, der einen Forschungsaufenthalt am Institut absolvierte. Die beiden sprachen darüber, wie oft Banzi Klagen von Studenten hörte, dass es keine preiswerten und einfach zu programmierenden Mikrocontroller-Plattformen für Kunstprojekte gebe. Sie entwickelten ihr eigenes Design für ein Board. Ausgehend von der Entwicklungsumgebung Processing schrieb David Mellis

eine Programmiersprache für das Projekt, und irgendwann muss die Idee entstanden sein, das Ganze nach der örtlichen Kneipe zu benennen.

Mit 3.000 Euro Startkapital wurden eine erste Serie von Arduinos bei einem kleinen Hersteller produziert und das komplette Design unter einer Creative-Commons-Lizenz im Internet veröffentlicht.

Creative Commons ist ein Satz von vorgefertigten Lizenzverträgen für die Veröffentlichung und Verbreitung digitaler Medieninhalte. Der einfachste CC-Lizenzvertrag verlangt vom Nutzer lediglich die Namensnennung des Rechteinhabers. Unter diesen Bedingungen ist auch das Hardwaredesign für Arduino verfügbar, solange es unter den gleichen Bedingungen weitergegeben wird. Lediglich den Namen »Arduino« möchten die Entwickler als Marke behalten und bitten darum, eigene Entwicklungen unter einem anderen Namen zu veröffentlichen.

Mittlerweile ist aus dem Projekt eine kleine internationale Firma geworden, unterstützt vom Professor Tom Igoe vom Interactive Telecommunications Program der New York University. Die Tatsache, dass jeder sich das Design von Arduino schnappen kann und in einer Fabrik in Fernost Kopien in Massenproduktion anfertigen kann, hat nicht geschadet. Die Offenheit ist nicht nur ein Grund für die Beliebtheit der Plattform, sondern hat auch dazu beigetragen, dass sich Schaltungen und Code für Arduino an vielen Stellen im Internet finden lassen. Eine Firma, die sich nicht der offenen Gemeinschaft von Enthusiasten bedienen könnte, weil sie die Verwertungsrechte mit Zähnen und Klauen verteidigt, hätte diesen unschlagbaren Vorteil nicht.

Arduino als Gemeinschaft ist vergleichbar mit dem Phänomen der Emergenz in der Natur. Dieser Begriff bezeichnet das spontane Herausbilden von Strukturen in Systemen durch das Zusammenspiel ihrer Elemente, ohne dass das Endergebnis auf die einzelnen Elemente direkt zurückführbar ist. Ob beim Ameisenbau oder bei der freien Mikrocontroller-Plattform: Das Ganze ist größer als die Summe seiner Teile.

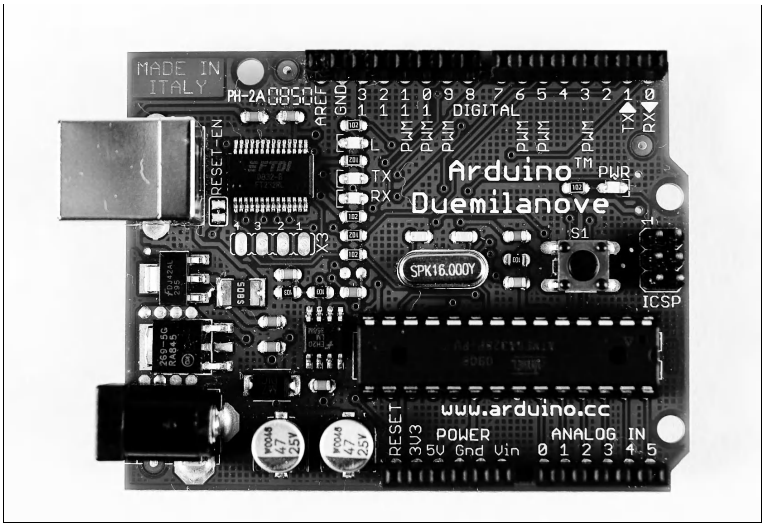
Der Arduino, das unbekannte Gerät

Arduino besteht aus einer Entwicklungsumgebung und einem Entwicklungsboard. Die Entwicklungsumgebung läuft auf einem normalen Computer und bildet den Softwareteil. In ihr werden Programme geschrieben, die dann auf dem Board, der Hardware,

ausgeführt werden. Das Board funktioniert dabei als Schnittstelle von der Welt der Bytes zur der Welt der Dinge: Es kann Eingaben, etwa von Sensoren, verarbeiten und elektrische Signale ausgeben. Zum Beispiel an *Aktoren*, also Bauteile, die eine Auswirkung auf die physikalische Welt haben. Das können Motoren sein, Relais oder aber Ausgabeschnittstellen wie Lichter, Lautsprecher oder Displays. Oder man steuert damit wiederum Sensoren, also Bauteile, die von der realen Welt beeinflusst werden und diese Veränderungen elektronisch weiterleiten, wie etwa einfache Schalter, Temperatur- oder Lichtsensoren. Seine Anweisungen erhält das Board von Programmen, die in der Arduino-Software geschrieben und anschließend in den Speicher geladen werden.

Dabei kann der Entwickler frei entscheiden, ob er das Board vom Computer abkoppeln und als eigenständiges Gerät einsetzen oder als Schnittstelle zwischen Computer und Außenwelt verwenden möchte. Indem es ständig mit dem Computer kommuniziert, kann es benutzt werden, um Programme wie PureData, Processing, Flash, VVVV oder Max/MSP zu steuern. Das Arduino-Board kann hier durchaus mehr Arbeit leisten, als nur Daten zu übermitteln. Es kann sie vorher verarbeiten, oder es kann für Benutzerschnittstellen mit Tastern, Drehreglern, Lichtern und vielen anderen Möglichkeiten eingesetzt werden.

Abbildung 1-1 ▶
Das Arduino-Board



Beim Arduino-Projekt wird viel Wert auf eine Entwicklungsmethode gesetzt, die iterativ und interaktiv ist: Man geht davon aus,

dass Elektronik und Programmieren am besten durch Basteln, Ausprobieren und Modifizieren erlernt werden können. Eine Schaltung versteht man am besten, indem man sie zunächst aufbaut und zum Laufen bringt. Anschließend kann man sie erweitern und modifizieren. Auf die gleiche Weise lernt man auch den Umgang mit einer Programmiersprache besonders gut. Des Weiteren gibt es im Internet eine riesige Menge an verschiedenen Beispielen und Schaltungen mit Arduino, die jeder bei sich zu Hause ausprobieren und weiter modifizieren kann. Dadurch können viele Schaltungen sehr einfach selbst gebastelt, analysiert und auch verstanden werden.

Die ursprüngliche Zielgruppe von Arduino waren Designer und Künstler. Deswegen sind viele Funktionen sehr einfach gestaltet, damit Benutzer, die noch nie zuvor programmiert und elektronische Schaltungen gebaut haben, sich schnell mit der Umgebung vertraut machen können. Schon nach einigen Stunden können eigene Prototypen und Projekte zum Laufen gebracht werden. Fortgeschrittenen Nutzern mag vieles ein bisschen umständlich oder ineffizient vorkommen, es steht ihnen aber auch nichts dabei im Wege, fortgeschrittene Programme zu schreiben, die vollen Zugriff auf die tieferen Funktionen haben. Das Ziel von Arduino ist, Komplexität und Aufwand von normalen elektronischen Projekten zu verringern, damit auch Anfänger sich schnell zurechtfinden und nicht abgeschreckt werden.

Die Arduino-Software ist frei unter *http://www.arduino.cc* erhältlich und wird als Open Source zur Verfügung gestellt. Das heißt, sie darf verändert und wieder zum Download angeboten werden. Auch die Schaltpläne für das Board sind frei, sodass es eine Vielzahl von Abwandlungen zu kaufen gibt. Das derzeit (April 2010) aktuellste Board ist der »Arduino Duemilanove«, der auch in diesem Buch verwendet wird. Die Sprache basiert auf den Grundideen von Processing, wobei sie keine Variante von Java ist, sondern vielmehr auf C++ basiert. Man braucht diese Sprachen, wenn man denn schon von ihnen gehört hat, nicht zu fürchten, denn ihre Komplexität wird bei Arduino durch viele hilfreiche Konstrukte vor dem Benutzer versteckt (sie ist aber für fortgeschrittene Benutzer immer noch zugänglich).

Natürlich gibt es noch eine Reihe von anderen Hardware-Entwicklungsumgebungen. Arduino bietet aber einige Vorteile:

- Arduino ist günstig. Die Software ist kann frei heruntergeladen werden, das günstigste Arduino-Board schon für ca. 25 Euro

zu haben. Der Hauptchip kostet nur ca. vier Euro, lässt sich also im schlimmsten Fall leicht austauschen.

- Arduino ist frei. Die Open-Source-Philosophie hat beim Arduino-Projekt viele Früchte getragen: Die Software wird ständig von vielen Menschen in der Welt weiterentwickelt und verbessert; dazu kommt eine Vielzahl an Bastelprojekten, die zum Nachbauen und Erweitern im Netz veröffentlicht werden. Auch die Schaltpläne für die Hardware stehen unter einer Open-Source-Lizenz. Das Ergebnis ist eine Vielzahl von eigenen Arduino-Versionen, die jeweils speziell an die Bedürfnisse eines Projektes angepasst wurden. Und zu guter Letzt ermöglicht es diese Freiheit, dass im Internet eine Vielzahl von Erweiterungen verkauft werden kann; sogenannte Shields, die auf das Board aufgesteckt werden. Mehr Informationen über Shields finden sich im entsprechenden Anhang.
- Arduino funktioniert überall. Die Arduino-Software ist für Windows, Mac OS X und Linux verfügbar und basiert auf Processing, das schon seit Langem von vielen Künstlern und Programmierern eingesetzt wird. Das Board kann mit einem USB-Kabel angeschlossen werden. Damit werden viele Probleme vermieden, die in anderen Projekten auftreten. Andere Schwierigkeiten lassen sich leicht lösen, weil Hard- und Software entsprechend weit verbreitet sind.
- Arduino hat viele Fans und deren Gemeinde wächst stetig. So ist es sehr einfach, sich mit anderen Nutzern auszutauschen oder Hilfe zu bekommen, wenn es Probleme gibt. Auch die Anzahl gut dokumentierter Beispielprojekte wächst ständig, sodass gute Ideen leicht nachzubauen und an die eigenen Bedürfnisse anzupassen sind.
- Arduino ist einfach. Die Programmiersprache basiert auf Processing, das für ein einfaches Lernen entwickelt wurde. Es ist besonders freundlich für Anfänger und macht es besonders einfach, »schnell mal was auszuprobieren«. Das hilft allerdings nicht nur Einsteigern, sondern auch hartgesottenen Profis: Es genügt, ein paar Zeilen zu schreiben und auf den »Upload and Run«-Knopf (siehe Kapitel 1) zu drücken, um ein neues Programm auf die Hardware zu laden und auszuprobieren. In vielen anderen Projekten muss man zuerst eine komplexe Softwareumgebung einrichten, bevor die erste Zeile Code geschrieben werden kann. Die Herangehensweise bei Arduino gibt viel mehr Raum für kreative Entfaltung. Viele

Ideen können einfach ausprobiert werden, ohne dass einem starren Vorgang gefolgt werden müsste, der eher Ingenieuren entgegenkommt als Künstlern oder Designern.

Die Arduino-Philosophie

Im Gegensatz zu einer traditionellen Entwicklung von elektronischen Projekten, bei denen der Prozess im Vordergrund steht, wie sie z.B. in Ingenieurschulen gelehrt wird, wird bei Arduino-Projekten Wert auf das Implementieren und Bauen gesetzt, weniger auf das lange und anstrengende Planen. Alles ist dazu gedacht, möglichst schnell und elegant einen funktionierenden Prototypen zum Laufen zu bringen. Diese Herangehensweise mag zuerst ein bisschen verwirrend klingen: Ist sorgfältige Planung, gerade bei Elektronik, nicht notwendig, um Fehler und Nebenwirkungen zu vermeiden? Bei Arduino ist der Weg ein großer Teil des Ziels: Erst beim Ausprobieren (und oft auch, wenn unvorhergesehene Fehler oder nicht geplantes Verhalten zutage treten) setzt man sich mit der Materie richtig auseinander. Und wenn ein schneller Erfolg eintritt, motiviert das zusätzlich die Entwicklung: Die Energie bleibt erhalten, auch wenn man sich durch größere Projekte hindurcharbeitet. Das Basteln und Ausprobieren ist bei Arduino ein Spiel, das Spaß machen soll. Anstatt zielstrebig von A nach B zu gehen, wie es sonst bei Projekten der Fall ist, kommt es hier oft vor, dass man sich auf dem Weg ein bisschen verirrt und dann auf einmal bei C landet.

Auch für fortgeschrittene Nutzer ist diese Eigenschaft von Arduino sehr erfrischend: Man muss nicht mühsam, sauber und korrekt Projekte bauen und Programme schreiben, sondern überrascht sich schnell dabei, wie man wilde Ideen »einfach nur so« ausprobiert, weil es eben sehr schnell möglich ist. Am besten lässt sich Arduino mit älteren Homecomputern vergleichen, die einen beim Anschalten gleich mit einem BASIC-Prompt einluden, verrückte Programme zu schreiben.

Im Sinne des schnellen Erzeugens von funktionierenden Prototypen wird bei Arduino viel Wert auf die kreative Nutzung von Technik gesetzt. Anstatt alles von Grund auf aufzubauen, werden oft fertige Komponenten (sogenannte Shields, siehe auch Anhang A) zusammengesteckt und mit fertigen Programmteilen angesprochen (sogenannten Libraries, siehe auch Anhang B). Ähnlich wie bei den schon kurz erwähnten grafischen Programmierumgebungen Max/MSP, Reaktor oder VVVV werden bei

Arduino oft einzelne Blöcke zusammengesteckt und mithilfe eines angepassten Programms kombiniert. Dadurch lassen sich auch viele aufwendige Schaltungen, die von anderen Entwicklern schon entworfen wurden, leicht wiederverwenden. Oft werden bei Arduino-Projekten auch herkömmliche Haushaltsgeräte oder elektronische Spielzeuge modifiziert und »missbraucht«, um bestimmte Schaltungen zu bauen.

Arduino-Projekte: eine kleine Vorstellung

Die Bandbreite an Arduino-Projekten reicht von Lichtinstallationen über Gebrauchsgegenstände bis hin zur Robotik. Hier sollen einige Projekte vorgestellt werden, die besonders interessant sind und auch als Motivation gedacht sind, sich durch das Buch zu arbeiten, die Programmiersprache zu erlernen und mit elektronischen Bauteilen zu experimentieren.

LilyPad und Wearable Computing

Das Arduino LilyPad, eine Abwandlung des in diesem Buch benutzten Arduino-Boards, lässt sich auch in Kleidung einnähen. Diese kann so mit LEDs ausgestattet werden, um etwa auf ihr Umfeld zu reagieren oder mit ihm zu kommunizieren. Am wichtigsten sind dabei leitende Stoffe oder solche, die ihre Form verändern können sowie elektronische Bauteile, die in Kleidung eingearbeitet sind. Das kann von Displays auf dem Rücken bis hin zu intelligenter Bekleidung reichen, die ihre Umwelt wahrnehmen oder mit anderen kommunizieren kann. In Kapitel 9 werden diese leitenden Fäden näher beschrieben und ein Projekt erläutert, das diesem sogenannten Weareable Computing zuzuordnen ist.

Interaktive Kunstinstallationen

Neben Designern und Modeschöpfern sind vor allem Künstler die großen Nutznießer von Arduino. Da Künstler sich meistens nicht mehr als nötig mit Technik und Programmierung beschäftigen, sondern sich lieber auf ihre kreative Arbeit konzentrieren wollen, ist ein einfach zu programmierender Mikrocontroller geradezu ein Segen und eröffnet ganz neue Möglichkeiten, den eigenen Ideen freien Lauf zu lassen.

Wundersame VGA-Bilder

Der koreanische Pionier der Videokunst Nam June Paik dürfte mit Sicherheit einen Einfluss auf das Projekt von Sebastian Tomczak gehabt haben, der ein VGA-Signal nutzt, um mit Signalstörungen Bilder auf einem Computermonitor hervorzurufen. Die dafür notwendigen Signale werden von einem Arduino gespeist, an dem einige Pins eines VGA-Kabels angeschlossen sind. Die anderen Kabel hängen an einem Laptop, der für die horizontalen und vertikalen Frequenzsignale sorgt. Damit können zum Beispiel visuelle Effekte passend zu laufender Musik erzeugt werden, und indem die Farbkanäle bei einem anderen Monitor umgedreht werden, sind sogar verschiedene Bilder auf einmal möglich. Weitere Informationen zu diesem Projekt finden Sie unter *http://littlescale.blogspot.com/2009/04/how-to-use-arduino-to-generate-glitchy.html*, wo nicht nur das Grundsetup erläutert wird, sondern auch einige Videobeispiele gezeigt werden.

Gute Luft

Das AIR-Projekt ist in erster Linie als soziales Experiment gedacht: Jeder Teilnehmer erhält ein Gerät, basierend auf einem Arduino, das Umweltdaten wie den Grad der Luftverschmutzung misst. Diese Daten werden über ein Netzwerk zu einem zentralen Server übertragen und an die anderen Teilnehmer weiterverbreitet. Die Nutzer können also selbstständig herausfinden, wo in ihrer Umgebung besondere Verunreinigungen existieren. Das Projekt will zum Beispiel die Reaktionen darauf herausfinden und sehen, ob die Menschen sich selbstständig darum kümmern, ihre direkte Umwelt lebenswerter zu machen. Zudem soll es Diskussionen rund um Umweltpolitik, Gesundheit und soziale Zusammenhänge innerhalb einer Gemeinschaft anregen. Die Website für das Projekt findet sich unter *http://www.pm-air.net/*.

Buchstabenklettern

Eine spielerische Kunstinstallation hat Olaf Val mit dem Digiripper geschaffen, eine Matrix von 7 mal 5 Elementen aus Plastik, die in eine Wand eingelassen sind. Dahinter befinden sich Leuchtdioden. Wenn diese aufleuchten, fahren die Plastikelemente aus und bieten dem Nutzer Halt, um darauf zu klettern. Wird das Licht ausgeschaltet, fährt das Element wieder ein, der Kletterer verliert seinen

Halt. Nun gilt es, sich eine Reihe von Buchstaben zu merken, die nacheinander angezeigt werden. Wechselt der Buchstabe, muss man schnell umgreifen, um nicht von der Wand herunterzufallen. Gewonnen hat, wer am längsten durchhält. Die Website des Projekts finden Sie unter *http://www.olafval.de/digigripper/*. Es findet regelmäßig seinen Platz in Ausstellungen im deutschsprachigen Raum.

Abbildung 1-2 ▶
Digigripper, © Olaf Val

Klangstufen

Um dem Betrachter ein Gefühl für die Klänge seiner Stadt zu geben, wurde die Installation »Klangstufen« geschaffen. Unter einer kleinen Holztreppe sind Entfernungssensoren angebracht, die erkennen, wenn ein Mensch auf einer Stufe steht. Der Betrachter kann nun einen Punkt in der Stadt auswählen und einen Kopfhörer aufziehen. Je nach Ort und Stufe hört er nun Klänge, die an diesem räumlichen Punkt aufgenommen wurden. Jede Stufe repräsentiert dabei einen Schritt in die Höhe, beginnend bei 0 Metern, auf denen man die U-Bahn, Wasser oder das Rauschen in der Erde hört. Von dort aus geht es weiter über den Klang der Stadt (200 m), des Umlands (1. 500 m), des Wetters (8.000 m) und der Flugzeuge (12.500 m). Das Projekt befindet sich immer noch in der Weiterentwicklung wird unter *http://gestaltung.fh-wuerzburg.de/blogs/reconqr/?p=809* dokumentiert, wo man sich auch ein Video ansehen kann.

Klang-Körper

Die Installation Sonic Body bringt dem Betrachter die Geräusche und Töne des menschlichen Körpers näher. In einem kleinen Raum sind die inneren Organe eines Menschen aus Stoff nachgebildet, sodass der Besucher das Gefühl hat, sich inmitten des Körpers zu befinden. Die einzelnen Organe sind dabei mit Berührungssensoren ausgestattet, die über einen Arduino zusammengeführt und mit MAX/MSP verbunden sind. Werden die Organe berührt, gedrückt oder gestreichelt, sorgt der angeschlossene PC für eine Kulisse von Geräuschen, die aus dem Inneren des Körpers aufgenommen wurden. Ziel des Ganzen ist, Besuchern ein Gefühl für sich selbst und für die wundersame Schönheit der Natur zu geben. Wer mehr erfahren möchte, kann unter *http://www.sonicbody.co.uk* eine volle Dokumentation finden.

Die binäre Brücke

Auch wenn der Arduino nur ein kleines Gerät ist, kann er auch für Kunstinstallationen von großem Ausmaß verwendet werden. Im Hafen von Malmö etwa haben vier Studenten eine Brücke mit zwölf Bewegungssensoren ausgestattet. Diese sind in der Lage, zu messen, wie viele Menschen sich gerade auf der Brücke befinden. Die Anzahl der Fußgänger wird dann in eine Binärzahl umgewandelt, die wiederum die Farbgebung von Lampen steuert, die unter der Brücke angebracht sind. Ist die Binary Bridge (*http://binarybridge.k3.mah.se/*) leer, ist sie auch nicht erleuchtet, ist sie voller Menschen, erstrahlt sie in vollem Glanz. Dazwischen leuchtet sie in allen möglichen Farben von Grün bis Purpur, je nachdem, welche Werte die Fußgänger gerade erzeugen.

Kreative Aggressionen

Boxsäcke eignen sich seit jeher recht gut, um angestauten Aggressionen freien Lauf zu lassen, ohne dass dabei Menschen zu Schaden kommen. Die Designer von Fluidforms (*http://fluidforms.eu/de/CassiusWhat.php*) nutzen die Fäuste ihrer Kunden, um individuelle Lampen zu erschaffen. Die Energie, die bei jedem Schlag in einen speziell präparierten Boxsack freigesetzt wird, wird mit Sensoren und einem Mikrocontroller an einen PC übermittelt. Dort läuft ein 3D-Programm, das einen weißen Zylinder anzeigt, der Schlag um Schlag verformt wird. Das Ergebnis kann anschließend gespeichert und mit einen 3D-Drucker gedruckt werden, welcher als Lampenschirm verwendet wird.

Roboter

Auch wenn Roboter noch weit davon entfernt sind, für uns zu denken oder die Weltherrschaft zu übernehmen, findet man sie immer mehr auch außerhalb von Produktionsstraßen in riesigen Fabrikhallen. Neben einer großen Anzahl an Spielzeugrobotern bevölkern heute auch automatische Staubsauger und Wischmops unsere Wohnzimmer, und die Anzahl der Projekte, die eigene Roboter bauen oder im Handel erhältliche Geräte für sich nutzbar machen, wird ständig größer.

Musikinstrumente

Einen ganz eigenen Bereich in der Kunst nehmen Projekte ein, die Arduino nutzen, um Musik zu machen oder zu steuern. In diesem Buch widmet sich Kapitel 10 der direkten Sounderzeugung und Ausgabe auf dem Arduino. Zudem gibt das Kapitel eine Übersicht über weitere Projekte in diesem Feld und beschreibt, wie man mit dem Arduino eigene Musikinstrumente bauen kann.

Spiele

Ein Mikrocontroller, wie er auf dem Arduino-Board aufgebracht ist, ist längst viel leistungsfähiger als die ersten Spielekonsolen, die Anfang der 1980er in die Wohnzimmer der Welt drängten. Das bedeutet natürlich, dass es möglich ist, einfache Computerspiele auf dem Arduino selbst zu entwickeln oder bekannte Klassiker nachzubauen. In diesem Buch finden sich zwei Spiele: In Kapitel 7 wird ein Würfel mit einem Kontaktsensor gebaut, in Kapitel 5 wird Processing und ein Lichtsensor für ein kleines Geschicklichkeitsspiel verwendet.

Pong auf dem Arduino

Das erste erfolgreiche Computerspiel aller Zeiten war Pong von Atari, das seit 1972 die Welt revolutionierte. Dabei basierte das Spiel nicht auf einem Mikroprozessor, sondern vielmehr auf einem fest verdrahteten Schaltkreis. Das ändert natürlich nichts daran, dass dieses berühmteste Tennis-Computerspiel der Welt auch auf dem Arduino programmiert werden kann. Dabei muss es nicht unbedingt auf dem Fernseher angezeigt werden. Auch Versionen zum Beispiel mit LED-Anzeigen sind möglich. Das Projekt Blinkenlights (*http://www.blinkenlights.org*) nutzt gar die Fassade eines

ganzen Hauses: In den Fenstern sind Lampen angebracht, die von einem zentralen Schaltkreis gesteuert werden. Auch wenn Blinkenlights nicht auf Arduino basiert, zeigt das Beispiel, was alles mit ein bisschen Kreativität und genügend Einsatz möglich ist. Wer seine eigene Pong-Variante mit dem Arduino nachbauen möchte, findet unter anderem bei Alastair Parker (*http://alastair.parker.googlepages.com/arduinopong*) eine Anleitung mit Schaltplan und passendem Quellcode.

Spielend die Hand trainieren

Vor einigen Jahren waren Powerballs kurzzeitig in Mode: Plastikkugeln, in deren Innerem ein schwerer Rotor verankert ist, der durch Kreisbewegungen auf Touren gebracht wird. Dabei baut sich ein Drehmoment auf, das der Hand entgegenwirkt, wenn diese den Ball seitlich kippt. Diese Bälle können dazu verwendet werden, das Handgelenk zu stabilisieren und bei langen Computerarbeiten für den Ausgleich in den Muskeln zu sorgen. Man kann sie aber auch an einen Arduino anschließen und damit Geschwindigkeiten messen. Damit hat der Niederländer Eric Holm ein Spiel namens Gyro als Abschlussarbeit gebaut: Die Messdaten steuern ein kleines Raumschiff, das in die Höhe fliegt und dabei Objekten ausweichen muss. Es ist unter *http://ericholm.nl/gyro/* beschrieben, wenn auch leider nur spärlich, was dem geneigten Bastler natürlich nur als Ansporn dienen sollte, die Funktionsweise selbst herauszufinden und nachzubauen.

Gotcha!

Es mag zwar nicht die feine Art sein, auf Menschen zu schießen, aber es kann durchaus Spaß machen, wenn es sich bei den Geschossen beispielsweise um Farbkugeln handelt. Beim »Lasertag« werden Infrarotstrahlen verwendet, wobei jeder Mitspieler neben seiner Waffe auch die passende Kleidung trägt, die mit Sensoren ausgestattet ist. Diese Sensoren können zum Beispiel mit einem Arduino (hier empfiehlt sich ein LilyPad) verbunden sein, der die Treffer registriert und sofort oder nach Ende des Spiels zur Auswertung an einen Rechner überträgt. Die Beschreibung eines vollständigen Projektes finden Sie unter *http://www.ibm.com/developerworks/views/opensource/libraryview.jsp?search_by=Arduino+laser*. Hiermit sei jedoch darauf hingewiesen, dass Laserspiele in Deutschland mit Verweis auf die Garantie der Menschenwürde im Grundgesetz verboten sind, da es bei den kommerziellen Angeboten darum geht, möglichst viele Treffer zu landen, und nicht, wie bei Paint-

ball oder Gotcha, eine Fahne oder ein Gebiet zu erobern. Nachbauen darf man das Projekt natürlich trotzdem, und wer es spielen möchte, wird sich auch auf moralisch akzeptable Spielregeln einlassen können.

Nicht wackeln!

Labyrinthe aus Holz haben schon Generationen von Kindern frustriert. Sie wissen schon: diese Spiele, bei denen eine Kugel auf einem mit einem Drehknopf in zwei Richtungen schwenkbaren Untergrund vom Start ins Ziel gebracht werden soll, vorbei an Löchern, durch die sie fallen kann. Dank dem Arduino ist nun die Zeit gekommen, diesen garstigen Maschinen ein Schnippchen zu schlagen, denn mithilfe zweier Servos kann das Labyrinth auch ohne zittrige Hände gesteuert werden, wie Jestin Stoffel zeigt. Man kann allerdings auch diesen Triumph über die Schwerkraft ruinieren, indem man ein Wii-Balance-Board anschließt, wie es unter *http://someoneknows.wordpress.com/2009/01/12/arduino-powered-robotic-labyrinth-game/* erklärt wird.

Arduinoboy

Um einen Arduino mit dem Thema Computerspiele in Verbindung zu bringen, braucht es nicht unbedingt ein Spiel: Längst wird auch die Hardware alter und neuerer Konsolen verwendet, um mit dem Arduino zu kommunizieren. Der Arduinoboy hingegen arbeitet mit einem Nintendo Gameboy zusammen, dem Urvater aller Handheld-Konsolen, der seit 1989 weltweit 118 Millionen mal verkauft wurde. Die Verbindung läuft dabei in die andere Richtung: Der Arduino wird an einen Gameboy angeschlossen, um diesem als Midi-Controller zu dienen. Für den Gameboy gibt es mittlerweile eine ganze Reihe von Programmen wie Nanoloop und LittleSoundDJ, die dem Soundchip Klänge entlocken können. Mehr Informationen zu diesem Projekt finden Sie unter *http://code.google.com/p/arduinoboy/*.

Das automatisierte Zuhause

Schon in den zukunftsfreudigen 1950er Jahren erträumte man neben atomgetriebenen Rasenmähern ein vollautomatisches Zuhause. Dank Physical Computing und bezahlbaren Bauteilen gibt es nun immer mehr Menschen, die sich diesen Traum zumindest teilweise erfüllen. Ob es eine Alarmanlage ist, die mögliche Einbre-

cher über Twitter melden kann, oder ein selbst gebauter Anschluss für das kommerzielle X10-System – die Möglichkeiten sind vielfältig. Kapitel 8 zeigt auf, wie mit begrenzten Mitteln eine eigene Heimautomatisierung über das DMX-Protokoll gestaltet werden kann. Hier geht es darum, Lampen und Steckdosen mit dem Arduino zu steuern und so zum Beispiel die Hausbar mit der nötigen Beleuchtung auszustatten.

Ein frisch Gezapftes, bitte!

Nutzer der Bierbrau-Community Brewboard verwenden Arduino, um die Temperatur in ihren Zapfanlagen zu regeln (*http://www.brewboard.com/index.php?showtopic=77935*). Das Projekt »BrewTroller« geht noch ein ganzes Stück weiter und versucht, den Bierbrauprozess mithilfe eines Sanguino, einer Arduino-Variante, zu optimieren. Der BrewTroller misst konstant das Volumen und die Temperatur der einzelnen Braukomponenten und kontrolliert die Hitzequellen, Pumpen und Ventile. Ziel des Ganzen ist der Auto-Brew-Modus, also das vollständig automatisch gebraute Bier. Natürlich zeigt auch dieses Projekt, dass Technik nicht alle Projekte lösen kann: Ob ein Bier wohlschmeckend ist, hängt auch immer von der Qualität der Zutaten ab, und das richtige Rezept kann kein Sensor oder Mikocontroller der Welt ersetzen.

Der Duft frischen Gebäcks

Geht man in ein Café in voller Vorfreude auf einen frisch gebackenen Muffin, kann die Enttäuschung groß sein: Das Gebackene schmeckt wie vom Vortag, weil das frische Blech noch nicht aus dem Ofen ist. Oder es gibt nur noch Schokolade, weil die Blaubeeren heute gar nicht auf dem Programm standen und der Inhaber zu beschäftigt war, um die Kreidetafel am Eingang zu aktualisieren. Der BakerTweet (*http://bakertweet.com/*) soll da Abhilfe schaffen, eine kleine Box, die mit einem Arduino, einem kleinen Drehknopf und Display ausgestattet ist. Dort können vorprogrammierte Backwaren einfach ausgewählt werden, und der Arduino meldet die Nachricht anschließend über eine Netzwerkverbindung an Twitter. Der Vorteil: Cafés können einfach und direkt für sich Werbung machen, und die Gäste wissen sofort Bescheid, wenn es sich lohnt, eine kleine Auszeit zu nehmen. Das erste Café, das den BakerTweet eingesetzt hat, war das Albion Café im Londoner Stadtteil Shoreditch; auf der Webseite des Projektes finden sich aber alle Details, um selbst in das twitternde Bäckerbusiness einzusteigen.

Gadgets

Nicht alle Physical-Computing-Projekte müssen direkt einleuchten oder einen Anwendungszweck verfolgen. Oft sind es auch die kleinen Ideen, die man schnell mithilfe eines Arduino umsetzen kann. Das können kleine Experimente sein oder nutzlose, aber schöne Ausstellungsobjekte. In Kapitel 4 erhält auch dieses Buch sein kleines Gadget: eine Gehirnwellenmaschine, mit der man sich selbst in einen Modus der Entspannung versetzen kann.

Schlaf beobachten mit dem Arduino

Weil die ehemalige MIT-Studentin Anita Lilie mehr über ihren Schlafrhythmus erfahren wollte, erfand sie den Sleeptracker. In erster Linie wollte sie damit herausfinden, warum sie so ein Morgenmuffel war. Sie hatte bemerkt, dass sie immer nur dann unausgeschlafen war, wenn sie von einem Wecker geweckt wurde. Wenn sie ohne aufwachte, ging es ihr gut. Sie ging von der Annahme aus, dass es nicht die Menge an Schlaf war, die das Problem verursachte, sondern vielmehr der Zeitpunkt innerhalb eines 90-minütigen Rahmens, in dem der Schlaf tiefer und leichter wird. Das Ziel war also, nicht etwa 6 oder 10 Stunden zu schlafen, sondern den genauen Zyklus herauszufinden, um sich während besonders leichten Schlafs wecken lassen zu können. Also schloss sie mehrere Beschleunigungssensoren an einen Arduino an und zeichnete die Daten auf. So konnte sie feststellen, wie lang ihr Schlafzyklus dauerte, und den Wecker entsprechend programmieren. Ob sie damit Erfolg hatte, ist leider nicht dokumentiert, was wohl auch daran liegt, dass die Daten sich als deutlich komplexer als erwartet darstellten. Leider ist das Projekt auch nicht als perfekter Wecker geeignet, da niemand gern jede Nacht mit drei langen Kabeln und Sensoren am Körper verbringen möchte. Dennoch ist es sicher interessant und empfiehlt sich für jeden, der neugierig ist und zufällig gerade drei Beschleunigungssensoren zu Hause hat. Eine genaue Dokumentation finden Sie unter *http://flyingpudding.com/projects/sleep_tracker/*.

Schöne Bilder

Digitale Bilderrahmen sind immer noch recht teuer und natürlich viel langweiliger als ein selbst gebasteltes Projekt. Das March Frame Project (*http://nat.org/blog/2009/04/march-frame-project/*) kann zwar keine digitalen Bilder anzeigen, aber immerhin mit ein-

fachen Methoden zwischen drei unterschiedlichen Motiven wechseln. Zwei dieser Bilder werden dabei in den Farben Rot und Blau übereinandergelegt und auf ein Papier gedruckt. Das dritte ist eine Schablone, die dahinter angebracht wird. Der Arduino steuert nun drei unterschiedliche LEDs (eine rote, eine blaue und eine weiße), die diese Motive zum Leuchten bringen. Wenn eines der beiden farbigen Lichter leuchtet, löscht es das entsprechende Motiv auf dem Bild aus, das andere wird klar dargestellt. Sind beide aus, kann die weiße LED im Hintergrund die Schablone auf das Papier projizieren. Das Ergebnis ist ein schöner Bilderrahmen (sofern die Elektronik stilvoll verkleidet wurde), der mit Sicherheit für Aufsehen unter Besuchern sorgt.

Die glückliche Katze

Wer eine Katze besitzt, weiß, dass sie recht unruhig werden kann, wenn der Futter- oder Trinknapf wieder einmal leer ist. Ein fürsorglicher Besitzer, der über einen Arduino verfügt, kann diesen dafür nutzen, rechtzeitig Meldung zu erstatten oder zumindest beim Wasser sogar automatisch für Nachschub sorgen. Unter *http://scoopmycatbox.com/blog/2009/03/ultrasonic-cat-dish-up/* findet sich ein Projekt, das mit einem Ultraschallsensor den Wasserstand im Napf messen und sich bemerkbar machen kann. Der Erschaffer des Projektes arbeitet derzeit auch daran, das Nachfüllen zu automatisieren, was nicht allzu schwer sein dürfte – Nachahmer und Bastler, die weitere Ideen umsetzen, sind dort sicherlich willkommen!

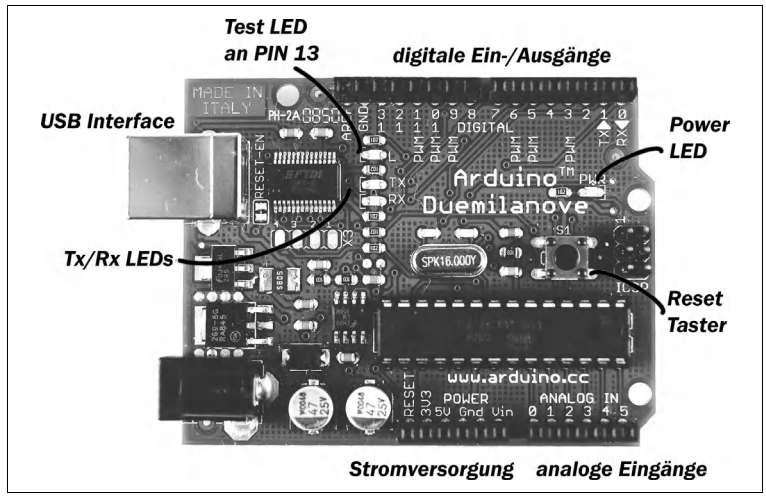
Hardware

Hält man zum ersten Mal ein Arduino-Board in der Hand, erkennt man nur einzelne Teile wieder, zum Beispiel den Anschluss für das USB-Kabel. Anders als beim PC sollte aber ein Arduino-Nutzer alle Bauteile seines Gerätes genau kennen.

Beschreibung

Alle Erklärungen in diesem Buch beziehen sich auf den Arduino Duemilanove, die 2009er-Ausgabe des Arduino-Boards. Dieses Board besteht im Wesentlichen aus den folgenden Bestandteilen.

Abbildung 1-3 ▶
Das Arduino-Board mit seinen einzelnen Bestandteilen



Mikrocontroller

Auf dem Arduino Duemilanove ist ein Atmega168- oder Atmega328-Mikrocontroller angebracht, ein kleiner Computer, der mit 16 MHz Taktfrequenz arbeitet. Er besitzt zwar wenig Speicher (16 bzw. 32 KByte Speicher und 2 bzw. 4 KByte Arbeitsspeicher) und kann nur 8-Bit-Befehle verarbeiten, dafür aber direkt mit anderer Hardware verbunden werden. Natürlich besitzt dieser »Computer« weder Festplatte noch Tastatur oder Display, aber er ist in der Lage, kleine Programme auszuführen, die meist jeweils eine einzelne Aufgabe erledigen.

Im Gegensatz zu anderen Entwicklungsboards muss der Mikrocontroller beim Arduino nicht direkt programmiert werden, der Code wird vor dem Upload automatisch umgewandelt, erweitert und in Maschinensprache übersetzt (kompiliert). Diese Maschinensprache ist zwar durchaus für Menschen lesbar, aber sehr abstrakt und aufwendig zu programmieren.

USB-Port und USB-Chip

Der USB-Port verbindet das Board mit dem Computer. Dabei dient er als Daten- und Stromquelle zugleich: Über das Kabel können neue Programme hochgeladen werden, denn die 5 Volt und 0,5 Ampere, die über USB fließen, bieten genügend Strom. Der Arduino kann aber auch konstant über USB Daten austauschen, etwa um als Steuerungselement für Computersoftware zu dienen. Da USB ein verhältnismäßig komplexes Protokoll ist, wird die Kommunikation im USB-Chip umgewandelt.

Stromanschluss

Um Arduino-Programme auch unabhängig von einem Computer laufen zu lassen, kann man auch ein Netzteil anschließen. So kann das Board in ein größeres Projekt eingebaut werden und auch dann laufen, wenn der Computer nicht angeschaltet ist. Optimal läuft das Board bei 7 bis 12 Volt, wobei es dabei selbst etwa 40 mA benötigt. Das kann sich aber deutlich erhöhen, wenn man zum Beispiel LEDs an den Arduino anschließt.

Reset-Schalter

Das aktuell laufende Programm lässt sich natürlich auch neu starten, wenn das Board nicht mit dem Computer verbunden ist. Dafür dient der Reset-Schalter: Wird der rote Knopf betätigt, beendet der Arduino alle derzeitigen Aktionen und begibt sich in den Anfangszustand, den er auch beim Einschalten des Netzteils oder der USB-Stromquelle hatte. Anschließend wird das aktuell auf dem Arduino geladene Programm von Neuem ausgeführt.

Betriebsleuchte und Übertragungsleuchten

Ist der Arduino angeschaltet und es läuft ein Programm, so leuchtet die mit *PWR* (für *Power*) markierte LED. Werden zwischen Arduino und Computer Daten übertragen, blinken die mit *TX* und *RX* markierten Leuchtdioden. *TX* steht für *Transmitter*, *RX* für *Receiver*, es wird also angezeigt, ob das Board gerade Daten sendet oder empfängt.

Bootloader/ICSP-Header

Schließt man den Arduino über ein *USB-Kabel* an, kommuniziert man nur mit dem sogenannten Bootloader – einem sehr kleinen Programm, das immer auf dem Chip vorhanden ist. Sobald der Prozessor mit Strom versorgt wird, wird es ausgeführt. Der Arduino-Bootloader ermöglicht es, über die USB-Schnittstelle ein neues Programm hochzuladen. Dadurch, dass der Bootloader immer zuerst ausgeführt wird, ist es möglich, auch bei einem komplett defekten Programm wieder eine korrigierte Version hochzuladen.

Über den *ICSP-Header* (*In-Circuit Serial Programming*) ist aber auch die direkte Kommunikation mit dem Mikroprozessor möglich. Dafür ist allerdings ein sogenannter *Programmierer* notwendig, der an den Computer angeschlossen wird und über diese In-

Circuit-Schnittstelle ein neues Programm auf den Arduino hochladen kann. Für die in diesem Buch erklärten Workshops ist die Benutzung des ICSP-Header nicht nötig. Die Verwendung eines externen Programmiergeräts wird allerdings dann gebraucht, wenn ein neuer Bootloader auf den Arduino-Prozessor hochgeladen werden soll (z.B. wenn man den Atmega168 auf einem älteren Arduino Duemilanove durch einen Atmega328 austauschen möchte). Es ist auch möglich, einen Arduino-Prozessor im Internet zu bestellen, der schon mit einem Bootloader programmiert ist.

Digitale Pins

An der oberen Seite des Boards befinden sich 16 Steckplätze. 14 davon sind sogenannte digitale *I/O-Pins* (*Input/Output*). Sie sind mit den Zahlen 0 bis 13 markiert und dienen als Anschlussmöglichkeit für alle *Sensoren* und *Aktoren*, die zum Betrieb nur digitale Signale, also Einsen und Nullen benötigen, um etwas ein- und auszuschalten. Dazu gehören zum Beispiel *LEDs* und *Schalter*, wie sie in den Workshops ab Kapitel 3 verwendet werden. Sechs dieser Pins haben zusätzlich die Beschriftung *PWM*. Mit dieser sogenannten *Pulsweitenmodulierung* können beispielsweise LEDs gedimmt werden (mehr dazu in Kapitel 3, Pulsweitenmodulierung). Beachten sollte man zusätzlich, dass die Pins 0 und 1 auch für die serielle Kommunikation mit dem Arduino (siehe Kapitel 5) verwendet werden. Setzt man diese gleichzeitig als digitale Aus- oder Eingänge ein, kann es zu merkwürdigen Nebenwirkungen kommen. Man sollte also beachten, dass sich serielle Kommunikation und andere Verwendung des Pins nicht in die Quere kommen. Außerdem kann über die Pins 10 bis 13 mit dem SPI-Protokoll kommuniziert werden, das allerdings nicht Bestandteil der Arduino-Sprache ist, weshalb es in diesem Buch nicht verwendet wird.

GND

Auf dem Board gibt es zwei Massepunkte, die als Minuspol für den Stromkreis dienen. Sie sind mit *GND* (*Ground*) beschriftet und befinden sich in den beiden Pinreihen oben und unten.

Analoge Pins

Die rechte Pinreihe auf der unteren Seite des Boards besteht aus sechs *Analogen Input-Ports*, die in der Lage sind, analoge Signale in *digitale 10-Bit-Signale* umzuwandeln. Das heißt, diese Signale können 1.024 Werte annehmen. Dabei wird von einer maximalen

Eingangsspannung ausgehend ein Wertebereich festgelegt. Liegt nun am Pin eine Spannung an, besteht diese aus einem Bruchteil des Maximums. Dieser Bruchteil kann nun als Wert zwischen 0 und 1.023 weiterverarbeitet werden.

Bezugsquellen

Und woher bekommt man solch ein Arduino-Board? Natürlich aus dem Internet.

Tatsächlich gibt es eine Handvoll Onlinehändler in Deutschland, die sowohl den Arduino Duemilanove als auch seine Vorgänger, die Bluetooth-Version und eine Vielzahl von Shields verkaufen. Die Preise unterscheiden sich kaum, sodass andere Faktoren wie Versandkosten oder die zusätzliche Verfügbarkeit von Materialien eine weitaus wichtigere Rolle spielen sollten. Eine aktuelle Liste von deutschen Arduino-Händlern wird auf der Projektwebsite unter *http://www.arduino.cc* geführt.

Besonders empfohlen sei das »Bausteln«-Projekt, auf dessen Website *http://www.bausteln.de* nicht nur Arduino-Boards verkauft werden, sondern die gesamte Philosophie behandelt wird. Die Betreiber bieten weitergehenden Support an, veranstalten monatlich Baustel-Abende und arbeiten daran, eine deutsche Community rund um Physical Computing und »Do-it-yourself«-Elektronik aufzubauen.

◀ **Abbildung 1-4**
bausteln.de

Nun, da Sie vermutlich ein Arduino-Board besitzen und auch wissen, wie es aufgebaut ist, kann mit der Installation der Entwick-

lungsumgebung begonnen werden. Im folgenden Abschnitt wird erklärt, wie das Arduino-Board an den PC angeschlossen und wie das erste Programm auf dem Arduino ausgeführt wird.

Die Arduino-Entwicklungsumgebung

Installation der Arduino-Software

Die Arduino-Entwicklungsumgebung kann von der Arduino-Website http://www.arduino.cc für alle drei gängigen Betriebssysteme (Windows, Linux, Mac OS X) heruntergeladen werden (zum Zeitpunkt des Schreibens dieses Buches ist die aktuelle Version *arduino-018*). Die Seite, die die aktuelle Version zur Verfügung stellt, ist *http://www.arduino.cc/en/Main/Software*. Dort kann das Archiv für das jeweilige Betriebssystem heruntergeladen werden. Die Umgebung enthält den Arduino-Editor, die Arduino-Dokumentation sowie eine Anzahl von Beispielprogrammen und Libraries. Im folgenden Abschnitt wird die Installation der Arduino-Umgebung unter Windows und Mac OS X beschrieben. Die Installation unter Linux ist ein bisschen komplizierter und hängt auch sehr von der verwendeten Linux-Distribution ab. Deswegen verweisen wir auf die Wiki-Seite zur Linux-Installation, http://www.arduino.cc/playground/Learning/Linux.

Nachdem die Archivdatei für die Arduino-Umgebung heruntergeladen wurde, muss sie mit einem Doppelklick entpackt werden: Es wird ein Ordner erzeugt, der *arduino-[version]* heißt. Dieser Ordner kann jetzt an eine beliebige Stelle kopiert werden, zum Beispiel in den Dokumentenordner oder nach *C:\Program Files* unter Windows oder nach */Applications* unter Mac OS X. Um Arduino zu starten, genügt es dann, in diesem Ordner das Programm *Arduino* zu öffnen. Vorher müssen allerdings noch ein paar Schritte ausgeführt werden.

Mitgeliefert werden auch Treiber für den USB-Baustein, der auf dem Arduino-Board installiert ist. Dieser Chip ist ein FTDI-seriell-nach-USB-Konvertierer. Für Windows und Mac OS X sind Treiber mitgeliefert, die installiert werden sollten, bevor das Board an den Computer angeschlossen wird. Unter Linux sind diese Treiber meistens schon im Distributions-Kernel enthalten. Je nachdem, welches Arduino-Board benutzt wird (ältere Boards haben noch keine eingebaute USB-Schnittstelle), muss ein externer USB-nach-seriell-Adapter verwendet werden.

Installation der Treiber unter Windows

Unter Windows muss zuerst das Arduino-Board an den Computer angeschlossen werden. Da das Betriebssystem noch keine Treiber für die USB-Schnittstelle installiert hat, taucht der Installationsassistent für unbekannte Hardware auf. Die automatische Installation von der Microsoft-Website und über Windows Update muss übersprungen und die Installationsoption *Von einer Liste oder bestimmten Quelle installieren* ausgewählt werden. Danach wird die Weiter-Taste betätigt. Aktivieren Sie die Option *Diese Position auch durchsuchen* und wählen Sie (über die Durchsuchen-Schaltfläche) den Ordner aus, in dem die lokale Arduino-Installation sich befindet. In diesem Ordner muss der Unterordner *Drivers\FTDI USB Drivers* ausgewählt werden. Anschließend kann über *OK* und *Weiter* die Installation durchgeführt werden. Diese Schritte müssen noch einmal wiederholt werden, weil die USB-nach-seriell-Schnittstelle zwei Treiber benötigt: einen für die USB Schnittstelle an sich und einen, um diese als seriellen Port im Betriebssystem anzubieten.

Nach diesen Schritten können das Arduino-Board erneut an den Computer angeschlossen und die Arduino-Entwicklungsumgebung gestartet werden.

Installation der Treiber unter Mac OS X

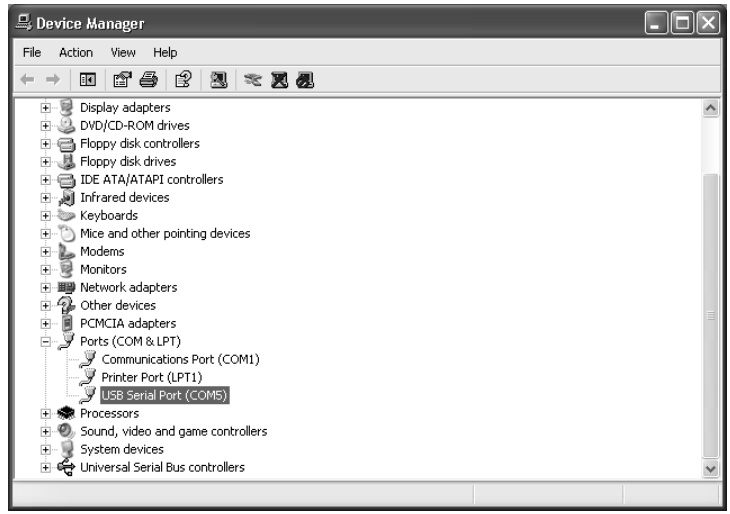
Unter Mac OS X sind im Unterordner *Drivers* die Treiber für den FTDI-USB-Chip zu finden. Diese heißen *FTDIUSBSerialDriver_x_x_x.dmg*, wobei *x_x_x* für die aktuelle Versionsnummer steht. Wird ein Macintosh-Rechner mit Intel-Prozessor verwendet (wie bei den meisten neuen MacBooks, Mac Minis, Mac Pros und iMacs), müssen die Treiber mit »Intel« im Namen installiert werden. Wenn ein älterer Macintosh-Rechner mit G4- oder G5-Prozessor verwendet wird, werden jene ohne »Intel« im Namen benötigt. Nachdem das Disk-Image geöffnet wurde, führen Sie das Installationsprogramm von FTDI aus. Die Installationsanleitung wird auf dem Bildschirm angezeigt, und Sie werden aufgefordert, das Administrator-Passwort einzugeben (was alles seine Richtigkeit hat, denn es handelt sich um Systemtreiber, die mit Administratorrechten installiert werden müssen). Nach der Installation ist es am sinnvollsten, den Rechner neu zu starten, um sicherzustellen, dass die neuen Treiber auch geladen werden. Danach kann das Arduino-Board an den Rechner angeschlossen werden. Funktioniert alles,

müsste die PWR-LED grün leuchten und die LED, die mit *L* gekennzeichnet ist, regelmäßig blinken. Ist das nicht der Fall, wird in unter »Fehlersuche in Elektronischen Schaltungen« erklärt, wie häufige Fehlerquellen zu erkennen und zu beheben sind.

Anschließen und Starten

Der Arduino Duemilanove wird mit einem USB-Kabel an den Rechner und damit an die Entwicklungsumgebung angeschlossen. Als nächster Schritt muss jetzt die Entwicklungsumgebung konfiguriert werden, damit sie die korrekte serielle Schnittstelle benutzt und die korrekte Boardbezeichnung verwendet. Im *Tools*-Menü der Arduino-Umgebung muss im Untermenü *Serial Port* die serielle Schnittstelle ausgewählt werden, an die das Arduino-Board angeschlossen ist. Unter Mac OS X heißt diese Schnittstelle meistens */dev/cu.usbserial-x*.

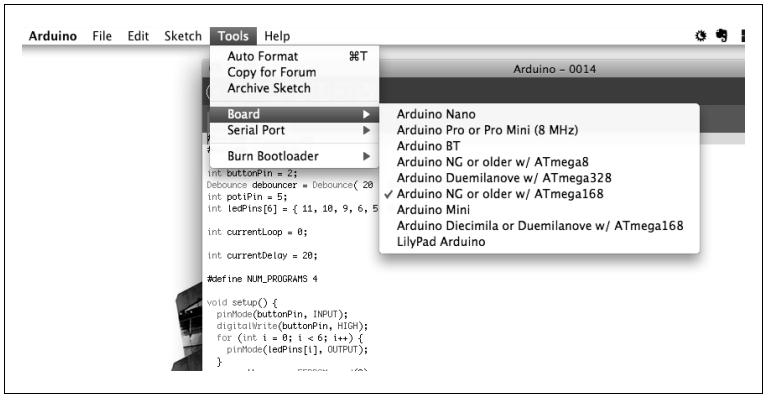
Abbildung 1-5 ▶
Der Windows Gerätemanager



Unter Windows ist dieser Schritt ein bisschen komplizierter, weil Windows standardisierte Namen mit *COMX:* an alle seriellen Schnittstellen vergibt. Um den genauen Namen der Arduino-USB-Schnittstelle zu finden, muss der Gerätemanager gestartet werden (über START → EINSTELLUNGEN → SYSTEMSTEUERUNG → COMPUTER). Unter dem Eintrag *Ports (COM & LPT):* werden die USB-Schnittstelle des Arduino und der entsprechende *COMX:*-Name

angezeigt. Eine Eigenart von Windows ist, dass diese Bezeichnung sich ändert, wenn das Arduino-Board an eine andere USB-Schnittstelle des Computers angeschlossen wird.

Im nächsten Schritt muss noch das korrekte Arduino-Board eingerichtet werden. Durch diese Einstellung weiß die Arduino-Entwicklungsumgebung, welcher Prozessor und welche Pin-Konfiguration auf dem Arduino-Board verwendet werden. Unter dem *Tools*-Menü müssen Sie im Untermenü *Board* das korrekte Board auswählen. Im Fall des Arduino Duemilanove muss sichergestellt werden, dass auch der korrekte Prozessor ausgewählt wird. Ältere Arduino Duemilanove-Boards benutzten den Atmega168-Prozessor, während neuere Boards den Atmega328-Prozessor verwenden.



◀ **Abbildung 1-6**
Boardkonfiguration

Falls die entsprechende Board-Option nicht im Untermenü vorhanden ist, muss eine neuere Version der Arduino-Entwicklungsumgebung heruntergeladen werden.

Die Arduino-Entwicklungsumgebung

Das Fenster der Umgebung besteht aus drei Hauptbestandteilen, nämlich der Symbolleiste und zwei Fenstern für den Programmcode und die Kommunikation mit dem Arduino. Das schwarze Kommunikationsfenster wird auch »Konsole« genannt.

Nachdem das Kabel verbunden wurde, müssen Sie in der Programmierumgebung das Board und den *seriellen Port*, also die Verbindung, unter der die Kommunikation laufen soll, auswählen. Beide Menüpunkte finden sich unter *Tools*.

Die Symbolleiste

Die Symbolleiste besteht aus acht Symbolen, von denen mindestens drei aus fast jedem anderen Programm bekannt sein sollten.

Abbildung 1-7 ▶
Die Symbolleiste

Abbildung 1-8 ▶
Das Symbol New

Das Symbol *New* öffnet einen neuen Sketch. Als »Sketches« werden Programme bezeichnet, wobei jedes Programm aus mehreren Dateien bestehen kann. Dieser Sketch ist komplett leer. Wird gerade ein anderer Sketch bearbeitet, sollte dieser gespeichert werden, wobei das Programm dabei auch noch einmal nachfragt.

Abbildung 1-9 ▶
Das Symbol Open File

Das Symbol *Open File* öffnet einen gespeicherten Sketch. Damit lassen sich auch Bibliotheken, die aus dem Netz heruntergeladen wurden, im Arduino-Programm ansehen und bearbeiten. Generell werden alle Sketches als Dateien gespeichert, sodass sie auch mit anderen Nutzern ausgetauscht werden können.

Abbildung 1-10 ▶
Das Symbol Save

Das Symbol *Save* speichert den aktuellen Sketch. Gerade beim Programmieren ist es hier wichtig, daran zu denken, dass man alle Versionen, die man ausprobiert, getrennt abspeichert. Oft kommt man erst nach langem Probieren darauf, dass ein Teil der ersten Version eine gute Lösung war.

Die restlichen Symbole sind in zwei Gruppen aufgeteilt: die Steuerung des Arduino sowie die Kommunikation mit ihm. Die Programmsteuerung ist dabei recht einfach.

Abbildung 1-11 ▶
Das Symbol Verify

Mit diesem Knopf wird der aktuelle Sketch kompiliert, also in Maschinensprache übersetzt, und dabei auf seine Korrektheit überprüft. Das Ergebnis des Kompiliervorgangs sowie die Größe des erzeugten Programms werden in der Konsole angezeigt. Damit kann überprüft werden, ob das Programm auch in den Speicher des Arduino passt (die Arduino-Variante mit Atmega168 hat 16 KByte CodeSpeicher, von denen 2 KByte vom Arduino-Bootloader belegt sind. Es sind also 14 KByte für den kompilierten Sketch übrig. Die Arduino-Variante mit Atmega328 hat 32 KByte Codespeicher, von denen 30 KByte für den kompilierten Sketch übrig sind). Ein Sketch belegt allerdings nicht nur Codespeicher, sondern auch Arbeitsspeicher. Dieser wird nach der Kompilierung als »data« angezeigt. Die

Arduino-Variante mit Atmega168 hat 2 KByte Arbeitsspeicher, die Arduino-Variante mit Atmega328 4 KByte. Es ist wichtig, sicherzustellen, dass genügend Speicher vorhanden ist. Ansonsten kann sich das Programm sehr merkwürdig verhalten. Nun kann man mit dem Upload-Knopf das Programm auf das Board laden.

◀ **Abbildung 1-12**
Das Symbol Stop

Das Stop-Symbol beendet das laufende Programm auf dem Arduino. Der Arbeitsspeicher wird anschließend gelöscht, sodass bei einem erneuten Start der gesamte Programmablauf wieder von vorn beginnt.

◀ **Abbildung 1-13**
Das Symbol Upload

Das Upload-Symbol verarbeitet (»kompiliert«) den aktuellen Sketch in Maschinencode und lädt ihn auf den Arduino. Er funktioniert also wie der Verify-Knopf, transportiert das Programm jedoch danach auf den Arduino. Anschließend wird auf dem Board ein Reset ausgeführt und damit begonnen, den Sketch abzuspielen.

◀ **Abbildung 1-14**
Das Symbol Console

Die Konsole dient als Kommunikationskanal zwischen dem Arduino und der Programmierumgebung, sobald das Programm einmal hochgeladen und gestartet ist. Dabei können Daten mit bis zu 9. 600 Bits pro Sekunde ausgetauscht werden, was genug ist, um Kommandos an den Arduino zu senden und Informationen zu empfangen. Eine genauere Erklärung der Kommunikation über die serielle Konsole findet sich in Kapitel 6.

Das Codefenster

Bevor mit der eigentlichen Programmierung in der Arduino-Umgebung begonnen wird, wird ein Beispielprogramm auf das Arduino-Board hochgeladen. Dazu wird eins der Beispiele, die mit der Umgebung mitgeliefert werden, geöffnet. Über FILE → SKETCHBOOK → EXAMPLES können diese Beispiele ausgewählt werden. Wie man sehen kann, wird schon eine große Menge von Beispielen mitgeliefert, die man sich als Anregung und als Anleitung anschauen kann. Zum Testen der Umgebung laden Sie über DIGITAL → BLINK das Programm *Blink*, das die LED, die sich auf dem Arduino-Board befindet, blinken lässt. Nachdem die Datei geöffnet ist, kann sie übersetzt und auf das Arduino-Board hochgeladen werden. Dazu muss der Upload-Knopf betätigt werden. Das Kompilieren kann ein paar Sekunden dauern, danach beginnen die RX- und TX-LEDs auf dem Arduino-Board zu flackern (was bedeutet, dass das übersetzte Programm übertragen wird). Nach einen paar weiteren Sekunden wird das Programm dann auch ausgeführt: Die LED

beginnt zu blinken und die Arduino-Entwicklungsumgebung zeigt *Upload Completed* an. Falls es Probleme beim Hochladen gibt, meldet die Umgebung in der braunen Zeile den Fehler. Als Erstes sollte dann überprüft werden, ob das Arduino-Board korrekt an den Computer angeschlossen ist und ob das richtige Board und die richtige serielle Schnittstelle eingestellt wurden. Weitere mögliche Probleme werden unter »Fehlersuche« in Kapitel 2 behandelt.

Code wird in der Arduino-Umgebung geschrieben wie in jedem anderen Texteditor auch. Dabei wird das sogenannte *Syntax Highlighting* unterstützt: Einzelne Programmelemente werden als solche erkannt und so eingefärbt, dass sie sich leicht erkennbar von den anderen unterscheiden. So lassen sich Fehler schon beim Schreiben des Codes vermeiden. Beim Speichern einer Datei wird diese in einem eigenen neuen Ordner unterhalb des sogenannten Sketchbook (also Skizzenbuch) gespeichert. Das Verzeichnis für das Skizzenbuch wird beim ersten Starten von der Arduino-Entwicklungsumgebung erstellt (unter Mac OS X unter *Dokumente/Arduino*, unter Windows in Eigene Dateien*Arduino*. Der Pfad zu diesem Verzeichnis kann leicht in den Einstellungen von Arduino geändert werden (unter Mac OS X über ARDUINO → PREFERENCES, unter Windows und Linux FILE → PREFERENCES). Dort lässt sich auch einstellen, ob jede neue Datei zuerst benannt werden soll (was sehr praktisch ist, weil das Skizzenbuch sich sonst schnell mit vielen »*sketch_XXX*« benannten Dateien füllt) und ob leere Dateien beim Verlassen der Umgebung gelöscht werden sollen. Weitere Einstellungen für fortgeschrittene Benutzer können in der Datei *preferences.txt*, deren Pfad bei den Arduino-Einstellungen angezeigt wird, von Hand mit einem normalen Texteditor eingetragen werden.

Mit dem Programmieren beginnen

Nun ist alles vorbereitet, um mit der Programmierung des Arduino zu beginnen. In den folgenden Kapiteln sollen einzelne Workshops nach und nach die Programmiersprache, aber auch die elektronischen und physikalischen Grundlagen erklären. Dabei genügt unser Platz natürlich nicht, um ausführlich große Basteleien zu erläutern. Aber das, was erklärt wird, reicht aus, um anschließend weitergehende Ideen zu verwirklichen. Zudem sind die einzelnen Übungen so angelegt, dass sie in kurzer Zeit und mit wenig Mitteln selbst ausprobiert werden können: Alle Bauteile sind günstig und im normalen Elektronikfachhandel erhältlich, wenn sie nicht sogar,

etwa aus Drähten, selbst hergestellt werden können. Eine umfassende Referenz zur Arduino-Programmiersprache finden Sie im Anhang. Sie erklärt strukturiert und ausführlich alle Bestandteile der Sprache und ist somit auch zum schnellen Nachschlagen geeignet, wenn Sie weitere Ideen ausprobieren wollen.

Um die Workshops sinnvoll auszuprobieren, empfiehlt es sich, eine Arbeitsecke einzurichten. Hier sollte genügend Platz frei sein, um Drähte zu schneiden und Bauteile zu verlöten. Nach jedem Basteln ist es auch wichtig, diesen Platz wieder sauberzumachen, denn am nächsten Tag hat man schon mal Probleme damit, eine halb fertig liegengelassene Schaltung noch zu verstehen.

Zudem erklärt ein kleiner Exkurs die physikalischen Grundlagen rund um Spannung, Stromstärke und Widerstand. Außerdem wird der Begriff des Physical Computing ein wenig präzisiert, wozu auch ein Ausblick auf bekannte und interessante Arduino-Projekte gehört.

KAPITEL 2

Physical Computing, elektrische Grundlagen und der Sprung ins kalte Wasser

In diesem Kapitel:

- Elektrische Grundlagen
- Schaltungen, Bauteile und Schaltbilder
- Löten
- Fehlersuche in elektronischen Schaltungen

Physical Computing ist die Verbindung von physikalischen Systemen mit Software und Hardware zu einem interaktiven Ganzen. Oft wird ein Computer immer noch als Gerät betrachtet, das nur über den Bildschirm und die Tastatur zu bedienen ist und Programme ausführt. In diesem Buch wird aber nicht mit Computern und Computerprogrammen gearbeitet, sondern mit dem allgemeineren Konzept des »informatischen Rechnens« (englisch »computing«). Mittlerweile sind in vielen alltäglichen Gegenständen kleine eigenständige (eingebettete) Computer, oder Mikrocontroller, eingebaut, die über physikalische Schnittstellen mit Umfeld und Benutzer kommunizieren. Diese Schnittstellen unterscheiden sich oft sehr von Tastatur, Maus und Display. So steckt zum Beispiel in einer Kaffeemaschine ein kleiner Computer, der zwar über ein Display und Tasten mit dem Benutzer kommuniziert, aber gleichzeitig auch über Temperatursensoren und Feuchtigkeitssensoren das Kochen von Kaffee überwacht, über einen Motor Kaffeebohnen mahlen und mit verschiedenen Pumpen und Druckventilen Wasser erhitzen und Milchschaum erzeugen kann. Beim Physical Computing geht es also um die Verbindung von Rechnen und Rechnern auf der einen Seite und der realen Welt auf der anderen. Mehr und mehr Gegenstände werden so zu kleinen intelligenten Geräten: eine elektrische Zahnbürste, die verschiedene Programme kennt, eine Lampe, die sich nur anschaltet, wenn Menschen anwesend und wach sind, ein Kühlschrank, der seinen eigenen Inhalt und Stromverbrauch überwacht und seinem Besitzer meldet, wenn bestimmte Ware nachbestellt werden muss.

Der Vorteil dieser intelligenten Geräte ist, dass man den Umgang mit ihnen deutlich flexibler gestalten kann. So werden viele kleine Vorgänge auf den Benutzer direkt angepasst. Schließt man einen Schalter an eine Lampe an, ohne einen Mikrocontroller einzusetzen, lässt sich kaum mehr erreichen, als dass man durch Betätigen des Schalters das Licht an- oder ausschaltet. Mit einem Mikrocontroller kann man allerdings das Licht zum Beispiel für eine bestimmte Zeit nach einem Schalterdruck anschalten (ähnlich wie mit einer Zeitschaltuhr), oder nur dann, wenn andere Lichter schon aus sind. Es ist auch möglich, über diesen einen Schalter deutlich kompliziertere Vorgänge zu erkennen: Der Benutzer kann etwa durch langes Drücken auf den Schalter den Zeitschaltmodus aktivieren oder die Lampe regelmäßig blinken lassen. Solche einfachen Lichtsteuerungen findet man heutzutage in jeder Fahrradbeleuchtung: Die unterschiedlichen Blinkmuster lassen sich ohne »Rechnen« nicht einfach bauen.

Der Arduino-Prozessor ist ein gutes Beispiel für einen solchen Mikrocontroller. Er verfügt über genügend Rechenleistung, um die meisten Anwendungen locker zu erledigen, und hat mit seinen Aus- und Eingängen Platz für Sensoren und Aktoren, mit denen viele Projekte umgesetzt werden können.

Der Begriff »Physical Computing« beschreibt diese Kommunikation zwischen Benutzer und einem physikalischen elektrischen Gerät. Dieses kann eingehende Information (auf Englisch »Input«) verarbeiten und anhand eines Programms in eine Reihe von Ausgangssignalen (auf Englisch »Output«) umwandeln. Ziemlich jedes Programm kann als fortgehende Schleife dargestellt werden, die Input zu Output verarbeitet. Dieser Ansatz wird auch in der Struktur von Arduino-Programmen deutlich, wie Sie im entsprechenden Abschnitt am Ende dieses Kapitels sehen werden.

Input erlaubt der realen Welt, mit dem Computer zu kommunizieren. Sensoren können ihre Umwelt wahrnehmen und in Signale verwandeln, die der Rechner verstehen kann. Dabei unterscheidet man zwischen digitalem Input, der nur zwei Zustände (An und Aus) kennt, und analogem mit einer Vielzahl von Zwischenwerten. Neben Sensoren gehören zum Input auch all jene Bauteile, über die ein Mensch Eingaben machen kann, wie etwa Schalter.

Die Ausgänge (oder auch der Output) ermöglichen es, die reale Welt in der Umgebung des Computers zu verändern. Aktoren

manipulieren Licht und Ton oder setzen Dinge in Bewegung. Generell ist der Output dabei der schwierigere Teil des Physical Computing, weil er oft auch vom Erschaffer verlangt, dass dieser sich mit den elektrischen und mechanischen Eigenschaften seiner Aktoren auskennt und sie sinnvoll einsetzt. Es ist zum Beispiel viel einfacher, die Umgebungshelligkeit zu messen, als sie mit richtig positionierten und angesteuerten Lichtern zu verändern. Zum Output zählt es aber auch, dass Bilder auf Monitoren angezeigt oder Töne an Soundkarten ausgegeben werden, ein Mikrocontroller also weitere angeschlossene Chips (und unter Umständen auch eigenständige Computer bzw. Mikrocontroller) steuert. Die gängigsten Sensoren und Aktoren werden in Kapitel 7 vorgestellt.

Ein wichtiger Bereich des Physical Computing ist die Erweiterung der Benutzerschnittstelle herkömmlicher Desktop-Computer. Normale Software wird angepasst, damit sie mit physikalischen Gegenständen interagieren kann. Dafür kann zum Beispiel die Programmierumgebung Processing verwendet werden, die sich als Schnittstelle zwischen Input und Output anbietet. Processing besteht aus einer einfachen Programmiersprache, die vor allem die Entwicklung von grafischen Programmen sehr leicht macht. In diesem Buch taucht Processing deshalb an vielen Stellen auf, zum Beispiel in Kapitel 5. Processing ist eng mit der Arduino-Programmierumgebung verwandt, in der Tat handelt es sich bei Arduino um eine Weiterentwicklung davon, um Programme auf Mikrocontrollern schreiben zu können.

Ein Beispiel für eine solche Benutzerschnittstellenerweiterung sind die Multitouch-Oberflächen, die es dem Benutzer erlauben, mit mehreren Fingern oder Händen direkt mit einem Bildschirm zu interagieren. Solche Oberflächen erinnern an den Film »Minority Report«, und es gibt im Internet beeindruckende Videos auf den Seiten der Forschungsgruppen, die sich mit diesem Konzept beschäftigen. Sie lassen sich relativ einfach mit Kameras und Beamern bauen und erobern zum Beispiel als Multitouch-Tische die Museen und Bühnen dieser Welt. Es wird nicht mehr lange dauern, bis wir diese auch in der freien Wildbahn beobachten können (das iPhone und das iPad von Apple sind ein gutes Beispiel für diese Technologie). Neue Eingabegeräte bedeuten auch einen neuen Umgang mit Computern, die mit der Zeit immer weniger als solche wahrgenommen werden.

Abbildung 2-1 ▶
Multitouch-Tisch: Reactable
(Foto: Xavier Sivecas Saiz)

Auch die großen Firmen haben die alternativen Eingabegeräte für sich entdeckt, vom EyeToy von Sony, das Bewegungen des Spielers über eine Kamera verfolgt, über Dance Dance Revolution, das den Spieler zu mehr oder weniger echten Tanzschritten animiert und diese als Grundlage für den Spielverlauf nimmt, bis hin zur Wii, die Kamera und Bewegungssensoren kombiniert. Von Microsoft soll Ende 2010 das Project Natal für die XBOX erscheinen, mit dem über Kameras die Bewegungen der Nutzer in nie dagewesener Präzision verarbeitet werden können. Allen gemeinsam ist der Trend, sich von althergebrachten Konzepten zu lösen und neue Wege der Benutzerinteraktion zu beschreiten.

Physical Computing harmoniert wunderbar mit Forschung und Lehre: Studenten wird ein möglichst einfacher und vor allem interessanter Einstieg in die Technik gegeben. Die Studenten wiederum lassen sich von den schier unbegrenzten Möglichkeiten inspirieren und entwickeln in Seminaren oder als Abschlussarbeiten die erstaunlichsten Projekte. Waren die Programmierung von Computern und das Arbeiten mit elektrischen Bauteilen über Jahrzehnte eine Domäne der Fakultäten für Informatik, Physik und Elektrotechnik, ist es nun auch Kunst- und Designhochschulen möglich, ihre Kreativität in eine weitere Dimension voranzutreiben. Interdisziplinäre Gruppen rund um den Globus entwickeln dabei Technologien, die schon bald auch in unseren Alltag übergehen werden, wobei sie die Industrie oft vor sich hertreiben.

Zunächst benötigt man für einen Einstieg in das Physical Computing jedoch einige Grundlagen, die im Folgenden erklärt werden. Dabei geht es zunächst um die Physik von Schaltungen und Stromkreisen, die Vorstellung häufig gebrauchter Bauteile und Werkzeuge sowie die Einführung ins Basteln, Löten und in die Fehlersuche. Im Anschluss folgt ein kurzer Überblick über die Arduino-Programmiersprache und die dazugehörige Software.

Elektrische Grundlagen

Der Kern aller Physical-Computing-Projekte ist der elektrische Schaltkreis. Alle Eingangs- und Ausgangsschnittstellen kommunizieren über elektrische Signale. Diese Signale übermitteln sowohl Information, indem diese in elektrische Werte codiert wird, als auch Energie, um z.B. Licht oder mechanische Kraft zu erzeugen (um Gegenstände zu bewegen oder Klänge zu generieren). Ein elektrischer Schaltkreis (auch Stromkreis genannt) besteht aus Bauteilen, die eine gewisse Funktion erfüllen und mit diesen elektrischen Signalen umgehen, sowie aus Verbindungen, die diese einzelnen Bauteile miteinander verknüpfen.

Elektrische Größen: Strom, Spannung, Widerstand

In einem Stromkreis gibt es drei wichtige Größen, die im Folgenden kurz näher beschrieben werden.

Die *Spannung*, gemessen in *Volt*, ist der relative Unterschied der Energie zwischen zwei Punkten. Vergleicht man die Elektrizität mit Wasser, dann entspricht die Spannung dem Wasserdruck in einer Leitung. Nur wenn der Wasserdruck an einer Stelle höher ist als an der anderen, fließt es auch durch eine Verbindung der beiden Stellen. Zum Beispiel fließt von einem Berg (hohe Spannung) das Wasser von oben nach unten. Das Gleiche passiert bei der Elektrizität: Auch hier benötigen wir einen Spannungsunterschied, damit ein Strom fließen kann. In einer Schaltung gibt es meistens einen Punkt (*GND* oder *ERDE* genannt), der als Referenz für alle Spannungsmessungen gilt. Bei einfachen Schaltungen verwendet man den tiefstmöglichen, also null, für diesen Zweck, sodass alle Spannungen in der Schaltung positiv sind. Wichtig ist, sich bei Spannungsangaben immer zu merken, dass damit eine Differenz zwischen zwei Punkten gemeint ist.

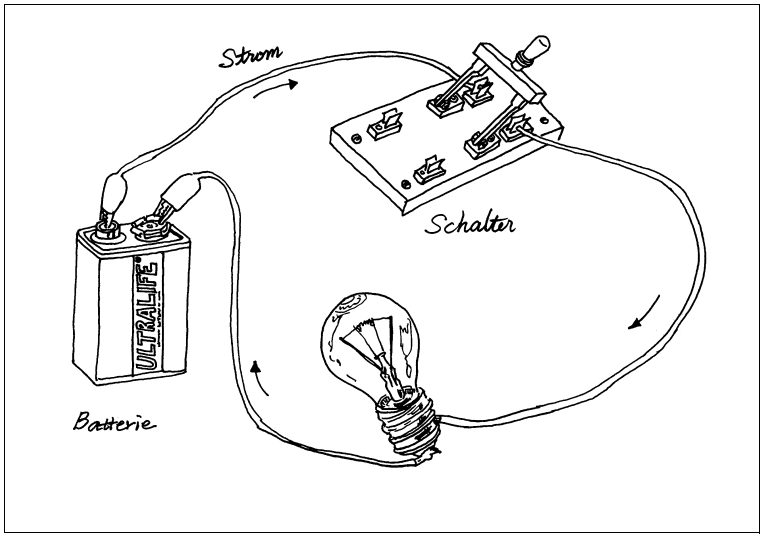
Strom ist die Menge der elektrischen Energie, die durch eine Verbindung in der Schaltung fließt, und wird in *Ampere* gemessen. Elektrischer Strom ist eigentlich die Bewegung von Abermillionen kleiner Partikel, die eine elektrische Ladung tragen: den Elektronen. Diese fließen von einem Punkt mit hoher elektrischer Energie (also hohem elektrischen Potenzial) zu einem Punkt mit niedriger. Im Wasserbeispiel entspricht der Strom der Menge des Wassers, die innerhalb einer konstanten Zeit durch eine Röhre fließt. Damit Strom von einer Stelle in einer Schaltung zu einer anderen Stelle fließen kann, müssen diese beiden Stellen verbunden werden, zum Beispiel mit einem Draht oder auch einem Bauteil; eine Lampe stellt auch eine Verbindung dar, durch die der Strom fließen kann.

Alle Teile einer Schaltung, durch die Strom fließt, stellen für den Strom unterschiedlich große Hindernisse dar: den *Widerstand*, gemessen in *Ohm*. Durch diese Strombegrenzung wird elektrische Energie umgewandelt. Ein einfacher Widerstand z.B. wandelt die Energie, die an ihm abfällt, einfach in Wärme um, während andere Bauteile Energie etwa in Bewegung (wie ein Motor) oder in Licht (LEDs und Lampen) umsetzen können. Digitale Bauteile benutzen diese Energie, um Berechnungen auszuführen. Drähte und Verbindungen auf einer Platine haben einen sehr geringen Widerstand, sie begrenzen den Stromfluss fast nicht. Schließt man zwei Punkte mit unterschiedlichen elektrischen Potenzialen über einen Draht zusammen, wird nur wenig Widerstand in den Weg gestellt, es fließt ziemlich viel Strom und es kommt zu einem Kurzschluss. Im besten Fall greifen dann Schutzmechanismen: Beim Arduino-Board, das über USB seinen Strom bekommt, wird der Strom dann vom angeschlossenen Rechner gekappt. In einer Wohnung springen dann meist die Sicherungen heraus. Es ist allerdings auch möglich, mit Kurzschlüssen Bauteile zu beschädigen oder sich selbst zu gefährden, weshalb man beim Zusammenbauen von elektrischen Schaltungen immer vorsichtig vorgehen sollte. Man sollte lieber ein zweites Mal überprüfen, ob auch keine Kurzschlüsse erzeugt werden.

Ein sehr einfacher Schaltkreis wäre z.B. eine Glühbirne, die an eine Stromquelle angeschlossen ist. Um die Lampe ein- und ausschalten zu können, wird noch ein Schalter in den Schaltkreis eingebaut. Dieser ist ein Bauteil, das eine elektrische Verbindung erzeugen und trennen kann: Ist er offen, besteht keine elektrische Verbindung, und es kann auch kein Strom durch den Schalter fließen; ist er geschlossen, verhält er sich wie ein Draht, und der Strom fließt hindurch. Die Lampe verhält sich wie ein Widerstand, der den Strom-

fluss begrenzt und die Energie des Stroms in Licht und Hitze umwandelt.

Jedes Bauteil im Schaltkreis braucht eine gewisse Menge Strom, um seine Aufgabe zu erledigen. Bekommt die Glühbirne nicht genug davon, wird der Draht, der das Licht erzeugt, nicht heiß genug, und die Birne leuchtet nicht. Bekommt sie aber zu viel Strom, brennt der Draht durch und die Glühbirne ist danach defekt. Die Menge Strom, die durch ein Bauteil fließt, und die Spannung, die an ihm anliegt, müssen also genau eingestellt werden.



◀ Abbildung 2-2
Anschließen einer Lampe

Ohmsches Gesetz

Um Strom und Spannung in einem Schaltkreis zu berechnen, lassen sich folgende Formeln einsetzen, die die drei elektrische Größen verbinden. Diese Formeln sind auch die am häufigsten gebrauchten, um Schaltungen zu entwerfen, und nach einer gewissen Zeit entwickelt man auch ein Bauchgefühl, wie ein Schaltkreis zu entwerfen ist. Sie werden auch Ohmsches Gesetz genannt und lauten wie folgt:

U (Spannung) = R (Widerstand) × I (Strom)

oder wenn man das Ganze umformt:

I (Strom) = U (Spannung) / R (Widerstand)

Anschaulich lassen sich diese Formeln wie folgt beschreiben: Je höher die Spannung zwischen zwei Punkten, desto mehr Strom

wird zwischen ihnen fließen, wenn sie zusammengeschlossen werden. Je größer der Widerstand zwischen diesen zwei Punkten, desto weniger Strom wird fließen. Wird also eine Spannung an ein Bauteil angeschlossen, fließt genau so viel, wie der Widerstand des Bauteils zulässt.

Als Beispiel wird ein Bauteil (Glühbirne) mit einem Widerstand von 15 Ohm verwendet (Im Zweifelsfall kann der genaue Widerstand mit einem Messgerät ermittelt werden) und es an eine Spannungsquelle von 5 Volt angeschlossen. Die URI-Formel ergibt:

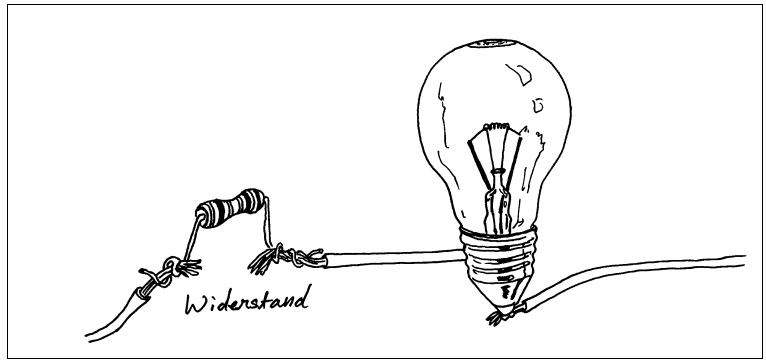
$$I = 5 \text{ Volt} / 15 \text{ Ohm} \rightarrow I = 0{,}33\text{A}$$

Bevor man das tatsächlich anschließt, sollte man auf der Glühbirne nachsehen, ob sie einen Strom von 0,33A verträgt. Bei 5V entspricht das $P = 5 \text{ Volt} \times 0{,}33\text{A} \rightarrow 1{,}65$ Watt

Befinden sich in einer Schaltung mehrere Bauteile hintereinander, so wird ihr Widerstand einfach addiert. Der Strom wird bei jedem Bauteil ein bisschen stärker eingeschränkt. So lässt sich auch der Strom einstellen, der durch eine Glühbirne fließt; man begrenzt ihn, damit der Draht der Glühbirne nicht durchbrennt. Bei Bauteilen, die hintereinander angeschlossen werden, sagt man, dass sie *in Serie* geschaltet sind. Der Gesamtwiderstand der in der Abbildung gezeigten Schaltung ist also

R (Gesamt) = R (Widerstand) + R (Glühbirne)

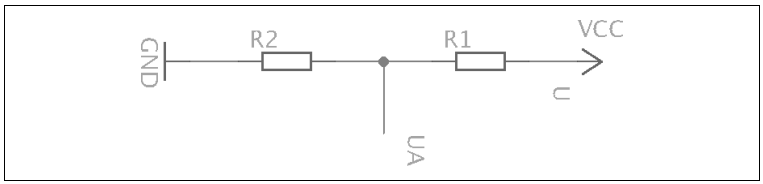
Abbildung 2-3 ▶
Widerstand und Glühbirne in Serie



Eine wichtige Anwendung von in Serie geschalteten Widerständen ist eine Schaltung, die man *Spannungsteiler* nennt. Werden zwei Widerstände mit den Werten R1 und R2 in Serie geschaltet und an ihren Enden eine definierte Spannung U angelegt, so fließt durch beide derselbe Strom, der durch I = U / (R1 + R2) gegeben ist.

Dadurch fällt an jedem Widerstand also eine Spannung ab, die proportional zu seinem Widerstandswert ist. Die Spannung an der Stelle A im Bild ist also UA = U × (R2 / (R1 + R2)). Diese Schaltung wird sehr oft benutzt, um zum Beispiel definierte Referenzspannungen zu erzeugen, eine Spannung zu skalieren oder um resistive Sensoren, also jene, die sich wie ein veränderbarer Widerstand verhalten, auszulesen.

Ein Spannungsteiler aus zwei gleich großen Widerständen teilt also eine Spannung genau in der Mitte, aus 5 Volt werden 2,5V. Allerdings nur so lange, wie kein oder fast kein Strom von dieser Mitte aus woanders hinfließt.



◀ **Abbildung 2-4**
Spannungsteiler

Bauteile können auch parallel angeschlossen werden, man spricht dann davon, dass sie *parallel geschaltet* sind. In diesem Fall teilt sich der Strom in mehrere Wege auf und fließt abhängig vom ihren einzelnen Widerständen.

Leistung

Eine weitere wichtige Größe ist die *Leistung*, die in *Watt* gemessen wird. Die Leistung gibt an, wie viel Energie ein Bauteil verbraucht, wenn es in einem Schaltkreis angeschlossen ist und Strom durch ihn fließt. Die Leistung, die ein Bauteil benötigt, kann berechnet werden, indem man die Spannung an seinen Anschlüssen mit dem fließenden Strom multipliziert, der durch ihn fließt. Dadurch lässt sich auch berechnen, wie viel Strom und Spannung notwendig sind, damit ein Bauteil seine Aufgabe erledigen kann. So braucht eine 60-Watt-Glühbirne bei einer Spannung von 12V also 5A Strom, damit sie mit ihrer vollen Helligkeit leuchtet.

Die Leistung ist auch wichtig, um zu wissen, wie viel Strom und Spannung ein Bauteil verträgt, bevor es durchbrennt. Ein handelsüblicher Widerstand verträgt zum Beispiel 0.125 bis 0.25 Watt. Da Energie niemals ohne Weiteres vernichtet werden kann (obwohl es einem oft so vorkommt), muss der Widerstand diese Leistung als Wärme wieder abgeben, er wird also heiß. Kann die Wärme nicht

schnell genug an die Umgebung (Luft oder Kühlkörper) abgegeben werden, wird der Widerstand so heiß, dass er zerstört wird. Dabei können sogar Gegenstände in seiner Umgebung in Brand geraten.

Sicherheitsregeln

Für Menschen sind Ströme von mehr als 5 bis 10 mA sehr gefährlich. Für Spannungen kleiner als 48 Volt kann im Normalfall davon ausgegangen werden, dass nichts passieren kann, da der Widerstand eines Menschen zu groß ist, um bei einer solchen Spannung so viel Strom fließen zu lassen.

Die auf dem Arduino-Board verwendeten Spannungen von 5 und 3 Volt stellen keine Gefahr für den Menschen dar. Trotzdem sollten direkte Verbindungen zum Menschen (Berührung, EEG usw.) mindestens mit einem 100kOhm-Widerstand in Reihe abgesichert werden, wenn das Arduino-Board nicht aus einer Batterie versorgt wird. Niemand kann garantieren, dass das Netzteil (z.B. im angeschlossenen PC) bei einem Defekt sicher bleibt. So können im Ernstfall plötzlich 230 Volt aus der Steckdose an der Schaltung anliegen.

Für Experimente mit höheren Spannungen sollte unbedingt ein Experte herangezogen werden. Mindestens aber sollte immer eine *Fehlerstromsicherung* verwendet werden. Diese Sicherungen sind als Zwischenstecker in Baumärkten erhältlich und trennen die Verbindung, sobald ein Strom von mehr als 10 bis 30mA über die Hand fließt.

Ist es unumgänglich, an einer unter Spannung stehenden Schaltung zu arbeiten, sollte eine Hand immer hinter den Rücken gehalten werden: So wird ein gleichzeitiges Berühren mit beiden Händen verhindert und somit ein Stromfluss direkt durch das Herz vermieden. Schlimme Unfälle passieren häufig dadurch, dass eine Hand am geerdeten Metallgehäuse ist, während die andere ein Strom führendes Teil berührt.

Analoge und digitale Schaltungen

Bei elektrischen Schaltungen wird oft zwischen rein elektrischen Schaltungen und elektronischen Schaltungen unterschieden. Als elektrische Schaltungen bezeichnet man im allgemeinen Schaltungen, die ohne Halbleiter auskommen (Transistor, IC). Eine Lampe mit einem Schalter ist zum Beispiel eine elektrische Schaltung.

In elektronischen Schaltungen werden oft Spannungen und Ströme verwendet, um Informationen zu übermitteln, ohne dass man mit dieser Energie direkt etwas antreiben möchte. Deshalb können diese Ströme sehr viel kleiner sein. So kann zum Beispiel ein Audiosignal als Spannung kodiert und in einer elektronischen Schaltung verändert werden (z.B. um die Anteile an Höhen und Bässen einzustellen, oder die Lautstärke). Erst im letzten Schritt wird dieses Spannungssignal dann so verstärkt, dass damit große Lautsprecher angesteuert werden können. Zur Verstärkung und Manipulation dieser Signale werden Transistoren eingesetzt. Für viele Aufgaben gibt es integrierte Schaltungen (ICs, integrated circuits), die diese Leistungstransistoren ersetzen. Diese werden meistens Treiber genannt. Diese integrierten Schaltungen können mehrere Millionen einzelne diskrete Bauteile enthalten (wie z.B. Transistoren) und werden in kleinen Plastikgehäusen mit Anschlusspins geliefert. Diese Bauteile nennt man Chips, und es gibt sie von sehr einfachen Varianten, die nur eine Handvoll Transistoren enthalten, über komplizierte mikromechanische Sensoren (wie z.B. den Beschleunigungssensor) bis hin zu großen Prozessoren wie z.B. einem Pentium-Prozessor.

In elektronischen Schaltungen werden oft Spannungen eingesetzt, die nur zwei Werte annehmen können: hohe Spannung (auch HIGH-Spannung) und niedrige (auch LOW-Spannung). Diese zwei unterschiedlichen Spannungen werden verwendet, um sogenannte Bits zu codieren: 0 oder 1. Man spricht auch von *diskreten Spannungswerten*, weil es sich um zwei klar getrennte und definierte Spannungen handelt, im Unterschied zu kontinuierlichen Spannungswerten, die über den gesamten Bereich von GND bis VCC gehen. Diese Codierung, d.h. Darstellung von Information in Spannung, in diskrete Werte nennt man auch *digitale Codierung* (und die verwendeten Spannungen werden digitale Spannungen genannt), im Vergleich zur *analogen Codierung* (wie sie z.B. für Audiosignale in einem Verstärker verwendet wird). Der größte Vorteil der digitalen Codierung ist, dass sich mit ihr Informationen »exakt« übermitteln lassen. Eine analoge Spannung ist immer mit Rauschen behaftet, das etwa durch die Umgebung erzeugt wird. Je mehr eine analoge Spannung bearbeitet und übertragen wird, desto stärker ist dieses Rauschen. Das lässt sich am besten mit analogen Audioaufnahmen verdeutlichen: Eine Audiokassette oder eine Schallplatte speichert analoge Informationen und wird mit der Zeit und der Abnutzung immer schlechter. Je länger das benutzte Kabel ist, das zum Anschluss des Abspielgeräts dient, desto schlechter

wird auch der Klang. Im Unterschied zu analogen Signalen lassen sich digitale Signale wieder ohne Verlust verstärken. Wenn also digitale Signale über lange Kabel gesendet werden, kann man sie regelmäßig einlesen und wieder ausgeben. So ist es jetzt möglich, über das Internet Leuten, die auf der anderen Seite des Globus wohnen, ohne Verlust Fotos, Audioaufnahmen und Filme zu senden. In einem deutlich kleinerem Maßstab wird diese Eigenschaft in Chips benutzt, um Millionen von Transistoren zu verbinden, ohne dass die ausgetauschten Spannungen am Ende vollends mit Rauschen behaftet sind.

Die digitale Elektronik bildet die Grundlage für das »Rechnen« im Physical Computing. Bei digitalen Signalen wird meistens kaum Leistung übertragen (ein bisschen Strom wird immer noch in Wärme umgewandelt, weswegen moderne Prozessoren auch sehr schnell warm werden), sondern die Signale dienen rein der Kommunikation. Viele Sensoren und Aktoren bieten, obwohl sie intern mit analogen Signalen und Werten arbeiten, eine digitale Schnittstelle an, sodass sie sehr einfach anzuschließen sind. Der Königsbaustein unter den digitalen Bauteilen ist der sogenannte Prozessor. Er ist ein frei programmierbarer Computer, der sich in Schaltkreise einbauen lässt. Hier wird zwischen Prozessor und Mikrocontroller unterschieden. Erstere findet man etwa in Desktopcomputern und Servern. Sie haben keine direkte Anbindung an die Umwelt, das ist die Aufgabe weiterer Bausteine auf dem Mainboard. Bei einem Mikrocontroller sind viele Schnittstellen (RS232, USB, Analog In, PWM, IO-Ports) schon eingebaut, oft ist auch der Speicher direkt integriert. Der Arduino-Prozessor hat eine interne Datenbreite von 8 Bit und ein paar Kilobyte Arbeitsspeicher. Er rechnet mit einem Takt von 16 MHz.

Zusammenfassung der elektrischen Regeln

Diese ganzen Stromregeln lassen sich wie folgt zusammenfassen:

- Zwischen zwei Punkten in einem Schaltkreis liegt eine Spannung an.
- Zwischen zwei elektrisch verbundenen Punkten fließt ein Strom.
- Der Fluss der Stroms hängt von der anliegenden Spannung und dem Widerstand ab.

- Seriell geschaltete Bauteile addieren ihren Widerstand.
- Parallel geschaltete Bauteile teilen den Strom entsprechend ihrem Widerstand auf.

Es reicht meistens aus, diese Regeln zu benutzen, um die Strom- und Spannungswerte für ein neues Bauteil zu berechnen. Bei schon einmal verwendeten Bauteilen lassen sich meistens ältere Schaltkreise und Einstellungen wiederverwenden, sodass sich nach einiger Zeit eine Art Routine einstellt. Bei einem neuen Bauteil muss im dazugehörigen Datenblatt, das die elektrischen Eigenschaften des Bauteils beschreibt, nachgeprüft werden, wie hoch die minimalen und maximalen Strom-, Spannungs- und Leistungswerte sind. Diese sind oft in einer Tabelle zusammengefasst.

Tipp

Es gibt im Internet viele hilfreiche Websites, die die Grundlagen der Elektronik erklären. Besonders hilfreich sind z. B. die Webseiten *http://www.elektronik-kompendium.de/* und *http://mikrokontroller.net/*.

Schaltungen, Bauteile und Schaltbilder

Schaltungen bestehen aus einzelnen elektrischen bzw. elektronischen Bauteilen, die miteinander über elektrische Leiter (Drähte) verknüpft werden. Schon mit dem Anschließen einer Lampe an eine Steckdose wird also eine, zugegebenermaßen recht einfache, elektrische Schaltung hergestellt. Für komplexere Schaltungen verwendet man heutzutage Platinen. Das sind Platten aus Glasfasern und Epoxidharz, auf denen Kupferleiterbahnen dafür sorgen, dass die Bauteile befestigt und verbunden werden. Das Mainboard eines Computers ist ein gutes Beispiel für eine solche Platine, wenn auch schon eine sehr große und komplexe.

Platinen sind nicht viel schwieriger zu entwerfen als Steckbrettaufbauten (siehe Kapitel 2, Abschnitt »Steckbretter«). Hierfür werden spezielle CAD-Programme eingesetzt. Es kann sinnvoll sein, die Schaltung vorher zu testen, da es schwierig ist, eine fertige Platine zu modifizieren. Deswegen werden beim Entwurf und für den Heimgebrauch oft spezielle Platinen eingesetzt, die Lochrasterplatinen und Steckbretter genannt werden. Auf diesen Platinen ist es leicht möglich, schnell Schaltungen aufzubauen und auch nachträglich zu verändern.

Bauteile

Bauteile sind die einzelnen physikalischen Komponenten in einer elektronischen Schaltung. Sie können zum einen rein elektrische Bauteile sein, die bestimmte Eigenschaften haben, aber kein eigenes variables Verhalten an den Tag legen. Diese Bauteile nennt man *passive Komponenten*, und zu ihnen gehören unter anderem Widerstände, Kondensatoren und Spulen. Zum anderen gibt es Bauteile, die ihre Eigenschaften mechanisch ändern können. So kann zum Beispiel ein elektrischer Kontakt mechanisch erstellt oder unterbrochen werden, wie es bei einem Schalter der Fall ist. Es ist auch möglich, wie bei einem Potentiometer mechanisch den Widerstand des Bauteils einzustellen. Andere Bauteile hingegen benutzen die elektrischen Eigenschaften von Halbleitermaterialen, um Signale elektronisch zu verarbeiten. Diese Halbleiterbauteile reichen von der einfachen Diode und dem LED über Transistoren bis hin zu Prozessoren in Desktopcomputern. Viele Halbleiterbauteile sind eigene elektronische Schaltungen, die (meistens) in Silizium realisiert und als ICs verpackt sind.

Jedes Bauteil kann es in einer Vielzahl von verschiedenen Versionen und Bauformen zu kaufen geben. Es gibt bei Bauformen zwei Hauptfamilien: bedrahtete Bauformen und SMD-Bauformen. Bedrahtete Bauteile haben, wie ihr Name schon sagt, Drähte zum Anschließen an weitere Komponenten. Dadurch sind sie recht einfach von Hand zusammenzustecken oder zusammenzulöten, weswegen sie für den Heimgebrauch besonders gut geeignet sind. In diesem Buch werden nur bedrahtete Bauteile verwendet. Die andere, mittlerweile extrem verbreitete Bauform ist die SMD-Bauform (surface mounted devices, was frei übersetzt so viel heißt wie »Oberflächenbauteile«). Diese Bauteile haben sehr kurze Anschlusdrähte, die direkt auf die Platine gelötet werden. Die SMD-Bauform ist für die industrielle Fertigung bestimmt, und solche Bauteile lassen sich nicht angenehm von Hand zusammenschließen. Oft werden Sensoren oder digitale Bausteine allerdings nur in SMD-Bauform vertrieben. Um sie trotzdem einfach mit dem Arduino-Board und anderen Schaltungen verbinden zu können, gibt es für Bastler kleine Adapterplatinen, die SMD-Bausteine mit längeren Drähten verbinden.

Nicht alle Bauteile des gleichen Typs verhalten sich auch gleich, besonders wenn es sich um komplexere Sensoren oder Aktoren handelt. In den sogenannten *Datenblättern* sind alle physikalischen

Eigenschaften und sonstige Informationen zusammengefasst, die für eine korrekte und vollständige Verwendung des entsprechenden Bauteils nötig sind. Im Internet gibt es eine ganze Reihe von Datenblattsammlungen, die die meisten gängigen Bauteile umfassen. Die Websites *http://www.alldatasheet.com* und *http://www.datasheetcatalog.net/de/* stellen tausende von PDFs zur Verfügung, die von den Herstellern dort hochgeladen wurden.

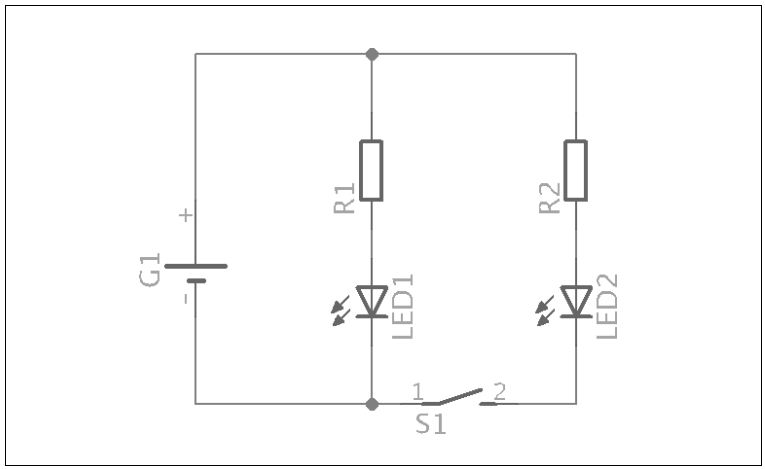
Diese unterschiedlichen Bauteile kann man bei vielen Distributoren und Elektronikläden erwerben. Das Suchen und Bestellen von elektronischen Komponenten kann schnell zu einem eigenen Sport werden: Meistens gibt es nicht nur eine Version von einem Bauteil, sondern gleich 15, die ähnliche Eigenschaften haben, sich aber leicht voneinander unterscheiden. Hier lohnt es sich oft, bei verschiedenen Läden und Distributoren nachzugucken, ob bestimmte Bauteile vielleicht billiger zu erhalten sind, oder in einer sinnvolleren Bauform oder mit besseren Eigenschaften. Zum Glück gibt es mittlerweile, gerade für Arduino-Anfänger, Läden, die sich auf den Hobbybastelbereich konzentrieren, wie z.B. *http://bausteln.de/*, *http://elmicro.com/* und *http://tinkersoup.de/*. Dort lassen sich auch viele verschiedene Arduino-Boards und -Shields (siehe Anhang A), Adapterplatinen, interessante Sensoren und sogar fertige Projekte zum Selbstbauen (sogenannte *Kits*, in denen alle Bauteile für ein Projekt schon zusammengelegt sind) erwerben.

Schaltbilder

Schaltbilder sind eine Art »Sprache«, um Informationen über elektronische Schaltungen auszutauschen. Sie werden verwendet, um Schaltungen in abstrahierter und vereinfachter Form grafisch darzustellen. So können Entwickler diese entwerfen und planen, ohne eventuelle physikalische Notwendigkeiten wie die eigentliche Kabelführung zu berücksichtigen.. Mit ihnen lassen sich vage bestimmte Schaltungen definieren, es ist aber auch möglich, genau jedes Bauteil zu spezifizieren. Viele elektronische Projekte im Internet werden auf diese Weise erklärt. Im unteren Schaltbild wird zum Beispiel beschrieben, wie zwei LEDs über ihren jeweiligen Vorwiderstand an eine Batterie anzuschließen sind. Die zweite LED ist über einen Schalter getrennt aktivierbar (siehe Abbildung 2-5).

Schaltbilder sind Diagramme, die zeigen, wie einzelne Bauteile miteinander verknüpft werden. Jedes Bauteil wird durch ein Symbol dargestellt. Ihre Darstellung variiert allerdings von Schaltbild zu

Abbildung 2-5 ▶
LED-Schaltbild



Schaltbild (in den USA werden zum Beispiel Widerstände als Zickzack-Linien dargestellt, in Europa dagegen als kleine Kästchen). Viele Bastler benutzen auch ihre eigenen Symbole um bestimmte Bauteile zu zeichnen. Jedes dieser Symbole wird dann mit Strichen mit anderen verbunden, und jeder dieser Striche steht für eine elektrische Verbindung. Die Bauteile werden meistens noch mit Namen und Werten beschrieben (z.B. den Widerstandswerten einzelner Widerstände, den Herstellernamen und den genauen Typbezeichnungen). Schaltbilder (oder Schaltpläne) können mit einer Reihe von Programmen erstellt und bearbeitet werden (es ist natürlich auch möglich, Schaltbilder auf eine Papierserviette zu zeichnen) und greifen dabei auf einen Satz von meist genormten Symbolen für die einzelnen Bestandteile zurück.

