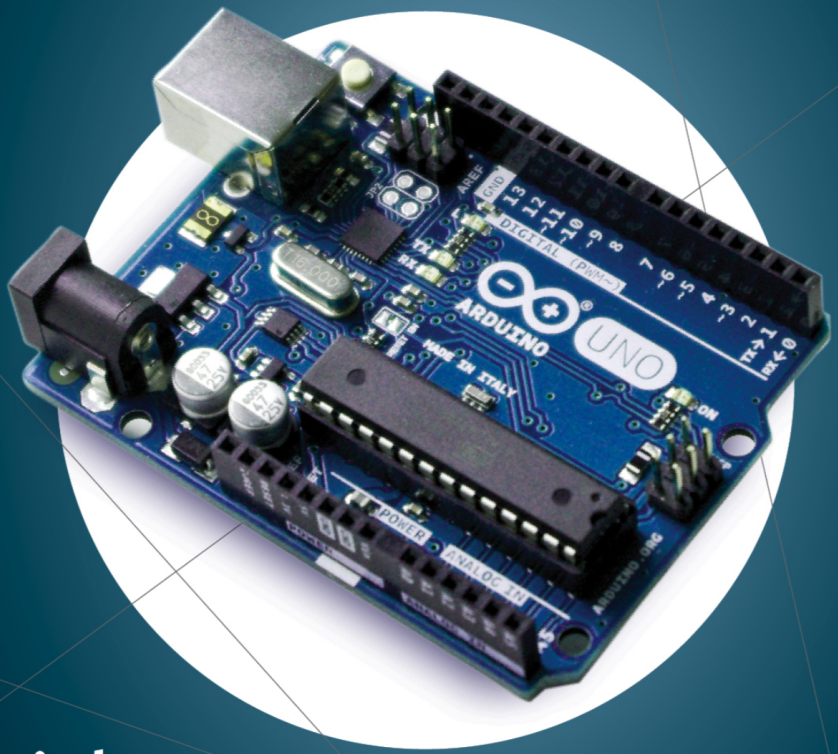
Erik Bartmann

Der **Arduino-**Bestseller in **4. Auflage**

# Mit Arduino die elektronische Welt entdecken




**48** Bastelprojekte mit dem Arduino

bombini verlag

Erik Bartmann

# Mit Arduino die elektronische Welt entdecken

«Bookwire»

**Bartmann E.**

Mit Arduino die elektronische Welt entdecken / E. Bartmann — «Bookwire»,

Der Arduino-Mikrocontroller ist aus der Elektronikwelt nicht mehr wegzudenken, er hat sich zu einem Standard im Hobbybereich entwickelt. In unzähligen Projekten kommt das Arduino-Board zum Einsatz, Hunderttausende von ausgereiften Softwarelösungen stehen für jeden zugänglich und unter freier Lizenz zur Verfügung. Der Arduino ist leicht zu programmieren. Preiswerte elektronische Bauteile wie LCDs, Sensoren und Motoren können an das Arduino-Board angeschlossen und damit gesteuert werden.Mit «Arduino die elektronische Welt entdecken» führt den Leser in die faszinierende Welt der Elektronik und Programmierung ein. Die Hardware wird leicht verständlich dargestellt und die Programmierung des Mikrocontrollers Schritt für Schritt grundsätzlich erklärt. Herzstück des Buches sind 48 detailliert beschriebene Arduino-Bastelprojekte, wobei sich die Komplexität von Projekt zu Projekt steigert. In jedem Bastelprojekt wird ein neues Grundlagenthema behandelt, neue Hardware wird eingeführt und neue Programmierkniffeund –werkzeuge werden vorgestellt.Jedes Bastelprojekt ist mit zahlreichen Fotos und Abbildungen illustriert und kann Schritt für Schritt nachgebaut werden. Alle verwendeten Bauteile werden genau erklärt und in ihrer prinzipiellen Funktionsweise vorgestellt. Die Bastelprojekte können beliebig erweitert und für andere Zwecke angepasst werden.Generationen von Hobbybastlern haben mit Erik Bartmanns Bestsellerbuch bereits die Arduino-Programmierung gelernt. In der komplett überarbeiteten 4. Neuauflage des Arduino-Standardwerkes wurden neue Bauteile wie der ESP32 oder LoRaWAN aufgenommen und neue Entwicklerwerkzeuge wie Node-RED, KiCad und MQTT behandelt.

# Содержание

Inhalt

# Mit Arduino die elektronische Welt entdecken

**Erik Bartmann**

# Impressum



Kommentare und Fragen können Sie gerne an uns richten:
Bombini Verlags GmbH
Kaiserstraße 235
53113 Bonn
E-Mail: service@bombini-verlag.de



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.

Umschlaggestaltung: Michael Oreal, Köln (www.oreal.de) **e**lectronic **pub**lication: III-satz, Husby, www.drei-satz.de

ISBN 978-3-946496-29-8

# Grußwort zur vierten Auflage von Teo Swee Ann, CEO von Espressif

Dear Erik,

In 2016, I came across a book of yours, which was about ESP8266, a microcontroller that my company Espressif had developed and released two years earlier. It was the first book ever written about ESP8266. Although I do not speak German and, therefore, I could not read it, I still understood the basic statements you made about ESP8266, as I was going through your text. Your beautiful and numerous illustrations helped me guess the accompanying text. Most importantly, I liked that you promoted the ESP8266 microcontroller as a tool for the Maker movement in as early as 2016. In any case, your assessment back then has been proved correct: Today, in every Maker workshop, no matter where in the world, one can find an ESP8266 or its successor, ESP32, alongside an Arduino and a Raspberry Pi board.