Tipp

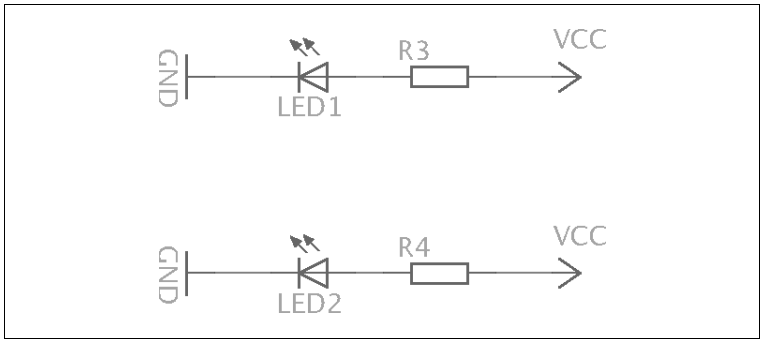
Für den Heimbereich bieten sich zum Zeichnen von Schaltbildern Programme an wie gEDA (eine Open-Source-Lösung, die aus vielen einzelnen kleinen Programmen besteht), KiCad (das auch frei ist) oder EAGLE (das für den Heimbereich frei ist). Es reicht meistens allerdings, die Schaltung auf einem einfachen Blatt Papier zu entwerfen.

GEDA: *http://www.gpleda.org/*
KiCad: *http://www.lis.inpg.fr/realise_au_lis/kicad/*
EAGLE: *http://www.cadsoft.de/*

Es kommt häufig vor, dass in einem Schaltbild Verbindungen gezeichnet werden, die sich überkreuzen. Deswegen werden diese, wenn sie auch wirklich zusammengeschaltet sind, mit einem kleinen Punkt gekennzeichnet, um zu verdeutlichen, dass es keine

getrennten Verbindungen sind. Weiterhin werden oft benutzte Spannungen in der Schaltung (meistens jene, die von der Stromversorgung zur Verfügung gestellt werden) mit einem einzelnen Symbol gezeichnet, sodass es nicht notwendig ist, jeden Baustein immer mit der Stromversorgung zu verbinden. Die Masse (also die Spannung 0 Volt) wird meistens mit einem umgekehrten T-Symbol dargestellt, während die Spannungsversorgung (bei Arduino-Schaltungen meistens 5 Volt) mit einem Pfeil nach oben dargestellt wird. Wie so oft sind diese Symbole aber von Schaltbild zu Schaltbild unterschiedlich, und es ist wichtig, hier nicht Spannungen zu verwechseln. Bei ICs wird die Spannungsversorgung oft gar nicht erst mit eingezeichnet, sondern es wird davon ausgegangen, dass dem IC auch Strom zugeführt wird.

In der unteren Abbildung ist diese Schaltung mit zwei LEDs und Vorwiderständen dieses Mal mit eigenen Symbolen für die Spannungsversorgung gezeichnet.



◀ **Abbildung 2-6**
Schaltbild mit Spannungsversorgungssymbolen

Nachdem eine Schaltung entworfen wurde, ist es Zeit, sie mit richtigen Bauteilen zu realisieren. Das Schaltbild ist also sozusagen ein Bauplan. Oft ist es nicht möglich, die Bauteile so anzuordnen, wie sie auf dem Schaltbild vorkommen. Bei Platinen werden dazu oft mehrere Schichten Kupfer benutzt, bei Steckbrettschaltungen werden für die Verbindungen sowohl die vertikalen und horizontalen Lochreihen des Steckbretts als auch einzelne gesteckte Drähte verwendet. Die Bauteile müssen nicht so angeordnet werden wie auf dem Schaltbild. Es ist allerdings sinnvoll, auch auf dem Steckbrett oder der Platine Bauteile, die in einem Zusammenhang miteinander stehen, nah beieinander aufzubauen. Wichtig ist, dass jede elektrische Verbindung, die auf dem Schaltbild steht, auch im physikalischen Aufbau hergestellt wird.

Schalter

Schalter sind mechanische Bauteile, die eine elektrische Verbindung herstellen oder trennen können. Meistens werden durch die physische Betätigung des Schalters zwei elektrische Leiter miteinander verbunden. Es gibt allerdings auch magnetische Schalter, bei denen die elektrischen Leiter durch einen Magneten aneinandergezogen und so verbunden werden. Interessant sind auch sogenannte »Ball Switches«, die den Kontakt über einen kleinen metallischen Ball herstellen, der auf die elektrischen Leiter rollt, wenn der Schalter gekippt wird. In Kapitel 7 wird ein kleiner Erschütterungssensor beschrieben, der auch als Schalter funktioniert: Ein kleiner Drahtring liegt lose auf zwei Kontakten und öffnet bei Erschütterungen kurz den Stromkreis.

Schalter vertragen eine bestimmte anliegende Spannung und eine bestimmte Menge Strom, der durch sie fließt. Bei den elektronischen Schaltungen in diesem Buch werden sehr kleine Spannungen und Ströme benutzt, sodass es hier bei den einsetzbaren Schaltern keine Beschränkungen gibt. Deswegen ist es auch möglich, aus allen möglichen elektrischen Leitern (z.B. Draht und Aluminiumfolie) auch eigene interessante und innovative Schalter zu entwerfen.

Es gibt eine Unmenge verschiedener Schalter, die man käuflich erwerben kann. Diese unterscheiden sich natürlich durch ihre mechanische Bauform, also ihre Größe, ihr Aussehen, die Materialien, aus denen sie gebaut sind, und dadurch, wie sie an einem Gehäuse angebracht werden. Für Entwicklungsschaltungen, die z.B. auf einem Steckbrett zusammengebaut werden, ist es meistens nicht so wichtig, welche Schalter eingesetzt werden. Soll eine Schaltung allerdings auf einer Platine gebaut (entweder auf einer richtigen Platine oder auf einer Lochrasterplatine) und später in ein Gehäuse eingebaut werden, ist es wichtig, sich schon im Voraus Gedanken über das fertige Gerät zu machen. Besonders muss dort auf die Größe und Befestigung der Schalter geachtet werden, denn nichts ist frustrierender, als im Nachhinein zu merken, dass alles gar nicht in das vorgesehene Gehäuse passt. Es ist auch durchaus möglich, die Komponenten zuerst rein mechanisch zusammenzusetzen, um zu sehen, ob auch alles seine Richtigkeit hat.

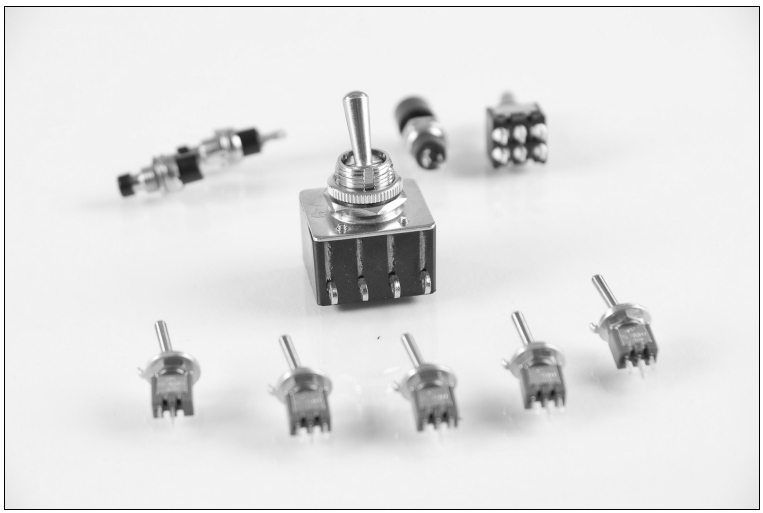
Bei Schaltern gibt es zwei unterschiedliche Funktionsweisen, nämlich Schalter und Taster. Bei einem Schalter wird meistens langfristig eine elektrische Verbindung geschaltet, während bei einem Taster nur ein Kontakt hergestellt wird, solange er betätigt wird.

Wird er wieder losgelassen, wird der Kontakt wieder unterbrochen (solche Taster nennt man »normally open« oder »n. o.«, also »im Normalzustand offen«). Es gibt auch Taster, die ohne Betätigung durch den Benutzer einen Kontakt herstellen und diesen erst beim Betätigen unterbrechen. Diese nennt man »normally closed« oder »n. c.«.

Weiterhin gibt es viele verschiedene Schalter, die sich durch die Anzahl der durch sie einstellbaren Verbindungen unterscheiden. So gibt es einfache, die nur eine elektrische Verbindung trennen oder aufbauen können. Kompliziertere Schalter können aber z.B. parallel zwei, drei oder mehr Verbindungen aufbauen und trennen. Generell werden Schalter durch die Anzahl der verschiedenen herstellbaren Verbindungen (auf Englisch »poles«) und die Anzahl der elektrischen Leitungen, die verbunden werden können (auf Englisch »throws«) beschrieben. In der folgenden Tabelle sehen Sie eine Übersicht herkömmlicher Schaltertypen.

◀ **Tabelle 2-1**
Schaltertypen

Schalterbezeichnung	Beschreibung
SPST (Single Pole, Single Throw)	Dieser einfacher Schalter stellt eine Verbindung her oder trennt sie.
SPDT (Single Pole, Double Throw)	Ein Eingang kann mit zwei unterschiedlichen Ausgängen verbunden werden.
DPST (Double Pole, Single Throw)	Zwei Verbindungen können parallel hergestellt oder getrennt werden.
DPDT (Double Pole, Double Throw)	Hier können zwei Eingänge parallel mit separaten Ausgängen verbunden werden.



◀ **Abbildung 2-7**
Schalter

Widerstand

Ein Widerstand ist ein Bauteil, das elektrische Energie in Hitze umwandelt. Dadurch begrenzt sich der Stromfluss. Der Wert eines Widerstands wird in Ohm angegeben; anhand der Regeln, die in den vorigen Abschnitten vorgestellt wurden, lässt sich dadurch die Strombegrenzung berechnen. Widerstände sind kleine Bauteile, die zwei Anschlüsse besitzen. In welcher Richtung sie eingesetzt werden, spielt keine Rolle, allerdings sehr wohl ihre Größe. Da ein Widerstand Energie in Hitze umsetzt, muss gewährleistet werden, dass er diese Energie auch aushalten kann. Für Schaltungen in diesem Buch, in denen mit kleinen Strömen und kleinen Spannungen gearbeitet wird, reichen meistens herkömmliche Widerstände mit 1/4 oder 1/8 Watt.

Widerstände werden aus verschiedenen Materialien hergestellt. So gibt es etwa Kohleschichtwiderstände und Metallschichtwiderstände. Diese haben unterschiedliche Eigenschaften; Kohleschichtwiderstände sind z. B. billiger, aber nicht so präzise wie Metallschichtwiderstände. Diese inhärente Genauigkeit von Widerstandswerten hat dazu geführt, dass sie in verschiedene Werteklassen eingeteilt werden. Diese sind standardisiert und werden als »E-Reihen« bezeichnet: E-3 ist eine Reihe mit drei Werteklassen von sehr grober Toleranz, E-6 hat sechs, die ein bisschen genauer sind, und so geht es weiter bis zu E-192 für Hochpräzisionswiderstände. Es kann deswegen z. B. unangenehm sein, einen genauen Spannungswert zu berechnen, nur um am Ende herauszufinden, dass man dazu einen Widerstand mit 25,98 Ohm und einen mit 1034 Ohm braucht, den es in dieser Form nicht gibt (oder der nicht bezahlbar ist). In unseren Schaltungen, die meistens mit digitalen Spannungen arbeiten, sind viele Widerstände als Pull-ups oder als Schutzwiderstände eingesetzt, weswegen ihr genauer Wert nicht so wichtig ist.

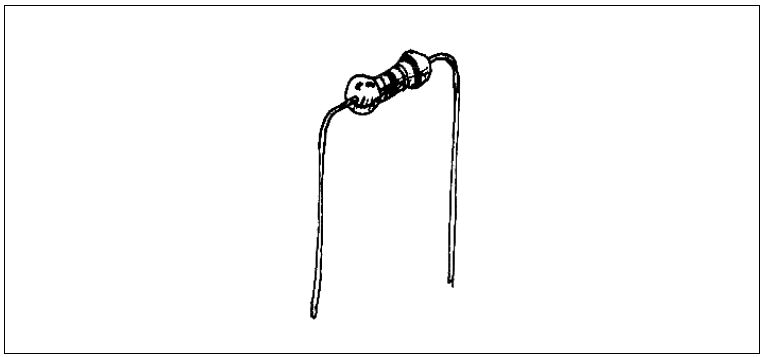
Die Werte auf Widerständen sind durch einen Farbcode angegeben: Verschiedene Streifen auf ihrem Körper geben an, welchen Widerstandswert in Ohm dieses Bauteil hat und mit welcher Genauigkeit zu rechnen ist. In der folgenden Tabelle werden die Farbcodes für Widerstände erläutert. Es gibt zwei verschiedene Farbcodes für Widerstände. Im ersten werden vier Farbbänder benutzt. Die ersten zwei geben den Widerstandswert an, das dritte die Größenordnung (also den Wert, mit dem die zwei ersten Bänder multipliziert werden), während das vierte Band (das meistens

ein bisschen abgesetzt von den anderen an einem Ende des Widerstands zu finden ist), die Toleranz angibt, also beschreibt, wie genau der angegebene Wert zum tatsächlichen passt. Beim zweiten Code werden fünf Farbbänder benutzt. Hier geben die ersten drei den Widerstandswert an, das vierte ist der Multiplikator, und das letzte Band gibt die Toleranz des Widerstands an.

◀ Tabelle 2-2
Widerstandsfarbcodes

Farbe	Erstes Band	Zweites Band	Drittes Band	Multiplikator	Toleranz
Schwarz	0	0	0	1 Ohm	-
Braun	1	1	1	10 Ohm	1,00%
Rot	2	2	2	100 Ohm	2,00%
Orange	3	3	3	1.000 Ohm	
Gelb	4	4	4	10.000 Ohm	
Grün	5	5	5	100.000 Ohm	0,50%
Blau	6	6	6	1.000.000 Ohm	0,25%
Violett	7	7	7	10.000.000 Ohm	0,10%
Grau	8	8	8	-	0,05%
Weiß	9	9	9	-	
Gold				0,1	5,00%
Silber				0,01	10,00%

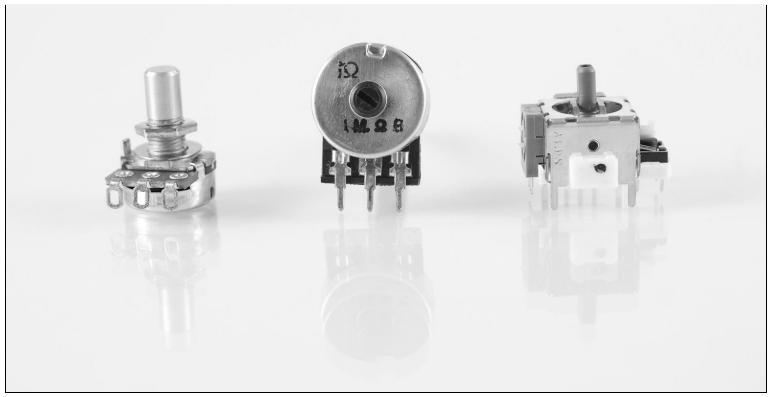
Es ist praktisch, Widerstände in kleine Kästchen oder Schubladen zu sortieren und jeweils mit einem numerischen Wert zu versehen. Dabei muss man aber sicherstellen, dass der angegebene Wert auch tatsächlich dem realen entspricht. Eine andere Methode, um den Widerstandswert eines Bauteils abzulesen, ist die Benutzung eines Multimeters, was im entsprechenden Abschnitt erklärt wird.



◀ Abbildung 2-8
Widerstand

Variabler Widerstand

Abbildung 2-9 ▶
Potentiometer



Ein variabler Widerstand ist, wie sein Name schon sagt, einer, bei dem der Wert eingestellt werden kann. Bekannte Beispiele für variable Widerstände sind Potentiometer und Schieberegler. Ein Potentiometer besteht aus einem speziellen Widerstand, an dem ein sogenannter Schleifer angebracht ist, der sich durch die Drehung entlang der leitenden Oberfläche bewegt. Dadurch vergrößert oder verkleinert er auch den Widerstandswert und somit die Spannung, die am Potentiometer anliegt. Ganz ähnlich wie Drehpotentiometer funktionieren auch Schieberegler (oder »Fader«). Dort ist der Schleifer nicht rund, sondern länglich, sodass der Regler bei einer Bewegung den Widerstand verkürzt. Bekannte Anwendungen sind Lautstärkeregler oder Crossfader auf Mischpulten. Im Arduino-Bereich können Fader z. B. für die Steuerung von Licht oder Ton verwendet werden.

Potentiometer und Schieberegler haben meistens drei Anschlüsse. Die zwei äußeren sind an beiden Enden des zu manipulierenden Materials angebracht, und der Widerstand zwischen ihnen verändert sich nicht. Der dritte Anschluss liegt am Schleifer an und der Widerstand von diesem Anschluss zu den beiden anderen variiert, je nachdem, wie der Schleifer bewegt wird. Wenn an den beiden äußeren Anschlüssen definierte Spannungen angebracht werden, wird ein Spannungsteiler gebildet. Dadurch variiert die Spannung am mittleren Anschluss zwischen den beiden äußeren Spannungen. Dieser Wert lässt sich dann durch einen analogen Eingang am Arduino auslesen.

Es gibt allerdings auch noch eine ganze Reihe anderer Bauteile, die sich wie variable Widerstände verhalten (man nennt diese Bauteile

auch *resistive Sensoren*). So gibt es lichtempfindliche Widerstände, die ihren Wert ändern, je nachdem, wie viel Licht aus sie einfällt. So einen Widerstand nennt man LDR (light dependent resistor) oder auch Fotowiderstand oder Fotozelle. Thermistoren sind Widerstände, die ihren Wert je nach Temperatur ändern. Drucksensoren und Flexsensoren sind Widerstände, die ihren Wert ändern, je nachdem, wie viel Druck auf sie ausgeübt wird oder wie stark sie gebogen werden. Wie Widerstände habe diese Bauteile auch zwei Anschlüsse, und ihre Richtung ist nicht relevant. Oft werden resistive Sensoren mit einem weiteren Widerstand an definierte Spannungen angeschlossen, um so einen Spannungsteiler aufzubauen. Dadurch lässt sich ein resistiver Sensor wie ein Potentiometer auslesen. Mehr Informationen zu resistiven Sensoren finden Sie in Kapitel 7.

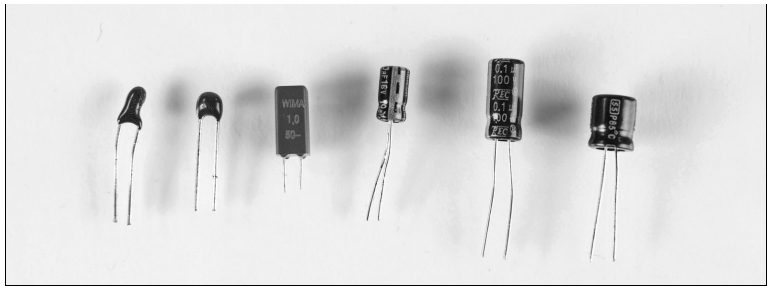
Kondensator

Einen Kondensator kann man als ein Speicherbauteil für elektrische Ladung betrachten. Fließt Strom hinein, speichert dieser intern eine Ladung. Wird die Spannung entfernt, leert sich der Kondensator und liefert Strom, bis er keine Ladung mehr enthält. Dieses Entleeren geschieht allerdings nicht sofort, sondern spielt sich über einen gewissen Zeitraum ab, der von der Kapazität des Kondensators (wie viel elektrische Ladung er aufnehmen kann, gemessen in *Farad*) und von der Menge des fließenden Stroms abhängt. Dadurch können Kondensatoren z.B. eingesetzt werden, um sich schnell verändernde Ströme zu begrenzen und »aufzuweichen«: Fließen kurze Stromstöße, sammelt der Kondensator sie auf, wird der Stromfluss kleiner, braucht er seine gespeicherte Ladung auf, um den Strom aufrechtzuerhalten.

Farad ist eine sehr große Größe, weshalb die Kondensatoren in den Arduino-Schaltungen meistens in Bereiche von Mikrofarads bis Picofarads gehen. Kondensatoren gibt es sowohl in unpolarisierter Bauform, bei der es keinen Einfluss hat, in welcher Richtung sie angeschlossen werden, als auch in polarisierter Bauform. Bei diesen polarisierten Kondensatoren (meistens Elektrolytkondensatoren, kurz auch Elkos genannt) steht auf dem Gehäuse, welcher Anschlusspin an die positive und welcher an die negative Seite angeschlossen werden muss. Bei digitalen Schaltungen werden sie oft eingesetzt, um die Spannungsversorgung für Bausteine zu glätten. Sie können auch benutzt werden, um kurze Impulse, wie sie z.B. aus Piezomikrofonen kommen, zeitlich zu verlängern, sodass

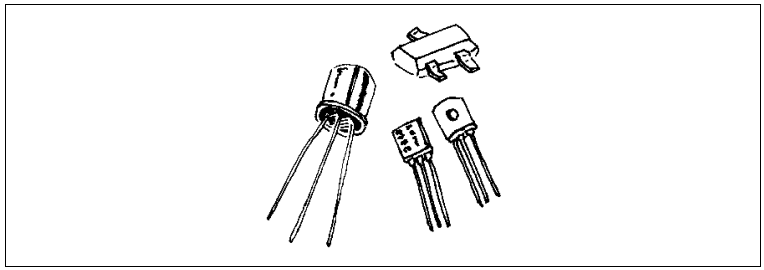
ein Mikrocontroller Zeit hat, diese zu verarbeiten. Ein wichtiger Einsatzbereich von Kondensatoren ist die Implementierung von Filtern, die ein Signal in bestimmten Frequenzen abschwächen oder verstärken. Wir verwenden einen Filter auf Basis eines Kondensators und eines Widerstands (ein sogenanntes RC-Netzwerk) in Kapitel 10, in dem es um Musik mit dem Arduino geht.

Abbildung 2-10 ▶
Kondensatoren



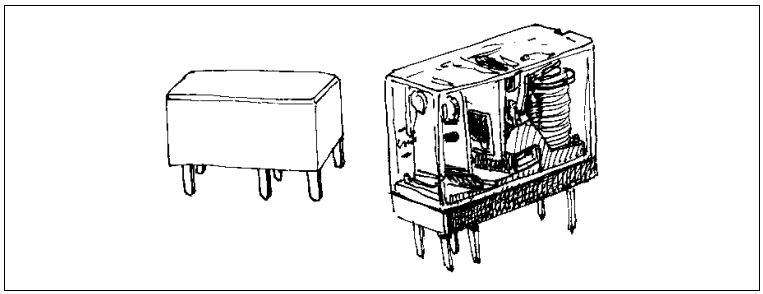
Transistor und Relais

Abbildung 2-11 ▶
Transistoren



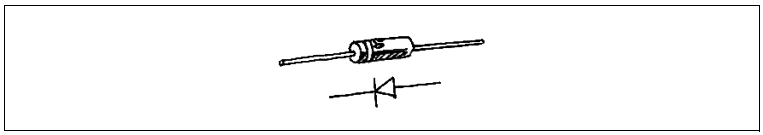
Transistoren und Relais (englisch Relays) sind Bauteile, die zum Schalten von größeren Strömen verwendet werden. Im Gegensatz zu ihren normalen, mechanischen Geschwistern werden Transistoren und Relais allerdings elektronisch geschaltet. Wenn in ein Relais ein kleiner Strom fließt, wird durch eine Spule ein Magnetfeld erzeugt, das einen mechanischen Schalter schließt. Damit lassen sich elektrisch getrennte Netzwerke schalten. Sie werden oft verwendet, um z.B. Bauteile mit Netzspannung zu betreiben. Relais werden genauer im Abschnitt über Aktoren beschrieben.

Im Gegensatz zu Relais benutzen Transistoren die Eigenschaften von Halbleitermaterialien (in den meisten Fälle Silizium), um Ströme zu steuern. Sie werden oft für Bauteile verwendet, die größere Ströme benötigen, etwa Motoren, Glühbirnen oder viele LEDs.



◀ **Abbildung 2-12**
Relais

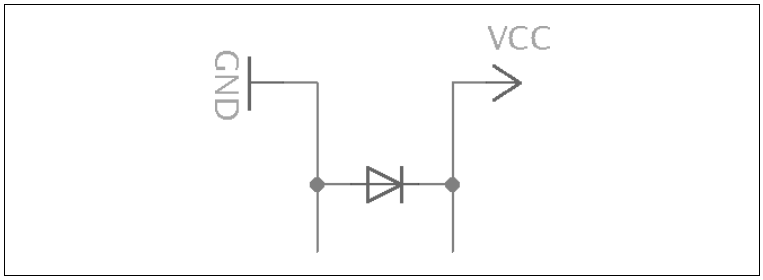
Dioden



◀ **Abbildung 2-13**
Diode

Eine Diode funktioniert wie eine Einbahnstraße: Die Elektrizität kann nur in eine Richtung fließen. Dioden haben zwei Anschlüsse, Anode (+) und Kathode (–). Ein Strom kann nur von der Anode zur Kathode fließen. Deswegen haben Dioden immer eine Richtung; meistens ist die Kathode durch einen Strich auf dem Diodengehäuse gekennzeichnet. Dioden werden verwendet, um Signale zu richten, wie im folgenden Bild gezeigt ist. Sie verhindern, dass z. B. eine negative Spannung anliegt, indem sie diese kurzschließt. Im Arduino-Prozessor sind an jedem Pin auch Schutzdioden angeschlossen, die sicherstellen, dass keine negative oder zu hohe Spannung angelegt werden kann.

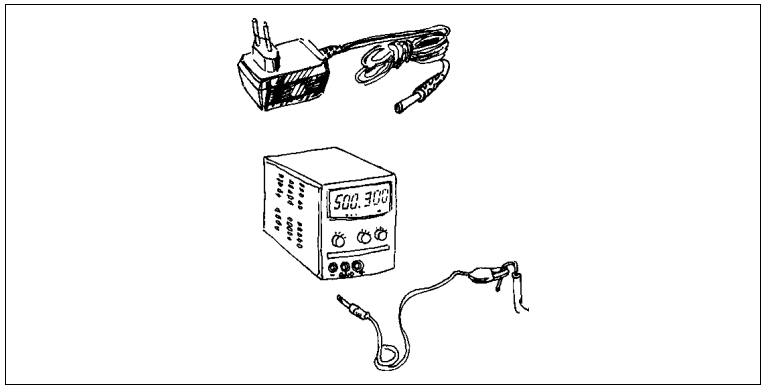
Es gibt verschiedene spezielle Dioden, zum Beispiel Zener-Dioden, die Strom ab einer bestimmten Spannung in der Sperrrichtung durchlassen, sodass sie zur Spannungsbegrenzung und -regelung verwendet werden können.



◀ **Abbildung 2-14**
Invertierungsschutz mit einer Diode

Netzteil

Abbildung 2-15 ▶
Netzteile



Für die meisten Arduino-Projekte wird eine Gleichspannung (DC) benötigt. Dazu können kleine Steckernetzteile oder die USB-Ports verwendet werden. Für kleinere Projekte sind 200 bis 300 mA ausreichend. Werden viele LEDs, Lampen oder Relais eingesetzt, kann der benötigte Strom natürlich viel höher sein, weswegen es ratsam ist, gleich ein Netzteil zu beschaffen, dass 1.000 mA liefern kann.

Vorsichtig sollte man mit älteren Universalnetzteilen sein, deren Regelung meist schlecht oder gar nicht vorhanden ist. Eine eingestellte Spannung kann ohne große Belastung durchaus 3 bis 5 Volt größer als angegeben sein. Dem Arduino-Board macht das allerdings nichts aus, da hier noch ein Spannungsregler verbaut ist, der diese Spannung anpasst.

Eine andere Variante sind die immer gebräuchlicher werdenden Schaltnetzteile, die man an ihrem geringen Gewicht erkennen kann. Sie bieten den Vorteil, ziemlich viel Strom liefern zu können, ohne dabei nennenswert warm zu werden, können allerdings empfindliche Schaltungen stören.

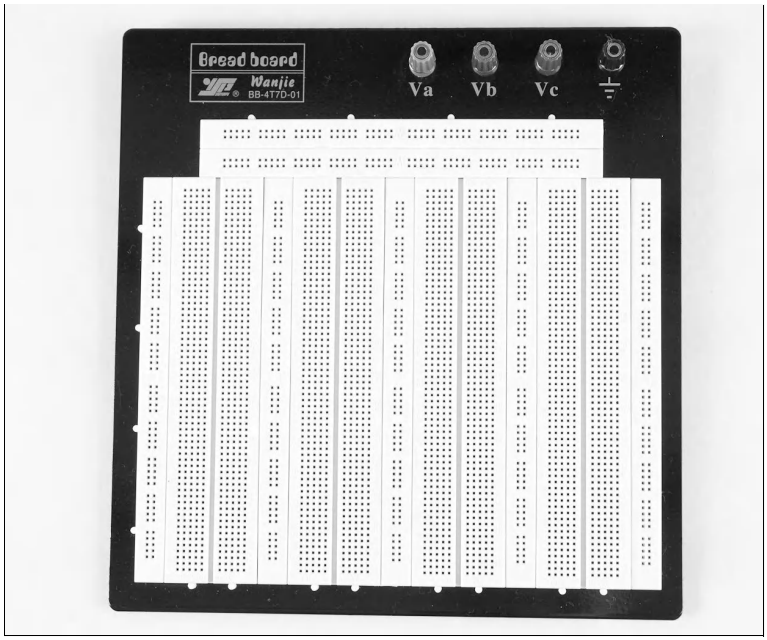
Spannungsregler

Lineare Spannungsregler erzeugen aus der ursprünglichen eine geregelte niedrigere Spannung. Dazu müssen am Eingang 2 bis 3 Volt mehr anliegen, als die Ausgangsspannung sein soll.

Häufig wird der Regler 7805 (5 Volt) verwendet. Er kann aus 7 bis 35 Volt eine geregelte Spannung von 5 Volt erzeugen. Je höher die Eingangsspannung, desto besser muss die Kühlung sein; hier wird empfohlen, ein kleines Stück Blech an den Regler zu schrauben.

Auf dem Arduino-Board ist schon ein Spannungsregler montiert, außerdem funktioniert der USB-Chip auch als Quelle für eine Spannung von 3,3 Volt. Die Spannung, die das Arduino-Board auf den Spannungsversorgungspins liefert, ist also schon reguliert.

Steckbretter



◀ **Abbildung 2-16**
Steckbrett

Steckbretter sind ein schneller Weg, um kleine Schaltungen auszuprobieren. Sie bestehen aus einem oder mehreren Plastikbrettern, die in regelmäßigen Abständen durchlöchert sind. In diese Löcher lassen sich dann herkömmliche Bauteile (z. B. ICs in der DIP-Form, bedrahtete Widerstände oder LEDs) und einzelne Drähte stecken. Innerhalb des Bretts sind diese Löcher elektrisch verbunden. So lassen sich sehr leicht Schaltungen entwerfen: Zuerst werden die benötigten Bauteile auf das Brett gesteckt und dann untereinander mit Drähten verbunden.

Steckbretter sind allerdings nicht einheitlich aufgebaut, weshalb man zuerst herausfinden sollte, wie die Stecklöcher intern verbunden sind. Im Normalfall gibt es oben und unten eine durchgehend verbundene Reihe. Die obere wird mit 5 oder 3 Volt verbunden, die untere mit *Ground* (*GND*). So kann man der Schaltung von jeder Stelle aus die zwei Referenzspannungen zuführen.

Wichtig ist, dass alle Drähte und Bauteile gut eingesteckt sind, weil es sonst zu Kurzschlüssen und Unterbrechungen in der Schaltung kommen kann. Besonders bei neuen Steckbrettern lassen sich Drähte und andere verdrahtete Bauteile oft nur mit Schwierigkeit in die Löcher hineindrücken.Deshalb ist es immer nützlich, eine kleine Zange zur Verfügung zu haben, um hartnäckige Schaltungen sauber aufzubauen. Ansonsten verbringt man viel Zeit mit unnötiger Fehlersuche. Wegen ihrer Bauweise sind Steckbretter auch sehr anfällig für Rauschen und Störsignale, sodass sich manchmal bestimmte Sensoren oder Bausteine (besonders solche, die mit kleinen Spannungen oder hoher Geschwindigkeit arbeiten) falsch verhalten. Auch hier gilt die Regel, alle Verbindungen möglichst kurz zu halten.

Ein weiteres Problem mit Steckbrettern ist, dass die Schaltungen, die auf ihnen aufgebaut sind, recht schnell kaputt gehen. Ein Steckbrett ist auch relativ groß und unhandlich, sodass es sich nicht einfach transportieren lässt. Auch ein Gehäuse ist aus diesen Gründen meistens nicht sehr praktisch. Deswegen werden Steckbretter vor allem für Entwicklung und Prototypentwurf eingesetzt. Wenn eine Schaltung auf dem Steckbrett funktioniert und später weiterverwendet werden soll, lohnt es sich, sie auf eine Lochrasterplatine zu transferieren oder gleich eine Platine zu entwerfen (siehe auch in den Abschnitten über Löten auf Lochrasterplatinen und Platinen in diesem Kapitel).

Auf den meisten Darstellungen in diesem Buch wird ein Steckbrett verwendet. Dazu werden einzelne Drähte von den Anschlusspins des Arduino-Boards damit verbunden. Es gibt auch spezielle Arduino-Boards und Arduino-Shields, die ihr eigenes kleines Steckbrett mitbringen, sodass sich direkt auf dem Board selbst kleine Schaltungen entwerfen lassen.

Kabel, Stecker und Buchsen

Zu den wichtigsten Bauteilen in elektronischen Schaltungen, die oft nicht erwähnt werden, gehören Kabel, Stecker und Buchsen. Kabel werden verwendet, um verschiedene Bauteile miteinander zu verbinden. Das kann in Form von Leiterbahnen auf der Platine geschehen, aber auch mit Drähten, die auf einer Lochrasterplatine angelötet werden, oder mit Kabeln. Hiervon gibt es zwei unterschiedliche Arten. Die eine hat als Leiter einen einfachen Draht und ist einfacher zu verwenden, weil sie sich zum Beispiel sehr

leicht in Steckbretter stecken lässt und weil sie nach dem Biegen ihre Form behält. Allerdings ist sie mechanisch nicht sehr belastbar, weil der einzige Draht in ihrem Kern sehr schnell brechen kann, wenn man das Kabel bewegt.

Deshalb gibt es eine andere Sorte Kabel, die in ihrem Innern eine Kupferlitze hat, die aus vielen kleinen einzelnen, biegsamen Drähten besteht. Diese Kabel sind deutlich robuster und werden oft verwendet, um zum Beispiel Schalter und Taster mit einer Platine zu verbinden oder Kabel über längere Distanzen zu verlegen. Es gibt sie auch in mehradriger Ausführung, z. B. als Netzwerk-, Telefon- und Druckerkabel. In einem mehradrigen Kabel sind mehrere einzelne Litzenkabel zusammen verlegt, sodass sich z. B. mehrere Datenleitungen mit einem einzigen Kabel verbinden lassen. Allerdings kann man Litze nicht einfach ins Steckbrett einstecken, sondern muss sie an eine Steckerleiste anlöten, die man dann in das Steckbrett einbauen kann.

Eine praktische Alternative zu Steckbrettern sind Kabel mit Krokodilklemmen, die sich gut dazu eignen, größere mechanische Bauteile miteinander zu verknüpfen, um Projekte, die viel Raum brauchen, aufzubauen. Allerdings halten Krokodilklemmen nicht besonders gut, sodass diese Art des Verkabelns wirklich nur kurzzeitig verwendet werden sollte, um dann z. B. durch robuste Stecker ersetzt zu werden.

Zusätzlich zu Kabeln gibt es auch eine unendliche Anzahl verschiedener Stecker und Buchsen; viele Kabel sind auch schon mit passenden Steckern versehen (alternativ kann man Stecker auch anlöten oder mit einer sogenannten Crimpzange ancrimpen). Es gibt sie in allen Formen und Größen, und sie werden verwendet, um Platinen mit Strom zu versorgen und verschiedene Platinen oder Bauteile miteinander zu verknüpfen, ohne dass jedes Mal eine Verbindung gelötet werden muss.

Eine wichtige und beim Basteln häufig verwendete Steckerform sind die sogenannten Pin- und Buchsenleisten, die meistens in einem 2,54-mm-Raster verwendet werden. Pinleisten bestehen aus einer oder zwei Reihen von Pins, die aus der Platine hervorstehen. Sie werden verwendet, um Flachbandkabel aufzustecken oder bestimmte Verbindungen mithilfe von Jumpern herzustellen. Auf dem Arduino-Board gibt es eine doppelreihige Pinleiste (2x3), die zum Programmieren des Arduino-Prozessors verwendet wird. Auf älteren Arduino-Boards wie dem Arduino NG wird auch eine

kleine Pinleiste in Verbindung mit einem Jumper benutzt, um die Stromversorgungsquelle auszuwählen.

Das Pendant zu Pinleisten sind Buchsenleisten, wie sie auf dem Arduino-Board verwendet werden, um die Eingangs- und Ausgangspins des Prozessors mit der Außenwelt zu verbinden. Buchsenleisten sind insbesondere für das Ausprobieren von Schaltungen auf einem Steckbrett praktisch, weil sich die Drähte, die auf dem Steckbrett verwendet werden, sehr leicht hineinstecken lassen. Allerdings fallen diese Kabel dementsprechend auch schnell wieder aus den Buchsen heraus, sodass hier ein bisschen Vorsicht geboten ist, wenn der Aufbau bewegt wird.

Ein weiterer Vorteil der Buchsenleisten auf dem Arduino-Board ist, dass es dadurch möglich ist, ein weiteres Board, das mit Pinleisten versehen ist, aufzustecken. Dadurch sind sofort alle Pins des Arduino-Prozessors mit den Pins der Schaltung auf dem zweiten Board verbunden, und der ganze Aufbau ist sehr solide. Es gibt eine ganze Reihe von Erweiterungsboards für den Arduino, die »Shields« (Schilde) genannt werden. Eine Auswahl dieser Shields wird in Anhang A vorgestellt.

Abbildung 2-17 ▶
Kabel

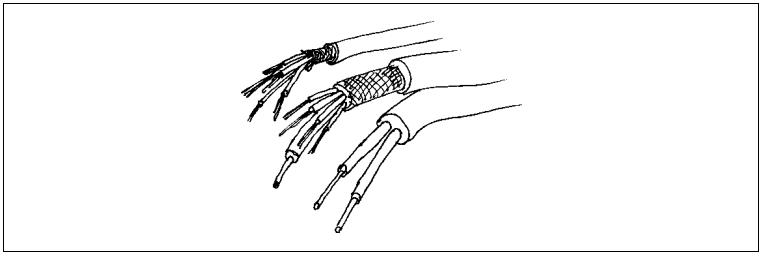
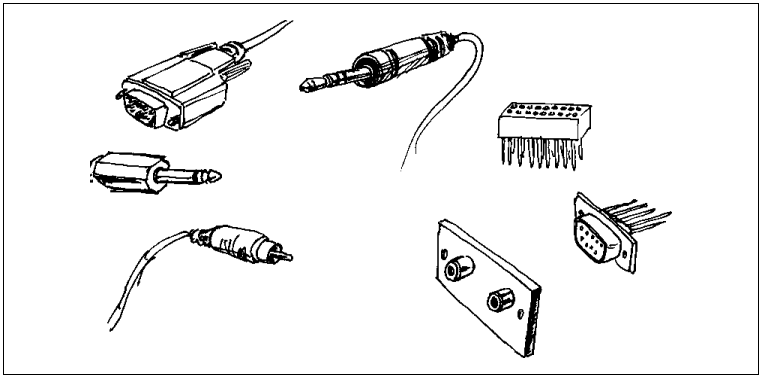


Abbildung 2-18 ▶
Stecker



ICs

ICs (integrated circuits, integrierte Schaltkreise), auch Chips genannt, sind komplette Schaltungen, die in einem eigenen kleinen Plastikgehäuse platziert werden und sich für spezielle Aufgaben in einer Schaltung einsetzen lassen. ICs sind also ein Ersatz für Schaltungen, die man meistens auch separat aufbauen könnte; aber durch die seit den 1960er Jahren immer weiter verbesserten und verfeinerten Herstellungsmethoden ist es möglich, solche Schaltungen auf dem Bruchteil eines Quadratzentimeters zu realisieren. Integrierte Schaltkreise können sehr einfache Schaltkreise sein (zum Beispiel eine Reihe von Transistoren zur Stromverstärkung, die man Treiberbausteine nennt), aber auch hoch komplizierte CPUs und Signalprozessoren. Auch der Arduino-Prozessor ist ein solcher Baustein.

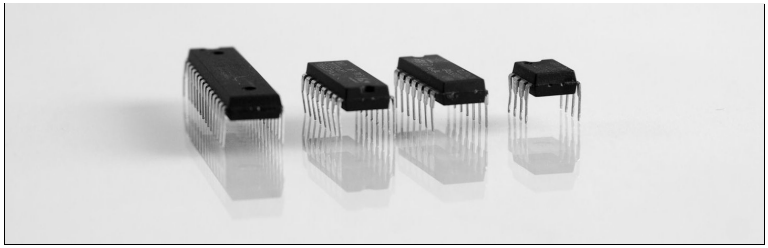
ICs gibt es in verschiedenen Bauformen; die meisten haben mittlerweile die SMD-Bauform, weshalb sie nicht so einfach beim Aufbauen von Schaltungen auf der Steckplatine zu verwenden sind. Dazu eignet sich die DIP-Bauform deutlich besser, weil diese sich direkt auf der Steckplatine einstecken lässt. Alle ICs haben Pins, die zur Stromversorgung angeschlossen werden sollen. Hier ist besondere Vorsicht geboten, damit man die Stromversorgung nicht invertiert (als GND an die positive Spannung anschließen und VCC an die Masse), weil dadurch oft die Chips kaputtgehen. Zu jedem IC gibt es vom Hersteller ein Datenblatt, das die Pinbelegung zeigt (hier heißt GND auch oft Vss) und die Funktionsweise des Chips erklärt. Weiterhin gibt es in jedem Datenblatt eine Tabelle, die Grenzwerte und Sollwerte für verschiedene Größen wie maximalen Strom, maximale Spannung und niedrigste und höchste Geschwindigkeit angibt. Oft ist es praktisch, im Arduino-Wiki unter *http://arduino.cc/* zu suchen, ob es schon Beispielschaltungen zu einem bestimmten Chip gibt (und vielleicht sogar schon eine Library, die die Kommunikation mit dem Chip implementiert).

Wenn eine Schaltung auf einer Platine gebaut wird (oder auch auf einer Lochrasterplatine), ist es bei DIP-Gehäusen sinnvoll, die Chips auf einem Sockel einzubauen. Der Sockel wird auf die Platine gelötet und der Chip dann nur in den Sockel eingesteckt. Mit einem Schraubenzieher oder einer dünnen Zange (es gibt auch spezielle Chipzangen) lässt er sich dann auch vorsichtig wieder herausnehmen. Dadurch lassen sich defekte Chips auswechseln, ohne gleich

eine neue Platine bauen zu müssen oder eine extensive Entlötaktion durchzuführen.

In diesem Buch kommen nur wenige ICs zum Einsatz, und wenn welche benutzt werden (z.B. der Ethernet-Controller in Kapitel 6), sind sie auf einem externen Shield angebracht. Es gibt allerdings eine Reihe wichtiger Chips, die häufig praktisch eingesetzt werden. Erwähnenswert sind hier sogenannte Shift-Register. Diese Bauteile können über eine synchrone serielle Schnittstelle (also zwei Pins, von denen der eine als Taktgeber benutzt wird und der andere zur Datenübertragung) Daten empfangen und senden. Es gibt zwei Arten von Shift-Registern: Solche, die als Ausgabe benutzt werden, indem sie über die serielle Schnittstelle Daten empfangen und beim Aktivieren eines sogenannten Latch-Signals an ihre Ausgänge legen, und solche, die als Eingabe benutzt werden. Sie übertragen die Daten, die an ihren Eingängen liegen, seriell an den Empfänger. Dadurch kann man mehrere digitale Ausgänge oder Eingänge über eine serielle Schnittstelle einlesen und spart sich so eine große Anzahl an belegten Pins. Ein weiterer Vorteil von Shift-Registern ist, dass man sie verketten kann. Es ist also möglich, über eine serielle Schnittstelle beliebig viele von ihnen auszulesen oder zu schreiben. Weit verbreitete Shift-Register sind z.B. das Ausgabe-Shift-Register 74HC595 und die Eingabe-Shift-Register 74HC165 oder CD4021. Ihre Verwendung ist im Arduino-Wiki unter *http://www.arduino.cc/en/Tutorial/ShiftIn* und *http://www.arduino.cc/en/Tutorial/ShiftOut* erklärt, und es gibt in der Arduino-Programmiersprache passende Funktionen, um diese Chips anzusprechen.

Ein sehr wichtiger Chip auf dem Arduino-Board ist der Mikrocontroller der AVR-Familie von Atmel. Dieser Chip enthält einen kompletten kleinen Mikroprozessor, auf dem alle Arduino-Programme ausgeführt werden. Das Arduino Duemilanove gibt es mit zwei verschiedenen Versionen des AVR-Chips: eine Version mit Atmega168 (so heißt die genaue Bezeichnung des Chips) und eine mit Atmega328. Die neue Version mit Atmega328 verfügt über ein bisschen mehr Speicher. Bei einem Arduino-Board mit Atmega168 kann man einfach einen neuen Atmega328-Chip mit Arduino-Bootloader (siehe Kapitel 1) bestellen und den Atmega168 austauschen. Dazu muss nur der alte Chip zunächst vorsichtig mit einem Schraubenzieher ausgehebelt werden, damit man den Atmega328 einstecken kann. Dabei muss man auf die Richtung achten: Auf der Platine ist eine Darstellung des Chips mit einer kleinen Kerbe eingezeichnet, die auch auf dem eigentlichen Chip vorhanden ist.



◀ **Abbildung 2-19**
ICs

Zangen und Pinzetten

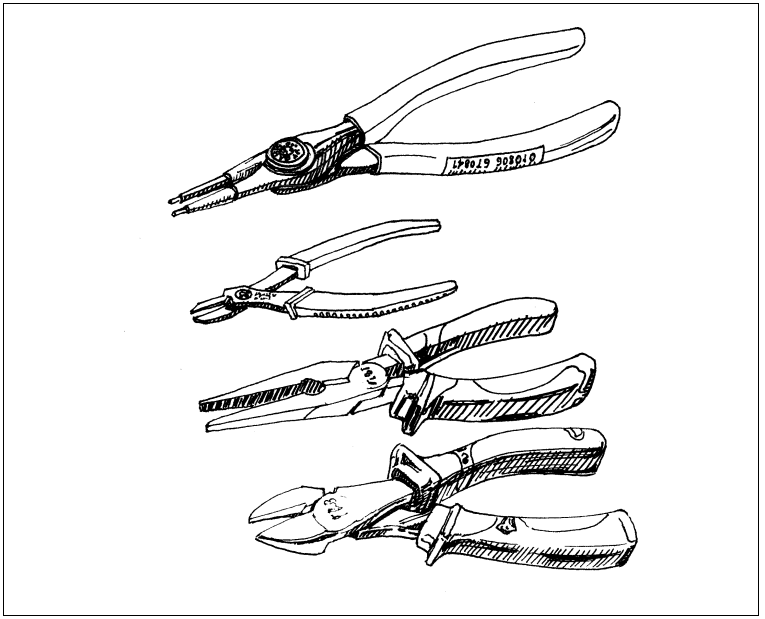
Es gibt eine Reihe sehr praktischer Werkzeuge für die Arbeit mit Drähten, Kabeln und elektronischen Komponenten. Diese Werkzeuge kommen immer wieder zum Einsatz, weswegen es auch nicht verkehrt ist, auf Qualität zu setzen und ein bisschen mehr auszugeben. Mit guten Werkzeugen macht das Basteln deutlich mehr Spaß, und das Ergebnis ist meistens auch weniger fehleranfällig.

Eine Zange, die oft zum Durchtrennen von Kabeln, Drähten und Bauteilbeinchen benutzt wird, ist der *Seitenschneider*. Es gibt spezielle kleine Seitenschneider zum Arbeiten mit elektronischen Schaltungen. Sie haben isolierte Griffe und Schneidekanten, die auf der einen Seite gerade sind. Dadurch lassen sich Komponentenbeine sehr nah an der Platine abschneiden und Drähte präzise trennen. Beim Durchtrennen von Drähten und Bauteilbeinchen sollte immer ein Finger die abzuschneidende Seite festhalten, damit diese nicht durch den Raum fliegt und unter Umständen Menschen verletzt. Es gibt auch spezielle Seitenschneider, die über eine kleine Metallkonstruktion gleich die abzuschneidenden Teile fixieren. Weiterhin ist es mit einem kleinen Trick auch möglich, mit dem Seitenschneider Kabel schnell abzuisolieren. Dazu drückt man den Seitenschneider nicht komplett zu, sodass nur die Hülle geschnitten wird, und zieht dann die Ummantelung vom Kabel ab. Das erfordert ein bisschen Übung, damit nicht die Litze oder der innere Draht beschädigt wird, spart aber später immer einen Werkzeugwechsel.

Für genau diesen Vorgang gibt es die *Abisolierzange*, eine spezielle Zange, mit der das Abisolieren präzise und wiederholbar durchgeführt werden kann (das ist insbesondere dann praktisch, wenn sehr viele Kabel abisoliert werden müssen). Für Bastelzwecke lohnt sich eine kleine solche Zange für die dünnen Drähte, die auf einem Steckbrett eingesetzt werden. Alternativ kann man auch schon fertig abisolierte Drähte für Steckbretter benutzen (die man in verschiedenen Farben und Größen kaufen kann).

Weitere wichtige Werkzeuge sind *Spitzzange* und *Pinzette*, mit denen man kleinere Komponenten platzieren kann. Oft ist das Steckbrett ein bisschen widerspenstig und es ist notwendig, Drähte und Komponenten mit einer dünnen Zange einzustecken. Weiterhin kann man mit einer Pinzette auch eingesteckte Kabel einfacher herausziehen, ohne andere Kabel in Mitleidenschaft zu ziehen (was bei größere Steckbrettprojekten meist ein Problem ist).

Abbildung 2-20 ▶
Zangen



Löten

Die meisten der Schaltungen in diesem Buch lassen sich ohne Löten auf einem Steckbrett anbringen. Allerdings ist diese Art, Projekte aufzubauen, weder sonderlich stabil noch elegant oder ästhetisch. Immer besteht die Gefahr, dass einzelne Kabel aus den Löchern herausfallen, wenn z.B. am Arduino gezogen wird. Auch das Steckbrett selbst hat oft nicht die gewünschten Dimensionen, sodass es nicht möglich ist, die komplette Schaltung in ein Gehäuse einzubauen oder im Rahmen einer Kunstinstallation an einem Gegenstand zu befestigen. In all diesen Fällen ist es sinnvoller, die Schaltung, sobald sie fertig und getestet ist, in eine endgültige Form zu bringen, indem man sie auf eine Platine lötet. Das ist

nur halb so wild, wie es klingt, und nach ein paar Versuchen stellt sich schnell heraus, dass Löten sogar Spaß macht. Vor allem werden dadurch die einzelne Projekte, die in diesem Buch vorgestellt oder später eigens entwickelt werden, zu richtigen benutzbaren und robusten Gegenständen, die immer wieder verwendet werden können.

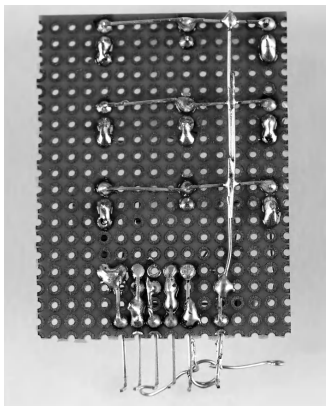
Im Internet gibt es eine große Gemeinde von Elektronikerinnen und Elektronikern, die Schaltungen und Projekte entwickeln und in Form von Schaltplänen und Bausätzen veröffentlichen. Mit Lötkenntnissen ist es dann sehr einfach, einen dieser Bausätze zu bestellen, zu bestücken und zum Laufen zu bringen. Diese Bausätze reichen von einfachen Blinkschaltungen (wie der Brainmachine von Mitch Altman, unter *http://makezine.com/10/brainwave/* zu finden und auch in Kapitel 4 beschrieben, und dem Open Heart Kit von Jimmie Rodgers, unter *http://www.jimmieprodgers.com/openheart* zu finden) über Gadgets wie das T.V.-be-Gone (auch von Mitch Altman, unter *http://www.tvbgone.com/*) bis hin zu komplizierten Projekten wie der x0xbox (einem Nachbau der legendären TR-303 Groovebox von Ladyada), irrsinnig vielen verschiedenen Synthesizermodulen (siehe auch Kapitel 10), Robotern, selbstfahrenden Autos und Ähnlichem. Das Löten eröffnet also eine riesige Welt an Möglichkeiten.

Lochrasterplatinen

Die einfachste Art, eine Schaltung von einem Steckbrett auf eine Platine zu bringen, ist die Benutzung einer Lochrasterplatine. Diese gibt es in verschiedenen Formen. Sie bestehen aus einer Grundplatte (meistens aus Epoxid, manchmal auch aus Karton), auf der in regelmäßigen Abständen Löcher eingestanzt sind. Diese Löcher sind von einer kleinen Kupferschicht umgeben und auf manchen Platinen auch schon rudimentär verknüpft (in sogenannten Rails, Verknüpfungen, die sich über das ganze Board ziehen). Dort lassen sich Komponenten hindurchstecken, ähnlich wie bei einem Steckbrett. Die Beinchen dieser Komponenten werden dann an die Kupferschicht gelötet, siehe Abschnitt »Löten«.

Wie man sich leicht vorstellen kann, tauchen hier deutlich mehr Probleme auf, als wenn eine Schaltung auf einem Steckbrett zusammensetzt wird. Ist ein Bauteil schon angelötet, ist es nur schwer wieder zu lösen. Weiterhin dürfen sich Drähte nicht kreuzen, weil

Abbildung 2-21 ▶
Gelötete Lochrasterplatine



es bei einer Berührung zu einem Kurzschluss zwischen zwei Leitungen kommt. Benutzt man allerdings isolierte Kabel, die sich kreuz und quer überschneiden, wird die Schaltung sehr schnell unübersichtlich. Es ist auch möglich, dass man plötzlich feststellt, dass eine bestimmte Leitung gar nicht ziehbar ist, weil es keinen Platz mehr auf der Platine gibt.

Deswegen ist es sehr wichtig, sich im Voraus Gedanken zu machen, wie die Bauteile angeordnet und miteinander verknüpft werden sollen. Es ist z.B. möglich, die Schaltung auf kariertem Papier zu zeichnen und dort die Bauteile anzuordnen. Als Grundregel gilt, dass alle Leitungen auf der Unterseite der Platine in vertikaler Richtung und vereinzelte Verknüpfungen dann auf der Oberseite horizontal mit einem weiteren Draht gezogen werden. Dadurch bleibt die Schaltung übersichtlich, und es ist einfach, sie zu erweitern. Dieses Vorgehen ähnelt dem eigentlichen Entwerfen von Platinen und ist eine Kunst für sich. Übung macht den Meister, und nach einigen Versuchen steht keiner Schaltung mehr die Platine im Weg! Es ist auch möglich, ein richtiges Entwurfsprogramm für Platinen zu verwenden (z.B. EAGLE, KiCad oder gEDA, siehe oben im Abschnitt »Schaltbilder«).

Lötkolben und Lötzinn

Wenn der Entwurf für die Schaltung steht, ist es Zeit, ans Löten zu gehen. Hierbei werden elektronische Bauteile mit geschmolzenem Metall verbunden und auf eine Platine montiert (im Gegensatz zu einem Steckbrett, wo die Bauteile nur eingesteckt und mit Drähten verknüpft werden). Die Metallelemente der Platine (die Leitungen)

und der Bauteile (die Drahtbeinchen) werden mit der Spitze des Lötkolbens erhitzt, und ein drittes Metall (das man Lötzinn oder Lot nennt) wird zum Schmelzen gebracht. Das geschmolzene Lötzinn haftet an beiden Metalloberflächen (auf Platinen wird meistens Kupfer verwendet) und erhärtet sich beim Kühlen, sodass es Bauteil und Platine zusammenhält.

In diesem Abschnitt werden die verschiedenen Werkzeuge und Verbrauchsmaterialien vorgestellt, die beim Löten verwendet werden.

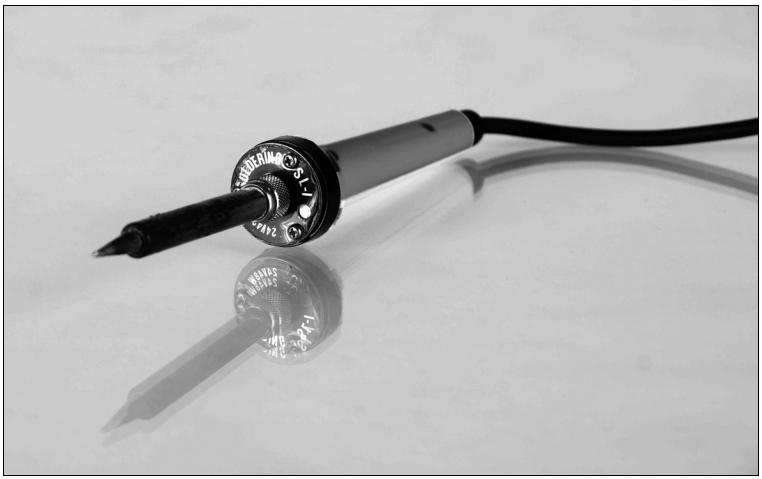
Das wichtigste Werkzeug ist natürlich der Lötkolben. Er besteht aus einer kleinen Lötstation (es gibt auch Lötkolben, die direkt an die Stromversorgung angeschlossen werden), mit der der Lötkolben an- und ausgeschaltet wird, dem Kolben an sich, der aus einem Griff und einem Heizelement besteht und über ein Kabel an die Lötstation angeschlossen ist, und einer Lötspitze aus Metall, mit der die eigentliche Lötarbeit durchgeführt wird.

Es gibt sehr viele Lötkolben, die man käuflich erwerben kann, je nachdem, welche Eigenschaften gewünscht sind. Auf zwei Dinge sollte geachtet werden: Zunächst die Leistung, die der Lötkolben erbringen kann. Diese wird in Watt gemessen und beeinflusst, wie schnell ein Kolben seine Lötspitze erhitzen kann. Die andere wichtige Eigenschaft ist, dass der Lötkolben über eine Temperaturregelung verfügen sollte. Dadurch lässt sich die Temperatur der Spitze einstellen und je nach benutzten Bauteilen und zu lötenden Flächen (für größere Bauteile und Flächen wird mehr Leistung benötigt) verändern. Besonders interessant sind Lötkolben mit automatischer Temperaturregelung, die die Leistung immer so anpassen, dass die Lötspitze die gewünschte Temperatur hat. Für elektronische Schaltungen braucht man ein Lötkolben, der ungefähr 25 bis 45 Watt Leistung hat. Bei größerer Leistung besteht die Gefahr, dass Bauteile und Platine beschädigt werden, und bei kleinerer Leistung heizt der Kolben das Lötzinn nicht genug, was die Gefahr schlechter Lötverbindungen erhöht.

Die Spitze des Lötkolbens ist praktisch gesehen der wichtigste Teil des Lötkolbens. Es ist wichtig, eine gute Lötspitze auszusuchen (aus Eisen, nicht aus Kupfer). Je dicker die Spitze ist, desto mehr Hitze kann an die zu lötenden Bauteile übertragen werden, aber umso vorsichtiger muss dann vorgegangen werden, um nicht angrenzende Bauteile und Lötpunkte mit zu erhitzen. Mit einer kleineren Spitze ist es zwar einfacher, präzise zu arbeiten, durch die geringere Hitzeübertragung ist es allerdings schwieriger, gute Lötpunkte zu erzeugen. Besonders wichtig ist, dass die Spitze nicht

oxidiert. Sonst überträgt sie keine Hitze mehr, und es wird sehr schwer, sauber zu löten. Deswegen muss sie regelmäßig gesäubert und anschließend mit Lötzinn geschützt werden (es wird ein bisschen Lötzinn auf der Spitze geschmolzen und dort gelassen). Säubern kann man die Spitze mit einem kleinen nassen Schwamm (allerdings nicht, wenn bleifreies Lötzinn verwendet wird) oder z. B. mit Stahlwolle. Es kann vorkommen, dass nach einiger Zeit die Lötspitze, besonders wenn sie schlecht gepflegt wird, komplett oxidiert und nicht mehr zum Löten verwendet werden kann. Bei den meisten Lötkolben lässt sich die Lötspitze austauschen; vor dem Austauschen sollte die Spitze natürlich abgekühlt sein.

Abbildung 2-22 ▶
Lötkolben



Das zweite wichtige Element beim Löten ist das Lötzinn. Auch hier gibt es eine große Menge verschiedener Arten, insbesondere seit in Europa die ROHS-Regelung in Kraft getreten ist, die für die industrielle Fertigung bleifreies Lötzinn vorschreibt. Im Hobbybereich ist es allerdings immer noch möglich, mit bleihaltigem Lötzinn zu arbeiten, was auch deutlich einfacher ist, denn bleifreies Lötzinn hat einen deutlich höheren Schmelzpunkt und benötigt starke Flussmittel. Diese sind schädlich und man kommt gerade beim Handlöten leicht mit ihnen in Kontakt. Als Lötzinn für die Projekte in diesem Buch und für weitere Hobbyelektronikprojekte sei normales bleihaltiges 60/40-Lötzinn mit Flussmittelkern empfohlen. Das bedeutet, dass 60% aus Zinn bestehen und 40% aus Blei. Wegen der enthaltenen Flussmittel (die dazu verwendet werden, die Metalle vor dem Verbinden zu säubern, um das Anhaften des Lötzinns zu erleichtern) lässt sich damit sehr gut arbeiten. Verfüg-

bar ist es in kleinen Drahtspulen mit verschiedenen Durchmessern. Es ist empfehlenswert, eher dünnes Lötzinn zu nehmen, weil sich dadurch die aufgetragene Menge einfacher kontrollieren lässt. Praktisch sind zum Beispiel Spulen mit 0,5 mm Durchmesser.

◀ **Abbildung 2-23**
Dritte Hand

Hilfreich beim Zusammenlöten von komplizierteren Schaltungen ist eine kleine Tischzange, die man *dritte Hand* nennt. In ihren Klemmen kann man zum Beispiel eine Platine einspannen, sodass man Bauteile leichter einstecken, festhalten und anlöten kann. Es ist auch möglich, damit einzelne mechanischen Komponenten festzuhalten. Kleiner Tipp am Rande: Man kann auch Lötzinn so verbiegen, dass es von selbst auf der Tischoberfläche hält, sodass man einfacher Komponenten anlöten oder Kabel verzinnen kann.

Löten

Der eigentliche Lötvorgang ist sehr einfach. Als Erstes muss ein Bauteil durch die Löcher in der Platine (sei es auf einer Lochrasterplatine oder auf einer richtigen) gesteckt und dort festgehalten werden. Das kann man entweder mit einer dritten Hand machen oder einfach, indem man die Beinchen des Bauteils leicht verbiegt, sodass es von selbst hängenbleibt. Im nächsten Schritt werden mit dem Lötkolben in der einen Hand und mit dem Lötzinn in der anderen das Beinchen und die Leitung auf der Platine erhitzt. Nach ein oder zwei Sekunden ist die Stelle heiß genug, dass man das Lötzinn anbringen und an die Lötspitze halten kann. Das Lötzinn

schmilzt sofort (wenn die Temperatur des Lötkolbens richtig eingestellt ist – ca. 300 Grad Celsius für bleihaltiges Lötzinn und 350 bis 400 Grad für bleifreies) und verteilt sich auf Platine und Beinchen. Anschließend werden Lötkolben und Lötzinn entfernt, der Lötpunkt kühlt ab, und das Bauteil ist mit der Platine verbunden. Der erzeugte Lötpunkt sollte glatt und glänzend sein (bei bestimmten bleifreien Lötzinnen sind die Lötpunkte allerdings immer grau und matt) und das komplette Loch auf der Platine überdecken. Ist der Lötpunkt matt und merkwürdig geformt, bildet er mit hoher Wahrscheinlichkeit eine kalte Lötstelle, die schlecht oder gar nicht leitet und wahrscheinlich auch leicht durch mechanische Belastung bricht.

Eine wichtige Regel für den Lötvorgang ist, dass weniger Lötzinn besser ist. Es sollte immer nur so viel Lötzinn angebracht werden, wie ausreicht, um eine stabile Verbindung herzustellen. Bei zu viel Lötzinn besteht die Gefahr, dass sich ein kleines Lötkügelchen bildet, das unter Umständen nicht richtig leitet. Ein weiterer wichtiger Punkt ist, dass der Vorgang schnell ausgeführt werden sollte. Die verschiedenen Schritte sollten zusammen nicht mehr als fünf Sekunden dauern. Je länger an einer Lötstelle geheizt wird, desto mehr steigt die Wahrscheinlichkeit, dass sowohl Bauteil als auch Platine beschädigt werden (gerade bei Lochrasterplatinen kommt es vor, dass sich die Kupferschicht vom Platinenmaterial löst). Je länger das Lötzinn erhitzt wird, desto weniger Flussmittel enthält es (weil dieses sehr schnell verdampft). Wenn kein Flussmittel mehr vorhanden ist, fließt das Lötzinn nicht mehr richtig und bleibt an Lötkolben, Bauteil und Platine haften, sodass er sich nicht mehr sauber verarbeiten lässt: Es kommt zu einer kalten Lötstelle. In diesen Fällen lässt sich das Problem durch Hinzufügen von Flussmittel oder durch Entfernen des Lötzinns lösen (siehe nächster Abschnitt »Entlöten«).

Nachdem ein oder mehrere Lötpunkte bearbeitet worden sind, muss die Spitze des Lötkolbens gereinigt werden. Dazu kommt der zuvor erwähnte Schwamm oder Stahlwolle zum Einsatz. Die Lötspitze sollte stets leicht von Zinn bedeckt und glänzend sein, um Oxidation zu vermeiden.

Um Kabel und Drähte anzulöten, müssen die abisolierten Enden der Kabel verzinnt werden. Bei Litze ist es praktisch, die einzelnen Kupferdrähte zusammenzudrehen, sodass sie nicht abstehen. Anschließend wird das Ende des Kabels leicht erhitzt und mit Lötzinn bedeckt. Dadurch lässt sich ein Kabel leicht an einem Bauteil befes-

tigen, weil das zuvor verzinnte Ende beim Aufheizen gleich angelötet wird.

Um Leiterbahnen auf einer Lochrasterplatine herzustellen, wird oft verzinnter Draht verwendet (mit Kupferlack bedeckter Draht funktioniert nicht so gut, weil diese Lackisolierung erst weggebrannt werden muss). Es ist wichtig, bei Drähten, die geknickt werden, alle möglichen Kanten auf der Lochrasterplatine zu befestigen, damit sich die Drähte nicht lösen oder verbiegen und so Kurzschlüsse erzeugen.

Entlöten

Trotz aller Vorsicht und Übung kommt es immer wieder vor, dass ein Bauteil entweder falsch angelötet wird oder beim Löten ein Fehler passiert, wenn zum Beispiel zwei benachbarte Lötpunkte aus Versehen verbunden werden. In diesen Fällen muss die fehlerhafte Stelle entlötet werden. Dazu gibt es verschiedene Möglichkeiten: Bei einem Bauteil müssen alle Beinchen auf einmal entlötet werden, damit es auch ausgesteckt werden kann. Im einfachsten Fall (wenn z.B. die Beinchen nahe beieinander liegen) können alle Beinchen auf einmal erhitzt werden, indem man den Lötkolben schräg hält (dabei auf mögliche Verbrennungsgefahr achten!) oder noch mehr Lötzinn auf die zu entlötenden Stellen gibt, damit alle Stellen verbunden sind. Während eine Stelle erhitzt wird, kann man dann das Bauteil entfernen (hier muss auch darauf geachtet werden, dass es nicht zu Verbrennungen kommt, weil das Bauteil und besonders seine Beine heiß sind). Geschwindigkeit ist dabei auch von Vorteil, weil dadurch die Bauteile nicht beschädigt werden. Beim Herausziehen muss darauf geachtet werden, dass die Lötstellen auch wirklich geschmolzen sind, da man sonst leicht die Platine beschädigen kann, wenn sich Leiterbahnen bzw. Kupferringe von ihr lösen.

Systematischer kann man vorgehen, indem man *Entlötlitze* verwendet. Entlötlitze ist eine eng geflochtene Kupferlitze, die Lötzinn aufsaugen und so von der Platine abtragen kann. Die zu entlötende Stelle wird durch die Litze hindurch erhitzt, und das geschmolzene Lötzinn wird von der Litze aufgesaugt. Wenn anschließend die Lötstelle frei von Zinn ist, können die Bauteile entfernt werden. Hier muss darauf geachtet werden, dass die Stellen wirklich sauber und frei sind, weil sonst die Platine beim Herausziehen beschädigt werden kann. Da die Litze aus Kupfer ist, wird auch sie sehr schnell heiß.

Alternativ kann man auch eine sogenannte *Entlötpumpe* verwenden, um Lötzinn von der Platine abzutragen. Die Entlötpumpe muss zuerst geladen werden, indem die interne Feder gespannt wird. Anschließend wird die zu entlötende Stelle mit dem Lötkolben erhitzt, die Entlötpumpe sehr nah darübergehalten und das geschmolzene Lötzinn aufgesaugt, indem der Pumpentaster, der die Feder auslöst, gedrückt wird. Dadurch wird das Zinn in die Pumpe aufgenommen und von der Platine abgetragen. Beim nächsten Aufladen der Pumpe wird das aufgesaugte Lötzinn nach außen gepresst und muss entsorgt werden. Bei der Verwendung ist Vorsicht geboten, weil man mit ihr sehr leicht Platinen beschädigen kann, wenn die Lötstelle nicht richtig erhitzt wurde.

Schließlich kann man z.B. für Chips oder mechanische Bauteile auch mit einer Heißluftpistole, die auf 300 bis 350 Grad kalibriert wird, großflächig entlöten. Die entsprechende Stelle wird mit der Heißluft erhitzt, bis das Zinn schmilzt. Anschließend kann man durch Umdrehen der Platine und leichtes Rütteln die Chips herunterfallen lassen. Bei diesem Vorgang können Chips schnell beschädigt werden, und der zu entlötende Bereich ist auch nicht sehr präzise einzustellen. Auch wird die Platine bei dem Vorgang extrem heiß, weshalb große Vorsicht geboten ist.

Sicherheit

Auch wenn Löten keine sonderlich gefährliche Aktivität ist, sollten ein paar Sicherheitsregeln beachtet werden. Die wichtigste ist natürlich, dass man den Lötkolben nicht an der Spitze berühren sollte. Das klingt zwar selbstverständlich, ist aber in der Realität doch nicht so einfach. Wie man auch schnell merkt, sind Metalle besonders gute Hitzeleiter, sodass Bauteilbeinchen oder Drähte, wenn sie mit dem Lötkolben erhitzt werden, schnell sehr heiß werden und auch nicht sonderlich schnell abkühlen. Deswegen ist es besser, beim Löten keinen direkten Kontakt zu erhitzten Metallteilen zu haben. Das gilt auch für Lötzinntropfen, die beim Löten abfallen. Diese können auch nach Minuten noch extrem heiß und geschmolzen sein. Beim Löten kommt es auch schnell zu kleinen Spritzern von Flussmittel und Lötzinn, weswegen es wichtig ist, eine Brille zu tragen, damit diese Spritzer nicht in die Augen gelangen. Es ist natürlich auch sinnvoll, den PC nicht in der unmittelbaren Nähe des Lötkolbens zu verwenden, besonders wenn es sich dabei um einen Laptop handelt: Zu schnell kann es passieren, dass die heiße Spitze an die Tastatur kommt und eine Taste zum

Schmelzen bringt oder im schlimmsten Fall sogar ein Loch in den Monitor brennt.

Beim Löten entstehen auch Dämpfe, die auf Dauer nicht gesund sind (besonders die Flussmitteldämpfe können allergische Reaktionen auslösen). Deswegen ist es wichtig, auf eine gute Lüftung am Lötplatz zu achten. Für viele Projekte reicht es schon, das Fenster aufzumachen. Es ist aber auch möglich, eine kleine Rauchabzugseinrichtung, die den Rauch mit einem Ventilator abzieht, in der Nähe des Lötplatzes aufzubauen. Dadurch werden die Dämpfe abgezogen und gelangen nicht in Ihre Lunge. Man kann sich auch eine solche Abzugseinrichtung selber mit einem kleinen Ventilator bauen. Wenn bleifreies Lötzinn verwendet wird, ist eine solche Abzugseinrichtung quasi unabdingbar, weil die verwendeten Flussmittel viel aggressiver sind. Wegen dieser Chemikalien und des Bleis im Lötzinn ist es wichtig, nach dem Löten gründlich die Hände zu waschen.

Fehlersuche in elektronischen Schaltungen

Wie man beim Basteln schnell feststellen kann, wird ein Großteil der Zeit beim Arbeiten an elektronischen Schaltungen und Projekten damit verbracht, Fehler zu identifizieren, zu isolieren und korrigieren. Das klingt natürlich nicht sehr erholsam und erfüllend, aber das Beheben eines Fehlers oder eines Problems ist eine sehr befriedigende Tätigkeit. Auch hier muss man allerdings mit System vorgehen, weil sonst schnell noch mehr Fehler eingebaut werden, bis am Ende vom schönen ursprünglichen Projekt nur noch ein großes Chaos übrig bleibt.

Häufige Fehlerquellen in Arduino-Projekten sind natürlich fehlerhafte Schaltungen (weil z.B. ein bestimmtes Bauteil falsch angeschlossen wurde oder falsch angesprochen wird). Auch mechanische und elektrische Probleme (z.B. Wackelkontakte oder fehlerhafte Bauteile) können zu interessanten Ergebnisse führen. Da in Arduino-Projekten auch Software eine große Rolle spielt, können viele Probleme von einem Fehler in der Programmierung stammen. Elektronische Schaltungen und Mikrocontroller-Programme (also Programme, die auf sehr kleinen Prozessoren laufen, die keinen Bildschirm oder keine Tastatur haben) sind eine ganz eigene Welt. Hier können obskure Probleme sehr wohl durch Fehler in der Software verursacht werden, es ist aber umgekehrt auch möglich, dass das scheinbar komplett zufällige Verhalten der Schaltung daher

rührt, dass ein Chip nicht mit Strom versorgt wird. Umso wichtiger ist es also, bei der Fehlersuche systematisch und Schritt für Schritt vorzugehen. In diesem Abschnitt werden einige Fehlerquellen vorgestellt sowie verschiedene Werkzeuge, mit denen sich Probleme in elektronischen Schaltungen identifizieren lassen. Nach einiger Zeit wird die Fehlersuche zu einer Selbstverständlichkeit, und es entwickelt sich allmählich eine Art Bauchgefühl dafür, welche Bauteile sich falsch verhalten könnten, welche Softwarekonstrukte anfälliger sind und welche dunklen Kräfte mal wieder Unheil stiften. Jeder Bastler hat seine eigenen Kriegsgeschichten über den unheilbaren Bug vom Sommer 1999 oder die gefürchtete asiatische Temperatursensortransistorverstärkungsschaltung.

Der wichtigste Schritt bei der Fehlersuche wird hier gleich zuerst verraten: tief durchatmen und eine Pause machen. Manchmal reicht eine erholsame Nacht oder ein kleiner Spaziergang, um plötzlich einen Fehler zu entdecken. Für die Autoren dieses Buchs gibt es den »Point of no return«. Ab einer gewissen Uhrzeit (meistens gegen 3 Uhr morgens) ist ein Punkt erreicht, an dem jeder Aufwand den Fehler nur noch schlimmer macht. Dann wird beschlossen, den Arbeitsplatz ein bisschen aufzuräumen und am nächsten Tag mit frischen Kräften weiterzumachen.

Der zweitwichtigste Schritt bei der Fehlersuche ist, die Schaltung einer Testperson zu erklären (z.B. der Lebensgefährtin oder der nächstbesten Person, die vorbeiläuft). Dabei ist es meistens relativ unwichtig, wie gut sich diese Person mit dem Thema auskennt, denn meistens wird beim Vortragen und Erklären des Problems die Fehlerquelle sofort klar. Unter Programmierern wird dieser Vorgang »Rubberducking« genannt (von englisch »Rubber Duck«, Plastikente), weil manche Programmierer sich eine Plastikente auf den Bildschirm stellen und ihr, die nun wirklich keine Programmierexpertin ist, bei Fehlern im Programm die möglichen Ursachen vortragen.

Die Fehlersuche kann man in verschiedene Bereiche aufteilen. Der erste Bereich ist das Verstehen der Schaltung und der Software: Hierbei sollte klar sein, wie die Schaltung funktioniert, welche Spannungen an welcher Stelle anliegen sollten, wie diese sich verändern, wie jedes Bauteil funktioniert und wie die Software sich zu verhalten hat. Bei Programmen sollte klar sein, welche Funktion welche Auswirkungen hat, von wo sie wie oft aufgerufen wird, welche Libraries verwendet werden, welche Variablen an welcher Stelle geschrieben und gelesen werden und wie der generelle Pro-

grammablauf gestaltet ist. Erst wenn die Schaltung richtig verstanden wurde, kann man beurteilen, ob Teilbereiche sich nicht so verhalten, wie sie sollten, ob ein Bauteil seine Aufgabe nicht erfüllt, oder ob es bei der Programmausführung zu Problemen kommt. Deswegen ist es natürlich sinnvoll, Schritt für Schritt vorzugehen und erst mit sehr einfachen Schaltungen und Projekten anzufangen, bis man mit ihnen gut vertraut ist. Anschließend kann man darauf aufbauend immer kompliziertere Projekte bauen.

Der zweite Bereich ist das Vereinfachen und Aufteilen des Projekts in kleine Bereiche, die sich getrennt testen lassen. Besteht ein Projekt aus einem Temperatursensor, einer LED, einer Tastatur und einer Servosteuerungsschaltung, ist es sinnvoll, jedes dieser Bauteile und Schaltungen getrennt zu überprüfen. So ist gewährleistet, dass keine Interaktion zwischen verschiedenen Bauteilen und Programmteilen Ursache des Problems ist. Durch diese Aufteilung wird auch die Komplexität der Schaltung und der Software dramatisch reduziert. Ist etwa die LED falsch herum angeschlossen, lässt sich das in einem separaten Test, bei dem man nur die LED an- und ausschaltet, deutlich einfacher überprüfen, als wenn die LED theoretisch nur angeht, wenn die Temperatur über 50 Grad liegt. Bei einer elektronischen Schaltung gibt es verschiedene Wege, die möglichen Fehlerquellen in kleine Bereiche zu isolieren.

Der einfachste Schritt ist, die Software so zu verändern, dass nur eine bestimmte Komponente angesprochen wird. Diese Testroutine lässt sich am besten als neuer Sketch in der Arduino-Umgebung schreiben. So kann für jede Teilschaltung (z.B. die LED, den Temperatursensor, die Tastatur und die Servosteuerung in der obigen Schaltung) getestet werden, ob sie funktioniert. Diese Test-Sketches sollten so geschrieben werden, dass sie keine komplexe Logik beinhalten (wie bei komplizierten Unterscheidungsfällen oder einer Benutzerschnittstelle), sondern sie sollten der Reihe nach verschiedene Funktionalitäten der untersuchten Schaltung testen. Funktioniert die untersuchte Schaltung, kann man Schritt für Schritt die weiteren Komponenten wieder mit in den Programmquellcode einbeziehen, bis die Schaltung nicht mehr funktioniert. Ab dem Moment weiß man, wo das Problem zu suchen ist. Wenn die mögliche Problemquelle identifiziert wurde, muss wieder der erste Schritt »Verstehen« angewendet werden, um nachzuvollziehen, was fehlerhaft ist. Oft lohnt es sich auch, ein kleines Logbuch zu führen, in das notiert wird, welche Fehlerquellen für welche Probleme verantwortlich sind und wie sie zu beheben sind.

Oft passieren nämlich immer wieder dieselben Probleme, ohne dass man sich im Eifer des Bastelns dran erinnern kann, was letztes Mal verbessert wurde.

Um Komponenten in Software zu testen, kann man die Debugging-methode »printf« anwenden. Dazu wird entweder die serielle Schnittstelle oder eine andere Ausgabemöglichkeit wie ein Textdisplay oder eine Reihe LEDs verwendet, um an bestimmten Stellen im Programm Informationen auszugeben. So kann z. B. bei jedem Durchlauf der Hauptschleife im Programm eine LED ein- und ausgeschaltet werden. Leuchtet die LED konstant, lässt sich daraus ableiten, dass das Programm irgendwo nach dem Anschalten hängen geblieben ist. Durch Verschieben der Anschaltanweisung kann man sich so Schritt für Schritt an die mögliche Fehlerquelle herantasten. Es ist Vorsicht geboten bei komplizierten Ausgabemethoden, z. B. der seriellen Schnittstelle oder einem grafischen Display, weil diese dann auch ein Teil des Problems werden können, wenn sie z. B. zu lange bei der Ausführung brauchen oder andere Komponenten elektrisch beeinflussen.

Wenn die getesteten Komponenten immer noch nicht funktionieren, ist es manchmal notwendig, die aufgebaute Schaltung auseinanderzunehmen, um sicherzustellen, dass nicht andere Komponenten durch eine nicht erkannte Interaktion die Schaltung stören. Hier ist es sinnvoll, vorsichtig, langsam und Schritt für Schritt vorzugehen. Nach jeder Änderung der Schaltung sollte ein Testprogramm ausgeführt werden, um nachzuprüfen, ob der Fehler verschwunden ist. Ist das der Fall, dann war die letzte Veränderung wahrscheinlich die Quelle des Fehlers. Deswegen ist es wichtig, immer nur eine Veränderung auf einmal zu machen, damit man anschließend feststellen kann, was die Ursache war. Werden gleichzeitig fünf Funktionen und drei Bauteile ausgetauscht, kann nicht mehr nachvollzogen werden, was die Ursache des Problems war. Bei Software ist es oft sinnvoll, Zwischenkopien des Programms bei Veränderungen zu machen (und nicht große Blöcke auszukommentieren, weil das zu einem schwer lesbaren und chaotischen Quelltext führt). Es gibt besondere Programme, die genau diese Aufgabe übernehmen: Source-Versioning-Systeme, die insbesondere für große Projekte sinnvoll sind, im Rahmen von Arduino-Projekten oft jedoch ein bisschen zu umständlich sein können.

So kann man systematisch an die möglichen Fehlerquellen kommen. Allerdings bestätigt die Ausnahme die Regel, und manchmal ist die Fehlerursache komplett woanders zu suchen. Solche Fälle

lassen sich nie voraussehen, und sie machen lustigerweise auch einen großen Teil des Spaßes aus. Wenn man nach acht Stunden endlich die Fehlerquelle identifiziert hat, die über fünf Umwege und drei Winkel an einer komplett anderen Stelle einen Fehler ausgelöst hat, fühlt man sich so erfolgreich wie Superman.

Häufige Fehlerquellen

In diesem Abschnitt werden einige Arduino-spezifische Fehlerquellen und Testvorgänge beschrieben. Als Erstes sollte immer sichergestellt werden, dass das Arduino-Board mit Strom versorgt ist. Das Board wird über USB an den Rechner angeschlossen. Die Power-LED sollte hellgrün leuchten. Leuchtet sie nur schwach oder flimmert, gibt es ein Problem mit der Stromversorgung: Entweder der USB-Port am Computer ist defekt oder das Kabel funktioniert nicht richtig. Ist die Power-LED komplett aus, gibt es ein Problem mit dem Computer, mit dem Kabel oder mit dem Board an sich. In diesen Fällen sollten ein anderes Kabel und ein anderer Computer ausprobiert werden. Wenn das Board immer noch nicht angeht, sollte es gewechselt werden.

Wenn der Arduino zum ersten Mal angeschlossen wird, sollte das Testprogramm, das mitgeliefert wird, ausgeführt werden und die L-LED regelmäßig blinken. Wenn sie das nicht tut, ist vermutlich schon ein anderes Programm hochgeladen worden. Das richtige Funktionieren des Prozessors und der LED kann überprüft werden, indem das LED-Testprogramm hochgeladen wird (siehe Kapitel 1). Lässt es sich nicht hochladen, kann das an einem Softwareproblem auf der Computerseite liegen (das ist auch die wahrscheinlichste Erklärung, keine Panik). Es sollte überprüft werden, ob die richtige serielle Schnittstelle und das richtige Boardmodell in der Arduino-Entwicklungsumgebung ausgewählt ist (siehe ebenfalls Kapitel 1). Weiterhin kann man unter Windows im Gerätemanager und unter Mac OS X im Programm System Profiler, das unter */Programme/Dienstanwendungen* zu finden ist, nachprüfen, ob das Arduino-Board tatsächlich als serielle Schnittstelle erkannt wird. Ist alles korrekt eingestellt, sollte auch überprüft werden, ob die Arduino-Entwicklungsumgebung die aktuellste Version ist.

Bei Schaltungen, die auf Steckbrettern aufgebaut werden, kann es schnell zu kleinen elektrischen Fehlern kommen, weil die eingesteckten Kabel leicht wieder herausrutschen können. Deswegen sollte immer sichergestellt werden, dass alle Kabel richtig sitzen

und die abisolierten Enden sich nicht berühren. Weiterhin kommt es manchmal auch vor, dass das Steckbrett an sich defekt ist, sodass z.B. die horizontale Durchkontaktierung einzelner Stecklöcher nicht mehr vorhanden ist oder im Gegenteil komplette Reihen kurzgeschlossen sind. Wird ein elektrisches Problem vermutet, sollte man ein bisschen an den Kabeln wackeln, um zu sehen, ob sie vielleicht das Problem sind, und zur Not die Schaltung auf einem anderem Bereich des Steckbretts aufbauen (oder auf einem komplett anderem Steckbrett).

Die Erklärung für viele weitere Arduino-Fehler (insbesondere Software- und Konfigurationsfehler) können Sie unter *http://www.arduino.cc/en/Main/FAQ* nachlesen. Weiterhin sei auf die Arduino-Foren unter *http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl* hingewiesen, in denen viele hilfsbereite Arduino-Bastler und -Bastlerinnen Projekte vorstellen und erklären und anderen Leute helfen. Die Teilnehmer in den Arduino-Foren sind alle freiwillig dort und möchten deshalb mit Respekt behandelt werden. Niemand ist verpflichtet zu helfen und schon gar nicht, einfache Aufgaben wie etwa das Googlen nach Informationen oder das Ausrechnen von Widerständen für andere zu übernehmen. Als Regel guten Umgangs gilt es, zunächst alle verfügbaren Anleitungen und Handbücher zu diesem Thema zu konsultieren und auch Suchmaschinen zu befragen.

Das Multimeter

Ein sehr wichtiges Werkzeug bei der Fehlersuche ist das Multimeter. Damit lassen sich verschiedenste elektrische Größen messen: Strom, Spannung, Widerstand und Kontinuität. Bestimmte Multimeter erlauben auch das Messen von Kapazitäten und das Überprüfen und Verstärkungsmessen bei Transistoren (diese Funktionalität ist allerdings oft nicht so wichtig). Es gibt analoge und digitale Multimeter: Erstere stellen die gemessene Größe mit einer Nadelanzeige dar und sind oft nicht sehr genau. Die meisten heutigen Multimeter sind allerdings digital und zeigen die gemessenen Werte auf einem LED-Display an.

Ein solches Gerät hat zwei Anschlüsse, an die sich Testspitzen anschließen lassen. Eine (meistens schwarz) wird an den Massestecker des Multimeters angeschlossen, während die andere (meistens rot) an den V-Stecker angeschlossen wird. Oft besitzen Multimeter zwei V-Stecker, von denen der eine für große Spannungen und Ströme ausgelegt ist und bei normalen Arduino-Schaltungen nicht

zum Einsatz kommt. Mit diesen Testspitzen kann man beliebige Punkte in der Schaltung berühren (natürlich müssen diese Punkte metallisch sein, damit auch ein Kontakt entsteht) und so elektrische Größen messen. Es ist wichtig, dass höchstens mit einer Hand eine der Testspitzen direkt berührt wird. Kommen beide Testspitzen in Kontakt mit dem Körper des Benutzers, wird seine Haut gleich mitgemessen, was insbesondere bei Widerständen zu falschen Ergebnissen führen kann.

Die gewünschte Funktionalität des Multimeters lässt sich meistens über ein Drehrad einstellen. Zu den verschiedenen Größen (A für Strom, V für Spannung, R oder Ohm für Widerstand) lässt sich auch die gewünschte Größenordnung einstellen.

◀ **Abbildung 2-24**
Multimeter

Elektrische Verbindungen überprüfen

Eine sehr praktische Funktion des Multimeters, die meistens mit einem kleinen Lautsprechersymbol gekennzeichnet ist, ist das Überprüfen der elektrischen Kontinuität, also ob zwei Punkte in einer Schaltung kurzgeschlossen sind. Bei der Überprüfung der elektrischen Kontinuität ist egal, welche der Testspitzen man benutzt. So kann man z. B. überprüfen, ob ein Bauteil tatsächlich an das Arduino-Board angeschlossen ist, indem man eine Testspitze auf das relevante Beinchen des Arduino legt und das andere an eins der Beinchen des Bauteils. Piept das Multimeter (manchmal wird auch eine Lampe eingeschaltet), besteht zwischen beiden Punkte

eine elektrische Verbindung. So lassen sich auch Kurzschlüsse zwischen zwei Punkten feststellen, die eigentlich nicht verbunden sein sollten. Wenn also vermutet wird, dass irgendwo eine Leitung getrennt ist oder ein Kurzschluss besteht, kann man das schnell mit dem Multimeter nachprüfen. Auch fehlerhafte Lötpunkte lassen sich so erkennen. Es ist oft sinnvoll, vor dem Anschließen einer neuen Schaltung zu überprüfen, ob vielleicht ein Kurzschluss zwischen Masse und Versorgungsspannung besteht.

LED und Diodenrichtung messen

Eine weitere praktische Funktion des Multimeters ist das Überprüfen von LEDs und Dioden. Meistens ist diese Funktion mit dem Symbol einer LED gekennzeichnet. Die rote Testspitze muss an die positive Seite der LED gebracht werden (die Anode, siehe Kapitel 3) und die schwarze Testspitze an die Kathode. Ist die LED richtig angeschlossen und nicht defekt, wird sie leicht aufleuchten, weil das Multimeter zum Testen der Dioden einen kleinen Strom fließen lässt. Zusätzlich zeigt das Multimeter auf dem Display an, ob Strom fließt (das kann von Multimeter zu Multimeter unterschiedlich angezeigt werden). So kann man leicht überprüfen, in welche Richtung eine LED angeschlossen werden muss. Ähnlich können auch herkömmliche Dioden geprüft werden.

Widerstandswerte messen

Das Multimeter kann als Ohmmeter eingesetzt werden, um den Widerstandswert von Widerständen zu messen. Beide Testspitzen müssen an die Anschlüsse des Widerstandes gebracht werden. Generell lassen sich Widerstände nur getrennt messen. Sind sie in einer Schaltung eingebaut, wird der parallele Widerstand des Rests der Schaltung auch mitgemessen, und der Ergebniswert ergibt nicht viel Sinn. Die Ohmmeter-Funktionalität des Multimeters kann mit dem Auswählen einer der Ohm-Größenordnungskategorien aktiviert werden. Ist die Größenordnung des zu messenden Widerstandes unbekannt, kann man sich herantasten, indem man zunächst die kleinste wählt.

Spannungen messen

Eine sehr wichtige Funktion zum Überprüfen von Schaltungen und für die Suche nach Fehlern in einer Schaltung ist die Voltmeter-Funktionalität des Multimeters. Ähnlich wie bei der Ohmmeter-

Funktionalität lassen sich hier verschiedene Spannungsgrößenordnungen auswählen. Da aber Spannungen in einer Arduino-Schaltung meistens unter 5 Volt liegen, kann man die entsprechende Größenordnung gleich auswählen. Die schwarze Testspitze sollte meistens an die Masse der Schaltung angeschlossen werden. Wird der Multimeter oft eingesetzt, kann man zum Beispiel ein Kabel mit Krokodilklemme verwenden, um die Masse fest zu verbinden. Mit der roten Testspitze lassen sich dann Spannungen an verschiedenen Stellen in der Schaltung untersuchen. Wichtig ist zum Beispiel zu überprüfen, ob jedes Bauteil an den vorgesehenen Pins auch mit der richtigen Spannung versorgt wird. Ist die Spannung an dem gemessen Punkt variabel, hängt die Anzeige von der Geschwindigkeit des Multimeters ab. Genaue Variationen lassen sich mit einem Multimeter nicht messen (dazu sollte ein Oszilloskop verwendet werden, siehe unten), man kann aber grob überprüfen, ob zum Beispiel Daten über eine serielle Leitung übertragen werden. Wichtig ist beim Messen einer laufenden Schaltung, dass mit den Testspitzen des Multimeters keine Kurzschlüsse erzeugt werden. Es ist dazu praktisch, die Testspitze möglichst vertikal über die zu messenden Punkte zu halten.

Ströme messen

Eine weitere Funktionalität zum Messen von laufenden Schaltungen, die allerdings nicht so oft verwendet wird wie die Voltmeter-Funktionalität, ist die Amperemeter-Funktionalität zum Messen von Strömen. Dafür muss das Multimeter allerdings in Serie mit der zu messenden Leitung eingebaut werden. Die Leitung muss also getrennt und das Multimeter sozusagen als »Verbindungskabel« eingesetzt werden. Dazu sind Krokodilklemmen sehr praktisch. Wenn das Multimeter in Serie angeschlossen und die Amperemeter-Funktionalität ausgewählt sind, werden die gemessenen Ströme auf dem Display angezeigt. Damit kann man zum Beispiel den Strom zu einer LED messen, um sicherzustellen, dass er den maximalen Wert nicht überschreitet.

Das Oszilloskop

Ein nützliches und aufschlussreiches Werkzeug, das allerdings nicht sehr erschwinglich ist, ist das Oszilloskop. Ein Oszilloskop ist eine Art grafischer Voltmeter, mit dem Spannungen nach Zeit gemessen werden. So ist es möglich, den zeitlichen Verlauf einzel-

ner Spannungen anzuzeigen. Damit kann zum Beispiel die Breite von PWM-Pulsen gemessen werden (siehe Kapitel 3) oder die Ausgangsspannung verschiedener analoger Sensoren. Alle Funktionen des Voltmeters können damit übernommen werden, um z.B. zu überprüfen, ob Taster richtig schalten, Kommunikationsprotokolle richtig implementiert werden und Taktsignale für verschiedene Chips schnell genug sind. Ein Oszilloskop ist ein extrem wertvolles Werkzeug, um kompliziertere Schaltungen zu untersuchen und generell einen grafischen Eindruck von der Elektronik zu bekommen.

KAPITEL 3

Workshop LED-Licht

In diesem Kapitel:

- Erste Schritte
- Eine blinkende LED – das »Hello World« des Physical Computing

Nun soll also der erste Workshop beginnen, dessen Ziel es ist, eine in allen Farben des Regenbogens leuchtende LED-Lampe zu bauen. Natürlich geht es nicht darum festzulegen, welche Form oder Farbe sie bekommen soll. Vielmehr wird dieses Kapitel hoffentlich genügend Anleitung geben, um im Anschluss ein eigenes Licht programmieren zu können. Dieses kann aus einer oder mehreren Lichtquellen bestehen, die sich auch in ihrer Helligkeit verändern lassen, entweder durch Programmierung oder durch ein Steuerungselement wie einen Schalter oder Drehknopf. Zudem wird erklärt, wie man aus Rot, Grün und Blau Farben mischen kann, um den Raum auch mehrfarbig zu erhellen. Am Ende des Kapitels werden zusätzlich weitere Projekte beschrieben, die mit Anleitungen aus dem Internet nachgebaut werden können. Auf dem Weg durch das Kapitel werden die Grundlagen erläutert, die zum Programmieren eines Arduino nötig sind. Dieser Workshop richtet sich also auch an Leute, die soeben zum ersten Mal ein Arduino-Board angeschlossen haben.

Erste Schritte

Um dieses Kapitel durchzuarbeiten, sind zum ersten Mal in diesem Buch einige Bauteile nötig, nämlich

- vier LEDs (drei davon in Rot, Grün und Blau) und passende 100-Ohm-Widerstände,
- ein Taster,
- ein Schalter und
- ein Drehknopf.

Ein einfaches Arduino-Programm: Übersicht

Ein Arduino-Programm kann aus vielen miteinander verbundenen Dateien bestehen, hat jedoch mindestens zwei Teile: das Setup und die Hauptfunktion.

Setup

Der Setup-Teil des Programms ist gekennzeichnet durch die Funktion `setup()`:

```
void setup()
{
}
```

Zwischen die geschweiften Klammern werden nun alle Befehle gesetzt, die vor dem Start des Hauptprogramms zum Einrichten des Arduino benötigt werden, etwa die Festlegung einzelner Pins als Ein- oder Ausgang.

Die Setup-Routine wird nur ein einziges Mal ausgeführt, wenn das Board neu an eine Stromquelle (oder per USB an den Rechner) angeschlossen oder neuer Code hochgeladen wird.

Loop

Die Funktion `loop()` wird auch als *Hauptfunktion* bezeichnet. Von hier aus werden andere Bestandteile des Programms aufgerufen oder Befehle abgearbeitet. Wie der Name schon verrät, läuft `loop()` in einer Schleife, das heißt sie beginnt immer wieder von vorn, sobald sie durchlaufen wurde. Ganz analog zur `setup()`-Funktion sieht auch `loop` so aus:

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

Eine blinkende LED – das »Hello World« des Physical Computing

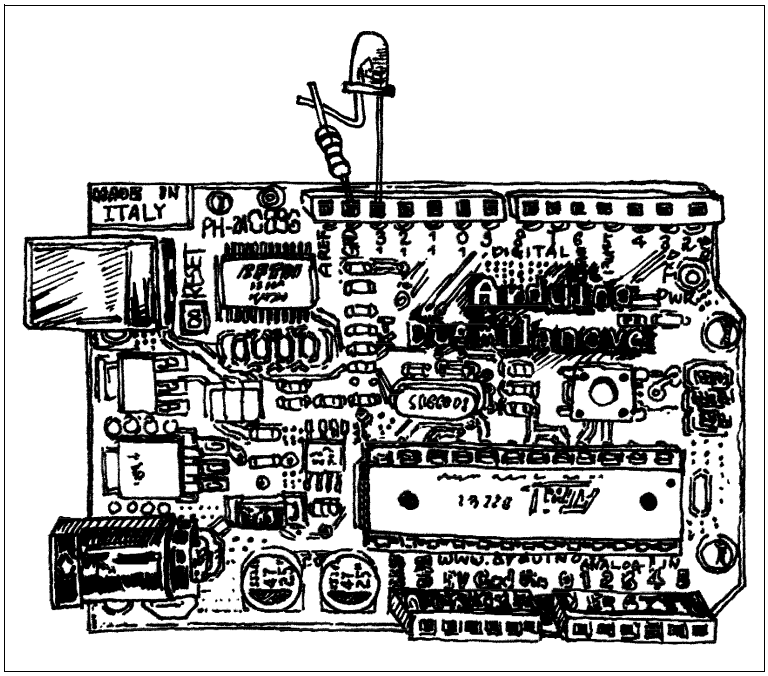
Beim Erlernen von Programmiersprachen ist es üblich, zuerst einen einfachen Text auf dem Bildschirm auszugeben, um über die Sprache einen Gruß an die Welt hinaus zu senden. Auf dem Arduino wird dieser Gruß mit Licht übermittelt, indem eine angeschlossene LED zum Blinken gebracht wird. LEDs sind Licht aussendende Dioden (light emitting diodes). Eine Diode ist ein Bauteil, das

Strom nur in eine Richtung durchlässt, bei einer LED wird dabei sichtbares oder unsichtbares Licht erzeugt.

Die Lichtfarbe einer LED hängt von ihrem Aufbau und den verwendeten Materialien ab. Lange Zeit war es nicht möglich, blaue und weiße LEDs herzustellen. In vielen älteren Geräten gibt es daher nur rote, grüne und gelbe LEDs. Durch neuere Technologien sind mittlerweile fast alle Farben herstellbar. Trotzdem hat sich eine Handvoll Farben etabliert. Mehr Informationen zu genauen Daten verschiedener LEDs finden Sie im Anhang.

Hardware

LEDs besitzen ein langes und eines kurzes Beinchen, die Pins: Anode und Kathode (kleine Merkhilfe: *k*urz = *K*athode). Sind die Pins bereits abgeschnitten worden, etwa um sie irgendwo zu verlöten, findet man die Kathode an der Seite, an der die LED leicht abgeflacht ist. Die LED wird mit der Anode, also der positiven Seite, mit dem 100-Ohm-Widerstand verbunden, und dieser wird am Arduino-Pin 13 angebracht, die negative Seite wird an der Masse (GND, zu englisch »ground«) angeschlossen.



◀ Abbildung 3-1
Arduino-Board

Die LED beginnt zu leuchten, wenn durch sie ein Strom von ca. 2 bis 20 mA fließt. Zu viel Strom ist schädlich für die LED und kann sie im schlimmsten Falle sofort zerstören. LEDs dürfen immer nur mit einem Vorwiderstand oder einer speziellen Schaltung betrieben werden, damit der Strom nicht zu groß wird.

- U = Spannung der Batterie/des Arduino-Boards (5 Volt)
- Uled = Flussspannung der LED (siehe Datenblatt, 3 Volt ist für die meisten LEDs ein guter Richtwert)
- I = der gewünschte Strom durch die LED (ein sicherer Wert ist 10 mA)
- R = der zu berechnende Widerstand
- R = (U-Uled) / I

Bei 5 Volt und einer LED mit einer Flussspannung von 3 Volt ergibt sich für 20 mA ein Vorwiderstand von 100 Ohm.

Programmierung

Zunächst wird Pin 13 mit einer Variablen verbunden. Das geschieht, um im weiteren Verlauf des Programms auf *ledPin1* zurückgreifen zu können. So kann schnell der Pin geändert werden, ohne dass das ganze Programm durchgearbeitet werden muss. Zudem steht so keine 13 im Programmverlauf, sondern eine verständliche Bezeichnung, die beim Lesen oder nachträglichen Bearbeiten des Codes behilflich ist.

Variablen können verschiedene Datentypen besitzen, je nachdem, welche Inhalte in ihnen gespeichert werden. In diesem Fall wird eine ganze Zahl (»Integer«) gespeichert, dem Variablennamen wird bei der Initialisierung also der Typ *int* vorangestellt.

Im Setup wird dieser Pin als Ausgang definiert. Im Hauptteil wird schließlich ein digitales Signal auf diesen Ausgang geschrieben. `HIGH` lässt dabei Strom durch den Ausgang fließen, während `LOW` diesen Stromfluss unterbricht. So leuchtet die LED oder erlischt. Dazwischen wird der Programmablauf mit dem Befehl `delay` für 1.000 Millisekunden, also eine Sekunde, unterbrochen – es wird gewartet.

Hinweis An jeder Stelle des Programms können Kommentare verfasst werden. Sie werden mit `//` oder `#` markiert und beim Übersetzen des Programms nicht berücksichtigt. Das erhöht die Verständlichkeit des Codes.

```
int ledPin1 = 13;         // LED an digitalen Pin 13 angeschlossen
void setup()
{
  pinMode(ledPin1, OUTPUT);  // setze digitalen Pin als Output
}
void loop()
{
  digitalWrite(ledPin1, HIGH);   // schalte LED ein
  delay(1000);                  // warte eine Sekunde
  digitalWrite(ledPin1, LOW);    // schalte LED aus
  delay(1000);                  // warte eine Sekunde
}
```

Aufgabe:

- Eine zweite LED anschließen und beide wechselnd blinken lassen.

LEDs über Schalter/Taster steuern

Die digitalen Pins können auch als Eingabepins definiert werden. In diesem Beispiel wird ein Schalter benutzt, um zwischen den beiden LEDs hin- und herzuschalten. Dabei wird die *if-else-Abfrage* eingeführt.

Taster und Schalter finden sich in jedem Haushalt: Entweder direkt in der Wand, um das Licht ein-zuschalten, oder an Geräten, um deren Funktionen einzustellen, zu starten oder zu stoppen.

Ein Schalter hat die zwei Stellungen `Ein` und `Aus` und verbleibt in der zuletzt vom Benutzer gewählten Stellung.

Im Unterschied dazu springt ein Taster nach dem Loslassen sofort wieder in den ungedrückten Zustand zurück. Er eignet sich also nicht, um eine Lampe dauerhaft einzuschalten. Für die Verwendung bei einer Klingel dagegen ist er ideal: Nach dem Betätigen soll sie wieder aufhören zu schellen, der Taster springt zurück.

Schalter und Taster besitzen mindestens zwei Anschlüsse. Es gibt sie aber in beliebig vielen Varianten:

- Mit drei Anschlüssen, als Umschalter (der mittlere Anschluss wird wahlweise mit dem einen oder dem anderen Pin verbunden)
- Mit mehreren Schaltern
- Mit mehr als zwei Schaltstellungen (damit kann man verschiedene Aktionen mit einem einzigen Schalter auswählen)

Abbildung 3-2 ▶
Das Schaltzeichen für einen Taster

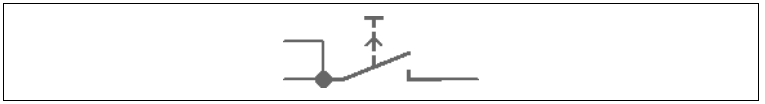
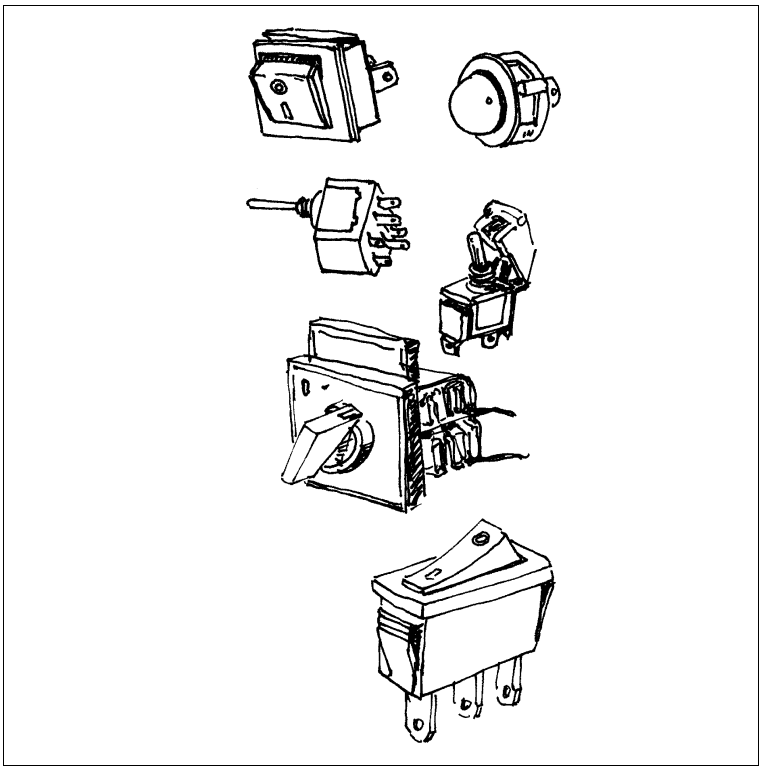


Abbildung 3-3 ▶
Das Schaltzeichen für einen Schalter

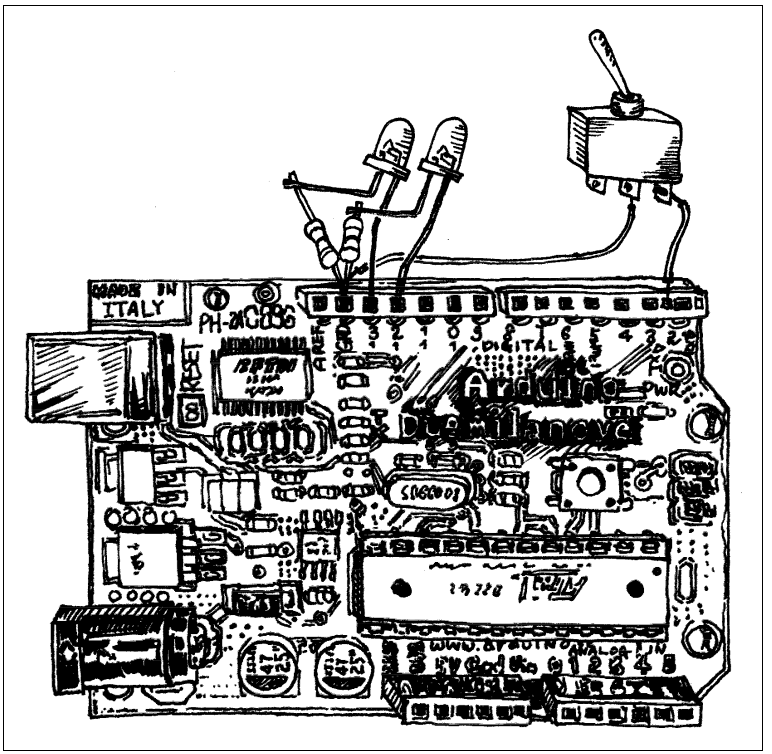
Abbildung 3-4 ▶
Verschiedene Schalter/Taster



Hardware

Um einen Schalter oder Taster mit dem Arduino abzufragen, wird die eine Seite des Bauteils mit einem IO-Pin (in diesem Fall Pin 2) und die andere mit GND verbunden.

Der Arduino erkennt an seinen IO-Pins hohe Spannungen (2–5 Volt) als eine 1 und kleinere Spannungen als eine 0. Ist der Taster/Schalter gedrückt, wird der PIN des Arduino mit GND verbunden, die gemessene Spannung ist also null.



◀ **Abbildung 3-5**
Schalteraufbau

Programmierung

Der Schalter wird, wie auch die LEDs, auf eine Variable gelegt. Im Setup wird er als Eingang definiert:

```
pinMode(schalter, INPUT);    // setze den digitalen Pin auf Input
```

Ist der Taster/Schalter nicht gedrückt, ist der Stromkreis unterbrochen. Der IO-Pin hat dann keine Verbindung, weder mit GND (0 Volt) noch mit VDD (5 Volt) – die vom Arduino gemessene Spannung ist dann von vielen Umweltfaktoren abhängig und schwankt zufällig hin und her. Um auch hier eine definierte Spannung an diesem PIN zu haben, wird ein sogenannter *Pull-up-Widerstand* eingesetzt, der den Spannungspegel nach oben zieht. Dieser kann entweder als richtiger Widerstand mit einem Wert von 1.000–100.000 verwendet werden, oder man verwendet im Arduino bereits integrierte Pull-up-Widerstände. Je größer der Widerstand, desto stromsparender, aber auch umso anfälliger für elektromagnetische Störungen wird unsere Schaltung.

Die internen Pull-up-Widerstände werden mit

```
digitalWrite(schalter, HIGH);
```

eingeschaltet.

Im Hauptteil wird durch `digitalRead` die Eingabe gelesen. Nun wird eine if-else-Abfrage benutzt. Sie steht für eine Verzweigung im Programmablauf: Wenn der Strom durch den Schalter fließt, ist das Eingangssignal mit GND verbunden, also `LOW`, und der zweite Programmblock wird ausgeführt, der durch die geschweiften Klammern begrenzt ist. Ansonsten liegt am Eingang durch den Pull-up ein `HIGH`-Signal vor, also wird der erste Programmblock ausgeführt.

Vergleiche in if-Abfragen werden dabei immer mit zwei Gleichheitszeichen vorgenommen. Wird das Gegenteil benötigt, steht `!=` für »ungleich«.

```
int ledPin1 = 13;     // LED an digitalen Pin 13 angeschlossen
int ledPin2 = 12;     // LED an digitalen Pin 12 angeschlossen
int schalter = 2;     // Schalter an digitalen Pin 2 angeschlossen
void setup()
{
  pinMode(ledPin1, OUTPUT);  // setze digitalen Pin als Output
  pinMode(ledPin2, OUTPUT);  // setze digitalen Pin als Output
  pinMode(schalter, INPUT);    // setze digitalen Pin auf Input
  digitalWrite(schalter, HIGH); // setze pullup-widerstand
}
void loop() {
  int val = digitalRead(schalter); // lies Input vom Schalter
  if (val == HIGH) {      // wenn der Wert von val gleich HIGH ist
    digitalWrite(ledPin1, HIGH);   // schaltet LED1 ein
    digitalWrite(ledPin2, LOW);    // schaltet LED2 aus
  }
  else {
    digitalWrite(ledPin1, LOW);
    digitalWrite(ledPin2, HIGH);
  }
}
```

Vier LEDs nacheinander blinken lassen

Schließt man mehr als zwei LEDs an den Arduino an, wird es schnell unübersichtlich: An jeder Stelle muss mehrfach der gleiche Code geschrieben werden. Muss dieser Code nachträglich verändert werden, treten schnell Probleme auf: Versteckt sich in einer Zeile ein Fehler, müssen gleich mehrere ähnliche Zeilen angepasst werden, um ihn zu korrigieren. Schnell wird dadurch auch eine der duplizierten Zeilen übersehen. Generell gilt: Je kleiner der Pro-

grammcode, desto einfacher ist er zu lesen und zu modifizieren, und umso geringer ist auch die Gefahr, dass sich Fehler einschleichen.

Im weiteren Verlauf dieses Beispiels wird erklärt, wie for-Schleifen benutzt werden.

Setup

Vier LEDs werden an die digitalen Pins 10, 11, 12, 13 angeschlossen, und ein Taster an Pin 2 (so wie der Schalter aus dem vorigen Beispiel).

Programmierung

Ein Array (also eine Tabelle) fasst mehrere Variablen in einer zusammen. Über sogenannte Indizes können die einzelnen Werte dann als Elemente angesprochen werden.

Ein Array wird ähnlich wie eine Variable initialisiert. Mit

```
int led[4] = { 10,11,12,13};
```

werden die vier Werte 10, 11, 12 und 13 auf ein Array mit vier Elementen gelegt. Da die Indizes bei 0 beginnen, ist der Wert von `led[0]` nun also 10, der von `led[1]` 11 und so weiter.

Nun können diese Werte mit einer Schleife durchlaufen (»iteriert«) werden. In diesem Fall wird `for()` zusammen mit einer Zählvariable `i` verwendet, die pro Schleifendurchlauf mit `i++` um eins erhöht wird.

Schleifen sind ein weiterer häufig benutzter Bestandteil von Programmiersprachen. Im Arduino-Hauptteil wird eine *loop()-Schleife* verwendet, die unendlich lange weiterläuft (sofern kein Programmabbruch festgelegt wird). Hinzu kommt nun die *for()-Schleife*, der drei Bestandteile hinzugefügt werden: Die Startbedingung (z. B. »beginn bei null«), die Endbedingung (z. B. »zähl bis vier«) und die Zählbedingung (z. B. »zähl immer eins dazu«).

Um Codezeilen zu sparen, wird die Zählvariable direkt in der Schleife initialisiert:

```
for (int i = 0; i<4; i++) {
   pinMode(led[i], OUTPUT);
 }
```

Das bedeutet: Definiere i = 0, solange i kleiner als 4 ist, und erhöhe i um 1 pro Schleifendurchlauf.

Innerhalb der Schleife wird jeder der zuvor eingetragenen Pins als Output festgelegt.

Funktionen

Um eine LED leuchten zu lassen, kann der Code der zweiten Übung benutzt werden. Allerdings wäre dieser nun pro Schritt vier statt bisher zwei Zeilen lang. Bei allen vier LEDs, die nacheinander leuchten sollen, wären das also 16 Zeilen und nicht mehr vier. Zeit für eine Vereinfachung!

Eine Funktion ist eine einzelne Einheit innerhalb eines Programms, die fast überall aufgerufen werden kann. Sie erledigt selbstständig eine Aufgabe und gibt, wenn gewünscht, einen Wert zurück. Je nach Typ dieses Werts werden Funktionen und Variablen, also Typen zugewiesen. Geben sie keinen Wert zurück, ist der Typ *void*. Zudem nimmt eine Funktion Werte an, mit denen sie arbeiten soll – sogenannte »Argumente«.

Um eine LED blinken zu lassen und die anderen auszuschalten, muss der Funktion mitgeteilt werden, um welche LED es sich handelt. Gleichzeitig ist die Aufgabe der Funktion lediglich die Ausgabe von Signalen. Sie gibt also keinen Wert zurück.

Der Code

```
void setLED(int ledNr) {
}
```

deklariert eine Funktion `setLED`, die keinen Wert zurückgibt (also `void`) und ein Argument vom Typ *int* annimmt. Das bedeutet, dass dieser Funktion ein Wert mit dem Namen `ledNr` mitgegeben wird, den diese Funktion nun verwenden kann. Dabei wird im Speicher eine Kopie der Variablen angelegt und während der Funktion verwendet. So kann die Funktion zwar mit dem Wert arbeiten, verändert ihn aber nicht dauerhaft.

Nun wird jede LED durchlaufen und auf `LOW` gesetzt, sofern sie nicht diejenige ist, die angeschaltet werden soll:

```
for (int i = 0;i<4;i++) {
   if (i == ledNr) {
    digitalWrite(led[i], HIGH);
   }
   else {
     digitalWrite(led[i], LOW);
   }
```

Damit alles nach einem Durchlauf wieder von vorn beginnt, wird nun eine weitere Funktion benötigt, die einen Zähler auf 0 zurücksetzt, sobald sie 4 erreicht. Diese Funktion muss den Zähler, mit dem sie arbeitet, wieder zurückgeben, sie ist also vom Typ *int*.

```
int setCount(int count) {
  if (count == 3) {
    count = 0;
  }
  else {
    count++;
  }
  return count;
}
```

Hinweis

Noch einfacher ließe sich das Problem mit dem »Modulo« lösen, dem Rest, der beim Teilen durch eine Zahl entsteht. Erhöht man pro loop()-Durchlauf die Variable count um eins, kann man `setLED` jederzeit mit `setLed(count mod 4)` oder auch `setLed(count % 4)` aufrufen. Es wird immer eine Zahl zwischen 0 und 3 übergeben.

Da der Taster sehr schnell abgefragt wird, soll sein Impuls nur dann zu einem Lichtwechsel führen, wenn er gerade gedrückt wurde. Dazu wird eine Variable `oldVal` eingeführt, auf die pro loop()-Durchlauf der aktuelle Wert des Eingangs geschrieben wird. Nur wenn diese sich im Vergleich zum vorigen Durchlauf ändert, wird die Funktion `setLED` wirklich aufgerufen.

Das sieht nun also wie folgt aus:

```
int led[4] = { 10,11,12,13};
int oldVal = 0;
int counter = 0;
int taster = 2;
void setup() {
   for (int i = 0; i<4; i++) {
   pinMode(led[i], OUTPUT);
 }
void setLED(int ledNr) {
   for (int i = 0;i<4;i++) {
     if (i == ledNr) {
       digitalWrite(led[i], HIGH);
     }
     else {
       digitalWrite(led[i], LOW);
     }
  }
}
```

```
int setCounter(int counter) {
  if (counter == 3) {
    counter = 0;
  }
  else {
    counter++;
  }
  return counter;
}
void loop()
{
  int val = digitalRead(taster);     // lies Input vom Taster
  if (val != oldVal && val == HIGH) {
    counter = setCounter();
    setLED(counter);
    delay(10);    // warte ein wenig
  }
  oldVal = val;
}
```

Die Debounce-Bibliothek

Möchte man den Taster als Umschalter verwenden, kann es oft zu mehrfachen Betätigungen kommen, selbst wenn man eine Abfrage wie die obige einbaut. Gegen dieses sogenannte »Prellen« hilft die Debounce-Bibliothek, mit der wir jetzt einen kurzen Ausflug in die Welt der Libraries machen.

Viele Anwendungen rund um Arduino sind im Prinzip recht einfach, benötigen aber hohen Programmieraufwand. Zum Beispiel möchte man einen bestimmten Sensor eigentlich einfach nur auslesen oder einen Motor nur mit einer bestimmten Geschwindigkeit betreiben. Die Technik dahinter ist aber recht komplex und benötigt verschiedene Signale und Voreinstellungen vom Arduino.

Bibliotheken fassen diese Funktionen zusammen. Denn wer einmal die Arbeit gemacht hat, kann sie auch anderen zur Verfügung stellen und sie über eine sogenannte API benutzen lassen. Der Begriff API steht für »Application Programming Interface« und ist quasi das Tor zur Bibliothek: ein Satz von Funktionen, mit denen die komplexeren Funktionalitäten der Bibliothek verwendet werden können. Für den Arduino gibt es eine ganze Reihe von Bibliotheken, die zum Teil die Programmierung einfacher machen, so wie die Debounce-Bibliothek, um die es hier geht. Wichtiger sind aber diejenigen Bibliotheken, die Bauteile und Geräte ansteuern können. Auf der Arduino-Webseite finden Sie Listen von Bibliotheken, genauso wie im Verzeichnis von *http://www.freeduino.org*.

Unter Prellen oder »Bouncing« versteht man ein mechanisches Problem, das durch die Eigenschaft von Schaltern und Tastern hervorgerufen wird: Weil in diesen Bauteilen federnde Effekte auftreten, öffnen und schließen sie sich beim Betätigen und Loslassen mehrmals, statt dass sofort ein elektrischer Kontakt zustande kommt. Hiergegen hilft nur, dass nach einem Kontakt zunächst jede weitere Eingabe vom Taster/Schalter für einige Millisekunden gesperrt wird.

Bibliotheken werden mit dem Befehl `include` in den aktuellen Sketch eingebunden. Damit sie beim Kompilieren auch gefunden werden, muss der komplette Ordner mit der Bibliothek ins Arduino-Verzeichnis unter *hardware\libraries* kopiert werden. Nun kann die Bibliothek auch in der Programmierumgebung über SKETCH → IMPORT LIBRARIES eingefügt werden. Hat man die Debounce-Bibliothek von *http://www.arduino.cc/playground/Code/Debounce* heruntergeladen, entpackt und in das entsprechende Verzeichnis kopiert, bindet man sie mit

```
#import <debounce.h>
```

in den Sketch ein. Nun stehen drei Funktionen zur Verfügung, die es bisher in der Arduino-Programmiersprache noch nicht gab: `read()`, `update()` und `write()`. Diese Funktionen gehören zu einem Objekt, das die aktive Bibliothek repräsentiert. Objekte sind Daten- und Funktionssammlungen. Das heißt, sie beherbergen neben den API-Funktionen auch weitere Funktionalitäten und Datenstrukturen. Man kann ein Objekt also mit einer Küche vergleichen, in der Essen vorbereitet wird. Dabei werden Herde und Öfen bedient, Teig vorgehalten und einzelne Gerichte gestapelt, bis eine Bestellung beisammen ist. Über eine oder mehrere Luken, Türen oder Zettelsysteme spricht die Bedienung mit dieser Küche und tauscht Informationen oder Nahrungsmittel aus. Dabei bleibt natürlich auch nicht alles zwischen Küche und Bedienung. Vielmehr gibt es einen Hinterausgang, über den neue Rohstoffe bestellt und geliefert werden.

Ein Objekt ist dabei immer die Instanz einer sogenannten Klasse. Das bedeutet, dass die importierte Bibliothek auch mehrfach verwendet werden kann, beispielsweise wenn man mehrere gleiche Geräte an einen Arduino hängen möchte. Klassen sind die eigentlichen Programmierkonstrukte, die im laufenden Betrieb zu Objekten werden. Sie verfügen zusätzlich über besondere Funktionen.

Der Konstruktor wird zum Beispiel aufgerufen, wenn das Objekt instanziiert, also erzeugt wird. Im Falle des Debouncers wird nun ein Objekt erstellt, dem 20 Millisekunden Debounce-Zeit eingeräumt werden:

```
// verbinde Taster mit Pin 5
int taster = 5;
// erschaffe ein neues Objekt vom Typ "debounce" mit 20
// Millisekunden Debounce-Zeit, verbunden an Pin 5
Debounce debouncer = Debounce( 20 , taster );
```

Im eigentlichen Programm können nun die oben erwähnten Funktionen aufgerufen werden. Die Funktion `update()` prüft nach, ob sich der Status geändert hat, und vermeldet das Ergebnis mit dem Rückgabewert `TRUE` oder `FALSE`. Mit `read()` kann dann der derzeitige Pin-Status gelesen werden, man erhält also `HIGH` oder `LOW` zurück. Um nun also die Informationen des Tasters korrekt zu lesen, benötigt man

```
// Debouncer-Status updaten
debouncer.update();
// Wert über den Debouncer auslesen
int tasterVal = debouncer.read();
```

Die anderen Funktionen in der Biblitohek sind `write()` und `interval()`. `write()` erlaubt das Senden eines digitalen Signals über den Debouncer. Da der Pin schon festgelegt ist, wird als Argument nur das Signal, also `HIGH` oder `LOW`, benötigt. Mit `interval()` lässt sich schließlich das Debounce-Intervall ändern, ohne dass das Objekt neu instanziiert werden muss.

Das gesamte Programm mit einer LED und der Debouncer-Bibliothek sieht also so aus:

```
#import <debounce.h>
// verbinde Taster mit Pin 5
int taster = 5;
// erschaffe ein neues Objekt vom Typ "debounce" mit 20
// Millisekunden Debounce-Zeit, verbunden an Pin 5
Debounce debouncer = Debounce( 20 , taster );
int ledPin1 = 13;          // LED an digitalen Pin 13
                             // angeschlossen
int schalter = 2;          // Schalter an digitalen Pin 2
                           // angeschlossen
void setup()
{
  pinMode(ledPin1, OUTPUT);  // setze digitalen Pin als Output
  pinMode(schalter, INPUT);    // setze digitalen Pin auf Input
  digitalWrite(schalter, HIGH);
}
```

```
void loop() {
  // Debouncer-Status updaten
  debouncer.update();
  // Wert über den Debouncer auslesen
  int tasterVal = debouncer.read();
  if (tasterVal == HIGH) {      // wenn der Wert von tasterVal
                                // gleich HIGH ist
    digitalWrite(ledPin1, HIGH);   // schaltet LED1 ein
    digitalWrite(ledPin2, LOW);    // schaltet LED2 aus
  }
  else {
    digitalWrite(ledPin1, LOW);
    digitalWrite(ledPin2, HIGH);
  }
}
```

Aufgabe:

- Einen »Funkenhimmel« bauen, der so viele LEDs wie möglich verwendet und sie zufällig blinken lässt.

Hinweis Die Funktion random(13) gibt bei jedem Aufruf eine neue zufällige Zahl zwischen 0 und 12 zurück.

LEDs mit Pulsweitenmodulation verschieden hell leuchten lassen

Bis zu diesem Punkt gab es für die LEDs nur die zwei Zustände `HIGH` und `LOW`, also An und Aus. Das liegt daran, dass über den Arduino keine unterschiedlichen Stromstärken ausgegeben werden können. Das menschliche Auge hingegen lässt sich recht leicht austricksen. Da es nur 25 Bilder pro Sekunde wahrnehmen kann, kommt Flackern in Frequenzen, die weit darüber liegen, nicht mehr als solches an. Vielmehr erscheint uns das Licht als heller oder dunkler, je nach Häufigkeit des Flackerns. Der Arduino hat diese sogenannte *Pulsweitenmodulation* (PWM) auf sechs Pins direkt eingebaut. Diese lassen sich nicht nur mit `digitalWrite()` ansprechen, sondern auch mit der Funktion `analogWrite()`, die nicht nur die Signale `LOW` und `HIGH` als Argumente annimmt, sondern 8 Bit verarbeiten kann, also 255 unterschiedliche Werte.

Hardware

Eine LED an Pin 10 anschließen.

Software

Die LED wird mit Pin 10 verbunden und als `OUTPUT` definiert. Das Programm soll nun von 0 bis 255 zählen und pro Schritt ein »stärkeres«, also schneller gepulstes Signal an den Ausgang senden. Anschließend soll der Zähler wieder von 255 bis 0 laufen.

Dafür werden ein Zähler (`counter`) und ein Vorzeichenmarker (`changeMarker`) initialisiert:

```
int counter = 0;
int changeMarker = 1;
```

Die Funktion `changeCounter` addiert nun pro Aufruf einmal `changeMarker` zum `counter`. Erreicht der Counter den Wert 255 oder 0, wird der `changeMarker` auf `-1` bzw. `1` gesetzt:

```
int ledPin = 10
void setup() {
  pinMode(ledPin, OUTPUT);
}
int counter = 0;
int changeMarker = 1;
int changeCounter() {
  if (counter == 255) {
    changeMarker = -1;
  }
  if (counter == 0) {
    changeMarker = 1;
  }
  counter = counter + changeMarker;
  return counter;
}
```

Nun steht einem pulsierenden Licht nichts mehr im Wege:

```
void loop()
{
  counter = changeCounter();
  analogWrite(led, counter);
  delay(10);    // ein bisschen warten ...
}
```

Beim genauen Hinsehen stellt man fest, dass das Licht nicht gleichmäßig pulsiert. Das liegt daran, dass das Auge das Licht nicht linear wahrnimmt, also geradlinig, sondern als logarithmische Funktion. Wenn die LED fast aus ist, sieht man kleinere Veränderungen sehr stark; ist die LED fast ganz eingeschaltet, ändert sich praktisch gar nichts mehr. Um diesem Umstand entgegenzuwirken, kann `changeCounter()` zu einer aufwendigeren Version ausgebaut werden, sodass der Logarithmus korrekt errechnet wird. Viel einfacher ist

jedoch eine Wertetabelle, die uns diese Übersetzung abnimmt, sie gilt für alle Projekte mit gedimmten LEDs. Für diesen Zweck reichen 64 Werte aus, da die PWM mit ihren 256 Stufen keine weitere Verfeinerung erlaubt:

```
int loga[64] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
16, 18, 20, 22, 25, 28, 30, 33, 36, 39, 42, 46, 49, 53, 56, 60,
64, 68, 72, 77, 81, 86, 90, 95, 100, 105, 110, 116, 121, 127, 132,
138, 144, 150, 156, 163, 169, 176, 182, 189, 196, 203, 210, 218,
225, 233, 240, 248, 255};
```

Das Programm muss nun an zwei Stellen geändert werden. In der Hauptfunktion wird

```
analogWrite(led, counter)
```

ersetzt durch

```
analogWrite(led, loga[counter]);
```

Schließt man vier LEDs an, verändert sich im Prinzip nicht viel. Um die LEDs in einer Welle pulsieren zu lassen, benötigt man vier Counter und vier Marker:

```
int counter[4] = {0,32,63,32};
int changeMarker[4] = {1, 1, -1, -1};
```

Das wird nun auch in die Funktion `changeCounter()` eingebaut, die nun in der Lage ist, einen der vier Counter bzw. Marker zu verändern. Zudem werden die ersten beiden Zeilen geändert, sodass die Funktion nun wie folgt aussieht:

```
void changeCounter(int i) {
  if (counter[i] == 63) {
    changeMarker[i] = -1;
  }
  if (counter[i] == 0) {
    changeMarker[i] = 1;
  }
  counter[i] = counter[i] + changeMarker[i];
}
```

So wird nun auch die Hauptfunktion verändert:

```
void loop()
{
  for (int i = 0;i<4;i++)
  {
    changeCounter(i); // verändere den Counter der LED i
    analogWrite(led[i], loga[counter[i]]);
    //  setze die neue Helligkeit der LED i
  }
  delay(10);    // ein bisschen warten ...
}
```

Aufgabe:

- Den Taster wieder anschließen und die Lichtstärke damit regulieren lassen.

Mach es bunt

Nun kann schon die erste Lampe gebaut werden, die mit Farbverläufen eine gemütliche Atmosphäre im Raum schafft. Dazu werden drei LEDs in den Grundfarben Rot, Grün und Blau benötigt. Mit diesen Farben lassen sich durch Mischung alle weiteren Farben des sichtbaren Spektrums erzeugen, indem man jede LED mit einer entsprechenden Helligkeit pulst. Die folgende Lampe wählt zufällig Farben aus und dimmt sich langsam von der einen zur anderen und weiter zur nächsten.

Hardware

Drei LEDs in den Grundfarben Rot, Grün und Blau werden über einen passenden Widerstand (100 Ohm) mit dem Arduino verbunden. Welcher Widerstand genau benötigt wird und mit welcher Eingangsspannung und Stromstärke die LEDs betrieben werden, hängt von der Bauart ab. Genauere Daten können Sie dem Datenblatt entnehmen. Es gibt auch fertige RGB-LEDs, die mit sechs Beinen ausgestattet sind und somit auf die gleiche Weise angeschlossen werden wie drei einzelne. Je nach Stromstärke müssen die Leuchtdioden an einen anderen Stromkreis angeschlossen werden, besonders wenn eine stärkere Leistung erwartet wird. Am Ende dieses Kapitels wird unter dem Stichwort »Mehr Power« erläutert, wie man das zum Beispiel mit Transistoren machen kann. Das folgende Programm geht davon aus, dass die rote LED an Pin 9, die grüne an 10 und die blaue an 11 angeschlossen ist.

Programmierung

Wie weiter oben beschrieben, werden drei LEDs in einer Tabelle initialisiert und mit den passenden Pins verbunden. Auch die Logarithmustabelle wird benötigt. Zudem braucht man zwei Tabellen, die Quell- und Zielfarbwert speichern.

```
// PINs für die RGB-LED
int ledPin[3] = { 9, 10, 11};
// Startfarbe
```

```
int sourceValue[3] = { 0, 0, 0};
// Zielfarbe
int targetValue[3] = { 0, 0, 0};
// Überblendposition
int currentPos = 0;
```

In der Setup-Funktion werden die beiden Farben mit zufälligen Werten initialisiert und die verwendeten PINs als Ausgang geschaltet.

```
void setup()
{
  for (int i = 0; i < 3; i++) {
    // Startfarbe zufällig wählen
    sourceValue[i] = random(64);
    // Zielfarbe zufällig wählen
    targetValue[i] = random(64);
    // LED-Pin = Ausgang
    pinMode(ledPin[i], OUTPUT);
  }
}
```

currentPos wird jetzt der Mittelwert nach der Formel *farbe* = *start* × *(128-currentPos)* + *ende* × *currentPos* auf ihre entsprechende LED geschrieben und die loop()-Funktion abgeschlossen:

```
void loop()
{
  // für alle drei Grundfarben Mittelwert an der
  // Stelle currentPos berechnen und ausgeben
  for (int i = 0; i < 3; i++) {
    int helligkeit = (sourceValue[i] * (128 - currentPos) +
                      targetValue[i] * currentPos)/128;
    analogWrite(ledPin[i],helligkeit);
  }
  currentPos++;
  if (currentPos > 128) {
    for (int i = 0; i < 3; i++) {
      sourceValue[i] = targetValue[i];
      targetValue[i] = random(64);
    }
    currentPos = 0;
  }
  // 10ms warten.
  delay(10);
}
```

Und fertig ist die Lampe! Nun kann damit experimentiert werden. Mit Sicherheit finden sich viele Möglichkeiten, dieses Projekt zu erweitern und anzupassen. Zum Beispiel könnte man eine weitere RGB-LED anschließen und die aktuelle Farbe von einer zur anderen wandern lassen.

Helligkeit mit einem Drehknopf steuern

Zum Abschluss dieses Workshops soll nun die Helligkeit einer LED mit einem Drehknopf gesteuert werden. Dazu wird ein sogenanntes Potentiometer verwendet. Es besteht aus einem Metallstift, der auf einer Fläche angebracht ist. Wird diese Fläche durch Drehen entlang des Stiftes bewegt, verändert sich der Widerstand und es fließt weniger Spannung an den Ausgang. Ein Potentiometer ist somit ein sogenannter resistiver Sensor. Mehr Informationen über resistive Sensoren finden Sie auch in Kapitel 7.

Die Messung erfolgt über den analogen Eingang, wobei das Signal niemals genau ist. Abweichungen im Wert müssen bei der Programmierung berücksichtigt werden. Somit können nicht die vollen 1.024 Stufen ausgeschöpft werden, was angesichts von 256 möglichen Helligkeitsstufen bei den pulsweitenmodulierten Digitalausgängen allerdings kein Problem darstellt, zumal ohnehin nur die 64 Stufen aus der Tabelle verwendet werden sollen.

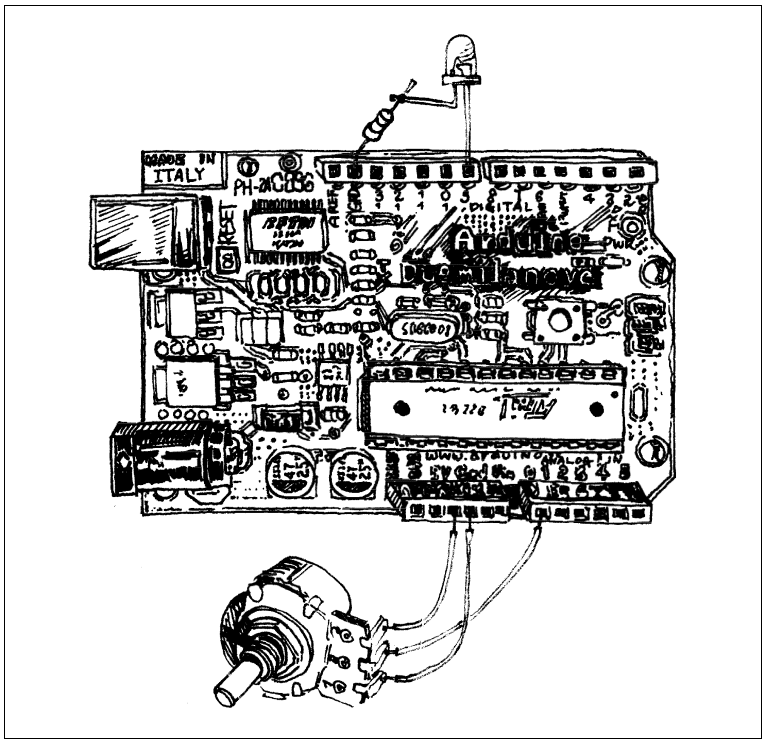
Hardware

An den Arduino wird zunächst eine LED an einen PWM-fähigen Digitalausgang angeschlossen, in diesem Fall Ausgang 9. Zudem wird ein Potentiometer mit dem mittleren Pin an den analogen Eingang 0 angeschlossen, die anderen beiden Pins an GND und den 5-Volt-Anschluss. Dabei kommt es darauf an, wie der Knopf gedreht werden soll: Je nachdem, wie der Strom fließt (von links nach rechts oder umgekehrt), muss auch gedreht werden, um die Spannung zu erhöhen oder zu senken.

Programmierung

Zunächst werden die beiden Pins festgelegt und im Setup als Ein- bzw. Ausgang definiert. Zudem wird wieder die Tabelle für die Helligkeitsstufen verwendet.

```
int potPin = 0;   // lege potPin als Pin 0 fest
int ledPin = 9;   // lege ledPin als Pin 9 fest
int potiVal = 0;  // eine Variable, um den Input zu speichern
int loga[64] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
16, 18, 20, 22, 25, 28, 30, 33, 36, 39, 42, 46, 49, 53, 56, 60,
64, 68, 72, 77, 81, 86, 90, 95, 100, 105, 110, 116, 121, 127, 132,
138, 144, 150, 156, 163, 169, 176, 182, 189, 196, 203, 210, 218,
225, 233, 240, 248, 255};
void setup() {
  pinMode(ledPin, OUTPUT);  // initialisiere ledPin als Output
}
```



◀ **Abbildung 3-6**
Aufbau für PWM-LE

Nun können der analoge Eingang gelesen und das Signal gespeichert werden. Anschließend werden das Signal durch 16 geteilt und der Ausgangswert anhand des oben erklärten Sinus berechnet. Schließlich wird der entsprechende Wert in der Logarithmustabelle auf den digitalen Ausgang geschrieben, die LED wird nun vom Potentiometer gesteuert.

```
void loop()
{
  int potiVal = analogRead(potPin); // lies analoges
                                    // Eingangssignal
  potiVal = potiVal / 16; // teile den Signalwert durch 16
  analogWrite(led, loga[potiVal]);  // schreibe den Tabellenwert
                                    // auf den Ausgang
  delay(10);
}
```

Flower Power

Nicht nur die Helligkeit kann mit einem Potentiometer geregelt werden, sondern auch die Wunschfarbe, passend zum Abendkleid.

Dazu verwenden wir den Quellcode von »Mach es bunt« in diesem Kapitel und verändern nur die Funktion `loop()`, sodass abhängig von der Reglerstellung eine Farbe ausgewählt wird. Nun gibt es keinen einfachen Weg durch alle Farben, es muss also ein bestimmter Verlauf einprogrammiert werden. Die 1.024, die maximal vom Potentiometer zurückgeliefert werden, teilen wir dazu in vier Teile auf. Im ersten Teil dimmen wir von Rot nach Grün, im zweiten Teil von Grün nach Blau, im dritten von Blau nach Rot, und mit dem verbleibenden Teil dimmen wir nach Weiß. Durch die Logarithmustabelle benötigen wir Werte von 0–63 für jede Farbe, wir können den Potentiometerwert also noch durch 4 teilen, um in jedem Teilabschnitt 64 Punkte zu haben.

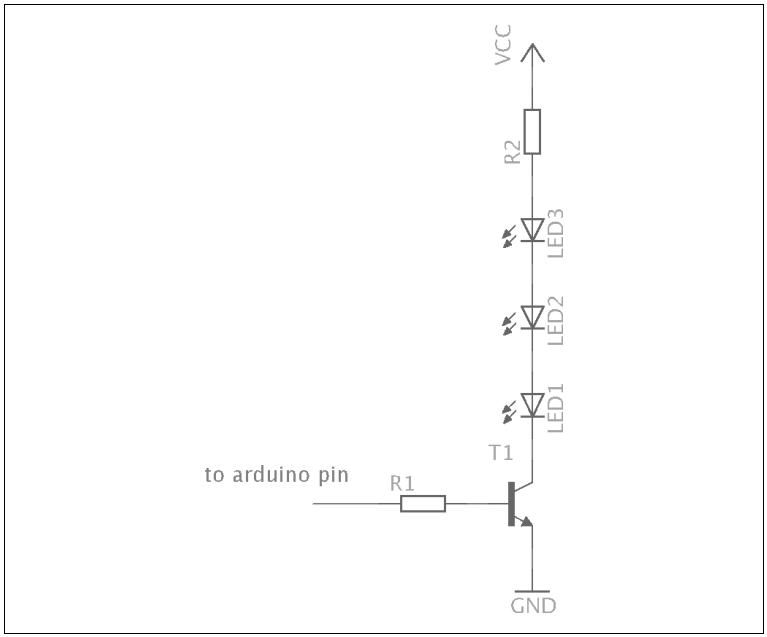
```
void loop() {
  int potiVal = analogRead(potPin) / 4; // lies analoges
                                         // Eingangssignal

  if (potiVal < 64)
  {                                     // fade Rot zu Grün
    analogWrite(ledPin[0], loga[63-potiVal]);// Rot
    analogWrite(ledPin[1], loga[potiVal]);// Grün
    analogWrite(ledPin[2], loga[0]);// Blau
  }
  else if (potiVal < 128)
  {                                     // fade Grün zu Blau
    analogWrite(ledPin[0], loga[0]);// Rot
    analogWrite(ledPin[1], loga[127 - potiVal)]);// Grün
    analogWrite(ledPin[2], loga[potiVal - 64]);// Blau
  }
  else if (potiVal < 192)
  {                                     // fade Blau zu Rot
    analogWrite(ledPin[0], loga[potiVal - 128]);// Rot
    analogWrite(ledPin[1], loga[0]);// Grün
    analogWrite(ledPin[2], loga[191 - potiVal]);// Blau
  }
  else
  {                                     // fade Rot zu Weiß
    analogWrite(ledPin[0], loga[63]);       // Rot
    analogWrite(ledPin[1], loga[255-potiVal]);// Grün
    analogWrite(ledPin[2], loga[255-potiVal]);// Blau
  }
}
```

Mehr Power

Bis jetzt wurden nur sehr kleine LEDs mit dem Arduino gesteuert, ein Pin kann nur ca. 20 mA Strom liefern. Bei einer 3-Volt-LED entspricht das 0,06 Watt. Damit bekommt man natürlich kein Zimmer erleuchtet.

Für größere LED können Transistoren oder Mosfets (Metalloxid-Halbleiter-Feldeffekttransistoren) verwendet werden, die schnell genug sind, um auch PWM für das Dimmen der LEDs zu ermöglichen. Transistoren sind in der Lage, mit einem kleinen Signal auf der einen Seite einen großen Stromkreis auf der anderen zu schalten.

Transistoren haben ihren Aufschwung in den 1960er Jahren erlebt, als sie zum ersten Mal in Form von Feldeffekttransistoren auf Galliumarsenid praktikabel genug wurden. Vorher hatte man Computer mit Vakuumröhren gebaut, die man günstig herstellen konnte und die das einzige Schaltelement waren, das auch schnell genug für diese Anforderungen war. Ein Transistor besteht aus drei Anschlüssen: Basis, Emitter und Kollektor. Transistoren gibt es in den beiden Polungen NPN und PNP. Ein NPN-Transistor als Schalter ist am besten für die negative Seite der Last geeignet (LED, Lampe, Motor usw.), PNP für die positive. Mehr Informationenen zu Relais und größeren Stromstärken findet man unter anderem auf der Arduino-Seite unter: *http://www.arduino.cc/playground/uploads/Learning/relays.pdf* sowie bei der New York University unter *http://itp.nyu.edu/physcomp/Tutorials/HighCurrentLoads*.



◀ Abbildung 3-7
Transistorausgangsschaltung

Gesteuert wird der Transistor über den Stromfluss, der in die Basis hineinfließt. Dabei muss der Basisanschluss mindestens 0,7 Volt

höher liegen als der Emitter. Dazu wird die Basis über einen Widerstand von 1.000–10.000 mit dem Ausgangspin des Arduino angeschlossen. Der Emitter wird direkt mit GND verbunden, und an den Kollektor kommt dann die eine Seite der Last, die andere Seite wird mit 5V verbunden. Das Schöne an dieser Schaltung ist, dass die Spannung auch viel größer sein kann, ohne dem Arduino zu schaden (das Maximum entnehmen Sie bitte dem Datenblatt des Transistors). Dadurch kann man viele LEDs in einer Reihe verbinden oder besonders starke LEDs verwenden, um den Raum hell zu erleuchten.

Und nun wird losgebastelt

Nun sollte es möglich sein, eine Lampe zu basteln. In den letzten Jahren sind LEDs immer günstiger und auch heller geworden, sodass inzwischen auch Lampen erschwinglich sind, die einen ganzen Raum erleuchten können. Interessant ist das insbesondere, wenn RGB-LEDs verwendet werden, die mit der oben gezeigten Logarithmustabelle immerhin 262.144 Farben darstellen können. Um noch weichere Farbübergänge zu erzeugen, müsste eine PWM von mehr als 8 Bit verwendet werden. Ein weiterer Vorteil von LEDs ist ihre Größe: Glühbirnen oder Leuchtstofflampen benötigen recht viel Platz, während Leuchtdioden sehr klein sind und in Tischtennisbälle, kleine Kästen, Flaschen und allerhand andere Behältnisse passen. Doch Vorsicht: Auch LEDs können je nach Stärke sehr heiß werden und müssen womöglich auf einen Kühlkörper aufgebracht werden (die gibt es inzwischen auch im Fachhandel, zum Beispiel in Onlineshops, die LEDs verkaufen). Man kann aber auch einen CPU-Kühler aus einem alten PC verbauen. Wichtig ist dabei allerdings, dass der Körper auch ausreichend groß ist und man Wärmeleitpaste verwendet.

Mit den sechs PWM-fähigen Pins des Arduino können zwei RGB-LEDs angeschlossen und entsprechend gepulst werden. Bringt man diese Lichter in entsprechender Stärke an zwei Enden des Raumes an oder lässt sie in entgegengesetzte Richtungen entlang einer Wand leuchten, ergeben sich schöne und stimmungsvolle Effekte. Diese Lampe lässt sich natürlich auch mit weiteren Sensoren ausstatten. Möglich ist zum Beispiel ein Mikrofon, das an einen Tiefpassfilter angeschlossen wird und die Lampe pulsen lässt, wenn ein Bass ertönt. Oder man nimmt einen passenden Gassensor, der das Licht melden lässt, wenn die Konzentration eines bestimmten Stoffes in der Luft zu hoch wird. Eine entsprechende Anleitung finden

Sie z.B. unter *http://www.instructables.com/id/How-To-Smell-Pollutants/?ALLSTEPS*.

Ein anderes mögliches Projekt könnte ein Lichtwecker sein, wie er für viel Geld auch im Handel erhältlich ist. Dabei wird eine Lampe ab einer festgelegten Uhrzeit allmählich immer heller, um Dämmerung und Sonnenaufgang zu simulieren. Einen solchen Wecker hat zum Beispiel Mark Ivey gebaut und unter *http://zovirl.com/2008/12/11/arduino-prototype-for-a-sunrise-alarm/* erläutert.

Ein gelungenes Beispiel ist die Milk Lamp von David Hayward, die aus einer Reihe an der Decke hängender Milchlampen mit weißen LEDs darin besteht. Beim Dimmen werden diese nicht wie gewohnt heller und dunkler geregelt, sondern sie werden der Reihe nach ein- und ausgeschaltet. Eine genaue Beschreibung des Projekts, das mithilfe der Informationen aus diesem Kapitel auch nachprogrammiert werden kann, finden Sie unter *http://functional-autonomy.net/blog/?p=442*.

KAPITEL 4

LEDs für Fortgeschrittene

In diesem Kapitel:

- LED-Matrix
- Animationen
- Interrupts
- Tamagotchi
- Brainwave und Biofeedback

Nun soll mit LEDs ein wenig mehr passieren, als nur Licht zu machen. In diesem Kapitel wird es zunächst darum gehen, viele LEDs zu steuern, wozu diese in einer sogenannten Matrix verschaltet werden. Damit soll dann eine kleine Art Tamagotchi selbst gebaut werden. Den Abschluss macht eine Gehirnwellenmaschine, die mit blinkendem Licht und Ton für Entspannung im Gehirn des Benutzers sorgen soll.

LED-Matrix

Reichen die 14 digitalen Pins nicht aus, gibt es die Möglichkeit, mit einer *Matrix* zu arbeiten. Dabei werden Reihen und Zeilen jeweils an einen Pin angeschlossen und die einzelnen Elemente der Matrix über eine Kombination der beiden angesprochen. Die einfachste Darstellungsform ist natürlich, wenn diese Matrizes auch als solche sichtbar sind, die LEDs also in Rechteckform angeordnet sind. Für 7x5 Bildpunkte benötigt man zwölf digitale Pins; damit kann man kleine Piktogramme oder Laufschrift anzeigen. Oder man programmiert kleine Spiele, die nicht mehr als diese 35 Punkte benötigen, wie das bekannte Schlangenspiel, in dem eine sich bewegende Schlange aus Punkten größer wird, wenn sie andere leuchtende Bildpunkte auffrisst. Die wohl größte Matrix dieser Art wurde beim Blinkenlights-Projekt konstruiert, das schon in Kapitel 1 erwähnt wurde: 18x8 Fenster eines Hochhauses dienten dabei als Bildpunkte eines Displays. Dort wurden kleine Animationen angezeigt, die man mit einem eigenen Programm erschaffen und hochladen konnte. Zudem war es möglich, mit einem Handy auf einer Nummer anzurufen und die Paddel eines Pong-Spiels mit den Tastentö-

nen zu steuern. In weiteren Varianten wurde das Projekt sogar noch ausgebaut und 2008 wurde das Rathaus von Toronto in ein Display von insgesamt 960 Bildpunkten verwandelt.

Im Alltag findet man solche LED-Matrizen zum Beispiel in den Laufschrift-Displays, mit denen kleinere Geschäfte für aktuelle Angebote werben. Es gibt aber zum Beispiel auch Gürtelschnallen, auf denen man Laufschriften anzeigen kann. Diese können meist direkt programmiert werden, es spricht aber auch nichts dagegen, sie an einen Arduino anzuschließen. Neben kleinen Blinkprojekten kann man solche Displays auch verwenden, um Informationen darzustellen. Oft reichen die im Vergleich zu LC-Displays viel günstigeren LEDs aus.

Für den nun folgenden Workshop wurde eine Kingbright TA20-11EWA LED-Matrix verwendet. Sie besteht aus sieben Reihen und fünf Spalten; diese oder baugleiche LED-Matrizes sind für wenig Geld in fast jedem Fachgeschäft und -Versand erhältlich. Natürlich funktioniert das auch mit einzelnen LEDs, nur ist der Lötaufwand dabei größer. Mit einer einfachen Google-Suche findet man das entsprechende Datenblatt, unter anderem direkt beim Hersteller Kingbright oder bei Alldatasheet.com (*http://www.alldatasheet.com/datasheet-pdf/pdf/232968/KINGBRIGHT/TA20-11EWA.html*). Die Matrix ist in Deutschland unter anderem bei Reichelt (*http://www.reichelt.de*) erhältlich.

Die Funktionsweise ist recht einfach: Strom kann bei einer LED nur von der Anode zur Kathode fließen. Wird also eine höhere Spannung an die Kathode als an die Anode angelegt, kann kein Strom fließen, die LED bleibt dunkel. Schließt man die Anoden der LEDs spaltenweise und die Kathoden zeilenweise an eine Spannungsquelle an, kann man einzelne Dioden individuell ansteuern, indem man ein Signal auf eine Spalte gibt und einzelne Zeilen auf `LOW` schaltet.

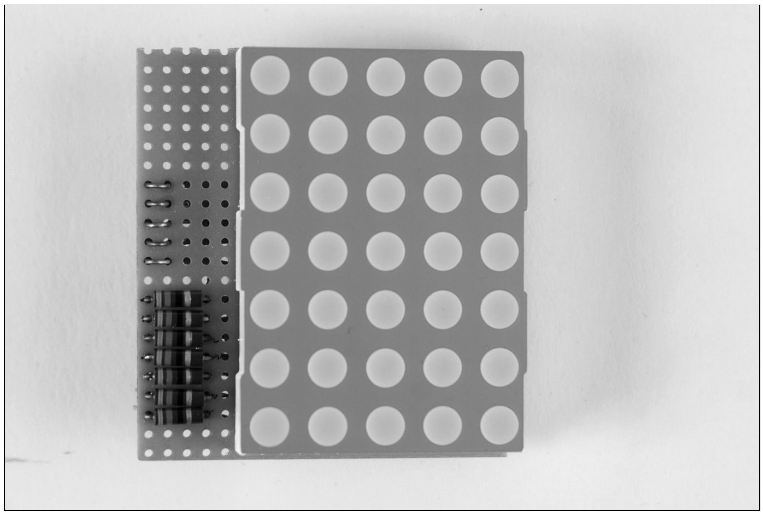
Würde man es dabei belassen, könnte man immer nur bestimmte Muster anzeigen, denn ist eine Zeile auf `LOW` geschaltet, leuchten alle LEDs in dieser Zeile bei entsprechendem Signal auf der Spalte auf. Abhilfe schafft die Darstellung des gewünschten Bildes Zeile für Zeile in einer schnell durchlaufenden Schleife, die für das Auge den gleichen Effekt hat wie die in Kapitel 3 beschriebene Pulsweitenmodulation: Bei entsprechend schneller Wiederholung kann das menschliche Auge kein Flackern mehr erkennen.

Eine andere Form des LED-Displays ist das sogenannte *Siebensegment*. Dabei handelt es sich um die von vielen Anzeigen bekannten kastenförmigen Zahlenanzeigen, die aus sieben einzelnen Segmenten (vier vertikal und drei horizontal) bestehen. Diese Displays können nach dem gleichen Prinzip wie die Matrix programmiert werden. Für den Arduino gibt es in den USA schon fertige Shields, die für 49 Dollar mit einem solchen Display und einem Temperatursensor ausgestattet sind. Wer den teuren Import nicht bezahlen möchte, kann auch im hiesigen Elektronikfachhandel einzelne Displays beziehen, muss sie allerdings selbst anbringen.

Diese Anzeigen eignen sich besonders gut dazu, Sensordaten direkt darzustellen. Man kann sie also zum Beispiel für ein eigenes Thermometer oder einen Luftfeuchtigkeitssensor verwenden, oder man bastelt sich einen Tacho fürs Fahrrad, wenn man auf ein LCD verzichten möchte.

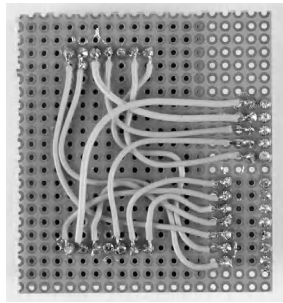
Hardware

Das Datenblatt zeigt für die Reihen eine Belegung für die Matrix-Pins 9, 14, 8, 12/5, 1, 7 und 2 an. Die Spalten werden an 13, 3, 4/11, 10 und 6 angeschlossen. Zwei Pins sind damit doppelt verfügbar, was beim Verlöten mehr Freiheiten erlaubt. In diesem Beispiel werden die Reihen an die Arduino-Pins 1–7, die Spalten an 8–12 angeschlossen. Die Schaltung lässt sich am besten auf eine Lochrasterplatine löten.



◀ Abbildung 4-1
LED-Matrix, Lochrasterplatine

Abbildung 4-2 ▶
LED-Matrix, Lochrasterplatine



Software

Zunächst werden die Reihen in Arrays dargestellt und mit den entsprechenden Pins verbunden:

```
// definiere Zeilen und Spalten als Array vom Typ uint8_t
uint8_t rows[7] = {1, 2, 3, 4, 5, 6, 7};
uint8_t columns[5] = {8, 9, 10, 11, 12};
void setup() {
  for (int i = 0; i < 7; i++) {
    pinMode(rows[i], OUTPUT);
  }
  for (int i = 0; i < 5; i++) {
    pinMode(columns[i], OUTPUT);
  }
}
```

Zudem wird eine Datenstruktur benötigt, die abbilden kann, was die Matrix darstellen soll. Da die Elemente von Arrays nicht aus Variablen bestehen müssen, können Arrays auch wiederum Arrays enthalten. Man spricht von mehrdimensionalen Arrays, in diesem Fall von zweidimensionalen. In unserem Projekt soll zunächst eine Diagonale leuchten, die entsprechenden Werte des Arrays werden also auf 1 gesetzt, der Rest auf 0. Eine genaue Beschreibung von ein- und mehrdimensionalen Arrays finden Sie im Anhang.

```
// Definiere eine 7x5-Matrix. uint8_t ist eine vorzeichenlose
// 8-Bit-Integer-Variable. Setze das Muster so, dass eine
// Diagonale angezeigt wird.
uint8_t matrix[7][5] = {
  { 1, 0, 0, 0, 0},
  { 0, 1, 0, 0, 0},
  { 0, 1, 0, 0, 0},
  { 0, 0, 1, 0, 0},
  { 0, 0, 1, 0, 0},
  { 0, 0, 0, 1, 0},
  { 0, 0, 0, 0, 1},
};
```

Nun muss die Matrix nur noch spaltenweise angezeigt werden. Dazu wird eine Funktion `showMatrix` definiert, die aus einer Schleife besteht. Für jede Spalte wird ein Schleifendurchlauf benötigt. Zunächst werden alle Zeilenwerte für die entsprechende Spalte ausgegeben, indem der negative Wert des entsprechenden Matrixeintrags als Signal geschrieben wird. Das geschieht durch den sogenannten *Negationsoperator*, das Ausrufezeichen. Danach wird die aktuelle Zeile eingeschaltet; die Funktion wartet eine Millisekunde und schaltet sie danach wieder aus, um keinen Konflikt mit der nächsten Zeile, also dem nächsten Schleifendurchlauf zu produzieren.

```
void showMatrix() {
  // durchlaufe alle fünf Spalten
  for (int x = 0; x < 5; x++) {
    // setze vorherige Zeile auf 0
    // für jede Zeile setze den Wert der Matrix auf den
    //entsprechenden Pin
    for (int y = 0; y < 7; y++) {
      digitalWrite(rows[y], !matrix[y][x]);
    }
      // setze aktuelle Spalte auf 1
     digitalWrite(columns[x], 1);
     delay(1);
     // Lösche aktuelle Spalte
     digitalWrite(columns[x], 0);
  }
}
```

Nun muss diese Funktion nur noch in `loop()` aufgerufen werden, um die Matrix anzuzeigen.

Aufgabe:

- Die Werte der Matrix verändern und somit ein Muster anzeigen.

Animationen

Nun sollen nicht nur statische Muster angezeigt werden, sondern eine kleine Animation. Das Setup soll zunächst gleich bleiben, hinzu kommen einige Funktionen, um die Anzeige zu verändern.

Programmierung

Zunächst wird eine Funktion benötigt, die die Matrix löscht und neu initialisiert. Es ist wichtig, die Matrix immer wieder komplett

auf null zu setzen, weil es sonst möglich ist, dass alte Pixel ihren Wert behalten. Um die Matrix zu löschen, wird eine geschachtelte Schleife verwendet, um beide Dimensionen zu durchlaufen, also die x- und die y-Achse der Matrix.

```
// setze die Matrix auf 0
void clearMatrix() {
  // für jede Reihe
  for (int x = 0; x < 7; x++) {
    // für jede Spalte
    for (int y = 0; y < 5; y++) {
      // setze das Matrixelement auf 0
      matrix[x][y] = 0;
    }
  }
}
```

Nun soll in jedem Durchlauf von `changeMatrix` eine weitere LED angeschaltet werden. Dazu werden wiederum eine Schleife für die Reihen und eine für die Spalten benötigt. In der Spaltenschleife wird pro Durchlauf ein weiteres Matrixelement auf `1` gesetzt, die Lichter gehen also nacheinander an. Anschließend wird die Matrix angezeigt. Um diese Anzeige ein wenig zu verzögern, wird jeder Schritt 50 Mal angezeigt und am Ende der Funktion `showMatrix` eine zusätzliche Verzögerung von einer Millisekunde pro Reihe eingebaut:

```
void loop() {
  // setze die Matrix auf 0
  clearMatrix();
  // für jede Reihe
  for (int x = 0; x < 7; x++) {
    // für jede Spalte
    for (int y = 0; y < 5; y++) {
      // setze den Wert des aktuellen Elements auf 1
      matrix[x][y] = 1;
      // zeige die aktuelle Matrix 50 Mal an
      for (int i = 0; i < 50; i++) {
        showMatrix();
      }
    }
  }
```

Aufgabe:

- Eine Animation erstellen, bei der eine leuchtende Zeile von oben nach unten wandert.

Interrupts

Betrachtet man die Matrix genauer, die oben angesteuert wird, fällt auf, dass die LEDs sehr instabil blinken. Manche sind heller als andere, und wenn man genauer hinsieht, erkennt man manchmal ein Flackern. Das liegt daran, dass beim Kompilieren noch viel mehr Code hinzugefügt wird. Der Prozessor arbeitet also keineswegs nur die im Programm angegebenen Anweisungen ab.

Will man nun die Matrix parallel zu etwas anderem laufen lassen, etwa während Sensoren abgefragt werden, verschlimmert sich das Problem weiterhin. Abhilfe schaffen sogenannte *Interrupts*, die die Prioritäten im Programmablauf verändern können. Interrupts unterbrechen das Programm abhängig von externen Ereignissen wie Timer (eine Art Wecker) oder wenn zum Beispiel Daten über die serielle Schnittstelle angekommen sind. Nachdem der Interrupt seinen Code ausgeführt hat, fährt das Programm dort fort, wo es aufgehört hat. Interrupts sind allerdings mit Vorsicht zu behandeln und können besonders für Einsteiger reichlich komplex sein. Es empfiehlt sich also, zunächst einige Erfahrung mit dem Arduino – oder generell mit Programmierung – zu sammeln, bevor man sich daran setzt.

Interrupts sind kein eigentlicher Bestandteil der Arduino-Programmiersprache. Es wird also eine externe Library benötigt, die Bestandteil des *avr*-Pakets ist, das wiederum schon bei der Programmierumgebung dabei ist. Mit

```
#include <avr/interrupt.h>
#include <avr/io.h>
```

werden die beiden nötigen Bibliotheken eingebunden.

Als Nächstes werden, wie schon zuvor, die Zeilen und Spalten der Matrix auf die Pins gelegt. Dabei ist es wichtig, dass die Pins der Reihen nicht verändert werden. Das hängt damit zusammen, dass dann alle Pins auf einen Rutsch geschrieben werden können. Die Funktion `digitalWrite()` wäre an dieser Stelle viel zu langsam.

```
uint8_t matrix_rows[7] = {
  1, 2, 3, 4, 5, 6, 7};
uint8_t matrix_columns[5] = {
  8, 9, 10, 11, 12};
```

Nun muss dem Arduino noch mitgeteilt werden, dass ein sogenannter Timer-Interrupt benötigt wird. Dazu werden ein paar Werte in spezielle Register geschrieben. Diese starten einen inter-

nen Zähler, der das Programm dann regelmäßig unterbricht und die Interrupt-Funktion aufruft.

```
unsigned char Matrix_SetupTimer2(){
  // Timer2 Settings: Timer Prescaler /8, mode 0
  // Timer clock = 16MHz/8 = 2Mhz or 0.5us
  TCCR2A = 0;
  TCCR2B = 0<<CS22 | 1<<CS21 | 1<<CS20;
  // Timer2 Overflow Interrupt Enable
  TIMSK2 = 1<<TOIE2;
}
```

Es folgt die eigentliche Interrupt-Funktion:

```
// Bildspeicher für die Matrix
// gefüllt mit einer diagonalen Linie zum Testen
uint8_t matrix[5] = {
  2+64, 4+128, 8, 16, 32};
// aktuelle Spalte für die Interruptausgabe
uint8_t matrix_col;
// Timer2 Interruptfunktion / wird in regelmäßigen Abständen
// aufgerufen, ohne dass man sich darum kümmern muss
ISR(TIMER2_OVF_vect) {
  // setze aktuelle Spalte auf 0
  digitalWrite(matrix_columns[matrix_col], 0);
  matrix_col++;
  if (matrix_col >= 5) matrix_col = 0;
  PORTD = ~ matrix[matrix_col];
  // setze aktuelle Spalte auf 1
  digitalWrite(matrix_columns[matrix_col], 1);
}
```

Hier passiert die eigentliche Arbeit: Als Erstes wird die zuletzt aktive Zeile ausgeschaltet. Dann wird die nächste bestimmt und die Daten dafür ausgegeben, und gleich danach wird diese Zeile dann aktiviert.

Relativ ungewohnt für ein Arduino-Programm ist die Zeile, die mit `PORTD` beginnt. PORTD ist eine acht Bit breite Speicherstelle im Atmel (dem Arduino-Mikrocontroller), die direkt den Zustand der Pins 0–7 beschreibt. Alle Werte, die dorthin geschrieben werden, erscheinen direkt als `HIGH`- und `LOW`-Signal auf den entsprechenden Pins. Eine Null würde zum Beispiel alle Pins auf null setzen. Eine 85 würde jeden zweiten Pin auf `HIGH` und alle anderen auf null setzen, da die 85 binär geschrieben 01010101 entspricht.

Der Tildenoperator (~) hat hier die Aufgabe, alle Bits umzudrehen: Aus Einsen macht er Nullen und umgekehrt. Das ist notwendig, weil an diesen Pins ja die Kathoden der LEDs angeschlossen sind, die leuchten, wenn die Kathode LOW = 0 ist, und nicht (wie man zuerst erwarten würde) bei HIGH = 1. Im Programmiersprachenan-

hang finden Sie noch mehr Informationen über Operatoren und ihr Verhalten und ihre Verwendung. Um Operatoren zu verstehen, sollten Sie sich also dort weiter informieren.

Das Schöne an dieser Lösung ist, dass alles, was in die Matrixvariable geschrieben wird, einfach dargestellt wird und man dann in der Funktion `loop()` keine Einschränkungen mehr hat, die das Timing der Matrix betreffen.

Buchstaben

Nun macht es keinen Spaß, Buchstaben und Symbole Pixel für Pixel mit `SetPixel(x,y)` zu zeichnen; deshalb gibt es dafür eine kleine Tabelle, die alle kleinen und großen Buchstaben und die Zahlen enthält. Die Funktion `DisplayChar()` holt diese dann aus dem Speicher und kopiert sie in die Variable `matrix`.

Damit können kleinere Texte und Symbole auf der LED-Matrix ausgegeben werden.

```
void SetPixel(uint8_t x, uint8_t y)
{
  matrix[x] |= _BV(y+1);
}
void ClrPixel(uint8_t x, uint8_t y)
{
  matrix[x] &= ~_BV(y+1);
}
void ClrMatrix(void)
{
  for(uint8_t x = 0; x <= 5; x++){
    matrix[x] = 0;
  }
}
static unsigned char __attribute__ ((progmem)) Font5x7[] = {
       0x7E, 0x11, 0x11, 0x11, 0x7E,// A
       0x7F, 0x49, 0x49, 0x49, 0x36,// B
       0x3E, 0x41, 0x41, 0x41, 0x22,// C
       0x7F, 0x41, 0x41, 0x22, 0x1C,// D
       0x7F, 0x49, 0x49, 0x49, 0x41,// E
       0x7F, 0x09, 0x09, 0x01, 0x01,// F
       (....komplettes Listing auf der Webseite....)
};
void DisplayChar(uint8_t c)
{
       for(uint8_t i = 0; i <= 4; i++)
       {
       matrix[i] = pgm_read_byte(&Font5x7[((c - 0x10) * 5) + i])
<< 1;
```

Hinweis

Die Daten für das Aussehen der Buchstaben werden im Flashspeicher des Arduinos gespeichert, anders als beim Ramspeicher kann auf das Flash leider nicht direkt zugegriffen werden. Dazu wird die Funktion `pgm_read_byte(adresse)` benötigt. Diese bekommt als Parameter die Adresse, welche gelesen werden soll, und liefert dann den Wert an dieser Stelle in Flash zurück.

Im ASCII-Zeichensatz (Das ist die Tabelle, die der Computer verwendet, um Zahlen zu Buchstaben zuzuordnen) sind für uns nicht alle Bereiche wichtig, die Tabelle fängt deswegen erst beim 16ten Buchstaben an. Also wird als erstes 16 oder 0x10 (Hexadezimale Schreibweise) vom gewünschten Zeichen »c« subtrahiert.

Da jeder einzelne Buchstabe aus 5 Bytes besteht, wird das Resultat jetzt mit 5 multipliziert, das ist jetzt die Nummer des ersten Eintrags in der Tabelle `Font5x7` für den Buchstaben »c«.

Bei jedem der 5 Schleifendurchläufe (für einen Buchstaben) erhöht sich i um eins, so das nach und nach alle 5 Bytes eines Buchstabens verarbeitet werden.

Das Zeichen »&« vor dem `Font5x7[...]` bewirkt dabei, dass die Adresse des Bytes innerhalb der Tabelle zurückgeliefert wird um dann mit der Funktion `pgm_read_byte` das tatsächliche Byte lesen zu können.

```
        }
}
void loop() {
  for (uint8_t i = 0; i < 32; i++)
  {
    DisplayChar('A'+i);      // Zeige
Buchstaben an.
    delay(500);
    DisplayChar(0x10+i);     // Zeige
Gesichter an.
    delay(500);
  }
}
```

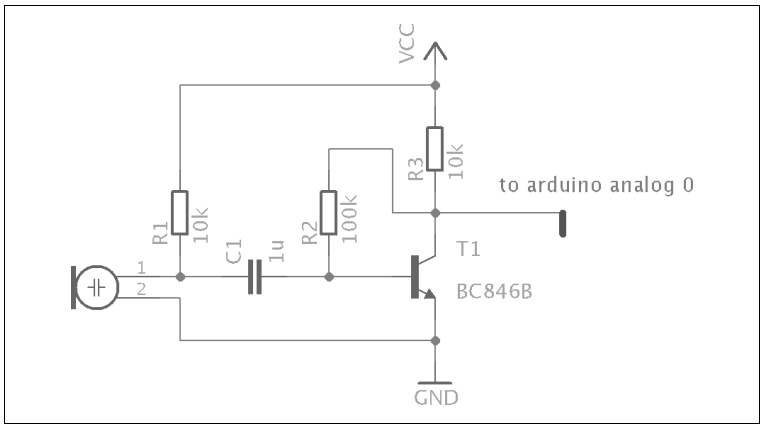
Tamagotchi

Tamagotchis sorgten im Jahr 1997 für einen Ansturm auf Kaufhäuser und Spielzeugläden in Deutschland und verschwanden dann fast ebenso schnell in der Versenkung, wie sie gekommen waren. Im Jahr 2004 wurde die Serie noch einmal aufgelegt, wenn auch mit keinem zur Urversion vergleichbaren Erfolg in Deutschland. Ein Tamagotchi ist ein kleines Plastikei mit einem LCD, auf dem ein kleines Küken dargestellt ist. Dieses Küken gibt vor zu leben und benötigt regelmäßig Wasser, Futter und Zuneigung. Nach und nach entwickelt es eine eigene kleine Persönlichkeit, die individuell

festlegt, wann das Tamagotchi um Aufmerksamkeit piepst. Die neueren Versionen sind in der Lage, sich auch mit anderen zu verbinden und zum Beispiel zu heiraten. Die Grundidee bleibt aber die gleiche: ein virtuelles Haustier, das nicht so anspruchsvoll ist wie ein echtes und damit auch für diejenigen Kinder geeignet, deren Eltern einem echten Lebewesen kein Risiko zumuten wollen. Hier soll nun ein eigenes kleines Haustier gebaut werden, das die oben beschriebene LED-Matrix verwendet. Um das Projekt von dem im Handel verfügbaren Spielzeug abzuheben, soll es »Ardugotchi« heißen. Das Ardugotchi ist ein sehr glückliches Lebewesen; es ernährt sich von Elektronen, braucht also nie gefüttert zu werden, freut sich allerdings über Geräusche und Musik.

Hardware

Wir verbinden das Mikrofon wie auf der Zeichnung mit dem Transistor und dem Arduino-Board, damit das Ardugotchi auch etwas hören kann. Ansonsten bleibt die Matrix aus den vorigen Projekten bestehen.



◀ **Abbildung 4-3**
Tamagotchi-Schaltung

Software

Zunächst kann der Programmcode des vorigen Beispiels übernommen werden, mit dem die Matrix angezeigt wird. Einige Bilder können hineingeladen werden, die die verschiedenen Gefühlszustände des Tierchens beschreiben. Sie sind bereits in der Zeichentabelle enthalten und unter den Codes 0x10 bis 0x1f zu finden. Den kompletten Code finden Sie auf der entsprechenden Buchseite bei *http://arduinobuch.wordpress.com*.

Die Spannung am analogen Eingang 0, an den das Mikrofon angeschlossen ist, wird eingelesen und mit dem Mittelwert der letzten 100 Messungen verglichen. Ist der Wert deutlich größer, so wechselt das Ardugotchi seinen Gesichtsausdruck. Um Gesichter darstellen zu können, enthält die Zeichentabelle 16 verschiedene Muster, sie können mit `DisplayChar(0x10+gesicht)` angezeigt werden.

```
int laut;
int laut_mittelwert;
int sad;
int gesicht;
// Dieser Wert muss je nach verwendetem Mikrofon angepasst werden.

// Kleinere Werte machen den Ardugotchi dabei empfindlicher.
int schwelle = 50;
void loop() {
  laut = analogRead(0);

  // Mittelwert über die letzten gemessenen Werte, wobei der neue
  // jeweils 1/100 zählt.
  laut_mittelwert = (laut_mittelwert*99 + laut)/100;

  if ((laut - laut_mittelwert) > schwelle)
  {
    // nächstes Gesicht auswählen, am Ende wieder zum Anfang
    // springen
    gesicht++;
    if (gesicht > 15) gesicht = 1;

    // Mittelwert anpassen, damit nicht gleich wieder ein Peak
    // erkannt wird.
    laut_mittelwert = laut;

    // Den Zähler für die schlechte Laune wieder auf 0 setzen
    sad = 0;
  }

  // Hört das Ardugotchi lange genug kein Geräusch, soll es sich
  // einsam fühlen und ein entsprechendes Gesicht anzeigen.
  sad++;
  if (sad > 10000)
  {
    // den Wert auf 10000 begrenzen, sonst ensteht ein Überlauf
    sad = 10000;

    // Schlecht gelauntes Gesicht auswählen
    gesicht = 0;
  }

  DisplayChar(0x10+gesicht);     // Zeige Gesichter an.

  // Geschwindigkeit auf 10x Sekunde begrenzen
  delay(100);
}
```

Brainwave und Biofeedback

Da ein Teil der Kommunikation unserer Nervenzellen mit kleinen Spannungsimpulsen funktioniert, können mit empfindlichen Elektroden am Kopf verschiedene Aktivitätslevel gemessen werden. Diese Messung erlaubt allerdings keine genauere Auswertung. Man kann sich das ungefähr so vorstellen wie wenn man anhand des Geräuschpegels eines Stadions erkennt, dass ein Tor gefallen ist. Interessant ist, dass die Aktivitätslevel in periodischen Wellen auftreten. Die Abstände dieser Wellen werden grob verschiedenen Gehirnzuständen zugeordnet.

- Deltawellen weisen die niedrigste Frequenz auf (1–4 Hertz). Sie werden zum Beispiel in der Tiefschlafphase emittiert, wenn das Gehirn nicht träumt. Auch werden diese Wellen bei Bewusstlosigkeit oder intuitivem, unbewusstem Verhalten gemessen. In diesem Zustand wird das Gehirn nur bei starken Reizen aktiv.
- Thetawellen liegen zwischen 4 und 7 Hertz und treten vor allem in leichten Schlafphasen auf, aber auch bei unbewussten, kreativen Denkprozessen oder Zuständen tiefer Entspannung wie etwa Meditation.
- Alphawellen liegen im Bereich zwischen 8 und 13 Hertz. Ein großer Teil dieser Wellen wird mit leichter Entspannung assoziiert, zum Beispiel mit geschlossenen Augen. Das Gehirn ist im Empfangsmodus, verhält sich aber passiv.
- Betawellen (14–30 Hz) sind der normale, aktive Zustand. Das Gehirn ist wach und bei vollem Bewusstsein. Nachdenken und Konzentration gehören ebenfalls dazu. Sie treten aber auch zum Beispiel im REM-Schlaf auf.
- Als Gammawellen wird der Frequenzbereich über 30 Hertz bezeichnet. Sie treten zum Beispiel bei starker Konzentration und Lernprozessen auf, aber auch in Angst- und Stresszuständen. Gammawellen sind allerdings bislang nicht gut erforscht.
- Oft geschieht es nun, dass man selbst Entspannung sucht, aber aufgrund der Hektik des Tages und der Umgebung nicht ruhig genug wird, um von der aktiven Betawellenphase auf Alpha herunterzukommen. Diesen Zustand kann man nun erreichen, indem man meditiert oder beispielsweise Yoga macht. Man kann versuchen, diesen Zustand auch mit externen Mitteln wie Lichtimpulsen oder Musik herbeizuführen.

Die Brain Machine

Die Theorie, die dieser Brain Machine zugrunde liegt, geht davon aus, dass über das Auge aufgenommene Lichtblitze die Aktivitätswellen im Gehirn verändern können. Treffen Lichtblitze mit 2,2 Hertz aufs Auge auf, werden vom Gehirn Deltawellen emittiert. Natürlich bedeutet das nicht automatisch, dass das Gehirn in diesen Zustand fällt, nur weil es mit entsprechendem Licht bombardiert wird. Aber eine geschickte Programmierung und ein bisschen Training können durchaus zu gewünschten Ergebnissen führen. Eine interessante Ressource zu Gehirnforschung und verwandten themen bietet Arvid Leyh unter *www.nurindeinemkopf.de*.

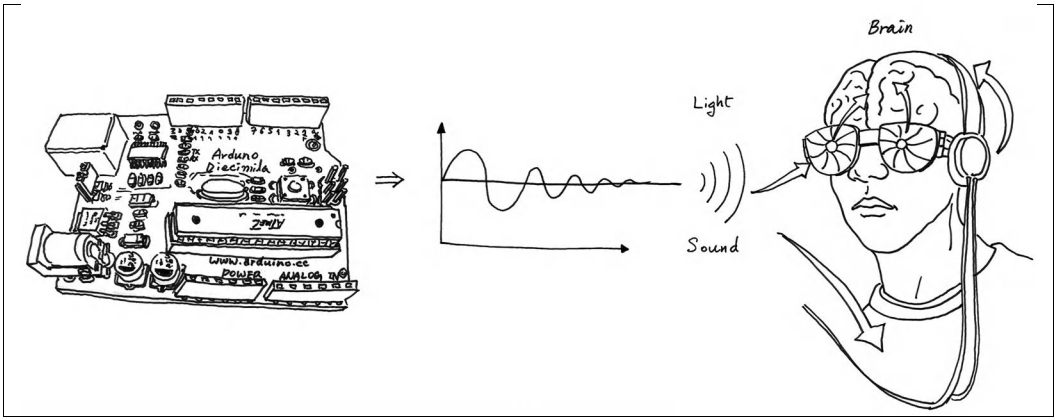


Abbildung 4-4 ▲
BrainMachine

Im Folgenden soll nun eine Gehirnmaschine gebaut werden, die in der Lage ist, das menschliche Gehirn in einen Zustand der Entspannung zu versetzen. Dabei sorgen an einer Brille angebrachte LEDs für die Lichtimpulse. Zudem wird versucht, das Ergebnis mit binauralen Klängen auf Kopfhörern noch zu steigern. Binaural bedeutet wörtlich übersetzt »Zwei-ohrig«, man hört Unterschiedliches auf beiden Ohren. Da wir so tiefe Frequenzen wie Deltawellen nicht hören können, spielen wir einen höheren Ton, der vom rechten zum linken Ohr genau um die gewünschte Frequenz auseinanderliegt. Die Überlagerung dieser zwei Töne erzeugt ein Pulsieren in der gewünschten Frequenz.

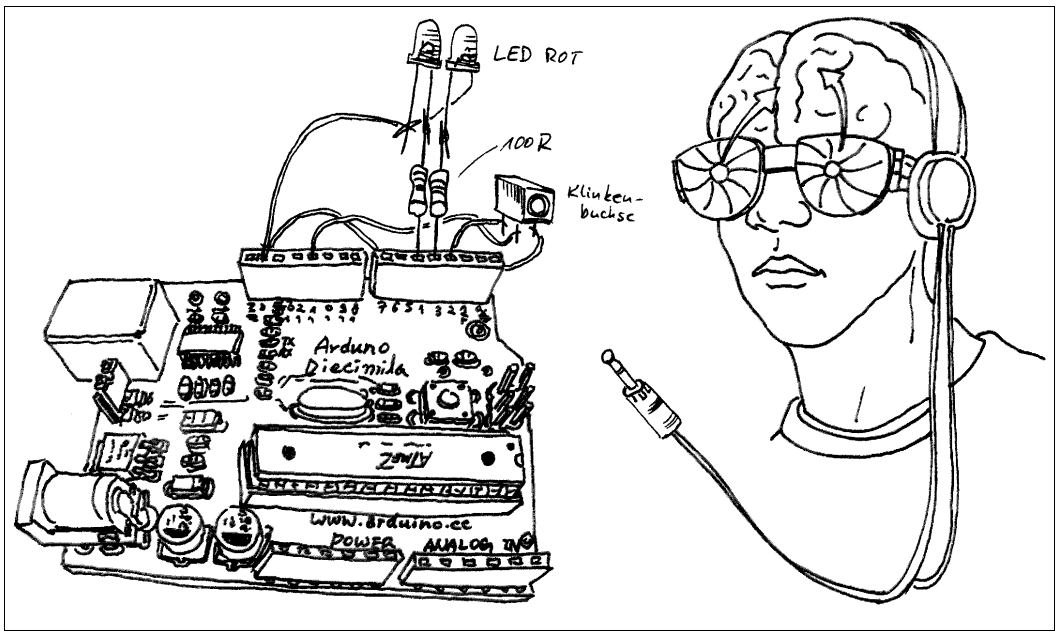
Setup

Für die Brain Machine werden einige Bauteile benötigt, allem voran eine Brille, um die LEDs vor dem Auge zu positionieren. Dafür genügt eine einfache Schutzbrille aus dem Baumarkt. Um die Elek-

tronik auf dieser Brille anzubringen, sollten Sie mit Silikonkleber oder Heißkleber arbeiten. Zudem werden zwei helle rote LEDs, passende Widerstände (siehe Kapitel 3), ein Kopfhörer und eine passende Klinkenbuchse benötigt.

Zunächst markiert man auf jedem Brillenglas einen Punkt, der direkt vor den Augen sitzt. Dort wird anschließend ein Loch gebohrt, durch das die LED gesteckt wird. Wenn die LED also direkt in die Pupille zeigt, wird sie mit Klebematerial fixiert und über einen Widerstand mit einem digitalen Arduino-Pin verbunden.

Als Nächstes muss die Klinkenbuchse korrekt angeschlossen werden. Sie besteht aus drei Anschlüssen: Der äußerste ist für die Masse, wird also mit GND verbunden. Die anderen beiden sind für die Tonsignale. Bei der Brain Machine ist es unwichtig, welche Frequenz auf welchem Ohr ertönt. Die Töne werden durch eine schnelle Pulsweitenmodulation erzeugt, wobei es angenehmer sein kann, sie noch ein wenig zu filtern, bevor sie auf den Kopfhörer gegeben werden. Andernfalls kann das Ergebnis relativ schrill klingen. Für den Tiefpass kann man einen Widerstand von 1.000 Ohm und einen 100-nF-Kondensator verwenden.



▲ **Abbildung 4-5**
BrainMachine-Schaltung

Damit ist die Brain Machine im Grunde fertig. Für den ersten Test empfiehlt es sich, alles unberührt zu lassen; möchte man die Brille

dann permanent verwenden, sollte man mit Schrumpfschläuchen, Klebeband und Heiß- oder Silikonkleber alles so anbringen, dass es stabil sitzt. Zum Schluss kann man die Brillengläser noch mit Papier oder Pappe abdichten. Unter *http://makezine.com/10/brainwave* können Sie ein passendes Muster herunterladen, das die Brille ein wenig psychedelisch aussehen lässt.

Programmierung

Wie eingangs erwähnt, müssen für die verschiedenen Gehirnwellen passende Frequenzen auf den LEDs und im Kopfhörer erzeugt werden. Dazu werden zunächst die nötigen Pins deklariert:

```
int ledPin1 = 4;            // LED rechtes Auge
int ledPin2 = 5;            // LED linkes Auge
int speakerPin1 = 11;       // Kopfhörer rechtes Ohr
int speakerPin2 = 3;        // Kopfhörer linkes Ohr
volatile uint16_t sample1;  // welches ist das nächste Sample aus
                                        // der Sinustabelle
volatile uint16_t sample2;  // die oberen 7 Bit zeigen in die
                            // Tabelle, die restlichen Bits sind
                            // Kommastellen
int diff = 5;               // Differenz der beiden Töne in
                                         // update-ticks
int tone = 500;             // Frequenz der beiden Töne in
                            // update-ticks
int vol = 0;                // aktuelle Lautstärke
int set_vol = 0;            // gewünschte Lautstärke
```

Um Daten mit der Interrupt-Funktion auszutauschen, werden diese in gemeinsam mit dem normalen Programm genutzte Variablen geschrieben. `diff` gibt den Frequenzunterschied der beiden Tonkanäle und die Pulsfrequenz der LED an. Der Maßstab dieser Variablen ist, um wie viel der Sample-Zähler 8.000 Mal pro Sekunde erhöht werden muss, um die gewünschte Frequenz zu erreichen. Diese Berechnung unternimmt eine gesonderte Funktion. Dadurch braucht diese Berechnung nur bei einer Tonänderung durchgeführt zu werden, was Rechenkapazität spart.

Schön weich klingende Töne erhält man mit einer Sinusfunktion. Da das Berechnen eines Sinus viel Zeit beansprucht, wird hier eine Vorberechnung auf dem PC verwendet, sodass die folgende Tabelle exakt 128 Werte lang ist. Da die Sinusfunktion sich nach 2ð wiederholt, erhält man den Sinus einer Zahl, wenn man an der Stelle *Zahl modulo 128* in der Tabelle nachschlägt. Modulo 128 bedeutet, dass die Zahl durch 128 geteilt und der Rest zurückgegeben wird. Das ist für einen Prozessor wiederum besonders einfach zu berechnen, es müssen nur alle Bits größer 128 weggelassen werden.

```
//prog_uint8_t sintab[] = {
const unsigned char sintab[] PROGMEM = {
  0x01,0x01,0x01,0x01,0x02,0x03,0x05,0x07,
  0x09,0x0c,0x0f,0x12,0x15,0x19,0x1c,0x21,
  0x25,0x29,0x2e,0x33,0x38,0x3d,0x43,0x48,
  0x4e,0x54,0x5a,0x60,0x66,0x6c,0x73,0x79,
  0x7f,0x85,0x8b,0x92,0x98,0x9e,0xa4,0xaa,
  0xb0,0xb6,0xbb,0xc1,0xc6,0xcb,0xd0,0xd5,
  0xd9,0xdd,0xe2,0xe5,0xe9,0xec,0xef,0xf2,
  0xf5,0xf7,0xf9,0xfb,0xfc,0xfd,0xfe,0xfe,
  0xfe,0xfe,0xfe,0xfd,0xfc,0xfb,0xf9,0xf7,
  0xf5,0xf2,0xef,0xec,0xe9,0xe5,0xe2,0xdd,
  0xd9,0xd5,0xd0,0xcb,0xc6,0xc1,0xbb,0xb6,
  0xb0,0xaa,0xa4,0x9e,0x98,0x92,0x8b,0x85,
  0x7f,0x79,0x73,0x6c,0x66,0x60,0x5a,0x54,
  0x4e,0x48,0x43,0x3d,0x38,0x33,0x2e,0x29,
  0x25,0x21,0x1c,0x19,0x15,0x12,0x0f,0x0c,
  0x09,0x07,0x05,0x03,0x02,0x01,0x01,0x01
};
```

Diese Tabelle gibt es auch auf der Webseite zum Buch: *http://arduinobuch.wordpress.com* herunterzuladen.

Nun folgt schon die komplizierteste Funktion im Programm, die Interrupt-Funktion: Sie wird von der Hardware automatisch 8.000 Mal pro Sekunde aufgerufen, hier werden die LEDs ein- und ausgeschaltet und die Werte für die Töne ausgegeben.

```
ISR(TIMER1_COMPA_vect) {
  // Zähler, um Lautstärkeänderung langsamer zu machen
  static int timer1counter;
  int wert;
  // Wert an der Stelle sample1/512 aus der sinus-Tabelle lesen
  wert = pgm_read_byte(&sintab[(sample1 >> 9)]);
  // Wert mit der aktuellen Lautstärke multiplizieren
  wert = (wert * vol) / 256;
  // PWM-Hardware anweisen, ab jetzt diesen Wert auszugeben
  OCR2A = wert;
  // das Gleiche für das andere Ohr
  wert = pgm_read_byte(&sintab[(sample2 >> 9)]);
  wert = (wert * vol) / 256;
  OCR2B = wert;
  // berechnen der LED-Pulse in Abhängigkeit vom gespielten Ton
  if (((sample1 & 0x8000) != 0) && ((sample2 & 0x8000) !=0)) {
    PORTD |= _BV(5) | _BV(4);     // LEDs einschalten
  }
  else {
    PORTD &= !(_BV(5) | _BV(4));  // LEDs ausschalten
  }
  // nächstes Sample in der Sinustabelle abhängig vom gewünschten
  // Ton auswählen
  sample1 += tone;
  sample2 += tone  + diff;
```

```
  // Lautstärke anpassen, wenn gewünscht (nur alle 50 Interrupts,
  // damit es schön langsam passiert
  timer1counter++;
  if (timer1counter > 50)
  {
    timer1counter = 0;
    if (vol < set_vol) vol++;
    if (vol > set_vol) vol--;
  }
}
```

Als Nächstes kommt die Funktion `startPlayback()`. Hier werden die Interrupt-Funktion und die Timer des Arduino so eingestellt, dass der Interrupt 8.000 Mal pro Sekunde aufgerufen wird und die PWM-Ausgabe mit 62.500 KHz läuft.

```
void startPlayback()
{
  pinMode(speakerPin1, OUTPUT);
  pinMode(speakerPin2, OUTPUT);
  // initialisiere den Timer 2 für die schnelle PWM zur
  // Soundausgabe auf Pin 11 & 3
  // verwende den internen Takt (Datenblatt Seite 160)
  ASSR &= ~(_BV(EXCLK) | _BV(AS2));
  // fast PWM mode  (Seite 157)
  TCCR2A |= _BV(WGM21) | _BV(WGM20);
  TCCR2B &= ~_BV(WGM22);
  // wähle die nicht invertierende PWM für pin OC2A und OC2B,
  // am Arduino ist das Pin 11 und 3
  TCCR2A = (TCCR2A | _BV(COM2A1) | _BV(COM2B1));
  // keine Vorteiler, denn wir wollen es schnell (Seite 158)
  TCCR2B = (TCCR2B & ~(_BV(CS12) | _BV(CS11))) | _BV(CS10);
  // Startwert = 0, sonst gibt es ein hässliches Plopp-Geräusch
  OCR2A = 0;
  OCR2B = 0;
  // initialisiere Timer 1 für 8.000 Interrupts/Sekunde
  cli();
  // set CTC mode (Clear Timer on Compare Match) (Seite 133)
  TCCR1B = (TCCR1B & ~_BV(WGM13)) | _BV(WGM12);
  TCCR1A = TCCR1A & ~(_BV(WGM11) | _BV(WGM10));
  // kein Vorteiler (Seite 134)
  TCCR1B = (TCCR1B & ~(_BV(CS12) | _BV(CS11))) | _BV(CS10);
  // gewünschte Frequenz: 8.000 KHz
  OCR1A = F_CPU / SAMPLE_RATE;    // 16e6 / 8000 = 2000
  // aktiviere den Interrupt für TCNT1 == OCR1A (Seite 136)
  TIMSK1 |= _BV(OCIE1A);
  // Startwerte
  sample1 = 0;
  sample2 = 0;

  // globale Interrupts wieder einschalten
  sei();
}
```

Vieles in dieser Funktion muss man nicht unbedingt auf Anhieb verstehen, in den meisten Fällen sind diese Hardwaredetails auf dem Arduino in Bibliotheken versteckt. Hier hilft nur das Lesen des Datenblatts vom verwendeten Mikrocontroller Atmega168 oder Atmega328, das direkt bei *www.atmel.com* eingesehen werden kann.

In der Setup-Funktion werden nur noch die Pins für die LEDs initialisiert und die Funktion `startPlayback` aufgerufen.

```
void setup()
{
  pinMode(ledPin1, OUTPUT);
  pinMode(ledPin2, OUTPUT);
  startPlayback();
}
```

Damit die gewünschte Tonhöhe und Hirnwellenfrequenz nicht in der Einheit *Timer Additions* angegeben werden muss, gibt es diese Hilfsfunktion:

```
// ändert die Ton und die LED-Frequenz
// ton (50-4.000 Hz)
// brainwave (0-30 Hz)
// volume (0-256);
void SetFreq(int ton, int brainwave, int volume)
{
  tone = (128ul*512ul*ton)/8000;
  diff = (128ul*512ul*brainwave)/8000;
  set_vol  = volume;
}
```

In der `loop()`-Funktion kann jetzt die Gehirnwellensequenz programmiert werden, im Beispiel eine Einschlafmeditation ohne Aufweckfunktion am Ende, also genau das Richtige, um nach einem anstrengenden Tag einzuschlafen.

Als Erstes werden immer wieder gebrauchte Werte in Konstanten gespeichert, danach folgt die Gehirnwellensequenz, die aber hier auf Grund ihrer Länge nicht komplett abgedruckt ist.

```
float DELTA =  2.2;      // tiefe Bewusstlosigkeit, Intuition und
                         // insight
float THETA =  6.0;      // Unterbewusstsein, Kreativität,
                         // Tiefenentspannung
float ALPHA = 11.0;      // Träumerisch, empfangsbereit und passiv
float BETA  = 14.4;      // bewusstes Denken, externer Fokus
int grundton = 200;

void loop()
{
  SetFreq(grundton,0.7,0);   // Einstimmung, langsames Pulsen ohne
                             // Ton
```

```
  delay(10000);              // Dauer 10 Sekunden
  SetFreq(grundton,BETA,10); // BETA langsam lauter werden
  delay(10000);
  SetFreq(grundton,BETA,20);
  delay(10000);
  SetFreq(grundton,BETA,30);
  delay(10000);
  SetFreq(grundton,BETA,40);
  delay(10000);
  SetFreq(grundton,ALPHA,50);  // 10 Sekunden Alphawellen
  delay(10000);
  SetFreq(grundton,BETA,60);   // 20 Sekunden Betawellen
  delay(20000);
  // (.....) - der Rest findet sich wiederum im Listing auf der
  // Webseite zum Buch
  SetFreq(grundton,DELTA,  2); //  Deltawellen immer leiser werden
  delay(5000);
  SetFreq(grundton,DELTA,  1); //  Deltawellen immer leiser werden
  delay(5000);
  SetFreq(grundton,DELTA,  0); //  Deltawellen immer leiser werden
  delay(5000);
  // ende
  for (;;) ;
}
```

Den genauen Ablauf einer kompletten Brainwave-Session können Sie auf der Website des Make Magazine sehen: *http://makezine.com/10/brainwave/*. Dort finden Sie auch weitere Informationen über die Wirkungsweise dieser Gehirnwellenmaschine.

KAPITEL 5

Sprich mit mir, Arduino!

In diesem Kapitel:

- Nach Hause telefonieren mit der seriellen Konsole
- Automatisierung mit Gobetwino
- Processing

Der Arduino ist beileibe keine einsame Insel, auf der nur einmal Passagiere abgeladen werden, ohne dass sie die Außenwelt informieren können. Bleibt das Board mit einem USB-Kabel (oder in der Bluetooth-Variante über Funk) mit einem PC verbunden, kann es auch Daten zurücksenden. Das folgende Kapitel soll zunächst erklären, wie einfache Statusmitteilungen über die serielle Konsole an die Arduino-Entwicklungsumgebung gesendet werden können. Anschließend wird die Kommunikation noch etwas ausgebaut: Die Processing-Programmiersprache etwa besitzt eine Bibliothek, die es sehr einfach macht, mit dem Arduino Daten auszutauschen. So können das Board und seine angeschlossenen Sensoren oder Taster benutzt werden, um Processing-Programme zu steuern.

Unter »digitaler Kommunikation« versteht man den Austausch von Daten zwischen zwei oder mehr Geräten über eine festgelegte Verbindung. Dabei bestimmt ein sogenanntes Protokoll alle Details. Alle Kommunikationsteilnehmer müssen sich dabei genau an dieses Protokoll halten, damit die Daten auch korrekt verstanden und weiterverarbeitet werden können. Protokolle können unterschiedliche Bereiche umfassen, und meistens werden mehrere Protokolle zusammen verwendet. Zu diesem Zweck werden Netzwerke in ihrem Aufbau nach Schichten getrennt, die im sogenannten *Open Systems Interconnection Model*, kurz OSI-Modell, festgeschrieben sind. Diese Schichten bauen aufeinander auf und beschreiben die komplette Kommunikation vom physischen Träger bis hin zu den Datenpaketen. Im Folgenden werden wir eine serielle Verbindung in ihren Grundzügen erklären.

Die unterste Schicht beschreibt, wie die beiden kommunizierenden Seiten physisch miteinander verbunden sind, wie die entsprechen-

den Anschlüsse aufgebaut und wie viele Verbindungen nötig sind, um Daten hin und her zu senden. Beim Arduino sind es eigentlich zwei Verbindungen über die RX- und TX-Pins. Da viele Rechner – insbesondere Laptops – keine RS232-Schnittstelle mehr besitzen, wird diese über USB emuliert. Dabei sorgt ein USB-Controller dafür, dass das entsprechende, deutlich kompliziertere USB-Protokoll verwendet werden kann. Die serielle Verbindung wird also quasi darin eingebettet. Einer der besonderen Vorteile von USB gegenüber einem seriellen Port (RS232), wie er in älteren Computern verwendet wurde, ist, dass an einem einzelnen Anschluss eine Vielzahl von Verbindungen registriert werden kann. USB-Hubs nutzen diese Technik zum Beispiel, um an die üblicherweise zwei oder drei Steckplätze viele Geräte anzuschließen. Jedes weitere Gerät, das sich am USB-Bus anmeldet, wird im Betriebssystem als ein neues serielles Gerät festgestellt. Unter Windows können zum Beispiel drei Geräte als COM8, COM9 und COM10 angesprochen werden, Mac OS X würde sie als

```
/dev/tty.usbserial-5B21
/dev/tty.usbserial-5B22
/dev/tty.usbserial-5B24
```

auflisten. Es ist also kein Problem, mehrere Arduinos an einen Rechner anzuschließen. In der Programmierumgebung können diese dann unter TOOLS → SERIAL PORT ausgewählt werden, und in Processing steht die Funktion `Arduino.list()` zur Verfügung (siehe unten).

Darauf folgt die elektrische Schicht, die festlegt, mit welchen Spannungen die Daten übertragen werden, also wie viel Volt nötig sind, um eine Kommunikation aufrechtzuerhalten.

Die dritte Schicht bezeichnet die Logik der Verbindung. Sie legt fest, ob eine Zunahme der Spannung eine Null oder eine Eins bedeutet. Bei der seriellen Verbindung wird ein 5-Volt-Signal als eine Eins interpretiert, ein 0-Volt-Signal dementsprechend als eine Null.

Die Datenschicht wiederum regelt das Timing der Bits, also wie viele Signale in einem bestimmten Zeitabstand gesendet werden. Das ist sehr wichtig, um die Daten auf der Gegenseite auch korrekt interpretieren zu können. Zudem wird hier festgelegt, ob die Daten einer Gruppe mit einer bestimmten Bitfolge gekennzeichnet oder abgeschlossen werden. Die serielle Verbindung überträgt 9.600 Bit pro Sekunde. Jedes Byte, also jedes Paket, enthält acht Bits und wird jeweils mit einem Start- und einem Stop-Bit markiert. Diese

Bits werden vom Arduino direkt verarbeitet und kommen hier nicht weiter vor. Sollen viele Daten pro Sekunde verarbeitet werden, ist es wichtig zu wissen, dass ein übertragenes Byte also eigentlich aus zehn Bit besteht, pro Sekunde also nicht mehr als 960 Bytes (z.B. Buchstaben) übertragen werden können.

Die Anwendungsschicht beschreibt, wie die einzelnen Bytes zu verstehen sind. Sowohl die Programmierumgebung (oder später die Processing-Sprache) als auch das Arduino-Programm, das auf das Board geladen wird, nehmen jeweils ein Byte an und verarbeiten es anschließend.

Die Kommunikation mit dem Arduino Duemilanove geschieht dabei über die digitalen Pins 0 und 1, die auch als RX und TX markiert sind. Nutzt man also die serielle Kommunikation, sollten diese Pins nicht anderweitig belegt werden, da sonst seltsame Fehler auftreten können. LEDs beispielsweise würden jedes Mal blinken, wenn Daten über die serielle Verbindung gesendet werden. Andersherum kämen auf dem PC Daten heraus, die nicht verarbeitet werden könnten, sobald auf dem Pin ein digitales Signal angelegt würde.

In diesem Kapitel wird nur die sehr einfach aufgebaute serielle Verbindung behandelt. Wird zum Beispiel über Ethernet kommuniziert, ist das Schichtenmodell deutlich komplexer, es bauen mehr Protokolle aufeinander auf.

Nach Hause telefonieren mit der seriellen Konsole

Der Arduino verfügt mit der beschriebenen seriellen Verbindung über einen Rückkanal, kann also Ereignisse an die Arduino-IDE zurückmelden. »Seriell« bedeutet dabei, dass die Daten bitweise nacheinander übertragen werden. Als Konsole wird das schwarze Fenster in der Arduino-Entwicklungsumgebung bezeichnet, das die Statusmeldungen des Arduino anzeigt. Lädt man etwa ein Programm auf das Board, wird dort wiedergegeben, ob alles in Ordnung war (z.B. *»Binary sketch size: 1694 bytes (of a 14336 byte maximum)«*) oder Fehler aufgetreten sind. Ist das USB-Kabel nicht richtig verbunden oder enthält der hochzuladende Arduino-Sketch Syntaxfehler, erscheint in der Konsole eine Reihe von roten Meldungen, die über die Natur des Fehlers Auskunft geben sollen. Darüber hinaus gibt es noch eine Reihe weiterer Fehlerquellen, etwa

wenn die Übertragung unterbrochen wurde. Aber auch wenn das Programm einmal hochgeladen ist, kann der Arduino sich noch melden, um beispielsweise Sensorwerte an den PC zu senden. Das kann zum einen genutzt werden, um Daten weiterzuverarbeiten, also z. B. zu speichern oder auf abnormale Messwerte zu reagieren. Zum anderen können vom Arduino geschriebene Nachrichten aber auch zur Fehlersuche verwendet werden.

Die serielle Verbindung arbeitet dabei standardmäßig mit 9.600 Baud. Als ein Baud wird die Einheit »ein Zeichen pro Sekunde« bezeichnet. Da hier ein Zeichen einem Bit entspricht, könnte man also auch »9.600 Bit pro Sekunde« sagen. Es gibt allerdings auch Kommunikationswege, bei denen ein Zeichen aus mehreren Bits besteht, indem das Signal moduliert wird.

Die serielle Konsole einrichten

Die serielle Konsole ist kein eigentlicher Bestandteil des Arduino. Vielmehr muss die Bibliothek *Serial* verwendet werden, die allerdings in der Programmierumgebung schon vorhanden ist. Es ist also nicht nötig, diese Bibliothek gesondert zu laden. Es reicht aus, im Setup-Abschnitt eine Verbindung zu erstellen:

```
void setup() {
   Serial.begin(9600);   // öffne seriellen Port, setze Datenrate
                         // auf 9.600 Baud
}
```

Nun können ankommende Daten gelesen und ausgehende Daten geschrieben werden. Zunächst sollen einfache Informationen in der Konsole der Entwicklungsumgebung angezeigt werden. Das geschieht mit der Funktion `Serial.println()`, die eine Zeile an den angeschlossenen PC sendet, oder auch `print()`, das an die Daten keinen Zeilenumbruch anhängt.

Die Funktionen können dabei vom Nutzer vorgegebenen Text verarbeiten, z. B. so:

```
Serial.println("Hallo")
```

Oder sie verarbeiten Variablen:

```
int zahl = 10;
Serial.println(zahl);
```

Dabei kann einer Variable noch ihre Darstellungsform mitgegeben werden, wenn die Zahl explizit in Dezimaldarstellung angezeigt werden soll, oder in binärer oder hexadezimaler Form:

```
int zahl = 10;
Serial.println(zahl, DEC); // sende "10" an den PC
Serial.println(zahl, BIN); // sende "1010" an den PC
Serial.println(zahl, HEX); // sende "A" an den PC
```

Der Testlauf für das Programm sieht so aus:

```
void loop() {
  int zahl = 10;
  Serial.println("Hallo");
  Serial.println(zahl);
  Serial.println(zahl, DEC);
  Serial.println(zahl, BIN);
  Serial.println(zahl, HEX);
  Serial.println("-----------");
  delay(10000);
}
```

Er bringt folgendes Ergebnis:

◀ **Abbildung 5-1**
Serieller Monitor

Fehlersuche mit der seriellen Konsole

Oft geschieht es beim Programmieren, dass sich Fehler einschleichen, die nicht mehr durch einfaches Durchlesen des Codes zu beheben sind. Je komplexer der Code, desto anfälliger wird er, aber auch gerade im Zusammenspiel mit Sensoren können beispielsweise unerwartete Werte auftauchen. Diese können eine if-Abfrage dazu bringen, einen bestimmten Programmteil gar nicht auszuführen, obwohl die Programmierung es eigentlich vorgesehen hat.

In diesem Beispiel nehmen wir an, von einem Drehknopf wird erwartet, dass er im »Aus-Zustand« eine Null sendet. Ist das der Fall, wird eine LED angeschaltet, ansonsten nicht.

Setup

Wie schon in Kapitel 3 wird ein Drehknopf an den digitalen Pin 11 angeschlossen. Für die Kommunikation bleibt der Arduino per USB mit dem Rechner verbunden.

```
int ledPin = 11;
int drehKnopf = 0;
```

```
void loop() {
  potiWert = analogRead(drehknopf);
    if (potiWert == 0) {
    digitalWrite(ledPin, HIGH);
  }
  else {
    digitalWrite(ledPin, LOW);
  }
}
```

Nun sei für dieses Beispiel angenommen, die LED flackere nun gelegentlich, weil der Sensor nicht präzise arbeitet. Es kann recht lange dauern, das herauszufinden. Sendet man nun den Wert des Drehknopfs an die serielle Konsole und zeigt zudem an, welche Verzweigung das Programm nimmt, kommt man viel schneller zu einem Ergebnis:

```
void loop() {
  potiWert = analogRead(drehknopf);
  Serial.println(potiWert);
    if (potiWert == 0) {
    Serial.println("Poti ist auf null.");
    digitalWrite(LED, HIGH);
  }
  else {
    Serial.println("Poti ist nicht auf null.");
    digitalWrite(LED, LOW);
  }
  delay(1000);
}
```

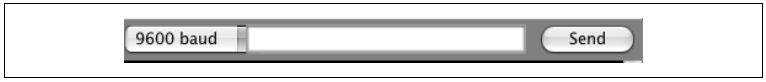
Nachrichten empfangen

Die serielle Konsole eignet sich nicht nur zum Senden von Nachrichten, sondern kann in beide Richtungen kommunizieren. Somit kann der Computer den Arduino auch steuern, nachdem das Programm kompiliert und hochgeladen wurde. Das ist zum Beispiel nötig, um eine volle Steuerung durch Processing zu ermöglichen.

Für den Empfang von Daten wird der TX-Pin genutzt, der bei der Programmierung funktioniert wie die anderen Eingänge auch: Hat man `Serial` initialisiert, kann man über die Funktion `available()` prüfen, ob und wie viele Bytes von Daten an das Board gesendet wurden. Die Funktion `read()` kann diese Daten nun byteweise auslesen und z. B. in einer Variable speichern. In einem einfachen Beispiel werden nun die Daten, die über den seriellen Port empfangen wurden, gelesen und wieder über die Konsole ausgegeben:

```
char incomingByte = 0;      // Variable für die einkommenden Daten
void setup() {
          Serial.begin(9600); // initialisiere seriellen Port auf
                              // 9.600 Baud
}
void loop() {
         // wenn serielle Daten vorhanden sind
         if (Serial.available() > 0) {
           // lies das einkommende Byte
           incomingByte = Serial.read();
           // sende die Daten zurück an den PC
           Serial.print("Empfangen: ");
           Serial.println(incomingByte);
         }
}
```

Um diese Daten zu senden, muss wieder einmal die serielle Konsole in der Arduino-Programmierumgebung geöffnet werden. Über dem schwarzen Ausgabefenster befindet sich eine kleine Textzeile mit einem SEND-Button, der dafür genutzt werden kann.



◀ **Abbildung 5-2**
Eingabezeile, Send-Button

Gibt man nun **`test`** ein, erscheinen die vier Buchstaben nacheinander auf dem Bildschirm, da `read()` immer nur ein Byte auf einmal empfängt und die darauf folgenden Bytes in einem so genannten *Ringpuffer* zwischenhält, der nach und nach ausgelesen werden kann. Ein Ringpuffer ist eine Datenstruktur, die mehrere Elemente (zum Beispiel Bytes) speichert und nach und nach bei jedem Aufruf zurückgibt.

◀ **Abbildung 5-3**
Empfangen im Monitor

Möchte man nun die gesamten Eingaben zusammen speichern, benötigt man ein Array, das man langsam füllt.

Legt man also

```
char incomingByte[255] = {};
```

fest, kann man den einkommenden Text auch so abspeichern, um ihn dann weiterzuverwenden, wenn man den Ringpuffer so lange aufruft, bis er leer ist.

```
while (Serial.available() > 0) {
  // lies das einkommende Byte
  incomingByte[i] = Serial.read();
       i++;
// ein wenig warten, damit das nächste Byte auch sicher
// übertragen wird
delay(3);
}
```

Möchte man nun auch das nächste Wort korrekt anzeigen, muss das Array geleert werden, da sonst noch unerwünschte Zeichen gespeichert sind. Das geschieht, indem das komplette Array durchlaufen und alle Elemente mit NULL, also einem leeren Zeichen überschrieben werden:

```
for (int i=0; i<=255; i++) {
    incomingByte[i] = NULL;
}
```

Steuerungseingaben

Jetzt können diese Eingaben verwendet werden, um zum Beispiel die LED auf dem Arduino-Board einzuschalten. Dazu müssen Pin 13 als Ausgang definiert und die serielle Konsole initialisiert werden:

```
int ledPin = 13;
void setup() {
    Serial.begin(9600);  // initialisiere seriellen Port auf 9.600
                         // Baud
    pinMode(ledPin, OUTPUT); // setze ledPin auf OUTPUT
}
```

Nun kann man beispielsweise die LED einschalten, sobald eine 1 empfangen wird. Eine 0 schaltet sie wieder aus:

```
void loop()
{
  if (Serial.available() > 0) {
    if (Serial.read() == '1') {
      digitalWrite(ledPin, HIGH);
    }
    else if (Serial.read() == '0') {
      digitalWrite(ledPin, LOW);
    }
  }
}
```

Dabei ist zu beachten, dass die Texteingabe in der Programmierumgebung als Buchstaben gesendet werden. Eine eingetippte »0« hat in der dabei verwendeten ASCII-Tabelle den Wert 48. Es kann also entweder mit 48 oder mit '0' verglichen werden, wobei der Compiler '0' durch 48 ersetzt.

Automatisierung mit Gobetwino

Gobetwino wurde vom Dänen Mikael Morup entwickelt, um sich größere Mühen zu ersparen, wenn ein PC mit dem Arduino kommunizieren soll. Es erlaubt zum Beispiel, Mails oder Chatnachrichten mit einer Ampel anzuzeigen. Nicht immer will man dafür das teure Ethernet-Shield aus Kapitel 6 verwenden oder speziell eine Anwendung in Processing programmieren, wie es am Ende dieses Kapitels beschrieben wird. Ist der Arduino bereits mit einem PC über USB verbunden, kann man unter Windows das Programm *Gobetwino* verwenden, das die serielle Verbindung verwendet. Die Software fungiert dabei als sogenannter Proxy, ein Vermittler, der die serielle Kommunikation übernimmt. Er ist in der Lage, Kommandos zu senden und zu empfangen. Diese können dann verwendet werden, um verschiedene Aufgaben abzubilden. So muss Gobetwino nicht erst umständlich programmiert werden. Stattdessen legt man über ein Drop-down-Menü fest, welche Aktionen wann ausgeführt werden sollen. Gobetwino kann

- ein Programm starten (und wenn gewünscht auf das Beenden warten),
- Daten vom Arduino zu einem Programm als Tastendrücke senden,
- E-Mails mit Daten vom Arduino versenden,
- Dateien aus dem Internet herunterladen und Teile davon an den Arduino senden,
- Daten aus dem Arduino in ein File schreiben (mit Timestamp) und
- eingehende E-Mails überprüfen.

Das funktioniert recht einfach, indem im Gobetwino-Programm mit einer Eingabemaske Funktionen angelegt werden. Dazu wählt man den Funktionstyp aus einer Drop-down-Box aus und gibt die dazu nötigen Parameter wie »Datei, in der Daten gespeichert werden sollen« an. Das genügt, um den Gobetwino zu konfigurieren, alles Weitere regelt das Programm von allein. Gobetwino können Sie unter *http://www.mikmo.dk/gobetwinodownload.html* herunterladen. Neben dem kompletten Programm enthält das ZIP-Archiv ein englischsprachiges Handbuch, in dem genau erklärt wird, wie man neue Befehle anlegt.

Damit Gobetwino erkennen kann, wo Nachrichten vom Arduino anfangen und aufhören, sind diese in Rauten (#) eingeschlossen. Nach dem ersten # kommt immer ein großes S und danach eine Pipe (|); darauf folgt dann der groß geschriebene Kommandoname, der in Gobetwino festgelegt werden kann. Nach einer weiteren Pipe folgen die Parameter, die von eckigen Klammern umschlossen werden und durch & getrennt sind.

Um zum Beispiel Messwerte in eine Logdatei zu speichern, genügt folgende Zeile:

```
#S|LOGTEST|[877]#
```

In Gobetwino müssen dann nur noch das Kommando `LOGTEST` angelegt und festgelegt werden, in welches File die Daten gespeichert werden sollen. Optional kann ein Zeitstempel hinzugefügt werden.

Auf dem Arduino sieht der passende Code dann so aus:

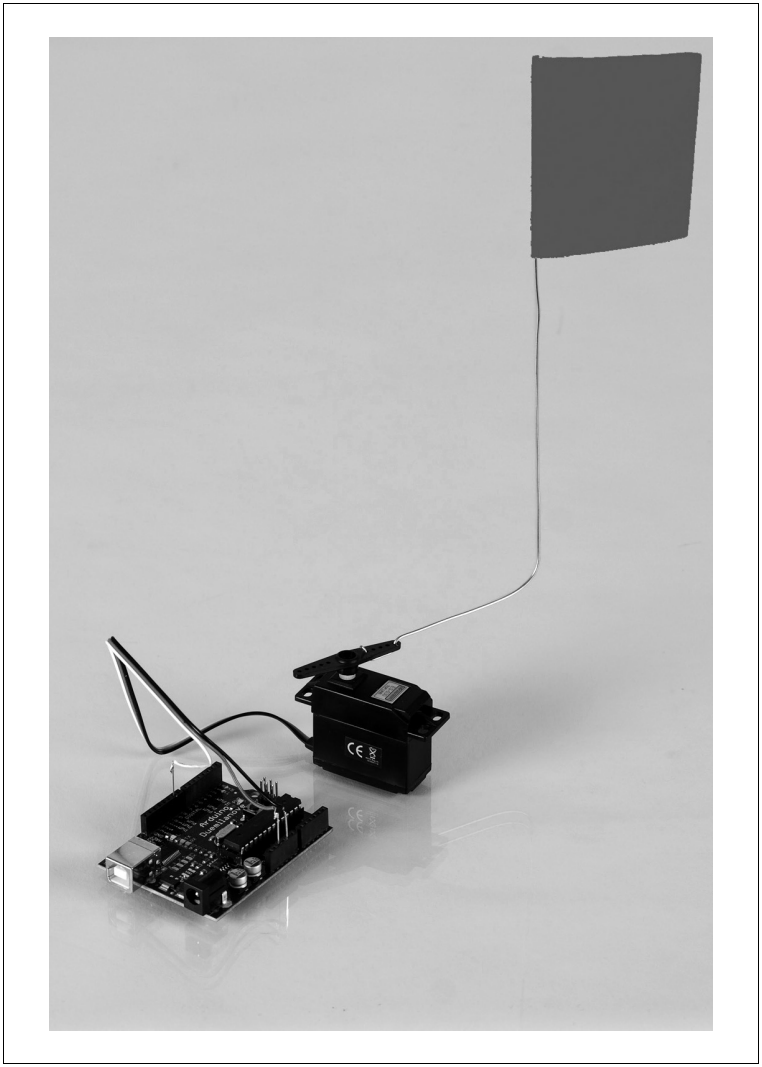
```
int value = 877;
char buffer[5];
Serial.print("#S|LOGTEST|[");
Serial.print(value);
Serial.println("]#");
```

Wichtig ist in diesem Beispiel, dass die letzte Zeile `println()` anstatt `print()` verwendet, wodurch ein Zeilenumbruch gesendet wird, der Gobetwino das Ende der Nachricht signalisiert.

In Gobetwino kann man danach sehr schön sehen, wie diese Nachricht eingelesen und verarbeitet wird. Hier erkennt man dann auch, wenn einmal etwas nicht ganz so läuft, wie es sollte.

Lebst du noch?

Wir wollen mit einem Ping die Erreichbarkeit unseres Webservers überprüfen und im Fehlerfall eine rote Fahne mit einem Servomotor (kurz: Servo) hissen. Das ist ein kleiner Motor aus dem Modellbauladen, den man in Geschwindigkeit und Position steuern kann. Das ist notwendig, damit der Motor am Ende der Fahnenstange auch wieder anhält. Die Fahnenstange kann man dabei wie eine echte basteln: Auf der einen Seite ist eine Fahne an einem kleinen Draht angebracht, der um eine Rolle gewickelt wird. Diese Rolle wird dann vom Servo gedreht.



◀ **Abbildung 5-4**
Gobetwino-Aufbau

Hardware

Als Erstes wird der Servo angeschlossen. Ein Servo hat die drei Anschlüsse GND (schwarz oder braun), VDD (rot) und SIGNAL. GND wird mit GND auf dem Arduino-Board verbunden, VDD mit 5V und SIGNAL mit PIN9. Servos werden über kurze Impulse von 1–2 ms Dauer gesteuert, die sich mit ca. 50 Hz wiederholen. Die Länge der Pulse ergibt die Position, die der Servo anfahren soll.

Man kann sich vorstellen, dass die Pulsdauer ziemlich exakt sein sollte, weil der Servo sonst hin- und her- ruckelt. Da ist es am besten, den PWM-Schaltkreis auf dem Arduino-Board zu verwenden, der unterschiedlich lange Pulse erzeugen kann. Da das PWM-Signal für PIN10 und PIN9 mit demselben PWM-Schaltkreis des Atmel erzeugt wird, können beide PINs nur entweder für `analogWrite()` oder `Servo.write()` verwendet werden.

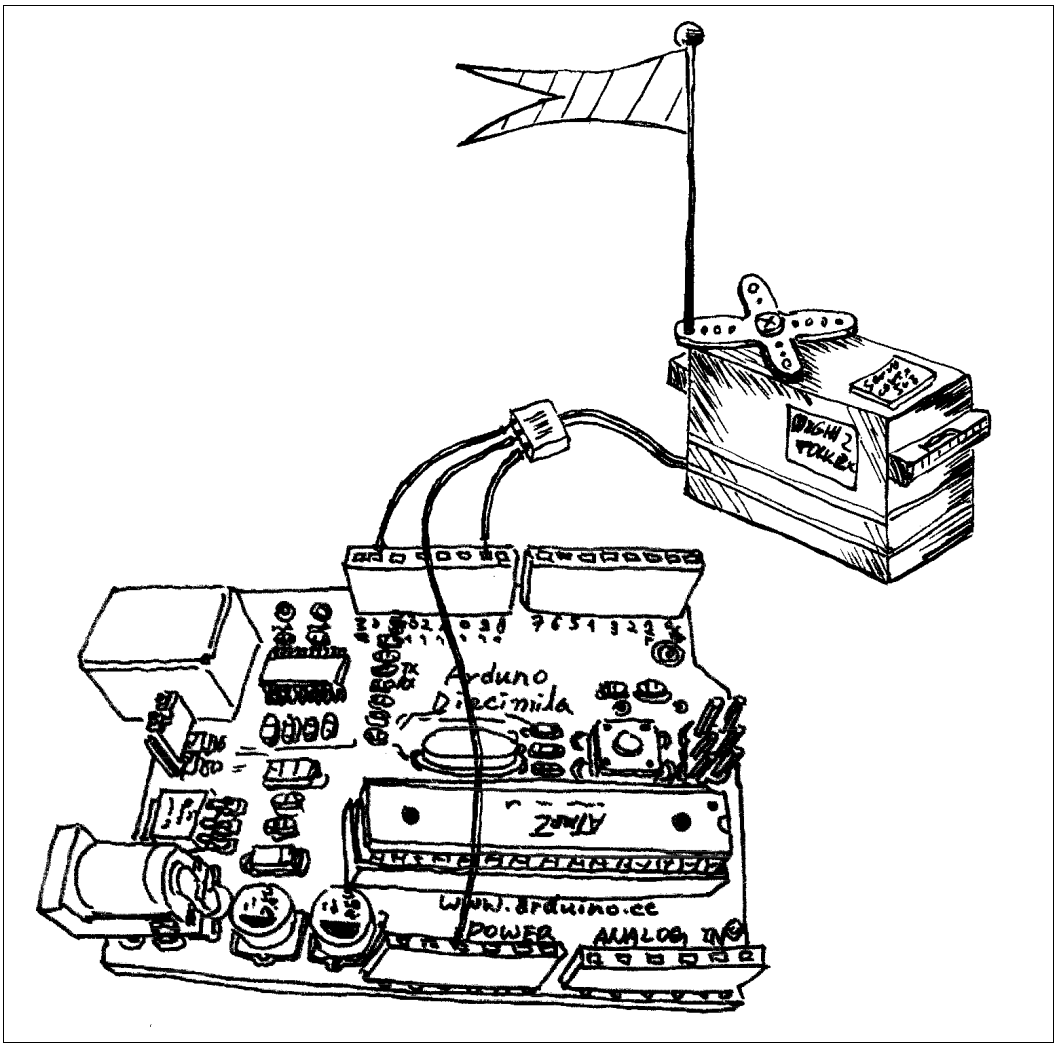


Abbildung 5-5 ▲
Gobetwino-Schaltung

Am Drehrad des Servos befestigt man jetzt noch die Fahne, und schon kann es losgehen mit dem Schreiben der Software.

Software

Als Erstes werden ein paar globale Variablen und eine Instanz der Servo-Klasse benötigt. Diese wird mit dem PIN9 des Arduino verbunden, an dem der Servo angeschlossen ist.

```
#include <Servo.h>
Servo myservo;               // erzeuge ein Servo-Objekt
int pos;                     // gewünschte Position des Servos
int serInStringLen = 25;     // wie viele Zeichen darf eine
                             // Nachricht vom PC maximal lang sein
char  serInString[serInStringLen]; // Speicherplatz für
                                   // Nachrichten
void setup()
{
  myservo.attach(9);   // verbinde das Servo-Objekt mit Pin 9
  Serial.begin(9600);  // initialisiere RS23-Verbindung zum PC
  delay(3000);         // warte ein wenig
}
```

Für die Kommunikation mit dem PC ist es hilfreich, wenn es eine Funktion gibt, die auf eine ganze Zeile Daten vom PC wartet, um diese am Stück auswerten zu können. Zunächst wird geprüft, ob die Zeit abgelaufen ist; ist das der Fall, beendet die Funktion sich, allerdings nicht ohne an das Ende des String noch eine 0 zu schreiben: Das ist das Zeichen dafür, dass ein String an dieser Stelle zu Ende ist, andernfalls könnten spätere Funktionen darüber stolpern.

Dann wird mit `Serial.available()` geprüft, ob überhaupt Zeichen vom PC aus gesendet worden sind; wenn ja, werden sie abgeholt und in das String-Array gespeichert, sofern es kein Zeilenendzeichen (ASCII-Code 13) war.

```
void readSerialString (char *strArray, int serInLen, long timeOut)
{
  // warte timeOut Milisekunden auf Daten
  long startTime=millis();
  int i = 0;
  char ch;
  while ((millis()-startTime < timeOut)) {
    if (Serial.available()) {
      ch = Serial.read();
      if (ch == 13) {
      strArray[i] = 0;
      return;
    }
    strArray[i] = ch;
    if (i < serInLen) i++;
  }
  strArray[i] = 0;
}
```

In der `loop()`-Funktion wird nun regelmäßig über Gobetwino abgefragt, ob der Server erreichbar ist:

```
void loop()
{
  Serial.println("#S|PING|[]");
  readSerialString (serInString, serInStringLen, 10000);
  int val = atoi(serInString);
  if (val == 0) {
    if (pos < 255) pos++;
  } else {
    if (pos > 0) pos--;
  }
  myservo.write(pos);
  delay(100);
}
```

Dieser Code erwartet, dass in Gobetwino ein Kommando mit dem Namen `PING` angelegt wird, das den zu überwachenden Server anpingt. Es sendet dann eine 0, wenn alles okay ist, und eine –1, wenn der Ping fehlschlägt. Als Reaktion darauf hebt oder senkt sich dann unsere Fahne, sodass man immer auf den ersten Blick den Zustand unseres Servers erkennen kann.

Dazu öffnet man Gobetwino und erzeugt über COMMANDS → NEW COMMAND ein neues Kommando. Als Typ wählt man PING und trägt die zu testende Adresse ein. Damit ist Gobetwino auch schon so weit konfiguriert, dass das Projekt getestet werden kann.

Processing

Processing ist in vielerlei Hinsicht eines der Vorbilder für Arduino, dessen Programmierumgebung und -sprache sehr ähnlich modelliert sind. Wie Arduino wurde Processing entwickelt, um Schülern und Studenten eine einfache Möglichkeit zu geben, programmieren zu lernen und Ideen auszuprobieren, um zu schnellen Ergebnissen zu gelangen. Und natürlich gibt es auch hier mittlerweile eine große Anhängerschaft, vor allem unter Künstlern und Bastlern.

Dabei hat Processing zunächst rein gar nichts mit Physical Computing und Mikrocontrollern zu tun; vielmehr wurde es geschaffen, um möglichst einfache Programmierung von Grafik zu ermöglichen. So läuft der Einstieg dort nicht über eine blinkende LED, sondern vielmehr über die Anzeige des Textes »Hello World« in einer großen, bunten Schrift. Mittlerweile gibt es eine Vielzahl unterschiedlicher Projekte, die auf Processing basieren. Längst gibt es Bibliotheken, die es sogar Bands wie R.E.M. ermöglichen, ihre Kon-

zerte visuell zu untermalen oder eigene Musikvideos zu erschaffen. Andere Module erlauben eine Nutzung, die noch viel weiter geht als das Bespielen eines Bildschirms. Sounds werden gesteuert und Daten visualisiert, zum Beispiel in Form eines Oszillators. Die Spikenzie Labs verwenden Processing, um Signale auszuwerten und in Midi umzuwandeln (siehe *http://www.spikenzielabs.com/SpikenzieLabs/Serial_MIDI.html*). So kann aus dem Arduino ein Midi-Controller gebaut werden, ohne entsprechende Bauteile zu benötigen. Processing übernimmt die weitere Kommunikation mit einem Midi-Programm. Das Augmentation-Blog (*http://augmentation.wordpress.com/*) verwendet einen selbstgebauten Datenhandschuh und Arduino, um im Zusammenspiel mit Processing ein selbstgebautes Motion-Capturing-System zu schaffen. Andersherum kann man zum Beispiel Processing verwenden, um einen Arduino zu steuern. In diesem Fall wird Musik in Daten umgewandelt, die die Helligkeit von LEDs steuern (siehe dazu *http://invalidfunction.com/index.php/2009/03/arduino-controlled-music-lights/*).

Auch wenn Processing auf Java basiert und somit einige Programmkonstrukte deutlich anders sind als beim auf C++ basierenden Arduino, gibt es einige Gemeinsamkeiten in der Sprache.

So werden die zum Programmablauf benötigten Teile ebenfalls in einer Setup-Funktion geladen. Hier werden Variablen deklariert und initialisiert und Elemente, die für das Programm nötig sind, eingerichtet. Möchte man beispielsweise einen Text in einer entsprechenden Größe und mit einer entsprechenden Schriftart anzeigen, legt man genau diese beiden Parameter in `setup()` fest:

```
void setup()
{
 PFont font = loadFont("myfont.vlw");
 textFont(font,20);
}
```

Nun sind alle Variablen festgelegt und können so in der Hauptfunktion, die in Processing `draw()` heißt, verwendet werden. In diesem Beispiel wird die Funktion `text()` verwendet, um einen Text anzuzeigen. Die beiden weiteren Parameter stellen die Position auf der x- und y-Achse dar, also wird »Hello World!« auf Position 30:50 in Schriftgröße 20 und der Schriftart myFont ausgegeben.

```
void draw()
{
 text("Hello World!", 30,50);
}
```

Möchte man nun Arduino und Processing verbinden, benötigt man zunächst Bibliotheken für beide Seiten. Beim Arduino ist diese Bibliothek schon in der Programmierumgebung enthalten. Sie nennt sich *Firmata* und kann mit `#include <Firmata.h>` ins Programm eingebunden werden. Alternativ kann man das unter SKETCH → IMPORT LIBRARY → FIRMATA tun. Unter FILE → SKETCHBOOK → EXAMPLES → LIBRARYFIRMATA finden Sie den gesamten passenden Firmata-Sketch, der alles kann, was in diesem Workshop benötigt wird.

Lädt man diesen Beispielsketch auf das Arduino-Board, ist es konfiguriert, um mit Processing zusammenzuarbeiten. Dabei muss im Prinzip keine weitere Programmierung auf dem Arduino vorgenommen werden. Wie das folgende Beispiel zeigt, kann Processing die Pins des Arduino selbstständig steuern und so auch die Signale verarbeiten.

Zunächst muss also die Arduino-Bibliothek in Processing eingebunden werden. Sie ist auf der Arduino-Website unter *http://www.arduino.cc/playground/Interfacing/Processing* erhältlich. Nachdem sie heruntergeladen wurde, wird sie entpackt und der Ordner *arduino* in den Ordner *libraries* der Processing-Programmierumgebung kopiert.

Nun können im Processing-Sketch die beiden Bibliotheken *serial* und *arduino* geladen und ein Arduino-Objekt erstellt werden:

```
import processing.serial.*;
import cc.arduino.*;
// initialisiere Arduino-Objekt
Arduino arduino;
```

Nun soll die LED 13 zum Blinken gebracht werden, wie auch schon im ersten Workshop dieses Buches. Zunächst wird also die Variable `ledPin` auf 13 gesetzt. Anschließend wird im Setup die Verbindung hergestellt. Dazu wird die Funktion `arduino.list()` verwendet, die alle verfügbaren seriellen Geräte auf diesem Computer anzeigt. Wenn das Arduino-Board verbunden ist, sollte dort eine passende serielle Verbindung in dieser Liste auftauchen. Hierbei wird nun angenommen, dass es sich um das einzige Gerät handelt und somit an Stelle 0 in der Liste befindet. Funktioniert `Arduino.list()[0]` nicht auf Anhieb, empfiehlt es sich, in der Konsole diese Liste auszugeben und die richtige Verbindung zu suchen.

Anschließend wird, wie schon bekannt, `ledPin` als Ausgang definiert.

```
void setup()
{
  // falls Arduino nicht das erste serielle Gerät ist,
  // die Kommentar-Slashes der nächsten Zeile entfernen und die
  // Liste anzeigen lassen
  // println(Arduino.list());
  // initialisiere die Arduino-Verbindung
  arduino = new Arduino(this, Arduino.list()[0]); // v2
  // setze ledPin auf OUTPUT
  arduino.pinMode(ledPin, Arduino.OUTPUT);
}
```

Auch der Rest des Programms verhält sich ähnlich wie schon im Workshop 3: Nun kann die LED alle 1.000 Millisekunden blinken, indem ein `HIGH`- bzw. `LOW`-Signal auf den entsprechenden Pin gesetzt wird:

```
void draw()
{
  arduino.digitalWrite(ledPin, Arduino.HIGH);
  delay(1000);
  arduino.digitalWrite(ledPin, Arduino.LOW);
  delay(1000);
}
```

Processing mit einem Drehknopf steuern

Nun soll ein Drehknopf (also ein Potentiometer) dazu verwendet werden, die Größe eines einfachen Kreises in Processing zu steuern.

Das Setup ist ebenfalls schon bekannt: ein Potentiometer wird mit einem analogen Pin (in diesem Falle Pin 0) verbunden, der als analoger Eingang nicht gesondert eingerichtet werden muss. Um den Drehknopf später besser verwenden zu können, wird die Variable `int potiPin = 0` definiert.

Nun kann der Input dieses Knopfes mit `arduino.analogRead(potiPin);` von Processing eingelesen werden. Für dieses Beispiel soll ein Kreis eine Größe von 0 bis 100 Pixeln Durchmesser annehmen. Dafür wird die Funktion `ellipse()` verwendet, die eine Ellipse zeichnet, bei der beide Achsen gleich lang sind. Da der Drehknopf 1.024 Werte übermitteln kann, soll der Kreis alle 10 Werte (abgerundet) um einen Pixel anwachsen. Diese grobe Übersetzung hilft auch, kleinere Ungenauigkeiten bei der Messung abzumildern, denn die werden unweigerlich auftreten.

Damit Processing die Kreise nicht übereinanderlegt, muss zudem pro Zeichenschritt der Bildschirm gelöscht werden. In diesem Fall

geschieht das, indem der Hintergrund mit `background(204);` in Grau neu gezeichnet wird. Um ein wenig mehr Farbe hineinzubringen, wird der Kreis selbst in einem satten Rot gezeichnet (RGB-Werte: 255, 0, 0).

Die fertige Draw-Routine sieht also wie folgt aus:

```
int potiPin = 0;
int potiVal = 0;
void draw() {
    potiVal = arduino.analogRead(potiPin);
    background(204);
    fill(255, 0, 0);
    ellipse(50, 50, floor(potiVal/10), floor(potiVal/10));
}
```

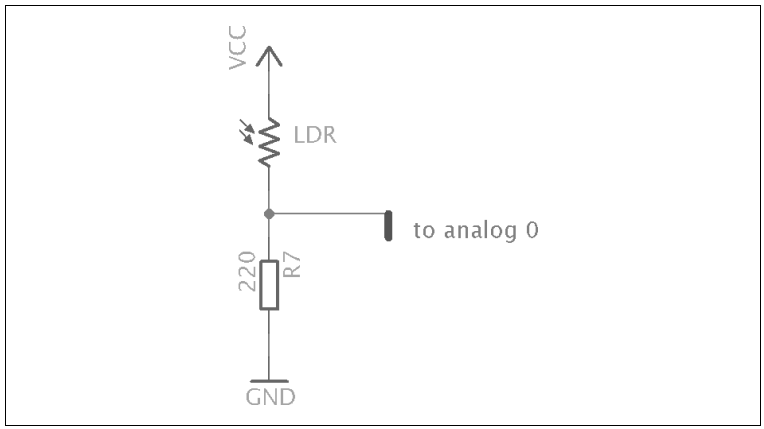
Let's Play – ein Spiel mit Lichtsensor

Zum Abschluss dieses Kapitels folgt nun ein kleines Processing-Spiel, das über einen am Arduino angeschlossenen Lichtsensor gesteuert werden soll. Dieser Sensor steuert einen kleinen Punkt (oder Ball), der langsam auf der Horizontale durch den Bildschirm fliegt. Fällt dabei mehr Licht auf den Sensor, fliegt der Ball nach oben, wird es dunkler, senkt er sich. Auf dem Bildschirm wird nun ein Bereich angezeigt, der durchflogen werden darf. Verlässt der Ball diesen Bereich, ist das Spiel beendet, und die Punkte werden angezeigt.

Setup

Ein Fotowiderstand wird an den analogen Eingang 0 des Arduino angeschlossen.

Abbildung 5-6 ▶
Lichtsensor-Aufbau



Programmierung

Zunächst werden alle Parameter des Spiels (also Durchmesser, Fenstergröße und Startposition) festgelegt und die Bibliotheken importiert:

```
import processing.serial.*;
import cc.arduino.*;
// initialisiere Arduino-Objekt
Arduino arduino;
// lege Fenstergröße fest
int windowWidth = 600;
int windowHeight = 400;
// horizontale Position
int x = 0;
// fand eine Kollision statt?
boolean collision = false;
// Durchmesser, Startzeit und Endzeit (für die Punktezahl)
int durchm = 20;
int startTime = 0;
int endTime = 0;
PFont fontA;
// Startposition und Startwert
int arduinoPos = 200;
boolean start = false;
// Pin für den Lichtsensor
int sensorPin = 0;
```

Anschließend werden in der Setup-Routine der Arduino initialisiert und das Fenster gezeichnet. Zudem wird ein Font festgelegt, mit dem am Ende des Spiels die Punktezahl angezeigt wird.

```
void setup() {
  size(windowWidth, windowHeight);        //  Fenstergröße
  background(0);                          //  Hintergrundfarbe
  createScene();                          //  Funktion, die den
                                          //  Tunnel zeichnet
  //  setze Font fest (muss im Datenverzeichnis liegen)
  fontA = loadFont("Ziggurat-HTF-Black-32.vlw");
  //  initialisiere den Font und seine Größe (in Pixeln)
  textFont(fontA, 16);
}
```

Als Nächstes werden Funktionen benötigt, um die »Kugel« (also einen Kreis) und den Tunnel zu zeichnen. Aus Platzgründen läuft dieser Tunnel nur schräg von links oben nach rechts unten. Natürlich können auch mit der Vertex-Funktion kompliziertere Level gebaut werden.

```
// zeichne den Ball
void drawCircle(int x, int y) {
  fill(255);
  ellipse(x, y, durchm, durchm);
```

```
  noFill();
}
// zeichne den Tunnel mit drei Vertexen
void createScene() {

  stroke(204, 102, 0);
  fill(204, 102, 0);
  //  Oberes Feld
  beginShape();
    vertex(0, 0);
    vertex(0, 100);
    vertex(windowWidth,300);
    vertex(windowWidth,0);
  endShape(CLOSE);

  // unteres Feld
  stroke(204, 102, 0);
  fill(204, 102, 0);
  beginShape();
    vertex(0, windowHeight);
    vertex(0, 200);
    vertex(windowWidth, 400);
    vertex(windowWidth, windowHeight);
  endShape(CLOSE);

  //  mittleres Feld
  fill(125);
  stroke(0);
  beginShape();
    vertex(0, 100);
    vertex(windowWidth, 300);
    vertex(windowWidth, 400);
    vertex(0, 200);
  endShape();
}
```

Schließlich wird noch eine Kollisionskontrolle benötigt. Eine Kollision findet statt, wenn die Kugel außerhalb des mittleren Feldes fliegt. Das Feld ist dabei durch zwei schräge Geraden bestimmt. Durch einfache Geometrie findet man heraus, dass die Gerade genau proportional zum aktuellen x-Wert verläuft. Das Feld ist also begrenzt durch *x / Fensterbreite * Höhenunterschied + y-Anfangswert*, also `x/windowWidth*200+100`. Zudem findet eine Kollision statt, wenn die Kugel den rechten Rand des Fensters erreicht.

```
// prüfe, ob die Kugel eine der Linien berührt
boolean detectCollision(int x, int y){
  // Ergebnis ist entweder wahr oder falsch
  boolean result = false;
  // wenn der Mittelpunkt der Kugel den rechten Rand berührt hat
  if (x >= windowWidth-durchm/2) {
    result = true;
```

```
  }
  // wenn die Kugel über der oberen Linie liegt
  if (y < (((x/windowWidth) * 100)+100)) {
    result = true;
  }
  // wenn die Kugel unter der unteren Linie liegt
  if (y > (((x/windowWidth) * 100) + 200)) {
    result = true;
  }
  return result;
}
```

Nun sind alle Voraussetzungen für das Spiel gegeben. Die Funktion `endGame()` zeigt einen Text mit der Punktezahl an, die sich aus der Dauer in Millisekunden ergibt, die das Spiel gelaufen ist. Die Hauptfunktion lädt nun die Grafik und beginnt das Spiel nach Druck der Taste S.

```
// Anzeige beim Ende des Spiels
void endGame(){
  // berechne die Endzeit
  endTime = millis();
  // beende die Draw-Funktion
  noLoop();
      // zeige weißen Text an
  fill(0);
  text("your score: ", 50, 60);
  // Punktezahl berechnen in Millisekunden seit Startzeit
  text(endTime-startTime, 200, 60);
  // Hier wäre noch platz für einen Restart-Knopf ;-)
}
// Hauptfunktion
void draw() {
  // lies Arduino-Eingabe und wandle sie um
  int arduino = Arduino.analogRead(sensorPin);
  // wenn Taste S gedrückt wird, beginn und miss Startzeit
  if(keyPressed) {
    if (key == 's') {
      start = true;
      // miss Startzeit
      startTime = millis();
    }
  }
  // wenn das Startsignal gegeben wurde
  if (start) {
    // wenn keine Kollision geschehen ist
    if (!collision) {
      createScene();
      stroke(0);      // setz die Farbe für den Ball
      drawCircle(x, arduino);  // zeichne den Ball
      collision = detectCollision(x, arduino); // prüf, ob es eine
                                               // Kollision gab
```

```
      x++;                          //  erhöhe die Ballposition
    // wenn es eine Kollision gab
    } else {
      // rufe Endfunktion auf
      endGame();
    }
  // wenn das Startsignal noch nicht gegeben wurde
  } else {
    // zeichne den Tunnel und den Ball
    createScene();
    drawCircle(0, arduino);
  }
}
```

Nun ist das Spiel fertig zum Ausprobieren. Wahrscheinlich muss je nach Lichteinfluss ein wenig mit den Werten des Sensors herumprobiert werden, indem man sie zum Beispiel teilt. Nur so kann je nach Sensor das Spiel auch bei der gewünschten Höhe der Hand funktionieren, ansonsten kann es sein, dass die Kugel sich nur innerhalb weniger Millimeter bewegt. Für mehr Spielspaß empfiehlt es sich aber ohnehin, das Spiel ein ganzes Stück zu erweitern.

Aufgaben:

- Ein Labyrinth basteln, sodass x- und y-Achse ab einem bestimmten Punkt wechseln und die Kugel vertikal am Labyrinth entlang nach unten fliegt.Überschreitet die Kugel also einen gewissen x- oder y-Wert, wird ihre Richtung geändert.
- Eine Reset-Funktion einbauen, die zum Beispiel auf Knopfdruck das Spiel von vorn beginnen lässt.
- Einen Piezo-Lautsprecher an den Arduino anschließen und bei einer Kollision einen Ton ausgeben lassen.

Flusskontrolle

Ein Programm, dessen Ablauf stark von der Übertragung serieller Daten abhängig ist, kann schnell ins Stocken geraten, wenn die Kommunikation asynchron läuft. Obwohl klar ist, welche Daten übertragen werden, kann es sein, dass zum Beispiel Daten vom Arduino zu einem falschen Zeitpunkt an den PC gesendet werden. Wenn zum Beispiel ein Processing-Programm Daten empfängt, die schon vor einiger Zeit gesendet wurden, können diese schon längst ungenau oder falsch sein. Möglicherweise hat der Arduino schon ein zweites Byte gesendet, das nun im Ringpuffer hängen geblieben ist.

Um dieses Problem zu vermeiden, wird eine Flusskontrolle eingesetzt. Das bedeutet, dass der Datenfluss zeitweise unterbrochen wird, damit der schnellere Sender keinen Datenüberfluss produziert. Beim Beispiel des Spiels wird der Arduino-Sensor direkt gelesen. Würden die Daten aber über eine serielle Verbindung übertragen, etwa weil sie vorher zunächst verarbeitet werden müssen, könnte es sein, dass eine schnelle Reaktion des Spielers möglicherweise zu spät ankommt; eine Kollision würde nicht vermieden, obwohl der Spieler es eigentlich geschafft hätte.

Nun könnte man den Arduino natürlich in seiner Sendefrequenz beschränken, indem man ihm Pausen auferlegt. Viel sinnvoller ist es aber, wenn er auf ein Signal von Processing wartet, um selbst aktiv zu werden. Das Arduino-Programm wird also verändert, sodass das Signal erst gesendet wird, wenn ein Byte von Processing empfangen wurde. Die Sensorabfrage geschieht jedoch weiterhin kontinuierlich, um die Reaktionszeit des Boards zu verringern:

```
void loop() {
     // lies Sensor
     sensor = analogRead(sensorPin);
     // verarbeite Sensordaten ...
     . . .
     if Serial.available() > 0) {
        int seriellesByte = Serial.read();
        Serial.print(sensorInfo, HEX);
     }
}
```

Das Processing-Programm muss nun erweitert werden, um vor dem Empfang der Daten selbst ein `Serial.print()` abzusetzen. Das kann eine Null oder auch ein Zeilenumbruch sein.

```
Serial.print('\r', BYTE);
arduinodaten = Serial.read();
```

Mangelnde Flusskontrolle führt häufig zu schwer erkennbaren Fehlern. Da die falschen Daten oft vom Menschen nicht bemerkt werden, treten subtile Probleme auf, die nur spät oder gar nicht entdeckt werden.

Weiter mit dem dynamischen Duo

Wie gezeigt wurde, erweitert Processing den Arduino und eröffnet hervorragende Möglichkeiten. Viele Projekte verwenden diese Kombination z. B., um Sensorinformationen zu visualisieren, ob nun in ernsthaften Anwendungen oder für ästhetische Effekte. Andere nutzen den Arduino wiederum als Eingabegerät, um Drehknöpfe oder

Schieberegler an einen PC anzuschließen, oder – wie eben gesehen – Spiele zu steuern. Bei der Entwicklung komplexerer Systeme, etwa von Flugobjekten, können die durch Processing verarbeiteten Daten auch dabei helfen, Fehler zu finden oder Feineinstellungen vorzunehmen. So werden mögliche Probleme schon vermieden, bevor sie auftreten und Schaden anrichten können. Man denke dabei zum Beispiel an Beschleunigungs- und Neigungssensoren in einer Flugdrohne, die bei falscher Konfiguration abstürzen würde. Auch in der Wissenschaft wird Processing in Verbindung mit Mikrocontrollern wie Arduino immer beliebter, um Versuchsreihen möglichst einfach zu gestalten und sich so auf das Wesentliche zu konzentrieren.

Im folgenden Kapitel soll nun die Kommunikation des Arduino nicht mehr nur über die serielle Schnittstelle geschehen. Mit einem Ethernet-Shield, also einer »Netzwerkkarte«, die auf das Board gesteckt wird, können Daten auch über das Internet übertragen werden. Damit lässt sich ein Arduino-Projekt auch aus der Ferne warten oder abfragen, was insbesondere dann sinnvoll ist, wenn es das tägliche Leben in der eigenen Wohnung automatisieren soll, auch während man gar nicht zu Hause ist.

KAPITEL 6

Arduino im Netz

In diesem Kapitel:

- Hello World – ein Mini-Webserver
- Sag´s der Welt mit Twitter
- Fang die Bytes – Datalogger

Das Internet wird immer mehr zu einem ständigen Begleiter. So ist es nicht verwunderlich, dass es vermehrt den Wunsch gibt, auch ohne ständig laufenden Rechner Daten aus dem Internet zu empfangen und über es zu versenden. Mit einem aufgesteckten Ethernet-Shield kann Arduino auch ganz ohne PC online gehen. Das schafft viele Möglichkeiten, um die physische Welt mit dem Netz kommunizieren zu lassen. Vorstellbar sind z. B. Sensoren, die ihren aktuellen Status bekannt geben und somit ihren Besitzer auch unabhängig von seinem Aufenthaltsort informieren können. Das folgende Kapitel erklärt zunächst den Ethernet-Shield und die einfache Kommunikation über das Internet. Danach soll der Arduino einfache Nachrichten über Twitter versenden.

Im in Kapitel 5 erwähnten Schichtenmodell basiert eine Internetverbindung zunächst auf Ethernet. Die physikalische und logische Schicht werden offiziell als »Bitübertragungsschicht« zusammengefasst, was einen Teil von Ethernet ausmacht. Der andere Teil ist die Datensicherungsschicht, die beschreibt, wie eine fehlerfreie Übertragung gewährleistet werden kann. Dazu werden die Daten in Blöcke eingeteilt und mit Folgenummern und Prüfsummen versehen.

In der Bitübertragungsschicht wird das Gerät über eine Hardware-Adresse, die sogenannten MAC-Adresse, identifiziert. Diese Adresse ist in der Regel vom Hersteller für jedes Gerät fest vorgegeben und ändert sich auch dann nicht, wenn es seinen Standort wechselt und sich zum Beispiel in einem anderen lokalen Netzwerk befindet oder an einer anderen Stelle mit dem Internet verbunden wird. Diese Hardware-Adresse besteht aus sechs hexadezimalen Zahlen, zum Beispiel 00:11:43:8C:D3:91.

Über der Ethernet-Schicht sorgt das Internet Protocol (IP) für eine Identifikation des Gerätes mithilfe der Netzwerkadresse. Netzwerk-

oder IP-Adressen dürften vielen Nutzern bekannt sein, die sich schon einmal beispielsweise mit der Einrichtung ihres heimischen PCs im lokalen Netzwerk beschäftigt haben. Um eine hierarchische Ordnung zu erhalten, gibt es zusätzlich noch die Subnetz-Adresse, die bestimmt, wie viele Geräte in einem Subnetz zusammengefasst werden. In einem kleinen Heimnetzwerk ist das meistens die 255. 255.255.0, was bedeutet, es bis zu 256 Geräte fassen kann. Die IP-Adresse selbst besteht aus vier Bytes. Dabei wird unterschieden zwischen dem globalen Adressraum, der von der Internet Assigned Numbers Authority vergeben wird, und lokalen Adressbereichen, die für lokale Netzwerke reserviert sind und dort von jedem Nutzer selbst gewählt werden können. Das bedeutet, dass zum Beispiel eine Adresse, die mit 192.168 beginnt (z.B. die 192.168.0.1), nicht vom gesamten Internet aus erreichbar ist. Dafür können mehrere Netzwerke jeweils die gleichen Adressen besitzen. Im Gegensatz dazu ist die Adresse 213.168.78.214 weltweit eindeutig und gehört dem deutschen Webserver des O'Reilly Verlags.

Die Verbindung von einem Netzwerk in ein anderes übernimmt ein sogenannter Router. Das sind für den Heimgebrauch meistens kleine Geräte, die heutzutage über mehrere Ethernet-Kabelanschlüsse und ein WLAN-Modul verfügen und die Übersetzung vom lokalen Netzwerk ins Internet übernehmen. Generell sind Router Geräte, die das Internet in Subnetze einteilen und so die Verteilung der Datenpakete übernehmen.

Die Übertragung von Daten übernehmen im Internet entweder das TCP- oder das UDP-Protokoll. Das UDP-Protokoll sendet munter Daten, ohne auf eine Antwort zu warten oder zu prüfen, ob sie auch wirklich ankommen. Man spricht von einer zustandslosen Datenübertragung. Bei UDP kann es vorkommen, dass Daten auf dem Weg verloren gehen, was aber für bestimmte Daten nicht weiter wichtig ist. Weil die Überprüfung nicht stattfindet, ist UDP schneller als TCP. Letzteres basiert auf dem sogenannten Drei-Wege-Handshake. Das bedeutet, dass jede Übertragung, also zum Beispiel die Auslieferung einer Website mit einem Anfrage-Paket (SYN) begonnen und von der Gegenseite mit einem »Acknowledged« (ACK) quittiert wird. Um diese Quittierung zu bestätigen, sendet sie zusätzlich noch ein weiteres SYN, auf das schließlich ein weiteres ACK von der Senderseite folgt. Meist geschieht diese Antwort in einem kombinierten ACK/SYN-Paket. Der Abbau einer Verbindung sieht ähnlich aus, statt SYN heißt das entsprechende Paket aber FIN.

Natürlich muss man im Internet kaum noch eine IP-Adresse eingeben, um sich mit einem bestimmten Rechner zu verbinden. Dafür

sorgt das Domain Name System (kurz DNS), das Domainnamen (wie etwa oreilly.de) in Adressen umwandelt. Da DNS-Abfragen keine sensiblen Daten darstellen, werden dafür UDP-Pakete verwendet. Für Physical-Computing-Projekte ist das DNS relativ unwichtig (man sollte nur wissen, dass es existiert), weswegen es hier nicht weiter behandelt wird. Umfassende Erläuterungen finden Sie unter anderem in der Wikipedia.

Nun können weitere Protokolle die eigentliche Anwendung repräsentieren. Das World Wide Web beispielsweise verwendet das Hypertext Transport Protocol (HTTP), um Daten zwischen Webserver und Browser auszutauschen, E-Mail läuft über die Protokolle POP3 (oder IMAP) und SMTP. Damit ein Gerät mit einer IP-Adresse eingehende Daten unterscheiden kann, werden sogenannte Ports verwendet. Die Portnummer wird als Teil des TCP/IP-Paketes versendet. HTTP läuft dabei standardmäßig auf Port 80 (bzw. 443 bei sicheren Verbindungen), das zum Versenden von Mails verwendete SMTP auf 25.

Die unterschiedlichen Protokolle werden ineinander gekapselt. Ein HTTP-Paket ist also Teil eines TCP-Paketes, das wiederum Teil eines IP-Paketes ist. Dieses befindet sich schließlich in einem Ethernet-Paket. Die Informationsdaten der einzelnen Bestandteile nennt man dabei Header, im Gegensatz zu den Nutzdaten, die das eigentliche Paket ausmachen.

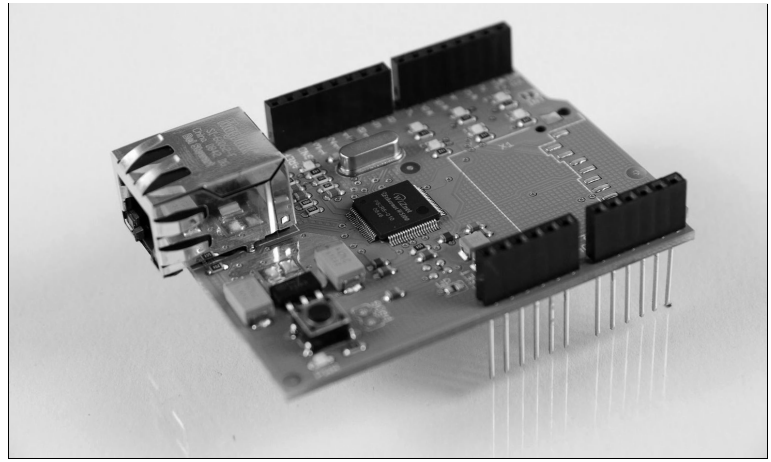
Manchmal kann es vorkommen, dass die Netzwerkverbindung Fehler aufweist. Es ist besonders schwer, die Ursache des Problems aufzuspüren, wenn ein Protokoll falsch implementiert ist. Ein Browser würde zum Beispiel bei einem fehlerhaften HTTP-Protokoll nicht ausreichend Informationen liefern. Abhilfe schaffen hier sogenannte Sniffer, mit denen man eine Ethernet-Verbindung überwachen und alle übertragenen Daten auswerten kann. Besonders empfohlen sei hier das Open-Source-Projekt Wireshark (*http://www.wireshark.org/*).

Hardware

Zunächst wird der Ethernet-Shield mit dem Arduino-Board verbunden, indem beide zusammengesteckt werden. Alle Pins sind auch über den Shield zugänglich. Selbst die LEDs und der Reset-Taste sind auf dem Ethernet-Shield noch einmal vorhanden, sodass man sie bequem bedienen kann. Die Pins 10 bis 13 werden allerdings schon für den Ethernet-Controller verwendet und sollten deshalb nicht für eigene Anwendungen eingesetzt werden, wäh-

rend das Netzwerk benutzt wird (das könnte zu merkwürdigen Effekten führen).

Abbildung 6-1 ▶
Ethernet-Shield



Der auf dem Board verbaute Chip Wiznet W1500 enthält einen kompletten Ethernet- und IP-Stack für TCP und UDP. Das bedeutet, dass er kein zusätzliches Betriebssystem benötigt, um mit dem Internet zu kommunizieren. Er kann somit direkt mit dem Arduino verwendet werden, bei der Programmierung muss man sich um nichts mehr kümmern. Den weiteren Aufwand übernimmt die Arduino-Ethernet-Bibliothek, die allerdings weder DNS noch DHCP implementiert hat, sodass man derzeit noch auf statische IPs zurückgreifen muss.

Software

In der Arduino-Entwicklungsumgebung und der Ethernet-Bibliothek ist bereits alles vorhanden, was man benötigt, um mit dem Ethernet-Shield zu arbeiten.

Das Board kann mit `Ethernet.begin(mac,ip[,gateway,subnet]` initialisiert werden. Die MAC-Adresse (oder Ethernet-ID) ist dabei die physische Adresse des Boards, die in diesem Fall vom Nutzer selbst festgelegt werden kann, da der Hersteller des Shield diese Adresse nicht vergeben hat. Die IP-Adresse muss je nach Netzwerk gewählt werden, in dem der Arduino eingesetzt wird. Befindet man sich in einem unbekannten Netzwerk, sollte man über den PC den IP-Adressbereich herausfinden und eine Adresse wählen, die mit Sicherheit in keinem Konflikt mit anderen steht. Die meisten Heim-

netzwerke haben einen Adressbereich von 192.168.xxx.xxx. Wählt man eine eigene IP-Adresse, darf diese sich nur im letzten Byte, also im letzten Abschnitt, von anderen unterscheiden; hat also der PC etwa die Adresse 192.168.1.2 und der Netzwerkrouter 192.168.1.1, so sind für den Arduino die Adressen 192.168.1.3 bis 255 möglich, je nachdem, welche anderen Netzwerkgeräte sich noch im selben Adressbereich befinden. Viel kann man allerdings nicht kaputt machen, wenn man die Adressen ausprobiert, im Falle eines Konfliktes funktioniert das Netz einzelner Geräte einfach nicht – dann muss man eben umstellen.

Verwendet man mehrere Ethernet-Shields, sollte man darauf achten, dass sowohl MAC- als auch IP-Adresse unterschiedlich sind; wenn man den einmal geschriebenen Code einfach nur auf andere Arduino-Boards lädt, können besonders schnell Fehler auftreten.

Will man das Board mit dem Internet kommunizieren lassen, benötigt man zusätzlich eine Gateway-Adresse. Diese ist in der Arduino-Bibliothek schon voreingestellt, auf den aktuell eingestellten Netzwerkadressbereich mit der Endung 1, also beispielsweise auf 192.168.1.1. Sollte das nicht funktionieren, muss eine andere Adresse eingestellt werden. Im Regelfall, also bei den heutzutage üblichen Heimnetzwerken, ist das die Adresse des Netzwerkrouters. Meist lässt sich auch diese über die Netzwerkinformationen des PCs herausfinden. Nur selten ist es hingegen nötig, eine zusätzliche Subnet-Adresse anzugeben. Diese ist im Normalfall auf 255.255.255.0 eingestellt – für alle Heimnetzwerke der Standard. Nur in großen, komplexen Netzwerken, kann dieses Subnet ein anderes sein. Sollte der Ethernet-Shield in einer solchen Umgebung, etwa dem Netzwerk einer Hochschule, eingesetzt werden, sollte man ein Gespräch mit dem zuständigen Systemadministrator führen.

Hello World – ein Mini-Webserver

Zunächst soll der Arduino als kleiner Webserver fungieren – ein guter Anfang für viele Projekte, die aus der Ferne gesteuert werden sollen. Der Webserver erwartet Anfragen auf dem Datenport 80 und sendet als Antwort zum Beispiel eine Webseite, die dann im Browser dargestellt werden kann. Über solch einen Webserver können Einstellungen vorgenommen werden, indem etwa bestimmte Links aufgerufen werden, die im Arduino wiederum Aktionen hervorrufen. Zudem kann dieser Webserver etwa Informationen über den Zustand des Arduino oder seine angeschlossenen Sensoren liefern.

Zunächst wird ein leeres Projekt erzeugt und mit `#include <Ethernet.h>` die Ethernet-Bibliothek eingebunden, damit alle benötigten Befehle zur Verfügung stehen.

Zudem werden zur Initialisierung die MAC- und die IP-Adresse des Boards (siehe oben) benötigt:

```
// MAC-Adresse des Arduino-Boards, 6 Bytes
byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
// IP-Adresse des Arduino-Boards, 4 Bytes
byte ip[] = {192, 168, 1, 10 };
```

Dabei sollten Sie darauf achten, dass beide Adressen passen. Die IP-Adresse muss ins lokale Netzwerk passen und darf dort keine Konflikte erzeugen, also nicht identisch mit schon bestehenden Geräten sein. Anschließend wird ein Serverobjekt erzeugt, das später die Verbindungen auf Port 80, dem Standard-Port für Webserver, annehmen kann. Das Objekt bietet eine Reihe von Funktionen, die jederzeit abgerufen werden können.

```
Server server = Server(80);// Port-Einstellung (Standard 80)
```

In der Setup-Funktion wird zuerst zuallererst mit `Ethernet.beginn(mac,ip);` die W1500-Hardware initialisiert. Danach wird der Server gestartet:

```
server.begin();     // Server wartet nun auf Clients
```

Nun muss in der `loop()`-Funktion auf einkommende Verbindungen geantwortet und HTML-Code zurückgesendet werden.

Dabei können nicht einfach rohe HTML-Daten über das Netz geschickt werden. Vielmehr muss der Webserver die Daten in HTTP verpacken. So verstehen Browser dieselbe Sprache wie Webserver und wissen, wie welche Daten zu interpretieren sind und was bei bestimmten Fehlermeldungen zu tun ist.

HTTP wurde 1989 am CERN von Tim Berners-Lee entwickelt, der damit den Grundstein für das World Wide Web legte. Zudem schuf er das Konzept von URLs, die eine eindeutige Lage jedes HTML-Dokuments beschreiben. Eine URL besteht aus dem Protokoll (http), dem Server, möglichen Unterverzeichnissen und dem Dokumentnamen, zum Beispiel http://www.oreilly.de/index.html. Zudem können einer URL noch weitere Informationen übergeben werden, wenn der Webserver in der Lage ist, diese zu interpretieren.

Wird eine URL im Browser eingegeben, sendet dieser eine Anfrage an den passenden Server. Diese Anfrage beinhaltet den Anfragetyp

(z.B. GET), den Servernamen, die Protokollversion und das angefragte Dokument:

```
GET /index.html HTTP/1.1
Host: www.oreilly.de
```

Erreicht diese Anfrage den Webserver, sendet dieser eine Antwort mit einem Code. War die Anfrage in Ordnung, ist dieser Code *200 OK*, aber es gibt eine ganze Reihe von anderen Möglichkeiten, etwa wenn kein Ergebnis gefunden wurde (*404 Not Found*).

Zudem beinhaltet die Antwort beispielsweise Informationen über den Webserver und das Betriebssystem, die Größe des zu übertragenden Dokumentes, dessen Sprache und Dokumenttyp und Anweisungen, was nach Erhalt dieser Antwort zu tun ist.

```
HTTP/1.1 200 OK
Server: Apache/1.3.29 (Unix) PHP/4.3.4
Content-Length: (Größe von index.html in Bytes)
Content-Language: de (nach ISO 639 und ISO 3166)
Content-Type: text/html
Connection: close
(Inhalt von index.html)
```

Nun kann der Browser diese Daten empfangen und anzeigen. Der kleine Arduino-Webserver kennt allerdings nur einen sehr kleinen Teil von HTTP: Er liefert immer eine »200 OK«-Antwort und kann auch keine URLs unterscheiden.

```
void loop() {
  Client client = server.available();   // prüfen, ob Client Seite
                                        // aufruft
  if (client) {                         // Seitenaufruf durch User
    // übertrage 200 OK-Code und identifiziere den Server als
    // arduino
    server.print("HTTP/1.0 200 OK\r\nServer: arduino\r\n");
    // übertrage den Dateityp
    server.print("Content-Type: text/html\r\n\r\n");
    //übertrage Daten
    server.print("<HTML><HEAD><TITLE>");
    server.print("Arduino Board");
    server.print("</TITLE>");
    server.print("</HEAD><BODY>");
    server.print("<b>Hello World!</b><br />");
    server.print("Arduino runs for ");
    server.print(millis());
    server.print(" ms.</BODY></HTML>");
    //Datenübertragung zu Ende
    delay(10);                // kurz warten, um Daten zu senden
    client.stop();            // Verbindung mit dem Client trennen
  }
}
```

Wenn man nun einen Webserver startet und die URL *http://192.168.1.10* öffnet, erscheint eine Webseite, die anzeigt, wie lange das Board schon läuft. Die Funktion `millis()` zeigt dabei die Anzahl Millisekunden seit dem Start des Programms.

Die Ausgabe könnte beispielsweise so aussehen:

```
Hello World!
Arduino runs for 14321 ms.
```

Dieser Webserver ist natürlich sehr einfach gestrickt. Er liefert auf jede Anfrage die gleiche Antwort und kann deshalb auch nicht dazu verwendet werden, Einstellungen im laufenden Arduino-Programm vorzunehmen. Das Projekt Webduino (*http://code.google.com/p/webduino/*) bietet hier mehr Möglichkeiten. Die Bibliothek ist in der Lage, komplexere Browseranfragen zu bearbeiten und Fehlercodes zurückzusenden. Sie kann also als Webserver eingesetzt werden, der alle Basisfunktionen versteht. Das sollte für fast alle Arduino-Projekte ausreichend sein.

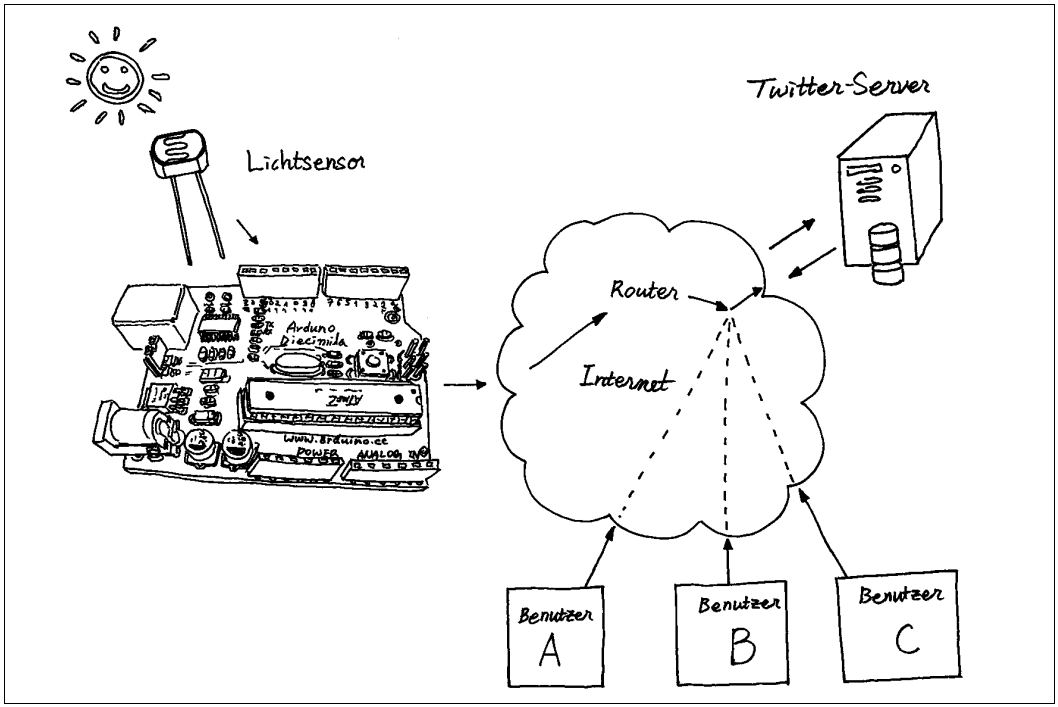
Sag´s der Welt mit Twitter

Twitter ist ein Onlinedienst, auf dem Benutzer kurze Nachrichten (»Tweets«) veröffentlichen können. Dabei ist die Nachrichtenlänge auf 140 Zeichen begrenzt. Jeder Nutzer erhält seine eigene kleine Webseite unter *http://www.twitter.com/nutzername*. Zudem kann jeder Nutzer angeben, welche anderen Nutzer er regelmäßig lesen, wem er also »folgen« möchte. So entstehen Kommunikationsnetzwerke, die ein wenig einem Chat ähneln, wobei sich die einzelnen persönlichen Chaträume der Nutzer teilweise überlappen. Dadurch, dass man mit *@username* auf die Tweets anderer antworten kann, entsteht eine neue Form der Kommunikation und Diskussion. Benutzte man jedoch lediglich die Webseite, würde alles recht schnell unübersichtlich. Abhilfe schafft die Twitter-API, eine Schnittstelle, über die es möglich ist, eigene Programme mit Twitter zu verbinden. Inzwischen dürfte die Mehrzahl aller aktiven Twitter-Nutzer ein solches Programm verwenden – zwei bekannte davon sind Tweetdeck oder Tweetie. Diese Programme informieren ihre Nutzer in regelmäßigen Abständen über neue Tweets.

Innerhalb von etwas mehr als zwei Jahren hat sich Twitter von einem Nischenspielzeug für eingeweihte Web-Enthusiasten zu einem Mainstream-Dienst entwickelt. Besonders bei Ereignissen von weltweiter Bedeutung, etwa bei Katastrophen, Terroranschlä-

gen oder auch Wahlen, wurden besondere Aktivitätssprünge verzeichnet. Der Grund dafür dürften vor allem die Aktualität und die Geschwindigkeit sein, mit denen sich Neuigkeiten so verbreiten. Als etwa am 15. Januar 2009 ein Flugzeug im New Yorker Hudson River notlanden musste, war das erste Bild bereits zwölf Minuten lang verbreitet worden, als die ersten Nachrichtendienste die Meldung aufschnappten. Bei den Protesten gegen das Wahlergebnis im Iran wurde Twitter ein nahezu unverzichtbares Kommunikationsmittel für die aufbegehrenden Studenten.

Diese Eigenschaften von Twitter lassen sich auch für Physical-Computing-Projekte nutzen. So kann ein mit dem Internet verbundener Arduino beispielsweise Daten von angeschlossenen Sensoren twittern, um seinen Besitzer (oder wen auch immer diese Informationen interessieren) zu benachrichtigen. Eines der bekanntesten Projekte dieser Art ist Botanicalls, das die Feuchtigkeit von Pflanzenerde misst und über Twitter meldet, wenn eine Pflanze gegossen werden muss.



▲ **Abbildung 6-2**
Twitter und Arduino

Dazu wird der Widerstand zwischen zwei in die Erde gesteckten Drähten gemessen, der analog zur Trockenheit zunimmt. So kann

der Arduino ab einem bestimmten Grenzwert einen Hilferuf über Twitter absenden. Fertige Bausätze gibt es bei Botanicalls *http://www.botanicalls.com/kits/* oder bei *http://www.bausteln.de*. Ein ähnliches Projekt erklärt auch Mats Vanselow unter *http://www.mats-vanselow.de/2009/02/08/arduino-lernt-twittern/*.

Der Fantasie sind hier keine Grenzen gesetzt: Fast alles, was man messen kann, kann auch in Tweets umgesetzt werden. Es gibt sogar ein Projekt, bei dem ein Bürostuhl twittert, wenn der »Besitzer« bestimmte Gase freisetzt. Wer daran Interesse hat, möge das Projekt unter *http://www.instructables.com/id/The-Twittering-Office-Chair/* nachlesen. Das Projekt in diesem Buch soll nicht ganz so exotisch sein, wir wollen nur die Helligkeit im Zimmer vermelden lassen. Geht das Licht aus, wird ein Tweet abgesendet, möglicherweise ein Hinweis für Freunde, die nicht wissen, ob sie noch anrufen können. Noch sinnvoller wird das Ganze, wenn man es in einem von mehreren Menschen genutzten Raum einsetzt. Wenn jemand erwägt, ob er noch ein gemeinsam genutztes Atelier, einen Vereinsraum oder Jugendzentrum besuchen soll, freut er sich vielleicht über die Information, ob in diesen Räumlichkeiten noch etwas los ist.

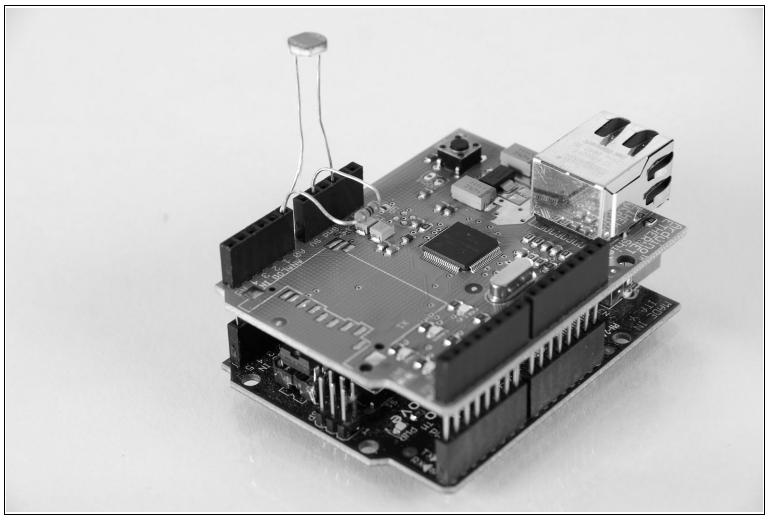
Hardware

An den Arduino werden ein LDR (Fotowiderstand) sowie ein weiterer Widerstand mit ca. 10.000–100.000 Ohm angeschlossen. Der Fotowiderstand ändert seinen Widerstand von ca. 1–100 Mio Ohm (MOhm) im Dunkeln zu 100–2.000 Ohm in der Sonne. Er wird an die die PINs 5V und A0 angeschlossen, der andere Widerstand an A0 und GND. So fungieren die Bauteile als Spannungsteiler, und die Spannung an A0 wird von der Helligkeit am LDR verändert (siehe Abbildung 6-3).

Software

Den Anfang machen wieder ein paar grundlegende Einstellungen, die gegebenenfalls angepasst werden müssten. Da der Ethernet-Shield keine DNS-Abfragen machen kann, muss die IP-Adresse von Twitter von Hand eingetragen werden. Sollte Twitter seine IPs ändern, muss diese Adresse angepasst werden.

Die Ethernet-Bibliothek wird beim Arduino-Programm mitgeliefert und kann dort einfach unter SKETCH → IMPORT LIBRARY eingebunden werden:



◀ **Abbildung 6-3**
Aufbau des Twitter-Lichtsensors

```
#include <Ethernet.h>
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
byte ip[] = { 192, 168, 0, 10 };          // unsere IP/anpassen an
                                          // das lokale Netzwerk
byte gateway[] = { 192, 168, 0, 1 };      // die IP des Routers
byte subnet[] = { 255, 255, 255, 0 };
byte server[] = { 128, 121, 146, 100 };   // IP von Twitter
(default: 128.121.146.100)
#define TWITTERURL "/statuses/update.xml" // URL zum Update-Skript
#define TWITTERUSERNAMEPW "geheim"  // Base64 kodiert USERNAME:
                                    // PASSWORT
Client client(server, 80);
void setup()
{
  Ethernet.begin(mac, ip, gateway, subnet); // Ethernet
                                            // initialisieren
  Serial.begin(9600);   // serielle Verbindung für Statusmeldungen
}
```

Zudem werden nun ein Twitter-Nutzername und -Passwort benötigt. Ein entsprechendes Nutzerkonto kann schnell auf *http://www.twitter.com* angelegt werden. Name und Passwort müssen durch einen Doppelpunkt getrennt, Base64-kodiert und in der Konstante *TWITTERUSERNAMEPW* gespeichert werden. Die Base64-Wandlung kann man zum Beispiel online unter *http://www.patshaping.de/projekte/kleinkram/base64.php* durchführen. Allerdings gibt man dort sein Twitter-Passwort in fremde Hände, und bei einer entsprechenden Suche findet man im Netz auch Downloadangebote mit Software, die die Umwandlung auf dem Rechner erledigen kann, ohne vertrauliche Daten durch die Gegend zu senden.

Nun werden zwei Routinen benötigt: Eine zum Senden der Nachricht und eine, die die Antwort vom Twitter-Server empfängt und an den seriellen Port leitet, damit sofort bemerkt werden kann, wenn etwas nicht funktioniert.

```
void sendTwitterUpdate(char* tweet)        // Nachricht an Twitter
                                           // übermitteln
{
  Serial.println("connecting...");
  if (client.connect())
  {
    Serial.println("connected");
    client.print("POST ");
    client.print(TWITTERURL);
    client.println(" HTTP/1.1");
    client.println("Host: twitter.com");
    client.print("Authorization: Basic ");
    client.println(TWITTERUSERNAMEPW);
    client.print("Content-Length: ");
    client.println(9+strlen(tweet));
    client.println("");
    client.println("status=");
    client.println(tweet);
    Serial.println("twitter message send");
  }
  else {
    Serial.println("connection failed");
  }
}
void fetchTwitterUpdate()     // Rückmeldung von Twitter auslesen
{
  if (client.available())
  {
    char c = client.read();
    Serial.print(c);
  }
  if (!client.connected())
  {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
  }
}
```

Es soll nur eine Nachricht getwittert werden, wenn der Lichtzustand sich gerade verändert hat, also wird der aktuelle Zustand in der Variable `licht` gespeichert, der letzte Zustand in `old_licht`. So kann auf Veränderungen reagiert werden.

```
int licht = 1;
int old_licht = 1;
int val;
```

```
void loop()
{
  delay(30000);
  val = analogRead(0);    // den Wert vom Sensor lesen
  if (val > 400) { licht = 1; } // diese Werte müssen empirisch
                                // ermittelt werden
  if (val < 200) { licht = 0; }
  if (old_licht != licht) {
    old_licht = licht;
    if (licht == 1) {
      Serial.println("Hell!");
      sendTwitterUpdate("Zimmer ist erleuchtet!");  // Nachricht
                                                    // versenden
    }
    else {
      Serial.println("Dunkel!");
      sendTwitterUpdate("alle Photonen aufgebraucht!");
                                          // Nachricht versenden
    }
  }
  fetchTwitterUpdate();      // eingehende Nachrichten durchleiten
}
```

Fang die Bytes – Datalogger

Viele Daten werden erst richtig interessant, wenn man sie über längere Zeit sammelt und dann gemeinsam auswertet. Dafür kann man das EEPROM des Arduino nutzen, auf dem in diesem Projekt die Daten gespeichert und anschließend auf einer Webseite zur Verfügung gestellt werden sollen. Ein EEPROM ist eine kleine Speicherzelle, die ein paar wenige Daten (beim Arduino sind es 512 Byte) aufnehmen kann. Die einzelnen Bits werden dabei in speziellen Transistoren gespeichert, sogenannten Floating Gates. Im Gegensatz zu den üblichen Transistoren können diese ihren Zustand, also leitend oder nicht leitend, auch speichern, wenn kein Strom anliegt. 512 Byte erscheinen auf den ersten Blick nicht viel, andererseits sollte es für die meisten Daten durchaus reichen, vor allem wenn man sie sorgfältig auswählt.

Für die Arduino-Programmiersprache sorgt die EEPROM-Bibliothek für das Beschreiben und Lesen des Speichers. Es gibt zwei Funktionen: `write()` und `read()`. Um diese zu verwenden, muss die Bibliothek mit `#include <EEPROM.h>` eingebunden werden. Da nur sehr wenige Daten zur Verfügung stehen, wird kein Dateisystem benötigt. Es können also keine Dateien oder Ordner abgelegt werden, und die Daten sind auch nicht speziell markiert. Beim Lesen

und Schreiben muss also bekannt sein, wohin und woher die Daten kommen.

Um das Ganze zu demonstrieren, soll der Wert eines Lichtsensors periodisch alle zehn Sekunden auf das EEPROM geschrieben werden. Nach einer Minute wird dann ebenfalls alle zehn Sekunden der eine Minute alte Wert über die serielle Konsole ausgegeben. Das Setup dürfte recht klar sein:

```
// verwende EEPROM-Bibliothek
#include <EEPROM.h>
// initialisiere Pin, Zähler und serielle Konsole
int lichtPin = 0;
int i = 0;
void setup() {
    Serial.begin(9600);
}
```

Nun kann der Wert ausgelesen und gespeichert werden. Der Input des analogen Pins ist allerdings 10 Bit lang. Zunächst sollen aber nur ein Byte lange Werte geschrieben werden. Entweder man verwendet also für vier Werte fünf Bytes, was später noch erklärt werden soll. Im einfacheren Falle genügt es, ungefähre Werte aus dem Sensor zu speichern. Teilt man den aktuellen Wert durch vier, erhält man die 8 Bit, die man bequem in einem einzelnen EEPROM-Byte ablegen kann.

```
void loop() {
    i++;
    int lichtWert = analogRead(lichtPin);
    lichtWert = lichtWert/4;
    EEPROM.write(i, lichtWert);
    if (i > 10) {
        int ausgabeWert = EEPROM.read(i-10);
        Serial.print(ausgabeWert, DEC);
    }
    delay(10000);
}
```

Nun fällt auf, dass der Speicher irgendwann voll sein wird. Es ist also sinnvoll, nach 5.120 Sekunden wieder bei 0 zu beginnen. Man fügt also am Anfang der `loop()`-Funktion Folgendes hinzu:

```
if (i == 512) {
    i = 0;
}
```

Gleichzeitig ist es dann sinnvoll, einen der hohen Werte auszugeben, wenn die Schleife gerade von vorn begonnen hat. Der Ausgabeteil wird also ersetzt durch eine Funktion, die vom aktuellen Zählerwert den passenden Ausgabewert ermittelt und schreibt:

```
function ausgabe(int i) {
    if (i < 10) {
        i = abs(i-512);
    }
    else {
        i = i-10;
    }
    int ausgabeWert = EEPROM.read(i);
    Serial.print(ausgabeWert, DEC);
}
```

Da man nie weiß, welche Werte sich beim Start noch im EEPROM befinden, sollte man zusätzlich zu Beginn alle Werte des Speichers auf null setzen. Dazu werden alle Speicherplätze durchlaufen und auf 0 gesetzt:

```
void deleteEEPROM() {
    // durchlaufe jedes Byte des EEPROM und setze es auf 0
    for (i = 0; i <512; i++) {
        EEPROM.write(i, 0);
    }
}
```

Angenommen, man benötigt doch die vollen 10 Bit des Sensorwertes, würde das bedeuten, dass ein Schreibvorgang immer volle 2 Bytes benötigt. 6 Bit pro Vorgang gingen verloren, bei gerade einmal 512 Byte eigentlich kaum akzeptabel. Betrachtet man die Werte bitweise, so erkennt man, dass vier Werte in fünf Bytes passen können. Dafür müssen sie lediglich umcodiert werden.

Die Funktionen `bitRead()` und `bitWrite()` können aus einer Variable einzelne Bitwerte auslesen und schreiben. Geht man also den Sensorwert bitweise durch und schreibt den Strom kontinuierlich in Bytes, erhält man folgenden Code:

```
int bitPos = 0;
int currentByte = 0;
int currentBytePos = 0;
void loop() {
   // lies Sensorwert ein
   int sensorVal = analogRead(sensorPin);
   // durchlaufe alle 10 Bits des Sensorwertes
   for (int i = 0; i < 10; i++) {
   // wenn das aktuelle Byte voll ist, also die Position im
   // Byte 7 übersteigt
   if (bitPos > 7)  {
     // springe auf das nächste Byte
     currentBytePos++;
     // schreibe das Byte an die aktuelle Byteposition
     EEPROM.write(currentBytePos, currentByte);
     // setze das Byte auf 0
     sensorByte = 0;
```

```
        // setze die Bitposition auf 0
        bitPos = 0;
      }
      //
      // lies das aktuelle Bit des Sensorwertes ein
      // und schreibe es an die entsprechende Position im
      // aktuellen Byte
      bitWrite(currentByte, bitPos, bitRead(sensorVal, i));
      // zähle Bitposition nach oben
      bitPos++;
    }
```

Um die Daten wieder aus dem EEPROM zu lesen, muss die Funktion umgedreht werden. Da die Größe der Sensordaten (10 Bit) bekannt ist, weiß man auch, welche Bits sich wo befinden, wenn man den Anfangswert kennt. Beginnt man bei null, liest man also zuerst die ersten beiden Bytes aus und liest und schreibt das erste Byte vollständig sowie die niedrigsten 2 Bits des zweiten Bytes in die Ausgabevariable. Dazu zählt man die aktuelle Bitposition sowie die Position des Bytes. Die Bitposition wird in jedem Schritt nach oben gezählt. Nach 8 Bits wird diese wieder auf `0` gesetzt und die Byteposition um eins erhöht. Sind 10 Bytes durchlaufen, wird die Variable ausgelesen und ausgegeben.

Zunächst wird eine Reihe weiterer Zählvariablen benötigt:

```
int readBytePos = 0;
int readBitPos = 0;
int outByte = 0;
```

Anschließend kann der folgende Programmteil gesetzt werden.

```
// lies aktuelles Byte aus dem Speicher
currentByte = EEPROM.read(readBytePos);
  // durchlaufe die zehn zu schreibenden Bits
  for (int i = 0; i < 10; i++) {
    // schreib aktuelles Bit aus dem gelesenen Byte in ein
    // Ausgabebyte
    bitWrite(outByte, i, bitRead(currentByte, readBitPos));
    // zähle Bitposition nach oben
    readBitPos++;
    // nach 8 Bits fange wieder von 0 an
    if (readBitPos > 7)  {
        readBitPos = 0;
      readBytePos++;
      // lies neues aktuelles Byte aus dem EEPROM
      currentByte = EEPROM.read(readBytePos);
    }
   }
```

Nicht selten kommt es vor, dass mehrere Sensordaten eines Programms geschrieben werden sollen. Schreibt man nun in einem

fort, ist es schwer, die Daten auseinanderzuhalten. Wählt man den gleichen Zähler, überschreiben sich die Daten gegenseitig. Hier empfiehlt es sich, das EEPROM aufzuteilen und zum Beispiel eine Gruppe in die ersten 256 Bytes und die zweite dahinterzusetzen. Dabei sollte der Speicher je nach Bedürfnis aufgeteilt werden. Benötigt der eine Wert zum Beispiel nur 4 und der andere 8 Bit, sollte man die Trennung beim 128. Byte vornehmen, um gleich viele Werte zu speichern.

Ein anderes Problem ist natürlich der Speicherplatz. Neben dem eben erklärten Ausnutzen aller verfügbaren Bits kann man sich natürlich auch überlegen, in welchen Abständen die Daten gespeichert werden sollen. Bei einem Temperatursensor, der nur die Raumtemperatur aufzeichnen soll, genügt womöglich auch ein Speichervorgang alle paar Minuten, bei anderen Werten sollte es vielleicht jede Sekunde passieren. Wenn nur die letzten Werte benötigt werden, kann man natürlich auch von vorn beginnen, wenn der Speicher voll ist, und die alten Daten überschreiben. Dazu sollte man sich natürlich in einer Variable merken, wo der aktuellste bzw. jüngste Wert liegen.

Sie sollten dabei beachten, dass das EEPROM nur eine begrenzte Anzahl von Schreibzyklen verträgt. Die im Datenblatt angegebenen *>100.000* Schreibzyklen können schnell erreicht werden, wenn man zum Beispiel an einer Stelle im EEPROM abspeichert, welches der nächste freie Platz zum Schreiben ist, da dieser Wert ja nach jedem Schreibvorgang aktualisiert werden muss. Speicherte man jede Sekunde einen Messwert, würde man diese Grenze schon nach etwas mehr als einem Tag erreichen! Es kann natürlich in Wirklichkeit viel, viel länger dauern, bis etwas passiert, da die Hersteller hier sehr konservative Werte angeben, allerdings ist es noch viel besser, gleich beim Algorithmus anzusetzen, um ein Vielfaches der Lebensdauer zu erhalten. Dabei sollte darauf geachtet werden, dass alle Speicherzellen ungefähr gleich oft beschrieben werden.

Gerade bei Messwerten kann oft darauf verzichtet werden, eine Null zu speichern; der Unterschied zur Eins bei einem 10-Bit-Messwert beträgt nur 0,01%, bei 8 Bit nur 0,4%, damit ist der Wert kaum relevant und die Messung kann vernachlässigt werden. Der aktuelle Messwert wird dann an die erste Stelle im EEPROM gespeichert, die gleich null ist, die nächste Stelle wird dann auf null gesetzt, sodass der nächste Zugriff zuverlässig seine Stelle findet. So kann auch gleich ein sogenannter Ringpuffer implementiert werden: Wenn das nächste Byte außerhalb der EEPROM-Größe liegt,

wird stattdessen das erste gelöscht. So hat man immer die zum Beispiel 512 letzten Messwerte zur Verfügung.

Durch diesen Trick kann jetzt die Lebensdauer des EEPROM bei sekündlichem Speichern auf fast ein Jahr verlängert werden. Reicht auch dieser Wert nicht aus, kann es eine Lösung sein, die Batteriespannung regelmäßig zu messen und die Werte erst dann aus dem RAM ins EEPROM zu verschieben, wenn die Spannung unter einen bestimmten Wert fällt. Bei geschickter Programmierung reicht dafür auch ein etwas größerer Kondensator an der Betriebsspannung, sodass gerade noch genug Zeit bleibt, alle Werte zu sichern. Dafür sollte diese Routine dann aber über einen Interrupt gestartet werde, damit der Zeitpunkt nicht verpasst wird.

Die einfachste Möglichkeit ist, einfach seltener zu messen, oder eventuell sekündliche Messwerte nur im RAM zu speichern und daraus berechnete Durchschnittswerte der letzten Minute ins EEPROM zu übertragen.

KAPITEL 7

Sensoren

In diesem Kapitel:

- Sensoren
- Aktoren
- Elektronischer Würfel

Dieses Kapitel beschäftigt sich näher mit verschiedenen Sensoren und Aktoren. Dabei werden zunächst die häufigsten verwendeten Bauteile erläutert und ihre möglichen Einsatzgebiete erklärt. Damit wird eine Grundlage für die darauf folgenden Workshops geschaffen, in denen zum Teil eigene Bauteile gebastelt, zum anderen die hier beschriebenen Sensoren und Aktoren eingesetzt werden.

Sensoren

Die wohl wichtigste Rolle beim Physical Computing spielen neben immer günstigeren Mikrocontrollern vor allem die verschiedenen Sensoren. Ob sie nun selbst angelötet werden, mit einem Steckbrett an den Arduino angeschlossen, oder ihren Weg über einen der zahllosen Shields nehmen, die verschiedenen Zwecke dienen: Ohne Sensoren ist ein Arduino-Board kaum etwas wert. Hier werden die wichtigsten Sensorenarten und ihre Ausprägungen vorgestellt, wobei dieses Kapitel längst nicht alle einzelnen Bauteile umfassen kann. Weitere Informationen finden Sie unter anderem im Tutorial-Bereich auf der Arduino-Website (*http://www.arduino.cc/playground/Learning/Tutorials)* oder in der ausführlichen Zusammenfassung bei Freeduino (*http://www.freeduino.org*).

Schalter und Taster

Schalter und Taster wurden schon detailliert in Kapitel 3 vorgestellt. Von den hier vorgestellten Sensoren sind sie diejenigen, deren Funktionsweise am einfachsten ist: Ist der Schalter oder Taster »eingeschaltet«, fließt Strom, ansonsten ist der Stromkreis unterbrochen.

Schalter und Taster werden in der Regel mit dem einen Pin an einen digitalen Arduino-Pin und mit dem anderen an GND angeschlossen. Bei mehrstufigen Schaltern werden natürlich auch mehrstufige Pins verwendet. Fließt der Strom, liegt am digitalen Eingang eine 1 an, ist der Stromkreis unterbrochen, ist das Ergebnis eine 0. Bei Schaltern und Tastern empfiehlt es sich, einen Pull-up-Widerstand zu verwenden (siehe Kapitel 3).

Die Anwendungen von Schaltern und Tastern sind dabei sehr vielfältig. So könnte eine LED-Leuchte beispielsweise mit einem Schalter durch verschiedene Modi wechseln oder auch nur ein- und ausgeschaltet werden. Ein Taster wiederum eignet sich vor allem zur Eingabe, etwa bei kleinen Spielen, die damit gesteuert werden. Auch für ein Wecker-Projekt könnte man einen Taster verwenden, um Uhr- und Weckzeit einzustellen.

Dreh- und Schieberegler

Im folgenden Abschnitt geht es um Bauteile, die in der Lage sind, für den Benutzer quasi stufenlose Eingaben an den Arduino zu senden. Dabei handelt es sich um Dreh- und Schieberegler, die je nach Einstellung eine unterschiedliche Spannung an den Ausgang lassen. Die analogen Input-Pins des Arduino sind in der Lage, bis zu 10 Bit, also Eingaben von 0 bis 1.023 zu verarbeiten. Je nach Verwendung empfiehlt es sich jedoch, diese Werte zu teilen (durch 4 oder sogar 8). Damit können die Messungen ein wenig fehlertoleranter werden, da die Eingaben aus analogen Bauteilen niemals konstant bei einem Wert liegen.

Potentiometer

Potentiometer sind auch als Drehknöpfe oder Drehregler bekannt und die wohl meisten verbreiteten Bauteile aus der Gruppe der stufenlosen Regler. Sie gehören zur Klasse der resistiven Sensoren, also derjenigen, bei denen die Spannung durch den Widerstand verändert wird.

Potentiometer gehören, ähnlich wie die vorhin beschrieben Schalter und Taster, zu den am häufigsten eingesetzten Steuerelementen. Drehknöpfe gibt es in allen möglichen Formen: als Lautstärkeregler einer Stereoanlage oder um die Temperatur eines Herdes zu steuern, aber auch in zahlreichen Physical-Computing-Projekten. Sie können die Frequenz oder Helligkeit von LEDs steuern oder als Eingabemittel für Processing verwendet werden, um dort zum Beispiel Formen und Farben einer Animation einzustellen und live zu verändern.

Joystick

Eine besondere Variante des Potentiometers ist der Joystick. Ein Joystick ist in der Lage, sich in zwei Achsen und somit alle Richtungen zu bewegen. Dabei nutzt er pro Achse ein Potentiometer, sodass er nur an zwei analoge Pins des Arduino (sowie an 5V und GND) angeschlossen werden muss. Gerade um bewegliche Objekte mit dem Arduino zu steuern, sind Joysticks sehr sinnvoll. So können zum Beispiel zwei Joysticks eine Fernbedienung für einen Roboter oder eine Flugdrohne bilden. Oftmals werden dabei auch nicht nur einfache Bauteile aus dem Elektronikfachhandel verwendet, sondern Computerjoysticks, deren Anschlüsse an den Arduino gelötet werden, oder der Controller der Nintendo Wii, die sogenannte Wiimote bzw. der Nunchuck-Teil. Der Vorteil dabei ist, dass gleichzeitig noch eine Menge anderer Sensoren dabei sind, die beim Nunchuck direkt und bei der Wiimote über Bluetooth angesprochen werden können.

Ribbon Controller und Touchpads

Unter einem Touchpad versteht man eine Fläche, die die Position eines oder mehrerer Finger feststellen und die Daten darüber weitervermitteln kann. Pads mit nur eine Achse nennt man auch Ribbon Controller. Diese werden vor allem in der Musik eingesetzt und sind unter anderem auf Synthesizern zu finden.

Dabei gibt es zwei grundsätzliche Ausprägungen dieser Pads: resistiv und kapazitiv. Erstere verwenden eine leitende Oberfläche, deren Widerstand durch die Position des Fingers verändert wird. Ähnlich wie bei den vorher beschriebenen Potentiometern liegt somit auch mehr oder weniger Spannung am Ausgang an, sodass ein Wert ermittelt werden kann.

Kapazitive Touchpads bestehen üblicherweise aus einer vertikalen und einer horizontalen Anordnung von Elektroden, die ein Gitter bilden. Die Oberfläche des Touchpads ist dabei eine isolierende Schutzschicht, die verhindert, dass die Finger direkt mit den Elektroden in Berührung kommen. Da der menschliche Körper und damit auch die Finger selbst wie eine Elektrode funktionieren, verändert sich die Kapazität der einzelnen Elektroden, wenn ein Finger das Pad berührt. Ein Schaltkreis darunter kann diese Änderungen messen und als veränderte Spannung weitergeben.

Touchpads finden außerhalb von Laptops vor allem in der Musik Anwendung. So kann ein Pad zum Beispiel für Midi-Eingaben ver-

wendet werden; mithilfe eines Arduino könnte man so teure Musikhardware selbst bauen. Aber auch für kleine Spiele auf dem Arduino – selbst mit einem Display – oder unter Processing gibt es einige Möglichkeiten, ein Touchpad einzusetzen.

Resistiver Touchscreen

Ein resistiver Touchscreen besteht aus einer leitfähig beschichteten Glasscheibe, über der sich eine leitfähige Kunststofffolie befindet. Auf der Folie sind viele ganz kleine Abstandhalter aufgedruckt (normalerweise ein Epoxidsiebdruck), sodass im Normalfall die Folie keinen Kontakt zur Glasscheibe hat. Drückt man mit einem Finger oder Stift auf die Folie, entsteht eine leitfähige Verbindung, und der Ort dieser Verbindung kann über die Widerstandswerte gemessen werden. Dazu hat eine leitfähige Schicht zwei horizontale und die andere zwei vertikale Anschlüsse. Zuerst wird durch Anlegen einer Spannung an die horizontale Schicht auf dieser ein Spannungsverlauf (wie bei einem Potentiometer) aufgebaut, über die beiden vertikalen Anschlüsse kann dann die Position in horizontaler Richtung gemessen werden. Danach wird das Verfahren umgedreht und die vertikale Richtung vermessen.

Dafür ist es ganz praktisch, dass die analogen Eingänge des Arduino-Boards auch als Ausgänge verwendet werden können. An vier dieser Ein-/Ausgänge kann direkt ein 4-Wire-Touchscreen angeschlossen werden.

Größere und langlebigere Touchscreens werden oft in 5-Wire-Technik ausgeführt, sie haben also fünf Anschlüsse. Die leitfähige Kunststofffolie wird hier nur als Messeingang verwendet, und die Potentiale werden mit den vier Elektroden auf der Glasplatte realisiert. Dadurch kann die Beschichtung auf der Folie viel dicker und niederohmiger ausgeführt werden, und Widerstandsschwankungen in der Folie verändern nicht den Messwert.

Allen resistiven Touchscreens ist gemeinsam, dass sie keine Multitouch-Funktion unterstützen, da man nur einen Berührungspunkt messen kann.

Lichtsensoren

Nach den mehr oder weniger mechanischen oder elektromechanischen Sensoren folgen nun jene, die auf optische Impulse wie etwa Helligkeit reagieren können.

Fotowiderstand

Ein Fotowiderstand oder LDR (light dependent resistor) besteht aus einer Halbleiterschicht, deren Widerstand mit zunehmender Lichtstärke abnimmt. Die Schicht besteht dabei meist aus Cadmiumsulfid (CdS), dessen Leitfähigkeit durch den inneren fotoelektrischen Effekt zunimmt. Schließt man diesen Sensor nun an den Arduino an, kann man Werte von 0 (völlig dunkel) bis 1.023 (sehr hell) messen. In diesem Buch werden zweimal LDR eingesetzt: um über Twitter zu melden, ob das Licht im Raum an ist, und für ein kleines Spiel im Zusammenhang mit Processing. Möglich wäre auch eine selbstregulierende Lampe, die die aktuelle Helligkeit misst und versucht, sie auf einem Level zu halten. Eine solche Lampe würde bei Sonnenuntergang langsam angehen. Um festzustellen, ob sie auch wieder ausgeht, müsste langsam heruntergedimmt und ab einem gewissen Schwellenwert erneut geprüft werden. Reicht die Helligkeit aus, bleibt die Lampe auf diesem Niveau, ist es zu dunkel, wird sie wieder heller gestellt. Andererseits könnte der Sensor natürlich auch außen angebracht und so von der Lampe getrennt werden. Weitere Möglichkeiten sind die Verwendung von MAX/MSP zusammen mit einem Arduino und einem Fotowiderstand, um elektronische Musikinstrumente zu steuern, oder der Einsatz eines LDR, um einen kleinen Roboter zu lenken.

Light Intensity IC

Ein nicht ganz günstiger Schaltkreis zur Lichtintensitätsmessung ist der TSL230R-LF von Taos. Dabei liegt der große Vorteil in der linearen und vermessenen Ausgangsfunktion. Der Schaltkreis gibt abhängig von der gemessenen Helligkeit unterschiedlich lange Impulse aus. Laut Datenblatt ist die Länge der Impulse definiert als 0,77 KHz/W. Damit ist also eine echte Helligkeitsmessung möglich. Eine ausführliche Referenz findet sich im Blog von Roaming Drone unter *http://roamingdrone.wordpress.com/2008/11/13/arduino-and-the-taos-tsl230r-light-sensor-getting-started/*. Diese Adresse findet sich, wie alle Adressen in diesem Buch und weitere Aktualisierungen im Blog zum Buch, unter *http://arduinobuch.wordpress.com*.

Fototransistor

Ein Fototransistor ist, wie der Name schon sagt, eine spezielle Form des Transistors. Bei entsprechendem Lichteinfall lässt er einen Stromfluss zu. Dabei ist er viel lichtempfindlicher als eine Diode, da er gleichzeitig als Verstärker wirkt. Das wohl bekannteste

Arduino-Projekt, das einen Fototransistor einsetzt, ist eine Schaltung, um Gewitterblitze zu fotografieren. Dabei ist der Transistor schnell genug, um im Falle eines hellen Blitzes in der Natur den Auslöser zu betätigen. Eine passende Schaltung ist sehr einfach, wenn die Kamera eine elektrische Auslöseverbindung besitzt. Diese wird an einen digitalen Ausgangspin angeschlossen, der Transistor an einen analogen Eingang. Misst man nun konstant die Lichtwerte, kann man bei einem ausreichend starken Anstieg ein digitales Signal an den Auslöser geben, sodass die Kamera ein hoffentlich gutes Foto eines Blitzes machen kann.

Eine Dokumentation dieses Projekts findet sich im Netz unter *http://www.glacialwanderer.com/hobbyrobotics/?p=16*.

Lichtschranke

Mithilfe eines handelsüblichen Lasers und einer Fotodiode lässt sich eine einfache Lichtschranke basteln. Dabei wird der Laser so ausgerichtet, dass sein Licht direkt auf die Fotodiode fällt. Wird das Licht des Lasers unterbrochen, sodass es nicht mehr auf die Diode fällt, fällt das Signal stark ab, und der Arduino ist in der Lage, dies zu verarbeiten. In der Fotografie gibt es weitaus mehr Gelegenheiten, Fotos in möglichst kurzer Zeit auszulösen, als nur bei Blitzen. Eine Lichtschranke ist etwa in der Lage, Tropfen in einer Flüssigkeit zu erkennen und eine entsprechende Fotografie auszulösen. Natürlich lassen sich so auch einfache Alarmsysteme basteln, die vermelden, wenn eine bestimmte Schranke durchkreuzt wurde, oder man misst die Geschwindigkeit eines Objektes, indem man den Zeitabstand zwischen der Durchquerung zweier Schranken berechnet.

Beschleunigungssensoren

Beschleunigungssensoren messen, wie ihr Name schon verrät, die aktuelle Beschleunigung in eine oder mehrere Richtungen. Im Folgenden werden Drei-Achsen-Beschleunigungssensoren vorgestellt, die direkt an den Arduino angeschlossen werden können, sowie weitere Bauteile und Geräte, die einen solchen Sensor beinhalten.

Einfache Beschleunigungssensoren

Moderne Beschleunigungssensoren wie der ADXL320 und der ADXL330 von SparkFun basieren auf sogenannten mikroelektrisch-mechanischen Systemen oder MEMS und werden aus Sili-

zium hergestellt. Diese Sensoren bestehen aus wenige Mikrometer breiten Siliziumstegen und einer ebenfalls aus Silizium bestehenden Masse. Bei einer Beschleunigung lenkt diese Masse aus und führt zu einer Kapazitätsänderung zwischen ihr und einer Bezugselektrode. Dabei besteht der gesamte Messbereich nur aus etwa einem pF (*picoFarad*, Billionstel Farad). Diese Änderungen können nicht auf das Arduino-Board übermittelt werden, sondern werden direkt auf dem Halbleiterbaustein verarbeitet. Der ADXL wird nun an drei analoge Pins und an 5V angeschlossen, damit die Beschleunigungswerte gelesen werden können. Möchte man nun wissen, wie viel G (Einheit für Beschleunigung) ein bestimmter Wert bedeutet, muss man diesen anhand einer Tabelle umrechnen. Diese erhält man aus dem Datenblatt.

Beschleunigungssensoren können auch verwendet werden, um Neigung festzustellen. Das ist beispielsweise bei einer Flugdrohne nützlich, die nicht nur von Wegpunkt zu Wegpunkt fliegen, sondern sich dabei auch stabil in der Luft halten soll. Natürlich eignen sich solche Sensoren auch für Spielereien wie das bekannte Murmelspiel, das z.B. in Processing implementiert werden kann. Der Sensor steuert das Brett, auf dem die Murmel liegt, die nun durch ein Labyrinth finden muss, ohne in eines der Löcher zu fallen.

Wiimote

Als Nintendo Ende 2006 die Wii-Konsole auf den Markt brachte, sorgte das nicht nur in Spielerkreisen für Aufruhr. Grund dafür war die Technik, die im Wiimote-Controller und dem angeschlossenen Nunchuck verbaut ist und einzeln im Elektronikfachhandel viel teurer ist als in der weißen Hülle.

Freundlicherweise hat Nintendo sich aber nicht darauf beschränkt, diese Daten verschlüsselt an die Wii zu senden. Vielmehr wird Bluetooth eingesetzt, um alle nötigen Informationen zu funken; der Nunchuck benutzt ein serielles Protokoll. Dafür gibt es inzwischen Bibliotheken, sodass vor allem Beschleunigungssensor und Joystick des Nunchuck, aber auch der daran angebrachte Knopf ausgelesen werden können. Nunchucks sind im Handel für 19,95 Euro erhältlich. In Kapitel 10 wird ein Nunchuck an den Arduino angeschlossen und als MIDI-Kontroller verwendet. Ein ausführliches Tutorial zur Verwendung mit dem Arduino findet sich unter *http://www.windmeadow.com/node/42*.

Die wohl nächstliegende Verwendung des Nunchuck liegt in der Kontrolle von Motoren. Kleine Servos können Roboter antreiben,

aber auch andere Geräte wie etwa eine Webcam. Dank Knopf, Joystick und Neigungssensor gibt es aber noch unzählige andere Einsatzgebiete. So findet sich im Internet beispielsweise ein Bastler, der seine Espressomaschine mit einem Nunchuck ausgestattet hat, um die verschiedenen Parameter wie Druck und Hitze einzustellen und schließlich mit der Produktion des perfekten Espresso zu beginnen.

Gyroskop

Ein Gyroskop ist ein schnell rotierender Kreisel, der sich in einem beweglichen Lager dreht, z.B. in einem Käfig. Aufgrund der Drehimpulserhaltung behält der Kreisel auch seine Lage im Raum, wenn der Käfig geneigt wird. Dadurch können Drehungen und Neigungen im Raum gemessen werden. Das Gyroskop kann also als Kompass dienen, um den richtigen Weg nach Norden anzuzeigen – selbst an den Polen, wo ja magnetische Kompasse versagen. Zudem liefert es echte Neigungsmesswerte, selbst wenn es sich bewegt. Beschleunigungssensoren können einen Winkel zur Schwerkraft nämlich nur dann korrekt messen, wenn sie sich im Ruhezustand befinden.

Elektronische Gyroskope werden heute ähnlich wie Beschleunigungssensoren als MEMS aufgebaut. Allerdings ist darin kein sich drehender Kreisel mehr vorhanden, sondern eine vibrierende Masse, deren Auslenkung bei Drehungen gemessen werden kann. Beispiele dafür sind der ADXRS300 und der ADXRS610 von Analog Devices. Um eine stabile Lage in der Luft zu halten, benötigen beispielsweise Quadrokopter drei Gyroskope. Sie werden allerdings zusätzlich von einem Beschleunigungssensor unterstützt, damit sich Messfehler nicht aufaddieren und den Hubschrauber zum Umkippen bringen.

Abstandssensoren

Es gibt viele Methoden, mit denen sich der Abstand von einem Objekt zum Sensor messen lässt. Das reicht von recht einfachen Mitteln wie einem kapazitiven Sensor bis hin zu Radar und Ultraschall. Im Folgenden werden die häufigsten Methoden beschrieben.

Ein kapazitiver Sensor zum Selbstbauen

Ein kontaktloser kapazitiver Sensor basiert, wie sein Name schon beschreibt, auf der elektrischen Eigenschaft der Kapazität. Sie

beschreibt die Fähigkeit eines Objektes, elektrische Ladungen zu speichern. Der Sensor zeigt an, wie zwei leitfähige Schichten (zum Beispiel Metallplatten), die durch eine nicht leitfähige getrennt sind (z. B. Luft), auf eine angelegte Spannungsdifferenz reagieren. Das wohl bekannteste Einsatzgebiet dieser Eigenschaft ist der Kondensator. Die Kapazität hängt von der Fläche und der Entfernung der leitfähigen Objekte ab.

Ein solcher Sensor lässt sich einfach selbst basteln, indem auf der einen Seite eine Antenne (ein Draht oder ein Stück Aluminiumfolie) verwendet wird, die mit dem Arduino verbunden ist. Die andere Seite kann zum Beispiel die Hand oder der Körper des Benutzers sein, da auch diese leitfähig sind.

Kapazitive Sensoren verwenden eine Wechselspannung, die immer wieder an einem Ende des Sensors angebracht wird. Diese Spannung erzeugt wiederum einen elektrischen Strom, der von der Kapazität abhängt, also vom Abstand zwischen Draht und Hand. Je größer und näher die Handfläche, desto mehr Strom wird erzeugt, natürlich auch in Abhängigkeit von der Form der Antenne. So lassen sich mit einem großen Stück Folie empfindlichere Bewegungen messen als mit einem kurzen Stück Draht. Wenn aber z. B. nur ein kleiner Bereich empfindlich auf Näherung reagieren soll, ist der Draht praktischer. Hier gilt wie so oft der Grundsatz »Probieren, Probieren, Probieren!«

Ein kapazitiver Sensor wird jeweils an zwei Arduino-Pins angeschlossen Der eine ist der Sendepin, der die Wechselspannung des Sensors erzeugt. Er wird regelmäßig von 0 auf 1 gesetzt und erzeugt so eine ständig wechselnde Spannung. Über einen relativ hochohmigen Widerstand wird er an den Empfangspin angeschlossen. In diesem Setup verwenden wir einen Widerstand von 5 Megaohm, es sind aber Werte von ca. 1–10 Megaohm möglich. Je nach Widerstand ist der Sensor mehr oder weniger empfindlich, es ist hier also nicht notwendig, den genauen Wert von 5 Megaohm einzusetzen.

An den Empfangspin wird die Antenne angeschlossen. Dazu lässt sich entweder ein langes Stück blanker Draht verlöten oder ein Stück Alufolie, das um etwas Draht gewickelt ist. Möglich sind auch leitende Gegenstände (wie ein Kochtopf), die mit dem Pin verbunden werden. Wichtig ist, dass der Antennengegenstand elektrisch leitend ist. Mit der Entfernung der Antenne zu anderen leitenden Gegenständen, die mit der Erde verbunden sind (z. B. der menschliche Körper des Benutzers oder eine Heizung) lässt sich die

Kapazität am Empfangspin verändern. Zusammen mit dem Widerstand, der Sende- und Empfangspin verbindet, wird ein sogenanntes RC-Netzwerk gebildet. RC steht für *resistor* und *capacitator*, also Widerstand und Kondensator. Diese beeinflussen, wie schnell eine Spannungsänderung am Ausgang des Netzwerks widergespiegelt wird. Wenn also der Arduino den Sendepin von 0 auf 1 schaltet, bestimmt die Kapazität an der Antenne, wie schnell diese Veränderung am Empfangspin ankommt. Misst man nun die Zeit, lässt sich die Entfernung des menschlichen Körpers bestimmen. Verwendet man mehrere Sensoren, kann man sogar die Position berechnen.

Natürlich kann man solche Sensoren auch für wenig Geld kaufen. In jedem Falle kann die Spannung analog ausgelesen und verarbeitet werden.

Hall-Sensor

Ein Hall-Sensor nutzt den nach dem Physiker Edwin Hall benannten Effekt, der das Auftreten von elektrischer Spannung in einem stromdurchflossenen Leiter beschreibt, wenn dieser sich in einem stationären Magnetfeld befindet. Die Spannung fällt dabei senkrecht sowohl zur Stromfluss- als auch zur Magnetfeldrichtung am Leiter ab und wird Hall-Spannung genannt. Dieser Sensor ist also kein Abstandssensor im eigentlichen Sinne, kann aber zum Beispiel als Schalter verwendet werden, ohne dass ein mechanischer Kontakt entstehen muss. Er kann aber auch verwendet werden, um Magnetfelder zu erkennen und ihre Stärke zu messen. Wenn Strom- und Magnetfeldstärke bekannt sind, funktioniert er auch als Metalldetektor. Ein recht einfaches Projekt (*http://mekonik.wordpress.com/2009/03/02/my-first-arduino-project/*), das einen Hall-Sensor verwendet, ist ein Elektromagnet, der durch die Regelung mittels Arduino ein magnetisches Objekt in der Luft schweben lassen kann. Dabei misst ein Hall-Sensor die Stärke des Feldes und der Arduino reguliert es, sodass das Objekt nicht zu nahe kommt oder aus dem Feld herausfällt. Andererseits kann der Hall-Sensor zum Beispiel auch benutzt werden, um die Geschwindigkeit eines Ventilators zu messen.

Reed-Relais

Das Reed-Relais schaltet einen Stromkreis, je nachdem, ob sich ein magnetisches Feld in der Nähe befindet oder nicht. Es besteht aus zwei Kontakten, die in einem Glaskörper so eingeschmolzen sind, dass sie sich durch ein externes Magnetfeld anziehen können. Je

nach Beschaffenheit schließen oder öffnen sie sich, wenn eine magnetische Spule oder ein Dauermagnet in die Nähe kommt. Fällt das Magnetfeld ab oder unterschreitet es eine bestimmte Schwelle, wird dieser Effekt beendet. Die wohl bekannteste Anwendung ist ein Tachometer am Fahrrad, bei dem der Magnet am Rad und das Reed-Relais am Rahmen angebracht sind. Weil der Reifenumfang bekannt ist, kann man dann die Geschwindigkeit und die zurückgelegte Strecke des Rades berechnen, indem man die Zeit zwischen zwei Kontakten misst. Natürlich kann ein solcher Schalter auch an einer Tür angebracht werden, um festzustellen, ob sie geöffnet oder geschlossen ist. Das Ergebnis lässt sich dann digital mit dem Arduino auslesen.

Sonar

Das Sonar kennt man aus U-Booten (bzw. den entsprechenden Filmen), wo es zur Ortung von fremden Gegenständen und Booten in der Umgebung verwendet wird. Ein Sonarsensor kann aber auch dazu verwendet werden, den Abstand von einem Objekt zum Sensor zu messen. Das geschieht, indem ein Ultraschallsignal ausgesendet und die Zeit gemessen wird, bis das Echo zurückkommt. Ein Beispiel für einen solchen Sensor ist der Ping))) von Parallax Inc. (*http://www.parallax.com/Store/Sensors/ObjectDetection/tabid/176/ProductID/92/List/1/Default.aspx?SortField=ProductName,ProductName*). Er ist in der Lage, Objekte in einer Entfernung von 2 cm bis zu 3 m zu orten. Die Messung übernimmt allerdings das daran angeschlossene Arduino-Board. Es setzt zunächst den mit dem Sensor verbundenen Pin als Output fest und sendet dann ein Signal und anschließend den Pin auf Input. Nun kann es die Zeit messen, bis ein Signal auf dem Pin anliegt, und diese Zeit in Entfernung umwandeln, da Schall sich in der Luft nahezu konstant bewegt. Laut Datenblatt sind das 1.130 Fuß (344,424 Meter) pro Sekunde, also 34,442 Zentimeter pro Millisekunde.

Infrarotsensor

Natürlich lässt sich auch mit Licht die Position relativ zu einem Objekt bestimmen. Dafür werden Infrarotwellen gemessen, die vom Sensor ausgesandt werden. Das angepeilte Objekt reflektiert die Lichtwellen, die in einem bestimmten Winkel wieder auf dem Sensor auftreffen. Diese Messung samt Abstandsberechnung nennt man Triangulation. Für einfache Projekte empfiehlt sich zum Beispiel der IR Range Finder GP2D12 von Sharp, der für ca. 14 Euro im Elektronikfachhandel erhältlich ist. Er kann Objekte in einer Entfernung zwischen 10 und 80 Zentimetern erkennen.

Temperatursensor

Temperatur lässt sich auf viele verschiedene Arten durch die Anwendung unterschiedlicher physikalischer Prinzipien messen. Am häufigsten sind Widerstandssensoren. Kaltleiter erhöhen ihren Widerstand, wenn die Temperatur zunimmt, während Warmleiter ihn senken. Auf der anderen Seite gibt es auch Halbleitersensoren, die – je nach Bauweise – zur Temperatur proportionalen Strom oder Spannung liefern.

Feuchtigkeitssensor

Grundsätzlich gibt es zwei verschiedene Arten von Sensoren, um Feuchtigkeit zu messen: Hygrometer, die die Luftfeuchtigkeit anzeigen können, und Messgeräte für die Messung der Feuchtigkeit in verschiedenen Stoffen (z.B. Tensiometer für die Bodenfeuchtigkeit).

Die interessanten Hygrometer sind vor allem jene, die die absolute Luftfeuchtigkeit messen können. Um die relative Luftfeuchtigkeit und den Taupunkt zu messen, gibt es ebenfalls Verfahren, die hier aber keine Rolle spielen.

Um die absolute Luftfeuchtigkeit zu messen, wird ein wasseranziehendes Material verwendet. Seine Eigenschaften verändern sich bei zunehmender Luftfeuchtigkeit. Bei elektrischen Hygrometern ist das ein Polymer, dessen Widerstand oder Kapazität sich verändern kann.

Kompass

Kompasse bestehen nicht notwendigerweise aus einer runden Messingdose mit Glasdeckel und einer magnetischen Nadel. Mit dem HMC6352 von Honeywell ist beispielsweise ein magnetischer Sensor verfügbar, der auch an den Arduino angeschlossen werden kann. Um die beiden Achsen auszulesen und somit die horizontale Lage des Sensors zu bestimmen, empfiehlt es sich, die Wire-Bibliothek zu verwenden, die auf der Website des Arduino-Projektes (*http://www.arduino.cc/en/Reference/Wire*) dokumentiert ist. Eine andere Bibliothek samt Beispiel finden Sie unter *http://rubenlaguna.com/wp/2009/03/19/arduino-library-for-hmc6352/*.

Ein Kompass kann als günstige Methode der Lagebestimmung in einem Roboter eingesetzt werden, wenn man diesem einen be-

stimmten Pfad mitgeben will. Man kann damit natürlich auch ein multifunktionales Gerät basteln, das etwa mit einem Display und weiteren Sensoren ausgestattet ist und als günstige Hilfe bei Aktivitäten in der Natur fungiert, etwa beim Bergwandern.

Allerdings sollte man Eisen in der Nähe des Sensors vermeiden, da dieser sonst unweigerlich falsche Werte anzeigt. In gewissen Grenzen kann dieser Effekt aber weggerechnet werden.

Mikrofone

Mikrofone wandeln Schallwellen in elektrische Wellen um, sodass man diese messen kann. Das Prinzip besteht immer aus einer Membran, die sich durch die Druckwellen in der Luft verformt. Diese Verformung kann über verschiedene Mechanismen gemessen werden.

Dynamisches Mikrofon

Ein dynamisches Mikrofon ist die Umkehrung eines Lautsprechers: Die von den Schallwellen bewegte Membran verschiebt die daran angebrachte Spule innerhalb eines Magneten, wodurch in der Spule Ströme induziert werden. Tatsächlich kann man einen Lautsprecher auch als Mikrofon verwenden. Vorteilhaft ist dabei, dass dynamische Mikrofone nicht so leicht übersteuert werden wie Elektret-Kondensatormikrofone (kurz: Elektretmikrofone), was besonders für Lichtorgel-Anwendungen im Bereich der Lautsprecher wichtig ist.

Der Ausgang eines dynamischen Mikrofons ist zu schwach, um direkt vom Arduino gemessen zu werden; er muss mit mehreren Transistoren oder einem Operationsverstärker verstärkt werden.

Elektret- und Kondensatormikrofone

Beim »normalen« Kondensatormikrofon bilden zwei elektrisch leitfähige Platten einen Kondensator. Der Abstand der Platten, und damit die Kapazität, wird vom Schalldruck verändert. Um diese Veränderung messen zu können, legt man eine hohe Spannung über einen Widerstand an den Platten an, und die veränderte Kapazität beeinflusst dann die Spannung am Kondensator. Ein Elektretmikrofon benötigt diese Spannung nicht, da das Elektret zwischen den beiden Platten »vorgespannt« ist: Es wird bei der Fertigung unter einer hohen Spannung abgekühlt, sodass die Elektronen in diesem Zustand verbleiben.

Piezoelektrisches Mikrofon

Der Piezoeffekt beschreibt die Erzeugung einer elektrischen Polarisation bei Festkörpern, wenn sie verformt werden. Wird also auf einen piezoelektrischen Sensor mechanisch eingewirkt, verändert sich die Spannung auf der Oberfläche des Sensormaterials. Umgekehrt kann ein piezoelektrischer Aktor sich verformen, wenn Spannung angelegt wird.

Das piezoelektrische Mikrofon reagiert auf Druckveränderungen in der Luft. Eine Membran folgt den Schwankungen des Schalls. Sie ist mit einem piezoelektrischen Element gekoppelt, das durch den Druck minimal verformt wird und so elektrische Spannungsschwankungen auslöst. Als Mikrofon ist ein Piezo nicht die beste Wahl, da die Tonqualität höhenlastig ist. Als Sensor kann es zum Beispiel in Verbindung mit einem Tiefpass-Filter eingesetzt werden, um Bassimpulse an den Arduino zu melden. Damit können z.B. Animationen in Processing gesteuert oder LEDs gepulst werden, um eine entsprechende Atmosphäre zur Musik zu schaffen. Oder man nutzt es, um Geräusche zu melden, etwa für eine Alarmanlage. Einer der Autoren dieses Buches nutzt eine Schaltung mit Piezomikrofon, um eine Tonaufnahme zu starten, sobald er wieder einmal im Schlaf zu reden anfängt.

Biometriesensor

Im eigentlichen Sinn gibt es keine biometrischen Sensoren, allerdings können Temperatur-, Spannungsdifferenz-, Druck- und Kraftsensoren dazu verwendet werden, biometrische Zustände zu messen.

Fingerabdrucksensor

Fingerabdrücke werden heutzutage nicht mehr nur in Science-Fiction-Filmen verwendet, um beispielsweise Schlüssel zu ersetzen. Viele Notebooks und auch elektronische Türsysteme besitzen solche Scanner. Diese sind nicht hundertprozentig sicher, sollten also nicht in hoch sensiblen Einrichtungen verwendet werden. Für den Gebrauch in der heimischen Umgebung, in der kaum damit gerechnet werden dürfte, dass das System für kriminelle Machenschaften missbraucht wird, sind sie allerdings geeignet, wenn auch derzeit noch etwas teuer. Für 70 bis 100 Euro kann man einen passenden Sensor für den Arduino erstehen.

EEG

Die Elektroenzephalografie, bei der Gehirnaktivität gemessen wird, war lange Zeit teuren Geräten in Krankenhäusern und Forschungseinrichtungen vorbehalten. So wurden wissenschaftliche Erkenntnisse gesammelt und klinische Diagnosen durchgeführt. Mittlerweile werden solche EEG-Geräte aber in einfachen Versionen selbst gebaut, zum Beispiel im Projekt OpenEEG (*http://openeeg.sourceforge.net/*). Die bekannteste Anwendung ist das BrainPong, eine Version des berühmten Tennis-Computerspiels, dessen Paddel durch die Gedanken des Spielers gesteuert werden. Nach kurzem Training ist der Computer in der Lage, verschiedene aktive Gedanken (wie etwa »oben« oder »unten«) zu unterscheiden. Dabei werden Elektroden auf dem Kopf angebracht, die die vom Gehirn ausgesandten elektromagnetischen Wellen messen. Weil diese Wellen je nach Aktivität in unterschiedlichen Bereichen emittiert werden, kann man sich mit mehreren Elektroden ein räumliches Bild davon verschaffen.

Gasdrucksensor

Der Druck in einer Leitung oder einem Gefäß kann über die Verformung einer Membran gemessen werden. Der dabei gemessene Wert entspricht dann immer der Differenz der Drücke an den zwei Seiten der Membran (*Differenzdrucksensor*). Die physikalische Einheit für Druck ist dabei Pascal (Pa). Fast alle Differenzdrucksensoren funktionieren entweder resistiv (ändern also ihren Widerstand) oder kapazitiv (ändern ihre Kapazität). Solche Sensoren werden zum Beispiel im in Kapitel 1 beschriebenen Bierbrauprojekt eingesetzt, wo der Druck im Brautank reguliert werden muss. Sie eignen sich aber auch, um große Mengen an Flüssigkeit zu regulieren oder beispielsweise Wasserschläuche zu steuern. So kann ein Gasdrucksensor am Ende eines Schlauches, der in eine Wasserzisterne getaucht wird, denn aktuellen Wasserstand messen.

Kraftsensor

Piezoelektrische Sensoren nutzen den gleichnamigen Effekt, um aus der Verbiegung eines Kristalls eine Spannung zu erzeugen. In einem Feuerzeug schlägt ein Federmechanismus auf einen Piezokristall. Die dabei erzeugte Spannung ist groß genug, um das Gas zu entzünden. EnOcean stellt mit diesem Effekt drahtlose Lichtschalter

her, die ihre Energie für das Funksignal nur aus dem Betätigen des Schalters gewinnen. Da die vom Piezokristall abgegebene Spannung aber proportional zur Änderung der Kraft ist, die auf den Kristall einwirkt, können nur Veränderungen gemessen werden. Man kann damit also keine Waage bauen, aber zum Beispiel Drumpads für ein elektronisches Schlagzeug.

Resistive Sensoren basieren auf Dehnungsmessstreifen (DMS), die bei Verformung ihren elektrischen Widerstand verändern. Wird der DMS in die Länge gezogen, verringert sich die Leiterbahnbreite, und der Widerstand steigt. Dieses Prinzip würde auch mit einem normalen Draht funktionieren, was aber aus zwei entscheidenden Gründen nicht praktikabel ist: Die Widerstandsänderung ist sehr klein, und normaler Kupferdraht würde sich nicht mehr zusammenziehen, nachdem die aufgebrachte Kraft wieder verschwunden ist. Fast alle elektrischen Waagen funktionieren mit DMS-Sensoren. Auch im Brückenbau, in Kraftwerken usw. werden sie eingesetzt, um frühzeitig Verformungen zu erkennen.

Kapazitive Drucksensoren enthalten einen Kondensator, der durch die aufgebrachte Kraft verformt wird, wodurch sich seine Kapazität ändert. Diese Änderung ist dann ein Maß für den Druck.

Der einfachste Anwendungsfall für einen Drucksensor ist natürlich eine Art Tastatur, die mehr als nur einen An/Aus-Wert übergeben muss. Einfache Sensoren können zum Beispiel in Kleidung eingenäht werden, um festzustellen, wenn der Träger umarmt wird oder hinfällt. Sicherlich kennen Sie den »Hau den Lukas« von Jahrmarktbesuchen. Unter einer Gummimatte und einem Holzbrett könnte der Drucksensor so die Hammerschläge messen, die der Arduino dann auf einer langen Linie von LEDs visualisiert. Ein anderes Beispiel sind Klopfsensoren; ein Drucksensor auf einer Tür kann diese nach dem richtigen Klopfsignal öffnen.

Biegungssensor

Biegungssensoren verhalten sich ähnlich wie piezoelektrische Kraftmesser. Sie verändern also ihren Widerstand, wenn sie mechanisch verändert werden – in diesem Fall gebogen. Diese Sensoren können beispielsweise in Kleidung eingearbeitet werden und bieten so die Möglichkeit eines einfachen Motion-Capturing. Auch wenn die Präzision wahrscheinlich nicht ausreicht, um damit filmreife 3-D-Modelle zu steuern, gibt es diverse Projekte damit: Die Designer Mika Satomi und Hannah Perner-Wilson nutzen selbstgebaute

Neopren-Sensoren (erklärt unter *http://www.instructables.com/id/Neoprene-Bend-Sensor-IMPROVED/*) für eine interaktive Tanzperformance. An verschiedenen Stellen in die Kleidung einer Tänzerin eingearbeitete Biegungssensoren steuern über ein Funksignal Instrumente, die die Darbietung mit Klängen untermalen. Dieses Projekt findet sich auf der Webseite der Designer unter *http://www.instructables.com/id/Puppeteer_Motion_Capture_Costume/*.

Ein anderes Projekt nutzt einen Biegungssensor, um eine heliotrope Pflanze zu steuern. Diese Pflanzen neigen sich stets nach der Sonne, sodass ihre Blätter sich im Tagesverlauf von Osten nach Westen drehen und am Morgen wieder Richtung Osten stehen. Das genannte Projekt misst die Neigung der Pflanze und dreht sie um 180 Grad, sobald sie einen bestimmten Grenzwert überschritten hat. So richtet sich der Stamm wieder auf und beginnt erneut seine Dehnungsübungen. Dieses Projekt ist auf der Webseite des Make Magazine unter *http://blog.makezine.com/archive/2006/12/bend_sensor_hel.html* beschrieben.

Aktoren

Aktoren sind allgemein Elemente, die eine physikalische Eingangsgröße in eine andersartige Ausgangsgröße umwandeln. In der Welt des Physical Computing ist diese Eingangsgröße natürlich Elektrizität. Die möglichen Ausgangsgrößen sind so unterschiedlich, wie die Physik vielschichtig ist, und reichen von Licht und Ton bis hin zu Wärme und Bewegung. Die gängigsten Aktoren werden hier nun kurz vorgestellt.

LEDs

Glühlampen und Leuchtstoffröhren sind zwar sehr hell, können aber nur schwer geregelt werden. Sie reagieren nur langsam und sind zudem groß und damit nur begrenzt einsetzbar. LEDs sind, wie in Kapitel 3 beschrieben wurde, Dioden, die binnen Mikrosekunden einen Lichtimpuls abgeben können. Die fortschreitende technische Entwicklung macht es inzwischen möglich, auch Lichtdioden mit sehr hohen Helligkeitswerten herzustellen, sodass die LED langsam auch die privaten Haushalte erobert. Weitere Vorteile neben den schon erwähnten sind, dass LEDs im Durchschnitt viel länger halten und über eine deutlich höhere Lichtausbeute pro Watt verfügen. Waren LEDs lange Zeit nur in weißer und roter Farbe erhältlich (andere Farben wurden über einen entsprechenden

Plastiküberzug erzeugt), gibt es mittlerweile auch grüne und blaue Leuchtdioden und Kombinationen aller drei Grundfarben (Rot, Grün und Blau), die gemeinsam auf einen Chip montiert sind. Kapitel 3 erklärt ausführlich, wie LEDs mit dem Arduino angesteuert und durch Pulsung in ihrer sichtbaren Helligkeit verändert werden können. Am Ende des Kapitels gibt es zudem eine Liste interessanter LED-Projekte.

Motoren

Kleine Motoren sind der Antrieb für alle Projekte im Bereich der Robotik. Sie können aber auch benutzt werden, um beispielsweise Lampen auszurichten oder Sensoren in eine gewünschte Position zu bringen. Im normalen Gebrauch unterscheidet man zwischen drei Arten: Gleichstrom-, Servo- und Schrittmotoren.

Gleichstrommotoren

Die günstigsten und einfachsten Motoren basieren auf einer oder mehreren magnetischen Spulen, die um eine Welle gefasst werden. Liegt Spannung an diesen Motoren an, bringt das Magnetfeld die Welle zum Rotieren. Ein normaler Gleichstrommotor kann dabei nur schwer gesteuert werden. Er bietet keinerlei Rückkopplung, sodass dem Benutzer weder die Position noch die Geschwindigkeit genau bekannt sind. Für präzise Operationen, wie etwa die Steuerung eines Fluggerätes oder die Drehung eines Objektes um wenige Grad, eignet er sich also nicht; kann man aber darauf verzichten, ist der Gleichstrommotor aufgrund seines Preises natürlich zu empfehlen.

Servomotoren

Benötigt man ein Feedback vom Motor, um seine Geschwindigkeit genau regeln zu können, empfiehlt sich ein Servomotor. Diese Motoren sind durch die Rückkopplung außerordentlich präzise, allerdings auch entsprechend teuer. Ein Servomotor besteht aus einem Gleich- oder Wechselstrommotor, enthält zusätzlich einen Rückkopplungsmechanismus, meist auf Basis einer Lichtschranke oder eines Potentiometers. So ist er in der Lage, genaue Geschwindigkeiten und sogar Positionen einzustellen. Die kleine Variante der Servomotoren sind die Modellbauservos. Diese können sich nur um ca. 180° drehen, haben aber ein einfaches digitales Interface, können also ohne weitere Bauteile mit dem Arduino verwendet

werden. Für Roboteranwendungen werden diese gerne so modifiziert, dass die Rückkopplung entfällt. Dafür erhält man dann für wenig Geld einen Motor mit integrierter Geschwindigkeitsregelung. Diese sind im Modellbaufachhandel für bezahlbares Geld erhältlich.

◀ **Abbildung 7-1**
Servomotor

Schrittmotoren

Ein Schrittmotor basiert auf einem drehbaren Motorteil (Rotor) mit einer Welle, das sich in einem magnetischen Feld befindet. Dieses Feld ist in Schritte unterteilt, sodass der Rotor sich um einen oder mehrere dieser Schritte drehen kann. Dadurch kann man im Prinzip Geschwindigkeit und auch Position bestimmen. Allerdings gibt es keinerlei Rückmeldung über mögliche Schrittverluste. Während ein Servomotor seine Position wieder einnimmt, wenn er beispielsweise durch einen Finger angehalten wurde, ist der Schrittmotor dazu nicht in der Lage. Zudem kann es bei großen Leistungen zu Problemen kommen. Dafür ist er deutlich preisgünstiger.

Der Arduino selbst ist nicht in der Lage, Motoren zu steuern, da er ja nur über digitale Ausgänge verfügt und ein Motor nicht einfach angeschaltet werden kann. Hier hilft ein Motor-Driver wie der L293 oder der SN754410NE. Unter *http://www.ladyada.net/make/mshield/* oder auch *http://www.nkcelectronics.com/freeduino-arduino-motor-control-shield-kit.html* findet man Shields für den Arduino, über die Richtung und Geschwindigkeit des Motors gesteuert werden können. Der Shield von Ladyada.net verwendet dabei 8-Bit-Werte für die Geschwindigkeit und wird über eine der

Bibliotheken angesteuert, die auf der Website heruntergeladen werden können. Der Driver von nkelectronics.com verwendet Pulsweitenmodulation und wird nur über ein entsprechendes `analogWrite()` angesprochen. Welchen man nun kauft, oder ob man sich dank verfügbarer Board-Layouts selbst einen Shield baut, hängt vom Verwendungszweck und dem Preis ab, den man zu zahlen bereit ist.

Relais

Oft möchte man einen Arduino verwenden, um Stromkreise zu schalten, die sich weit über der Betriebsspannung des Boards von 5 Volt bewegen. Das ist natürlich zunächst ein Problem, weil starke Ströme dem Board erheblichen Schaden zufügen können. Um beispielsweise Glühbirnen anzuschalten, werden die vollen 220 Volt aus der Steckdose benötigt. Abhilfe können hier Relais schaffen. Das sind Schalter, die durch einen elektronischen Impuls einen weiteren Stromkreis schalten können. Ein mechanisches Relais arbeitet meist nach dem Prinzip des Elektromagneten. Liegt Strom in einer Spule an, zieht diese einen mechanischen Anker an, der als Schalter im zweiten Stromkreis fungiert. Diese Relais eignen sich vor allem, um zum Beispiel Geräte im Haushalt anzuschalten. Wie wäre es mit einem Kaffeewecker, der durch den Duft frischen Espressos auf sich aufmerksam macht, oder mit einem Lichtschalter, der unter einer Fußmatte neben dem Bett angebracht ist und den Raum beleuchten lässt, sobald man aufsteht?

Neben diesen mechanischen Relais gibt es auch eine Reihe anderer; dazu zählen auch Transistoren bzw. Halbleiterrelais, auch wenn diese streng genommen keine Relais sind. Diese können zum Beispiel recht einfach verwendet werden, um leistungsfähige LEDs zu schalten (siehe Kapitel 1 und 3), da Transistoren im Gegensatz zur Mechanik auch schnell genug sind, um die nötige Pulsung zu leisten.

Ein Relais wird – ähnlich wie eine leistungsfähige LED – immer über einen Transistor an den Arduino angeschlossen. Dabei sollte man die Diode nicht vergessen, sonst kann die Induktivität des Relais den Transistor beschädigen.

Solenoide

Ein Solenoid ist eine Zylinderspule, die magnetische Kräfte entwickelt, wenn Spannung angelegt wird. Diese kann etwa um einen

Metallstift gewickelt werden, oder aber zum Beispiel als Ventil funktionieren.

Die Metallstiftvariante des Solenoids besteht aus einem beweglichen Stift, der genau in der Mitte der Spule platziert wird. Wird auf dieser Spule Spannung angelegt, treibt die magnetische Kraft diesen Metallstift nach außen. Dieser Impuls ist stark genug, um den Solenoid als Aktor zu verwenden. So gibt es Projekte, die daraus ein Trommelkonzert machen. Entweder hämmert der Solenoid direkt auf verschiedene Gegenstände, um so unterschiedliche Töne zu erzeugen, oder er treibt wiederum einen Holzstock an, der beispielsweise auf eine Trommel schlägt. Oder man setzt ihn ein, um selbst als musikalisch unbegabter Mensch aus einem Glockenspiel schöne Melodien ertönen zu lassen. Dieser Solenoid benötigt viel Strom; ein Anschluss über einen Transistor wie in Kapitel 3 beschrieben ist also dringend erforderlich.

Das Solenoid-Ventil eignet sich, um genaue Mengen von Flüssigkeiten oder Gasen abzugeben. Dazu muss man nur wissen, mit welcher Geschwindigkeit sich welche Menge des gewünschten Stoffes durch einen vom Solenoid abgeschlossenen Schlauch oder ein Rohr bewegt. Schließt man dieses Ventil nun über eine entsprechende Schaltung an den Arduino an, ergeben sich viele Möglichkeiten, nicht nur für den wissenschaftlichen Bereich. So könnte man etwa selbst als Hobbyfilmer eine günstige und leicht zu steuernde Regenmaschine bauen. Oder man verwendet zusätzlich einen Feuchtigkeitssensor, um Pflanzen vollautomatisch zu bewässern. Natürlich eignet sich ein solches Ventil auch für einen Roboter, der in der Lage ist, perfekte Cocktails zu mixen, weil er immer die korrekte Menge einer Zutat abgibt, sofern es sich dabei nicht um Limetten oder gestoßenes Eis handelt. Angesteuert werden sie wie Relais mit einem Transistor.

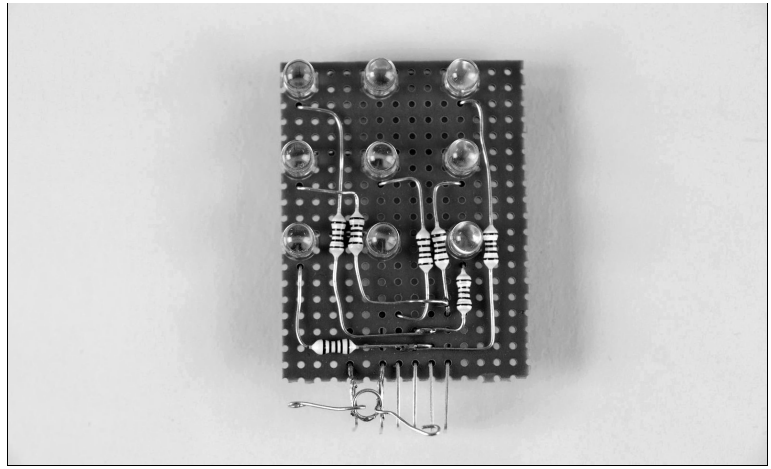
Elektronischer Würfel

Nun soll ein elektronischer Würfel gebaut werden. Dieser besteht aus einem elektronischen Display, das die gewürfelte Zahl darstellt, sowie einem Taster, der den Würfel anstößt. Anschließend wird mit einem Piezo-Lautsprecher noch ein entsprechender Ton ausgegeben.

An Material werden dafür sieben LEDs, sieben Widerstände 470R, eine Lochrasterplatine und ein piezoelektrischer Sensor benötigt.

Auf den folgenden Bildern sind neun LEDs zu sehen, allerdings sind die zwei LEDs oben und unten in der Mitte nicht weiter angeschlossen. Sie können also weggelassen werden.

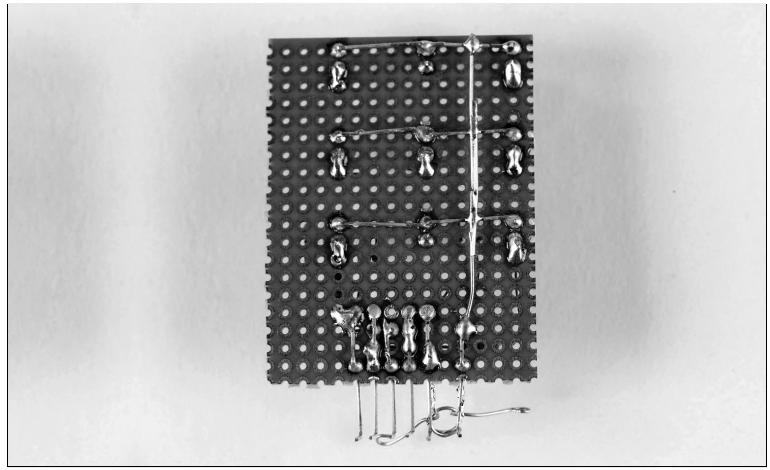
Abbildung 7-2 ▶
Würfelaufbau, Lochrasterplatine oben



Setup

Die mittlere LED wird mit einem 470R-Widerstand mit Pin 11 verbunden. Die beiden LEDs oben links und unten rechts mit Pin 10, die LEDs unten links und rechts mit Pin 9 und die verbleibenden zwei LEDs mit Pin 8.

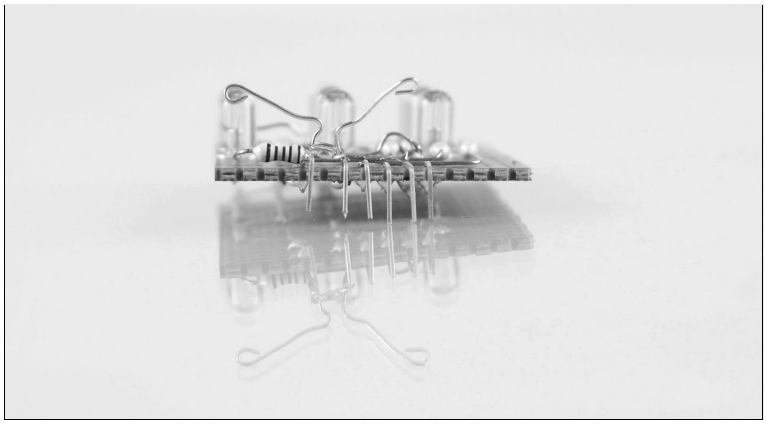
Abbildung 7-3 ▶
Würfelaufbau, Lochrasterplatine unten



Es wäre einfach, einen Taster zu verwenden, um den Würfel seine Zahl wählen zu lassen. Viel näherliegend für den Benutzer ist es

aber, zu schütteln, um zum Ergebnis zu gelangen. Dazu wird ein Erschütterungssensor aus den von den Widerständen und LEDs übrig gebliebenen Anschlussdrähten gebaut, indem ein Ring gebogen und die Enden verlötet werden. Dieser Ring kommt über zwei nach außen umgebogene Drähte (damit der Ring nicht abfallen kann) an GND und Pin 12.

Wird die Schaltung bewegt oder geschüttelt, bewegt sich der kleine Drahtring und öffnet und schließt dabei den Kontakt zwischen diesen beiden PINs; die Eingabe kann verwendet werden, um den Würfel zu steuern.



◀ **Abbildung 7-4**
Rüttelsensor

Software

Zunächst werden die sieben LEDs in einem leeren Worksheet angebunden (wie auch in Kapitel 3 erklärt), dann wird Pin 12 mit dem Erschütterungssensor als Eingang definiert und mit einem HIGH-Signal für den internen Pull-up-Widerstand belegt.

Nun kann durch einzelne HIGH- und LOW-Signale herausgefunden werden, wie sich mit dieser Schaltung alle benötigten Augenbilder eines Würfels abbilden lassen. Das Ergebnis wird in eine eigene Funktion geschrieben.

Dabei kann das Konstrukt `switch() {.. case …}` verwendet werden. Es erlaubt wie `if...else` bedingte Sprünge, kann aber abhängig von einem Wert aus mehreren Verzweigungen wählen:

```
void display(int num)
{
  switch(num)
  {
```

```
  case 0:
    digitalWrite(led1Pin, LOW);
    digitalWrite(led2Pin, LOW);
    digitalWrite(led3Pin, LOW);
    digitalWrite(led4Pin, LOW);
    break;
  case 1:
    digitalWrite(led1Pin, HIGH);
    digitalWrite(led2Pin, LOW);
    digitalWrite(led3Pin, LOW);
    digitalWrite(led4Pin, LOW);
    break;
und so weiter...
```

Dieser Code ist allerdings wenig elegant, zieht er sich doch in die Länge und wiederholt sich dabei in großen Teilen. Mit ein wenig Nachdenken kommt man allerdings auf einige Gesetzmäßigkeiten:

LED1 ist immer an, wenn `num` nicht durch 2 teilbar ist (1, 3, 5).

LED2 ist immer an, wenn `num` > 1 ist (2, 3, 4, 5, 6).

LED3 ist immer an, wenn `num` > 3 ist (4, 5, 6).

LED4 ist nur bei `num` = 6 an.

Also kann das Ganze durch Folgendes ersetzt werden.

```
// zeigt eine Zahl von 0 bis 6 auf den Würfel-LEDs an
void display(int num)
{
  if ((num % 2) == 0) {
    digitalWrite(led1Pin, LOW);
  } else {
    digitalWrite(led1Pin, HIGH);
  }
  if (num > 1) {
    digitalWrite(led2Pin, HIGH);
  } else {
    digitalWrite(led2Pin, LOW);
  }
  if (num > 3) {
    digitalWrite(led3Pin, HIGH);
  } else {
    digitalWrite(led3Pin, LOW);
  }
  if (num == 6) {
    digitalWrite(led4Pin, HIGH);
  } else {
    digitalWrite(led4Pin, LOW);
  }
}
```

Der Erschütterungssensor verhält sich wie ein Taster. Je nachdem, wie der Drahtring liegen bleibt, ist unbekannt, ob der Stromkreis

im Ruhezustand offen oder geschlossen ist. Deshalb wird überprüft, ob der Zustand sich geändert hat, und nur ein Signal registriert, wenn das der Fall ist.

```
int getTaster(void)
{
  taster = digitalRead(tasterPin);
  if (taster != taster_old)
  {
    taster_old = taster;
    return 1;
  }
  return 0;
}
```

Die Funktion `getTaster()` liefert also nur dann eine 1 zurück, wenn sich der Erschütterungssensor bewegt hat.

Steht der Würfel still, so soll der Wurf erst gestartet werden, wenn viele Male in kurzer Zeit so eine Veränderung gemessen worden ist, andernfalls würde der Würfel schon auf leichtes Anstoßen des Tisches reagieren. Dazu wird eine Zählvariable benutzt: `taster_entprellen` wird bei jedem Ereignis um 20 erhöht; tritt nichts ein, wird eins abgezogen. Erst wenn diese Zahl größer als 1.000 ist, soll der Wurf gestartet werden.

```
void loop()                    // Hauptschleife
{
  // warten auf einen Tastendruck zum Starten des Würfels
  taster_entprellen = 0;
  while( taster_entprellen < 1000)
  {
    taster_entprellen += getTaster()*20;
    if (taster_entprellen > 0) taster_entprellen--;
    delay(1);
  }
```

Die Variable `speed` bestimmt die Geschwindigkeit, mit der sich der Würfel dreht. Sie wird nun willkürlich auf 100 gesetzt und der Wurf gestartet. Dabei wird eine `do...while`-Schleife verwendet. Diese läuft mindestens einmal durch, maximal so lange, bis eine bestimmte Bedingung erreicht wird. Setzt man die `while`-Bedingung an den Anfang der Schleife, kann es passieren, dass die Schleife nicht ausgeführt wird, im Falle des Würfels soll aber zumindest einmal eine neue Zahl ermittelt werden.

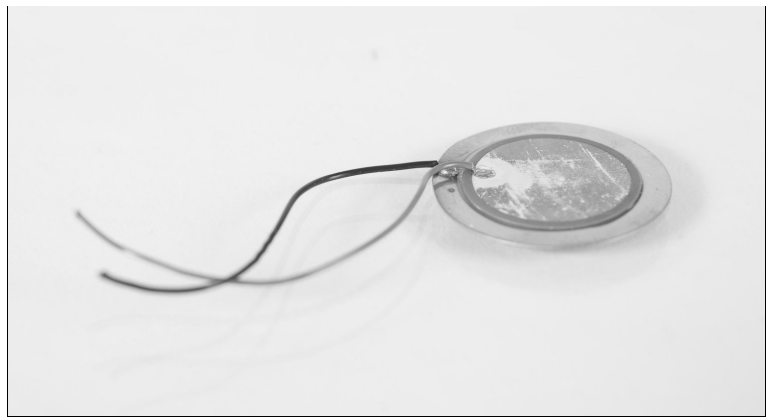
```
  // nun wird der Würfel gestartet
  // setzen der Würfelgeschwindigkeit auf hoch
  speed = 100;
  do
  {
```

```
    // immer, wenn der Taster seinen Zustand wechselt,
    // Geschwindigkeit wieder hochsetzen
    if (getTaster() == 1)
    {
      speed += 10;
    }
    // Würfel um eins weiterzählen
    // wenn größer als 6, wieder auf 1 setzen
    num++;
    if (num > 6) num = 1;
    // anzeigen
    display(num);
    // ein bisschen warten; je groeßer "Speed", desto weniger
    // wird gewartet
    delay(1000/speed);
    Speed--; // langsam langsamer werden.
  }
  while (speed > 1);
  // der Würfel ist stehen geblieben
}
```

Piezolautsprecher

Der Würfel soll nun einen Piezolautsprecher erhalten, um beim Wurf einen passenden Ton auszugeben.

Abbildung 7-5 ▶
Piezo



Piezoelektrische Elemente sind Bauteile, die beim Anlegen von Spannung ihre Form verändern oder bei einer Veränderung ihrer Form Spannung erzeugen. Piezolautsprecher gehören in die erste Kategorie: Liegt eine hohe Spannung an, können sie Ton ausgeben. Die möglichst hohe Spannung wird erreicht, indem der Piezo gleich an zwei Ausgangspins angeschlossen wird. Diese werden dann immer gegensätzlich geschaltet. In unserem Beispiel werden die

beiden Variablen `piezo1` und `piezo2` mit den Pins 6 und 7 verbunden und als Ausgang definiert.

In der Würfel-Displayfunktion werden nun bei jedem Hochzählen von `num` die beiden Pins umgeschaltet, der Piezo gibt ein leises Knacken von sich.

```
if ((num % 2) == 0)
{
  digitalWrite(piezo1, LOW);
  digitalWrite(piezo2, HIGH);
} else {
  digitalWrite(piezo1, HIGH);
  digitalWrite(piezo2, LOW);
}
```

Diese Schaltung kann dann in ein kleines Gehäuse gebaut werden, wenn man den Arduino mit einer Batterie ausstattet.

KAPITEL 8

Automation

In diesem Kapitel:

- Alles hört auf mein Kommando
- DMX
- Barlicht
- RF-Steckdosen
- Gespensterschreck

Alles hört auf mein Kommando

In diesem Kapitel geht es darum, Gegenstände wie Lampen, Heizung, Ventilatoren und Rollläden mit dem Arduino anzusteuern. Wie in Kapitel 1 und 2 angesprochen kann zur Beeinflussung von größeren Spannungen und Strömen ein Relais, Transistor oder Triac verwendet werden. Allerdings kann man dabei nicht vermeiden, dass man mit gefährlichen Spannungen hantieren muss.

Aus diesem Grund konzentrieren wir uns hier auf zwei ganz andere Methoden. Erstens DMX, ein weit verbreitetes Protokoll zur Ansteuerung von Scheinwerfern auf Bühnen und Diskotheken. Noch etwas universeller, aber mit mehr Bastelei verbunden: das Ansteuern von Funksteckdosen. Diese können für wenig Geld im Baumarkt um die Ecke oder beim Discounter erworben werden.

Einige Projekte sollen hier kurz vorgestellt werden, um die ganze Bandbreite der Möglichkeiten zu zeigen und die Fantasie anzuregen.

Buzzer to DMX

Für eine Quizshow werden an den Arduino mehrere Knöpfe (Buzzer) angeschlossen. Wird einer davon betätigt, sendet der Arduino über DMX den Befehl, den Kandidaten zu beleuchten. Leider sind z. Zt. auf der Webseite (*http://benjaminschneider.ch/?p=10*) weder Quellcode noch Schaltpläne vorhanden.

Specdrum

Lichtsäulen, die interaktiv von einer Trommelgruppe gesteuert werden. Sensoren in den drei Trommeln erkennen die einzelnen

Schläge. Je nach Tonhöhe werden über DMX verschiedene Farben zu RGB-Scheinwerfern in großen, matten Plastiktrommeln gesendet. *http://www.tangibleinteraction.com/blog/prototype_specdrum*

Forcefield Interactive

Mittels verschiedener RFID-Karten kann das »Lichtfeld« dieser Installation beeinflusst werden. Ein einzelner Arduino kontrolliert hier 192 Lichtstäbe. Die Farbmuster basieren hauptsächlich auf Perlinrauschen und den Daten der RFID-Karten der Besucher. Zusätzlich können mitgebrachte Gegenstände mittels eines Farbsensors abgescannt werden (*http://blogs.driversofchange.com/emtech/2009/01/forcefield_interactive.html)*.

DMX

DMX ist ein Standard zur Steuerung von Scheinwerfern in der Veranstaltungs- und Bühnentechnik. Er ersetzte die davor übliche Ansteuerung mit 0-10V Signalen. Bei diesen Systemen benötigte man zu jedem Scheinwerfer ein eigenes Kabel. Bei 30 oder mehr Scheinwerfern kann man sich vorstellen, wie die Rückseite des Lichtmischpults ungefähr ausgesehen haben mag. Dafür war aber auch nur ein beliebiges Multimeter notwendig, um den Zustand jedes Kabels inklusive der übertragenen Daten zu kontrollieren. DMX ersetzt alle diese Kabel durch ein einziges, und die analogen Signale werden digital übertragen. Bis zu 512 einzelne Geräte können in einer langen Kette hintereinander an diesen Bus angeschlossen und darüber gesteuert werden. An jedem kann, meistens über ein sogenanntes Mäuseklavier, eine individuelle Adresse eingestellt werden. Benötigt ein Gerät, z.b. ein LED-DMX-Scheinwerfer, mehrere Parameter (rot, grün, blau) so belegt dieser drei aufeinander folgende Adressen. Der nächste Scheinwerfer muss also drei Adressen weiter hinten anfangen. Die meisten LED-RGB-Scheinwerfer belegen fünf oder sogar sechs Adressen, weil dort noch weitere Sonderfunktionen vorhanden sind. So kann ein DMX-Bus mit seinen 512 Adressen rund 80–100 dieser Scheinwerfer kontrollieren.

Für rund 30 € stellen diese Scheinwerfer eine einfache und kostengünstige Möglichkeit dar, Haus und Hof effektvoll zu beleuchten.

Da DMX für sehr lange Kabel entworfen wurde, werden die Daten symmetrisch übertragen. Das bedeutet, es gibt zwei Datenleitungen. Ist die eine auf »High« geschaltet, wird die andere auf null

gesetzt und umgekehrt. Dadurch werden abgestrahlte Störungen weitgehend unterbunden, und der Empfänger kann eingestrahlte Störungen dadurch entfernen, dass er nur auf die Differenz der beiden Leitungen horcht. Da eine Störung auf beide Leitungen wirkt, verschwindet diese beim Differenzbilden wieder. Für ganz kurze Kabel und nur ein bis zwei Busteilnehmer könnte man dieses Signal auch mittels zwei Pins des Arduinos erzeugen. Besser ist es aber, dafür ein geeignetes IC zu verwenden. Dafür kann man den MAX-485 oder den günstigeren SN75176 einsetzen.

Wer dabei nicht löten möchte, kann ein DMX Shield einsetzen (*http://www.freeduino.de/wiki/arduino-dmx-shield*).

Protokoll

Das DMX-Protokoll basiert wie RS232 auf asynchronen (also ohne extra Taktleitung) übertragenen Bytes mit Start- und Stopbits. Die Geschwindigkeit ist mit 250kbit/s festgelegt. Das bedeutet damit auch, dass nur ca. 40–45 Änderungen pro Sekunde übertragen werden können, wenn alle 512 Adressen auf dem Bus belegt sind. Für normale Scheinwerfer ist das aber schnell genug.

Alle Daten werden in Paketen, sogenannten Frames, übertragen. Damit die DMX-Teilnehmer den Beginn eines solchen erkennen können, sendet der Master mindestens 88 µs lang eine Null auf den Bus. Durch die Stopbits kann dieser Zustand während einer normalen Übertragung niemals auftreten, so dass der Beginn eines Frames zweifelsfrei festgestellt werden kann.

Danach folgt als Erstes das Startbyte, das immer Null und für die spätere Erweiterung vorgesehen ist. Es folgen die entsprechenden Bytes, die immer mit der Adresse Null beginnen. Das heißt, ein DMX-Empfänger wartet auf den Start eines Frames und die darauf folgende Null. Dann ignoriert er so viele Bytes auf dem Bus, wie es seiner Adresse entspricht. Die dann folgenden Bytes sind die für ihn bestimmten Daten.

Software

Von *http://tinker.it* gibt es eine schöne Bibliothek für das DMX-Protokoll. Diese kann unter *http://code.google.com/p/tinkerit/wiki/DmxSimple* heruntergeladen werden. Wie schon von anderen Bibliotheken bekannt, werden die Dateien in das Verzeichnis [Arduino-Verzeichnis]/hardware/libraries/DmxSimple entpackt. Nach dem

Neustart der Arduino-Software kann die Bibliothek dann verwendet werden. Sie muss dann auch im Library-Menü auftauchen.

Mit #include <DmxSimple.h> wird die Bibliothek eingebunden (alternativ über das Menü) .

```
void setup() {
     DmxSimple.usePin(3); // Hier muss der Pin angegeben werden an
                          // welchem das DMX
                          // Interface angeschlossen ist. Meistens
                          // ist das PIN3.
}

void loop() {
   int brightness;
   for (brightness = 0; brightness <= 255; brightness++) {
      DmxSimple.write(1, brightness); // Setze DMX Kanal 1
      delay(10); // ein wenig warten, damit es nicht zu schnell wird.
    }
}
```

Bei manchen Geräten kann es vorkommen, dass sie nur funktionieren, wenn alle im Gerät vorhandenen Adressen auch beschrieben werden.

Barlicht

In diesem Projekt wird eine Thekenbeleuchtung für eine Bar gebaut. Je nach Länge der Theke besteht diese aus 1 bis 32 einzelnen DMX-LED-RGB-Scheinwerfern, die in einer Reihe von der Decke herab die Bar beleuchten. Solche Scheinwerfer können ab 35 EUR z.B. bei *www.thomann.de* bezogen werden. Die Grundidee bei allen abgespielten Animationen sollte dabei sein, dass der Gast nie im Dunkeln sitzt, und anders als bei einem Discolicht sind eher langsame, weiche Übergänge sinnvoll.

Hardware: Mehrere DMX-Scheinwerfer, die über das entsprechende Kabel und das DMX-Shield mit dem Arduino verbunden werden.

Je nach Modell des verwendeten Scheinwerfers benötigt dieser 3 bis 6 DMX-Adressen. Bei mehr als einem Scheinwerfer werden die Adressen so verteilt, dass sie sich nicht überschneiden. Bei Scheinwerfern mit 6 Adressen/Gerät z.B.: 0, 6, 12. Dabei kann es auch sinnvoll sein, den 6er-Abstand bei Geräten mit kleinerem Adressbereich einzuhalten, weil dann in der Software alle Scheinwerfer mit dem gleichen Schema angesprochen werden können.

```
#include <DmxSimple.h>

const int AnzahlScheinwerfer = 4;
int color[AnzahlScheinwerfer+1][3];
int update = 0;

void setup() {
  DmxSimple.usePin(3); // Hier muss der Pin angegeben werden an
                       // welchem das DMX
                       //  Interface angeschlossen ist. Meistens
                       // ist das PIN3.

  // Beim Start alle Scheinwerfer aus.
  for(int i = 0; i <= AnzahlScheinwerfer; i++) {
    color[i][0] = 0;    // Rot
    color[i][1] = 0;    // Gruen
    color[i][2] = 0;    // Blau
  }
}


void loop() {
  // Immer wenn update = 0 ist, denken wir uns eine neue Farbe aus.
  if (update == 0) {
    color[0][0] = random(0,256);    // Rot
    color[0][1] = random(0,256);    // Gruen
    color[0][2] = random(0,256);    // Blau
  }

  // Beim letzen Scheinwerfer anfangen und jeweils die Farbe
  // leicht an den Vorgänger angleichen, dadurch fließen die
  // Farben durch die Scheinwerfer-Reihe
  for(int i = AnzahlScheinwerfer; i > 0; i--) {
    // Farbberechnung getrennt für Rot,Grün,Blau durchführen.
    // jeweils 63 Teile alte Farbe + 1 Teil neue Farbe (vom nächst-
    // kleineren Nachbarn) geteilt durch 64 als neue Farbe annehmen.
    color[i][0] = (63*color[i][0] + color[i-1][0])/64;    // Rot
    color[i][1] = (63*color[i][1] + color[i-1][1])/64;    // Gruen
    color[i][2] = (63*color[i][2] + color[i-1][2])/64;    // Blau
    DmxSimple.write(i*6+0, color[i][0]); // Setze DMX Kanal rot
    DmxSimple.write(i*6+1, color[i][1]); // Setze DMX Kanal gruen
    DmxSimple.write(i*6+2, color[i][2]); // Setze DMX Kanal blau
  }

  // Zähler erhöhen, damit wir nach 10 eine neue Farbe setzen
  update++;
  if (update > 100) update = 0;

  delay(100); // ein wenig warten, damit es nicht zu schnell wird.
}
```

Wie in den Kommentaren beschrieben, wird alle 10 Sekunden der imaginäre Scheinwerfer 0 mit einer zufälligen Farbe geladen. Zehn mal pro Sekunde wird jetzt immer ein Vierundsechzigstel der Farbe an den Nachbarscheinwerfer weitergereicht, sodass ein fließender Farbverlauf erreicht wird.

RF-Steckdosen

In Bau- und Supermärkten werden immer wieder günstige Sets von Funksteckdosen angeboten. Damit können Lampen und andere Geräte im ganzen Haushalt kontrolliert werden, ohne dabei mit 230V in Berührung zu kommen oder neue Kabel verlegen zu müssen. Damit folgende Modifikationen durchgeführt werden können, sollte im dazugehörigen Sender ein HX2262, PT2262 oder LP801-IC werkeln. Das ist z.B. bei den Sets von Intertechno oder Unitec der Fall, die man in vielen Baumärkten bekommt.

Protokoll

Die ICs HX2262 und PT2262 gibt es in zwei verschiedenen Bauformen. Bei den 18poligen ICs ist der Pin 17 und bei den 20poligen der Pin 19 der Datenausgang zum eigentlichen Sendemodul. Bei dem von Intertechno verwendeten LP801 ist es der Pin 1. Wer die gesendeten Daten also genauer ansehen möchte, sollte hier sein Oszilloskop anschließen. Aus den Datenblättern und den Beobachtungen im Oszilloskop erhält man folgendes Datenprotokoll:

Alle Daten bestehen entweder aus einem kurzen Puls (25%) und einer langen Pause (75%) oder einem langen Puls (75%) und einer kurzen Pause (25%). Die erste Variante bezeichnen wir ab hier als »0«, die zweite als »1«.

Um möglichst viele verschiede Steckdosen mit diesem IC schalten zu können, hat sich der Hersteller einen interessanten Trick ausgedacht: Alle Eingänge können entweder mit Masse (0), oder mit der Betriebsspannung (1) verbunden werden, zusätzlich können diese aber auch unbeschaltet gelassen werden (F). Dadurch kann jeder Eingang drei verschiedene Zustände annehmen.

Bei nur zwei Möglichkeiten (0/1) hätte man bei zwölf Eingängen 2^12 Möglichkeiten, was 4096 Kombinationen entspricht. Hier sind es aber drei (0/1/F), also 3^12, also 531441 Möglichkeiten.

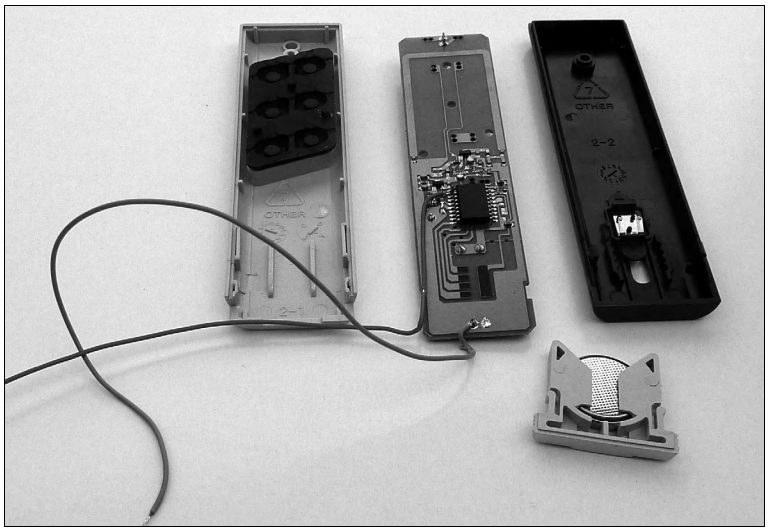
Wie können aber 3er-Zustände mittels Nullen und Einsen übertragen werden? Für jeden Eingang werden einfach zwei Bits nach folgender Tabelle gesendet:

Eingang	Daten	Datenausgang
Masse (0)	00	kurzer Puls, lange Pause, kurzer Puls, lange Pause
Betriebsspannung (1)	11	langer Puls, kurze Pause, langer Puls, kurze Pause
Offen (F)	01	kurzer Puls, lange Pause, langer Puls, kurze Pause

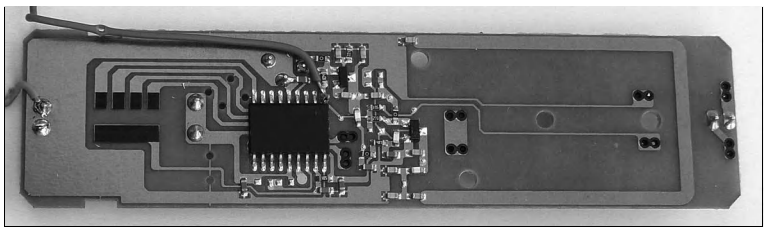
Es werden bei jedem Tastendruck so lange Telegramme gesendet, bis der Taster wieder losgelassen wird. Minimal müssen mindestens zwei identische Telegramme empfangen werden, damit der Schaltvorgang durchgeführt wird. Das soll Störungen vermeiden. Ein Telegramm besteht aus den Daten der 12 Eingangspins und einer abschließenden »0«, auf die 32 Takte Pause folgen.

Hardware

Am GND des Senders (da, wo der Minuspol der Batterie angeschlossen ist) wird eine Leitung angelötet, welche mit GND vom Arduino verbunden wird. Der Pin 2 des Arduino wird dann Pin 17 (18polige Bauform) oder Pin 19 (20polige Bauform) des HX2262 verbunden (Pin 1 bei den Intertechno Sendern).



◀ Abbildung 8-1



◀ Abbildung 8-2

Software

Als Erstes brauchen wir die RF-Bibliothek von *http://randysimons.com/overige/browsable/433MHz/ArduinoRemoteSwitchLibrary.7z* (Kopie bei *arduinobuch.wordpress.com*) welche ins Verzeichnis Arduino/libraries/RemoteSwitch entpackt wird.

Das wichtigste beiliegende Beispiel ist das Show_received_code-Script. Es ist eigentlich zur Verwendung mit einem Empfänger und zum Decodieren von eingehenden Funksignalen gedacht.

Da aber an dem hier verwendeten Pin 2 des Arduino nicht nur von uns Daten in das Funkmodul eingespeist werden können, sondern auch die zu sendenden Daten bei jeder Betätigung eine Knopfes auf der Fernbedienung herauskommen, können wir uns damit ansehen, welche Codes mit welcher Datenrate gesendet werden.

Nachdem der Code in den Arduino geladen und der Sender mit dem Pin 2 des Arduinos verbunden ist, sollte in der seriellen Konsole (»serial monitor«, letzter Button oben rechts im Arduinofenster) bei jedem Tastendruck auf der Fernbedienung Folgendes zu sehen sein:

```
Code: 26, period duration: 362us.
Code: 24, period duration: 362us.
Code: 4400, period duration: 362us.
Code: 4398, period duration: 362us.
Code: 1484, period duration: 362us.
Code: 1482, period duration: 362us.
```

Code 26 entspricht also der ersten Taste der Fernbedienung, Code 24 der zweiten und so weiter. Die 362us sind ein Maß für die Sendegeschwindigkeit der jeweiligen Fernbedienung. Alle diese Daten benötigen wir, wenn wir vom Arduino aus Schaltbefehle senden wollen.

Für das Senden von Kommandos ist die Funktion `RemoteSwitch::sendTelegram(code,sendPin)` zuständig. Allerdings ist der hier verwendete Code anders aufgebaut als bei der Empfangsfunktion. Hier hilft die Funktion `SendCode`, die die benötigten Umwandlungen vornimmt.

```
void SendCode(unsigned long code, int period) {
  // Datenformat:
  // pppppppp|prrrdddd|dddddddd|dddddddd (32 bit)
  // p = Geschwindigkeit (9 bit unsigned int
  // r = Wiederholungen als 2log. r = 3 bedeutet 2^3=8 mal senden.
  // d = Code

  code |= (unsigned long)period << 23;
  code |= 3L << 20;      // 3L = 8 mal wiederholen
                         // 4L = 16 mal wiederholen
  RemoteSwitch::sendTelegram(code,sendPin);
}
```

Ein einfaches Beispiel:

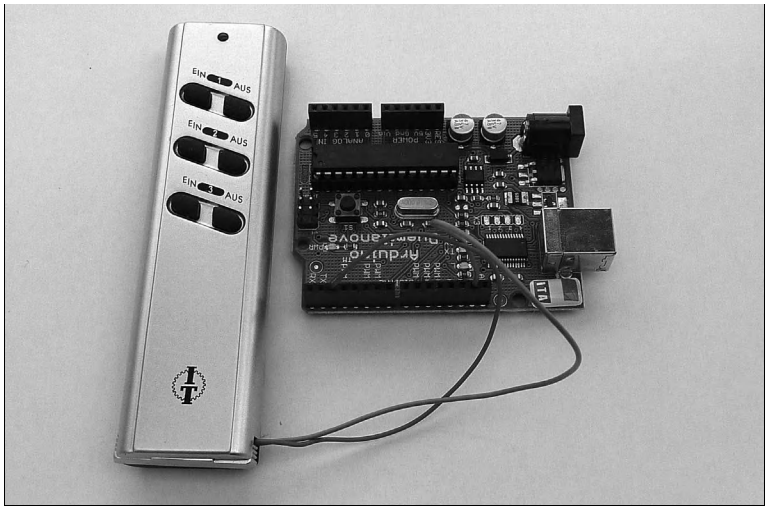
```
void loop() {
  // Steckdose A1 an
  SendCode(24,362);
  delay(5000);    // 5 Sekunden warten
  // Steckdose A1 aus
  SendCode(26,362);
  delay(5000);    // 5 Sekunden warten
}
```

Gespensterschreck

Nicht nur Gespenster, sondern auch Diebe und Einbrecher lassen sich mit unregelmäßig ein- und ausgeschalteten Lichtern vertreiben. Mittels einfacher Zeitschaltuhren lässt sich das zwar auch realisieren, man müsste dann aber beim Verlassen des Hauses immer daran denken, sie auch alle zu aktivieren.

Hardware

Zusätzlich zum Arduino werden eine oder mehrere Funksteckdosen-Sets benötigt. Wie unter Hardware beschrieben wird der Sender der Funksteckdosen modifiziert und mit dem Arduino verbunden.



◀ Abbildung 8-3

Software

Als Erstes benötigen wir eine Tabelle mit allen vorhandenen Steckdosen und deren zugehörigen Ein- und Ausschaltcodes. (Der auf-

merksame Leser wird hier erkennen, dass der Code für AUS bei der Intertechno-Fernbedienung immer um zwei kleiner ist als der dazugehörige Code für EIN. Es würde also genügen, nur einen der beiden zu speichern, allerdings wäre der Code dann auf genau diese Fernbedienung festgeschrieben.)

```
int anzahlSteckdosen = 3;

unsigned long steckdosen[][2] = {
  {26,24},            // Steckdose 1 (an/aus)
  {4400,4398},        // Steckdose 2 (an/aus)
  {1484,1482}         // Steckdose 3 (an/aus)
};
```

Mit der Hilfe der Funktion `random(max)` werden jetzt beliebige Steckdosen an- oder ausgeschaltet. Zwischen den Schaltvorgängen wird jeweils 15 Minuten gewartet, damit es realistisch aussieht.

```
void loop() {
  // zufällig eine Steckdose auswählen.
  int steckdose = random(anzahlSteckdosen);
  // zufällig entweder an oder aus senden.
  int aktion    = random(2);
  SendCode(steckdosen[steckdose][aktion],period);
  // 15 Minuten warten
  for (int i = 0; i < 15; i++) {
    delay(60*1000); // 1 Minute warten
  }
}
```

Die Erweiterung, dass nur Funksignale gesendet werden, wenn ein zusätzlicher Helligkeitssensor, am Arduino angeschlossen, Dunkelheit meldet, sei dem geneigten Leser als Übung überlassen.

Referenzen

http://randysimons.com/overige/browsable/433MHz/
RF-Bibliothek

http://www.mikrocontroller.net/topic/124084
Intertechno-Funksteckdosen per AVR steuern

http://avr.börke.de/ARCTECHsteckdosen.htm
ARCTECH-Funksteckdosen

http://avr.börke.de/Funksteckdosen.htm
Die Ansteuerung von Funksteckdosen

http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1216065789
Arduino & HT2262

KAPITEL 9

Wearable Computing

In diesem Kapitel:

- Programmierbare Kleidung
- Wearable Komponenten
- Die iPod-Steuerung im Mantel

Das folgende Kapitel behandelt einen kleinen Exkurs in die Welt des Wearable Computing. Gleichzeitig soll erklärt werden, wie ein Arduino mit einem iPod oder iPhone kommunizieren kann. Als praktisches Beispiel wird eine tragbare Fernbedienung für einen iPod oder ein iPhone gebaut. Im Folgenden wird das iPhone nicht mehr gesondert erwähnt, da es keinen für das Projekt relevanten Unterschied gibt.

Programmierbare Kleidung

Der Trend zur programmierbaren Kleidung hat in den letzten Monaten stark zugelegt. Gab es bei der Erstellung der ersten Auflage dieses Buches nur sehr wenige Ressourcen und war Elektronik zum Anziehen eher ein Nischenthema, gibt es nun eine ganze Reihe von Webseiten, die sich damit befassen. Unter dem Schlagwort Wearable Computing werden zwei grundsätzliche Bestandteile zusammengefasst: Zum einen Elemente wie Fäden und Stoffe, die auch in herkömmlicher Kleidung vorkommen. Diese gibt es auch in leitenden Varianten und werden als solche in erster Linie in der Industrie eingesetzt, oder zum Beispiel bei Fechtanzügen. Leitende Fäden können zum Beispiel von Sparkfun erstanden werden, hier gibt es in Deutschland einige Händler wie *http://www.watterott.com*.

Leitende Stoffe sind schwieriger und oft nur über die USA zu beziehen. Sachdienliche Hinweise hierzu nehmen wir gerne auf der Website zum Buch, *http://arduinobuch.wordpress.com*, entgegen.

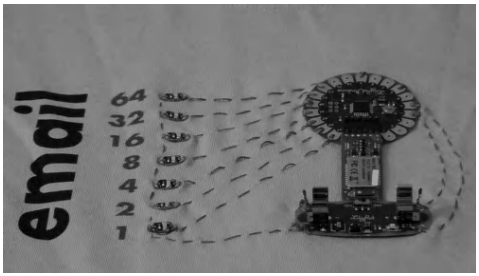
Zum anderen besteht das Wearable Computing aus klassischer Elektronik, die oftmals angepasst wurde, damit sie besonders gut in

Kleidung vernäht werden kann. Die Firma Sparkfun hat in Zusammenarbeit mit Leah Buchley (*http://web.media.mit.edu/~leah/LilyPad/*) das LilyPad entworfen, ein Arduino-Board, das sich durch seine Eigenschaften besonders gut in Kleidung einarbeiten lässt. Inzwischen ist unter diesem Label eine ganze Reihe an Komponenten entstanden. Dazu gehören LEDs, Temperatursensoren oder Schalter. Eine ganze Bandbreite dieser Produkte wird unter *http://www.tinkersoup.com* angeboten.

Im Internet findet man mittlerweile eine große Reihe an interessanten Projekten, die Kleidung und Computer verbinden. Am häufigsten dürfte dabei die LED vorkommen. Jacken, T-Shirts und Kleider, die dank Elektronik leuchten, haben schon eine recht lange Tradition und sind nicht erst seit dem Aufkommen von LilyPad und Co. aufgetaucht. Ein besonders interessantes Projekt, das LEDs verwendet, ist eine Jacke für Radfahrer, die Leah Buchley auf ihrer Seite beschreibt. Zeigt man eine Richtung an, leuchtet ein Pfeil in die entsprechende Richtung. Besonders im Dunklen eine sinnvolle Sache. Die koreanische Künstlerin Joo Youn Paek schuf 2006 eine Klanginstallation, die durch das Öffnen und Schließen verschiedener Reißverschlüsse gesteuert werden konnte.(*http://itp.nyu.edu/~jyp243/jy/ziporch.htm*)

Laura Bioffi, eine Studentin aus Kopenhagen, strickte einen Handschuh, der zusammen mit einem LilyPad in der Lage ist aufzuzeichnen, wann immer einem Kommilitonen die Hand gegeben wird.

Abbildung 9-1 ▶
T-Shirt mit LilyPad



Madeleine und Chris Ball entwarfen ein T-Shirt, das die Anzahl ungelesener E-Mails visualisiert. Dazu wurde die LilyPad-Platine mit einer Bluetooth-Erweiterung verbunden, die damit über ein Handy die Zahl der ungelesenen E-Mails abrufen kann. Um bis zu 127 ungelesene EMails visualisieren zu können, wird die

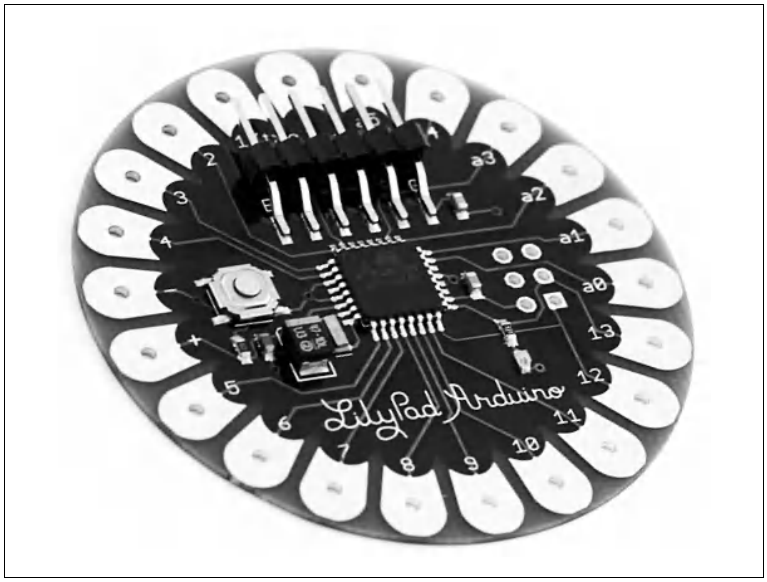
Anzahl binär dargestellt: *http://blog.printf.net/articles/2010/03/30/email-counting-tshirt*.

Andere Projekte basieren auf nützlicheren Konzepten, wie etwa Vibrationsmodulen, die Menschen mit Gleichgewichtsstörungen helfen sollen, ihre Balance zu halten. Eine besonders umfangreiche Website, die sowohl viele Tipps zum Selbermachen bietet als auch regelmäßig interessante Projekte vorstellt, ist *http://www.talk2myshirt.com*.

Wearable Komponenten

LilyPad

Das LilyPad ist ein abgespeckter Arduino, speziell für das Vernähen in Wearable-Projekten konzipiert. Da der ganze Programmierteil des Arduinos fehlt, kann ein vorhandener Arduino Hilfe leisten. Im Prinzip verwendet man dessen TX- und RX-Pins, muss jedoch auch beachten, dass man den Mikrocontroller vorher vom Board entfernt hat. Eine genaue Anleitung findet man unter *http://web.media.mit.edu/~leah/LilyPad/01_computer_attach.html*. Oder noch einfacher verwendet man das dazu passende FTDI-Breakeout Board, welches direkt mit dem LilyPad bezogen werden kann.



◀ **Abbildung 9-2**
LilyPad (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)

Leitender Faden

Einer der wichtigsten Bestandteile von Elektronik in Stoffform ist der leitende Faden. Er besteht meistens aus metallisch (mit Silber) bedampften Trägerfäden. Je nach verwendetem Metall und Schichtdicke erhält man unterschiedliche Widerstände und Haltbarkeit gegenüber Waschen und Brechen. Gewöhnlich liegt der zu erwartende Widerstand bei ca. 0,5 Ohm pro Zentimeter. Zu lange Leitungen, gerade für etwas stromhungrige Bauteile wie LEDs können ein Problem darstellen. Dem kann durch mehrfache Fäden abgeholfen werden. Leitender Faden kann inzwischen bei einigen Elektronik-Fachhändlern bezogen werden. So führt zum Beispiel Watterott Elektronik zwei verschiedene Sorten leitenden Faden. Sparkfun führt auch eine stetig wachsende Zahl von Wearable-Komponenten.

Abbildung 9-3 ▶
Leitender Faden (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)

Leitender Stoff

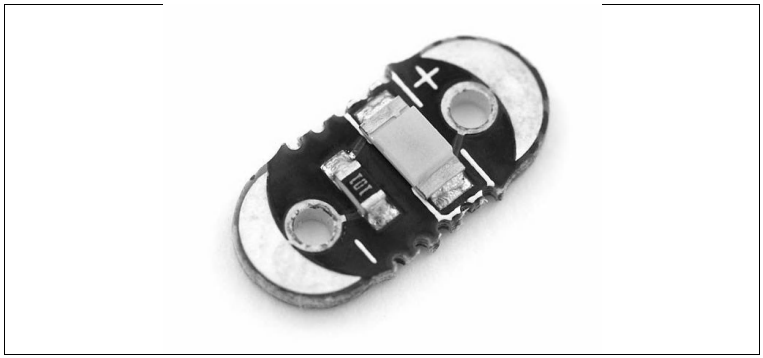
Für großflächige Kontakte und mehrlagige Konstruktionen ist leitfähiger Stoff eine interessante Sache. Allerdings ist er sehr schwer zu bekommen. Eine Quelle ist der Fechtturnierbedarf. Dort werden sogenannte Elektrowesten verwendet, um Treffer des Gegners erkennen zu können. Diese sind aus leitfähigem Stoff hergestellt.

Leitender Schaumstoff

Viele Elektronikkomponenten werden zum ESD-Schutz (das sind Beschädigungen durch elektrostatische Aufladung) auf schwarzem oder rosa Schaumstoff ausgeliefert. Je nach Hersteller ist die Leitfähigkeit aber leider sehr verschieden, von wenigen 100 Ohm bis zu mehreren Megaohm. Mit diesem Schaumstoff lassen sich z.B. Drucksensoren bauen, indem man diesen Schaumstoff zwischen zwei Stoffschichten vernäht, die jeweils einen leitfähigen Faden enthalten. Je stärker man auf sie eindrückt, desto niederohmiger werden sie. Diese Veränderung kann mit einem PullUp als Analogsignal vom Arduino gelesen und verarbeitet werden.

LEDs

Ganz normale bedrahtete LEDs lassen sich, nachdem man die Anschlussdrähte zu kleinen Ösen gebogen hat, mit leitendem Faden vernähen. Besonders kleine SMD-LEDs integrieren sich gut in Kleidungsstücke (Sie reißen nicht so leicht ab wie ihre großen Schwestern, wenn sie an den Anschlüssen mit kleinen Ösen aus Draht versehen werden. Passend zur LilyPad-Platine gibt es von Sparkfun bereits vorgefertigte kleine LED-Platinen, die ebenfalls Ösen zum Vernähen haben.

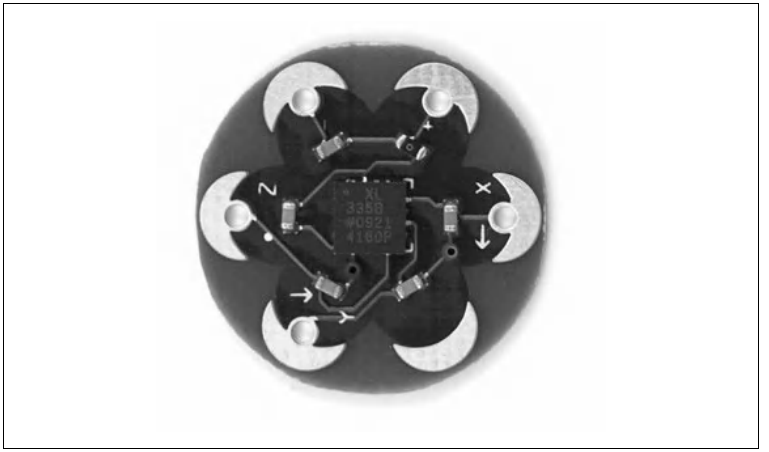


◀ **Abbildung 9-4**
LilyPad-LED (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)

Beschleunigungssensor

Ebenfalls bei Sparkfun gibt es einen Beschleunigungssensor zum Annähen. Damit kann man zum Beispiel die Armpositionen messen oder Bewegungsmuster erkennen, um dann verschiedene LED-Muster abzuspielen oder einfach nur einen Tagesablauf aufzuzeichnen.

Abbildung 9-5 ▶
LilyPad-Beschleunigungssensor (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)



Lautsprecher

Ein kleiner magnetischer Lautsprecher, gerade laut genug, um den Träger auf wichtige Zustände und Meldungen aufmerksam zu machen.

Abbildung 9-6 ▶
LilyPad-Lautsprecher (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)

Stromversorgung

Verschiedene Batteriefächer können einen Arduino oder ein LilyPad mit Strom versorgen. 3 AA- oder AAA-Batterien/Akkus reichen dabei für ein LilyPad aus. Ein Arduinoboard benötigt etwas mehr Spannung, da darauf ein einfacher Spannungsregler verbaut ist. Dieser benötigt mindesten ca. 7V um den Arduino mit 5V zu versorgen. Ohne diesen Regler läuft auch das Arduinoboard mit 3 Batterien. Sind diese nicht mehr ganz voll, kann der Mikrocontroller unter Umständen aber nicht mehr richtig arbeiten. Sparkfun bietet

eine Lösung mit nur einer Batterie, hierbei wird die Spannung mittels eines Schaltreglers von 1,5V auf 5V erhöht.

◀ **Abbildung 9-7**
LilyPad-Batteriehalterung (Abbildung mit freundlicher Genehmigung von Sparkfun Electronics)

Die iPod-Steuerung im Mantel

Um die grundlegenden Prinzipien des Wearable Computing zu erläutern, soll im Folgenden nun eine Jacke, ein Mantel oder ein anderes geeignetes Kleidungsstück mit einer einfachen Steuerung für den iPod ausgestattet werden.

Materialien

Neben dem Arduino (wenn möglich einem LilyPad) werden die typischen Utensilien zum Nähen benötigt. Also eine Nähnadel oder Nähmaschine, leitender Faden oder sehr dünner Draht, metallene Textil-Druckknöpfe und Schaumstoff. Für die Verbindung mit dem iPod oder iPhone (was natürlich ebenfalls vorhanden sein sollte) benötigt man einen iPod-Dock-Konnektor. Außerdem ist es sinnvoll, gleich daran zu denken, dass man das Kleidungsstück, nennen wir es hier »die Jacke«, auch unabhängig von einem Netzteil betreiben möchte. Ein Batteriefach für 3 AA- oder AAA-Batterien kann dabei sehr hilfreich sein.

Technik

Wie oben schon erwähnt, lohnt es sich bei Wearables-Projekten über den Kauf eines »LilyPad« nachzudenken. Diese Arduino-Board ist rund, flach und klein, sodass es besonders gut in einem Kleidungsstück versteckt werden kann. Angeblich ist es waschbar, auf jeden Fall eignet es sich jedoch besonders gut, um leitenden Faden anzuknüpfen, weil die Pins mit Löchern versehen sind.

Druckknöpfe: Der wohl denkbar einfachste vernähbare Schalter ist ein einfacher, metallener Druckknopf. Solche Knöpfe sind in jedem Kurzwarenladen oder Kaufhaus erhältlich, meist in 12er-Packungen für unter 3 Euro. Da beide Teile des Knopfes leitend sind, ist das Prinzip einfach: ist er offen, so fließt kein Strom, wird er geschlossen, so entsteht auch ein Stromkreis.

Abbildung 9-8 ▶
Druckknopfschalter

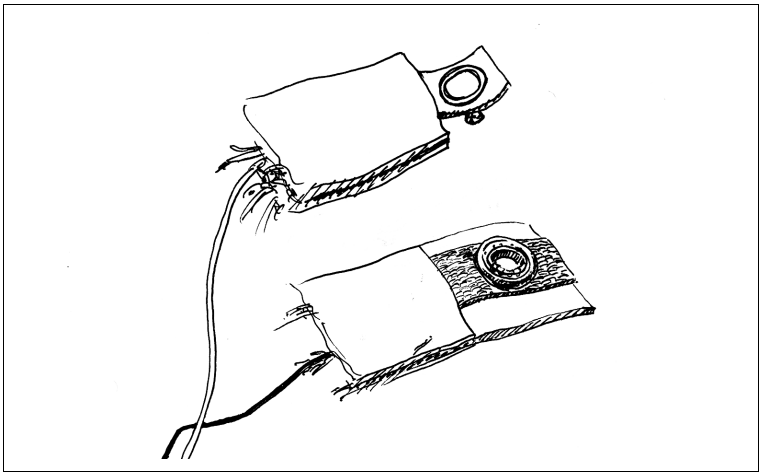
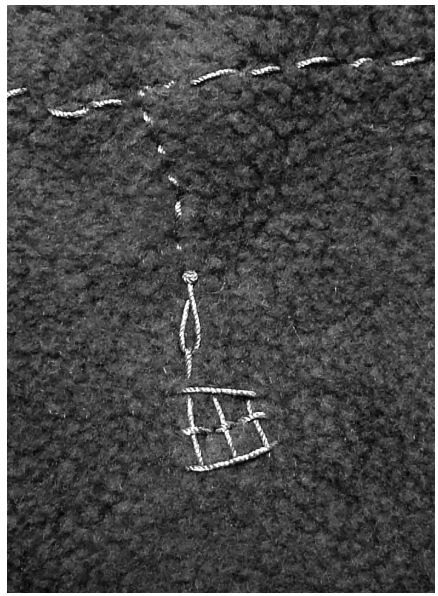
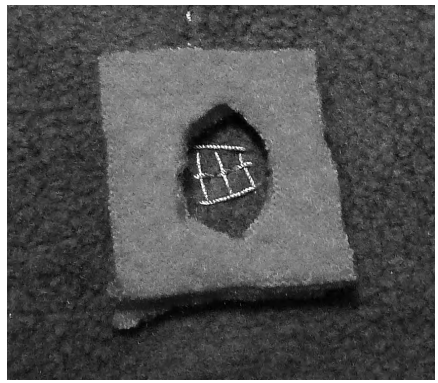


Abbildung 9-9 ▶
Druckknopfschalter plus LED

Schaumstofftaster: Aus Schaumstoff können einfache Taster gebaut werden. Da sich der Schaumstoff wieder in seine Ausgangsposition zurückbewegt, nachdem er gedrückt wurde, kann er als Abstandhalter zwischen zwei leitenden Schichten verwendet werden. Drückt man ihn zusammen, schließt sich der Stromkreis, lässt man los, weitet sich der Taster wieder, es fließt kein Strom mehr.



◀ **Abbildung 9-10**
Vorbereitete Stelle für den Taster



◀ **Abbildung 9-11**
Abstandhalter aus Schaumstoff oder anderem dicken Stoff.

Abbildung 9-12 ▶
Vorbereiteter »Deckel« mit leitfähigem Faden in der Mitte zur Kontaktierung

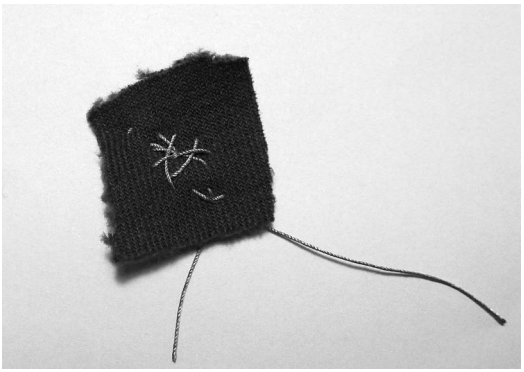


Abbildung 9-13 ▶
Fertiger Tasterstapel mit Anschlussfaden.

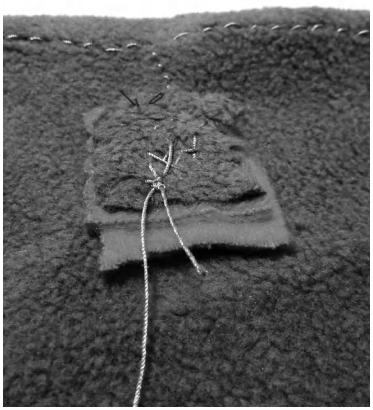


Abbildung 9-14 ▶
Fertiger Taster mit LED

iPod/iPhone: Das Dock des iPods enthält einen Stecker mit 30 Pins mit unterschiedlichen Funktionen, darunter auch die Möglichkeit den iPod fernzusteuern. Eine genaue Belegung kann unter *http://pinouts.ru/PortableDevices/ipod_pinout.shtml* gefunden werden. iPod-Dock-Konnektoren können bei einigen Anbietern, etwa Tinkersoup, gekauft werden und haben den Vorteil, dass alle Pins verlötet werden können. Sieht man sich den Stecker eines iPod-USB-Kabels an, stellt man fest, dass nur einige Pins tatsächlich verlötet sind. Es ist allerdings nicht auszuschließen, dass es, etwa bei eBay, Kabel von anderen Anbietern gibt, die man verwenden kann. Für dieses Projekt sind die Pins 1, 12 und 13 relevant. Sie stellen die TX- und RX-Verbindungen der seriellen Schnittstelle zum iPod dar. Im verlaufe dieses Projektes wird auch das Protokoll erläutert, mit dem der iPod ferngesteuert werden kann.

Nähen: Im Rahmen dieses Buches sind wir leider nicht in der Lage, einen detaillierten Exkurs in die Kunst des Nähens zu bieten. Im Internet gibt es jedoch hervorragende Tutorials, die jede mögliche Technik und jeden Stich genau erklären. Hier sei zum Beispiel auf *http://sewingtutorials.blogspot.com/* verwiesen; aber auch auf der Webseite von Burda Moden (*http://www.burdafashion.com*) gibt es gut gemachte Videos. Der Einfachheit halber wird im Folgenden nur von »Nähen« gesprochen. Ob dies mit der Nähmaschine oder per Hand geschieht, bleibt dem Leser selbst überlassen. Allerdings gilt Folgendes zu beachten, wenn man leitenden Faden in der Nähmaschine verwenden will. Dieser neigt dazu, sich sehr schnell zu verdrehen. Deshalb empfiehlt es sich, den leitenden Faden als Unterfaden einzunähen. Dazu wird er auf die Unterfadenspule einer Nähmaschine gerollt und ein einfacher, dünner Faden von oben durch die Nadel geführt. Die beiden Fäden werden ineinander verschlungen, sodass der dünne Faden den leitenden auf dem Stoff festzurrt. Eine genaue Anleitung findet man unter anderem unter *http://www.burdastyle.de/videos/how-tos/tutorial-ober-und-unterfaden_aid_1543.html*

Aufbau: Dieses Projekt hat den Anspruch, vollständig waschbar zu sein. Um zu verhindern, dass das Board, sei es nun LilyPad oder ein anderes, zu großer Belastung ausgesetzt wird, und auch um Korrosion am iPod-Konnektor zu vermeiden, empfiehlt es sich, eine Tasche für all die Elektronik zu nähen. Packt man so viel Elektronik wie möglich, vor allem Board und Batteriefach, in diese Tasche, kann man alle Pins über leitenden Faden und Druckknöpfe auf die äußere Seite der Tasche verbinden. Die Tasche kann also jederzeit

komplett aus dem Kleidungsstück genommen werden und muss nicht mitgewaschen werden.

Play-Knopf: Der wohl einfachste Teil dieses Projektes ist ein Play/Pause-Knopf. Hier dient ein Druckknopf, der auf der einen Seite mit einem Pin, auf der anderen mit GND verbunden wird. Der Arduino wird später so programmiert, dass er ein »Play«- oder »Pause«-Kommando an den iPod sendet, sobald der Knopf geschlossen oder geöffnet wird.

Lautstärke: Das Fernbedienungsprotokoll des iPod kennt nur die Kommandos »Lautstärke erhöhen« und »Lautstärke verringern«. Es ist also leider nicht möglich, den aktuellen Status festzuhalten. Für dieses Projekt empfehlen sich deshalb zwei Taster. Hierzu wird, wie in Abbildung 9-11 dargestellt, ein Stück Schaumstoff zwischen zwei leitende Stoffschichten genäht. Für diese Schichten reicht es, leitenden Faden einzunähen, wobei die eine Schicht mit dem GND-Pin des Arduino, die andere mit einem beliebigen digitalen Pin verbunden wird. In den Schaumstoff wird ein fingergroßes Loch geschnitten, durch das sich die beiden Schichten verbinden können, sobald man mit einem Finger darauf drückt.

Aufbau: Hat man die Einzelteile zusammen, kann man einen ersten Aufbau versuchen: Der Druckknopf und die beiden Taster werden mit einem Arduino verbunden. Die beiden TX/RX-Pins des Arduino werden mit den Pins 12 und 13 des iPod-Konnektors verlötet, zusätzlich muss der GND des Arduinos mit einem der GND-Anschlüsse des iPod-Konnektors verbunden werden. Nun kann mit der Programmierung begonnen werden. Zunächst soll der Druckknopf als Schalter fungieren. Der entsprechende Pin wird also auf »Input« gesetzt, und in der `loop()`-Funktion kontinuierlich ausgelesen.

Programmierung

Zunächst empfiehlt es sich, den iPod einmal ohne die selbst gebastelten Sensoren anzusteuern. So kann sichergestellt werden, dass die serielle Verbindung fehlerlos funktioniert. Hierzu wird zunächst die Bibliothek `Serial` eingebunden. Anschließend wird im `setup()` eine serielle Verbindung mit 19200 Baud gestartet. Die Verbindung läuft nur mit dieser Geschwindigkeit. Anschließend können Kommandos an den iPod gesendet werden. Diese Kommandos fol-

gen dem iPod Accessory Protocol, das unter anderem unter *http://ipodlinux.org/wiki/Apple_Accessory_Protocol* dokumentiert ist.

header	2	0xFF 0x55
länge	1	länge des modus + kommando + parameter
modus	1	der Modus, auf den sich das Kommando bezieht
kommando	2–4	das zwei bis vier Byte lange Kommando
parameter	0..n	optionaler Parameter, je nach Kommando
checksum	1	0xXX - (8 bit additive checksum) (0x100 - (summe aller länge/modus/kommando/parameter bytes) & 0xFF)

Die Modi beziehen sich dabei auf verschiedene Möglichkeiten, mit dem iPod zu kommunizieren. In unserem Falle geht es um die einfache Fernbedienung (Simple Remote), die im Vergleich zu anderen Protokollen (etwa Nike+) vollständig entschlüsselt ist. Die wichtigsten Kommandos finden sich in einer verkürzten Tabelle (siehe Tabelle unten) – für eine erweiterte Liste, die den Rahmen dieses Buches sprengen würde, empfiehlt sich der oben angegebene Link.

Kommando	Funktion
0x00 0x00	Button Released
0x00 0x01	Play/Pause
0x00 0x02	Vol+
0x00 0x04	Vol-
0x00 0x08	Skip >>
0x00 0x10	Skip <<
0x00 0x80	Stop
0x00 0x00 0x01	Play (kein Pause)
0x00 0x00 0x02	Pause (kein Play)

Für dieses Projekt werden vier Kommandos benötigt: 0x00 0x00 0x01 für Play, 0x00 0x00 0x02 für Pause und 0x00 0x02 bzw 0x00 0x04 für Vol+ und Vol-. Diese Kommandos benötigen keine weiteren Parameter, also ist dieser Teil 0. Das Längen-Byte des Kommandos wird also je nachdem auf 3 oder 4 gesetzt. Für die Checksumme empfiehlt es sich, eine eigene kleine Funktion zu schreiben, wenn man mehr als nur ein paar Befehle benötigt:

```
int checkSum(int* cmd) {
  int sum = 0;
  for(int i=2; i<=cmd[2]+2;i++) {
    sum += cmd[i];
```

```
  }
  int checksum = 0x100 - (sum & 0xFF);
  return checksum;
}
```

Für den Play-Pause-Befehl gilt demnach zum Beispiel die Checksum FA, die man sich per Serial.print(checksum, HEX) auch ausgeben lassen kann.

So muss man sich keine Gedanken mehr um die benötigte Checksum machen und kann ein einfaches Kommando an den iPod schicken. Zu Anfang sollte stets ein »Button release«-Kommando gesendet werden, um zu signalisieren, dass kein Knopf auf dem iPod (oder auf dem Kleidungsstück) gedrückt ist:

```
int buttonRelease[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x00,0xFB};
```

Diese Tabelle kann nun nach und nach gesendet werden:

```
for (int i = 0; i < 8; i++) {
Serial.print(buttonRelease[i],BYTE);
}
```

Nun können auch die anderen Kommandos in solche Tabellen geschrieben werden: Hierzu wird eine Funktion genommen, die als Eingabe die einzelnen Parameter nimmt und die entsprechende Tabelle zurückgibt. Da verschiedene Kommandos aus einer unterschiedlichen Anzahl Bytes bestehen, empfiehlt es sich, diese Information dem Array, welches das Kommando speichert, mitzugeben. In diesem Falle wird einfach die ohnehin später benötigte Längeninformation genutzt, sodass das Kommando ohne Checksum so aussieht:

```
int buttonPlay[] = { 0xFF, 0x55, 0x04, 0x02, 0x00,0x00, 0x01};
```

Nun kann eine Funktion »sendCommand« geschrieben werden, die das entsprechende Kommando zusammenbaut und gleich versendet:

```
void sendCommand(int* cmd) {
  int cs = checkSum(cmd); //Checksum bauen
  for(int i = 0;i<=cmd[2]+2;i++) {
    Serial.print(cmd[i], BYTE); // alle Bytes des Kommandos
übermitteln
 }
 Serial.print(cs, BYTE); //Checksum übermitteln
}
```

Nun kann alles zusammengefügt werden:

```
#include <Bounce.h>

// IPod Befehle
int buttonRelease[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x00};
```

```
int buttonPlayPause[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x01};
int buttonPlay[] = {0xFF, 0x55, 0x04, 0x02, 0x00,0x00, 0x01};
int buttonPause[] = {0xFF, 0x55, 0x04, 0x02, 0x00,0x00, 0x02};
int buttonVolPlus[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x02};
int buttonVolMinus[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x04};
int buttonVolRelease[] = {0xFF, 0x55, 0x03, 0x02, 0x00, 0x00};

// Arduino Pins für Taster und Play/Pause Switch
int volPlusPin = 2;
int volMinusPin = 3;
int playPausePin = 4;
int ledPin = 13;
int playStatus;

// Die Bouncezeit auf 500ms festlegen, damit
// zufällige Berührungen nichts bewirken.
Bounce bouncePlusPin = Bounce( volPlusPin, 500);
Bounce bounceMinusPin = Bounce( volMinusPin, 500);
Bounce bouncePlayPausePin = Bounce( playPausePin, 500);

void setup() {
  Serial.begin(19200);

  // Eingänge
  pinMode(volPlusPin, INPUT);
  pinMode(volMinusPin, INPUT);
  pinMode(playPausePin, INPUT);

  // Ausgang für LED
  pinMode(ledPin, OUTPUT);

  // Pull-Ups einschalten, dann brauchen wir keine
  // externen Widerstände.
  // Taster und Schalter werden mit GND verbunden.
  digitalWrite(playPausePin, HIGH);
  digitalWrite(volMinusPin, HIGH);
  digitalWrite(volPlusPin, HIGH);

}


void sendCommand(int* cmd) {
  int cs = checkSum(cmd);
  for(int i = 0;i<=cmd[2]+2;i++) {
    Serial.print(cmd[i], BYTE);
  }
  Serial.print(cs, BYTE);
}

int checkSum(int* cmd) {
```

```
  int sum = 0;
  for(int i=2; i<=cmd[2]+2;i++) {
    sum += cmd[i];
  }
  int checksum = 0x100 - (sum & 0xFF);
  return checksum;
}

void loop() {
  if (bouncePlayPausePin.update()) {
    // Druckknopf geöffnet -> Pull-Up macht den Eingang HIGH
    if ( bouncePlayPausePin.read() == HIGH) {
      // Musik aus
      sendCommand(buttonPause);
      sendCommand(buttonRelease);
      playStatus = 0;
    }
    else {
      // Druckknopf geschlossen -> Pin = GND/LOW
      // Musik an
      sendCommand(buttonPlay);
      sendCommand(buttonRelease);
      playStatus = 1;
    }
  }

  if (bouncePlusPin.update()) {
    // Taster gedrückt = GND/LOW
    if ( bouncePlusPin.read() == LOW) {
      sendCommand(buttonVolPlus);
      sendCommand(buttonRelease);
    }
  }

  if (bounceMinusPin.update()) {
    // Taster gedrückt = GND/LOW
    if ( bounceMinusPin.read() == LOW) {
      sendCommand(buttonVolMinus);
      sendCommand(buttonRelease);
    }
  }

  if (playStatus == 1) {
    // Ist der Ipod in Betrieb, soll die LED blinken
    digitalWrite(ledPin,(millis()/256) % 2);
  } else {
    // LED aus
    digitalWrite(ledPin,LOW);
  }
}
```

Tipps & Tricks

Der etwas lose Aufbau mit leitendem Draht und vielen nur verknoteten Kontaktstellen kann manchmal zu unerwarteten Ergebnissen führen. Meistens kann das mit einem Voltmeter oder Durchgangsprüfer einfach gefunden werden. Besonders hilfreich ist es da, in der Software das zu beobachtende Signal mittels einer LED (z.B. bei einem normalen Arduino Pin 13 mit der eingebauten LED) wieder auszugeben. Dann kann man das Stoffobjekt bewegen und knautschen, um den Fehler hervorzurufen, und hat mittels der LED eine direkte Beobachtungsmöglichkeit.

Bei Tastern und ähnlichen Kontakten hilft die Debounce-Bibliothek, zu kurze Signale von zufälligen Berührungen zu ignorieren. Wenn eine Funktionalität wirklich nur willentlich ausgelöst werden soll, hilft eine Kombination von mehreren Tastern/Sensoren die entweder gleichzeitig oder in einer bestimmten Reihenfolge gedrückt werden müssen, damit eine Reaktion ausgelöst wird.

Ein Taster könnte z.B. alle anderen Taster für 5 Sekunden freischalten. Wobei das nicht für alle Situationen ausreichend ist, zum Beispiel wenn ein größerer Gegenstand alle Taster betätigt. Da hilft dann nur eine komplexere Sequenz von Tastendrücken und das konsequente Abbrechen, sobald dabei eine Taste falsch oder zu lange gedrückt wurde – ähnlich wie man das von der Tastensperre beim Mobiltelefon kennt.

KAPITEL 10

Musik-Controller mit Arduino

In diesem Kapitel:

- Musik steuern mit dem Arduino
- Das MIDI-Protokoll
- Die MidiDuino-Bibliothek
- Ein MIDI-Zauberstab
- MIDI-Input

Musik steuern mit dem Arduino

Die zwei letzten Kapitel dieses Buches befassen sich mit elektronischer Musik in ihren unterschiedlichen Ausprägungen. Die elektronische Musik mit ihrer relativ kurzen Geschichte ist einer der Pionierbereiche des 20. Jahrhunderts und hat eine lange Tradition selbstgebauter Instrumente.

Die ersten Einsätze von Elektronik im musikalischen Bereich waren imposante Maschinen wie das Telharmonium von Thaddeus Cahill, das 1897 entwickelt wurde.

◀ Abbildung 10-1
Telharmonium

Diese Mischung aus mechanischen Klangerzeugern und elektronischer Steuerung, die man *elektromechanische Klangerzeugung* nennt,

lässt sich bis zum heutigen Tage verfolgen, mit faszinierenden Geräten wie z.B. den Instrumenten von Godfried-Willem Raes von der Logos Foundation aus Belgien, der herkömmliche Instrumente modifiziert und automatisiert, sodass sie sich elektronisch steuern lassen.

Allgemein bekanntere elektronische Musikinstrumente sind komplett elektronische Klangerzeuger, die man Synthesizer nennt. Diese benutzen elektronische Schaltungen, um verschiedenste Klänge zu erzeugen und zu verändern. Eins der ersten dieser elektronischen Instrumente war das Theremin, 1919 vom russischen Physikprofessor Leon Theremin erfunden. Die erste Theremin-Aufführung fand 1920 in St. Petersburg am Physikalisch-Technischen Institut statt. Bekannt ist das Theremin vor allem aus der Filmmusik, wo es seit den ersten Horror- und Science-Fiction-Filmen für Soundeffekte eingesetzt wurde, unter anderem in Danny Elfmans Soundtrack zum Film »Charlie und die Schokoladenfabrik« aus dem Jahr 2005.

Das Theremin war das erste ganz ohne Berührung spielbare Instrument: Über zwei Antennen lassen sich Lautstärke und Tonhöhe getrennt steuern. In diesem und dem nächsten Kapitel werden wir mit einem Arduino und ein paar Drähten ein Theremin bauen.

Ein Thereminbausatz war das erste Produkt, das Bob Moog in den 1950er Jahren vertrieb. Bob Moog wurde später bekannt als Erfinder und Entwickler eines der ersten weit verbreiteten Synthesizer. Der von Bob Moog entwickelte Synthesizer war ein modularer Synthesizer: Er bestand aus vielen kleinen einzelnen elektronischen Modulen, die jeweils eine bestimmte Aufgabe erfüllten. So konnte eins dieser Module zum Beispiel einen Ton in einer bestimmten Höhe erzeugen, ein zweites Klangsignale aus verschiedenen Quellen zusammenmischen und ein drittes Steuerparameter erzeugen, die einem bestimmten Hüllkurvenverlauf entsprachen.

Modulare Synthesizer sind heutzutage immer noch sehr beliebt und weit verbreitet, gerade weil sie ihrem Benutzer so viel kreative und technische Freiheit lassen. Im Vergleich zu digitalen Synthesizern, die einen Prozessor und digitale Algorithmen verwenden, um Klänge zu erzeugen, werden analoge Synthesizer oft als »wärmer« empfunden, weil die vielen Ungenauigkeiten in der elektronischen Schaltung und den einzelnen Komponenten einen sich immer wieder leicht verändernden Klang erzeugen, der deswegen auch »menschlicher« klingt. Es gibt mittlerweile auch viele Module für modulare Synthesi-

zer, die digital ihre Klänge erzeugen. Viele dieser Module (analoge und digitale) lassen sich mit herkömmlichen elektronischen Komponenten in einer relativ kurzen Zeit bauen, und es gibt viele Seiten im Internet, die Bausätze und Anleitungen anbieten. Sehr empfehlenswert ist Ray Wilsons Seite *http://www.musicfromouterspace.com*, die z.B. einen kompletten Synthesizer zum Selbstbauen anbietet.

Dieses Konzept von einzelnen kleinen Modulen, die man zusammenstecken kann, hat eine gewisse Ähnlichkeit mit dem Arduino und seinen verschiedenen Erweiterungsmodulen, und so ist es nicht überraschend, dass die Arduino-Plattform benutzt wird, um analoge modulare Synthesizer zu steuern und zu ergänzen. Hier sei z.B. auf das ModularDuino-Projekt hingewiesen, das den Arduino als Klangerzeugermodul in einem modularen Synthesizer einsetzt.

Lange Zeit war die elektronische Musik vor allem ein Bereich für moderne klassische Komponisten. Diese Pioniere entwickelten durch ihre kompositorischen Ansätze und mit Einsatz von Oszillatoren, Radios und Bandgeräten viele der Techniken, die später auch in herkömmlichen Synthesizern und elektronischen Musikgeräten zum Einsatz gekommen sind. Parallel zur akademischen Musik wurden in den Studios von Radiosendern, z.B. beim BBC Radiophonic Workshop, neue Klangkonzepte entwickelt und umgesetzt.

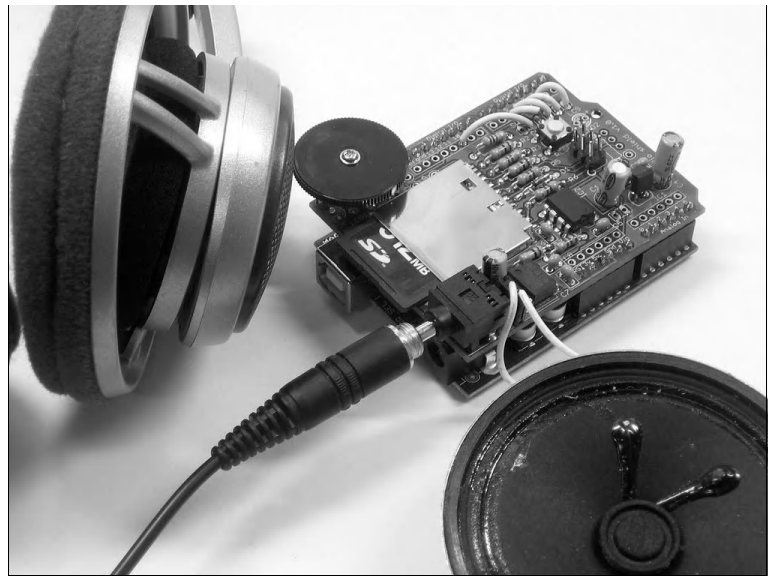
Schnell wurden Module und Geräte gebaut, die es möglich machten, musikalische Kompositionen elektronisch zu speichern, abzuspielen und zu modifizieren: Diese Geräte wurden *Sequencer* genannt. Auf den ersten, analogen Sequencern konnte man mithilfe einer Reihe von Potentiometern verschiedene Steuerwerte einstellen und in einer Sequenz wiederholen. Diese Steuerwerte konnten z.B. Tonhöhen sein oder Lautstärken, aber auch Filtereinstellungen und andere Klangparameter. Dadurch ließen sich relativ einfach kurze melodische Fragmente eingeben und verschiedene rhythmische Muster einstellen, die immer wiederholt wurden: Es war nicht mehr notwendig, Noten einzugeben, sondern man konnte komplette Stücke elektronisch erstellen, und zwar so leicht wie auf einem Klavier.

Parallel zu den Entwicklungen in der analogen Elektronik wurden die Computer immer schneller und kleiner, und viele Komponisten und Musiker begannen, sie einzusetzen, um Musik zu schreiben und zu erzeugen. Das erste Musikprogramm wurde schon 1951 geschrieben, konnte allerdings nur einfache Melodien spielen. Angesichts des Computers, auf dem dieses Programm lief, mit sei-

nen Röhren und seinem Quecksilberspeicher, war das schon eine beachtliche Leistung. Ähnliche Melodien können wir auch mit dem Arduino generieren, wie es in Kapitel 11 gezeigt wird. Mit wachsendem Speicher und fortschreitender Miniaturisierung war es jedoch bald möglich, Computer zu benutzen, um komplette musikalische Werke zu speichern und zu bearbeiten, mit sogenannten Sequencerprogrammen (die im Vergleich zu analogen Sequencern digital ihre Parameter speichern und darum viel umfangreichere Editiermöglichkeiten boten).

Mit den größeren Speichern wurde es Computern auch möglich, digital Musik aufzunehmen und wieder abzuspielen. Das digitale Speichern von Klängen, das Sampling, war geboren. Diese Klangerzeugungsmöglichkeit veränderte in den 1980er Jahren mit Hip-Hop das Gesicht der populären Musik. Auch auf dem Arduino ist es dank dem WaveShield von LadyAda möglich, Klangsamples abzuspielen.

Abbildung 10-2 ▶
Arduino WaveShield



Mittlerweile sind Heimcomputer so leistungsstark, dass man auf ihnen problemlos Dutzende, wenn nicht Hunderte von digitalen Synthesizern laufen lassen kann. Diese Synthesizer nennt man auch virtuelle Instrumente (*virtual instruments*), weil sie keine eigenständigen Geräte sind. Im Endeffekt entsprechen sie aber von Aufbau und Klangeigenschaften her eigenständigen Hardwaresynthesizern. Es gibt mittlerweile eine ganze Reihe an Musikprogrammen, die die Möglichkeiten des Computers, Programme auszuführen und zu

speichern, voll ausnutzen. Diese Programme bieten dem Benutzer eine Schnittstelle, mit der er selbst seine eigenen Synthesizer und Sequencer programmieren kann. Manche dieser Programme sind eigentlich Softwareentwicklungsumgebungen, in denen Programme auf herkömmliche Weise als Textdatei geschrieben und dann ausgeführt werden (ein bisschen wie in der Arduino-Umgebung, nur dass anstelle von Anweisungen, die die Hardware steuern, hier Anweisungen zur Verfügung stehen, die Musiknoten spielen oder Klangparameter verändern).

Um diese ganzen Geräte und Computer miteinander zu verbinden (sodass ein Sequencer ohne Probleme einen beliebigen Synthesizer ansteuern kann), wurde der MIDI-Standard im Jahre 1981 entwickelt. MIDI ist ein Kommunikationsprotokoll, das den Austausch symbolischer musikalischer Information zwischen verschiedenen elektronischen Instrumenten ermöglicht. »Symbolisch« heißt in diesem Kontext, dass keine Audiodaten ausgetauscht werden, sondern kurze Nachrichten, die eine symbolische Bedeutung tragen. So können von einem MIDI-Keyboard aus Noten und zusätzliche Parameter, wie Tastenanschlagsgeschwindigkeit oder ob das Pedal gedrückt ist, an einen Synthesizer übermittelt werden. Auch ein Sequencer kann MIDI-Informationen speichern und abspielen und so z.B. für Hintergrundharmonien oder komplette Lieder eingesetzt werden. Das MIDI-Protokoll unterstützt auch die Synchronisierung von unterschiedlichen Geräten, sodass mehrere Sequencer gleichzeitig eingesetzt werden können, ohne dass sie den Takt verlieren und die so erzeugte Musik wie eine Blaskapelle nach dem Dorffest klingt. MIDI hat sich schnell durchgesetzt und ist auch heutzutage, 25 Jahre später, immer noch die erste Wahl, um elektronische Musikinstrumente zu verbinden.

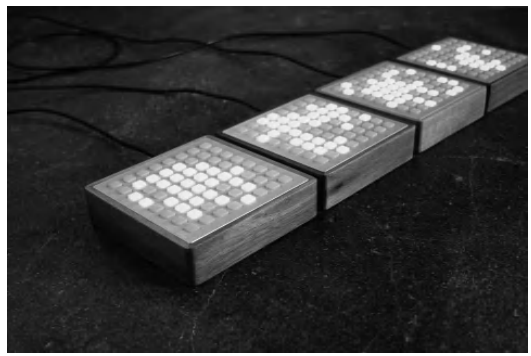
Wegen der vielen Parameter, die man auf Synthesizern einstellen kann, wurden Geräte entwickelt, die nur zur Steuerung von anderen MIDI-Geräten verwendet werden und oft keine Gemeinsamkeiten mehr mit herkömmlichen Instrumenten haben. Diese Geräte nennt man MIDI-Controller. Ein typisches Beispiel für einen solchen MIDI-Kontroller ist ein Gerät, das dem Musiker Drehknöpfe zur Verfügung stellt. Jeder dieser Drehknöpfe ist mit einem bestimmten Parameter auf dem gesteuerten Synthesizer verknüpft. In gewissem Sinne ist auch ein MIDI-Keyboard ein MIDI-Controller, und so gibt es auch Controller, die eine Gitarre nachbauen, oder Controller, die wie ein Saxophon funktionieren (wie der EWI-4000), nur eben keinen Klang erzeugen, sondern MIDI-Daten. Wenn man aber das Konzept ein bisschen weiter verfolgt,

lassen sich beliebige Sensoren zu einem MIDI-Controller umbauen, und genau das ist einer der Bereiche, in dem der Arduino glänzend zum Einsatz kommt.

Besonders interessant sind Geräte, die sich von dieser starren Auftrennung in Klangerzeuger, Sequencer und Controller verabschieden und all diese Aspekte unter einem übergreifenden Konzept kombinieren. Ein besonders interessantes und mittlerweile weit verbreitetes Gerät ist der Monome von der Firma selben Namens (*http://monome.org/*). Das ist ein Musikcontroller, der aus 64 beleuchteten Tastern besteht (es gibt auch Versionen mit 128 und 256 Tasten). Diese 64 Tasten sind in einer 8-mal-8-Matrix angeordnet. Unter jeder Taste befindet sich eine LED, die durchscheint. Der Monome wird an einen Computer angeschlossen und steuert dort die Software MAX/MSP, wofür es spezielle Patches gibt. Das Gerät an sich sendet nur gedrückte Tasten über eine USB-Schnittstelle und empfängt vom Computer Kommandos, um die LEDs an- und auszuschalten. Durch diese recht schlichte Oberfläche sind allerdings innovative und intuitive Steueroberflächen möglich. Der Künstler Daedalus, der an der Entwicklung des Monome beteiligt war, benutzt die Oberfläche, um einzelne musikalische Loops (kurze, sich wiederholende Audiosamples) zu starten und zu verändern.

Der Monome-Controller ist komplett quelloffen: Monome stellt auf seiner Website die Schaltpläne und den Quellcode der Software zur Verfügung. Dadurch ist um den Monome-Controller, der nur in kleinen Stückzahlen (und auch als Bausatz) verkauft wird, eine große Gemeinschaft interessierter Musiker und Programmierer entstanden, die das Produkt weiterentwickeln. Aus dieser Gemeinschaft heraus wurde ein Selbstbauklon des Controllers entwickelt, der auf Arduino und 4-mal-4-Button-Pads von Sparkfun basiert.

Abbildung 10-3 ▶
Monome-Controller



In diesem Kapitel wird das MIDI-Protokoll genauer vorgestellt, und es wird gezeigt, wie ein Arduino benutzt werden kann, um MIDI-Daten zu senden und zu empfangen. Damit ist es möglich, herkömmliche MIDI-Controller zu bauen (also Controller, die hauptsächlich Taster, Schieberegler und Drehknöpfe verwenden), die genau an die Wünsche des Musikers angepasst sind (siehe Kapitel 7). Es ist aber auch möglich, alle Sensoren, die wir bis jetzt an den Arduino angeschlossen haben, zu benutzen, um Synthesizer zu steuern oder Musik zu erzeugen. Letztendlich ist es auch möglich, auf dem Arduino MIDI-Daten zu empfangen und weiterzuverarbeiten, um z.B. Klänge zu erzeugen, indem in der realen Welt Aktoren angesteuert werden.

Das MIDI-Protokoll

Das MIDI-Protokoll ist ein asynchrones serielles Protokoll (siehe Kapitel 5), das getrennte Kabel für die Sende- und die Empfangsrichtung verwendet. MIDI-Geräte verfügen also meistens über eine Eingangsbuchse (MIDI IN) und eine Ausgangsbuchse (MIDI OUT). MIDI-Kabel verwenden einen DIN-5-Stecker, und die dazugehörigen Buchsen lassen sich problemlos auf ein Steckboard stecken. MIDI-Geräte bieten oft einen sogenannten THRU-Anschluss an, der Daten, die am MIDI-IN anliegen, spiegelt. Dadurch ist es möglich, mehrere Synthesizer hintereinander anzuschließen, um sie von demselben Keyboard oder Sequencer aus anzusprechen. Frühere Computer hatten entweder von Haus aus eingebaute MIDI-Schnittstellen (wie der Atari 1024 ST, der bis ins 21. Jahrhundert hinein eine beliebte Computermusikplattform war) oder boten diese an der Soundkarte an (über den Joystick-Port). Mittlerweile ist es allerdings üblich, einen USB- oder Firewire-MIDI-Adapter zu verwenden, um MIDI-Geräte an Heimcomputer anzuschließen. Das MIDI-Protokoll kann man auch komplett ohne physikalische Schnittstelle verwenden, wenn man unterschiedliche Musikprogramme auf einem Rechner verwendet. Diese benutzen dann virtuelle MIDI-Ports, die nur intern im Computer existieren.

In diesem Kapitel werden Projekte vorgestellt, die MIDI-Daten entweder senden oder empfangen. Das heißt, dass diese Projekte ohne weitere MIDI-Hardware oder MIDI-Software nicht viele Auswirkungen haben werden. Falls ein Hardware-Synthesizer oder ein Hardware-Sequencer zur Verfügung steht, lässt sich dieser über herkömmliche MIDI-Kabel und die unten beschriebenen Schaltungen direkt an den Arduino anschließen. Es ist allerdings auch mög-

lich, mit dem Computer MIDI-Daten zu erzeugen und direkt über die USB-Schnittstelle an den Arduino zu übermitteln. Dazu muss auf dem Betriebssystem der Fluss, der über die serielle Schnittstelle gesendet wird, als MIDI interpretiert und an den MIDI-Stack im Computer übergeben werden. Dazu kann unter Mac OS X die Ardrumo-Software benutzt werden, die es frei herunterzuladen gibt (*http://code.google.com/p/ardrumo/*). Unter Windows kann die Software s2midi benutzt werden, die es ebenfalls frei unter *http://www.memeteam.net/2007/s2midi/* gibt. Für beide Plattformen können Sie auch das Programm Serial-Midi benutzen, das Sie unter *http://www.spikenzielabs.com/SpikenzieLabs/Serial_MIDI.html* finden.

MIDI-Nachrichten

Das MIDI-Protokoll ist ein binäres Protokoll (Nachrichten werden als numerische Werte übertragen, nicht als Zeichenketten), das auf 8-Bit-Werte aufbaut. Allerdings wird in jedem übertragenen Wert das oberste Bit als Statusbit benutzt: Ist es gesetzt, handelt es sich nicht um einen numerischen Wert, sondern signalisiert, dass es sich um einen Befehl handelt und nicht um normale Daten. Dieses Statusbit vereinfacht die serielle Kommunikation sehr (siehe Kapitel 6, um mehr über serielle Protokolle zu erfahren): Es reicht zu erkennen, ob das erste Bit gesetzt ist, um zu wissen, ob ein empfangenes Byte ein Befehl oder ein numerischer Wert ist. Dadurch kann man auch problemlos MIDI-Kabel ein- und ausstecken, ohne dass sich Geräte neu synchronisieren müssen. Werden am Anfang unbekannte Daten empfangen, reicht es, bis zum nächsten Byte mit gesetztem Statusbit zu warten.

Bei Befehlbytes codieren die untersten 4 Bits den MIDI-Kanal. Dieser Kanal ist eine virtuelle Aufteilung des MIDI-Kabels in 16 getrennte Kabel. Jedes angeschlossene Gerät wird auf seinen eigenen Kanal konfiguriert und reagiert nur auf Nachrichten, die in den unteren 4 Bits des Befehlbytes diesen Kanal codiert haben. Dadurch ist es möglich, Noten nur an einen bestimmten Synthesizer zu schicken, auch wenn mehrere über eine MIDI-THRU-Kette angeschlossen sind (es kann auch sein, dass unterschiedliche Einheiten eines einzelnen Synthesizers auf verschiedenen Kanälen reagieren). Oft sind Synthesizer jedoch auf den Omnikanal eingestellt; reagiert ein Synthesizer also auf die falschen Nachrichten oder gar nicht, lohnt es sich nachzuprüfen, auf welchem Kanal MIDI-Nachrichten geschickt werden und welche Kanaleinstellungen auf dem Synthesizer aktiv sind.

Wenn ein Computer benutzt wird, um MIDI-Daten zu empfangen und zu senden (heutzutage meistens über einen USB-Adapter oder, falls der Computer über eine professionelle Soundkarte verfügt, über die Soundkarte), ist es oft nützlich, einen sogenannten *MIDI-Sniffer* zu benutzen, um solchen Problemen auf die Schliche zu kommen. Unter Windows ist die Software MIDI-OX zu empfehlen, die es auch erlaubt, virtuelle MIDI-Geräte zu erstellen, um verschiedene MIDI-Programme untereinander zu verknüpfen. MIDI-OX kann unter *http://www.midiox.com/* heruntergeladen werden. Unter Mac OS X ist die Software Midi Monitor zu empfehlen, die unter *http://www.snoize.com/MIDIMonitor/* heruntergeladen werden kann. Diese Sniffer-Programme zeichnen MIDI-Kommunikation auf, dekodieren die einzelnen Befehle, die aufgenommen wurden, und zeigen sie in lesbarer Form an. Besonders nützlich ist das beim Debuggen von MIDI-Programmen, die auf dem Arduino laufen, weil in solchen Fällen die normale serielle Schnittstelle nicht mehr funktioniert, da sie als MIDI-Schnittstelle verwendet wird und auf eine Geschwindigkeit eingestellt ist, die nicht standardmäßig erkannt wird.

Wegen der Kennzeichnung von Befehlbytes über das obere Statusbit können nur numerische Werte übertragen werden, die 7 Bits lang sind, also Werte von 0 bis 127 (deswegen kommt es in Computersoftware und in Synthesizern auch oft vor, dass Parameterwerte nur von 0 bis 127 gehen). Es gibt drei Gruppen von MIDI-Nachrichten: normale, Echtzeit- und exklusive MIDI-Nachrichten.

Die erste Gruppe umfasst die am häufigsten gebrauchten MIDI-Nachrichten. Nachrichten mit Befehlbyte 0x80 und 0x90 werden benutzt, um Noteninformationen zu übertragen. Wird auf einem MIDI-Keyboard eine Taste angeschlagen, wird eine *Note On*-Nachricht erzeugt, die die Tonhöhe der angeschlagenen Taste enthält sowie die Geschwindigkeit, mit der diese Taste angeschlagen wurde (falls das Keyboard über Anschlagsdynamik verfügt). Wird die Taste wieder losgelassen, wird eine *Note Off*-Nachricht geschickt. (Mittlerweile benutzen die meisten Keyboards allerdings eine *Note On*-Nachricht mit einer Anschlagsgeschwindigkeit von 0).

Zwischen diesen beiden Nachrichten können beliebige weitere Nachrichten über die MIDI-Verbindung übertragen werden. Wie man sich leicht vorstellen kann, kann es bei intensiver Datenübertragung (wenn z. B. viele Daten aus einem Sequencer gesendet werden) zu Verzögerungen kommen, da die MIDI-Verbindung nur eine begrenzte Geschwindigkeit hat. Es ist daher unter Umständen

wichtig, separate MIDI-Verbindungen zu verwenden, wenn viele Daten an mehrere Synthesizer übertragen werden sollen. Die *Note On*- und *Note Off*-Nachrichten sind wahrscheinlich die am häufigsten vorkommenden MIDI-Nachrichten.

Eine weitere MIDI-Nachricht ist die *Controller Change*-Nachricht, die benutzt wird, um beliebige Parameter zu steuern. Sie wird zum Beispiel verwendet, um die Lautstärke, die Filteröffnung und weitere Syntheseparameter zu steuern. Welche Controllerwerte an welche Parameter auf einem MIDI-Gerät gebunden sind, lässt sich in den jeweiligen Handbüchern nachschlagen. Die meisten MIDI-Controller, die nicht nur Noten senden, verwenden *Controller Change*-Nachrichten, um ihre Informationen zu übermitteln. Bei einem Blasinstrument-Controller werden z.B. der Luftdruck und die gedrückten Ventile als *Controller Change*-Nachrichten übermittelt. Bei Controllern, die Potentiometer oder Encoder verwenden, lässt sich meistens die Controllernummer jedes einzelnen Drehknopfes einzeln zuweisen, sodass sich der Benutzer eine eigene Kontrolloberfläche für sein gesteuertes MIDI-Gerät zurechtlegen kann. Es können wegen der Beschränkungen des MIDI-Protokolls nur bis zu 128 Parameter gesteuert werden. Jeder dieser Parameter kann auf einen Wert von 0 bis 127 gesetzt werden.

Weitere MIDI-Nachrichten, die eher instrumentenspezifische Aufgaben übernehmen, sind die *Program Change*-Nachricht, mit der verschiedene »Programme« (also Einstellungen) auf Synthesizern aufgerufen werden, sowie die *Key Pressure*- und *Channel Pressure*-Nachrichten, die bei Keyboards den Tastendruck übermitteln. Diese Nachrichten werden wir in diesem Kapitel nicht verwenden.

Echtzeit-MIDI-Nachrichten sind eine spezielle Klasse von MIDI-Nachrichten, die meistens nur ein Byte lang sind (mit gesetzten Statusbit) und hauptsächlich zur Temposynchronisation von MIDI-Geräten eingesetzt werden. Echtzeit-MIDI-Nachrichten, die nur ein Byte lang sind, können jede beliebige Nachricht unterbrechen und sind meistens sehr zeitkritisch. Wenn zwei MIDI-Geräte untereinander synchronisiert werden sollen, übernimmt eins dieser Geräte die Rolle des *Timing Master* (also »Zeitchef«), während das andere Gerät die Rolle des *Timing Slave* übernimmt. Der Timing Master bestimmt das generelle Tempo und kann auch anhand der *Start*-, *Pause*- und *Stop*-Nachrichten das Abspielen steuern. Der Timing Master schickt regelmäßig kurze Echtzeitnachrichten (die *Zeit-Ticks*) an den Timing Slave, und der Slave synchronisiert sich auf diese regelmäßigen Nachrichten. Die MidiDuino-Library, die wir in

diesem Kapitel verwenden, unterstützt sowohl die Slave- als auch die Master-Seite der Synchronisation. Allerdings sind für die korrekte Slave-Synchronisation einige Eingriffe in das Arduino-System notwendig, weswegen wir uns in diesem Kapitel auf die Master-Rolle beschränken.

Systemexklusive Nachrichten schließlich sind spezielle Nachrichten, die benutzt werden, um beliebige Daten über MIDI auszutauschen. Binäre Daten können als *Sysex-Nachrichten* übermittelt werden. Eine Sysex-Nachricht fängt mit der speziellen Nachricht 0xF0 an, die gefolgt wird von einer beliebigen Anzahl weiterer binärer Datenbytes (allerdings muss bei diesen Bytes das oberste Bit auf 0 gesetzt sein, weil sie ja sonst als Befehlbyte erkannt werden).

Das Erkennen von einkommenden MIDI-Nachrichten ist eine recht komplizierte Angelegenheit, die von der MidiDuino-Library übernommen wird. Die Library kann alle Tricks von MIDI bearbeiten und so alle MIDI-Nachrichten richtig erkennen.

MIDI an den Arduino anbinden

Elektrisch betrachtet, ist die serielle Schnittstelle ein Stromkreis. Das heißt, dass Bits nicht als Spannungen übertragen werden, wie es sonst üblicherweise vorkommt, sondern dass Stromfluss eine »1« signalisiert und kein Strom eine »0«. Um Stromflüsse wieder in Spannungen zu konvertieren, damit der Arduino die übertragenen Bits korrekt interpretieren kann, wird ein Optokoppler eingesetzt. Ein Optokoppler ist ein Chip, der intern eine LED und einen Fotowiderstand enthält. Der Eingangsstrom bringt die LED zum Leuchten, und der empfangende Fotowiderstand schaltet einen Transistor, sodass auf der Ausgangsseite eine Spannung geschaltet wird. So lassen sich Informationen ohne elektrische Verbindung übertragen (die Übertragung geschieht optisch, daher der Name Optokoppler). Dadurch lassen sich Grundschleifen vermeiden, die dann vorkommen, wenn mehrere Geräte, die physikalisch voneinander entfernt sind und deswegen oft unterschiedliche Massenpotentiale haben, über ein Kabel verbunden werden. Dadurch kommt es manchmal zu einem Brummen, wenn diese an einen Verstärker angeschlossen sind, was natürlich in einem musikalischen Kontext nicht erwünscht ist. Die Schaltung, um von einem Arduino aus MIDI zu empfangen, ist allerdings sehr einfach und lässt sich mit einem Optokoppler (hier hat sich der 6N138 bewährt), zwei 220-Ohm-Widerständen und einer Diode bauen.

Abbildung 10-4 ▶
Schaltbild MIDI-Input

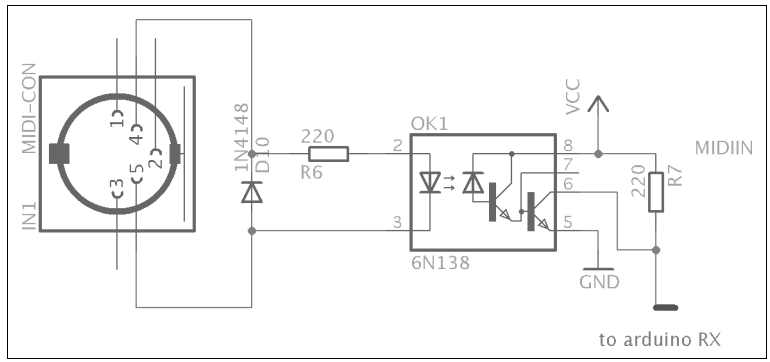
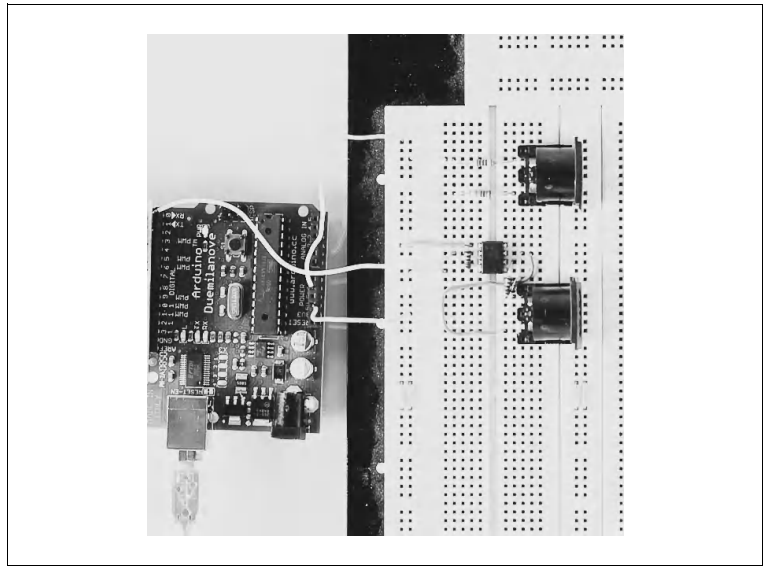
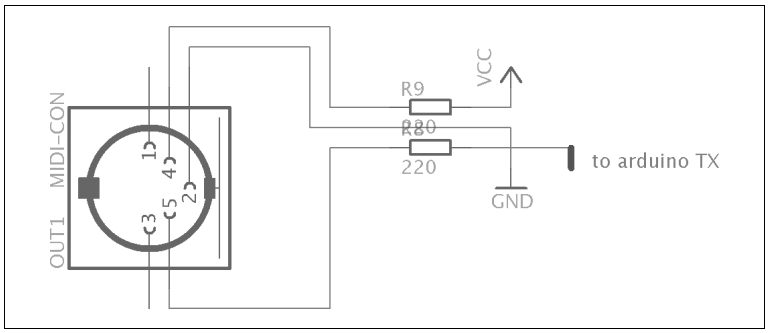


Abbildung 10-5 ▶
Schaltung MIDI-Input und MIDI-Output



Um MIDI-Daten von einem Arduino aus zu senden, ist es nur notwendig, den seriellen Ausgangspin TX (Pin 2 auf dem Arduino-Board) über einen 220-Ohm-Widerstand an Pin 5 der MIDI-Buchse zu führen, und Pin 2 der MIDI-Buchse über einen weiteren 220-Ohm-Widerstand an 5V zu verbinden.

Da die MIDI-Schnittstelle direkt an die serielle Schnittstelle des Arduino angeschlossen wird, kommt es auf dem RX-Pins zu einer »elektrischen Kollision«. Deswegen ist es notwendig, vor dem Hochladen eines neuen Sketches den RX-Pin freizustellen, weil sonst von der USB-Seite keine Daten empfangen werden können. Kommt eine Timeout-Fehlermeldung beim Hochladen, bedeutet das, dass der RX-Pin noch an den Optokoppler angeschlossen ist.



◀ **Abbildung 10-6**
Schaltbild MIDI-Output

Die MidiDuino-Bibliothek

In diesem Kapitel wird die MidiDuino-Bibliothek verwendet, die Sie unter *http://ruinwesen.com/mididuino* herunterladen können. Diese Bibliothek wurde von einem der Autoren dieses Buches geschrieben und wird ständig weiterentwickelt. Beim Schreiben dieses Buches wird die erste Version 0.1 der Bibliothek verwendet. In der ZIP-Datei der Bibliothek sind verschiedene kleine Bibliotheken zu finden, die alle in den Ordner *hardware/libraries* der Arduino-Installation kopiert werden müssen.

Die MidiDuino-Bibliothek besteht aus zwei großen Teilen. Der erste ist der `MidiUart`-Teil, der sich um die serielle Schnittstelle kümmert. Dieser Teil wird auch verwendet, um MIDI-Nachrichten zu senden. Der zweite Teil ist der eigentliche MIDI-Stack, der einkommende Nachrichten von der seriellen Schnittstelle liest und analysiert und daraus einzelne Nachrichten erkennt und an verschiedene Callback-Funktionen weiterreicht. Callback-Funktionen sind Funktionen aus dem Sketch, die als Argument an den MIDI-Stack übergeben und automatisch aufgerufen werden.

In diesem ersten Sketch verwenden wir den `MidiUart`-Teil, um in regelmäßigen Abständen Noten über MIDI zu schicken. Statt die serielle Schnittstelle mit `Serial.init()` zu initialisieren, wird diese mit dem Aufruf `MidiUart.init()` initialisiert. Wenn MIDI mit den früher erwähnten Seriell-nach-MIDI-Konvertierungsprogrammen S2midi oder Ardrumo verwendet wird, muss als zusätzliches Argument an `MidiUart.init()` noch die serielle Geschwindigkeit angegeben werden, ansonsten wird die standardmäßige MIDI-Geschwindigkeit von 31.250 Bps verwendet. Dadurch, dass MIDI-Daten binär direkt über die serielle Schnittstelle gesendet werden, kann die Serial-Bibliothek nicht mehr verwendet werden, um

Daten in die serielle Konsole auszugeben. Stattdessen kann einer der MIDI-Sniffer verwendet werden, um bestimmte Debug-Nachrichten zum Beispiel als Noten oder Kontrollnachrichten zu übertragen.

Wenn die USB-Schnittstelle verwendet wird, um direkt MIDI-Daten an den Computer zu übermitteln (mithilfe von Ardrumo oder S2Midi), muss die serielle Schnittstelle allerdings auf eine der Standardgeschwindigkeiten umgestellt werden. Diese Geschwindigkeit (z.B. 9.600 Bps oder 112.500 Bps) kann man einfach als Argument an `MidiUart.init()` übergeben.

Nachdem die serielle Schnittstelle mit `MidiUart.init()` initialisiert wurde, kann sie verwendet werden, um einzelne MIDI-Nachrichten zu übertragen. In diesem ersten Sketch werden Noten mit den Funktionen `MidiUart.sendNoteOn()` und `MidiUart.sendNoteOff()` gesendet. Diese Funktionen kriegen im Fall der *Note On*-Nachricht als Parameter einen MIDI-Kanal, eine Notenhöhe und eine Anschlagsgeschwindigkeit. Der Parameter MIDI-Kanal ist optional: Wenn er nicht angegeben wird, wird der Defaultwert `MidiUart.currentChannel` verwendet. Um den Default-Kanal zu ändern, reicht ein einfaches Zuweisen an die Variable `MidiUart.currentChannel`.

```
// Einbinden der MidiDuino-Library
#include <MidiUart.h>
#include <Midi.h>
MidiClass Midi;

void setup() {
  // Initialisieren der MIDI-Schnittstelle
  MidiUart.init();
  // Wenn Ardrumo oder S2Midi verwendet werden, wird stattdessen
  // MidiUart.init(9600);
  // verwendet
}

void loop() {
  // Zuerst Senden einer Note On-Nachricht
  MidiUart.sendNoteOn(100, 100);
  delay(200);
  // Dann senden einer Note Off-Nachricht
  MidiUart.sendNoteOff(100);
  delay(2000);
}
```

Dieses Programm lässt sich leicht um Kontrollnachrichten ergänzen, die mit der `MidiUart.sendCC()`-Funktion gesendet werden.

Diese Funktion nimmt als Parameter einen optionalen MIDI-Kanal, die Parameternummer und den Parameterwert. So lässt sich die `loop()`-Funktion aus dem vorigen Sketch um Kontrollnachrichten erweitern.

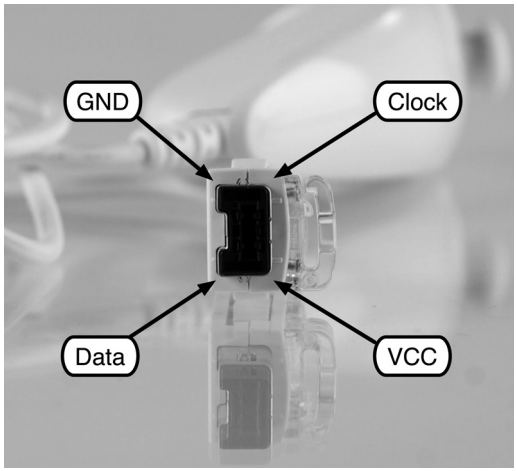
```
void loop() {
  // Zuerst Senden einer Note On-Nachricht
  MidiUart.sendNoteOn(100, 100);
  MidiUart.sendCC(5, 100);
  delay(200);
  // Dann Senden einer Note Off-Nachricht
  MidiUart.sendNoteOff(100);
  MidiUart.sendCC(5, 10);
  delay(2000);
}
```

Ein MIDI-Zauberstab

In diesem Sketch verwenden wir einen Beschleunigungssensor, um MIDI-Daten zu erzeugen. Das Prinzip des Beschleunigungssensors wurde schon in Kapitel 7 erläutert. In diesem Sketch verwenden wir den Nunchuck-Kontroller einer Wii-Konsole von Nintendo, weil das die billigste und einfachste Methode ist, an einen Beschleunigungssensor zu kommen. Für einen Preis von ungefähr 20 EUR bekommt man einen Dreiachsen-Beschleunigungssensor, einen Joystick und zwei Taster. Der Controller versendet diese Daten über einen digitalen synchronen seriellen Bus, den man *i2c* nennt. Dieser Bus ist weit verbreitet und wird von vielen verschiedenen Schaltungen und Chips verwendet. Die Wire-Bibliothek, die bei Arduino mitgeliefert wird, übernimmt alle Kommunikationsaufgaben, sodass der Anschluss des Nunchuck an das Arduino-Board leicht gelingt.

Um den Nunchuck zu verbinden, kann man Drähte in die einzelnen Pins des Anschlusskonnektors stecken und diese an den Arduino anschließen. Von vorn gesehen, ist der linke obere Pin die Clock-Leitung, der rechte obere Pin GND, der linke untere VCC und der rechte untere die Datenleitung. Die Wire-Bibliothek verwendet die analogen Pins 4 und 5 als Kommunikationsschnittstelle, und in diesem Sketch werden die analogen Pins 2 und 3 als Stromversorgungspins verwendet. GND wird an den analogen Pin 2 angeschlossen, VCC an den analogen Pin 3, Data an den analogen Pin 4 und Clock an den analogen Pin 5. In der folgenden Abbildung wird der Anschluss gezeigt.

Abbildung 10-7 ▶
Nunchuck-Pinbelegung



Der Code zum Einlesen der Nunchuck-Daten ist relativ lang und wird hier deshalb nicht komplett abgebildet. Als Erstes müssen in der Setup-Routine sowohl der MIDI-Stack als auch die Wire-Verbindung zum Nunchuck initialisiert werden. An den Nunchuck wird dann auch gleich ein Initialisierungsbefehl geschickt.

```
#define pwrpin PORTC3 // analoger pin 3
#define gndpin PORTC2 // analoger pin 2

void nunchuckInit() {
  // Initialisierung der Stromversorgungspins
  DDRC |= _BV(pwrpin) | _BV(gndpin);
  PORTC &=~ _BV(gndpin);
  PORTC |= _BV(pwrpin);
  delay(100);

  // Initialisieren der Wire-Bibliothek und des Nunchuck
  Wire.begin();
  Wire.beginTransmission(0x52);
  Wire.send(0x40);
  Wire.send(0x00);
  Wire.endTransmission();
}

void setup () {
  MidiUart.init();
  nunchuckInit();
}
```

Die Nunchuck-Daten werden in der Funktion `nunchuckReadData()` eingelesen und in `nunchuckParseData()` analysiert und decodiert.

```
int nunchuckReadData(byte outbuf[]) {
  nunchuckSendZero();
  delay (30);

  Wire.requestFrom (0x52, 6); // request data from nunchuck
  int cnt = 0;
  while (Wire.available ()) {
    outbuf[cnt] = nunchukDecodeByte(Wire.receive ());
    cnt++;
  }

  return cnt;
}

byte nunchukDecodeByte(byte x) {
  x = (x ^ 0x17) + 0x17;
  return x;
}

void nunchuckSendZero() {
  Wire.beginTransmission(0x52); // transmit to device 0x52
  Wire.send(0x00); // sends one byte
  Wire.endTransmission(); // stop transmitting
}

int joystick[2] = { 0 };
int oldJoystick[2] = { 0 };
byte joystickCC[2] = { 108, 109 };

int accel[3] = { 0 };
int oldAccel[3] = { 0 };
byte accelCC[3] = { 4, 5, 6 };

int button[2] = { 0 };
int oldButton[2] = { 0 };
byte buttonNote[2] = { 60, 62 };

void nunchuckParseData(byte outbuf[]) {
  joystick[0] = outbuf[0];
  joystick[1] = outbuf[1];
  accel[0] = outbuf[2] << 2;
  accel[1] = outbuf[3] << 2;
  accel[2] = outbuf[4] << 2;

  button[0] = bitRead(outbuf[5], 0);
  button[1] = bitRead(outbuf[5], 1);

  for (int i = 0; i < 3; i++) {
    accel[i] += bitRead(outbuf[5], i + 2) * 2 + bitRead(outbuf[5],
    i + 3);
  }
}
```

Die Ergebnisse des Auslesens werden in den Tabellen *joystick[]*, *accel[]* und *button[]* abgelegt, die anschließend in der Funktion `nunchuckSendMidi()` ähnlich wie beim Code im Miniaturcontroller in MIDI-Nachrichten konvertiert werden.

```
void nunchuckSendMidi() {
  for (int i = 0; i < 2; i++) {
    if (abs(oldJoystick[i] - joystick[i]) >= 2) {
      MidiUart.sendCC(joystickCC[i], map(joystick[i], 0, 255, 0,
      127));
      oldJoystick[i] = joystick[i];
    }
  }

  for (int i = 0; i < 3; i++) {
    if (abs(oldAccel[i] - accel[i]) >= 7) {
      int value = map(accel[i], 300, 800, 0, 127);
      MidiUart.sendCC(accelCC[i], constrain(value, 0, 127));
      oldAccel[i] = accel[i];
    }
  }

  for (int i = 0; i < 2; i++) {
    if (oldButton[i] != button[i]) {
      if (button[i] == 0) {
        MidiUart.sendNoteOn(buttonNote[i], 100);
      } else {
        MidiUart.sendNoteOff(buttonNote[i]);
      }
      oldButton[i] = button[i];
    }
  }
}
```

Mit dieser einfachen Schaltung lässt sich also leicht ein umfassender MIDI-Controller mit Joystick, Tastern und Beschleunigungssensoren bauen.

MIDI-Input

In diesem Sketch wird die MidiDuino-Bibliothek benutzt, um MIDI-Daten einzulesen. Wenn ein Notenanschlag vom Arduino erkannt wird, wird eine LED geschaltet. Dazu müssen die Daten der seriellen Schnittstelle eingelesen und an den eigentlichen MIDI-Stack übergeben werden, der bis jetzt nicht verwendet wurde. Das Erkennen der Noten wird von diesem Teil übernommen. Die Kopplung an den Sketch erfolgt über sogenannte *Callback-Funktionen*. Das sind Funktionen, die mit dem MIDI-Stack registriert aufgerufen werden, sobald eine bestimme Nachricht erkannt wurde.

In diesem Sketch werden zusätzlich zur MIDI-Schaltung, die an die Pins TX und RX angeschlossen ist, noch zwei LEDs an Pin 10 und 12 angeschlossen. Vor dem Programmieren muss die RX-Leitung zum MIDI-Board kurz getrennt werden, damit der Arduino den Sketch auch empfangen kann.

Beim Empfangen von CC-Nachrichten wird die Helligkeit der zweiten LED auf den empfangenen CC-Wert gesetzt. So lassen sich durch das Anschließen von herkömmlichen MIDI-Controllern Parameter von Arduino-Sketchen steuern, ohne dass Potentiometer oder andere Steuerungsoberflächen angeschlossen werden müssen. Das ist besonders in Verbindung mit Licht- oder Soundschaltungen sehr praktisch, weil robuste kommerzielle Controller verwendet werden können.

Das Registrieren der Callback-Funktionen geschieht über die Aufrufe an `Midi.setOnNoteOnCallback()`, `Midi.setOnNoteOffCallback()` und `Midi.setOnControlChangeCallback()`. Die Callback-Funktionen im Sketch bekommen als Argument die binäre empfangene MIDI-Nachricht.

```
// Einbinden der MidiDuino-Library
#include <MidiUart.h>
#include <Midi.h>
MidiClass Midi;

// Die erste LED ist an Pin 10 angeschlossen.
int ledPin = 13;
// Die zweite LED ist an Pin 11 angeschlossen.
int ledPin2 = 11;

void setup() {
  // Initialisieren der MIDI-Schnittstelle
  MidiUart.init();
  // Setzen der LED-Pins auf Ausgang
  pinMode(ledPin, OUTPUT);
  pinMode(ledPin2, OUTPUT);

  // Beim Erkennen einer Note On-Nachricht wird onNoteOnCallback
  // aufgerufen.
  Midi.setOnNoteOnCallback(onNoteOnCallback);
  // Beim Erkennen einer Note Off-Nachricht wird onNoteOffCallback
  // aufgerufen.
  Midi.setOnNoteOffCallback(onNoteOffCallback);
// Beim Erkennen einer Control Change-Nachricht wird
// onControlChangeCallback aufgerufen.
  Midi.setOnControlChangeCallback(onControlChangeCallback);
}

void loop() {
```

```
  // Überprüfen, ob MIDI-Daten empfangen wurden
  if (MidiUart.avail()) {
    // Einlesen des empfangenen MIDI-Bytes
    byte c = MidiUart.getc();
    // Übergeben an die MIDI-Library, damit diese die Daten
    // erkennt
    Midi.handleByte(c);
  }
}

// Wenn eine eingehende Note On-Nachricht von der MIDI-Library
// erkannt wurde, wird diese Funktion aufgerufen.
void onNoteOnCallback(byte msg[]) {
  // Die Helligkeit der LED wird auf die Tonhöhe gesetzt.
  analogWrite(ledPin, map(msg[1], 0, 127, 0, 255));
}

// Wenn eine eingehende Note Off-Nachricht von der MIDI-Library
// erkannt wurde, wird diese Funktion aufgerufen.
void onNoteOffCallback(byte msg[]) {
  // Ausschalten der LED
  digitalWrite(ledPin, LOW);
}

// Wenn eine eingehende Control Change-Nachricht von der
// MIDI-Library erkannt wurde, wird diese Funktion aufgerufen.
void onControlChangeCallback(byte msg[]) {
  // Prüfen, ob es sich um CC Nummer 1 handelt
  if (msg[1] == 1) {
    // Helligkeit der zweiten LED auf den CC-Wert setzen
    analogWrite(ledPin2, msg[2]);
  }
}
```

Es ist also sehr einfach, mit der MidiDuino-Bibliothek MIDI-Schaltungen auf dem Arduino zu bauen. Die MidiDuino-Bibliothek bietet auch eine sehr angenehme Schnittstelle für die Zeitsynchronisation und das Sequencen von MIDI-Daten, sodass mit ihr eigenständige Musikinstrumente und Musiksequencer gebaut werden können.

KAPITEL 11

Musik mit Arduino

In diesem Kapitel:

- Töne aus dem Arduino
- Erster Sketch: Töne mit langsamer PWM
- Zweiter Sketch: Angenehme Klänge mit schneller PWM
- Dritter Sketch: Steuerung von Klängen
- Vierter Sketch: Berechnungen in einer Interrupt-Routine
- Fünfter Sketch: Musikalische Noten

Töne aus dem Arduino

In diesem Workshop kommt die volle Rechenkraft des Arduino zum Einsatz, um Töne zu generieren. Es werden zuerst einfache Klänge erzeugt, um dann später musikalische Noten zu berechnen und so den Arduino in einen Synthesizer zu verwandeln. Als Letztes wird der Arduino so programmiert, dass er automatisch Melodien und Rhythmen generieren kann. Es werden zwei Aspekte der elektronischen Musik beleuchtet: auf der einen Seite Klangerzeugung und Klangmodifizierung, auf der anderen algorithmische Musik (also die Beeinflussung von Tönen, Melodien, rhythmischen Mustern und harmonischen Reihenfolgen durch programmierte Regeln).

Sogar ein kleines Bauteil wie der Arduino kann dazu verwendet werden, musikalische Klänge und Geräusche zu erzeugen. Einige Projekte sollen hier kurz vorgestellt werden, um die ganze Bandbreite der Möglichkeiten zu zeigen und die Fantasie anzuregen.

Pump up the Volume!

Die beiden Stuttgarter Künstler Daniel Dihardja und Frank Arnold verwenden Handluftpumpen, die mithilfe von Piezo-Sensoren an einem Arduino angebracht werden. Diese Sensoren messen den Luftdruck, die Signale werden dann am PC in Klänge umgewandelt. Diese interaktive Soundinstallation namens Soundbeats erforscht ganz neue Bereiche der Klangerzeugung und soll, so die Initiatoren, die körperliche Tätigkeit des Pumpens mit der eigentlich unkörperlichen Musik verbinden. Das Ergebnis ist durchaus hörenswert, wie

ein Video auf der Projektwebsite unter *http://xciba.de/pumpbeats* zeigt. Dort heißt es: »Aus der Symbiose von Pumpe und Computer entsteht eine fruchtbare Bedeutungserweiterung beider Komponenten. »Pumpbeats« spielt demzufolge mit der Verbindung einer ›alten Kunst‹ (= Musik) und den neuen Medien.«

Eine Harfe aus Licht

Eines der spektakulärsten Arduino-Projekte ist die Laserharfe von Stephen Hobley (*http://www.stephenhobley.com/build/*): Ein Laserscanner sendet zehn Strahlen vom Boden an die Decke und sorgt mit passendem Nebel für grüne Saiten aus Licht. Hält nun der Harfenspieler eine Hand in das Licht, wird das von einem Sensor erkannt, der den daraus resultierenden Wert in ein MIDI-Signal verwandelt; ein Synthesizer sorgt dafür, dass Sounds entstehen. In der ersten Version der Harfe wurden noch Reflektionssensoren eingesetzt, um die Position der Hand zu messen. Da sich das aber als sehr kompliziert und unpräzise herausstellte, hat der Erfinder die Infrarotkamera einer Wiimote so modifiziert, dass sie die Frequenz eines grünen Lasers erkennen kann. So kann nun die Hand genauer geortet werden, die damit auch die Tonhöhe auf einem einzelnen Strahl moduliert.

Diver Sequencer

Der Diver Sequencer ist eine Halbkugel, auf der vier Drehregler angebracht sind. Ein Arduino gibt nun in regelmäßigen Abständen vier Töne nacheinander ab. Diese einzelnen Töne können durch die Drehregler bestimmt werden, indem die Frequenz moduliert wird. Zusammen mit der Geschwindigkeit, die ebenfalls gesteuert werden kann, ergibt sich also ein kleiner Sequencer, mit dem sich rhythmische wiederkehrende Tonfolgen erzeugen lassen. Diese Töne sind allerdings relativ rau, sodass die blinkende Halbkugel mit ihren Knöpfen wohl mehr Aufsehen erregen dürfte als das, was sie produziert. Mehr über das Projekt können Sie sich unter *http://bDruzed.com/2009/04/08/diver-sequencer-on-acid/* ansehen.

Klimpern bis zum Stubenarrest

So manche Eltern bereuten in den 1980er Jahren recht schnell, dass sie ihrem Kind eines dieser kleinen elektronischen Klaviere gekauft hatten, mit denen japanische Firmen den Markt überschwemmten. Die bunten Plastikgeräte, die nervig piepsende Töne von sich

gaben, konnten für stundenlange Begeisterung und blank liegende Nerven gleichzeitig sorgen. Wohl im Gedanken an diese Zeiten ist Pocket Piano Shield (*http://www.critterandguitari.com/home/store/arduino-piano.php*) entstanden, ein Aufsatz für den Arduino, mit dem genau solche Minisynthesizer selbst gebaut werden können. Und möglicherweise eignet sich dieses Gerät nicht nur für die musikalische Früherziehung, sondern kann Kindern auch gleichzeitig die Welt der Elektronik und Programmierung näherbringen.

Komponenten

Die Anzahl zusätzlicher Komponenten ist bewusst klein gehalten. Verwendet werden eine Stereoklinkenbuchse zum Anschluss eines Kopfhörers, ein Potentiometer (10.000 Ohm logarithmisch) und ein Widerstand (1.000 Ohm) zum Einstellen der Lautstärke, sowie zwei 1 mikro-Farad- Folienkondensatoren zum Filtern des digitalen Ausgangs aus dem Arduino. Für steuerbare Sketche werden zwei weitere Potentiometer (linear, von 5.000 Ohm bis 100.000 Ohm) und zwei Taster eingesetzt.

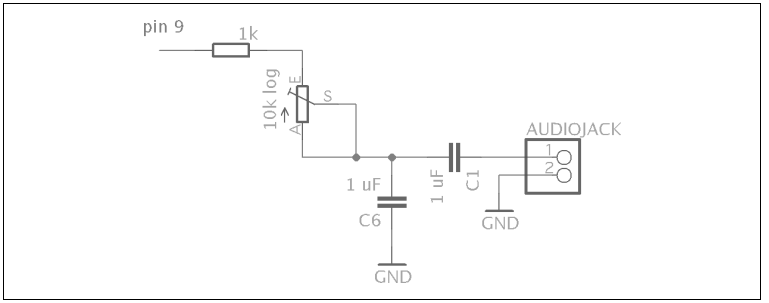
Warnung

In diesem Workshop kann es schnell zu lauten und unkontrollierten Tönen kommen, die sowohl das menschliche Gehör als auch angeschlossene Lautsprecher oder Kopfhörer beschädigen können. Schließen Sie keine hochwertigen Lautsprecher an die Schaltung und tragen Sie gegebenenfalls Hörschutz (oder halten Sie eine Hand auf dem Lautstärkeregler).

Schaltungsaufbau

Der Ausgangspin 9 des Arduino wird verwendet, um Klänge zu erzeugen. Dazu sind eine Kopfhörerbuchse, ein Widerstand, ein Potentiometer und zwei Folienkondensatoren notwendig. Der Widerstand wird eingesetzt, um allzu hohe Lautstärken zu verhindern, während das Potentiometer benutzt wird, um die Lautstärke generell einzustellen. Das Potentiometer ist ein logarithmisches Potentiometer, das im Bereich kleiner Widerstandswerte eine größere Auflösung hat. Der Grund dafür ist, dass das menschliche Gehör Lautstärke logarithmisch empfindet, wodurch es möglich ist, sowohl leichtes Getröpfel als auch einen Düsenjäger zu hören. Die Folienkondensatoren werden sowohl als Entkopplungskondensator (siehe Kasten »Elektronische Klangerzeugung«) wie auch als Tiefpassfilter eingesetzt.

Abbildung 11-1 ▶
Schaltplan (ohne Steuerung)



Elektronische Klangerzeugung

Das menschliche Ohr ist eines der genauesten Sinnesorgane. Es kann minimale Tonhöhenunterschiede erkennen, was auch die Anforderungen an den Arduino deutlich erhöht. Klang im Raum wird durch eine Veränderung des Luftdrucks erzeugt, z.B. durch die vibrierende Membran eines Lautsprechers oder Kopfhörers. Eine periodische Veränderung des Luftdrucks (durch eine Klangwelle) nimmt das menschliche Gehirn als Tonhöhe war. Je schneller die Luft vibriert, desto höher ist die wahrgenommene Tonhöhe.

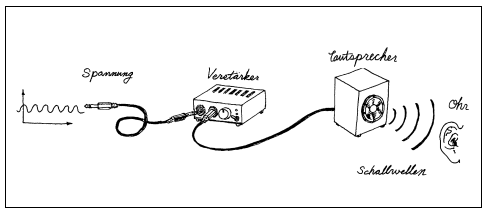
Die Ausbreitung der Kopfhörer- oder Lautsprechermembran lässt sich über eine Spannung steuern (die meistens von einen Verstärker verstärkt wird, um auch große Membranen bewegen zu können). Je höher die Spannung, desto größer die Ausbreitung der Membranen. Diese Spannungen werden als analoge Spannungen bezeichnet, weil sie eine große Menge von Werten annehmen können, z.B. von – 5 V bis + 5 V (also –5 V, – 4,8 V, 3 V, 3,1123 V usw . . .). Digitale Spannungen können nur zwei Werte annehmen, die »An« oder »Aus« signalisieren.

Weiterhin bewegt sich die

Lautsprechermembran sowohl nach vorn als auch nach hinten (vom Ruhezustand aus gesehen), sodass Audiosignale meistens als

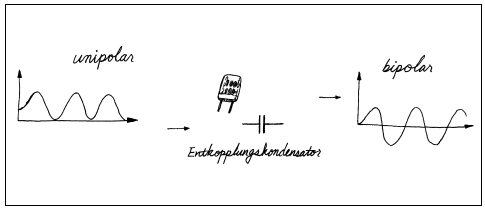
bipolare Spannungen übermittelt werden, die sowohl negativ als auch positiv sein können. (Spannungen, die nur negativ oder nur positiv sind, nennt man unipolare Spannungen.) Weiterhin können konstante Spannungen (die sich nicht verändern) Lautsprecher beschädigen, weil ihre Membran sich dauernd in Bewegung befinden muss. Um solche Beschädigungen zu vermeiden, werden

Entkopplungskondensatoren eingesetzt, die konstante Spannungen herausfiltern. Eine willkommene Nebenwirkung ist, dass so ein Kondensator auch eine sich verändernde unipolare Spannung in eine bipolare umwandelt.



▲ **Abbildung 11-2:** Klangerzeugung

Anschaulich heißt das, dass das Audiosignal 0 Volt wird, wenn es lange gleich bleibt, und sonst um 0 Volt herumpendelt.



▲ **Abbildung 11-3:** Bipolare Signale und Entkopplung

Der Arduino ist ein digitaler Mikrocontroller: Er kann keine analogen Spannungen erzeugen, wie sie für die Erzeugung von Musik notwendig sind (siehe Kasten »Elektronische Klangerzeugung«). Die digitalen Ausgangsspannungen des Arduino (0 und 5 Volt) müssen also in analoge Spannung konvertiert werden. Hier wird wieder die Pulsweitenmodulation eingesetzt (Kapitel 3). Ein zweiter Kondensator wird verwendet, der die harten Kanten des PWM-Signals filtert. Die PWM wird mit einer sehr hohen Frequenz angewendet (62.500 Hertz) und dann relativ hart gefiltert (die hohen Frequenzen werden abgeschnitten). Dadurch bleibt nur noch der Mittelwert der PWM übrig. Da die PWM so schnell ist, ist sie allerdings nicht von Menschen hörbar, sodass der Kondensator nur fakultativ ist.

Testweise kann man den Tiefpassfilterkondensator aus der Schaltung entfernen, um die hohen Frequenzen der PWM zu hören (die sind allerdings so hoch, dass man sie als Mensch kaum hört).

Dieser Weg ist immer noch relativ grob aufgelöst und ungenau, sodass meistens in Synthesizern ein Digital-Analog-Wandler zum Einsatz kommt. Ein DA-Wandler ist ein dedizierter Baustein, der ein digitales Signal in ein analoges Signal umwandelt. Jede Soundkarte besteht zum Beispiel aus mehreren DA- und AD-Wandlern (um Eingangssignale aufzunehmen, z.B. von einem Mikrofon).

Erster Sketch: Töne mit langsamer PWM

In diesem ersten Sketch kommt die hauseigene Pulsweitenmodulation von Arduino zum Einsatz, um Klänge zu erzeugen. Pin 9 ist der Audioausgang, der an den Kopfhörerausgang angeschlossen ist.

```
#define AUDIO 9 // Definieren des Audioausgangspins
void setup() {
  pinMode(AUDIO, OUTPUT); // der Audiopin ist ein Ausgangspin
}
int time = 200; // die Zeit zwischen steigender und fallender
                // Flanke
void loop()  // die Arduino-Hauptschleife
{
  analogWrite(AUDIO, 255); // hohe Flanke auf Audioausgang
  delayMicroseconds(time); // Länge der Flanke = time in
                           // Mikrosekunden
  analogWrite(AUDIO, 0);   // tiefe Flanke auf Audioausgang
  delayMicroseconds(time); // Länge der Flanke = time in
                           // Mikrosekunden
}
```

Als Erstes wird hier Pin 9 als Ausgang definiert. In der Hauptroutine wird mit `analogWrite()` der Pulsweitenmodulationswert auf dem Ausgangspin bestimmt. Durch Verändern des Wertes der Variablen `time` können verschiedene Tonhöhen erzeugt werden.

Es können auch interessantere elektronische Geräusche erzeugt werden, indem z.B. die Frequenz in einer Schleife verändert wird.

```
void loop() {
  // in dieser Schleife wird time von 0 bis 2000 hochgezählt
  {
    analogWrite(AUDIO, 255); // hohe Flanke auf Audioausgang
    delayMicroseconds(time); // Länge der Flanke = time in
                             // Mikrosekunden
    analogWrite(AUDIO, 0);   // tiefe Flanke auf Audioausgang
    delayMicroseconds(time); // Länge der Flanke = time in
                             // Mikrosekunden
  }
}
```

Man kann auch die Zeit immer zufällig wählen lassen:

```
void loop() {
  int volume = random(0, 255); // jede Note bekommt eine zufällige
                               // Lautstärke
  int time = random(1, 2000);  // und eine zufällige Tonhöhe
  // jede Note wird für 50000 Mikrosekunden gehalten (ungefähr)
  for (unsigned int i = 0; i < 50000; i += time) {
    analogWrite(AUDIO, volume); // hohe Flanke am Audioausgang
    delayMicroseconds(time);    // Länge der Flanke = time in
                                // Mikrosekunden
    analogWrite(AUDIO, 0);      // tiefe Flanke am Audioausgang
    delayMicroseconds(time);    // Länge der Flanke = time in
                                // Mikrosekunden
  }
}
```

Zweiter Sketch: Angenehme Klänge mit schneller PWM

Normalerweise läuft die Pulsweitenmodulation auf dem Arduino mit 500 Hz. Das heißt, dass der höchste Ton, den der Arduino erzeugen kann, im unteren Bereich des menschlichen Gehörs liegt. Dadurch klingen viele der Klänge seltsam und unangenehm, weil die Frequenz zu niedrig ist, um »glatte« analoge Spannungen zu erzeugen. Die PWM-Frequenz des Arduino vermischt sich mit der Frequenz, die erzeugt werden soll. Im nächsten Schritt wird die PWM modifiziert, damit sie anstatt mit 500 mit 62.500 Hertz läuft, was deutlich ansprechendere Klänge ermöglicht.

Um die PWM-Frequenz zu erhöhen, müssen spezielle Register des Arduino-Prozessors geschrieben werden. Diesen Teil kann man eigentlich getrost überspringen; wer es jedoch ganz genau wissen will, kann in der technischen Dokumentation von Atmel auf *http://www.atmel.com/dyn/products/product_card.asp?PN=ATmega328P* nachlesen, wie die PWM-Einheit funktioniert. Um die Frequenz des PWM-Ausgangs auf Pin 9 und Pin 10 auf 62.500 Hz zu erhöhen, müssen folgende Zeilen in die `setup()`-Funktion eingefügt werden.

```
TCCR1A = _BV(WGM10) | _BV(COM1A1);
TCCR1B = _BV(CS10) | _BV(WGM12);
```

Weiterhin wird nicht mehr die Funktion `analogWrite()` verwendet, um den PWM-Wert zu setzen, sondern es wird gleich das Hardwareregister gesetzt (wie schon in Kapitel 4 bei der LED-Matrix angewandt). Um den Code lesbar zu halten, wird eine Funktion definiert.

```
void writeAudio(uint8_t val) {
   OCR1A = (val);
}
```

Warnung Obwohl die schnelle Pulsweitenmodulation die Qualität des Klangs erheblich verbessert, sind Steckboards, wie sie mit Arduino verwendet werden, keine gute Grundlage, um Hi-Fi-Klänge zu erzeugen. Das Rauschen auf solchen Boards ist sehr stark. Deutlich bessere Ergebnisse lassen sich erreichen, indem die Schaltung auf einem kleinen Protoboard zusammengelötet und in einem abschirmenden Metallgehäuse befestigt wird.

In diesem ersten Sketch mit schneller PWM wird nur ein einzelner statischer Ton erzeugt. Es wird ein Zähler hochgezählt, der von 0 bis 1.024 geht. Da die PWM-Breite nur von 0 bis 255 geht, wird dieser Wert durch 4 geteilt, bevor *er mit der Funktion* `writeAudio()` ausgegeben wird. Dadurch wird eine Sägezahn-Wellenform erzeugt, weil die erzeugte analoge Spannung immer von 0 bis 5 Volt geht und dann wieder zurück auf 0 fällt. Die Geschwindigkeit, mit der diese Spannung hochgezählt wird, bestimmt auch die Tonhöhe. Diese lässt sich in der Variablen `inc` einstellen.

```
int audioPin = 9;  // Definition des Audio-Ausgangspins
// Funktion zum Setzen des PWM-Wertes, dazu wird das PWM-Register
// OCR1A verwendet
void writeAudio(uint8_t val) {
   OCR1A = (val);
}
```

```
// In der Initialisierungsroutine werden der Audioausgangspin als
// Ausgang definiert und die deutlich schnellere PWM konfiguriert.
void setup() {
  pinMode(audioPin, OUTPUT);
  TCCR1A = _BV(WGM10) | _BV(COM1A1);
  TCCR1B = _BV(CS10) | _BV(WGM12);
}
// In dieser Hauptschleife wird eine Sägezahn-Wellenform erzeugt,
// indem der PWM-Wert (also die analoge Ausgangsspannung) alle 20
// Mikrosekunden hochgezählt wird.
void loop() {
  int inc = 20;
  // Hochzählen der Spannung für die analoge Wellenform
  for (unsigned int j = 0; j < 1024; j += inc) {
    // Die PWM-Breite geht nur von 0 bis 255, also wird die
    // Variable j durch 4 geteilt. Dass j bis 1.024 geht,
    // ermöglicht einen größeren Raum an Tonhöhen.
    writeAudio(j / 4);
    // kurz warten, damit die Töne nicht zu hoch sind
    delayMicroseconds(20);
  }
}
```

Dritter Sketch: Steuerung von Klängen

In diesem Abschnitt wird der Arduino-Klangerzeuger um externe Steuerungsmöglichkeiten ergänzt. Zum Einsatz kommen ein Taster, der an den digitalen Eingang 4 angeschlossen wird, und ein Potentiometer, das an den analogen Eingang 0 angeschlossen wird.

Der Ton soll in diesem Sketch nur erklingen, wenn ein Taster gedrückt wird. Der Taster wird in der Hauptroutine abgefragt, und der Arduino erzeugt nur dann eine Wellenform, wenn der digitale Wert am Tastereingang LOW ist. Weiterhin wird die Frequenz der erzeugten Wellenform an gleichmäßigen Intervallen zufällig neu gesetzt. Der Frequenzbereich (also die Höhe der zufälligen Töne) wird durch das Potentiometer gesteuert, das an den analogen Eingang 0 angeschlossen ist.

```
int audioPin = 9;
void writeAudio(uint8_t val) {
    OCR1A = (val);
}
// Der Taster wird an den digitalen Eingang 4 angeschlossen.
int buttonPin = 4;
void setup() {
  // Hier wird der Tastereingang definiert.
  pinMode(buttonPin, INPUT);
  // restliche Initialisierung wie üblich
  pinMode(audioPin, OUTPUT);
```

```
  TCCR1A = _BV(WGM10) | _BV(COM1A1);
  TCCR1B = _BV(CS10) | _BV(WGM12);
}
int inc = 2; // Zähler zum Einstellen der Tonhöhe
void loop() {
  // Die äußere Schleife ist dazu da, Noten im regelmäßigen
  // Abständen zu ändern.
  for (int i = 0; i < inc * 5; i++) {
    // Erzeugen der Sägezahnwellenform, von 0 bis 1024
    for (int j = 0; j < 1024; j += inc) {
      // Wenn der Taster gedrückt ist, wird die Wellenform auch
      // ausgegeben.
      if (digitalRead(buttonPin) == LOW) {
        writeAudio(j / 4);
      // sonst wird der Ausgang auf stumm gestellt
      } else {
        writeAudio(0);
      }
      delayMicroseconds(20);
    }
  }

  // Nach Ablaufen der äußeren Schleife wird hier eine neue
  // Tonhöhe zufällig eingestellt. Der Bereich, aus dem die neue
  // Frequenz gesetzt wird, wird durch das Potentiometer
  // eingestellt.
   int tieferTon = analogRead(0) / 16;
   inc = random(tieferTon, tieferTon + 10);
}
```

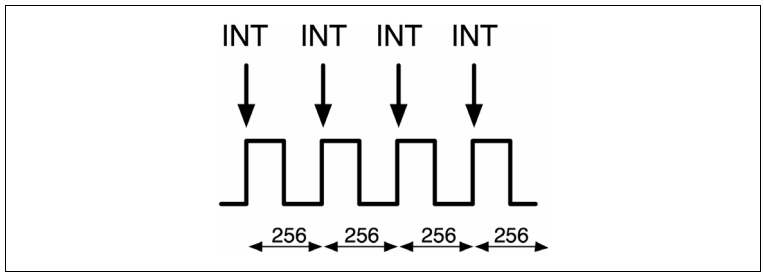
Vierter Sketch: Berechnungen in einer Interrupt-Routine

Im Gegensatz zu blinkenden oder gedimmten LEDs, bei denen es nur grob auf zeitliche Genauigkeit ankommt, ist es bei der Klangerzeugung enorm wichtig, das Zeitverhalten sehr genau zu berechnen. Kleinste Abweichungen führen zu einem Synthesizer, der verstimmt klingt. Weiterhin lässt sich feststellen, dass der Steuerungsprogrammcode eng mit dem Code zum Erzeugen der Wellenformen verknüpft ist, was schnell unhandlich wird. Im nächsten Sketch wird ein weiterer technischer Trick eingeführt, um die Berechnung der Wellenform von der Hauptschleife zu trennen.

Bis jetzt wurde die zeitliche Synchronisation der Wellenform mithilfe der Funktionen `delay()` und `delayMicroseconds()` implementiert. Diese Funktionen sind allerdings nicht sehr genau, und auch die Ausführungszeit des Programmcodes in der Hauptroutine hat einen Einfluss auf die Tonhöhe. Um eine genaue Synchronisation

zu erreichen, muss eine Interrupt-Routine eingeführt werden. Das ist eine Funktion, die vom Arduino automatisch aufgerufen wird, wenn z.B. der Zähler der PWM-Einheit überläuft. Dadurch ist es möglich, eine Funktion sehr regelmäßig aufzurufen. Da die PWM-Einheit zum Erzeugen der analogen Ausgangsspannung mit 62.500 Hz läuft, ist es maximal möglich, 256 Befehle in dieser Routine auszuführen. Der Programmcode muss also besonders sorgfältig geschrieben werden.

Abbildung 11-4 ▶
Aufruf der Interrupt-Routine



In diesem Sketch wird innerhalb der Interrupt-Routine eine rechteckige Wellenform generiert. Dazu wird die Zählervariable `phase` hochgezählt wie im vorigen Sketch. Der Zähler geht von 0 bis 65.535. Ist der Zählerwert größer als 32.768 (also höher als die Hälfte des möglichen Wertes), wird die Ausgangsspannung auf 255 gesetzt; ist der Zählerwert kleiner als 32.768, auf 0. Dadurch wird eine Rechteck-Wellenform generiert, die deutlich langsamer ist als die PWM-Rechteck-Wellenform. Die Frequenz dieser Wellenform lässt sich über die Variable `speed` festlegen. Sie wird bei jedem Aufruf der Interrupt-Funktion zur Variable `phase` hinzuaddiert. Ist `phase` größer als 65.535, läuft die Variable automatisch über und fängt wieder bei 0 an.

In der Interrupt-Routine wird als Erstes die analoge Ausgangsspannung gesetzt. Dadurch hat die Dauer des folgenden Programmcodes keinen Einfluss auf die zeitliche Genauigkeit der Routine. In der Hauptschleife wird jetzt nur noch die Tonhöhe gesetzt, indem der Wert des Potentiometers abgelesen und in der Variable `speed` gespeichert wird. Dadurch ist eine saubere Trennung von Kontrollcode und Audioberechnung gewährleistet.

```
// Definition des Ausgangspins
int audioPin = 9;
// Funktion zum Setzen der analogen Spannung
void writeAudio(uint8_t val) {
   OCR1A = (val);
}
```

```
// digitaler Eingang für den Taster
int buttonPin = 4;
// Variable für die Erzeugung der Wellenform innerhalb der
// Interrupt-Routinephase ist ein Zähler, der von 0 bis 65535
// geht
uint16_t phase = 0;
// speed bestimmt, wie schnell phase hochgezählt wird.
uint16_t speed = 200;
// sample ist die Variable, in der die analoge Ausgangsspannung
// gespeichert wird.
uint8_t sample = 0;
// Das ist die Definition der Interrupt-Routine, die 625.000 Mal
// pro Sekunde aufgerufen wird (bei jedem Überlauf des PWM-
Timers).
SIGNAL(TIMER1_OVF_vect) {
  // Als Erstes wird die analoge Spannung gesetzt.
  writeAudio(sample);
  // phase wird hochgezählt: je größer speed ist, desto schneller
  phase += speed;
  // Wenn phase größer als die Hälfte ist, wird die analoge
  // Ausgangsspannung auf 5V gesetzt, sonst auf 0V.
  // Dadurch wird eine regelmäßige Rechteck-Wellenform generiert.
  if (phase >= 32768)
    sample = 255;
  else
    sample = 0;
}
// In der Routine setup werden der PWM-Timer konfiguriert und die
// Interrupt-Routine aktiviert.
void setup() {
  pinMode(audioPin, OUTPUT);
  pinMode(buttonPin, INPUT);
  TCCR1A = _BV(WGM10) | _BV(COM1A1);
  TCCR1B = _BV(CS10) | _BV(WGM12);
  // Aktivieren der Interrupt-Routine
  TIMSK1 |= _BV(TOIE1);
}
void loop() {
  // In der Hauptschleife wird nur noch die Tonhöhe bestimmt,
  // indem die Variable speed gesetzt wird.
  speed = analogRead(0) + 1;
}
```

Durch diese Trennung von Kontroll- und Audioberechnungscode lassen sich jetzt deutlich kompliziertere Klänge generieren, ohne dass der Programmcode unübersichtlich wird. Im nächsten Sketch wird die Tonhöhe des generierten Tons durch eine zweite, langsamere Wellenform kontrolliert. So eine langsame Wellenform, die keinen Klang an sich erzeugt, sondern Parameter steuert, wird bei Synthesizern *LFO* genannt, Low Frequency Oscillator, was so viel

heißt wie »Niedrigfrequenzoszillator«. Dadurch lassen sich z.B. Sirenenklänge erzeugen. Hier wird nur die Hauptfunktion vorgestellt, `setup()` und Interrupt-Routine bleiben gleich.

```
uint16_t lfo_phase = 0;
uint16_t lfo_speed = 200;
uint8_t  lfo_sample = 0;
void loop() {
  lfo_speed = analogRead(0) * 5 + 1;
  lfo_phase += lfo_speed;
  lfo_sample = lfo_phase >> 8;
  speed = lfo_sample * 5;
  delay(5);
}
```

Der Code zum Berechnen der langsamen Wellenform (hier eine Sägezahn-Wellenform) ist fast derselbe wie zur Berechnung der Rechteck-Wellenform. Diese Vorgehensweise beim Berechnen der Wellenform nennt man einen »Phasenakkumulator«. Deswegen heißt der Zähler für die Wellenformen auch `phase`. Hier steuert das Potentiometer die Geschwindigkeit der langsamen Wellenform.

Fünfter Sketch: Musikalische Noten

Bis jetzt wurden in den Sketchen nur Geräusche und Klänge erzeugt, jedoch keine Noten. Mit der Berechnung der Interrupt-Routine ist jedoch ein solides Fundament gegeben, um genaue Tonhöhen zu erhalten. Jede musikalische Note in unserem Zwölftonsystem entspricht einer Frequenz, und jeder dieser Frequenzen entspricht auch ein Wert der Variablen `speed`. Damit diese Werte nicht jedes Mal mit einer komplizierten Formel berechnet werden müssen, werden die Werte von `speed` in einer Tabelle gespeichert.

Im nächsten Sketch wird die Tonhöhe bei jedem Druck auf den Taster auf eine zufällige Note aus einem Moll-Arpeggio erzeugt. Dadurch lassen sich angenehm klingende Melodien erzeugen. Es wird nur der Programmcode für die Hauptroutine vorgestellt.

In der Tabelle *freqtable* werden die Werte der Variablen `speed` für jede Note eingetragen. 60 Werte entsprechen fünf Oktaven. In der Tabelle *arp* werden nur die vier Noten aus einem Moll-Arpeggio festgehalten: 0 (Grundton), 3 (Mollterz), 7 (Quinte) und 10 (kleine Septime). Wird der Taster betätigt (geht also der Wert am digitalen Eingang von HIGH auf LOW), wird eine zufällige Note aus *arp* gelesen (`arp[random(4)]`), und eine zufällige Oktave hinzuaddiert (`random(4) * 12`). Der Wert von `speed` wird dann aus der Tabelle *freqtable* gelesen.

```
// Werte der Variablen speed für 60 chromatische Noten
// (fünf Oktaven) const uint16_t freqtable[60] = {
  69,    73,   77,   82,   86,   92,   97,  103,  109,  115,  122,  129, // 12
  137,  145,  154,  163,  173,  183,  194,  206,  218,  231,  244,  259, // 24
  274,  291,  308,  326,  346,  366,  388,  411,  435,  461,  489,  518,  // 36
  549,  581,  616,  652,  691,  732,  776,  822,  871,  923,  978,  1036, // 48
  1097, 1163, 1232, 1305, 1383, 1465, 1552, 1644, 1742, 1845, 1955, 2071, // 60
};
// Noten in einem Moll-Arpeggio
uint8_t arp[4] = { 0, 3, 7, 10 };
// Speichern der vorigen Werts des digital Eingangs für den
// Taster; dadurch lassen sich einzelne Tasterdrücke erkennen
int oldButtonPress = HIGH;
void loop() {
  int buttonPress = digitalRead(BUTTON1);
  // Wurde der Taster betätigt, wird speed ein neuer Wert
  // zugewiesen, der einer zufälligen Note in einem Moll-Arpeggio
  // entspricht.
  if (oldButtonPress == HIGH && buttonPress == LOW) {
    speed = freqtable[arp[random(4)] + random(4) * 12];
  }
  // Speichern des jetzigen Tasterwerts, um beim nächsten
  // Durchlauf ein Drücken zu erkennen
  oldButtonPress = buttonPress;
}
```

Anhand der Frequenztabellen können jetzt auch Melodien auf dem Arduino implementiert werden. Dazu speichert eine Tabelle die Reihenfolge der Tonhöhen, und eine andere die Länge der Noten. Es werden nur die Programmzeilen ab `uint8_t arp` geändert:

```
// Diese Tabelle speichert die Noten der Melodie.
uint8_t melody[8] = { 12, 15, 17, 12, 24, 27, 12, 19 };
// Diese Tabelle speichert die Länge der Noten.
uint8_t lengths[8] = { 100, 50, 100, 50, 100, 50, 100, 50 };
void loop() {
  // Die Hauptroutine läuft durch die Noten und Längentabellen und
  // spielt die Noten ab.
  for (int i = 0; i < 8; i++) {
    // Setzen der Tonhöhe der aktuell gespielten Noten
    speed = freqtable[melody[i]];
    // Warten für die gespeicherte Länge
    delay(lengths[i] * 2);
  }
}
```

Durch Ändern der Werte in den Tabellen lassen sich neue Melodien schreiben. Das Konzept kann auch weitergeführt werden, um neue Melodien auf Knopfdruck zu generieren. Besonders interessant ist, Tonlängen unabhängig von den Tonhöhen zu generieren. Wenn der Taster gedrückt wird, werden beide Tabellen neu gefüllt, und eine separate Variable mit der Länge der Tabellen wird neu gesetzt.

Frequenzen, Filter und Tiefpassfilterung

Bis jetzt wurde Klang als eine Folge von Spannungen in der Zeit betrachtet, wodurch man relativ viel über die Lautstärke des Signals aussagen kann: Je größer die Ausschläge des Signals, desto höher seine Lautstärke. Aber über die eigentliche Klangfarbe ließ sich relativ wenig aussagen. Klänge kann man als schrill oder weich einordnen, blechern oder warm, statisch oder lebendig, und die meisten dieser Bezeichnungen beziehen sich unmittelbar auf den Frequenzinhalt eines Klangs.

Eine andere Möglichkeit, Klanginformation zu betrachten, ist das Frequenzspektrum. Jede Note aus einem Synthesizer besteht aus einem Grundton, der die stärkste Frequenz in dem Ton und auch die wahrgenommene Tonhöhe ist. Zusätzlich zu diesem Grundton gibt es eine Reihe von Obertönen, die bestimmen, wie der Ton klingt. Der Ton einer Geige hat ein deutlich anderes Frequenzspektrum als der einer Oboe. Grundsätzlich gilt: Je mehr Obertöne, desto härter und metallischer klingt der Ton.

Diese Darstellungsweise ist auch für Geräusche nützlich. Zum Beispiel umfasst Rauschen alle möglichen Frequenzen, sodass kein einzelner Ton stärker heraussticht als ein anderer.

Zusätzlich zu diesen Obertönen ist auch der zeitliche Verlauf der Frequenzinformation sehr wichtig. Eine Gitarre zum Beispiel hat einen recht harten Anschlag mit vielen Obertönen, aber nach diesem kurzen Anschlag ist der eigentliche Klang recht warm, d.h. mit weniger Obertönen.

Die zeitliche Wellenform des Klangs umfasst auch den Frequenzinhalt, nur ist dieser nicht so leicht ersichtlich. Als Faustregel gilt: Je »schärfer« die Kanten, desto mehr obere Frequenzen gibt es. Eine Sinuswelle (weiche Wellenform) hat keine Obertöne, während eine Rechteckwellenform viele hat. Deswegen ist auch die vom Arduino bei der Pulsweitenmodulation erzeugte Wellenform sehr obertonlastig; sie klingt sehr »digital«.

Mit sogenannten Filtern wird der Frequenzinhalt eines Klangs bearbeitet. Ein Tiefpassfilter schneidet obere Frequenzbereiche ab, während ein Hochpassfilter nur höhere Frequenzbereiche durchlässt. Der Entkopplungskondensator ist in diesem Sinne ein Hochpassfilter, der sehr niedrige (konstante) Frequenzen blockiert. Filter lassen sich sowohl als Programmcode implementieren (das nennt man dann generell DSP-Programmierung, Digital Signal Processing) als auch als elektronische Schaltungen, wie man es aus analogen Synthesizern oder Pedalen für E-Gitarren kennt. Ein Kondensator und ein Widerstand bilden schon einen elektronischen Filter, und in diesem Kapitel werden zwei davon eingesetzt: der Entkopplungskondensator und der Tiefpassfilter, der die Obertöne aus dem Arduino wegfiltert.

Der Tiefpassfilter in der vorgestellten Arduino-Schaltung ist sehr tief gesetzt und filtert die PWM-Rechteckwelle so stark, dass am Ende nur noch eine konstante Spannung übrig bleibt. Die hohe Geschwindigkeit der eingestellten PWM ermöglicht es, auch mit einem digitalen Ausgang schnelle, »runde« Wellenformen wie z.B. Sinuswellen zu erzeugen.

Theremin

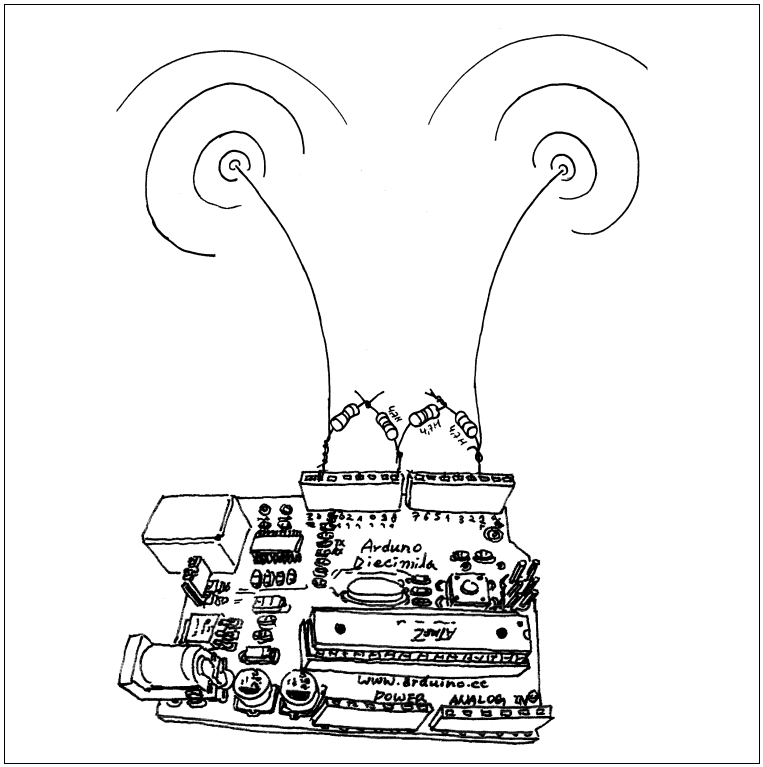
Der Klang des ursprünglichen Theremins (ein elektronisches Musikinstrument, das ohne körperliche Berührung gespielt wird) ähnelt im Vergleich mit klassischen Instrumenten am ehesten einer Geige und klingt ähnlich wie ein Fuchsschwanz.

Gesteuert wird das Theremin mit den Händen, die zwei kapazitive Sensoren rechts und links am Gehäuse beeinflussen. Die Form die-

ser Antennen ist so ausgeklügelt, dass sich ein nahezu linearer Zusammenhang zwischen Tonhöhe und Abstand ergibt. Zu der Zeit der Erfindung des Theremins 1919 waren Mikrocontroller allerdings noch unbekannt; heute ist es uns möglich, die teilweise fehlerhaften Sensordaten mithilfe von Software vergleichsweise leicht zu korrigieren und so einen noch besseren Klang zu erzeugen.

Hardware

Die zwei Antennen des Theremins werden aus zwei ca. 30 bis 40 cm langen Drähten gebogen und mit den Pins 4 und 5 verbunden. Zusätzlich wird jeweils ein 10-Mega-Ohm-Widerstand mit der Antenne und Pin 3 verbunden. Da 10-Mega-Ohm-Widerstände nicht so leicht zu finden sind, können mehrere 1-Mega-Ohm Widerstände zusammengelötet werden.



◀ **Abbildung 11-5**
Theremin-Aufbau

Software

Um das Theremin hörbar zu machen, gibt es verschiedene Möglichkeiten: Die Sensordaten können an den PC gesendet werden

(Kapitel 10), wo z.B. ein Softwaresynthesizer oder Processing Töne erklingen lässt, oder man schließt direkt einen Kopfhörer an den Atmel an, wie bei den vorigen Beispielen beschrieben ist.

Als Erstes wird ein stabiler Nullpunkt für Lautstärke und Tonhöhe benötigt. Da sich die Kapazität der Umwelt ständig ändert, soll sich dieser Nullpunkt dynamisch anpassen. So ist ein ständiges Nachstimmen des Instruments wie beim Original nicht notwendig. Dazu wird in den Variablen `ton_null` und `volume_null` der jeweils kleinste jemals gemessene Wert gespeichert. Allerdings gibt es immer wieder Messfehler. Um diese etwas abzumildern, wird der Nullpunkt nur um einen Bruchteil verändert, sodass einzelne Ausreißer nicht in die Summe eingehen.

```
float volume_null = 0;
float ton_null = 0;
float gew_neg = 0.1;
float gew_pos = 0.0001;
void Adjust_Null(float volume_raw, float ton_raw)
{
    if (volume_raw < volume_null) {
      volume_null = volume_null * (1-gew_neg) + volume_raw * gew_neg;
    }
    if (volume_raw > volume_null) {
      volume_null = volume_null * (1-gew_pos) + volume_raw * gew_pos;
    }
    if (ton_raw < ton_null) {
      ton_null = ton_null * (1-gew_neg) + ton_raw * gew_neg; }
    if (volume_raw > volume_null) {
      ton_null = ton_null * (1-gew_pos) + ton_raw * gew_pos;
    }
}
```

Mit den zwei Werten `gew_neg` und `gew_pos` kann die Geschwindigkeit, mit der sich der Nullpunkt an geänderte Bedingungen anpasst, verändert werden. Dabei ist aber zu beachten, dass `gew_pos` ziemlich klein sein sollte, da sich das Instrument sonst beim Spielen verstimmt.

Die Werte `ton_null` und `volume_null` können nun vom Raw-Wert abgezogen werden: Das Ergebnis ist eine Zahl, die nicht mehr von der Umgebung abhängig ist. Damit ist aber noch nicht festgelegt, was der höchste Wert und damit der höchste Ton oder die lauteste Lautstärke ist. Auch hier kann der gleiche adaptive Algorithmus verwendet werden, sodass nach dem Einschalten nur einmalig die Hand bis ganz zum Instrument (aber ohne die Antenne zu berühren) geführt werden muss, um immer den gleichen Tonumfang zu

erhalten. Dazu müssen nur in der Funktion `Adjust_Null` oben und unten vertauscht werden.

```
float volume_max = 0;
float ton_max = 0;
void Adjust_Max(float volume_raw, float ton_raw)
{
  if (volume_raw > volume_max) {
    volume_max = volume_max * (1-gew_neg) + volume_raw * gew_neg;
    }
    if (volume_raw < volume_max) {
      volume_max = volume_max * (1-gew_pos) + volume_raw * gew_pos;
    }
    if (ton_raw > ton_max) {
      ton_max = ton_max * (1-gew_neg) + ton_raw * gew_neg; }
    if (volume_raw < volume_max) {
      ton_max = ton_max * (1-gew_pos) + ton_raw * gew_pos;
    }
}
```

Jetzt ist es ein Leichtes, direkt brauchbare Werte zu berechnen. Die Töne sollen von 0 bis 12 (für eine Oktave) und die Lautstärke von 0 bis 100 skaliert werden.

```
ton     = (ton_raw - ton_null)/(ton_max-ton_null)*12;
volume  = (volume_raw - volume_null)/(volume_max-volume_null)*12;
```

Je nach Laune könnte man die Töne jetzt noch quantisieren, also auf ganze Zahlen aufrunden, wodurch das Spielen einfacher wird, man aber auch viel von der Ausdruckskraft des Theremins verliert, die ja gerade in der Möglichkeit besteht, frei zu spielen.

Wir wollen für das Theremin einen schönen, weichen Sinusklang erzeugen. Dafür verwenden wir eine Tabelle mit 256 Sinuswerten, da das Berechnen eines Sinus viel zu lange dauern würde. Abhängig von der Tonhöhe müssen jetzt Werte aus dieser Tabelle gelesen und mit PWM ausgegeben werden. Dazu wird ein Zähler benutzt, der angibt, welches der nächste Wert aus der Tabelle ist. Abhängig davon, um wie viel dieser Zähler in jedem Durchgang erhöht wird, entsteht eine andere Tonhöhe.

Würden hier ganze Zahlen verwendet, könnten nur der Grundton und Vielfache davon erzeugt werden. Ein 16-Bit-Wert kann aber auch so interpretiert werden, als ob die oberen 8 Bit die Ganzzahl darstellen und die unteren hinter dem Komma stehen würden. Wird jetzt z.B. jedes Mal 64 zum Zähler addiert, ändern sich die oberen 8 Bit erst nach dem vierten Mal. Die oberen 8 Bit geben uns also an, welcher Wert aus der Sinustabelle verwendet werden soll; dieser Wert muss nur noch mit der gewünschten Lautstärke multipliziert werden, bevor er ausgegeben wird.

ANHANG A

Arduino-Boards und -Shields

In diesem Kapitel:

- Arduino-Boards
- Arduino-Shields

Arduino-Boards

Arduino Duemilanove/Diecemila

Der Arduino Duemilanove ist wohl der derzeit am meisten verkaufte Arduino und kann als das »Standardmodell« angesehen werden. In diesem Buch wird bei allen Workshops ein Duemilanove eingesetzt. Er verfügt über 14 digitale Pins, von denen sechs PWM-fähig sind, sowie sechs Analoge Eingänge. Weiterhin ist auf dem Board ein USB-Port angebaut, der dem Duemilanove erlaubt, über eine serielle Verbindung mit einem PC zu kommunizieren. Als Stromquelle dient entweder der USB-Port oder ein 5-Volt-Netzteil, für das eine Buchse angebracht ist. Der Duemilanove läuft standardmäßig mit einem Atmega168- oder Atmega328p-Controller, der auf einem Steckso-ckel angebracht ist. Eine ausführliche Erläuterung finden Sie unter *http://arduino.cc/en/Main/ArduinoBoardDuemilanove*.

Der Diecemila (italienisch für 10.000) ist der Vorgänger des Duemilanove. Die Unterschiede dabei liegen in Details. So muss die Stromquelle für den Duemilanove nicht mehr per Hand umgeschaltet werden. Da der Diecemila keinerlei Vorteile bietet, empfiehlt es sich nicht, ihn noch zu kaufen. Wer ein solches Board besitzt, kann damit aber die Workshops in diesem Buch ebenso durchführen wie mit einem Duemilanove. Mehr Informationen finden Sie unter *http://arduino.cc/en/Main/ArduinoBoardDiecimila*.

Arduino Mega

Der Arduino Mega hält, was der Name verspricht: Ein Atmega1280 ist in der Lage, 54 digitale Pins zu steuern, von denen 14 PWM-fähig sind. Er verfügt zudem über 16 analoge Inputs, vier serielle Ports, 128 KByte Speicher und 8 KByte Arbeitsspeicher. Dieses Monster kann eingesetzt werden, wenn die Fähigkeiten eines Duemilanove nicht mehr ausreichen, zum Beispiel bei Installationen mit vielen LEDs oder Displays oder wenn man mit mehr als einem Gerät kommunizieren will. Sie sollten dabei die höhere Betriebsspannung von 7 bis 12 Volt beachten. Der Arduino Mega ist mit 50 Euro allerdings auch etwa doppelt so teuer wie der Duemilanove.

Arduino Bluetooth

Die besonders edle Variante des Arduino ist der Arduino Bluetooth. Er besitzt statt eines USB-Ports einen Bluetooth-Chip, der mit 115.200 Bit pro Sekunde mit einer Gegenstelle kommunizieren kann. Das ist zwar für die Entwicklung relativ egal, bietet aber enorme Vorteile, wenn der Arduino irgendwo im Raum angebracht werden und dennoch mit einem Rechner kommunizieren soll. Dank dem Bluetooth-Chip ist diese Variante allerdings so teuer, wie sie komfortabel ist. Einige Informationen zu diesem nicht sehr weit verbreiteten Board gibt es unter *http://arduino.cc/en/Main/ArduinoBoardBluetooth*.

Arduino Pro 5V

Die auf dem Arduino Duemilanove angebrachten Steckverbindungen sind zwar praktisch, können aber auch hinderlich sein, wenn man zum Beispiel Drähte anlöten oder den Arduino in einem Gehäuse verbauen möchte, das möglichst dünn sein soll. Der Arduino Pro kommt in zwei Varianten (3,3 Volt und 5 Volt) daher und ist im Prinzip ein »nacktes« Arduino-Board. Der Atmega168 ist fest verlötet, die Pins sind nur als Kontaktstellen auf dem Board verfügbar. Das macht den Arduino Pro auch um etwa ein Drittel günstiger als den Duemilanove. Wer also professionellere Ansprüche hat und auf Luxus verzichten kann, mag mit einem Pro gut beraten sein. Einsteigern empfehlen wir trotzdem den Duemilanove. Mehr Informationen zum Arduino Pro finden Sie unter *http://arduino.cc/en/Main/ArduinoBoardPro*.

Arduino LilyPad

Das Lilypad ist speziell dafür ausgelegt, in Kleidung eingebettet zu werden. Es ist mit etwa 5 cm Durchmesser und nur 3 mm Dicke sehr klein und verfügt über keine Pins oder Teile, welche die umgebende Kleidung zerreißen könnten. Trotzdem verfügt das Board über 14 digitale (sechsmal PWM) und sechs analoge Pins. Mit 2,7–5,5 Volt Betriebsspannung ist es zudem geeignet, um mit kleinen Batterien betrieben zu werden. Das ist gerade bei Kleidung immens wichtig, weil niemand einen oder gar mehrere 9-Volt-Blöcke mit sich herumtragen möchte. Das LilyPad ist auf der Website des MIT Medialab genauer beschrieben, von dem es entwickelt wurde (*http://web.media.mit.edu/~leah/LilyPad/*).

Arduino Nano und Mini

Neben dem LilyPad gibt es weitere Boards, die für kleinere und kleinste Anwendungen geeignet sind: Beim Arduino Nano und dem Arduino Mini ist die Fläche des Boards minimiert: Der Nano ist 1,85 x 4,31 cm klein, der Mini sogar nur 1,8 x 3,3 cm. Dabei verfügen beide über dieselbe Anzahl digitaler Pins wie ihre großen Brüder und zusätzlich sogar über zwei analoge Pins mehr. Was beim Nano fehlt, ist ein zusätzlicher Stromanschluss, sodass er über den USB-Port betrieben werden muss. Der Mini verfügt sogar nur über einen seriellen Anschluss: Wer ihn also mit USB betreiben möchte, benötigt einen Adapter. Die Minivariante ist auch als Pro-Version (mit 3,3 und 5 Volt Betriebsspannung) verfügbar, um Arduino-Projekte zu ermöglichen, die noch kleiner sind und deren Board sich noch besser einfügt.

Weitere Boards

Die Menge der Arduino-Varianten ist riesig, wächst stetig und kann kaum erfasst werden. Die Änderungen reichen dabei von einfachen Layoutabweichungen und anderen Mikrocontroller-Varianten bis hin zu Boards, die für spezielle Einsatzgebiete wie etwa die Robotik angepasst sind.

Der Sanguino (*http://www.sanguino.cc*) wurde zum Beispiel entwickelt, um den RepRap anzutreiben, einen 3-D-Drucker. Ursprünglich hatte man dafür einen Arduino verwendet, musste dabei aber Kompromisse eingehen. Der Sanguino verwendet 32 digitale Pins

und hat seine eigene Bauform, die etwas kompakter und in die Länge gezogen ist.

Mit dem Roboduino (*http://www.curiousinventor.com/kits/roboduino*) soll es leicht möglich sein, die Motoren und anderen Bauteile von Robotern anzusteuern. Dafür verfügt das Board neben jedem digitalen Pin noch über zwei Pins für einen weiteren Stromkreis. Das soll vor allem das Anbringen erleichtern, da nun die anderen Bauteile über einfache Steckverbindungen angebunden werden können.

Weil der Arduino Duemilanove gerade für viele LEDs oder Matrizen nicht ausreicht, wurde der Illuminato (*http://www.liquidware.com/shop/show/ILL/Illuminato*) entwickelt. Er soll verhindern, dass jeder Pin auf dem Arduino extra mit Erweiterungsbauteilen gegabelt werden muss. Der Illuminato verfügt daher über 42 digitale Pins. Der Arduino Fio ist mit anderen Arduino-Boards vergleichbar, verfügt aber neben 8 analogen und 14 digitalen Pins auch zusätzlich über einen Xbee-Socket.

Weil sein Erfinder es leid war, sich mit den Steckverbindungen eines Arduino herumzuärgern, schuf er den Boarduino, dessen Bauform so entworfen ist, dass er genau auf ein Steckbrett passt. Das macht natürlich das Entwickeln von Prototypen besonders einfach. Mehr dazu finden Sie unter *http://www.adafruit.com/index.php?main_page=product_info&cPath=19&products_id=72*, wo es auch eine USB-Variante des Boards zu kaufen gibt.

Der Ardupilot wurde entwickelt, um alle wichtigen Funktionen einer autonomen Flugdrohne wahrnehmen zu können. Dieses spezielle Board hat deshalb Pins für Beschleunigungssensoren und ein GPS-Modul sowie die Kontrolle von Servos. Zudem besitzt das Board einen zweiten Stromkreis, der beim Ausfall des ersten aktiv wird. Das verhindert das Abstürzen des Flugobjektes. Mehr dazu finden Sie unter *http://diydrones.com/profiles/blog/show?id=705844%3ABlogPost%3A44814*.

Wie in Kapitel 10 ausführlich erklärt wurde, eignet sich der Arduino auch zur Verwendung als Midi-Controller. Über eine entsprechende Steckverbindung kann er an Midi-Geräten angebracht werden, um beispielsweise Ton über besondere Eingabemethoden zu steuern. Der Miduino wurde dafür geschaffen, mit genau diesen Midi-Buchsen zu arbeiten. Sie können ihn unter *http://tomscarff.110mb.com/miduino/miduino.htm* bestellen, wo auch die Schaltpläne und eine genaue Dokumentation verfügbar sind.

Sieht der Protoduino auf den ersten Blick heillos komplex aus, besteht er doch lediglich aus einer Leiterplatte, auf der nur die nötigsten Leiterbahnen miteinander verbunden sind, um einen Mikrocontroller, Stromversorgung und einen 16-MHz-Taktgeber anzubringen. Zudem verfügt er über ein Netzwerk von Widerständen für den schnellen Betrieb von LEDs. Der Protoduino ist vor allem dann geeignet, wenn man ein eigenes Board entwerfen und zunächst ausprobieren möchte. Dazu muss man Leiterbahnen selbstständig auflöten, hat aber maximale Freiheit in dem, was man tun möchte. Mehr Informationen dazu finden Sie unter *http://east-ham-lee.com/protoduino.html*.

Arduino-Shields

Möchte man viele Sensoren oder Aktoren an den Arduino anschließen, kann das zu komplexen Schaltungen führen, die möglicherweise auch nicht ganz stabil angebracht werden können. Dafür gibt es viele verschiedene sogenannte Shields, die auf die Pins des Arduino aufgesteckt werden können. Einige gängige Shields sollen hier kurz vorgestellt werden. Jeder Absatz enthält auch einen Link zu weiteren Informationen über den entsprechenden Shield. Viele davon sind leider nur als Import erhältlich, allerdings sollten Sie auch die deutschen Websites wie *http://www.bausteln.de* oder *http://www.tinkersoup.d*e im Auge behalten. Zudem werden auch bei herkömmlichen Elektronik-Webshops wie *http://www.elmicro.de* oder *http://www.segor.de* Shields angeboten.

Prototypen-Shields

Was der Protoduino für das Arduino-Board, ist der Arduino Proto Shield für die passenden Aufsätze: eine Platine, auf der lediglich die Pins des Arduino sowie zwei LEDs, der Reset-Knopf und ICSP-Header aufgebracht sind. Er kann zum Entwickeln von weiteren Shields verwendet werden und zum Beispiel ein Steckbrett einfassen. Das Shield ist unter *http://www.ladyada.net/make/pshield/* beschrieben.

Ähnlich verhält sich das Breadboard-Shield, auf dem bereits ein Steckbrett angebracht ist. Er ist deshalb nicht so flexibel wie der Proto Shield, aber er ist ja auch nur zum schnellen Ausprobieren und nicht zum Entwickeln gedacht (*http://todbot.com/blog/2006/07/11/arduino-breadboard-shield/*)

Arduino RepRap Shield

Das RepRap ist ein Gestell, das unter anderem als 3-D-Drucker verwendet werden kann. Um die nötigen Motoren, Düsen und anderen Teile anzuschließen, wurde der RepRap-Shield entwickelt. Er bietet robuste Schraubanschlüsse für den Arduino Duemilanove und ist deshalb auch für andere Anwendungen geeignet. Mehr Informationen bietet das Blog des RepRap, das Sie unter *http://blog.reprap.org/2008/04/new-board-arduino-breakout-v11.html* finden.

Propellurino

Der Propellurino steuert nicht etwa ausschließlich Propeller, wie der Name vielleicht vermuten lässt. Vielmehr verfügt er zusätzlich über Anschlüsse für einen VGA-Monitor, eine PS/2-Maus oder PS/2-Tastatur, Midi und ein Mikrofon. Zudem hat er zwei DAC-Ausgänge, um Ton auszugeben. Bastelanleitungen und andere Informationen finden Sie auf der deutschen Website *http://www.hobby-roboter.de/forum/viewtopic.php?f=5&t=72*.

Battery Shield

Möchte man den Arduino autonom betreiben, also ohne externe Stromquelle, kann man den Battery Shield verwenden. Er repliziert alle Pins des Arduino und hat in der Mitte eine Halterung für einen Lithium-Ionen-Akku. Je nach Akkutyp kann das Projekt dann viele Stunden betrieben werden, bevor es neu aufgeladen werden muss, was über den USB-Port möglich ist. Dieser Shield ist in Deutschland leider nicht erhältlich, kann aber aus den USA von *liquidware.com* bezogen werden (*http://www.liquidware.com/products/show/BP/Lithium+BackPack*).

Adafruit Ethernet Shield

Diese Ethernet-Shield wurde bereits recht ausführlich in Kapitel 6 erklärt: Er bietet die Möglichkeit, ein Netzwerkkabel anzuschließen und so den Arduino mit dem Internet oder einem Heimnetzwerk zu verbinden. Im Handel sind mehrere Ethernet-Shields erhältlich, zum Beispiel das von *ladyada.net*. Diese Shields sind weit verbreitet, sodass man sie auch in Deutschland leicht bestellen kann. Eine genauere Beschreibung finden Sie unter *http://www.ladyada.net/make/eshield/*.

Liquidware TouchShield

Auf dem TouchShield (*http://www.liquidware.com/shop/show/TS/TouchShield+Stealth*) ist ein Touchscreen mit 128 x 128 Pixeln angebracht, dessen Eingabedaten vom Arduino verwendet werden können. Auch der Monitor wird vom Arduino gesteuert, wobei entsprechende Bibliotheken für eine einfache Ansteuerung sorgen. Dieser Shield ist mit 140 US-Dollar teuer, was an den sehr vielfältigen Möglichkeiten liegt, die er bietet.

Drahtlose Kommunikation

Neben Bluetooth, für das man am besten einen Arduino Bluetooth verwendet, gibt es noch weitere drahtlose Übertragungsprotokolle. WLAN ist allerdings für die meisten Zwecke übertrieben, komplex und ein Stromfresser. Zigbee hingegen wird in vielen Projekten eingesetzt, weil es einfach ist und eine gute Reichweite hat. Dieses Protokoll wurde ursprünglich entwickelt, um Haushaltsgeräte zu steuern und Sensordaten zu übermitteln. Der Xbee-Shield bringt dieses Protokoll auf den Arduino und kann zum Beispiel verwendet werden, um Roboter oder Fluggeräte zu bedienen. Mehr Informationen darüber gibt es unter *http://www.arduino.cc/playground/Shields/Xbee01*.

Die Radio Frequency Identification, kurz RFID, findet ihren Weg in immer mehr Bereiche. Ein Lesegerät sendet dabei elektromagnetische Frequenzen aus, die einen stromlosen Schaltkreis anregen. Dieser emittiert dann darin gespeicherte Daten, um sich zu identifizieren. Je nach Bauart können so über mehrere Meter oder wenige Zentimeter einige Kilobytes an Informationen übermittelt werden. Marc Boon bietet auf seiner Website einen entsprechenden Shield an, den man auch selbst herstellen kann, wenn man möchte. Alle Informationen dazu finden Sie unter *http://rfid.marcboon.com/#category2*.

Musik-Shields

Mit dem Wave Shield (ebenfalls von *LadyAda.net*) können Sounddateien abgespielt werden. Tonausgabe auf dem Arduino ist recht kompliziert und begrenzt. Der Shield bietet Platz für eine SD-Karte, auf der unkomprimierte Wave-Dateien (also keine MP3s und ähnliche Formate) gespeichert werden können. Ein Soundchip sorgt für eine qualitativ hochwertige Ausgabe auf dem ebenfalls angebrach-

ten Kopfhöreranschluss. Mehr über den Wave Shield finden Sie unter *http://www.ladyada.net/make/waveshield*.

Eine weitere Möglichkeit, Musik zu machen, ist der SID-Chip, der schon dem Commodore 64 8-Bit-Klänge entlockte. Er ist immer noch sehr beliebt, weil er einen ganz eigenen Klang produziert, statt reale Musikinstrumente nachzuahmen. Im auf der Arduino-Website beschriebenen SID-Shield (*http://www.arduino.cc/playground/Main/SID-emulator*) arbeitet ein Atmega168-Mikrocontroller, der diesen SID-Chip emuliert und von einem Arduino gesteuert wird.

Der Arduinome ist kein Shield im eigentlichen Sinne, da er nicht für weitere Projekte verwendbar ist. Vielmehr wird er auf den Arduino aufgesteckt, um ein Monome zu emulieren, einen USB-Midi-Controller, der auf einer Matrix von Tasten basiert, die jeweils mit einer LED beleuchtet werden. Der Arduinome kann so an einen PC angeschlossen werden, um diese Eingabedaten zu verarbeiten, oder er erzeugt selbst weiteren Output. Das Arduinome-Projekt finden Sie unter *http://bricktable.wordpress.com/30/*.

Adafruit GPS Shield

Der GPS-Shield bietet die Möglichkeit, ein GPS-Modul zu montieren und die ermittelten Positionsdaten auf einer SD-Karte zu speichern. So lässt sich mit dem Arduino ein kompaktes Gerät entwickeln, das den Weg einer Wanderung oder einer Fahrt durch die Stadt speichert. Diese Daten können beispielsweise für das Projekt OpenStreetMap (*http://www.openstreetmap.org*) verwendet werden, das zum Ziel hat, eine frei verfügbare Datenbank mit Landkarten zur Verfügung zu stellen und so eine freie Alternative zu kommerziellen Anbietern wie Google Maps zu liefern. Mehr Informationen zu dem Shield finden Sie auf der LadyAda-Website (*http://www.ladyada.net/make/gpsshield*).

MicroSD Module

Der Speicher des Arduino ist vergleichsweise klein und eignet sich kaum dazu, über einen längeren Zeitraum Werte aufzuzeichnen. Glücklicherweise gibt es auch dafür eine Lösung: das MicroSD-Modul (*http://www.sensor-networks.org/index.php?page=0827727742*), das eine MicroSD-Karte fassen und beschreiben kann. Die kann man dann später über den Arduino oder einen Kartenleser auslesen.

DMX-shield

DMX ist ein Bussystem, mit dem in der Lichttechnik Lampen angesteuert werden. Es wird zum Beispiel bei Konzerten und in Diskotheken eingesetzt, um von einem Mischpult aus die Beleuchtung des Raumes oder der Bühne zu steuern. DMX steht für Digital Multiplex. Möchte man sein eigenes Mischpult oder andere Ansteuerungsmöglichkeiten (zum Beispiel über Sensoren) entwickeln, kann man auf den DMX-shield zurückgreifen. Das Shield wird unter anderem in Kapitel 8 beschrieben. Weitere Informationen finden sich auf der Arduino-Website unter *http://www.arduino.cc/playground/DMX/DMXShield*.

Eingabe-Shields

Eingabesensoren wie Schieberegler oder Joysticks benötigen keine komplexe Programmierung. Passende Shields bieten vor allem Fassungen, um eine möglichst einfache Montage zu erlauben. Der Danger Shield von Zach Hoeken (*http://www zachhoeken.com/danger-shield-v1-0*) ist so einer und verfügt unter anderem über drei Schieberegler und drei Taster. Zudem hat er allerlei andere Sensoren (z.B. einen Temperatur- und einen Klopfsensor) und Aktoren (wie ein Siebensegment-Display und einen Piezo-Lautsprecher). Er ist also quasi ein Allround-Shield, der für viele verschiedene Arduino-Projekte verwendet werden kann.

Ein anderer Shield ist der InputShield von LiquidWare (*http://www.liquidware.com/shop/show/INPT/InputShield*). Er eignet sich zum Beispiel dazu, einen Controller für eine Spielkonsole zu bauen, und verfügt über einen Joystick, zwei Knöpfe, einen Vibrationsmotor und ein serielles Interface. Damit kann man zum Beispiel eine kleine Handheld-Konsole bauen, deren Vorbild der Game Boy von Nintendo ist.

Motoren und Rotoren

Der Tank Shield von Liquidware (*http://www.liquidware.com/shop/show/TANK/TankShield*) ist mehr als nur ein Shield. Er verfügt über einen Kettenantrieb, mit dem man aus dem Arduino einen Panzer-Roboter bauen kann. Dazu verfügt er über zwei Motorenantriebe und eine steuerbare Achse. Dank seinem Design ist er in der Lage, nicht nur einen Arduino, sondern auch einige weitere Shields mit sich zu führen.

Der Adafruit Motor Shield erlaubt den Anschluss mehrerer Motoren. So enthält er Anschlüsse für zwei 5-Volt-Servos, vier Gleichstrommotoren oder zwei Schrittmotoren. Mehr Informationen gibt es unter *http://www.ladyada.net/make/mshield/*.

ANHANG B
Arduino-Bibliotheken

In diesem Kapitel:

- EEPROM-Bibliothek: Werte langfristig speichern
- Ethernet-Bibliothek: mit dem Internet kommunizieren
- Firmata-Bibliothek
- LiquidCrystal-Bibliothek
- Servo-Bibliothek
- Debounce-Bibliothek
- Wire-Bibliothek
- capSense-Bibliothek

Die Arduino-Umgebung bringt eine ganze Reihe nützlicher Bibliotheken (libraries) mit, die hier kurz vorgestellt werden. Weiterhin haben viele Benutzer von Arduino selbst Bibliotheken geschrieben und online gestellt. Diese Libraries bieten oft eine angenehme und einfache Programmieroberfläche für spezielle Bauteile, Sensoren und Aktoren sowie für spezifische Arduino-Shields. Eine Liste der verschiedenen Arduino-Bibliotheken können Sie auf der Arduino-Website unter *http://www.arduino.cc/en/Reference/Libraries* finden.

Eine Arduino-Bibliothek besteht aus einer Sammlung von C- und C++-Quellcodedateien, die in einem separaten Verzeichnis gespeichert werden. Eine dieser Dateien ist eine sogenannte Header-Datei, die nicht die Funktionalität und den Quellcode an sich enthält, sondern die Oberfläche beschreibt, die der Benutzer der Bibliothek in seinem eigenen Programm aufrufen kann. Eine solche Oberfläche nennt man oft auch API (application programming interface).

Bibliotheken werden meistens als Archivdatei im Netz zum Herunterladen angeboten. Diese Datei enthält ein Verzeichnis, das wie die Bibliothek heißt und in den Ordner *hardware/libraries* der Arduino-Entwicklungsumgebung kopiert wird. Anschließend muss die Header-Datei der Bibliothek in einen Sketch eingebunden werden, indem am Anfang der Sketch-Datei die Zeile `#include <headerdatei.h>` geschrieben wird. Die Arduino-Programmierumgebung erkennt diese Zeile, sucht nach der genannten Datei in ihrem Bibliotheksordner und übersetzt die Dateien der Bibliothek zusammen mit den restlichen Dateien des Sketches. Wenn die Arduino-Umgebung gestartet wird, indiziert diese die verfügbaren Bibliotheken und bietet sie unter SKETCH → IMPORT LIBRARIES an. Wenn dort eine Bibli-

othek ausgewählt wird, ergänzt die Entwicklungsumgebung den aktuellen Sketch automatisch um die Zeile `#include`.

Für fortgeschrittene Benutzer wird auf der Arduino-Website unter *http://arduino.cc/en/Hacking/LibraryTutorial* im Detail erklärt, wie man eine eigene Bibliothek erstellt.

EEPROM-Bibliothek: Werte langfristig speichern

Wie in Kapitel 6 beschrieben wird, kann der Arduino-Prozessor Werte langfristig in seinem internen EEPROM-Speicher festhalten. Um Werte aus dem EEPROM zu lesen und zu speichern, bietet Arduino die EEPROM-Bibliothek. Sie muss mit der Zeile `#include <EEPROM.h>` eingebunden werden.

EEPROM.read(adress)

Mit dieser Funktion kann ein Byte an der angegebenen Adresse aus dem EEPROM-Speicher gelesen werden. Mögliche Adresswerte gehen von 0 bis 511. Werte, die noch nie beschrieben worden sind, sind im Arduino-Prozessor auf 255 initialisiert. Man sollte im Auge behalten, dass die geschriebenen Werte auch nach dem Ausstecken des Arduino erhalten bleiben, im Speicher also durchaus noch Werte von anderen Programmen stehen können.

EEPROM.write(adress, value)

Mit dieser Funktion kann ein Bytewert an der angegebenen Adresse in den EEPROM-Speicher geschrieben werden. Auch hier sollte man bedenken, dass die Adresse vielleicht auch von einem anderen Sketch verwendet wird, um Daten zu speichern. Gehen Sie beim Schreiben also vorsichtig vor, wenn noch gespeicherte Daten von anderen Sketchen verwendet werden sollen.

Beispiel

```
void setup() {
    Serial.begin(9600);
    EEPROM.write(0, 15); // 15 an Adresse 0 des EEPROM-Speichers
                         // schreiben
    byte daten = EEPROM.read(0); // Adresse 0 lesen
    Serial.println(daten);    // und an den Computer schicken
}
void loop() {
}
```

Ethernet-Bibliothek: mit dem Internet kommunizieren

Die Ethernet-Bibliothek wird zusammen mit einem Ethernet-Shield eingesetzt. Sie implementiert zum einen eine einfache Serverfunktionalität, die es einem Arduino ermöglicht, verschiedene Serverdienste wie z.B. einen Webserver oder einen Telnet-Server anzubieten, und zum anderen eine Clientfunktionalität, mit der der Arduino mit anderen Servern im Internet kommunizieren kann. Die Ethernet-Bibliothek ist deswegen in einen Server- und einen Clientteil untergliedert. Bevor einer dieser beiden Bereiche verwendet werden kann, muss die Bibliothek initialisiert werden.

Ethernet.begin(mac, ip) / Ethernet.begin(mac, ip, gateway) / Ethernet.begin(mac, ip, gateway, subnet)

Mit dieser Funktion werden der Ethernet-Chip auf dem Ethernet-Shield initialisiert und die Ethernet-Adresse und die IP-Adresse des Arduino festgelegt. Diese Funktion muss aufgerufen werden, bevor die Server- und/oder die Client-Funktionalität der Ethernet-Library verwendet wird. Zusätzlich können der IP-Funktion der Ethernet-Bibliothek noch die Adresse der Default-Gateway angegeben werden (also des Routers, der die Schnittstelle des lokalen Netzwerks zum Internet ist, z.B. der Router, der vom Provider zur Verfügung gestellt wird) sowie die Netzmaske des lokalen Netzwerks. Standardmäßig wird die Gateway-Adresse auf die IP-Adresse des Arduino gesetzt, mit einer 1 als letzter Zahl. Die Default-Netzwerkmaske ist *255.255.255.0*. Die Mac-Adresse ist eine Tabelle von sechs Bytes, während die IP-Adresse und die Gateway-Adresse jeweils eine Tabelle von vier Bytes sind.

Beispiel

```
#include <Ethernet.h>
byte mac[] = { 0xde, 0xad, 0xbe, 0xef, 0xfe, 0xed };
byte ip[] = {10, 0, 0, 27};
void setup() {
    Ethernet.begin(mac, ip);
}
void loop() {}
```

Serverfunktionalität

Mit der Serverfunktionalität der Ethernet-Bibliothek kann der Arduino einen Internetdienst anbieten. Dieser wird mit durch ein

Serverobjekt erstellt, dem der Port mitgegeben wird, auf dem dieser Dienst lauscht. Dieses Serverobjekt kann dann benutzt werden, um Verbindungen anzunehmen und Daten von verbundenen Clients zu lesen und zu schreiben.

Server(port)

Mit dieser Funktion wird ein Serverobjekt erzeugt, das auf dem angegebenem Port lauscht. Standardwerte für `port` sind z.B. `23` für einen Telnetdienst, `25` für einen Mailserver, und `80` für einen Webserver.

Server.begin()

Nachdem `Server.begin()` aufgerufen wurde, kann ein Serverobjekt Verbindungen annehmen.

Server.available()

Diese Funktion überprüft, ob ein Serverobjekt eine Verbindung angenommen hat. Wenn das der Fall ist, gibt sie ein Clientobjekt zurück, das benutzt werden kann, um mit dem Gegenpunkt der Verbindung zu kommunizieren. Es besteht kein Unterschied zwischen einem Clientobjekt, das als Folge einer Serververbindung erzeugt wurde, und einem Clientobjekt, das erzeugt wurde, weil der Arduino sich mit einem Server verbunden hat (siehe unten). Wenn keine Verbindung angenommen wurde, gibt `Server.available()` NULL zurück.

Server.write(data)

Mit dieser Funktion können Daten an alle verbundenen Gegenstellen des Serverobjekts geschrieben werden. `data` ist dabei ein Byte oder ein Buchstabe.

Server.print(value) / Server.print(value, base)

Mit dieser Funktion können ähnlich wie mit `Serial.print` (siehe Kapitel 5) beliebige Werte formatiert an alle verbundenen Gegenstellen gesendet werden. `value` kann dabei wie bei `Serial.print` ein beliebiger numerischer Wert sein (allerdings keine Fließkommazahl) oder ein Zeichenkette. Zusätzlich kann auch noch die Darstellungsbasis angegeben werden.

Server.println(value) / Server.println(value, base)

Mit dieser Funktion können ähnlich wie mit `Serial.println` (siehe Kapitel 5) beliebige Werte formatiert an alle verbundenen Gegenstellen gesendet werden. Der gesendete Wert wird von einer Zeileneinrückung gefolgt.

Beispiel

```
#include <Ethernet.h>
byte mac[] = { 0xde, 0xad, 0xbe, 0xef, 0xfe, 0xed };
byte ip[] = {10, 0, 0, 27};
Server server(23);
void setup() {
    Ethernet.begin(mac, ip);
    server.begin();
}
void loop() {
    Client client = server.available();
    if (client) {
      server.write(client.read());
    }
}
```

Clientfunktionalität

Mit der Clientfunktionalität der Ethernet-Bibliothek kann der Arduino sich mit einem Internetdienst verbinden. Weiterhin wird die Clientfunktionalität verwendet, wenn eine externe Gegenstelle sich mit einem Server verbindet, der auf dem Arduino läuft. Man kann daher die Clientfunktionalität auch als »Verbindungsfunktionalität« betrachten. Eine Verbindung wird als ein Clientobjekt dargestellt, das entweder von Hand erzeugt wird, um mit einem externen Server zu kommunizieren (in diesem Fall werden Adresse und Port des Dienstes auf diesem Server angegeben), oder beim Aufruf von `Server.available()` (siehe oben).

Nachdem der Arduino sich erfolgreich mit der Gegenseite verbunden hat (über einen Aufruf von `Client.connect()` oder weil das Clientobjekt von einem auf dem Arduino laufenden Server erzeugt wurde), können mit dem Clientobjekt Daten an die Gegenstelle geschickt und von ihr empfangen werden.

Client(ip, port)

Mit dieser Funktion wird ein Clientobjekt erzeugt, das mit dem angegebenen Dienst verbunden wird. Die IP-Adresse wird wie gehabt als Tabelle von vier Bytes angegeben, und der Port als Integer-Wert.

Client.connected()

Diese Funktion gibt zurück, ob das benutzte Clientobjekt mit der Gegenstelle verbunden ist (also TRUE, wenn eine laufende TCP-IP Verbindung besteht, sonst FALSE). Diese Funktion gibt auch TRUE zurück, wenn die Verbindung schon geschlossen wurde, aber noch ungelesene Daten existieren.

Client.connect()

Wenn diese Funktion aufgerufen wird, versucht der Ethernet-Shield sich mit der Gegenstelle zu verbinden. Gelingt das, gibt die Funktion TRUE zurück, kann keine Verbindung erstellt werden, FALSE.

Client.write(data)

Mit dieser Funktion können Daten an alle verbundenen Gegenstellen des Clientobjekts geschrieben werden. `data` ist dabei ein Byte oder ein Buchstabe.

Client.print(value), Client.print(value, base)

Diese Funktion ähnelt `Server.print()`, außer dass Daten nur an die Gegenstelle der Verbindung geschickt werden.

Client.println(value), Client.println(value, base)

Diese Funktion ähnelt `Server.println()`, außer dass Daten nur an die Gegenstelle der Verbindung geschickt werden.

Client.available()

Ähnlich wie bei der seriellen Schnittstelle werden bei einer TCP-Verbindung die empfangenen Daten auf dem Arduino gepuffert. Mit dieser Funktion kann abgefragt werden, ob sich Daten in diesem Zwischenpuffer befinden, die dann später mit `Client.read()` ausgelesen werden können. Diese Funktion gibt die Anzahl der vorhandenen Bytes zurück.

Client.read()

Mit dieser Funktion kann ein einzelnes Byte aus dem Puffer ausgelesen werden. Mit `Client.available()` kann zuerst überprüft werden, ob überhaupt Bytes im Puffer vorhanden sind. Sind keine Bytes vorhanden, gibt `Client.read()` `-1` zurück.

Client.flush()

Mit dieser Funktion kann der Puffer der Verbindung geleert werden (es werden also alle ungelesenen Bytes gelöscht).

Client.stop()

Mit dieser Funktion wird die TCP-Verbindung mit der Gegenstelle getrennt.

Beispiel

```
#include <Ethernet.h>
byte mac[] = { 0xde, 0xad, 0xbe, 0xef, 0xfe, 0xed };
byte ip[] = {10, 0, 0, 27};
byte server[] = {64, 233, 187, 99 }; // Google
Client client(server, 80); // Verbindung zum Google-Webserver
void setup() {
   Ethernet.begin(mac, ip);
   Serial.begin(9600);
   server.begin();
   if (client.connect()) {
     Serial.println(»Mit google verbunden");
     client.println("GET /search?q=arduino HTTP/1.0"); // Suchkette
                                                      // verschicken
     client.println();
   } else {
     Serial.println("Verbindung erfolglos");
   }
}
void loop() {
   if (client.available()) {
     char c = client.read();
     Serial.print(c);
   }
   if (!client.connected()) {
     Serial.println("Verbindung getrennt");
     client.stop();
     for (;;) ; // Leere Schleife
   }
}
```

Firmata-Bibliothek

Die in Kapitel 5 vorgestellte Firmata-Bibliothek ist ein nützliches Werkzeug, um einen Arduino mit einem normalen Desktopcomputer zu verbinden (unter anderem mit Programmen, die in Processing oder Max/MSP geschrieben sind). Mit Firmata können über die serielle Schnittstelle viele Befehle an den Arduino gesendet und Informationen vom Arduino empfangen werden (z.B. der Status

einzelner Pins, aber auch beliebige Zeichenketten und numerische Werte). Die Bibliothek ist eine sehr große und etwas komplexe Bibliothek, mit der die Firmata-Funktionalität in normale Sketche mit eingebunden werden kann (im Vergleich zur »normalen« Firmata-Firmware, die nur die Standardfunktionalität des Protokolls unterstützt). Hier sei auf die ausführliche Dokumentation des Firmata-Protokolls und der Firmata-Bibliothek im Netz verwiesen, die viele nützliche Beispiele über ihre Verwendung gibt. Die Dokumentation finden Sie unter *http://www.arduino.cc/playground/Interfacing/Firmata*.

LiquidCrystal-Bibliothek

Mit der Bibliothek LiquidCrystal kann ein LCD-Bildschirm, der auf einem Hitachi HD44780 oder einem seiner zahlreichen Klone aufgebaut ist, angesprochen werden. Es gibt viele verschiedene Varianten dieser LCD-Bildschirme, die kostengünstig bei Vertrieben wie *http://conrad.de/*, *http://reichelt.de/* oder *http://pollin.de* bestellt werden können. Diese LCD-Bildschirme können Buchstaben anzeigen und haben meistens Größen wie 2 × 16 (zwei Zeilen mit je 16 Buchstaben), 1 × 16, 2 × 40, 4 × 16 usw. Da es viele HD44780-kompatible Chips gibt, kann es oft zu leichten Unterschieden zwischen den einzelnen Displays kommen. Oft sind gerade kleine Zeitunterschiede bei der Initialisierung oder beim Senden von Daten wichtig, weshalb manchmal eine Anpassung der Sketche notwendig ist. Um die genauen Zeiten herauszubekommen, müssen Sie im Datenblatt des Chips nachsehen, wie lange z. B. Daten anliegen müssen oder wie lange nach bestimmten Kommandos gewartet werden muss. Diese Zeiten kann man dann mit `delayMicroseconds()` einbauen. Auch die Pinbelegung der Displays kann verschieden sein, besonders der Anschluss der Rückbeleuchtung variiert. Oft ist es notwendig, die Rückbeleuchtung über einen Transistor zur Stromverstärkung anzuschließen.

Displays können in zwei verschiedenen Modi angesteuert werden: im 4-Bit-Modus oder im 8-Bit-Modus. Beim 8-Bit-Anschluss werden zwar mehr digitale Output-Pins am Arduino benutzt, die Datenübertragung ist dafür aber deutlich schneller, sodass zum Beispiel flüssigere Animationen auf dem Display dargestellt werden können. Im 4-Bit-Modus werden vier Datenleitungen (*d0* bis *d3*) weniger benutzt.

LiquidCrystal(rs, rw, enable, d4, d5, d6, d7) / LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7)

Mit dieser Funktion, die eine variable Anzahl an Parametern akzeptiert, wird ein LiquidCrystal-Objekt erzeugt, mit dem ein Display gesteuert werden kann. Die einzelnen Leitungen des Displays müssen mit digitalen Pins am Arduino verbunden werden. Die Pinnummern der einzelnen Leitungen werden an die LiquidCrystal-Funktion übergeben. Werden *d0* bis *d3* angegeben, wird das Display im 8-Bit-Modus konfiguriert, ansonsten im 4-Bit-Modus. Mit dem erzeugten Objekt kann das Display dann angesteuert werden.

LiquidCrystal.clear()

Mit dieser Funktion werden das LCD-Display gelöscht und der Kursor nach links oben zurückgesetzt.

LiquidCrystal.home()

Mit dieser Funktion wird der Kursor des LCD-Displays in die linke obere Ecke bewegt. Ab dieser Position werden dann weitere Ausgaben auf dem Display angezeigt.

LiquidCrystal.setCursor(col, row)

Mit dieser Funktion kann der Kursor frei auf dem LCD-Display bewegt werden. `col` bezeichnet dabei die Spalte und `row` die Zeile, in die der Kursor bewegt werden soll. `0` ist dabei die erste Spalte bzw. Zeile.

LiquidCrystal.write(data)

Mit dieser Funktion kann ein einzelner Buchstabe auf dem Display angezeigt werden.

LiquidCrystal.print(value), LiquidCrystal.print(value, base)

Ähnlich wie bei der Funktion `Serial.print()` können mit `LiquidCrystal.print()` Zeichenketten und numerische Werte auf dem Display angezeigt werden.

Beispiel

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 10, 5, 4, 3, 2); // LCD im 4-Bit-Modus
void setup() {
   lcd.print("HALLO WELT");
}
void loop() {}
```

Servo-Bibliothek

Mit dieser Bibliothek können ein oder zwei Servomotoren an den Arduino angeschlossen und gesteuert werden, ohne dass anderer Programmcode dadurch beeinflusst wird (die Steuerung läuft parallel in einer Interrupt-Routine). Dazu werden die zwei PWM-Pins 9 und 10 benutzt. Wird nur ein Servo angeschlossen, kann der andere Pin nicht als PWM-Ausgang benutzt werden (man kann die Funktion `analogWrite` also nicht auf ihn anwenden). Wie in Kapitel 6 beschrieben wurde, schließt man einen Servomotor an GND, VCC und Pin 9 oder 10 an. Um einen Servomotor zu kontrollieren, muss ein Objekt des Typs *Servo* erzeugt werden, der dann mit der Funktion `attach` vor der Benutzung konfiguriert wird.

Servo.attach(pin), Servo.attach(pin, min, max)

Mit dieser Funktion kann ein Servo-Objekt mit einem der beiden Servo-Pins (9 und 10) verbunden werden. Optional kann die Pulsbreite in Millisekunden (ms) angegeben werden, die für den kleinsten Winkel (0 Grad am Servo) und den größten Winkel (180 Grad) benutzt wird. Als Standardwerte werden 544 ms für den kleinsten Winkel und 2.400 ms für den größten Winkel verwendet.

Servo.write(winkel)

Mit dieser Funktion kann der Servo kontrolliert werden. Auf einem normalen Servo wird so der Winkel eingestellt (von 0 bis 180), auf einem kontinuierlichen Servo wird die Geschwindigkeit des Servos eingestellt. (In diesem Fall ist ein Winkel von 0 die volle Geschwindigkeit in die eine Richtung und 180 die volle Geschwindigkeit in die andere Richtung, und ein ungefährer Wert von 90 die Position, an der sich das Servo nicht bewegt.)

Servo.read()

Mit dieser Funktion kann der zuletzt eingestellte Winkel ausgelesen werden. Sie gibt einen Wert von 0 bis 180 zurück.

Servo.attached()

Mit dieser Funktion kann überprüft werden, ob das Servo-Objekt schon mit einem Pin verbunden wurde. Sie gibt TRUE zurück, wenn das Objekt schon verbunden wurde, sonst FALSE.

Servo.detach()

Mit dieser Funktion kann das Servo-Objekt von seinem Pin getrennt werden. Wurden alle Servo-Objekte von ihren Pins getrennt, können Pin 9 und 10 wieder als normale PWM-Pins verwendet werden.

Beispiel

```
#include <Servo.h>
Servo meinServo; // Servo-Objekt
int pos = 0; // Variable zum Speichern der Servoposition
void setup() {
   meinServo.attach(9); // Der Servo ist an Pin 9 angeschlossen.
}
void loop() {
   // In dieser Schleife wird der Servo komplett von links nach
   // rechts bewegt.
   for (pos = 0; pos < 180; pos++) {
     meinServo.write(pos);
     delay(15);
   }
   // und wieder zurück
   for (pos = 180; pos >= 1; pos–) {
     meinServo.write(pos);
     delay(15);
   }
}
```

Debounce-Bibliothek

Wie in Kapitel 3 beschrieben wurde, können bei Tastern und Schaltern schnell Fehler auftreten. Weil sie auf federnden Effekten basieren, öffnen und schließen sie sich beim Betätigen mehrmals, am Pin kommen also mehrere HIGH- und LOW-Signale an, was ein unerwünschter Effekt ist. Dieses Phänomen wird als »Prellen« oder »Bouncing« bezeichnet, weshalb die Bibliothek, die dagegen helfen soll, »Debounce« heißt. Sie sorgt dafür, dass der Input-Pin nach einer Eingabe für eine bestimmte Anzahl Millisekunden gesperrt wird, sodass alle weiteren Signale ausgesperrt werden.

Debounce(unsigned long debounceZeit byte pin)

Instanziiert ein Debounce-Objekt. Es wird eine Debounce-Zeit in Millisekunden festgelegt und das Objekt mit einem Input-Pin verbunden.

Debounce.update()

Liest den Pin neu ein und prüft, ob sich das Signal unter Berücksichtigung der Debounce-Zeit geändert hat. Ist das der Fall, wird TRUE zurückgegeben, ansonsten FALSE.

Debounce.interval(unsigned long debounceZeit)

Ändert die Debounce-Zeit, die dem Objekt zugewiesen wird.

Debounce.read()

Liest den aktuellen Wert des mit dem Objekt verbundenen Pins ein und berücksichtigt dabei die Debounce-Zeit.

Debounce.write()

Schreibt den aktuellen Debounce-Wert in den Speicher des Objekts und auf den mit dem Objekt verbundenen Pin.

Beispiel

```
// schalte eine LED durch Betätigung eines Schalters an und aus
#include <Debounce.h>
#define SCHALTER 5
#define LED 13
// instanziiere ein Debounce-Objekt mit einer Debounce-Zeit von 20 ms
Debounce debouncer = Debounce( 20 , SWITCH );
void setup() {
  pinMode(SWITCH,INPUT);
  pinMode(LED,OUTPUT);
}
void loop() {
 // update den Debouncer
  debouncer.update ( );
 // verwende den neuen Wert, um die LED damit zu schalten
   digitalWrite(LED, debouncer.read() );
}
```

Wire-Bibliothek

Um mit dem Arduino über das Two-Wire-Interface bzw. das Protokoll I²C (inter-integrated circuit) zu kommunizieren, verwendet man die Wire-Bibliothek. Das I²C-Protokoll ist ein sogenannter Datenbus. Das heißt, dass auf einer Leitung mehrere Geräte angeschlossen werden können. Ein Hauptgerät, der *Master*, sendet Daten, während die *Slaves* diese empfangen und darauf reagieren. Die Wire-Bibliothek erlaubt, sowohl als Master- als auch als Slave-Gerät an einem TWI/I²C-Bus teilzunehmen. Der Bus basiert dabei

auf zwei Leitungen, einer Datenleitung (SDA), die auf dem analogen Input-Pin 4 angeschlossen wird, sowie einer Taktleitung (SCL) auf dem analogen Pin 5. Auf dem Arduino Mega werden die Leitungen auf Pin 20 (SDA) und 21 (SCL) angeschlossen.

Wire.begin() /Wire.begin(address)

Meldet den Arduino am Bus an. Wird eine Adresse (7 Bit, 0–127) mitgegeben, geschieht diese Anmeldung als Slave, ohne Adresse als Master.

Wire.requestFrom(address, quantity)

Fordert von einem Gerät Informationen an. `address` bezeichnet dabei die 7-Bit-Geräteadresse, `quantity` die Anzahl von Bytes, die erwartet werden.

Wire.available()

Gibt die Anzahl von Bytes zurück, die auf dem Bus gelesen werden können. Wenn der Arduino als Slave registriert ist, kann er so Daten vom Master empfangen, andersherum wird festgestellt, wie viele Bytes der Slave nach einer `requestFrom`-Anfrage gesendet hat.

Wire.receive()

Empfängt das nächste Byte, das auf dem Bus für das Gerät vorliegt und gibt dieses als »Byte« zurück.

Wire.beginTransmission(address)

Beginnt eine Übertragung als Master an die in `address` angegebene Slave-Adresse.

Wire.send(value) / Wire.send(string) / Wire.send(data, quantity)

Speichert Daten in einer Warteschleife, die dann an eine Slave-Adresse gesendet werden. `Wire.send()` wird nach `wire.beginTransmission` aufgerufen. Die Daten können dabei entweder ein Bytewert (`value`), ein `string` oder ein Array von Bytes (`data`) sein, wobei im letzten Fall zusätzlich die Anzahl von Bytes übergeben wird.

Wire.endTransmission()

Beendet den Übertragungsblock, der mit `beginTransmission()` gestartet und mit `send()` vorbereitet wurde. Mit `endTransmission()` wird der gesamte Block übertragen.

Wire.onReceive(handler)

Mit dieser Funktion kann eine weitere Funktion festgelegt werden, die aufgerufen werden soll, wenn ein Slave Daten vom Master erhält. Dabei wird der Funktionsname übergeben, die Funktion selbst wird im weiteren Verlauf des Sketches deklariert. Sie sollte genau einen `Int`-Parameter besitzen, an den die Anzahl der übertragenen Bytes übergeben wird.

Wire.onRequest(handler)

Diese Funktion verhält sich wie `onReceive()`, allerdings für den Fall, dass der Master Daten anfordert. Die mitgegebene Funktion hat weder einen Rückgabewert noch Parameter. Sie wird bei einem Request aufgerufen und kann zum Beispiel die Anweisung enthalten, einen Sensor auszulesen und mit `write()` wieder an den Master zu senden.

Beispiel

```
// schließt den Arduino als Master an und sendet eine Anfrage an
// einen Slave mit Adresse 1; dann werden die Antwort gelesen und
// der Inhalt wieder an den Slave gesendet
#include <Wire.h>
void setup() {
    // initialisiere die Bus-Verbindung
    Wire.begin();
}
void loop() {
    // fordere ein Byte von Gerät 1 an
    Wire.requestFrom(1, 1);
    if (Wire.available()) {
byte empfangeneDaten = Wire.receive();
        Wire.beginTransmission(1);
        Wire.send(empfangeneDaten);
        Wire.endTransmission();
    }
}
```

capSense-Bibliothek

Mit dieser Bibliothek können auf dem Arduino einfach kapazitive Sensoren implementiert werden. Wie in Kapitel 7 beschrieben wurde, sind dazu nur ein Widerstand (von 100.000–50 Mio. Ohm) und ein Stück Draht oder Aluminiumfolie notwendig. Der Widerstand muss zwischen zwei Pins gesteckt werden, bei dem der eine Pin als Sendepin fungiert und der andere als Empfangspin. An den Empfangspin muss der Draht oder die Alufolie als Antenne angeschlossen werden. Für einen kapazitiven Sensor wird die CapSen-

sor-Bibliothek verwendet, um ein CapSense-Objekt zu erzeugen, mit dem der Wert des Sensors (also die Entfernung zum menschlichen Körper) gemessen werden kann.

Ein Widerstand von einem Megaohm oder weniger kann eingesetzt werden, damit der Sensor nur bei tatsächlicher Berührung aktiviert wird; mit einem Widerstand von ungefähr 10 Megaohm wird der Sensor bei einer Entfernung von 10–15 cm aktiviert; mit einem Widerstand von 40 Megaohm wird der Sensor bei einer Entfernung von 30–40 cm aktiviert. 40-Megaohm-Widerstände sind nicht so leicht zu finden, weshalb man auch vier in Serie gelötete 10-Megaohm-Widerstände einsetzen kann.

CapSense(sendePin, empfangsPin)

Mit dieser Funktion wird ein CapSense-Objekt erzeugt, mit dem ein kapazitiver Sensor gemessen werden kann. Der Widerstand muss zwischen Sende- und Empfangspin gesteckt werden (beides müssen digitale Pins sein) und die Antenne an den Empfangspin angeschlossen sein.

CapSense.capSenseRaw(samples)

Mit dieser Funktion kann die Kapazität des Sensors gemessen werden, in einer beliebigen Einheit. Der `samples`-Parameter gibt die Zeit an, über die gemessen wird. Je länger diese Zeit und damit der Messvorgang ist, desto genauer der gemessene Wert. Das Ergebnis wird nicht durch die Zeitdauer geteilt, sondern es wird einfach der rohe Messwert zurückgegeben. Die Funktion gibt -2 zurück, wenn der gemessene Wert über `CS_Timeout_millis` liegt (das mit der Funktion `set_CS_Timeout_Millis` eingestellt werden kann und standardmäßig 2.000 Millisekunden beträgt).

CapSense.capSense(samples)

Ähnlich wie die `CapSense.capSenseRaw`-Funktion misst diese Funktion die Kapazität des Sensors über einen gewissen Zeitrahmen. Allerdings misst `CapSense.capSense` auch die Kapazität im nicht aktivierten Zustand und zieht diese vom gemessenen Wert ab. `CapSense.capSense()` sollte deswegen im nicht aktivierten Zustand einen niedrigen Wert zurückgeben. Die automatische Kalibrierung wird in regelmäßigen Intervallen ausgeführt, die mit der Funktion `set_CS_AutocaL_Millis` eingestellt werden können. Die Autokalibrierung wird normalerweise alle 20 Sekunden ausgeführt. Diese Funktion gibt -2 zurück, wenn der gemessen Wert über `CS_Timeout_millis` liegt.

CapSense.set_CS_Timeout_Millis(millis)

Mit dieser Funktion wird die maximale Messzeit eingestellt.

CapSense.reset_CS_AutoCal()

Mit dieser Funktion wird sofort eine Kalibrierung der Kapazität im nicht aktivierten Zustand ausgelöst.

CapSense.set_CS_AutoCaL_Millis(millis)

Mit dieser Funktion kann die Periode der automatischen Kalibrierung eingestellt werden. Die automatische Kalibrierung kann deaktiviert werden, indem `millis` auf `0xFFFFFFFF` gesetzt wird.

Beispiel

```
#include <CapSense.h>
CapSense cs = CapSense(4, 2); // Widerstand zwischen Pin 4 und 2
                              // 2 ist der Empfangspin
CapSense cs2 = CapSense(4, 3); // ein weiterer Sensor an
                               // Empfangspin 3
void setup() {
   cs.set_CS_AutocaL_Millis(0xFFFFFFFF);
   Serial.begin(9600);
}
void loop() {
   long start = millis(); // Messdauer festhalten
   long wert1 = cs.capSense(30); // ersten Sensor messen
   long wert2 = cs2.capSense(30); // zweiten Sensor messen
   // Messzeit und Messwerte ausgeben
   Serial.print(millis() - start);
   Serial.print(": ");
   Serial.print(wert1);
   Serial.print(", ");
   Serial.println(wert2);
   delay(10);
}
```

ANHANG C

Sprachreferenz

In diesem Kapitel:

Übersicht: Programmiersprachen

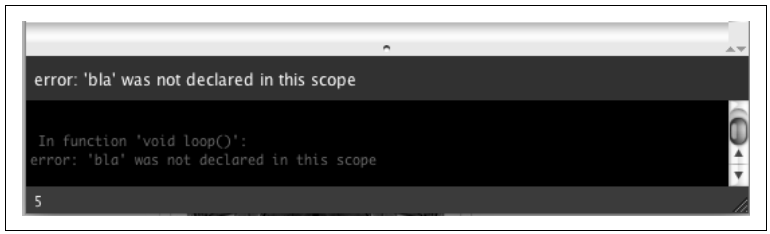
In diesem Anhang wird die Arduino-Programmiersprache ausführlich erklärt. Es werden zuerst die Syntax der Programmiersprache (also was in der Sprache zulässig ist) und die Struktur eines Arduino-Sketches erklärt (notwendige Sketch-Funktionen, Programmflusskonstrukte, mathematische Operatoren). Anschließend werden häufig benutzte Funktionen zur Steuerung der Arduino-Hardware gezeigt. An vielen Stellen wird auf potenzielle Fehlerquellen hingewiesen (Programmieren ist nicht gerade die leichteste Aufgabe). Das soll auf keinen Fall der Abschreckung dienen, sondern eher die Fehlersuche erleichtern und Ihnen generell ein Gefühl für mögliche Fallen geben. Es kann praktisch wenig passieren, außer dass der Arduino sich merkwürdig benimmt oder stehen bleibt. Es reichen also bei einem Programmfehler meistens ein Reset, ein bisschen Nachdenken und das erneute Hochladen der korrigierten Software.

Ein Arduino-Programm ist eine Folge von Programmierbefehlen, die dem Arduino-Prozessor Anweisungen dafür geben, welche Berechnungen er auszuführen hat. Eine Folge von Anweisungen, die eine bestimmte Aufgabe erfüllen, nennt man Algorithmus. Diese Anweisungen können rein mathematische Anweisungen sein (»nimm den Wert, der an dieser Stelle im Speicher steht, addiere 3 und speichere ihn an dieser anderen Stelle«) oder auch Programmflussanweisungen (»wenn der Eingabewert größer als 5 ist, dann führe diese Anweisungen aus, ansonsten führe diese anderen Anweisungen aus«). Weiterhin gibt es eine Reihe von Strukturanweisungen, die es ermöglichen, bestimmte Teile eines Programms

flexibel zu gestalten und auch getrennt zu bearbeiten. Die Programmierbefehle, die diese Anweisungen beschreiben, werden vom Programmierer in einer sogenannten Programmiersprache oder Computersprache (also nicht auf Englisch oder Deutsch) in eine Textdatei geschrieben.

Eine Computersprache braucht eine feste Syntax, damit der Computer verstehen kann, was mit dem Programm bewirkt werden soll. Beim Hochladen eines Sketches auf den Arduino wird der Programmtext in Maschinencode übersetzt. Dieser Schritt heißt »kompilieren« und wird von einem eigenen Programm namens »Compiler« durchgeführt. Falls Arduino in diesem Schritt einen Programmfehler antrifft (z.B. einen Tippfehler), wird der Vorgang abgebrochen und der Fehler in der Konsole angezeigt. Der Arduino-Editor springt an die fehlerhafte Zeile im Programmcode und markiert sie gelb. So lassen sich schnell Probleme in einem Programm finden. Das erfolgreiche Übersetzen des Programms heißt jedoch noch nicht, dass es auch korrekt ist, denn es können noch viel mehr Fehler auftreten als nur Syntaxfehler.

Abbildung C-1 ▶
Programmfehler im Arduino-Edito



Die Arduino-Programmiersprache baut auf der Programmiersprache C++ auf. Die Grundsyntax der Programmiersprache (welche Wörter eine besondere Bedeutung haben und welche Sonderzeichen benutzt werden, um Abschnitte des Programms abzugrenzen) ist also dieselbe wie die von C++. Falls Sie C++ schon kennen, werden Sie merken, dass viele der erweiterten Konstrukte (z.B. Klassen) von C++ in normalen Sketchen nicht verwendet werden.

Struktur, Werte und Funktionen

Ein Arduino-Programm kann man grob in drei Bereiche unterteilen: Programmstruktur (der Ablauf eines Programms), Programmwerte (Variablen und Konstanten) und Programmfunktionen (eine

logische Einteilung von Programmfunktionalität, um diese verständlicher und wiederverwendbar zu gestalten).

Die Programmstruktur kann man als das »eigentliche« Programm betrachten. Es umfasst Kontrollstrukturen, die es ermöglichen, Algorithmen zu implementieren, indem der Programmfluss gesteuert wird: `if`, `if...else`, `switch`, `while` usw. Es umfasst auch arithmetische Operationen (addieren, subtrahieren, vergleichen, logische Operatoren), Zuweisungen (den Wert einer Variablen setzen oder lesen) und Funktionsaufrufe.

Programmwerte kann man in Variablen und in Konstanten unterteilen. Konstanten sind symbolische Namen für konstante numerische Werte, die eine bestimmte festgelegte Bedeutung haben (z.B. den logischen Wert für eine hohe Spannung an einem Eingangspin). Variablen sind Ausdrücke, die einen Wert speichern können, z.B. eine Zahl, die von einem analogen Eingang gelesen wird, oder eine Zeichenkette. Variablen haben unterschiedliche Datentypen, je nachdem, welche Art von Werten gespeichert werden soll.

Letztendlich werden Abschnitte von Programmstrukturen in Funktionen unterteilt, um sie logisch zu trennen und wiederverwendbar zu machen. Anstatt immer wieder denselben Programmcode zu schreiben (unter Umständen mit nur leichten Änderungen), kann man diesen als Funktion abkapseln und mit Funktionsaufrufen immer wieder neu ausführen. Einem Funktionsaufruf können auch Parameter übergeben werden, damit die Funktion flexibler wird und so vielfältiger eingesetzt werden kann. Libraries, die bestimmte Funktionalitäten anbieten (z.B. Schnittstellen zu diversen Sensoren und Aktoren oder bestimmte mathematische Berechnungen) werden meistens über einige bestimmte Funktionen angesprochen. Die Zusammenfassung dieser Funktionen nennt man Library-Schnittstelle (oder API, *application programming interface*).

Syntax

Eine Programmiersprache ist eine formale Sprache, die strikt definierten Regeln folgt, damit der Computer sie auch verstehen kann. Anstatt Sätze zu schreiben, werden in einer Programmiersprache Befehle an den Arduino geschrieben, die man Anweisungen nennt. Eine Anweisung kann zum Beispiel sein: »Addiere 2 und 5, oder setze den Ausgangspin 5 auf HIGH.«

In Arduino muss jede dieser Anweisungen mit einem Semikolon abgeschlossen werden. Vergisst man, eine Anweisung mit einem Semikolon abzuschließen, wird Arduino einen Fehler melden. Es kann allerdings passieren, dass der Fehler, der durch ein fehlendes Semikolon ausgelöst wird, sehr kryptisch ist, weil der Compiler die Programmstruktur nicht korrekt erkennen kann. Wenn Arduino also einen »absurden« Fehler meldet, ist es sinnvoll, zuerst nach Syntaxfehlern im Programmquellcode zu suchen.

Beispiel

```
int buttonPin = 2;
a = a + 3;
meineFunktion();
```

Mehrere Anweisungen können zu einer übergeordneten zusammengefasst werden, indem sie mit geschweiften Klammern gruppiert werden. Diese Gruppierungen nennt man *Programmblock*. Jede öffnende geschweifte Klammer muss von einer passenden schließenden geschweiften Klammer gefolgt werden. Der Arduino-Editor ist dort eine große Hilfe: Positioniert man den Cursor hinter einer schließenden Klammer, wird die dazugehörige öffnende Klammer hervorgehoben. Die Programmblöcke, die durch Klammern erzeugt werden, sind enorm wichtig für die Programmstruktur. Fehlende oder falsch gesetzte Klammern können (wie fehlende Semikola) zu kryptischen Programmfehlern führen (oder, noch schlimmer, zu syntaktisch korrekten, aber fehlerhaften Programmen). Es lohnt sich also, vorsichtig mit Klammern umzugehen und immer sicherzustellen, dass Klammerpaare korrekt sind.

Geschweifte Klammern ({}) werden benutzt, um mehrere Statements zu einem Funktionskörper zusammenzufassen und einzelne Pfade bei Programmstruktur-Konstrukten zu bilden.

Beispiel

```
void meineFunktion(){
   statement1;
   statement2;
   statement3;
}
if (i > 0) {
   statement1;
   statement2;
} else {
   statement3;
   statement4;
}
```

Es ist auch möglich, Textabschnitte in einem Programm so zu markieren, dass sie vom Compiler ignoriert werden. So kann ein Programm mit normalen Sätzen auf Deutsch oder Englisch kommentiert werden, ohne dass der Compiler versucht, diese als Programmcode zu interpretieren und deswegen einen Fehler meldet. Sieht der Compiler die Zeichenkette `//`, wird alles bis zum Ende der Zeile ignoriert. Mit `//` kann man also kurze Kommentare, die sich auf eine Zeile beziehen, einfügen.

Größere Blöcke von Programmcode lassen sich durch `/*` und `*/` auskommentieren. Alles, was sich zwischen diesen beiden Zeichenketten befindet, wird vom Compiler ignoriert. Es können schwere Fehler erzeugt werden, wenn solche Kommentare ungründlich verschachtelt werden.

Kommentare sind natürlich in erster Linie dazu da, komplizierte Programme für menschliche Leser zu beschreiben. So können Algorithmen erläutert, Hinweise auf Nebenwirkungen oder Fehlermöglichkeiten gegeben oder Anleitungen zu Anwendung und Modifikation des Codes gegeben werden. Es gilt hier allerdings: Kommentare sind genauso wichtig wie der Programmcode selbst und genauso aufwendig zu warten. Schlimmer als keine Kommentare sind falsche Kommentare, die nicht zum kommentierten Programmcode passen.

Es ist auch sehr praktisch, Teile des Programms auszukommentieren, um verschiedene Möglichkeiten zu testen oder Programmteile auszugrenzen, die Probleme erzeugen. Das kann besonders nützlich sein, wenn der Compiler kryptische Fehler meldet oder ein Bug nicht genau eingegrenzt werden kann. Mit Kommentaren können auch Debugging-Anweisungen ein- und ausgeschaltet werden.

Beispiel

```
x = 5; // Dieser Kommentar geht bis zum Zeilenende.
x = x + 1; // Einzeilige Kommentare kann man so gestalten, dass sie
           // über mehrere Zeilen gehen.
/* So lassen sich auch Kommentare schreiben,
   die über mehrere Zeilen gehen.
   Diese sind besonders nützlich für längere Erklärungen, oder um
   komplette Programmteile auszukommentieren.
*/
/* Es ist wichtig, keine Kommentare zu verschachteln.
if (i > 0) {
   statement1; /* denn das funktioniert nicht und führt zu
               Programmfehlern */
```

```
    statement2; // Einzeilige Kommentare kann man allerdings
                // schachteln.
}
*/
```

Um Programmcode leserlich zu halten (ihn zu schreiben, ist immer viel einfacher, als ihn nachträglich zu verstehen und zu modifizieren), ist es sehr wichtig, gewisse Schreibregeln einzuhalten. So wird Code, der in geschweiften Klammern gruppiert ist, eingerückt, damit man schon beim Überfliegen der Datei sehen kann, wie das Programm strukturiert ist. Der Arduino-Editor hilft einem dabei, indem der Cursor automatisch an die richtige Stelle gesetzt wird, wenn eine neue Zeile eingefügt wird. Wird jedoch durch späteres Editieren die Programmstruktur verändert, kann es leicht vorkommen, dass die Einrückungen nicht mehr korrekt sind. Der komplette Programmcode kann zurechtgerückt werden, indem die Funktion TOOLS → AUTOFORMAT ausgeführt wird (alternativ auch Apfel-T auf Mac OS X und Steuerung-T auf Linux und Windows).

Neben den Syntaxsymbolen werden in der Arduino-Programmiersprache numerische Werte (also Zahlen) und Wörter verwendet. Ein Teil der möglichen Wörter ist als Teil der Programmiersprache definiert (z.B. `int`, `if`, `else`, `return`, `break` usw., aber auch Operatoren wie `+`, `-`, `<<`, `&&` und `&`). Diese Wörter und Zeichen nennt man »reserviert«, denn sie haben eine von der Programmiersprache definierte Bedeutung. Weitere Wörter können vom Programmierer als Variablen (siehe Abschnitt über Variablen) und Funktionen (siehe Abschnitt über Funktionen) definiert werden. Diese frei definierbaren Wörter müssen mit einem Buchstaben anfangen (Spezialbuchstaben wie ä oder ü sind nicht erlaubt, außer in Zeichenketten) und dürfen nur Buchstaben, Zahlen und das _-Zeichen enthalten. Groß- und Kleinschreibung ist in Arduino-Programmen wichtig. Im folgenden Beispiel werden einige mögliche Variablennamen gezeigt.

Beispiel

```
byte meineVariable; // Das Trennen von Teilwörtern in einem Namen
           // mit Großbuchstaben ist eine weit verbreitete
           // Programmierkonvention, die man "CamelCase" nennt.
const byte MeineZweiteVariable; // Eine weitere Konvention ist ein
           // Großbuchstabe als erster Buchstabe bei Konstanten
           // (siehe Konstanten).
```

Programmwerte (Variablen, Datentypen und Konstanten)

Die meisten Arduino-Programme gehen mit Daten um, die in dem internen Speicher des Arduino abgelegt oder von den Eingangspins abgelesen werden. Diese Daten werden im Programm in Variablen gespeichert, damit man als Programmierer systematisch auf den Speicher zugreifen kann.

Eine Variable hat einen Namen, einen Datentyp und einen Wert. Der Name ist eine beliebige Zeichenkette, die keine Sonderzeichen oder Leerzeichen enthalten darf, die mit einem Buchstaben anfangen muss und die nicht zu den reservierten Wörtern der Arduino-Programmsprache gehören darf.

Datentypen

Weiterhin hat eine Variable einen Datentyp. Der interne Speicher des Arduino besteht aus einer langen Tabelle von 8-Bit-Werten; eine Variable kann allerdings noch viel mehr Arten von Daten enthalten. Datentypen kann man in zwei Gruppen unterteilen: numerische und zusammengesetzte Datentypen.

Numerische Datentypen werden benutzt, um Zahlen oder Boolesche logische Werte abzubilden. Sie unterscheiden sich grundsätzlich durch den Zahlenbereich, den sie abbilden können. Hier muss man zwischen ganzzahligen numerischen Typen unterscheiden, die nur ganze Zahlen darstellen können, und Fließpunktkommazahlen, die Kommazahlen darstellen können. Die Datentypen *float* und *double* werden benutzt, um Fließkommazahlen zu speichern. Der Arduino basiert auf einem 8-Bit-Prozessor, sodass Kommazahlen nur mit viel Mehraufwand eingesetzt werden können. Falls schnelle Berechnungen notwendig sind (oder ein kürzerer Programmcode) ist es besser, ganze Zahlen einzusetzen.

Zusammengesetzte Datentypen lassen sich in Tabellen und Datenstrukturen unterteilen. Datenstrukturen sind ein fortgeschrittenes Thema und werden in diesem Buch nicht behandelt. Datentabellen sind Listen von vielen Werten eines Datentyps, auf die indexiert zugegriffen werden kann. Diese Werte werden hintereinander im Speicher abgelegt. Einen speziellen zusammengesetzten Datentyp bilden Zeichenketten, die als Tabelle von `char`-Werten im Speicher abgebildet werden. Der letzte Wert in einer Zeichenkette ist immer `0`.

Im Folgenden sehen Sie die meistbenutzten Datentypen in Arduino-Programmen.

Tabelle C-1 ▶ Datentypen

Datentyp	Beschreibung	Beispiel
`boolean`	Ein `boolean` hat entweder den Wert `true` oder den Wert `false`. Dieser Datentyp wird benutzt, um Binärlogikwerte zu speichern.	`boolean a = true;`
`char`	Speichert einen Buchstaben in einem Byte Speicher.	`char c = 'a';`
`byte`	Speichert einen numerischen Wert von 0 bis 255 in einem Byte Speicher.	`byte b = 35;`
`signed byte`	Speichert einen numerischen Wert von –128 bis 127 in einem Byte Speicher.	`unsigned byte b = 35;`
`int`	Speichert einen numerischen Wert von –32.768 bis 32.767 in 2 Bytes Speicher.	`int i = -9892;`
`unsigned int`	Speichert einen numerischen Wert von 0 bis 65.535 in 2 Bytes Speicher.	`unsigned int i = 9723;`
`long`	Speichert einen numerischen Wert von – 2.147.483.648 bis 2.147.483.647 in 4 Bytes Speicher.	`long l = 98237498L;`
`unsigned long`	Speichert einen numerischen Wert von 0 bis 4.294.967.295 in 4 Bytes Speicher.	`unsigned long l = 834759824L;`
`float`	Speichert eine Fließkommazahl von mittlerer Auflösung in 4 Bytes Speicher.	`float f = 4.34;`
`double`	Auf dem Arduino ist `double` äquivalent zu `float`.	`double d = 4.34;`
`string`	Speichert eine Zeichenkette als 8-Bit-Tabelle mit abschließender 0.	`char str[] = "arduino";`
`array`	Speichert eine Tabelle von Variablen. Die Größe wird angegeben.	`int meineWerte[] = {1, 2, 3, 4, 5};`
`void`	Wird nur bei Funktionsdeklarationen verwendet und gibt an, dass die Funktion keinen Rückgabewert hat.	

Numerische Datentypen und binäre Darstellung

Wie man anhand der Tabelle sehen kann, gibt es eine Vielzahl numerischer Datentypen in der Arduino-Programmiersprache. Diese unterscheiden sich durch ihre Größe und den numerischen Bereich, den sie abdecken. In diesem Abschnitt wird ein bisschen tiefer gehend erklärt, wie Zahlen auf dem Arduino-Chip abgelegt werden, da es sehr wichtig ist, das zu verstehen, um Fehler zu vermeiden und den richtigen Datentyp auszuwählen.

In einem Prozessor werden sämtliche Werte als Folgen von Bits abgespeichert. Ein Bit ist die grundlegende Speichereinheit und kann

als Wert entweder 0 oder 1 annehmen. Da man mit einem einzelnen Bit noch nicht viel rechnen kann, werden sie in Bytes zusammengefasst, wobei ein Byte 8 Bits umfasst. Dabei wird das sogenannte Binärsystem benutzt, das ähnlich wie unser Dezimalsystem funktioniert. Wenn wir 13 schreiben, heißt das soviel wie 1 * 10 + 3, jede Stelle nach links bedeutet also zehn Mal so viel wie die auf der rechten Seite. Im Binärsystem ist jede Stelle nach links nur zweimal wichtiger, weil eine Stelle eben nur zwei Werte darstellen kann. In der folgenden Tabelle sehen Sie ein paar Zahlen in Binärdarstellung aufgelistet. Mehr über das Binärsystem findet man zum Beispiel in der Wikipedia unter *http://de.wikipedia.org/wiki/Binärsystem*.

Wenn ein numerischer Wert als Folge von Bits interpretiert wird, redet man von einer *Bitmaske*. Für den Arduino-Prozessor gibt es keinen Unterschied zwischen einer normalen Zahl und einer Bitmaske, diese Unterscheidung ist also nur für den Programmierer von Relevanz. Bitmasken sind bei der Programmierung von eingebetteten Prozessoren, wie der Arduino-Prozessor einer ist, oft von Relevanz. So lassen sich in einer Bitmaske digitale Werte (also 0 oder 1) sehr speichereffizient speichern (siehe Kapitel 4 über LED-Matrix). Anstatt ein Byte zu verwenden, um einen digitalen Wert zu speichern, kann ein Byte als Bitmaske interpretiert gleich acht digitale Werte speichern (siehe Abschnitt über logische Operationen). Mit einer Bitmaske lassen sich auch mehrere Pins auf einmal schalten, was in bestimmten Anwendungen (siehe Kapitel 4, LED-Matrix und Kapitel 10 über Musik) aus Geschwindigkeitsgründen wichtig sein kann. Da Zahlen generell als Bitmasken gespeichert werden, lassen sich bestimmte mathematische Operationen mit Bitmasken sehr effizient berechnen (siehe Abschnitt »Logische Operationen«).

◀ **Tabelle C-2**
Binärdarstellung

Dezimaldarstellung	Hexadezimaldarstellung	Binärdarstellung
0	0x0	0
1	0x1	1
2	0x2	10
10	0xa	1010
15	0xf	1111
255	0xff	11111111
260	0x104	1 00000100
60000	0xea60	11101010 01100000
-128 (8 bit)	-0x80	10000000
-127 (8 bit)	-0x7f	10000001
-128 (16 bit)	-0x80	11111111 10000000
-127 (16 bit)	-0x7f	11111111 10000001

Zahlen von 0 bis 255 lassen sich in einem Byte speichern, größere Zahlen werden in mehreren Bytes gespeichert. Intern arbeitet der Prozessor auf einem Arduino-Board mit 8 Bit, d. h. er kann intern 8-Bit-Werte addieren, subtrahieren und multiplizieren. Werden größere numerische Werte benutzt, werden diese in mehrere 8-Bit-Werte zerlegt und nach der Bearbeitung im Prozessor wieder zusammengefügt. Wenn viele Daten sehr schnell bearbeitet werden müssen (z. B. bei bestimmten Sensoren), ist es also wichtig, die passende Größe für numerische Werte zu benutzen.

Negative Zahlen werden in der sogenannten Zweierkomplement-Darstellung gespeichert. Dazu wird das größte Bit benutzt, um zu speichern, ob die Zahl positiv oder negativ ist. Ist das größte Bit gesetzt, wird die Zahl in den unteren Bits auf den kleinsten negativsten Wert addiert (bei 8 Bit z. B. –128). Die Datentypen, die negative und positive Zahlen erlauben, werden »signierte« (englisch *signed*) Datentypen genannt, im Unterschied zu den »unsignierten« (*unsigned*) Datentypen, die nur positive Werte speichern können.

Dadurch, dass der größte und der kleinste Wert einer Variable beschränkt sind, kann es dazu kommen, dass eine Variable überläuft bzw. unterläuft. Wird z. B. zum 8-Bit-Wert 255 eine 2 addiert, kommt als Ergebnis nicht 261 heraus, sondern 1, weil die Variable übergelaufen ist. Das ist eine häufige Quelle von Fehlern, weshalb es notwendig ist, bei der Wahl von numerischen Datentypen immer sehr vorsichtig vorzugehen. Besonders kritisch und kompliziert wird es, wenn signierte und unsignierte Datentypen in mathematischen Operationen vermischt werden. In solchen Operationen ist es sehr schwer nachzuvollziehen, wo und wie ein Überlauf passieren kann. Wann immer mit unsignierten und signierten Datentypen gerechnet wird, lohnt es sich also, die unsignierten Datentypen in signierte Datentypen umzuwandeln und erst dann zu rechnen.

Beispiel

```
unsigned char x = 0; // unsignierter Datentyp, 0-255
char y = 0; // signierter Datentyp, -128 – 127
char x2 = x; // umwandeln von x in einen signierten Datentyp
             // x muss kleiner als 128 sein!
char ergebnis = y + x;
```

In der Arduino-Programmiersprache können Zahlen in verschiedenen Darstellungen angegeben werden. So ist es natürlich möglich, Zahlen in der üblichen Dezimaldarstellung zu schreiben. Ganz ähnlich können Kommazahlen (als Datentyp `float`) in der Dezimaldarstellung angegeben werden. Als Komma wird das Punktzeichen verwendet, wie es in englischsprachigen Ländern üblich ist.

Weiterhin können Zahlen aber auch in der Hexadezimaldarstellung angegeben werden, in der sie mit 0x beginnen. Bei der Hexadezimaldarstellung werden die Ziffern 0–9 und die Buchstaben a-f benutzt. a ist dabei äquivalent zu 10 in der Dezimaldarstellung, b zu 11, c zu 12 usw. Eine Ziffernstelle in Hexidezimaldarstellung geht also von 0 bis 15, nicht von 0 bis 10 wie in Dezimaldarstellung. Das Praktische an der Hexadezimaldarstellung ist, dass eine Ziffer genau 4 Bit im Speicher belegt, ein Byte also als zwei Ziffern angegeben werden kann. Dadurch lässt sich beim ersten Blick auf die Zahl sehen, wie viel Speicher sie benötigt. Deswegen werden häufig Adresswerte und Zahlenwerte in technischen Artikeln und Datenblättern in Hexadezimaldarstellung angegeben.

In der Arduino-Programmiersprache (aber nicht im normalen C++) lassen sich Zahlenwerte auch direkt in Binärdarstellung angeben. Das ist besonders praktisch bei der Ansteuerung bestimmter Sensoren und Aktoren, bei denen einzelne Bits eine bestimmte Bedeutung haben. Zahlen in Binärdarstellung werden mit 0b am Anfang angegeben.

Beispiel

```
unsigned char a = 128; // Dezimaldarstellung
float f = 3.14; // Kommadarstellung
unsigned char x = 0x80; // Hexadezimaldarstellung
unsigned char b = 0b10101010; // Binärdarstellung
```

Fließkommazahlen

Fließkommazahlen werden benutzt, um Zahlen darzustellen, die ein Komma (das ja durch einen Punkt dargestellt wird) haben. Allerdings sind diese Zahlen nicht genau, sondern nähern diesen Wert soweit möglich. Der darstellbare Zahlenbereich geht von –3.4028253E+38 bis 3.4028235E+38. Dadurch, dass diese Zahlen nicht genau sind, kann es vorkommen, dass z. B. 6.0 geteilt durch 3.0 nicht gleich 2.0 ist. Der Prozessor auf einem Arduino-Board kann nicht mit Fließkommazahlen umgehen, sodass eine zusätzliche Library benutzt wird, die diese Berechnung auf 8-Bit-Zahlen abbildet. Dadurch sind Fließkommaberechnungen sehr langsam, und bei zeitkritischen Berechnungen ist es oft notwendig, sie auf ganze Zahlen umzuwandeln.

Buchstaben (char)

Buchstaben werden in der Arduino-Programmiersprache als numerische Werte gespeichert. Ein Buchstabe ist ein Byte groß und kann

Werte von 0 bis 255 annehmen. Um zuzuordnen, welcher Buchstabe welchen numerischen Wert hat, wird die ASCII-Tabelle benutzt. Diese enthält die numerischen Werte für viele übliche Buchstaben (auch viele der deutschen Spezialzeichen sowie Sonderzeichen wie Zeilenumbruch oder Leerzeichen). Damit in Programmen aber nicht immer Zahlen als Buchstaben angegeben werden, können einfache Anführungszeichen verwendet werden. Den Buchstaben a kann man also als `'a'` notieren. Es ist sehr wichtig, zwischen einfachen Anführungszeichen, die zum Kennzeichnen von einzelnen Buchstaben benutzt werden, und doppelten Anführungszeichen, die zum Kennzeichnen von Zeichenketten verwendet werden (siehe Abschnitt »Variablen« in diesem Anhang), zu unterscheiden.

Tabellen (Arrays)

Tabellen speichern Listen von Werten desselben Datentyps. Anhand eines numerischen Index kann dann auf einen Wert aus einer Liste zugegriffen werden. Es gibt mehrere Arten, Arrays zu erzeugen.

Beispiel

```
int meineTabelle[6];
int meineWerte[] = {4, 5, 6, 7, 8, 9, 10};
int meineNummern[5] = { 4, 5, 6, 7, 8}
```

In der ersten Zeile in diesem Beispiel wird eine Tabelle mit dem Datentyp *int* erzeugt, die sechs Werte speichern kann. Im Speicher werden also zwölf Bytes benutzt (weil ein *int*-Wert zwei Bytes belegt), um diese sechs Werte zu speichern. Diese sechs Werte werden automatisch auf 0 initialisiert.

In der zweiten Zeile wird die Größe des Array nicht explizit definiert, sondern implizit durch die folgenden (sieben) Initialisierungswerte gesetzt.

Im der dritten Zeile werden die Größe des Array definiert und Initialisierungswerte übergeben. Stimmt die Größe der Initialisierungswerte nicht mit der Größe des Array überein, wird vom Compiler ein Fehler gemeldet.

Diese Tabellen werden im Arbeitsspeicher des Arduino-Prozessors gehalten, der relativ klein ist (2 oder 4 KByte auf einem Arduino Duemilanove, je nach Herstellungszeitpunkt). Der Arduino-Editor wird allerdings keinen Fehler signalisieren, wenn eine Tabelle zu groß für den Arbeitsspeicher ist. Es ist also wichtig, sich darüber im

Klaren zu sein, wie viel Speicher größere Tabellen belegen, und ob so viel noch vorhanden ist.

Der Zugriff auf einzelne Tabellenwerte erfolgt über einen sogenannten Indexwert. Dieser gibt an, an welcher Stelle innerhalb der Tabelle ein Wert geschrieben oder ausgelesen werden soll.

Beispiel

```
meineTabelle[0]; // Lesezugriff auf den ersten Wert aus
                 // meineTabelle
meineTabelle[6]; // Lesezugriff auf den letzten Wert aus
                 // meineTabelle
meineTabelle[0] = 1; // Schreibzugriff auf den ersten Wert aus
                     // meineTabelle
```

Indexe für Tabellen fangen bei 0 an und gehen bis zur Größe der Tabelle minus eins. Deswegen wird in Programmen generell von 0 aus gezählt, nicht von 1 aus (das ist besonders bei Schleifen wichtig, siehe Abschnitt Kontrollstrukturen). Bei einer Tabelle der Größe 6 geht der Indexwert also von 0 bis 5. Es ist sehr wichtig sicherzustellen, dass der Indexwert innerhalb der zugelassenen Größe liegt. Der Arduino-Prozessor wird das beim Ausführen des Tabellenzugriffes nicht überprüfen und wird irgendwo im Speicher lesen (was nicht so schlimm ist, aber zu komischen Ergebnissen führt) oder schreiben (was deutlich schlimmer ist und zu sehr schwer nachvollziehbaren Fehlern führt, z.B. dass das Arduino-Board spontan neu startet oder in Funktionen springt, die gar nicht angegeben worden sind). Bei Tabellenzugriffen ist also äußerste Vorsicht geboten.

Es ist oft nützlich, eine Tabelle mit einer `for`-Schleife zu durchlaufen (siehe Abschnitt Kontrollstrukturen). Dazu wird eine Variable hochgezählt, die jeden möglichen Index-Wert für die Tabelle annimmt.

Beispiel

```
for (int i = 0; i < 5; i++) { // i nimmt alle Werte von 0 bis 5 an
   meineTabelle[i] = i; // Zugriff auf das i-te Element aus
                        // meineTabelle
}
```

Eine fortgeschrittene Nutzung von Tabellen ist die Möglichkeit, mehrdimensionale Tabellen zu erzeugen. Auf eine mehrdimensionale Tabelle wird anhand von zwei oder mehr Indexwerten zugegriffen. Damit hat man die Möglichkeit, z.B. zweidimensionale Tabellen zum Speichern von Grafiken für eine LED-Matrix zu erzeugen. Zum Erzeugen einer mehrdimensionalen Tabelle werden meh-

rere Größen in eckigen Klammern angegeben. Zum Initialisieren mehrdimensionaler Tabellen werden geschachtelte Initialisierungswerte in geschweiften Klammern angegeben. Der Speicheraufwand einer mehrdimensionalen Tabelle ist desto größer, je mehr Dimensionen verwendet werden. So belegt eine 5-mal-5-Tabelle genauso viel Speicher wie eine eindimensionale Tabelle mit 25 Werten.

Beispiel

```
int meine2DTabelle[5][5]; // Diese Tabelle speichert 5 x 5 = 25
                          // Werte.
int nochEineTabelle[3][2] = { // Diese Tabelle wird auch
                              // initialisiert.
  { 0, 1 }, // Hier ist es wichtig zu beachten, dass die Tabelle
            // dreimal
  { 2, 3 }, // zwei Werte speichert, nicht zweimal drei Werte.
  { 4, 5 }
};
```

Hier ist es auch wichtig, die Grenzen der einzelnen Größen zu beachten. Beim Zugriff muss sich der erste Indexwert innerhalb des für die erste Tabelle festgelegten Rahmens befinden, der zweite innerhalb des Rahmens der zweiten Tabelle usw..

Beispiel

```
int meine2DTabelle[5][5]; // Diese Tabelle speichert 5 x 5 = 25
                          // Werte.
meine2DTabelle[0][0] = 1; // Zugriff auf verschiedene Elemente
                          // innerhalb
meine2DTabelle[0][1] = 2; // der zweidimensionalen Tabelle
meine2DTabelle[3][3] = 3;
```

Ähnlich wie bei einer eindimensionalen Tabelle kann man eine mehrdimensionale Tabelle mit mehreren geschachtelten `for`-Schleifen durchlaufen.

Beispiel

```
// for-schleife zum Durchlaufen der ersten Dimension
for (int y = 0; y < 5; y++) {
   // for-schleife zum Durchlaufen der zweiten Dimension
   for (int x = 0; x < 5; x++) {
    meine2DTabelle[y][x] = x; // Zugriff auf die zweidimensionale
                              // Tabelle
   }
}
```

Zeichenketten

Zeichenketten sind einfach Tabellen von *char*-Werten. Zusätzlich muss allerdings noch die Länge der Zeichenkette angegeben wer-

den. Dazu wird an das Ende der Tabelle eine 0 angehängt, die signalisiert, dass damit die Zeichenkette zu Ende ist. Um zur Initialisierung von Zeichenketten nicht immer explizit diese 0 angeben zu müssen, gibt es die Möglichkeit, die Zeichenkette in doppelte Anführungszeichen zu setzen. Dadurch wird automatisch eine Zeichentabelle angelegt und die schließende 0 eingefügt.

Es ist sehr wichtig, diese 0 anzugeben. Fehlt sie, werden die meisten Funktionen, die mit Zeichenketten arbeiten, das Ende der Zeichenkette nicht erkennen und weiter im Speicher lesen oder schreiben, mit den problematischen Folgen, die schon erläutert wurden. Werden statt doppelter Anführungszeichen (") einfache Anführungszeichen (') benutzt, wird auch eine Tabelle für die Zeichenkette erzeugt, allerdings ohne schließende 0.

Beispiel

```
char meinString[8]; // leere Zeichenkette der Größe 8
// explizite Initialisierung der Zeichenkette mit schließender 0
// Die Null wird als '\0' (Buchstabenwert 0) angegeben.
char meinString2[8] = { 'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0' };
// explizite Initialisierung der Zeichenkette, die 0 ist implizit
char meinString3[8] = { 'a', 'r', 'd', 'u', 'i', 'n', 'o' };
// Initialisierung mit doppelten Anführungszeichen
char meinString4[] = "arduino";
// Initialisierung mit expliziter Größe
char meinString5[8] = "arduino";
```

Wegen der schließenden 0 sind für die Zeichenkette `arduino` also acht Buchstaben notwendig. Es ist auch möglich, direkt eine mit doppelten Anführungszeichen gekennzeichnete Zeichenkette innerhalb eines Programms zu benutzen, ohne eine eigene Variable dafür einzurichten. Es wird allerdings immer noch der notwendige Speicherplatz für diese Variable benutzt. Wenn also viele Zeichenketten in einem Programm verwendet werden, ist es unter Umständen auch nötig, ein Auge auf den Speicherplatzverbrauch zu werfen.

Oft muss man in Anwendungen, die viel mit Zeichenketten arbeiten (z.B. Anwendungen, die ein Display verwenden), eine Tabelle von Zeichenketten anlegen. Dazu wird eine sogenannte Zeigertabelle verwendet. Zeiger sind ein fortgeschrittenes Konstrukt der Arduino-Programmiersprache, das hier nur verwendet, aber nicht weiter erläutert wird. Ein Zeiger »zeigt« auf einen Speicherbereich, in dem andere Daten gespeichert werden. Für unseren Gebrauch lassen sich Zeiger so benutzen wie Tabellen. Man kann also grob sagen, dass die untere Zeigertabelle weitere Tabellen speichert, die wiederum unsere Zeichenketten enthalten.

Beispiel

```
char *meineKetten[] = {
   "meine erste Kette",
   "meine zweite Kette",
   "meine dritte Kette"
};
Serial.println(meineKetten[0]);
Serial.println(meineKetten[1]);
```

So wie Tabellen können auch Strings durchlaufen werden. Allerdings ist die Länge der Zeichenkette nicht bekannt, und die schließende 0 muss erkannt werden.

Beispiel

```
char meinString[] = "arduino";
for (int i = 0; meinString[i] != 0; i++) {
   sendCharacter(meinString[i]);
}
```

Variablen

Mit Variablen lassen sich Speicherbereiche mit symbolischen Namen verbinden. Das erleichtert erheblich die Programmierung, weil sich so mit einem informativen Namen Werte auslesen und speichern lassen. Zum Beispiel kann die Variable `sensorWert` benutzt werden, um den zuletzt ausgelesenen Statuswert eines Sensors zu speichern.

Jede Variable ist mit einem Datentyp verbunden. So kann beim Kompilieren des Programms berechnet werden, welcher Bereich im Speicher belegt ist und wie mit den dort gespeicherten Werten gerechnet werden muss. Dadurch können Tabellenzugriffe und Zeichenkettenzugriffe berechnet werden.

Variablen sind eines der wichtigsten Konzepte beim Programmieren, und sie sind in quasi jedem Programm mehrfach anzutreffen. Die Wahl der benutzten Variablen (also der Namen für Speicherbereiche) ist äußerst wichtig für die Strukturierung eines Programms. Namen dienen auch zur menschlichen Verständlichkeit eines Programms. So sollte eine Variable namens `zaehler` nicht benutzt werden, um eine Zeichenkette zu speichern, da das verwirrend sein kann, wenn man ein Programm nach einiger Zeit wieder modifizieren und dazu verstehen will.

Beispiel

```
char meinString[] = "hallo"; // Die Variable meinString speichert
                             // einen String.
int meineZahl = 5; // Die Variable meineZahl speichert eine Zahl
```

```
                    // vom Datentyp int.
int meineTabelle = 5; // Der Name meineTabelle ist schlecht
                      // gewählt, da meineTabelle eine Zahl
                      // speichert.
```

Um die Arbeit mit Variablen zu erleichtern, benutzt die Arduino-Programmiersprache das Konzept der *Variablensichtbarkeit*. Nicht an allen Orten im Programm sind alle Variablen erreichbar. Als Faustregel gilt, dass innerhalb von geschweiften Klammern definierte Variablen nur in diesem Bereich auch sichtbar sind (man nennt sie *lokale Variablen*). Die äußerste Ebene des Programms ist die globale Ebene, in der sogenannte *globale Variablen* (und Funktionen) definiert werden können. Diese globalen Variablen sind im gesamten Programm sichtbar und können deshalb benutzt werden, um Informationen zu speichern, die zentral für das Programm sind. Weiterhin kann eine Variable nicht benutzt werden, bevor sie deklariert wird.

Geschweifte Klammern können geschachtelt werden (besonders mit Kontrollstrukturen, siehe den entsprechenden Abschnitt). Variablen sind somit auch geschachtelt, und es gilt immer die zuletzt definierte Version einer Variablen. Hier ist auch Vorsicht geboten, weil so mehrere Variablen mit demselben Namen (aber in anderen Speicherbereichen) definiert werden und dann im Programm-Quellcode verwirrend sein können (und zu Fehlern führen).

Beispiel

```
int globaleZahl = 5; // globaleZahl ist eine globale Variable und
                     // von überall aus zugreifbar.
void setup() {
        int lokaleZahl = 5; // lokaleZahl ist nur innerhalb der
                            // setup-Funktion erreichbar.
        for (int i = 0; i < 10; i++) {
             int lokalereZahl = 8; // i und lokalereZahl sind nur
                                   // innerhalb der for-Schleife
                                   // erreichbar.
             int lokaleZahl = 10; // Innerhalb der for-schleife
                                  // wurde auch eine neue
                                  // Variable namens lokaleZahl
                                  // definiert. Innerhalb der
                                  // Schleife hat lokaleZahl den
                                  // Wert 10.
             lokaleZahl = 18; // Hier wird die Variable
                              // lokaleZahl überschrieben, die
                              // innerhalb der for-Schleife
                              // definiert wurde.
        }
        // Hier hat lokaleZahl wieder den Wert 5.
}
```

Eine weitere Faustregel ist, dass Variablen immer im kleinstmöglichen Bereich definiert sein sollten. Werden Variablen außerhalb geschweifter Klammern nicht benutzt, ist es sinnvoll, sie in diesem Bereich auch zur definieren.

Oft werden globale Variablen als Konfigurationsvariablen benutzt. Diese setzen dann z. B. die Pinbelegung für das Arduino-Programm fest oder ermöglichen es, schnell bestimmte Zeitintervalle zu definieren. Es ist oft nützlich, diese globalen Konfigurationsvariablen unmittelbar am Anfang des Programms zu schreiben, damit sie alle an einem Ort gruppiert erscheinen. Späterer Benutzer des Programms können dann schnell bestimmte Konfigurationswerte einstellen, ohne im kompletten Programm nach ihnen suchen zu müssen.

Konstanten

Konstanten sind eine besondere Art von Variablen, die nicht verändert werden können. Konstanten belegen meistens auch keinen Speicherplatz (außer Tabellen und Zeichenketten) und werden oft benutzt, um bestimmten festen Werten symbolische Namen zuzuordnen (zum Beispiel die Nummer eines Pins, der eine bestimmte Funktion erfüllt, die Nummer eines bestimmten Kommandos auf einem externen Sensor oder ein Konfigurationsparameter der Software). Eine Konstante wird mit dem Wort `const` deklariert. Bei Schreibzugriffen auf diese Variable meldet die Arduino-Software einen Programmierfehler.

Beispiel

```
const int ledPin = 5; // ledPin kann nicht verändert werden und
                      // hat den Wert 5.
```

Allgemein ist es sehr praktisch, häufig benutzte explizite Werte (zum Beispiel eine Pinnummer, die häufig verwendet wird, oder die Dauer für einen `sleep()`-Aufruf) als Konstanten zu deklarieren. Damit sind diese Konfigurationswerte später leicht zu ändern, weil sie nur an der Definitionsstelle der Konstante geändert werden müssen, nicht an jeder Einsatzstelle. Weiterhin wird mit der Definition der Konstante jeder Programmierer, der das Programm zu einem späteren Zeitpunkt verwendet, darauf hingewiesen, dass ein bestimmter Konfigurationswert benutzt wird, und welche Bedeutung dieser Wert hat (wenn der Name der Konstante gut gewählt wurde).

Beispiel

```
const int ledPin = 5;
digitalWrite(ledPin, HIGH); // Diese Schreibweise ist leichter zu
                            // verstehen ...
digitalWrite(5, HIGH); // ... als diese Schreibweise.
```

Ausdrücke und Anweisungen

In der Arduino-Programmiersprache wird zwischen zwei Arten von Programmierbefehlen unterschieden. Auf der einen Seite gibt es Ausdrücke, die einen bestimmten Wert berechnen, den *Rückgabewert*. Diese Art von Programmierbefehlen ist mit einer Funktionsdefinition in der Mathematik zu vergleichen: Aus bestimmten Werten (Konstanten oder Variablen) wird ein neuer Wert berechnet, zum Beispiel werden mit dem Programmierbefehl `a + b` die Variablen `a` und `b` addiert. Ausdrücke lassen sich in beliebiger Weise kombinieren und schachteln, mit ähnlichen Regeln wie in der Mathematik. Es ist oft wichtig, runde Klammern ((und)) zu benutzen, um die Reihenfolge der Ausdrücke korrekt zu berechnen.

Da jeder Ausdruck einen Wert berechnet, hat jeder Ausdruck auch einen entsprechenden Datentyp, den man *Rückgabetyp* nennt. Dieser hängt sowohl vom Ausdruck als auch von den im Ausdruck benutzten Datenwerten ab. Im nächsten Abschnitt werden die verschiedenen Ausdrücke der Arduino-Programmiersprache erläutert. Eine spezielle Art von Ausdrücken sind Funktionsaufrufe, die in dem Abschnitt über Funktionen erklärt werden.

Auf der anderen Seite gibt es sogenannte Anweisungen, die im Unterschied zu Ausdrücken keine Werte berechnen, sondern organisatorische Rollen übernehmen und den Status des Prozessors beeinflussen (einen Wert zu berechnen, verändert erst einmal nichts, dazu muss er zunächst gespeichert werden). Sie bestimmen auch die Struktur des Programms, also welche Funktionen es gibt und welche Variablen definiert werden. Sie bestimmen auch, welcher Weg durch das Programm letztendlich benutzt wird (welche Anweisungen ausgeführt werden). Diesen Weg und seine Steuerung nennt man den *Programmkontrollfluss*.

Ausdrücke sind auch Anweisungen, allerdings bewirken Ausdrücke (außer den Zuweisungsausdrücken, siehe den Abschnitt über Ausdrücke) nichts: Die Werte, die sie berechnen, werden nirgendwo gespeichert, und meistens werden allein stehende Ausdrücke vom Compiler ignoriert.

Eine Art von Anweisungen wurde schon erläutert, die Variablendeklaration. Eine weitere Kategorie von Anweisungen sind Kontrollstrukturen und Funktionsdefinitionen, die in in ihren entsprechenden Abschnitten erläutert werden.

Ausdrücke

Die einfachste Art Ausdruck in der Arduino-Programmiersprache haben wir schon oft gesehen. Es ist ein einfacher expliziter Wert, z.B. eine Zahl oder die Angabe einer Zeichenkette. Dieser Ausdruck hat den Typ des expliziten Werts, und der Wert ist natürlich die angegebene Zahl. Bei Zeichenketten ist der Wert die Adresse der Speicherkette, dieses Detail ist allerdings in den meisten Arduino-Programmen nicht von Bedeutung.

Beispiel

```
5; // Zahlenausdruck mit Wert 5
"arduino"; // Der Wert dieses Ausdrucks ist die Adresse der
           // Zeichenkette.
// Beide Ausdrücke werden vom Compiler ignoriert.
```

Variablenzuweisung

Ein weiterer Ausdruck ist die Variablenzuweisung, die das Gleichheitszeichen (=) als Symbol verwendet. Auf der linken Seite des = steht die Variable, der ein Wert zugewiesen werden soll, und auf der rechten Seite steht der Ausdruck, der berechnet und anschließend gespeichert werden soll.

Eigentlich ist die Variablenzuweisung eine Anweisung, sie hat jedoch den zugewiesenen Wert als Rückgabewert und den Typ der zugewiesenen Variable als Rückgabetyp (das ist wichtig bei der automatischen Datentypkonvertierung, die weiter unten erklärt wird). Dadurch ist es möglich, mehrere Variablenzuweisungen zu schachteln und Variablenzuweisungen innerhalb von weiteren Ausdrücken zu verwenden. Hier ist aber wieder Vorsicht geboten, weil deswegen die Zuweisungsreihenfolge von Variablen nicht unbedingt klar ersichtlich ist. Es ist deswegen ratsam, Zuweisungen besser als getrennte Anweisungen in das Programm zu schreiben.

Beispiel

```
int i = 0;   // Die Variable i wird mit 0 initialisiert
             // (kein Ausdruck).
int j = 2;   // Die Variable j with mit 2 initialisiert.
i = 5;       // Der Variable i wird der Wert 5 zugewiesen.
```

```
j = i = 6;   // Der Variable i wird der Wert 6 zugewiesen,
             // dann wird der Variable j der Wert 6 zugewiesen.
j = (i = 5) * 2; // Der Variable i wird der Wert 5 zugewiesen,
                 // der Variable j wird der Wert 10 zugewiesen.
i = 5;           // Es ist allerdings ratsamer, die vorige Zeile
                 // in zwei getrennte
j = i * 2;       // Zuweisungen zu schreiben. Der erzeugte
                 // Maschinencode ist in beiden Fällen identisch.
```

Ein häufiger Fehler beim Lernen der Arduino-Programmiersprache (und von C und C++) ist das Verwechseln der Variablenzuweisung = und der Vergleichsoperation == (siehe Abschnitt Vergleichsoperationen). Im ersten Fall wird ein Wert gespeichert, und der Rückgabewert des Ausdrucks ist dieser Wert. Im zweiten Fall werden zwei Werte verglichen (es wird kein Wert gespeichert), und der Rückgabewert des Ausdrucks ist ein Boolescher Wert `true` oder `false`. Es kommt ziemlich häufig vor, dass = statt == verwendet wird, und der Compiler meldet auch keinen Fehler. Im Programm ist ein solcher Fehler auch schwer zu erkennen, weil = und == ähnlich aussehen. Ein Weg, um das zu umgehen, ist, bei Vergleichsoperationen mit literalen Werten den literalen Wert auf die linke Seite zu schreiben. Wird dann = mit == verwechselt, meldet der Compiler einen Fehler, weil in einem literalen Wert nichts gespeichert werden kann.

Beispiel

```
if (i = 2) { // Hier wird nicht i mit 2 verglichen, sondern i wird
             // 2 zugewiesen.
}
if (2 == i) { // Mit dieser Schreibweise lassen sich solche
              // Verwechslungen oft vermeiden.
}
```

Operationen

Der Großteil der Ausdrücke in der Arduino-Programmiersprache sind Operationen, die aus einem oder mehreren Argumenten einen neuen Wert berechnen. Es gibt mehrere Arten von Operationen, die innerhalb von Ausdrücken benutzt werden können. Eine Operation nimmt einen oder mehrere Werte und berechnet daraus einen neuen Ergebniswert. Je nach Operationsart und Typ der Eingabewerte werden Werte mit unterschiedlichen Datentypen berechnet.

Wir stellen Ihnen jetzt zuerst die verschiedenen Operationen im Überblick vor und erklären sie im weiteren Abschnitt dann detailliert.

Arithmetische Operationen berechnen aus Zahlenwerten einen neuen Zahlenwert: Es sind die bekannten Operationen wie Addieren, Subtrahieren, Multiplizieren und Teilen. Logische Operationen berechnen aus Bitmasken neue Bitmasken (Bitmasken sind binär dargestellte Zahlen), es sind die Operationen »und« (`&`), »oder« (`|`), »exklusiv oder« (`^`), »Negieren« (`~`) und »Shiften« (`<<` und `>>`). Boolesche Operationen berechnen aus den Booleschen Werten »wahr« und »falsch« neue Boolesche Ergebniswerte, die Operationen »und« (`&&`), »oder« (`||`) und »Negieren« (`!`). Diese sind sehr einfach mit den logischen Operationen zu verwechseln, sind aber ganz anders definiert (die Unterschiede werden weiter unten im Abschnitt über logische Operationen erläutert).

Viele Operationen arbeiten mit Booleschen Werten (also mit »wahr« oder »falsch«). Diese Werte müssen natürlich auch im Speicher des Arduino-Prozessors gespeichert werden, entweder als Bit in einem größeren numerischen Wert oder als eigener Wert (z.B. wenn dieser Boolesche Wert in einer Variable gespeichert wird). Deswegen gibt es Konvertierungsregeln, die einen Booleschen Wert in einen numerischen Wert (oder Bitwert) konvertieren und anders herum. Falsch wird generell als `0` kodiert (also als numerischer Wert 0 oder als Bitwert 0). Wahr sind dementsprechend alle Werte, die nicht 0 sind (also der Bitwert 1 und jeder mögliche numerische Wert ungleich 0). Diese Unterscheidung mag ein bisschen übertrieben klingen, ist aber sehr wichtig, besonders wenn Vergleichsoperationen und logische Operationen verwendet werden.

Datentypkonvertierung

Diese Konvertierung von numerischen Werten zu Booleschen Werten ist eigentlich nur ein kleiner Einblick in ein größeres Thema: die *Datentypkonvertierung*. Generell ist es möglich, Operationen auf unterschiedliche Datentypen anzuwenden, um zum Beispiel eine *long*-Zahl zu einer *short*-Zahl zu addieren oder um eine *signed*-Zahl mit einer *unsigned*-Zahl zu vergleichen. Es gibt zwei Arten von Datentypkonvertierungen: implizite und explizite. Implizite Datentypkonvertierungen werden automatisch ohne spezielle Markierung vom Programmierer vom Compiler eingefügt, wenn z.B. unterschiedliche Datentypen im selben Ausdruck vorkommen. Explizite Datentypkonvertierungen werden vom Programmierer eingefügt. Es ist nur selten notwendig, explizite Datentypkonvertierungen einzuführen, aber besonders auf einem Mikrocontroller wie dem Arduino-Prozessor, der über eine begrenzte Menge an Speicher verfügt und intern mit 8 Bit rechnet, ist es in fortgeschrittenen

Fällen notwendig, genau zu kontrollieren, welche Datentypen eingesetzt werden.

Die einfachste Art von implizierter Konvertierung geschieht bei der Zuweisung einer Variable. Hier kann es zwei Möglichkeiten geben. Im unproblematischen Fall umfasst der Zieldatentyp den Quelldatentyp. Im problematischen Fall ist der Zieldatentyp kleiner als der Quelldatentyp (z.B. wenn ein *unsigned*-Wert in einen *signed*-Wert konvertiert wird, oder wenn ein *int*-Datentyp in einen *byte*-Datentyp konvertiert wird). In diesem Fall wird der Wert trunkiert, je nachdem, welche Konvertierung vorgenommen wird. Belegt der Zieldatentyp weniger Platz im Speicher, wird einfach der obere Teil des Wertes verworfen, was zu interessanten Ergebnissen führen kann, wie im folgenden Beispiel gezeigt wird. Wird eine signierte Konvertierung vorgenommen, wird der alte Wert jetzt signiert interpretiert, was unter Umständen zu einer kompletten Umkehrung des Wertes führen kann (siehe das Beispiel). Es ist bei Konvertierungen also kritisch zu überprüfen, dass keine fehlerhaften Konvertierungen passieren.

Beispiel

```
byte b = 5;
int i;
i = b; // Der byte-Wert von b wird in einen int-Wert konvertiert.
i = 260;
b = i; // Der int-Wert von i wird in einen byte-Wert konvertiert.
       // Da int zwei Byte belegt, wird das oberste Byte einfach
       // verworfen. b hat jetzt den Wert 4.
unsigned byte b2 = 160;
signed byte b3 = b2; // Hier wird 160 als signierte Zahl
                     // interpretiert.
                     // Der Wert von b3 ist -96.
b2 = -80; // Hier wird -80 als unsignierte Zahl interpretiert.
          // Der Wert von b2 ist 176.
int i2 = 60 * 100; // Der Wert von i2 ist 6000.
int i3 = 60 * 1000; // Der Wert von i3 ist -5536 (!).
```

Werden unterschiedliche Datentypen im selben Ausdruck verwendet, konvertiert der Compiler sie automatisch in den »größten« Datentyp, der alle benutzten Argumente und das Ergebnis der Operation beinhalten kann. Wird eine *byte*-Zahl zu einer weiteren *byte*-Zahl addiert, ist das Ergebnis eine *int*-Zahl, weil das Ergebnis unter Umständen (z.B. 250 + 8) nicht mehr in das eine Speicherbyte einer *byte*-Zahl passen würde. Wird eine *byte*-Zahl zu einer *int*-Zahl addiert, ist das Ergebnis eine *long*-Zahl, weil wie im vorigen Beispiel das Ergebnis nicht mehr in die zwei Speicherbytes einer *int*-Zahl passen würde.

Besonders knifflig wird die Thematik, wenn *signed*-Werte und *unsigned*-Werte im selben Ausdruck verwendet werden. Dadurch wird der Ausdruck zu einem signierten Ausdruck, und es kann bei der Zuweisung des Ergebnisses oder beim Vergleich mit unsignierten Werten schnell zu unerwarteten Ergebnissen kommen. Solche Fehler sind besonders schwer herauszufinden, weil sie beim Lesen des Programms nicht sofort ersichtlich sind. Falls also bei Rechenausdrücken oder Vergleichen (insbesondere in Schleifen) unerwartete Werte auftreten, ist es manchmal notwendig, die Ausdrücke ganz genau unter die Lupe zu nehmen, und zur Not explizite Datentypkonvertierungen zu verwenden.

Einer der häufigsten Fehler ist der Vergleich von *signed*- und *unsigned*-Zahlen, der immer wahr oder immer falsch ist und zum Beispiel zu nie endenden Schleifen führen kann. Solche simplen Fehler werden häufig vom Compiler erkannt, aber nur als *Warning* gekennzeichnet, sodass man sie in der Arduino-Umgebung, wenn die *verbose*-Kompilation nicht angeschaltet ist, nicht zu sehen bekommt.

Explizite Datentypkonvertierungen werden vom Programmierer eingefügt. Hier wird vor dem zu konvertierenden Ausdruck in Klammern der Zieldatentyp angegeben. Es gelten die gleichen Regeln und Warnungen wie für implizite Datentypkonvertierungen.

Beispiel

```
short s = 2;
int i = (int)s; // explizite Konvertierungen von short nach int
```

Präzedenzregeln

Bei Ausdrücken gibt es in der Arduino-Programmiersprache festgelegte Präzedenzregeln (in welcher Reihenfolge Ausdrücke berechnet werden). Generell gelten die üblichen arithmetischen Regeln (Multiplikation und Division vor Addition und Subtraktion), ansonsten wird der Ausdruck von links nach rechts ausgewertet. Das ist bei längeren Ausdrücken schwer nachzuvollziehen, insbesondere wenn Operationen verwendet werden, die programmiersprachenspezifisch sind. Es ist möglich, die Auswertungsreihenfolge explizit zu beschreiben, indem man runde Klammern benutzt. Ausdrücke innerhalb von Klammern werden als Erstes berechnet und ihr Ergebnis im übergeordneten Ausdruck eingesetzt. Runde Klammern können beliebig geschachtelt werden.

Auch hier ist es manchmal angebracht, der Lesbarkeit halber den Ausdruck in mehrere einzelne Ausdrücke aufzutrennen, die in lokale Variablen gespeichert werden. Oft hat das keinen Einfluss auf die Geschwindigkeit eines Programms.

Beispiel

```
int i = 5 + 2 * 6 - 8 / 2; // Der Wert von i ist 13.
int i2 = ((5 + 2) * 6 - 8) / 2; // Der Wert von i2 ist 17.
// Hier wird der Ausdruck von i2 in mehrere einzelne lokale
// Variablen aufgetrennt.
int tmp1 = 5 + 2;
int tmp2 = tmp1 * 6 - 8;
int i3 = tmp2 / 2; // Der Wert von i3 ist 17.
```

Arithmetische Operationen

Die arithmetischen Operationen in der Arduino-Programmiersprachen sind die üblichen mathematischen Operationen: Addition, Subtraktion, Multiplikation und Division.

Eine spezielle arithmetische Operation ist die Modulo-Operation, die den Restwert einer ganzzahligen Division berechnet. Die Modulo-Operation ist auf Fließkommazahlen nicht definiert. Die Modulo-Operation wird häufig benutzt, um einen Wert innerhalb eines bestimmten Bereichs zu halten (z.B. innerhalb der Größe einer Tabelle). Die Modulo-Operation mit einem Teiler, der eine Zweierpotenz ist (also 2, 4, 8, 16, 32 usw.), lässt sich auf einem Computer besonders effizient ausführen. Es ist also aus Geschwindigkeitsgründen wichtig, bei häufig benutzten Tabellen, auf die mithilfe der Modulo-Operation zugegriffen wird, eine Zweierpotenz als Größe zu wählen.

Beispiel

```
byte b = 5 % 3; // Der Wert von b ist 2, weil 2 der Restwert von 5
                // geteilt durch 3 ist.
byte b2 = 65 % 20; // Der Wert von b2 ist 5, weil 5 der Restwert
                   // von 65 geteilt durch 20 ist.
int tabelle[64]; // eine Tabelle mit 64 Werten
int index = 40;
tabelle[index % 64] = 5; // So ist gewährleistet, dass der
                         // Indexwert immer innerhalb der Tabelle
                         // ist.
tabelle[index % 64]; // Diese Operation ist um ein Vielfaches
                     // schneller als:
tabelle[index % 63];
```

Wird die Divisionsoperation auf ganzzahlige Wert angewendet, ist das Ergebnis auch eine ganzzahlige Zahl, es wird also die ganzzah-

lige Division angewendet. Soll eine Fließkommazahl berechnet werden, muss mindestens eins der Divisionsargumente eine Fließkommazahl sein (also vom Datentyp *float* oder *double*). Das lässt sich entweder durch explizite Datentypkonvertierung erreichen, oder indem literale Werte mit einem Komma (geschrieben als Punkt) angegeben werden (zur Not mit ».0«).

Beispiel

```
byte b = 5 / 2; // Der Wert von b ist 2.
float f = 5 / 2; // Der Wert von f ist 2.0.
float f2 = 5.0 / 2.0; // Der Wert von f2 ist 2.5.
```

In der folgenden Tabelle werden alle arithmetischen Operationen zur Referenz aufgelistet.

Tabelle C-3 ▶
Arithmetische Operationen

Zeichen	Name	Erklärung
+	Addition	addiert zwei Zahlen
-	Subtraktion	subtrahiert zwei Zahlen
*	Multiplikation	multipliziert zwei Zahlen
/	Division	teilt zwei Zahlen (wenn beide Argumente ganzzahlig sind, ist die Division ganzzahlig)
%	Modulo	Berechnet den Restwert der ganzzahligen Division von zwei Zahlen

Logische Operationen

Logische Operationen rechnen mit Bitwerten, also mit 0 und 1. Da sich aber einzelne Bitwerte nicht direkt auf dem Arduino-Prozessor speichern lassen, arbeiten die logischen Operationen mit den einzelnen Bits der normalen numerischen Datenwerte (also mit der Bitmaskendarstellung, siehe Abschnitt Datentypen). Die logische Operation wird auf jedes einzelne Bit des numerischen Wertes getrennt angewendet.

Das logische UND (&) und das logische ODER (|) ähneln den Booleschen Operatoren UND (&&) und ODER (||), und beide Operatoren sind leicht zu verwechseln. Die logischen Operationen arbeiten jedoch auf numerische Werte und beeinflussen jedes einzelne Bit dieser Werte, während die Booleschen Operatoren mit Booleschen Werten arbeiten (0 ist falsch, alles andere ist wahr). Dazu werden numerische Argumente erst in Boolesche Werte konvertiert (siehe oben im Abschnitt »Datentypkonvertierung« in diesem Anhang).

Viele der logischen Operatoren werden benutzt, um einzelne Bits auf 0 oder auf 1 zu setzen. Deswegen bietet die Arduino-Programmierumgebung ein Funktion, um eine Bitmaske zu erzeugen, in der

ein einziges Bit gesetzt ist. Diese Funktion heißt `bit(bitNummer)`. Hier muss beachtet werden, dass `bitNummer` bei 0 anfängt, also `bit(0)` verwendet wird, um eine Bitmaske zu erzeugen, in der das unterste (also das letzte) Bit auf 1 gesetzt ist.

Beispiel

```
byte a = bit(4); // a wird auf 00010000 gesetzt.
a = bit(0); // a wird auf 00000001 gesetzt.
```

In der folgenden Tabelle sind alle logische Operationen zur Referenz aufgelistet.

◀ Tabelle C-4
Logische Operationen

Zeichen	Name	Erklärung
&	Und	logisches UND, Maskierung (`1 & 1 = 1`, sonst `0`)
|	Oder	logisches ODER (`0 | 0 = 0`, sonst `1`)
^	Exklusiv-Oder	exklusives ODER (gibt `1` zurück, wenn genau eins der Argumente `1` ist)
~	Bitweise negieren	invertiert jedes Bit (aus 1 wird 0, aus 0 wird 1)
<<	Shift nach links	bewegt alle Bits um eine Stelle nach links und füllt mit `0` auf
>>	Shift nach rechts	bewegt alle Bits um eine Stelle nach rechts und füllt mit dem Sign-Bit nach, wenn notwendig, sonst mit `0`

& : Logisches UND (Maskierung)

Die &-Operation ist der logische UND-Operator. Er gibt `1` zurück, wenn seine beiden Argumente auch `1` sind. Seine Funktionsweise lässt sich am leichtesten mit der folgenden Tabelle erläutern. Das &-Zeichen muss immer mit Leerzeichen abgetrennt werden, da es in der Arduino-Programmiersprache eine fortgeschrittene (und ganz andere) Bedeutung hat, wenn es direkt an einem Wort hängt .

◀ Tabelle C-5
Logische UND-Operation

a	b	a & b
0	0	`0 & 0 = 0`
0	1	`0 & 1 = 0`
1	0	`1 & 0 = 0`
1	1	`1 & 1 = 1`

In den folgenden Beispielen wird der logische UND-Operator auf numerische Datenwerte angewendet (die obige Funktionstabelle wird auf jedes einzelne Bit der Datenwerte angewendet).

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte b = 3; // b ist 00000011 in Binärdarstellung.
byte c = a & b; // c ist a & b = 00000101 & 00000011 = 00000001 = 1.
a = 0xFE; // a ist 11111110 in Binärdarstellung.
c = a & b; // c ist a & b = 11111110 & 00000011 = 00000010 = 2.
```

Üblicherweise wird der Maskierungsoperator verwendet, um einzelne Bits aus einem Wert zu extrahieren oder zu testen. So kann man das letzte Bit aus einem Wert (oder einem Eingangsport des Arduino) extrahieren, indem man ihn mit 1 maskiert. Das letzte Bit in einem Wert kennzeichnet, ob der Wert gerade oder ungerade ist, so lässt sich also mit dem Maskierungsoperator bestimmen, ob eine Zahl gerade oder ungerade ist, ohne sie durch 2 teilen zu müssen.

Beispiel

```
byte a = 5;
if (a & 1) {
   // a ist ungerade, da das letzte Bit gesetzt ist.
} else {
   // a ist gerade, da das letzte Bit nicht gesetzt ist.
}
```

Ähnlich lassen sich z.B. die vier unteren Bits oder die vier oberen Bits aus einem Wert extrahieren (das ist manchmal notwendig, wenn z.B. ein Sensor zwei 4-Bit-Werte in einem Byte zurückgibt). Um die Werte auch richtig zu bearbeiten, wird oft noch der Shift-Operator (siehe Abschnitt über Shift-Operator) eingesetzt, um die extrahierten Bits an die unterste Stelle zu bewegen.

Beispiel

```
byte a = 0xfe;
byte untereBits = (a & 0xF); // extrahiert die 4 unteren Bits, da
                             // 0xF = 00001111
byte obereBits = (a & 0xF0) >> 4; // extrahiert die 4 oberen Bits,
                                  // da 0xF0 = 11110000
```

Mit dem Maskierungsoperator ist es auch möglich, bestimmte Bits auf 0 zu setzen. Im folgenden Beispiel wird gewährleistet, dass das oberste Bit der Variable `value` auf 0 gesetzt wird.

Beispiel

```
value = value & 0x7F; // Das oberste Bit von value ist immer 0, da
                      // 0x7F = 01111111.
```

Oft werden auch Masken als Konstanten definiert (siehe Abschnitt über Konstanten), um den Programmtext besser lesbar zu machen. Die obigen Beispiele lassen sich auch wie folgt schreiben:

Beispiel

```
const byte UntereBitsMaske = 0x0F;
const byte ObereBitsMaske = 0xF0;
byte a = 0xfe;
byte untereBits = (a & UntereBitsMaske);
byte obereBits = (a & ObereBitsMaske) >> 4;
const byte ValueByteMaske = 0x7F; // Diese Maske wird auf alle
                                  // Values angewendet.
value = value & ValueByteMaske;
```

|: Logisches ODER

Das | ist der logische ODER-Operator. Er gibt 1 zurück, wenn mindestens eins seiner Argumente auch 1 ist. Seine Funktionsweise lässt sich am besten mit der folgenden Tabelle erläutern.

◀ **Tabelle C-6**
Logische ODER-Operation

a	b	a|b
0	0	0 | 0 = 0
0	1	0 | 1 = 1
1	0	1 | 0 = 1
1	1	1 & 1 = 1

In den folgenden Beispielen wird der logische ODER-Operator auf numerische Datenwerte angewendet (die obige Funktionstabelle wird auf jedes einzelne Bit der Datenwerte angewendet).

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte b = 3; // b ist 00000011 in Binärdarstellung.
byte c = a | b; // c ist a | b = 00000101 | 00000011 = 00000111 = 7.
a = 0x8E; // a ist 10001110 in Binärdarstellung.
c = a | b; // c ist a | b = 10001110 | 00000011 = 10001111 = 0x8F.
```

Üblicherweise wird der ODER-Operator verwendet, um einzelne Bits in einem Wert auf 1 zu setzen. Hier lässt sich die Funktion `bit(bitNummer)` einsetzen, um gezielt einzelne Bits auf 1 zu setzen.

Beispiel

```
value = value | 1; // Das unterste Bit in value wird auf 1
                   // gesetzt.
byte a = bit(0) | bit(2) | bit(4); // a wird auf 00010101 gesetzt.
```

^: Exklusives ODER

Das ^ ist der exklusive ODER-Operator. Er gibt 1 zurück, wenn genau eins seiner Argumente auch 1 ist. Seine Funktionsweise lässt sich am besten mit der folgenden Tabelle erläutern .

Tabelle C-7 ▶
Exklusive ODER-Operation

a	b	a ^ b
0	0	0 ^ 0 = 0
0	1	0 ^ 1 = 1
1	0	1 ^ 0 = 1
1	1	1 ^ 1 = 0

In den folgenden Beispielen wird der exklusive ODER-Operator auf numerische Datenwerte angewendet (die obige Funktionstabelle wird auf jedes einzelne Bit der Datenwerte angewendet).

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte b = 3; // b ist 00000011 in Binärdarstellung.
byte c = a ^ b; // c ist a ^ b = 00000101 ^ 00000011 = 00000010 = 2.
a = 0x8E; // a ist 10001110 in Binärdarstellung.
c = a ^ b; // c ist a ^ b = 10001110 ^ 00000011 = 10000001 = 0x81.
```

Üblicherweise wird der exklusive ODER-Operator verwendet, um einzelne Bits umzukehren, ohne andere Bits zu beeinflussen. Das ist besonders praktisch, um zum Beispiel eine bestimmte LED bei jedem Durchlaufen einer Funktion blinken zu lassen (beim ersten Durchlauf anschalten, beim nächsten Durchlauf ausschalten, dann wieder anschalten usw.), wenn der Wert dieser LED zusammen mit den Werten anderer LEDs in einer Variable gespeichert wird (wie bei unserer LED-Matrix, siehe).

Beispiel

```
byte ledZeile = 0; // ledZeile speichert den Wert von 8 LEDs
void toggleLed() {
    ledZeile = ledZeile ^ 1; // die unterste LED in ledZeile wird
                             // invertiert
}
```

~: Logische Negierung

Die Tilde ~ ist der logische Negierungsoperator, der ein einzelnes Argument annimmt (das auf eine rechte Seite geschrieben wird). Er gibt 1 zurück, wenn er auf 0 angewendet wird, und 0, wenn er auf 1 angewendet wird, wie der folgenden Tabelle zu entnehmen ist.

Tabelle C-8 ▶
Logische Negierung

a	~a
0	~0 = 1
1	~1 = 0

In den folgenden Beispielen wird die logische Negierung auf numerische Datenwerte angewendet (die obige Funktionstabelle wird auf jedes einzelne Bit der Datenwerte angewendet).

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte c = ~a; // c ist ~a = ~00000101 = 11111010 = 0xFA.
a = 0x8E; // a ist 10001110 in Binärdarstellung.
c = ~a; // c ist ~a = ~10001110 = 01110001 = 0x71.
```

Üblicherweise wird der Negierungsoperator verwendet, um jedes Bit in einem Wert umzukehren.

<<: Shift nach links

Die <<-Operation verschiebt die Bits in einem Wert um eine bestimmte Anzahl von Stellen nach links. (Auf Englisch heißt verschieben »to shift«, diese Operation wird auch auf Deutsch üblicherweise »shiften« genannt.) Die dadurch »entstehenden« Stellen werden mit Nullen gefüllt. Bits, die nicht mehr in den Datentyp hineinpassen, verschwinden.

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte b = a << 2; // b ist 00010100 in Binärdarstellung,
                 // der Wert von a wurde um 2 Stellen nach links
                 // verschoben.
byte c = 0x7F; // c ist 01111111.
byte d = c << 3; // d ist 11111000.
```

In Kombination mit der |-Operation lassen sich so beliebige Bitmasken erstellen. Eine eigene Implementierung der `bit(bitNummer)`-Funktion (die allerdings nur für `byte`-Bitmasken funktioniert) könnte in etwa so aussehen wie im folgenden Beispiel (um mehr zu Funktionsdefinition zu erfahren, siehe den Abschnitt Funktionen).

Beispiel

```
byte meinBv(uint8_t bitNummer) {
   return (1 << bitNummer);
}
```

Da auf einem Computer Zahlen in Binärdarstellung gespeichert werden, ist die Shift-nach-links-Operation eine effiziente Art, Werte mit einer Zweierpotenz zu multiplizieren. Einen Wert um eine Stelle nach links zu verschieben, entspricht einer Multiplikation mit der Zahl 2. Einen Wert um zwei Stellen nach links zu verschieben, entspricht einer Multiplikation mit der Zahl 4 usw. Deswegen lassen sich Multiplikationen mit Zweierpotenzen oft schneller ausführen als allgemeine Multiplikationen.

Beispiel

```
byte a = 5;
byte b = a << 2; // b ist gleich 5 * 4 = 20.
```

>>: Shift nach rechts

Die >>-Operation verschiebt die Bits in einem Wert um eine bestimmte Anzahl von Stellen nach rechts. Im Unterschied zur Shift-nach-links-Operation werden allerdings die »entstehenden« Stellen nicht immer mit Nullen gefüllt. Wegen der Zweierkomplement-Darstellung (siehe Abschnitt über Datentypen) ist bei negativen Zahlen das oberste Bit auf 1 gesetzt. Um das Vorzeichen eines Wertes beim Shiften nach rechts zu erhalten, wird dementsprechend bei negativen Zahlen mit Einsen aufgefüllt. Deswegen ist Vorsicht beim Shiften nach rechts geboten, weil unerwartete Werte herauskommen können.

Um zu vermeiden, dass mit dem obersten Bit bei negativen Zahlen aufgefüllt wird, kann der Wert vor dem Shiften mit einer expliziten Datentypkonvertierung zuerst in einen *unsigned*-Wert konvertiert werden.

Beispiel

```
byte a = 5; // a ist 00000101 in Binärdarstellung.
byte b = a >> 2; // b ist 00000001 in Binärdarstellung,
                 // der Wert von a wurde um zwei Stellen nach
                 // rechts verschoben. Da a keine signed Nummer
                 // ist, wird hier mit 0 aufgefüllt.

signed byte c = -16; // c ist 11110000 in Binärdarstellung.
signed byte d = c >> 3; // d ist 11111110 in Binärdarstellung.
signed byte e = (unsigned)c >> 3; // d ist 00011110 in
                                  // Binärdarstellung.
```

Ähnlich wie die >>-Operation entspricht die Shift-nach-rechts-Operation einer ganzzahligen Division durch eine Zweierpotenz.

Beispiel

```
byte a = 100;
byte b = a >> 2; // b ist gleich 100 / 4 = 25.
```

Mit dem >>-Operator und dem logischen UND (&) lassen sich einzelne Bits aus einem Wert extrahieren und in einem anderen speichern.

Beispiel

```
byte b = (a >> 2) & 1; // Das dritte Bit von a wird in b
                       // gespeichert.
```

Vergleichsoperationen

Vergleichsoperationen sind eine der am häufigsten eingesetzten Arten von Operationen, insbesondere kombiniert mit dem Einsatz

von Kontrollstrukturen (siehe Abschnitt von Kontrollstrukturen). Mit Vergleichsoperationen lassen sich zwei Werte vergleichen. Der Rückgabetyp von Vergleichsoperationen ist ein Boolescher Wert, der sich auch als numerischer Wert einsetzen lässt (wahr ist 1, falsch ist 0).

Der ==-Operator vergleicht, ob zwei Werte gleich sind. Dieser Operator sollte nicht mit dem Zuweisungsausdruck (=) verwechselt werden. Der !=-Operator ist das genaue Gegenteil des ==-Operators und gibt immer »wahr« zurück, wenn zwei Werte unterschiedlich sind.

Beispiel

```
byte a = 255;
byte b = 101;
signed byte c;
a == 255; // ist wahr
a != 255; // ist falsch
a == b; // ist falsch
 a != b; // ist wahr
byte d = a == b; // d hat jetzt den Wert 0.
 c == 128; // ist immer falsch, egal welchen Wert c hat,
                // da der Datentyp von c 128 nicht umfasst
c != 128; // ist immer wahr
a = b; // Das ist eine Zuweisung und kein Vergleich!
char x = 'a';
x == 'a'; // ist wahr
```

Zeichenketten lassen sich allerdings nicht so einfach vergleichen, weil der »eigentliche« Wert einer Zeichenkette ihre Adresse im Speicher ist, nicht die Zeichenketten als solche. Auch Tabellen lassen sich nicht mit Vergleichsoperationen vergleichen.

Beispiel

```
char meineKette[] = "foobar";
meineKette == "foobar"; // ist falsch
int meineTabelle[] = { 1, 2, 3};
int meineTabelle2[] = { 1, 2, 3};
meineTabelle != meineTabelle2; // ist wahr
```

Ähnlich wie ihre verwandten Operatoren in der Mathematik vergleichen die Operatoren > (größer als), < (kleiner als), >= (größer als oder gleich) und <= (kleiner als oder gleich) numerische Werte. Hier sei wieder auf die vom Compiler ausgeführte implizite Typkonvertierung und die möglichen Werte der einzelnen Datentypen hingewiesen.

Beispiel

```
byte a = 5;
a < 10; // ist wahr
a > 2; // ist wahr
```

```
a > 5; // ist falsch
a >= 5; // ist wahr
a < 300; // ist für jeden Wert von a wahr
```

In der folgenden Tabelle sind alle Vergleichsoperationen zur Referenz aufgelistet.

Tabelle C-9 ▶
Vergleichsoperationen

Zeichen	Name
==	gleich
!=	nicht gleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich

Boolesche Operationen

Boolesche Operationen ähneln sehr ihren äquivalenten logischen Operationen. Allerdings arbeiten Boolesche Operationen mit den Wahrheitswerten »wahr« und »falsch«, nicht mit Bits. Mit Booleschen Operationen lassen sich verschiedene Boolesche Ausdrücke (z. B. Vergleichsoperationen) miteinander verknüpfen.

Das Boolesche UND && (im Unterschied zum logischen UND &) gibt »*wahr*« zurück, wenn seine beiden Argumente wahr sind. Das Boolesche ODER || (im Unterschied zum logischen ODER |) gibt »*wahr*« zurück, wenn mindestens eines seiner Argumente wahr ist. Die Boolesche Negierung ! (im Unterschied zur logischen Negierung ~) gibt »*wahr*« zurück, wenn ihr Argument falsch ist, und »*falsch*«, wenn ihr Argument wahr ist.

Da mit Booleschen Operatoren oft kompliziertere Vergleichsoperationen verknüpft werden, ist es meist ratsam, runde Klammern einzusetzen, um die Auswertungsreihenfolge explizit zu formulieren – sonst kann es unter Umständen zu fehlerhaften Ausdrücken kommen (insbesondere wenn mehrere &&- und ||-Operatoren verknüpft werden). Wie man leicht sehen kann, können Boolesche Ausdrücke ähnlich kompliziert werden wie arithmetische Ausdrücke. Auch hier ist es oft hilfreich (und meistens nicht langsamer), Zwischenvariablen einzuführen, um die Ausdrücke zu vereinfachen.

Beispiel

```
byte a = 5;
byte b = 10;
(a < 10) && (a > 4); // ist wahr
```

```
(a > 6) && (b == 10); // ist falsch
(a < b) || (b > 10); // ist wahr
 !(a == 5); // ist falsch
!((a < b) || !(b > 10)); // ist falsch
```

Ein wichtiger Unterschied zu logischen Operatoren ist, wie Boolesche Operatoren ihre Argumente auswerten. »Auswerten« heißt in diesem Fall, dass geprüft wird, ob der Wert des Ausdrucks auch tatsächlich berechnet wird. Das ist insbesondere bei Zuweisungen und Funktionsaufrufen wichtig, da diese Nebenwirkungen haben. Wird ein Ausdruck wie z.B. »2 + 5« oder »a / b« ausgewertet, ändert sich eigentlich gar nichts, außer dass der Arduino-Prozessor ein bisschen Rechenzeit verbraucht hat. Wird allerdings ein Ausdruck wie »a = 5« ausgewertet, wurde a ein neuer Wert zugewiesen.

Bei logischen Operatoren werden beide Argumente immer ausgewertet. Bei Booleschen Operatoren werden allerdings nur so viele Argumente ausgewertet wie nötig. Ist das erste Argument eines Booleschen UND (&&) falsch, braucht das zweite Argument gar nicht erst ausgewertet zu werden, weil das Ergebnis auf jedem Fall falsch sein wird. Ähnlich verhält es sich, wenn das erste Argument eines Booleschen ODER (||) wahr ist, denn dann ist das Ergebnis des Ausdrucks auf jeden Fall wahr.

Beispiel

```
(a == 1) & (b = 5); // In diesem Fall wird b immer auf 5 gesetzt.
(a == 1) && (b = )5; // b wird nur dann auf 5 gesetzt, wenn a == 1.
```

Diese unterschiedlichen Auswertungsstrategien können zu fehlerhaften Programmen führen, wenn sie nicht beachtet werden. Sie lassen sich jedoch auch einsetzen, um bestimmte Kontrollvorgänge knapper zu formulieren, wie im folgenden Abschnitt über Kontrollstrukturen gezeigt wird. In der folgenden Tabelle sind die Booleschen Operationen zur Referenz aufgelistet.

◀ **Tabelle C-10**
Boolesche Operationen

Zeichen	Name	Beschreibung
&&	und	wahr, wenn beide Argumente wahr sind
||	oder	wahr, wenn mindestens eins der Argumente wahr ist
!	negieren	wahr, wenn das Argument falsch ist, falsch, wenn das Argument wahr ist

Zusammengesetzte Operationen

Es kommt häufig vor, dass mit einer Variablen gerechnet und das Ergebnis anschließend wieder in dieser Variablen gespeichert wird. Um das Schreiben solcher Vorgänge zu vereinfachen, gibt es eine

Reihe zusammengesetzter Operationen, die eine Zuweisungsoperation mit einer arithmetischen oder logischen Operation verknüpfen.

Zwei spezielle zusammengesetzte Operation sind die Inkrementierungsoperation, die 1 zu einer Variablen hinzuaddiert, und die Dekrementierungsoperation, die 1 von einer Variablen subtrahiert. Beide werden oft in Schleifen eingesetzt, um einen Wertebereich zu durchlaufen. Bei diesen beiden Operationen gibt es zwei mögliche Schreibweisen. Man kann den Operator vor die zu inkrementierende (oder dekrementierende) Variable schreiben – in diesem Fall spricht man von Prä-Inkrementierung bzw. Prä-Dekrementierung. Wenn man ihn hinter die Variable schreibt, spricht man von Post-Inkrementierung bzw. Post-Dekrementierung. Der Unterschied zwischen beiden Schreibweisen ist der Rückgabewert des Ausdrucks. Im Fall der Prä-Operationen wird zuerst die Variable modifiziert; der Rückgabewert des Ausdrucks ist der Wert der Variablen nach der Inkrementierung oder Dekrementierung. Im Fall der Post-Operationen wird die Variable erst nach der Auswertung der Variablen inkrementiert oder dekrementiert, d.h. der Wert des Ausdrucks ist der Wert der Variablen vor der Modifikation. In den meisten Fällen werden Post-Operationen verwendet.

Inkrementierung und Dekrementierung werden oft bei `while`-Schleifen eingesetzt, die im folgenden Abschnitt erläutert werden.

Beispiel

```
byte a = 5;
byte b = a++;
// b hat jetzt den Wert 5, a hat den Wert 6.
b = ++a;
// b hat jetzt den Wert 7, a hat auch den Wert 7.
```

In der folgenden Tabelle sind die zusammengesetzten Operationen zur Referenz aufgelistet und jeweils mit einem entsprechenden ausgeschriebenen Ausdruck dokumentiert.

Tabelle C-11 ▶
Zusammengesetzte Operationen

Zeichen	Name	Ausdruck	Äquivalenter Ausdruck
`++`	inkrementieren	`a++`	`a = a + 1`
`--`	dekrementieren	`a--`	`a = a – 1`
`+=`	addieren und zuweisen	`a += 4`	`a = a + 4`
`-=`	subtrahieren und zuweisen	`a -= 4`	`a = a – 4`
`*=`	multiplizieren und zuweisen	`a *= 4`	`a = a * 4`
`%=`	teilen und zuweisen	`a *= 4`	`a = a % 4`
`&=`	und und zuweisen	`a *= 4`	`a = a & 4`

Zeichen	Name	Ausdruck	Äquivalenter Ausdruck
`|=`	oder und zuweisen	`a *= 4`	`a = a | 4`
`<<=`	Shift nach links und zuweisen	`a *= 4`	`a = a << 4`
`>>=`	Shift nach rechts und zuweisen	`a *= 4`	`a = a >> 4`

Kontrollstrukturen

Bis jetzt wurden Datentypen, Variablen und Ausdrücke vorgestellt. Mit diesen Konstrukten lassen sich Werte berechnen, es fehlt aber noch eines der grundlegenden Fundamente der Programmierung: Kontrollstrukturen. Ohne Kontrollstrukturen ist es kaum möglich, auf unterschiedliche Daten (aus externen Sensoren, dem Datenspeicher oder der seriellen Schnittstelle) zu reagieren.

Kontrollstrukturen steuern den Programmfluss, also in welcher Reihenfolge der Arduino-Prozessor das Programm abarbeitet. Ein Algorithmus (also eine Folge von Anweisungen, die eine bestimmte Aufgabe ausführen) lässt sich oft auf unterschiedliche Arten implementieren (also als Programm niederschreiben). Viele der Kontrollstrukturen der Programmiersprache lassen sich durch andere ersetzen. Die Auswahl der benutzten Kontrollstrukturen wird durch mehrere Parameter beeinflusst: Effizienz (wie schnell kann der Arduino-Prozessor diese Anweisungen ausführen), Lesbarkeit (je nach Aufgabe sind bestimmte Kontrollstrukturen besser geeignet, und einen Algorithmus zu beschreiben) und Programmierstil (jeder Programmierer hat seine eigenen Vorlieben). Bestimmte Strukturen lassen sich auch leichter debuggen, falls Fehler auftreten, manche anderen lassen sich später leichter modifizieren.

In diesem Abschnitt stellen wir die unterschiedlichen Kontrollstrukturen der Arduino-Programmiersprache vor und gehen auf ihre jeweiligen Einsatzbereiche, Vor- und Nachteile ein.

if

Die `if`-Kontrollstruktur ist die einfachste Kontrollstruktur in der Arduino-Programmiersprache und auch eine der am häufigsten benutzten. Mit ihr lässt sich ein Programmblock ausführen, wenn eine bestimmte Bedingung zutrifft. Diese Bedingung ist ein Boolescher Ausdruck (also ein Ausdruck, der als Boolescher Wert interpretiert wird). Meistens ist dieser Ausdruck ein Vergleichsoperator, wie in den folgenden Beispielen. Die Syntax der `if`-Kontrollstruktur ist folgende:

```
if (Boolescher Ausdruck) {
   // Dieser Block wird nur dann ausgeführt, wenn der Boolesche
   // Ausdruck wahr ist.
}
```

Beispiel

```
if (a == 5) {
 // Dieser Funktionsaufruf wird nur ausgeführt, wenn die Variable
 // a den Wert 5 hat.
Serial.println("a ist 5");
}
if ((a < 5) || (b > 10)) {
 // Dieser Funktionsaufruf wird nur ausgeführt,
 // wenn a kleiner als 5 ist oder wenn b größer als 10 ist.
 Serial.println(»a ist kleiner als 5 oder b ist größer als 10.");
}
```

Es lassen sich natürlich beliebige Ausdrücke innerhalb der runden Klammern nach dem `if` benutzen. Dabei werden die in Abschnitt »Datentypkonvertierung« in diesem Anhang vorgestellten Konvertierungsregeln verwendet. So ist es also möglich, z. B. ein bestimmtes Bit zu testen und nur dann ein Block auszuführen, wenn dieses Bit gesetzt ist.

Beispiel

```
if (a & bit(3)) {
  Serial.println("Das vierte Bit von a ist gesetzt.");
}
if (ledMatrix[3] & bit(4)) {
  // Die LED wird angeschaltet, wenn das fünfte Bit von
  // ledMatrix[3] gesetzt ist.
  digitalWrite(ledPin, HIGH);
}
```

if … else

Die `if ... else`-Kontrollstruktur ist eine Erweiterung der `if`-Kontrollstruktur. Mit ihr es möglich, die Alternative zu beschreiben, falls die Bedingung falsch ist, ohne gleich eine zweite `if`-Struktur benutzen zu müssen. Die einfache Syntax der `if ... else`-Kontrollstruktur ist folgende:

```
if (Boolescher Ausdruck) {
  // Dieser Block wird nur dann ausgeführt, wenn der Boolesche
  // Ausdruck wahr ist.
} else {
  // Dieser Block wird nur dann ausgeführt, wenn der Boolesche
  // Ausdruck falsch ist.
}
```

Wenn der Boolesche Ausdruck wahr ist, wird der erste Block ausgeführt. Diesen Block nennt man auch den `if`-Zweig. Ist der Boolesche Ausdruck allerdings falsch, wird der erste Block übersprungen und der zweite Block, der nach dem `else` definiert wird, ausgeführt. Diesen zweiten Block nennt man auch den `else`-Zweig.

Es ist auch möglich, die geschweiften Klammern wegzulassen, wenn ein Block nur aus einer einzigen Anweisung besteht. Das bringt allerdings keine Vorteile (das Programm wird dadurch nicht kleiner), aber eine ganze Reihe von Problemen mit sich. Erstens wird dadurch der Programmtext schwerer zu lesen, weil nicht mehr klar ersichtlich ist, wo der `if`-Zweig aufhört und wo der `else`-Zweig anfängt. Wird später einer der Zweige mit weiteren Anweisungen ergänzt, müssen die geschweiften Klammern hinzugefügt werden. Besonders haarig wird es, wenn mehrere `if ... else`-Ausdrücke geschachtelt werden. Spätestens in diesem Fall ist es notwendig, geschweifte Klammern einzuführen, weil sonst nicht mehr nachvollziehbar ist, welches `else` zu welchem `if` gehört.

So lassen sich die vorigen Beispiele wie folgt ergänzen.

Beispiel

```
if (a & bit(3)) {
  Serial.println("Das vierte Bit von a ist gesetzt.");
} else {
  Serial.println("Das vierte Bit von a ist nicht gesetzt.");
}
if (ledMatrix[3] & bit(4)) {
  // Die LED wird angeschaltet, wenn das fünfte Bit von
  // ledMatrix[3] gesetzt ist.
  digitalWrite(ledPin, HIGH);
} else {
  // Sonst wird die LED ausgeschaltet.
  digitalWrite(ledPin, LOW);
}
```

Die `if ... else`-Kontrollstruktur erlaubt auch die Verknüpfung von mehreren `if`-Kontrollstrukturen in eine große Kontrollstruktur, indem mehrere Bedingungen verknüpft werden. Die verschiedenen Bedingungen werden der Reihe nach ausgewertet, bis eine den Wert »*wahr*« zurückgibt. In dem Fall wird der dazugehörige Programmblock ausgeführt und schließlich das Programm nach dem Ende der `if ... else`-Kontrollstruktur weiter ausgeführt. Trifft keine der Bedingungen zu, wird der `else`-Teil der Kontrollstruktur (falls vorhanden) ausgeführt.

Beispiel

```
if (a > 10000) {
 Serial.println("a ist groß.");
} else if (a > 500) {
 Serial.println("a ist mittelgroß.");
} else if (a > 10) {
 Serial.println("a ist klein.");
} else {
 Serial.println("a ist sehr klein.");
}
```

An diesem Beispiel kann man sehen, dass die Reihenfolge der Bedingungen unter Umständen relevant sein kann. Folgende Schreibweise wäre z. B. falsch, weil sie für alle Werte von a, die größer als 10 sind, »a ist klein.« anzeigen würde. Selbst bei verhältnismäßig einfachen Kontrollstrukturen ist es also leicht, Fehler zu machen. Die meisten Programmfehler rühren von fehlerhaften Kontrollstrukturanweisungen her.

Beispiel

```
if (a > 10) {
 Serial.println("a ist klein.");
} else if (a > 500) {
 Serial.println("a ist mittelgroß.");
} else if (a > 10000) {
 Serial.println("a ist groß.");
} else {
 Serial.println("a ist sehr klein.");
}
```

Komplexe bzw. sehr lange `if ... else`-Strukturen lassen sich oft mit der `switch`-Kontrollstruktur einfacher beschreiben.

Es ist natürlich möglich, mehrere `if` und `if ... else` zu schachteln. Dazu ist es wichtig, den Programmtext sauber einzurücken (beim Schreiben oder mit Apfel-T bzw. Steuerung-T), damit dieser gut lesbar bleibt.

Beispiel

```
if (a > 10) {
 if (b > 10) {
  Serial.println("a und b sind größer als 10.");
 } else {
 Serial.println("a ist größer als 10, b aber nicht.");
 }
} else {
 if (b > 10) {
  Serial.println("a ist kleiner oder gleich 10, aber b ist größer
                 als 10.");
 } else {
  Serial.println("a und b sind kleiner oder gleich 10.");
 }
```

Besondere Vorsicht ist beim Einsatz von Booleschen Operatoren in `if ... else`-Ausdrücken geboten. Der `else`-Zweig wird nur ausgeführt, wenn der komplette Boolesche Ausdruck falsch ist, was zu nicht unbedingt sofort ersichtlichen Fallunterscheidungen führt (besonders bei langen Booleschen Ausdrücken.)

Beispiel

```
if ((a > 10) && (b > 10)) {
  Serial.println("a und b sind größer als 10.");
} else {
  Serial.println("a ist kleiner oder gleich 10, oder b ist kleiner
                 oder gleich 10.");
  Serial.println("Es kann aber sein, dass a größer ist als 10.");
  Serial.println("Es kann auch sein, dass b größer ist als 10.");
}
```

switch ... case

Die `switch ... case`-Kontrollstruktur kann man als spezialisierte `if ... else`-Kontrollstruktur betrachten. Ein Ausdruck wird mit einer Reihe von Werten verglichen, und falls er mit einem dieser Werte gleich ist, wird ein bestimmter Programmblock ausgeführt. Ist der Ausdruck mit keinem der angegebenen Werten gleich, wird ein Default-Programmblock (falls vorhanden) ausgeführt. Die Syntax der `switch ... case`-Kontrollstruktur ist folgende:

```
switch (Ausdruck) {
   case Wert1:
      programmBlock1;
      break;
   case Wert2:
      programmBlock2;
      break;
   case Wert3:
      programmBlock3;
   break;
   default:
      defaultProgrammBlock;
      break;
}
```

Die `break`-Anweisung bewirkt einen Sprung ans Ende der `switch ... case`-Kontrollstruktur. Falls die `break`-Anweisung fehlt, wird der nächste Programmblock in der Struktur ausgeführt, was in speziellen Fällen praktisch sein kann (man sagt, dass das Programm in den nächsten `switch ... case`-Block »fällt«). Meistens ist das aber eher eine Fehlerquelle, weshalb es wichtig ist, sicherzustellen, dass die `break`-Anweisung am Ende der einzelnen Programmblöcke auch wirklich vorhanden ist.

Es ist sinnvoll, immer einen `default`-Block zu definieren, der undefinierte Fälle abfängt und zur Not einen Fehler anzeigt oder das Programm neu initialisiert. Damit lassen sich beim Entwickeln Fehler abfangen.

Programmblöcke sind einfache Listen von Anweisungen, die mit einem Semikolon getrennt werden. Es ist auch möglich, nach einer `case`-Anweisung lokale Variablen zu definieren, indem der Programmblock mit geschweiften Klammern abgeschlossen wird.

```
switch (Ausdruck) {
   case Wert1: {
      int lokaleVariable;
      programmBlock1;
   }
    break;
   case Wert2:
      programmBlock2;
      break;
```

`switch ... case`-Kontrollstrukturen lassen sich natürlich auch schachteln. Auch dabei ist es wichtig, den Programmtext sauber einzurücken, da er sonst sehr schwer zu lesen sein kann.

Eine `switch ... case`-Kontrollstruktur benutzt den üblichen ==-Vergleichsoperator. Deswegen ist es nicht möglich, Zeichenketten mit einer `switch ... case`-Struktur zu unterscheiden.

Beispiel

```
char zeichenKetten[] = "arduino";
// Falsch! Zeichenketten und Tabellen lassen sich nicht mit switch
// ... case bearbeiten.
switch (zeichenKette) {
   case "foobar":
      break;
   case "arduino":
      break;
}
```

Ein häufiger Einsatzbereich von `switch ... case`-Kontrollstrukturen ist die Fallunterscheidung zwischen unterschiedlichen Modi eines Programms (z.B. ob das Programm sich gerade im Konfigurationsmodus oder im Betriebsmodus befindet). Dazu wird eine Statusvariable deklariert, die eine bestimmte Anzahl von vordefinierten Werten annehmen kann. Diese Werte werden oft als Konstanten deklariert, damit der Programmquelltext besser zu lesen ist. Im folgenden Beispiel wird vorausgesetzt, dass zwei LEDs und ein Taster am Arduino angeschlossen sind. `blinkErsteLed()`, `blinkZweiteLed()` und `buttonPressed()` sind Funktionen, die nicht gezeigt, sondern nur als Beispiele benutzt werden.

Beispiel

```
const int BetriebsModus = 0;
const int KonfigurationsModus = 1;
int programmStatus = BetriebsModus;
void loop () {
   switch (programmStatus) {
      case BetriebsModus:
         // Im Betriebsmodus wird die erste LED geblinkt.
         blinkErsteLed();
         if (buttonPressed()) {
            // Wurde der Taster gedrückt, wird in den
            // Konfigurationsmodus gewechselt.
            programmStatus = KonfigurationsModus;
         }
         break;
      case KonfigurationsModus:
         // Im Konfigurationsmodus wird die zweite LED geblinkt.
         blinkZweiteLed();
         if (buttonPressed()) {
            // Wurde der Taster gedrückt, wird in den
            // Betriebsmodus gewechselt.
            programmStatus = BetriebsModus;
         }
         break;
      default:
         // Falls die programmStatus-Variable durch einen Fehler
         // einen unbekannten Wert annimmt, wird diese auf den
         // Betriebsmodus zurückgesetzt.
         programmStatus = BetriebsModus;
         break;
   }
}
```

Die `switch ... case`-Kontrollstruktur lässt sich auch einsetzen, um unterschiedliche Kommandos zu erkennen, die z.B. als Bytes über die serielle Schnittstelle vom Computer aus an den Arduino gesendet werden (siehe Abschnitt serielle Schnittstelle). Im folgenden Beispiel werden drei Kommandos erkannt, die ebenfalls die hier nicht gezeigten Funktionen `blinkLed()`, `motorAn()` und `motorAus()` aufrufen.

Beispiel

```
const int BlinkLedCommand = 10;
const int MotorAnCommand = 11;
const int MotorAusCommand = 12;
void loop() {
   if (Serial.available()) {
      switch (Serial.read()) {
         case BlinkLedCommand:
            blinkLed();
            break;
```

```
            case MotorAnCommand:
               motorAn();
               break;
            case MotorAusCommand:
               motorAus();
               break;
            default:
               // unbekanntes Kommando ignorieren
               break;
         }
      }
   }
```

for

Die for-Kontrollstruktur wird eingesetzt, um einen Programmblock mehrmals in einer Schleife auszuführen. Oft wird diese Struktur gemeinsam mit einem Schleifenzähler eingesetzt, um den Programmblock eine bestimmte Anzahl von Malen zu wiederholen oder eine Tabelle oder Zeichenkette zu durchlaufen (siehe Abschnitt über Tabellen und Zeichenketten). Die Syntax der for-Kontrollstruktur sieht so aus:

```
for (Initialisierung; Schleifenbedingung; Schleifenanweisung) {
   // Schleifencode
}
```

Zu Beginn der Schleife wird die Initialisierungsanweisung ausgeführt. Mit dieser Anweisung kann z. B. die Schleifenvariable initialisiert werden. Bei jedem Durchlauf wird der Programmcode innerhalb des Schleifenblocks ausgeführt. Nach jedem Schleifendurchlauf wird die Schleifenanweisung ausgeführt (die oft dazu verwendet wird, den Schleifenzähler zu inkrementieren). Letztendlich wird die Schleifenbedingung ausgewertet. Ist der Rückgabewert der Schleifenbedingung »*wahr*«, wird die Schleife erneut durchlaufen. Ist er »*falsch*«, wird die Schleife abgebrochen, und das Programm wird nach dem Ende der for-Kontrollstruktur weiter ausgeführt. Im folgenden Beispiel wird die LED 10 Mal ein- und wieder ausgeschaltet.

Beispiel

```
int i; // i wird als Schleifenvariable verwendet.
for (i = 0; i < 10; i++) {
 digitalWrite(ledPin, HIGH); // LED wird angeschaltet.
 delay(200); // kurz warten
 digitalWrite(ledPin, LOW); // LED wird ausgeschaltet.
 delay(200);
}
```

In diesem Beispiel wird zu Beginn der Schleife die Variable `i` mit `0` initialisiert. Anschließend wird die LED einmal ein- und ausgeschaltet. Dann wird `i` inkrementiert (wegen der Schleifenanweisung `i++`). Im nächsten Schritt wird der Ausdruck `i < 10` ausgewertet, und falls `i` noch klein genug ist, wird die Schleife weiter ausgeführt.

Selbst in einfachen Schleifen kann es zu Problemen kommen. Es ist besonders wichtig, beim Einsatz von Schleifenvariablen zu überprüfen, ob diese Schleifenvariable den gewünschten Bereich durchläuft (insbesondere, wenn diese Schleifenvariable als Index in einer Tabelle verwendet wird, weil sonst auf den falschen Speicher zugegriffen werden könnte). Wäre der Schleifenkopf wie folgt geschrieben, würde die Schleife entweder nicht genug oder zu oft ausgeführt.

Beispiel

```
// Die Schleife wird nur 9 Mal ausgeführt.
for (i = 1; i < 10; i++) {
}
// Die Schleife wird 11 Mal ausgeführt.
for (i = 0; i <= 10; i++) {
}
```

Diese Art von Fehlern, die insbesondere bei komplizierteren Schleifen auftritt, wird so häufig gemacht, dass sie ihren eigenen Namen verdient hat: Man nennt sie »off by one«-Fehler, weil die Schleifenvariable meistens um genau 1 neben dem beabsichtigten Wert liegt.

Hier kommen auch die ganzen Vorsichtsanweisungen zum Tragen, die bei den verschiedenen Vergleichsoperatoren vorgestellt wurden. Folgende Schleife zum Beispiel wird nie enden (der Schleifenblock wird also immer wieder bis in alle Unendlichkeit wiederholt):

Beispiel

```
int i;
for (i = 0; i < 40000; i++) {
 // i ist ein int, ist also immer kleiner als 40000.
 // Die Bedingung ist also immer erfüllt, und die Schleife bricht
 // nie ab.
}
```

Die `for`-Kontrollstruktur ist sehr flexibel. So lassen sich in der Arduino-Programmiersprache auch direkt Variablen innerhalb der Initialisierungs-Anweisung definieren. Diese Variablen sind nur innerhalb des Schleifenblocks sichtbar. Somit lässt sich immer wieder derselbe Name für eine Schleifenvariable verwenden, ohne dass es zu Konflikten oder Mehrfachbenutzungen kommt.

Beispiel

```
for (int i = 1; i < 10; i++) {
 // Die Variable i ist nur innerhalb des Schleifenblocks sichtbar.
 analogWrite(ledPin, i * 25);
}
```

Es ist auch möglich, mehrere Anweisungen in der Initialisierung oder den Schleifenanweisungen auszuführen. Dazu werden die Anweisungen mit dem sogenannten sequenziellen Operator `,` verknüpft. So können z. B. mehrere Schleifenvariablen initialisiert und inkrementiert werden (es ist allerdings nicht möglich, mehrere Schleifenvariablen mit unterschiedlichen Typen zu deklarieren).

Beispiel

```
for (int i = 1, j = 2; i < 5; i++, j += 2) {
  // i und j durchlaufen die Werte
  // i = 1, j = 2
  // i = 2, j = 4
  // i = 3, j = 6
  // i = 4, j = 8
}
```

Es ist durchaus möglich, mehrere `for`-Kontrollstrukturen zu schachteln. So lassen sich z. B. mehrdimensionale Tabellen durchlaufen. Hier ist wichtig, die Schleifenvariablen nicht zu verwechseln.

Beispiel

```
int ledMatrix[10][10];
for (int x = 0; x < 10; x++) {
  for (int y = 0; y < 10; y++) {
    // Jeder Wert der ledMatrix wird auf 1 gesetzt.
    ledMatrix[x][y] = 1;
  }
}
```

Innerhalb des Schleifenblocks kann der Programmfluss der Schleife auch beeinflusst werden. Die `break`-Anweisung wird benutzt, um aus der Schleife auszusteigen. Das Programm wird anschließend nach dem Ende der `for`-Schleife weitergeführt. Dadurch lassen sich im Schleifenblock weitere Terminierungsbedingungen überprüfen. Der Programmfluss innerhalb der Schleife kann auch durch die `continue`-Anweisung beeinflusst werden. Die `continue`-Anweisung bewirkt, dass ans Ende des Programmblocks innerhalb der Schleife gesprungen wird. Es werden also direkt nach der `continue`-Anweisung die Schleifenanweisung ausgeführt (meistens wird dann der Schleifenzähler inkrementiert), die Schleifenbedingung ausgewertet und, falls diese wahr ist, der nächste Schleifendurchlauf ausgeführt.

Will man zum Beispiel die fünfte Zeile der Matrix überspringen, würde sich folgende Schleife anbieten:

Beispiel

```
int ledMatrix[10][10];
for (int x = 0; x < 10; x++) {
 // fünfte Zeile wird übersprungen
  if (x == 4)
    continue;
  for (int y = 0; y < 10; y++) {
    // Jeder Wert der ledMatrix wird auf 1 gesetzt.
    ledMatrix[x][y] = 1;
  }
}
```

Bei geschachtelten Kontrollstrukturen beziehen sich `break` und `continue` jeweils auf die zuletzt definierte Kontrollstruktur (`break` und `continue` lassen sich auch bei den `switch ... case`-, `while`- und `do ... while`-Kontrollstrukturen einsetzen).

Ein spezielle Schreibweise der `for`-Kontrollstruktur findet man in Arduino-Programmen oft, wenn ein Endpunkt erreicht wurde und das Programm sozusagen beendet werden soll. Da es dem Arduino nicht möglich ist, sich selbst auszuschalten, wird eine unendliche Schleife ausgeführt. Dazu genügt eine `for`-Schleife ohne Abbruchbedingung.

Beispiel

```
for (;;) {
  // unendliche Schleife
}
```

while

Die `while`-Kontrollstruktur ist eine weitere Kontrollstruktur, mit der sich Schleifen implementieren lassen. Im Vergleich zur `for`-Kontrollstruktur, die eher darauf ausgelegt ist, Schleifen mit Zählervariablen zu beschreiben, führt eine `while`-Schleife den Schleifenblock so lange aus, wie die Schleifenbedingung wahr ist. Die Schleifenbedingung wird vor dem ersten Durchlauf der Schleife ausgewertet. Ist sie dort schon falsch, wird die Schleife gar nicht ausgeführt. So lässt sich z. B. eine Schleife ausführen, bis ein Sensor einen bestimmten Wert meldet oder ein Taster betätigt wird. Die Syntax der `while`-Kontrollstruktur ist folgende:

```
while (schleifenBedingung) {
    // Schleifenblock
}
```

Im folgenden Beispiel wird die LED solange an- und ausgeschaltet, bis ein Taster gedrückt wird.

Beispiel

```
while (!tasterGedrueckt()) {
  blinkLed();
}
```

Wie bei der `for`-Schleife kann die `break`-Anweisung benutzt werden, um aus der Schleife herauszugelangen. Die `continue`-Anweisung führt einen neuen Schleifendurchlauf aus (falls die Bedingung wahr ist).

do ... while

Die `do ... while`-Kontrollstruktur ist eng verwandt mit der `while`-Kontrollstruktur. Bei dieser Kontrollstruktur wird die Schleifenbedingung allerdings erst nach dem Schleifenblock ausgewertet, sodass die Schleife mindestens einmal ausgeführt wird. Die Syntax der `do ... while`-Kontrollstruktur ist folgende:

```
do {
    // Schleifenblock
} while (schleifenBedingung);
```

Funktionen

Mit den bis jetzt erklärten Konstrukten lassen sich schon alle möglichen Programme schreiben. Allerdings kommt es relativ oft vor, dass bestimmte Programmabschnitte immer wieder ausgeführt werden sollen, nur mit leicht unterschiedlichen Werten oder Programmanweisungen. Diese immer wieder verwendeten Programmblöcke kann man als getrennte Funktionen schreiben, die aufgerufen werden können (das nennt man Funktionsaufruf, und viele der bisherigen Programmbeispiele haben schon solche Funktionsaufrufe enthalten), das heißt, sie werden von einer anderen Stelle im Programm aus angesprungen und ausgeführt.

Beim *Funktionsaufruf* kann man einer Funktion eine Anzahl von Werten übergeben, die man *Parameter* nennt. Mit Parametern lässt sich das Verhalten der Funktion steuern. Man kann zum Beispiel Berechnungen mit unterschiedlichen Werten ausführen oder bestimmte Teile der Funktion deaktivieren. Nachdem eine Funktion ausgeführt wurde, springt sie wieder zur Stelle zurück, an der sie aufgerufen wurde, und gibt gegebenenfalls einen Wert zurück (den Rückgabewert).

Dieses Auftrennen von Programmcode ist eins der wichtigsten Konstrukte in Programmiersprachen allgemein. Auf der einen Seite wird dadurch der Programmtext kürzer, weil oft verwendete Abschnitte nur einmal geschrieben werden müssen. So lassen sich auch viele Fehler vermeiden, weil nur noch eine Stelle korrigiert werden muss, wenn ein Problem auftritt. Den Prozess des Extrahierens bestimmter Programmteile und ihre Definition in eigenen Funktionen nennt man »Refaktorieren«. Besonders nach einer längeren Programmierphase ist es oft ganz nützlich, sich das Programm in Ruhe anzugucken und Stellen zu identifizieren, die sich angenehmer als getrennte Funktionen definieren lassen würden. Als Faustregel gilt: Kommt ein Stück Programmcode mehr als zwei Mal vor, wird daraus eine Funktion definiert. Auch sehr lange Funktionen (z. B. mehr als einen Bildschirm lang) sollte man in kleinere Funktionen auftrennen, die jeweils bestimmte Aufgaben erledigen. Dadurch wird das Programm viel klarer zu lesen.

Auch das erzeugte Programm, das auf dem Arduino gespeichert wird, wird dadurch kleiner. Da jede Funktion auch einen eigenen Namen hat, wird das Programm auch einfacher zu verstehen. Der Programmname erklärt sofort (wenn er gut gewählt wurde), was die Funktion bewirkt. Es ist auch bei sehr kleinen Funktionen (die z.B. nur einen bestimmten Wert, der oft verwendet wird, einer weiteren Funktion übergeben) sinnvoll, eine eigene Funktion zu definieren, weil der Compiler diese effizienter bearbeiten kann.

Funktionen erhöhen auch die Wiederverwendbarkeit von Programmcode. Oft werden in einem Programm Funktionen benutzt, die auch im Kontext eines anderen Programms sinnvoll sind. Man kann also Funktionen von einem Programm zum anderen wiederverwenden. Viele sinnvolle Funktionen sind schon in der Arduino-Programmiersprache enthalten: Das sind auf der einen Seite die eingebauten Funktionen der Arduino-Umgebung, die in den Workshops ab Kapitel 3 vorgestellt und beschrieben werden, auf der anderen Seite die Funktionen, die in zusätzlichen Libraries enthalten sind und eine bestimmte Funktionalität implementieren (z. B. die Kommunikation mit einem Sensor).

Funktionen sind auch der beste Weg, um Programme oder Programmteile mit anderen Leuten auszutauschen. Es ist nicht mehr notwendig, ein Stück Programmcode genau zu erklären, sondern es reicht, die generelle Funktionsweise einer Funktion zu beschreiben sowie ihre Argumente und ihren Rückgabewert. Damit weiß jeder Programmierer Bescheid, wie die Funktion einzusetzen ist. Auch

zur eigenen Referenz ist es sinnvoll, einen kurzen Kommentar vor die Funktion zu schreiben, der die Argumente und den Rückgabewert beschreibt. In diesem Kommentar sollte man auch mögliche Nebenwirkungen der Funktion auflisten (z.B. dass sie einen bestimmten Pin in der Richtung ändert).

Funktionsaufruf

Das Aufrufen einer Funktion ist ein Ausdruck (und keine Anweisung), d.h. er hat einen Wert, der in weiteren Ausdrücken eingesetzt werden kann. Es ist allerdings auch möglich, den Rückgabewert einer Funktion zu ignorieren (das sollte man allerdings mit Vorsicht tun, oft wird der Rückgabewert benutzt, um z.B. Fehler zu signalisieren). Die Syntax eines Funktionsaufrufs sieht so aus:

```
funktionsName(argumente);
```

Die Argumente sind eine Folge von Werten, die an die Funktion übergeben wird. Sie werden mit einem einfachen Komma getrennt. Es ist wichtig, die richtige Anzahl von Argumenten zu geben, da der Compiler sonst einen Fehler meldet (bestimmte Funktionen können eine variable Anzahl von Argumenten bearbeiten, was zum Beispiel im Anhang Libraries deutlich wird. Es ist auch wichtig, dass die einzelnen Argumente den passenden Datentyp haben. Dazu werden die gleichen Konvertierungsregeln benutzt, die in Abschnitt »Datentypkonvertierung« vorgestellt wurden. Genauso werden bei der weiteren Bearbeitung des Rückgabewerts (z.B. Speichern in einer Variablen) dieselben Konvertierungsregeln angewandt. Im Folgenden werden einige Funktionsaufrufe vorgestellt.

Beispiel

```
machNichts(); // ruft die Funktion machNichts auf, die keine
              // Argumente braucht
byte knopfWert = knopfGedrueckt(); // Der Rückgabewert von
                                   // knopfGedrueckt wird in
                                   // knopfWert gespeichert.
int x = analogRead(sensorEinsPin) + 2 * analogRead(sensorZweiPin);
do {
 machWas(); // führt die Funktion machWas aus, solange der Knopf
            // nicht gedrückt wurde
} while (!knopfGedrueckt())
```

Funktionsdefinition

Das Hinschreiben einer Funktion (mit Namen, Parametern und Programmrumpf) nennt man Funktionsdeklaration oder Funktionsdefinition. Eine Funktionsdeklaration ähnelt stark einer Vari-

ablendeklaration. Es ist allerdings nicht möglich, Funktionen innerhalb von beliebigen Programmblöcken zu deklarieren (im Gegensatz zur Deklaration von lokalen Variablen). Funktionen lassen sich nur auf der obersten Ebene des Programmtextes definieren. Im Unterschied zu Variablendefinitionen können allerdings Funktionen aufgerufen werden, *bevor* sie definiert werden.

Die Syntax einer Funktionsdeklaration ist folgende:

```
returntyp funktionsName(argumentenListe) {
   programmRumpf;
}
```

Ähnlich wie eine Variable besitzt eine Funktion einen Datentyp, den man Rückgabetyp nennt. Dieser Typ beschreibt den Datentyp der Rückgabewerte der Funktion. Der Rückgabetyp einer Funktion umfasst alle numerischen Datentypen, die im Abschnitt Datentypen vorgestellt wurden. Es ist jedoch nicht möglich, eine Tabelle oder eine Zeichenkette aus einer Funktion zurückzugeben; das funktioniert nur mit Pointern, die ein fortgeschrittenes Konstrukt der Arduino-Programmiersprache sind und keinen Platz in diesem Buch gefunden haben. Es kommt noch ein weiterer spezieller Rückgabetyp hinzu: der `void`-Datentyp. `void` ist eigentlich kein Datentyp, sondern kennzeichnet, dass eine Funktion keinen Wert zurückgibt.

Funktionsnamen folgen denselben Regeln wie Variablennamen: Sie müssen mit einem Buchstaben anfangen, dürfen Buchstaben, Zahlen und Unterstriche (_) enthalten, und sie dürfen nicht mit schon definierten Namen kollidieren (also mit reservierten Wörtern der Programmiersprache und mit schon definierten Funktionsnamen und Variablennamen).

In der Argumentenliste, die in runden Klammern nach dem Funktionsnamen geschrieben wird, werden die Argumente definiert, die die Funktion annimmt (und bearbeitet). Argumente ähneln stark den lokalen Variablen, die beim Funktionsaufruf initialisiert werden und innerhalb des Funktionsrumpfes definiert sind (man kann nicht auf Argumente außerhalb der Funktion zugreifen, sondern sie nur beim Funktionsaufruf initialisieren). In der Argumentenliste werden Argumente wie Variablen mit einem Namen und einem Datentyp definiert. Mehrere Argumentdefinitionen werden durch Kommata voneinander getrennt. Es ist auch möglich, eine Funktion ohne Argumente zu definieren. Dazu wird eine leere Argumentenliste verwendet (die runden Klammern sind aber nach wie vor notwendig). Bei einem Funktionsaufruf wird der erste übergebene

Parameterwert in das erste Argument kopiert, der zweite Parameterwert in das zweite Argument usw.

Es ist auch möglich, Tabellen und Zeichenketten als Argumente zu übergeben. Dazu wird der Tabellentyp in eckigen Klammern angegeben. Unten sehen Sie verschiedene Argumentenlisten als Beispiele. Oft ist es bei Tabellen notwendig, der Funktion auch noch die Tabellengröße zu übergeben (bei Zeichenketten kann die Funktion die abschließende Null erkennen), es sei denn, alle Tabellen, die der Funktion als Argumente übergeben werden, haben dieselbe Größe.

Funktionsargumente werden als Werte übergeben (diese Eigenschaft nennt man in der Programmierwelt »call-by-value«). Wird also an einer Stelle eine Variable als Argument übergeben und dieses Argument innerhalb der Funktion modifiziert, wird die ursprüngliche Variable nicht modifiziert. Stattdessen wird beim Aufruf der Funktion eine lokale Kopie der Variablen erzeugt und im Funktionsrumpf verwendet. Das ist ein bisschen gewöhnungsbedürftig (und nicht unbedingt leicht zu verstehen), erhöht aber die Wiederverwendbarkeit von Funktionen und verhindert auch viele Programmfehler.

Funktionen können deswegen nur den globalen Status eines Programms (globale Variablen) verändern (und natürlich den Zustand des Arduino-Prozessors). Weitere Veränderungen müssen vom Programmierer explizit angegeben werden, indem zum Beispiel der Rückgabewert der Funktion gespeichert wird. Nach dem Zurückkehren aus einer Funktion werden alle lokalen Variablen (und alle Argumente) gelöscht und sind nicht mehr zugänglich. Es ist also nicht möglich, einen lokalen Wert als Speicherplatz zu verwenden, der von Funktionsaufruf zu Funktionsaufruf bestehen bleibt. Dazu muss eine globale Variable verwendet werden.

Beispiel

```
int zaehlHochFalsch() {
   // Diese Funktion gibt immer 1 zurück, weil bei jedem
   // Aufruf a neu initialisiert wird.
   int a = 0;
   a++;
   return a;
}
int a = 0;
int zaehleHochRichtig() {
   // Diese Funktion gibt bei jedem Aufruf einen hochgezählten
   // Wert zurück (also 1, 2, 3 usw.). Nach 32767 wird der
   // Wert -32768, weil die Variable a übergelaufen ist.
```

```
    a++;
    return a;
}
```

Es ist allerdings besser, die Anzahl von globalen Variablen, die innerhalb einer Funktion verwendet werden, zu begrenzen. Dadurch wird nämlich nicht die Wiederverwendbarkeit der Funktion eingeschränkt: Auf der einen Seite hängt die Funktion dann von mehreren globalen Variablen ab, weshalb es nicht mehr möglich ist, sie ohne Weiteres in ein neues Programm einzusetzen. Auf der anderen Seite steigt durch globale Variablen auch die Anzahl von Nebenwirkungen der Funktion, d.h. der Programmierer, der die Funktion aufruft, muss jetzt immer berücksichtigen, dass dadurch mehrere globale Variablen verändert werden können. Werden globale Variablen benutzt, sollte man sich also überlegen, ob diese durch weitere Argumente oder durch einen Rückgabewert ersetzt werden können.

Vorsicht ist allerdings bei Tabellen und Zeichenketten geboten. Diese können innerhalb der Funktion doch modifiziert werden, weil sie beim Aufruf der Funktion nicht kopiert werden; stattdessen wird der Funktion ein Zeiger auf die Tabelle oder die Zeichenkette übergeben (das nennt man in der Programmierwelt »call-by-reference«). Es erhöht die Geschwindigkeit der Funktion, weil keine Daten kopiert werden, kann aber zu Programmfehlern führen, wenn Tabellen unabsichtlich modifiziert werden. Auf der anderen Seite kann genau diese Eigenschaft benutzt werden, um Tabellen zu modifizieren. Zum Beispiel werden viele der Funktionen im Workshop zur LED-Matrix in Kapitel 4 verwendet, um die gespeicherte Tabelle der Matrix zu modifizieren.

In seltenen Fällen ist es notwendig, mehrere Werte aus einer Funktion zurückzugeben. Dazu kann man einzelne Argumente als »call-by-reference« übergeben (wird im Funktionsrumpf einem Argument ein Wert zugewiesen, wird auch in die ursprüngliche Variable geschrieben). Dazu wird der Referenz-Operator `&` benutzt. Dieser muss direkt vor den Argumentnamen in der Argumentenliste geschrieben werden (ohne Leerzeichen zwischen `&` und Variablennamen). Diese Schreibweise ist nur in sehr seltenen Fällen notwendig und sollte sparsam eingesetzt werden.

Beispiel

```
int machWas(int a) { // a ist ein call-by-value-Argument.
    a = a + 10; // Nur die lokale Kopie von a wird verändert.
    return a;
}
```

```
int machWasMitReferenz(int &a) { // a ist jetzt ein call-by-
reference-Argument.
    a = a + 10; // Auch die ursprüngliche wird verändert.
}
int foo = 0;
machWas(foo);
// foo ist immer noch 0.
foo = machWas(foo);
// foo ist jetzt 10.
machWasMitReferenz(foo);
// foo ist jetzt 20.
foo = machWas(foo);
// foo ist jetzt 30 (wird aber zweimal zugewiesen).
```

Der Funktionsrumpf einer Funktion beinhaltet alle Programmieranweisungen der Funktion. Diese werden in geschweiften Klammern hinter dem Funktionsnamen und der Argumentenliste angegeben. Bei einem Funktionsaufruf werden die Anweisungen wie ein normales Stück Programmcode ausgeführt. Erreicht der Prozessor die letzte Zeile des Funktionsrumpfes (und damit die schließende geschweifte Klammer), und hat die Funktion keinen Rückgabewert (wurde also mit dem Datentyp *void* definiert), wird aus der Funktion zurückgekehrt. Das Programm wird nach dem ursprünglichen Funktionsaufruf weiter ausgeführt.

Die `return`-Anweisung wird verwendet, um aus einer Funktion zurückzukehren. Diese Anweisung lässt sich also einsetzen, um an beliebigen Stellen innerhalb des Funktionsrumpfes (und nicht nur am Ende der Funktion) aus der Funktion auszubrechen. Gibt die Funktion einen Wert zurück (hat also nicht den Datentyp *void*), geschieht das mit dem Wort `return` und dem Wert oder Namen der Variable, die zurückgegeben werden soll.

Innerhalb eines Funktionsrumpfes ist es möglich, lokale Variablen zu definieren, wie in jedem Programmblock, der durch geschweifte Klammern abgeschlossen ist. In einer Funktion kann man natürlich auch weitere Funktionen aufrufen.

Beispiel

```
// Dies ist eine sehr einfache Funktion, die nichts macht und auch
// keinen Wert zurückgibt. Sie dient nur als Beispiel und hat
// keinen praktischen Nutzen.
void machNichts() {
}
// Diese Funktion macht auch gar nichts, gibt allerdings den Wert
// 5 (vom Datentyp int) zurück. Sie dient auch nur als Beispiel
// und hat keinen praktischen Nutzen.
int fuenf() {
    return 5;
```

```
}
// Diese Funktion schaltet die LED aus, die an ledPin hängt. Diese
// Funktion ist an sich auch nicht sehr nützlich. Allerdings wird
// beim Ersetzen der ledAus()-Funktion deutlich, was der
// Programmcode bewirkt.
void ledAus() {
    digitalWrite(ledPin, LOW);
}
// Diese Funktion addiert 5 zum Argument x hinzu und gibt diesen
// Wert zurück.
int addierFuenf(int x) {
    return x + 5;
}
// Diese Funktion schaltet die LED an, wenn a größer ist als 10.
int ledAn(int a) {
    if (a > 10) {
       digitalWrite(ledPin, HIGH);
    } else {
       return;
    }
}
// Diese Funktion addiert 10 zu x hinzu, wenn x positiv ist, sonst
// gibt sie 0 zurück. x wird hier nur innerhalb der Funktion
// modifiziert.
int addierZehnPositiv(int x) {
    if (x > 0) {
      x = x + 10;
      return x;
    } else {
      return 0;
    }
}
```

Bei einem Funktionsaufruf wird der Funktionsrumpf einer Funktion immer ausgeführt. Es ist also oft sinnvoll, den Rückgabewert einer Funktion zu speichern, anstatt immer wieder die Funktion mit denselben Argumenten aufzurufen. Dabei sollte natürlich berücksichtigt werden, welche Nebenwirkungen eine Funktion hat (z.B. ob sie bestimmte Werte auf die serielle Schnittstelle schreibt). Generell ist es aber möglich, Ausdrücke deutlich effizienter zu gestalten, wie Sie bei den Beispielen sehen können.

Beispiel

```
// Anstatt des Ausdrucks
int x = square(y) + square(y); // y-Quadrat wird zweimal berechnet
// ist das hier effizienter:
int y2 = square(y); // y-Quadrat wird nur einmal berechnet
int x = y2 + y2;
```

Rückgabewerte werden auf zwei unterschiedliche Arten benutzt. Im ersten Fall wird die Funktion hauptsächlich benutzt, um einen

Wert zu berechnen. Das Ergebnis dieser Berechnung wird als Rückgabewert an die aufrufende Programmstelle zurückgegeben. Der Rückgabewert wird aber auch oft benutzt, um den Status der Ausführung einer Funktion zurückzugeben (um zum Beispiel zu signalisieren, dass ein Fehler bestimmte andere Ereignisse eingetreten sind). Für einen einfachen Rückgabestatus »erfolgreich« bzw. »nicht erfolgreich« reicht ein `boolean`-Rückgabetyp. Dadurch kann man auch gleich den Funktionswert in einer Kontrollstruktur einsetzen, wie im folgenden Beispiel gezeigt wird. Hier können Sie außerdem sehen, wie man eine Funktion flexibler gestaltet, indem man ihr einen zusätzliches Argument übergibt.

Beispiel

```
boolean istKnopfEinsGedrueckt() {
  if (digitalRead(knopfEinsPin) == HIGH) {
    return true;
  } else {
    return false;
  }
}
// Diese Funktion kann man auch flexibler schreiben, indem man ihr
// die Nummer des Pins übergibt, an dem ein Knopf angeschlossen ist.
boolean istKnopfGedrueckt(int knopfPin) {
  if (digitalRead(knopfPin) == HIGH) {
    return true;
  } else {
    return false;
  }
}
```

Es ist aber auch möglich, einen numerischen Wert als Rückgabetyp zu verwenden, um unterschiedliche Statuswerte zu signalisieren. So ist es z. B. möglich, verschiedene Fehlerquellen zu signalisieren. Es ist sinnvoll, diese unterschiedlichen Statuswerte als Konstanten zu deklarieren, um das Programm besser lesbar zu machen. Diese Rückgabewerte lassen sich am besten mit einer `switch ... case`-Kontrollstruktur bearbeiten. Es ist oft auch praktisch, Fehlerwerte als negative Werte zu definieren und Erfolge als positive Werte, weil man so mit einem einfachen arithmetischen Vergleich bestimmen kann, ob ein Fehler aufgetreten ist.

Beispiel

```
const int KeinKnopfGedruecktFehler = -1;
const int NurKnopfEinsGedruecktFehler = -2;
const int NurKnopfZweiGedruecktFehler = -3;
const int BeideKnoepfeGedruecktErfolg = 1;
byte sindBeideKnoepfeGedrueckt() {
```

```
  if (!istKnopfGedrueckt(knopfEinsPin) &&
    !istKnopfGedrueckt(knopfZweiPin)) {
  return KeinKnopfGedrueckt Fehler;
  } else if (istKnopfGedrueckt(knopfEinsPin)) {
    return NurKnopfEinsGedrueckt Fehler;
  } else if (istKnopfGedrueckt(knopfZweiPin)) {
    return NurKnopfZweiGedrueckt Fehler;
  } else {
    return BeideKnoepfeGedrueckt Erfolg;
  }
}
// Die folgenden Zeilen zeigen einen möglichen Aufruf von
// sindBeideKnoepfeGedrueckt.
int retWert = sindBeideKnoepfeGedrueckt();
if (retWert < 0) {
   // Der Rückgabewert ist negativ, also ist ein Fehler
   // aufgetreten, der jetzt gemeldet wird.
   switch (retWert) {
     case KeinKnopfGedrueckt Fehler:
       Serial.println("Es ist kein Knopf gedrueckt worden.");
       break;
     case NurKnopfEinsGedrueckt:
       Serial.println("Es ist nur Knopf eins gedrueckt worden.");
       break;
     case NurKnopfZweiGedrueckt:
       Serial.println("Es ist nur Knopf zwei gedrueckt worden.");
       break;
     }
} else {
     // Der Rückgabewert ist positiv, es wurden also beide Knöpfe
     // gedrückt.
     Serial.println("Es sind beide Knoepfe gedrueckt worden.");
}
```

Es ist möglich, aus einer Funktion heraus dieselbe Funktion nochmal aufzurufen. Man kann auch eine zweite Funktion aufrufen, die wiederum die erste aufruft. Diese Programmiertechnik nennt man *rekursive Funktionsaufrufe*. Sie ist allerdings mit großer Vorsicht zu genießen und bei den meisten Arduino-Programmen nicht notwendig. Durch jeden Funktionsaufruf werden neue temporäre Variablen angelegt und ein bisschen Speicher belegt. Dadurch kann es schnell zu einem Überlauf des Speichers kommen.

Sketch-Struktur

Ein Arduino-Programm wird *Sketch* genannt. Ein Sketch wird im Arduino-Editor geschrieben und als Ordner gespeichert, der alle Daten und Programmdateien enthält, die für das Ausführen notwendig sind. Die Programmdateien haben die Endung *.pde*.

In der Hautpdatei des Sketches müssen zwei Funktionen auf jeden Fall implementiert werden: `setup()` und `loop()`.

Die Funktion `setup()` wird beim Programmstart aufgerufen (also wenn das Arduino-Board eingeschaltet oder Reset gedrückt wird). In dieser Funktion werden Variablen initialisiert, Pinmodi gesetzt sowie benutzte Libraries und zusätzliche Hardware initialisiert. `setup()` wird nur einmal ausgeführt.

Beispiel

```
int buttonPin = 2;
int meinZaehler;
void setup()
{
   meinZaehler = 0;
   pinMode(buttonPin,   INPUT);
   Serial.begin(9600);
}
```

Zwischen die geschweiften Klammern werden nun alle Befehle geschrieben, die vor dem Start des Hauptprogramms zum Einrichten des Arduino benötigt werden. Dazu gehört die Festlegung einzelner Pins als Ein- oder Ausgang. In diesem Beispiel hier sind das die Zeile `pinMode(buttonPin, INPUT)`, die Initialisierung zusätzlicher Libraries (hier die Initialisierung der seriellen Schnittstelle auf 9.600 Baud) und das Initialisieren von Variablen (hier das Initialisieren von `meinZaehler` auf `0`).

Die Setup-Routine wird nur ein einziges Mal ausgeführt, wenn das Board neu an eine Stromquelle (oder per USB an den Rechner) angeschlossen oder neuer Code hochgeladen wird.

Loop

Die Funktion `loop()` wird auch als Hauptfunktion bezeichnet. Von hier aus werden andere Bestandteile des Programms aufgerufen und Befehle abgearbeitet. Wie der Name schon verrät, läuft `loop()` in einer Schleife, das heißt sie beginnt immer wieder von vorn, sobald sie durchlaufen wurde. In dieser Funktion wird meistens die gesamte Funktionalität des Sketches implementiert. Einzige Ausnahme ist die Funktionalität, die über Interrupt-Routinen implementiert wird. Interrupts sind allerdings ein fortgeschrittenes Konstrukt von Arduino und werden meistens in Libraries gekapselt.

Beispiel

```
void loop()
{
   if (meinZaehler > 100) {
```

```
    serialWrite("ueberLauf");
    meinZaehler = 0;
  } else {
    meinZaehler++;
  }
  if (digitalRead(buttonPin == LOW)) {
    serialWrite('H');
  } else {
    serialWrite('L');
  }
  delay(1000);
}
```

Mehrere Dateien in einem Sketch

Meistens wird ein Arduino-Programm in einer Datei gespeichert, die auch den Namen des Sketches trägt (mit der Endung *.pde*). Bei komplizierteren Programmen wird diese Datei allerdings ziemlich lang, und es ist oft schwer, einzelne Funktionen oder Variablendeklarationen wiederzufinden. Deswegen ist es möglich, im Arduino-Editor mehrere Dateien anzulegen, die vor dem Kompilieren zu einer großen Datei zusammengefügt werden. So können in einem größeren Programm unterschiedliche Bereiche in getrennte Dateien geschrieben werden (z. B. eine Datei für alle LED-Funktionen, eine für die serielle Kommunikation und eine für die Kommunikation mit einem Sensor). Dazu muss für jede neue Datei ein neuer Tab angelegt werden, indem der Knopf mit dem Pfeilsymbol auf der rechten Seite gedrückt wird (nicht der Knopf mit dem Pfeilsymbol in der Hauptleiste, der verwendet wird, um einen Sketch auf das Arduino-Board hochzuladen). Danach erscheint ein Menü, mit dem die einzelnen Tabs (sprich Dateien) verwaltet werden können. Zum Anlegen einer neuen Datei muss NEW TAB ausgewählt werden. Der Arduino-Editor fragt nach einem Namen für den Tab und speichert die so erzeugte Datei im Verzeichnis des Arduino-Programms.

Beim Kompilieren des Arduino-Programms werden alle Tabs zu einer großen Datei zusammengefügt und anschließend übersetzt. Das kann zu Problemen führen, wenn in einer Datei globale Variablen benutzt werden, die in einer anderen Datei definiert wurden (die Reihenfolge der Variablendefinition ist nämlich wichtig, siehe Abschnitt Variablen), weil die Reihenfolge des Zusammenfügens sich nicht steuern lässt. Es ist deswegen sinnvoll, globale Variablen so zu definieren, dass sie nur in einer Datei verwendet werden, und als Schnittstelle zwischen den einzelnen Dateien Funktionen zu definieren. Das gilt auch für Programmabschnitte, die in `loop()`

und `setup()` verwendet werden. Eine Strukturierung, die sich als sinnvoll erwiesen hat, ist, in jeder Datei passende Funktionen `loopDateiname()` und `setupDateiname()` zu definieren, die dann in den eigentlichen `loop()`- und `setup()`-Funktionen aufgerufen werden. Zugriff auf globale Konfigurationswerte wird in sogenannte *Getter* und *Setter* gekapselt. Eine Getter-Funktion liefert nur den Wert einer Variablen zurück, während eine Setter-Funktion den Wert einer Variablen setzt. Im folgenden Beispiel werden der Zugriff auf die globale Variable `ZeitInterval` gekapselt und eine zweite Datei *Zeit.pde* benutzt.

Beispiel

```
// Dies ist die Hauptdatei des Sketches.
int zeitInterval = 10; // globale Konfigurationsvariable
int getZeitInterval() { // Getter-Funktion zum Lesen von
                        // zeitInterval
   return zeitInterval;
}
void setZeitInterval(int wert) { // Setter-Funktion zum Schreiben
                                 // von zeitInterval
   zeitInterval = wert;
}
void setup() {
   setupZeit(); // Aufruf der setup-Funktion in der Datei Zeit.pde
}
void loop() {
   loopZeit(); // Aufruf der loop-Funktion in der Datei Zeit.pde
}
// Dies ist die Datei Zeit.pde des Sketches.
void setupZeit() {
   setZeitInterval(20); // Setzen von zeitInterval auf 20 in der
                        // Initialisierungsphase
}
void loopZeit() {
   delay(getZeitInterval()); // Hauptroutine der Datei Zeit.pde mit
                             // Lesen der zeitInterval-
                             // Konfigurationsvariable
}
```

Funktionsreferenz

In diesem Abschnitt werden die in der Arduino-Programmiersprache vordefinierten Funktionen vorgestellt. Sie decken eine breite Menge von unterschiedlichen Bereichen ab, von der Kommunikation mit der elektronischen Außenwelt (digitale und analoge Ein- und Ausgabe) über Zeitfunktionen (Zeitmessen und Warten) und mathematische Funktionen bis hin zur Kommunikation mit dem Computer über die serielle Schnittstelle (bzw. die USB-Schnittstelle).

Digitale Ein- und Ausgabe

Die Funktionen für digitale und analoge Ein- und Ausgabe ermöglichen die Kommunikation des Arduino mit der Außenwelt, indem Werte von Pins eingelesen und auf diese Pins geschrieben werden können.

pinMode(pin, mode)

Diese Funktion initialisiert die »Richtung« eines digitalen Pins. `pin` ist die Nummer des Pins, wie sie auf dem Arduino-Board geschrieben steht. Jeder Pin, auch die als »analog« markierten, kann als digitaler Eingangs- und Ausgangspin initialisiert werden. `.mode` definiert die Richtung und kann entweder `INPUT` sein, um den Pin als Eingangspin zu initialisieren, oder `OUTPUT`, um den Pin als Ausgangspin zu initialisieren.

Beispiel

```
pinMode(13, OUTPUT); // initialisiert Pin 12 als Ausgangspin
pinMode(9, INPUT); // initialisiert Pin 9 als Eingangspin
```

digitalWrite(pin, value)

Diese Funktion setzt die Spannung eines als Ausgangspin definierten digitalen Pins oder aktiviert die Pull-up-Funktionalität für Pull-up-Widerstände eines als Eingangspin definierten digitalen Pins.

Wird der Wert des digitalen Ausgangspins auf HIGH gesetzt, legt der Arduino-Prozessor eine Spannung von 5 Volt (bzw 3,3 auf Boards mit 3,3-Volt-Versorgungsspannung) an den Pin an. Wird der Wert auf LOW gesetzt, legt der Arduino-Prozessor eine Spannung von 0 Volt an.

Ist der Pin als Eingangspin definiert, wird beim Schreiben eines HIGH-Wertes der interne Pull-up-Widerstand des Arduino aktiviert, und der Pin wird auf die Versorgungsspannung »hochgezogen«. Wird auf den Eingangspin der Wert LOW geschrieben, wird der Pull-up-Widerstand deaktiviert.

Beispiel

```
pinMode(13, OUTPUT); // initialisiert Pin 12 als Ausgangspin
digitalWrite(13, HIGH); // Jetzt liegen 5V an Pin 13 an.
digitalWrite(13, LOW); // Jetzt liegen 0V an Pin 13 an.
pinMode(9, INPUT); // initialisiert Pin 9 als Eingangspin
digitalWrite(0, HIGH); // Der Pull-up-Widerstand auf Pin 9 ist aktiviert.
digitalWrite(0, LOW); // Der Pull-up-Widerstand auf Pin 9 ist deaktiviert.
```

int digitalRead(pin)

Mit dieser Funktion lassen sich digitale Werte (HIGH oder LOW) auslesen, die an einem digitalen Eingangspin anliegen. Der Pin muss vorher als digitaler Eingangspin festgelegt worden sein. Liegt keine definierte Spannung am Eingangspin an (z.B. wenn nichts angeschlossen wurde), kann `digitalRead` einen zufälligen Wert zurückgeben.

Beispiel

```
pinMode(9, INPUT); // initialisiert Pin 9 als Eingangspin
int val = digitalRead(9);
```

Analoge Ein- und Ausgabe

Auf dem Arduino-Prozessor gibt es eine Reihe von Pins, die sich als analoge Eingänge konfigurieren lassen (auf dem Arduino Duemilanove sind es die Pins 14 bis 19). Im Unterschied zu digitalen Eingängen können diese den kompletten Spannungsbereich (und nicht nur 0 und 5V) auslesen. Dazu wird intern auf dem Arduino-Prozessor ein sogenannter *AD-Wandler* eingesetzt, der analoge Werte in digitale konvertiert. Dieser AD-Wandler hat eine Auflösung von 10 Bit, d.h. Eingangsspannung 0 bis zur Referenzspannung (meistens 5V, siehe aber `analogReference`) werden zum numerischen Bereich 0 bis 1.023 konvertiert, es ist also möglich, mit einer Genauigkeit von 5 / 1.024 = 4.9mV Spannungen zu messen. Hier sei angemerkt, dass diese Eingänge oft sehr empfindlich für Rauschen sind, die Genauigkeit entspricht also nur in den seltensten Fällen der theoretisch möglichen. Analoge Pins lassen sich auch problemlos als digitale Ein- und Ausgangspins einsetzen. Das umgekehrte Vorgehen ist leider nicht möglich.

int analogRead(pin)

Mit `analogRead` wird die Spannung, die an einem analogen Eingangspin anliegt, ausgelesen. Der Rückgabewert geht von 0 (0V) bis 1.023 (Referenzspannung). Das Einlesen analoger Werte dauert deutlich länger als das Einlesen digitaler Werte, was unter Umständen berücksichtigt werden muss.

Wie bei `digitalRead` werden bei einem Pin, an dem keine definierte Spannung anliegt, einigermaßen zufällige Werte zurückgegeben (je nachdem, wie nah der Benutzer am Arduino ist, wie hoch die Luftfeuchtigkeit ist und ob mit einem Gummihuhn über der Schaltung gewedelt wird).

Beispiel

```
pinMode(9, INPUT); // Initialisiert Pin 9 als Eingangspin
int val = digitalRead(9);
```

analogWrite(pin, value)

Obwohl die `analogWrite`-Funktion in der analogen Kategorie aufgelistet wird, wird sie konkret benutzt, um den Arbeitszyklus eines PWM-Pins festzulegen. Wie in Kapitel 3 beschrieben, wird bei PWM-Pins (auf dem Arduino Duemilanove sind das die Pins 3, 5, 6, 9, 10 und 11) periodisch zwischen HIGH und LOW alterniert. Mit `analogWrite` lässt sich die Länge der HIGH-Periode einstellen (aber nicht die Periodendauer).

analogReference(type)

Mit `analogReference` lässt sich die Referenzspannung, die der AD-Wandler benutzt, umstellen. Dies ist eine fortgeschrittene Funktion und sollte mit großer Vorsicht eingesetzt werden, weil man damit den Arduino-Prozessor beschädigen kann. Die Referenzspannung bestimmt, welche Spannung den Wert 1.023 beim Aufruf von `analogRead()` bekommt. Ist die Referenzspannung 5 Volt, wird 0 Volt als 0 gelesen, 5 als 1.023, und 2,5 als 512. Ist die Referenzspannung allerdings 1 Volt, wird 0 Volt als 0 gelesen, 1 als 1.023, und 0,5 als 512. Wird eine höhere Spannung als die Referenzspannung an einen analogen Eingang angelegt, besteht die Möglichkeit, den Arduino-Prozessor zu beschädigen. Es ist also große Vorsicht geboten, wenn die Referenzspannung verändert wird, was bei bestimmten Sensoren zur Erhöhung der Genauigkeit notwendig ist. Die Referenzspannung kann nicht höher als die Versorgungsspannung liegen (in den meisten Fällen 5 Volt, auf einigen Arduino-Clones allerdings 3,3).

Weiterhin ist es sehr sinnvoll, Referenzspannungen an den AREF-Pin über einen 5.000-Ohm-Widerstand anzuschließen, um den Arduino-Prozessor zu schützen. Dadurch wird ein zu großer Stromfluss verhindert, falls die aktuellen Referenzspannungseinstellungen nicht kompatibel zum Aufbau der Schaltung sind.

Es gibt drei mögliche Einstellungen für die Referenzspannung, die mit dem `type`-Argument gesetzt werden. Die normale Einstellung (wenn als Argument `DEFAULT` übergeben wird), die für die meisten Fälle ausreichen sollte, ist das Verwenden von AVCC (also der analogen Betriebsspannung, die 5 Volt beträgt) als Referenzspannung. Der AREF-Pin muss frei sein und wird intern mit AVCC beschrieben.

Die zweite Möglichkeit ist das Anschließen einer internen Spannung von 1,1 Volt als Referenzspannung. Dazu muss `type` als `INTERNAL` gesetzt werden.

Die dritte Möglichkeit ist das Konfigurieren der Referenzspannung über den AREF-Pin, indem `type` als `EXTERNAL` gesetzt wird. Dazu wird die Spannung an AREF angelegt (am besten über einen 5.000-Ohm-Widerstand, um Fehler zu vermeiden). Diese Spannung wird dann als Referenzspannung benutzt.

Fortgeschrittene Ein- und Ausgabe

Bei komplizierten Aufbauten müssen oft viele Werte über einen synchronen seriellen Bus ausgegeben werden. Das klingt ziemlich kompliziert, ist aber eigentlich sehr einfach. Z.B. sind die Shift-Register, die wir in Kapitel 10 (Musik-Controller) eingesetzt haben, so angeschlossen. Über solche Busse werden längere digitale Werte übertragen (z.B. 16 Bits), aber nur zwei oder drei digitale Ausgangspins benutzt.

Anders als bei einer asynchronen seriellen Schnittstelle, wie wir sie in Kapitel 5 gesehen haben, wird bei einer synchronen Schnittstelle ein expliziter Takt mitgegeben: Nach jedem übermittelten Bit (das auf eine Datenleitung gesetzt wird) wird ein sogenanntes Clock-Signal gepulst, um zu signalisieren, das gerade ein Bit anliegt. Das Pulsen des Clock-Signals muss eine gewisse Länge haben, damit der empfangende Chip genügend Zeit hat, den Wert einzulesen, kann aber sonst in einem beliebigen Tempo gesetzt werden. Die Clock-Leitung und die Datenleitung sind die einzigen notwendigen Leitungen, um Daten zu übermitteln. Oft wird aber noch eine dritte Leitung verwendet, um die Kommunikation zu starten (bzw. zu beenden). Bei einem Shift-Register wird zum Beispiel die Enable-Leitung verwendet, um die übermittelten Daten dann letztendlich auch zu aktivieren. Der komplette Kommunikationsablauf sieht also wie folgt aus: Kommunikation starten (wenn notwendig), erstes Datenbit anlegen, Clock-Signal pulsen, zweites Bit anlegen, Clock-Signal pulsen usw. Zum Abschluss muss ggf. noch die Kommunikation beendet werden.

shiftOut(dataPin, clockPin, bitOrder, value)

Diese Funktion vereinfacht das Übermitteln von Daten über eine synchrone serielle Schnittstelle. Übergeben werden `dataPin`, die Nummer des digitalen Ausgangspins, der für Datenübermittlung

verwendet wird, clockPin, die Nummer des digitalen Ausgangspins, der für die Clock-Leitung verwendet wird, bitOrder, die Reihenfolge, in der die Bits übermittelt werden sollen, und value, der numerische byte-Wert (also 8 Bits lang), der übermittelt werden soll.

Die Bitreihenfolge (bitOrder) kann zwei unterschiedliche Werte annehmen: LSBFIRST und MSBFIRST. LSBFIRST ist die Abkürzung für »least significant bit first«, und heißt, dass das niedrigste Bit aus value zuerst übertragen wird. So wird zum Beispiel Byte 19, also 00010011 in Binärdarstellung, wie folgt übertragen: 1, 1, 0, 0, 1, 0, 0, 0. MSBFIRST ist die Abkürzung für »most significant bit first«, und Byte 19 würde als 0, 0, 0, 1, 0, 0, 1, 1 übertragen. Je nach empfangenden Chip ist die eine oder die andere Darstellung erforderlich.

Beispiel

```
int clkPin = 9;
int dataPin = 10;
pinMode(clkPin, OUTPUT);
pinMode(dataPin, OUTPUT);
shiftOut(dataPin, clkPin, MSBFIRST, 19); // übertragen von 19 auf
                                         // der synchronen
                                         // seriellen
Schnittstelle
                                         // auf Pins 9 und 10
```

unsigned long pulseIn(pin, value) / unsigned long pulseIn(pin, value, timeout)

Manchmal werden Sensorwerte (z.B. beim Beschleunigungssensor ADXL, siehe) als Pulse übertragen. Je länger der Puls, desto höher ist der übertragene Wert. Es bedarf also einer Funktion, mit der sich diese Pulslänge messen lässt. Diese Funktion lässt sich z.B. mit den unten beschriebenen Zeitfunktionen definieren, ist dann allerdings nicht sehr genau (weil die Zeitfunktionen der Arduino-Umgebung nur als grobe Hilfen zur Verfügung stehen). Deswegen wird für solche Zwecke die Funktion pulseIn verwendet.

pulseIn gibt die Länge eines Pulses auf dem digitalen Eingangspin pin zurück. Dabei gibt value an, welchen Wert der Puls haben will. Wird als value HIGH übergeben, wartet die Funktion, bis der Eingangspegel am Pin auf HIGH geht, und misst die Zeit, bis der Eingangspegel wieder auf LOW geht. Die gemessene Zeit wird in Mikrosekunden zurückgegeben. Wird LOW als value übergeben, wartet die Funktion darauf, dass der Eingangspegel auf LOW geht, und misst die Zeit, bis er wieder zu HIGH zurückkehrt. Wird nach einer bestimmten Timeout-Zeit kein Pegel erkannt (als Standardwert wird eine Sekunde benutzt, es ist allerdings möglich, den

Timeout als drittes Argument zu übergeben), gibt die Funktion 0 zurück. Die Zeitmessungen dieser Funktionen können von 10 Mikrosekunden bis ungefähr 3 Minuten gehen, werden allerdings ungenauer, je länger der Puls ist.

Beispiel

```
pinMode(9, INPUT); // initialisiert Pin 9 als Eingangspin
int pulsLaenge = pulseIn(9, HIGH); // misst einen positiven Puls
                                   // auf Pin 9
pulsLaenge = pulseIn(9, HIGH, 400);// misst einen positiven Puls
                                   // auf Pin 9, gibt 0 zurück, falls
                                   // innerhalb von 400 Mikrosekunden
                                   // kein Puls erkannt wurde
```

Zeitfunktionen

Es ist oft notwendig, Zeiten zu messen und auch bestimmte Zeiten zu warten, um Programme zu implementieren. Diese Zeiten lassen sich in zwei größere Ebenen einteilen: »menschliche Zeit«, also Zeitintervalle, die für Menschen bemerkbar sind. Diese reichen von ein paar Millisekunden (wenn z.B. LEDs blinken sollen) bis zu mehreren Sekunden oder gar Minuten (wie z.B. beim Brainmachine-Workshop in). Auf der anderen Seite gibt es die »Computerzeit«, die deutlich schneller ist. Sie wird z.B. benötigt, um auf bestimmte Sensoren oder Chips zu warten, um ein Kommunikationsprotokoll genau zu implementieren, um Lichter zu dimmen (sie also so schnell flackern zu lassen, dass der Mensch es nicht bemerkt).

Deswegen sind die zwei Zeitfunktionen (eine Funktion zum Messen von Zeit und eine Funktion zum Warten) immer in zwei Versionen vorhanden: eine für Millisekunden und eine für Mikrosekunden (es gibt 1.000 Millisekunden in einer Sekunde und 1.000 Mikrosekunden pro Millisekunde). Der Arduino-Prozessor läuft mit 16 MHz, d.h. er kann bis zu 16 Millionen Instruktionen pro Sekunde ausführen. Das setzt auch die unterste Grenze für die Auflösung gemessener Zeitwerte fest. Jeder Zyklus ist genau 0,0625 Mikrosekunden lang, es ist also nicht möglich, mit dem Arduino sehr genaue Werte in Mikrosekunden überhaupt zu messen. Weiterhin entstehen durch den Aufruf der Funktion und unterschiedliche andere Nebenwirkungen (wie z.B. periodische Interrupts, die in Arduino verwendet werden) Verzögerungen, die sich nicht voraussagen lassen. Dadurch ist die Genauigkeit dieser Zeitfunktionen mit Vorsicht zu genießen.

unsigned long millis()

Diese Funktion gibt die Anzahl von Millisekunden zurück, die seit dem Anschalten des Arduino abgelaufen sind. Der Zähler wird bei jedem Neuprogrammieren, Neuanschließen und Resetten des Arduino-Boards neu initialisiert. Nach ungefähr 50 Tagen läuft die Variable `unsigned long` über und beginnt wieder bei 0 (das ist ein ähnliches Problem wie beim Y2K-Bug). Das folgende Programm gibt die abgelaufene Zeit in Millisekunden auf der seriellen Schnittstelle aus.

Beispiel

```
unsigned long time;
void setup() {
    Serial.begin(9600);
}
void loop() {
    time = millis(); // Auslesen der abgelaufenen Millisekunden
                     // seit Programmanfang
    Serial.println(time); // Ausgeben auf der seriellen Konsole
    elay(1000); // eine Sekunde warten
}
```

unsigned long micros()

Diese Funktion ist das Mikrosekunden-Pendant zur Funktion `millis()`. Sie gibt die abgelaufene Anzahl an Mikrosekunden seit Programmanfang zurück und läuft schon nach ungefähr 70 Minuten über. Durch den Aufrufmehraufwand gibt diese Funktion auf einem Arduino-Board immer ein Vielfaches von 4 Mikrosekunden zurück (wenn das Arduino-Board mit 16 MHz läuft; 8-MHz-Arduino-Boards wie z.B. das LilyPad liefern Vielfache von 8 Mikrosekunden zurück).

delay(ms)

Diese Funktion hält das Ausführen des Programms für die angegebene Zeit `ms` (Millisekunden) an. In Wirklichkeit läuft der Prozessor weiter (es werden z.B. auch Interrupt-Funktionen weiter aufgerufen) und führt nur Anweisungen aus, die nichts bewirken. Die serielle Schnittstelle läuft deswegen problemlos weiter und empfangene Bytes werden weiter im Puffer gespeichert. Auch mit PWM gepulste Pins (siehe) laufen während eines `delay`-Aufrufs weiter. Allerdings ist es in dieser Zeit auch nicht möglich, z.B. weitere Werte aus einem Sensor oder von Tastern auszulesen. Deswegen ist es bei längeren Pausen (von mehr als nur ein paar

Millisekunden) besser, auf die Funktionen `micros()` und `millis()` auszuweichen, um bestimmte zeitabhängige Verhalten zu implementieren.

Da Interrupts weiterlaufen, während `delay()` ausgeführt wird, ist die zeitliche Genauigkeit der Funktion nicht sehr hoch. Sie sollte nicht eingesetzt werden, um zeitkritische Verhalten zu implementieren. Auch in diesem Fall ist es besser, die Funktion `millis()` zu benutzen oder die Interrupts zu deaktivieren.

In dem folgenden einfachen Beispiel wird eine LED zum Blinken gebracht.

Beispiel

```
int ledPin = 13;
void setup() {
    pinMode(ledPin, OUTPUT); // Der LED-Pin wird als Ausgang
                             // initialisiert.
}
void loop() {
    digitalWrite(ledPin, HIGH); // LED anschalten
    delay(1000); // 1000 Millisekunden warten (also eine
                 // Sekunde)
    digitalWrite(ledPin, LOW); // LED ausschalten
    delay(1000); // eine weitere Sekunde warten
}
```

Das folgende Beispiel zeigt, wie man das Blinken der LED ohne `delay()` implementieren kann. Dazu wird in jedem `loop()`-Aufruf die Zeit gemessen, die verlaufen ist, und entsprechend die LED an- und ausgeschaltet. Das Beispiel scheint relativ uninteressant zu sein, aber im Vergleich zum vorigen Programmbeispiel lassen sich jetzt z. B. problemlos Taster abfragen. Würde man `delay()` mit Tastenabfragen kombinieren, würden viele verloren gehen, weil der Prozessor in dieser Zeit damit beschäftigt wäre, `delay()` auszuführen.

Beispiel

```
int ledPin = 13;
int ledValue = LOW; // der aktuelle Wert der LED
long ledSchaltZeit = 0; // die Zeit, zu der die LED zuletzt
                        // geschaltet wurde
long intervall = 1000; // die Länge des Intervalls zwischen jedem
                       // Schalten
void setup() {
   pinMode(ledPin, OUTPUT); // Der LED Pin-wird als Ausgang
                            // initialisiert.
}
void loop() {
   // zeit speichert die Dauer seit dem letzten Schalten.
```

```
    unsigned long zeit = millis() - ledSchaltZeit;
    // überprüfen, ob genügend Zeit seit dem letzten Schalten
    // abgelaufen ist
    if (zeit > intervall) {
       // es ist genügend Zeit abgelaufen, jetzt muss die LED
       // geschaltet werden
       if (ledValue == LOW) {
          ledValue = HIGH;
       } else {
          ledValue = LOW;
       }
       digitalWrite(ledPin, ledValue);
       // jetzt wird die aktuelle Zeit als Schaltzeit gespeichert
       ledSchaltZeit = millis();
    }
    // Hier können jetzt Sensoren und Taster abgefragt und andere
    // Aufgaben ausgeführt werden.
}
```

delayMicroseconds(us)

`delayMicroseconds()` ist das Mikrosekunden-Pendant zur `delay()`-Funktion. Auch hier gibt es wichtige Unterschiede. Im Unterschied zur `delay()`-Funktion ist `delayMicroseconds()` zwischen ungefähr 3 Mikrosekunden und 16.383 Mikrosekunden sehr genau. Um diese Genauigkeit zu erreichen, werden allerdings die Interrupts deaktiviert, d.h. es werden keine Daten auf der seriellen Schnittstelle empfangen, und Zeiten, die von `millis()` und `micros()` zurückgegeben werden, werden nicht hochgezählt. `delayMicroseconds()` sollte man deswegen nur für sehr kurze Pausen benutzen, `delay()` dagegen für längere. Wegen der Art, wie `delayMicroseconds()` geschrieben wurde (um diese Genauigkeit einzuhalten), führt `delayMicroseconds(0)` zu einer deutlich längeren Pause von ungefähr 1.020 Mikrosekunden.

Mathematische und trigonometrische Funktionen

Viele häufig verwendete mathematische Funktionen werden auch von der Arduino-Programmiersprache zur Verfügung gestellt. Sie decken Bereiche ab von der einfachen Berechnung (Minimum, Maximum) bis hin zu komplexeren trigonometrischen Funktionen wie Sinus und Kosinus.

min(x, y)

Diese Funktion gibt die kleinere der beiden Zahlen `x` und `y` zurück. Sie kann auf alle numerischen Datentypen angewandt werden. Aus

nicht ganz intuitiven Gründen wird min oft benutzt, um die obere Grenze einer Variable zu setzen. Aus Implementationsgründen sollte man vermeiden, weitere Ausdrücke in die Parameter von min zu schreiben, da die Gefahr besteht, sie mehrmals auszuführen. Deswegen sollte man unbedingt darauf achten, reine Zahlen und Variablen als Argumente an min (und an max) zu übergeben.

Beispiel

```
int x = 13;
x = min(x, 12); // x wird nie größer als 12 sein.
min(x++, 20); // Diese Schreibweise ist falsch, x++ könnte
              // mehrmals ausgeführt werden.
x++;
min(x, 20); // Dies ist die richtige Schreibweise.
```

max(x, y)

Diese Funktion gibt die größere der beiden Zahlen x und y zurück. Wie min kann auch max auf alle numerischen Datentypen angewandt werden. Auch nicht ganz intuitiv ist die Anwendung von max zum Setzen der unteren Grenze einer Variablen. Zum Einschränken einer Variablen nach oben und unten sollte man allerdings auf die schwer lesbare Kombination von min und max verzichten und stattdessen die Funktion constrain einsetzen. Auch hier sollten nur Variablen und Zahlen als Argumente übergeben werden.

Beispiel

```
int x = 13;
x = max(x, 5); // x wird nie kleiner als 5 sein.
// Folgendes ist schwer zu lesen.
int y = 80;
y = max(min(y, 100), 20); // y wird immer zwischen 20 und 100
sein.
```

abs(x)

Diese Funktion gibt den Absolutwert einer Zahl zurück (also x, wenn x positiv ist, und –x, wenn x negativ ist) und kann auf alle numerischen Datentypen angewandt werden. Auch an abs dürfen nur Variablen und numerische Zahlenwerte übergeben werden, keine beliebigen Ausdrücke.

Beispiel

```
int x = 13;
int y = abs(x); // y ist 13.
x = -13;
y = abs(x); // y ist 13.
```

constrain(x, a, b)

Anstatt eine Kombination aus `min` und `max` zu verwenden, um einen Wert nach oben und unten zu beschränken, lässt sich die Funktion `constrain` einsetzen. Sie gibt `x` zurück, wenn `x` größer oder gleich `a` und kleiner oder gleich `b` ist. Ist `x` kleiner als `a`, wird `a` zurückgegeben. Ist `x` größer als `b`, wird `b` zurückgegeben.

Beispiel

```
int x = 13;
x = constrain(x, 10, 20); // x ist immer zwischen 10 und 20.
```

map(value, fromLow, fromHigh, toLow, toHigh)

Soll ein Wert in einem bestimmten numerischen Bereich in einen anderen numerischen Bereich konvertiert werden (also nicht nur beschränkt werden), ist die `map`-Funktion zu benutzen. Die Zahl `value`, die sich im Bereich `fromLow` bis `fromHigh` befindet, wird in den Bereich `toLow` bis `toHigh` abgebildet. So kann z.B. ein Wert in dem Bereich 0 bis 100 konvertiert werden, damit er im Bereich von 50 bis 60 liegt. Dann wird z.B. 0 zu 50, 50 zu 55, 100 zu 60. So lässt sich auch der Wertebereich invertieren, indem man zum Beispiel eine Zahl von 0 bis 100 auf den Bereich 100 bis 0 konvertiert. Weiterhin kann die `map`-Funktion auch Bereiche mit negativen Zahlen bearbeiten, sodass man z.B. den Bereich 20 bis 40 auf den Bereich –20 bis 80 legen kann (20 wird zu –20, 30 wird zu 30, 40 wird zu 80).

Die `map`-Funktion beschränkt Eingangswerte nicht auf den angegebenen Bereich, weil es oft auch nützlich ist, Ausreißer zu berücksichtigen. Die Funktion arbeitet mit ganzzahligen Werten, auch wenn je nach Eingabe mathematisch eine Fließkommazahl erzeugt werden müsste.

Beispiel

```
int val = analogRead(0); // analoger Sensorwert, von 0 bis 1023
int y = map(val, 0, 1023, 0, 255); // jetzt von 0 bis 255
analogWrite(9, y); // und als PWM-Wert für Pin 9 eingesetzt
// für mathematisch interessierte Leser ist hier die Definition
// der Funktion:
long map(long x, long in_min, long in_max, long out_min, long out_max) {
   return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}
```

sq(x)

Diese Funktion gibt das Quadrat einer Zahl zurück.

pow(base, exponent)

Diese Funktion gibt die Potenz der Zahl `base` zurück. Der Grad der Potenz wird im Argument `exponent` übergeben. Die Argumente der `pow`-Funktion sind Fließkommazahlen, sodass auch Bruchwerte als Potenzen eingesetzt werden können (z. B. um Wurzeln oder exponentielle Werte zu berechnen). Allerdings ist deswegen die Funktion relativ langsam und sollte sparsam (oder mithilfe von Lookup-Tabellen, siehe Kapitel 4, »LED-Matrix«) eingesetzt werden. Alle weiteren Funktionen (insbesondere die trigonometrischen) in diesem Abschnitt arbeiten mit Fließkommazahlen und komplizierten Algorithmen und sind deswegen sehr langsam.

sqrt(x)

Diese Funktion gibt die quadratische Wurzel einer Fließkommazahl zurück.

sin(rad)

Diese Funktion gibt den Sinus eines Winkels `rad`, der in Radian angegeben ist, zurück.

cos(rad)

Diese Funktion gibt den Kosinus eines Winkels `rad`, der in Radian angegeben ist, zurück.

tan(x)

Diese Funktion gibt den Tangens eines Winkels `rad`, der in Radian angegeben ist, zurück.

Zufallszahlen

Es ist möglich, auf dem Arduino-Prozessor Zufallszahlen zu erzeugen. Allerdings sind diese nicht »echte« Zufallszahlen, sondern werden algorithmisch generiert. Ein sogenannter PRNG (*pseudo-random number generator*, pseudozufälliger Zahlengenerator) rechnet immer wieder neue Zahlen aus, die in keiner ersichtlichen logischen Reihenfolge zueinander stehen. Wird dieser PRNG mit demselben Wert initialisiert (das Initialisieren nennt man auf englisch *seeding*, also »Samen pflanzen«, den Initialisierungswert *seed*, also »Samen«), generiert er auch dieselbe Folge an Zufallszahlen. Das kann ein Vor- oder ein Nachteil sein. Auf der einen Seite können immer wieder gleiche Zufallszahlen nach einer gewissen Zeit

ihren Reiz verlieren, auf der anderen Seite ist es so möglich, einen bestimmten Ablauf zu wiederholen, falls z.B. ein Fehler oder eine besonders interessante Folge gefunden wurde. Um jedes Mal eine neue Folge zu generieren, kann man den PRNG mit einer »richtigen« Zufallszahl initialisieren (z.B. dem Ergebnis von `analogRead` auf einem nicht angeschlossenen Pin). Echte Zufallszahlen sind aber nicht gleich verteilt, während die erzeugten Zufallszahlen einigermaßen gleich verteilt und deshalb besser einzusetzen sind.

randomSeed(seed)

Mit dieser Funktion wird der PRNG initialisiert.

long random(max) / long random(min, max)

Diese Funktion liefert einen zufälligen Wert als Ergebnis. Dieser Wert befindet sich zwischen 0 (bzw. `min`, wenn `min` als Argument angegeben wurde) und der oberen Schranke `max`.

Beispiel

```
randomSeed(analogRead(0)); // initialisiert den PRNG auf einen
                           // wirklich zufälligen Wert, weil 0
                           // nicht angeschlossen ist
int val = random(300); // liefert einen zufälligen Wert zwischen 0
                       // und 300
int val2 = random(10, 20); // liefert einen zufälligen Wert
                           // zwischen 10 und 20
```

Serielle Kommunikation

Mit den folgenden Funktionen lässt sich die serielle Schnittstelle des Arduino-Prozessors konfigurieren und benutzen (siehe). Diese serielle Schnittstelle ist über einen weiteren Chip an die USB-Schnittstelle angebunden und kann so einfach an jedem Computer angebracht werden. Die Funktionen in diesem Abschnitt können verwendet werden, um sowohl Daten vom Arduino-Prozessor an den Computer zu schicken als auch Daten vom Computer aus an den Arduino-Prozessor zu übermitteln.

Serial.begin(speed)

Bevor die serielle Schnittstelle verwendet werden kann, um Daten zwischen Computer und Arduino-Prozessor auszutauschen, muss erst die Geschwindigkeit der Übertragung eingestellt werden. Diese Geschwindigkeitseinstellung (und weitere Initialisierungen der seriellen Schnittstelle und der dazugehörigen Speicherdaten) werden

durch die Funktion `Serial.begin()` implementiert. Als Argument wird ihr die gewünschte Geschwindigkeit der Schnittstelle in Bits pro Sekunde (oder auch Baud) mitgegeben. Aus technischen und historischen Gründen sind allerdings nicht alle Geschwindigkeiten möglich, sondern nur folgende (die vielleicht noch aus Modemzeiten bekannt sind): 300, 1.200, 2.400, 4.800, 9.600, 14.400, 19.200, 28.800, 38.400, 57.600 oder 115.200 Bits pro Sekunde. Es ist wichtig, dass auf der empfangenden Seite (auf dem Computer) und im Sketch die gleichen Geschwindigkeiten eingestellt sind.

Es ist nicht möglich, die serielle Schnittstelle vor der Initialisierung zu benutzen. Deswegen ist in den meisten Sketches, die die serielle Schnittstelle verwenden, ein Aufruf an die Funktion `Serial.begin()` als erste Anweisung in der Funktion `setup` zu sehen. Ein weiterer wichtiger Punkt ist, dass nach dem Initialisieren der seriellen Schnittstelle die Benutzung der ersten zwei digitalen Pins nicht mehr möglich ist, weil diese Pins die eigentliche serielle Schnittstelle bilden (ein Pin für den Datenempfang und einer für den Datenversand).

int Serial.available()

Der Arduino-Prozessor empfängt Daten, die vom Computer aus über die serielle Schnittstelle geschickt wurden, automatisch in einer getrennten Interrupt-Routine. Diese Routine speichert die empfangenen Daten in einem kleinen Puffer, damit der Sketch, der auf dem Arduino-Prozessor läuft, nicht dauernd unterbrochen wird. Die Funktion `Serial.available()` prüft, ob Daten in diesem Zwischenpuffer empfangen wurden, und gibt zurück, wie viele Bytes an Daten vorhanden sind. Es ist wichtig, regelmäßig Daten aus diesem Puffer mithilfe der Funktion `Serial.read()` auszulesen oder ihn mit `Serial.flush()` zu leeren, damit er nicht überläuft (er kann maximal 128 Bytes zwischenspeichern).

int Serial.read()

Diese Funktion liest das nächste Byte aus dem Empfangspuffer und gibt es zurück. Dadurch wird wieder ein Byte in diesem Puffer frei. Falls keine Daten empfangen wurden, gibt die Funktion –1 zurück.

Beispiel

```
void setup() {
   Serial.begin(9600); // Initialisieren der seriellen
                       // Schnittstelle auf 9600 Baud
}
```

```
void loop() {
   // überprüfen, ob Daten über die serielle Schnittstelle
   // empfangen wurden
   if (Serial.available() > 0) {
      // es wurden Daten empfangen
      byte b = Serial.read(); // Auslesen des ersten
                              // Bytes und Ausgabe auf der
seriellen Schnittstelle
      Serial.print("Empfangen: ");
      Serial.println(b, DEC);
   }
}
```

Serial.flush()

Mit dieser Funktion lässt sich der komplette Empfangspuffer leeren (z.B. beim Neuinitialisieren des Programms). Diese Funktion ist auch nützlich, wenn z.B. ein Überlauf des Empfangspuffers erkannt wurde und die enthaltenen Daten nicht mehr gebraucht werden.

Serial.print(data) / Serial.println(data)

Mit diesen zwei Funktionen können beliebige ganzzahlige Werte und Zeichenketten auf der seriellen Schnittstelle ausgegeben werden. Die Funktion `Serial.println()` fügt eine Leerzeile an das Ende der Ausgabe ein.

Wird `Serial.print` ohne zweites Argument verwendet, wird das erste Argument in Dezimaldarstellung ausgegeben. Es ist allerdings möglich, die Darstellung als zweites Argument an `Serial.print()` zu übergeben, um so z.B. eine Zahl in Hexadezimaldarstellung auszugeben. Es ist so möglich, mit `DEC` eine Zahl in Dezimaldarstellung auszugeben, mit `HEX` in Hexadezimaldarstellung, mit `OCT` in Oktaldarstellung, mit `BIN` in Binärdarstellung und mit `BYTE` als ASCII-Zeichen.

Es ist nicht möglich, Fließkommazahlen auf der seriellen Schnittstelle auszugeben; sie werden in ganzzahlige Zahlen konvertiert. Es ist allerdings möglich, diese Fließkommazahlen vor dem Ausgeben noch mit 10 oder 100 zu multiplizieren, um so zumindest ein paar Nachkommaziffern anzeigen zu können.

Beispiel

```
void setup() {
   Serial.begin(9600); // Initialisieren der seriellen
                       // Schnittstelle auf 9600 Baud
}
void loop() {
   // gibt die Zahlen von 0 bis 9 auf der seriellen Schnittstelle aus
```

```
  for (int x = 0; x < 10; x++) {
    Serial.print("Normal: ");
    Serial.println(x);
    Serial.print("DEC: ");
    Serial.println(x, DEC);
    Serial.print("OCT: ");
    Serial.println(x, OCT);
    Serial.print("HEX: ");
    Serial.println(x, HEX);
    Serial.print("BIN: ");
    Serial.println(x, BIN);
  }
}
```

ANHANG D

Händlerliste

In der folgenden Tabelle finden Sie einige Lieferanten, die Arduino-boards führen.

Händler	Internet	
bausteln	www.bausteln.de	
Geist Electronic-Versand GmbH Hans-Sachs-Strasse 19 78054 VS-Schwenningen Telefon 0049 (0)7720 / 36673 Fax 0049 (0)7720 / 36905	www.geist-electronic.de	
SEGOR-electronics GmbH Kaiserin-Augusta-Allee 94 10589 Berlin	www.segor.de	
TinkerSoup	www.tinkersoup.de	
Watterott electronic Winkelstr. 12a 37327 Hausen	www.watterrott.com	

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Über die Autoren

Manuel Odendahl ist Selbstständiger und stellt unter dem Namen »Ruin & Wesen« Opensource-MIDI-Kontroller her.

Julian Finn ist Informatikstudent kurz vor dem Diplom und arbeitet bei der Gameforge AG in Karlsruhe, sowie als freier Autor. Er hat die Texte für mehrere Computerspiele verfasst und beschäftigt sich ansonsten mit den Randbereichen von Technik, Kunst und Gesellschaft. Dazu gehört die Arbeit an Medienkunstinstallationen genauso wie Text- und Diskussionsbeiträge zu digitaler Kultur, freiem Wissen und Netzpolitik.

Alex Wenger wollte schon als Kind wissen wie die Welt im Inneren funktioniert und kein elektrisches Gerät war vor Ihm sicher. Nach dem Physik- und Dipl. Ing. Informationstechnikstudium entwickelt er hauptberuflich Software und Elektronik für Display-Anwendungen. Zu den weiteren Beschäftigungen gehört die Arbeit als Medienkünstler und Dozent.

Yunjun Lee ist in Südkorea geboren und arbeitet momentan als Gastkünstler am Karlsruher Zentrum für Kunst- und Medientechnologie (ZKM). Yunjun Lee entwickelt vielseitige Medienkunstprojekte, bei denen er seine Fähigkeiten als Maler, Bildhauer und Musiker mit technischen Mitteln erweitert.

Jonas Zilius studiert Architektur an der Universität Karlsruhe. Neben seiner Tätigkeit als freier Grafiker für Print- und Web-Medien ist seine Leidenschaft die Photographie. Er ist studentischer Mitarbeiter in der zentralen Fotowerkstatt der Fakultät Architektur. Arbeiten aus den Bereichen Veranstaltungen/Konzerte, Sport, Portrait und Produkte zählen zu seinen Referenzen. Ein Reise-Bildband über Tirana/Albanien ist in Arbeit.

Kolophon

Das Coverlayout dieses Buchs hat Michael Oreal gestaltet. Als Textschrift verwenden wir die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed und die Nichtproportionalschrift für Codes ist LucasFont's TheSansMono Condensed.