As I learned later, you had also written an extensive book about Arduino. Since its first edition in 2011, this 1,000-page book has been published in three editions already, and has become a reference book about Arduino in German. In this book, not only you do describe the functionality of the Arduino hardware and software, but you also cover the entire Maker catalogue of relevant topics. You explain the basics about electronics in detail and in an easy-to-understand way, you give an introduction to programming, and you show in numerous projects what great things can be developed with the Arduino platform.

In addition, you let the Maker think outside the box, discussing cutting-edge ideas that push the Arduino hobbyist's creativity further. You have included MQTT, Node-RED, and LoRa-related topics, among many others that can help every Maker develop truly awesome and even complicated projects on their own. I am pleased to see that you have also covered ESP32 with its own project in your book.

I wish you much success for the fourth edition of your Arduino book.

**Teo Sween Ann**


ESPRESSIF
DELL

# Einleitung
# Arduino für alle

Als ich mein erstes Buch über Arduino schrieb, gab es zwei oder drei Händler in Deutschland, über die man das Arduino-Board beziehen konnte, und ungefähr 800 Videos wurden bei YouTube aufgelistet, wenn man das Suchwort Arduino eingab. Für Elektronikprofis war dieser Mikrocontroller kaum ein Thema und wenn, dann belächelten sie eher seine Leistungsfähigkeit.

Die Arduino-Entwickler, die sich 2005 in der italienischen Kunst-Uni Ivrea ans Werk machten, einen leicht zu programmierenden Mikrocontroller zu entwerfen, hatten nicht den Elektronikprofi im Visier, sondern den Kunststudenten mit wenig Programmier- und Hardwarekenntnissen. Ihr Fokus lag dabei auf dem schnellen Realisieren von Ideen, dem Prototyping: eine künstlerische Idee, die der Student ohne Unterstützung von Ingenieuren und Programmierern selbst realisieren konnte.

# Der Layoutfehler, der Geschichte machte

Die Arduino-Macher, allesamt keine Hardwarespezialisten, stellten den ersten Arduino-Mikrocontroller, den Arduino Uno, unter eine freie Hardwarelizenz, wobei sie die Erfahrungen der Open-Source-Bewegung auf die Hardwarewelt übertrugen. Sie legten die Schaltpläne offen, so dass jedermann frei darin war, das Board nach- oder umzubauen. Und die Arduino-Programmierumgebung – oder auch IDE genannt (Integrated Development Environment) – stellten sie ebenfalls unter eine Open-Source-Lizenz, jeder konnte die Software frei nutzen.

Bald schon wurde das Arduino-Board nicht nur von Studenten der italienischen Uni genutzt, sondern entwickelte sich zu einer praktischen Prototyping-Plattform, erst für Elektronik-Nerds, dann für Hobbyisten auf der ganzen Welt. Wesentlich beigetragen zur erfolgreichen Verbreitung dieser Plattform hat das Internet, das es ermöglichte, Schaltungen und dazugehörige Softwareprogramme leicht zu verbreiten – und Dank der freien Lizenz – auch nutzbar für alle zu machen. Ein riesiger weltweiter Pool von Ideen entstand, entwickelt von Hobbyisten, die stolz einer weltweiten Community ihre Projekte vorstellten. Projektideen wurden von irgendjemandem auf der Welt übernommen, vielleicht sogar verbessert und wieder ins Netz gestellt.

In den Folgejahren entwickelten sich der Arduino und seine Entwicklungsumgebung zu einem Quasistandard in der weltweiten Bastlerwelt. Eine neue, leicht zugängliche Plattform für E-Bastler war entstanden. Das blieb auch nicht den großen Hardwareproduzenten verborgen, die registriert hatten, dass zunehmend der Arduino für alle möglichen Steuerungs- und Messaufgaben eingesetzt wurde. Um die Möglichkeiten des Arduino-Boards zu erweitern, wurden Ergänzungsplatinen, die genau auf den Arduino Uno passten, sogenannte Shields, entwickelt.

Spätestens zu dem Zeitpunkt, als einer der größten Produzenten von Hardware seine Entwicklungsabteilung anwies, zukünftig das Layout des Arduino-Boards als Basis für ihre Hardware anzusehen, war klar, dass der Arduino aus der Technikwelt nicht mehr wegzudenken war. Kleine Anekdote am Rande: Wie schon gesagt, die Arduino-Entwickler waren zu Beginn noch keine Hardwarespezialisten, so dass sie leider einen Layoutfehler in das Arduino-Uno-Board einbauten. Um kompatibel mit dem Quasistandard zu bleiben, übernahm der namhafte weltweite Hersteller brav diesen Layoutfehler. Bis heute.

# Von den Segnungen des Copy&Paste – und seinen Beschränkungen

Als ich mein erstes Arduino-Buch plante, fragte ich mich, was ein Bastler vor allem braucht, um mit dem Arduino eigene Projekte zu realisieren. Es war ja schon zu diesem Zeitpunkt möglich, sich über das Internet und über die gerade entstehenden Arduino-Communities und -foren lauffähige Programme zu besorgen, sie auf den eigenen Arduino hochzuladen und schon lief das Projekt auch in heimischer Umgebung, vielleicht noch versehen mit ein paar gezielten Eingriffen in den Code. Musste zusätzliche Hardware verbaut werden, orientierte man sich beim Nachbau an der Schaltung.

Was also sollte ein Buch über den Arduino noch enthalten außer der Beschreibung des Boards und der Entwicklungsumgebung und wie man die Programme vom PC auf den Mikrocontroller bekommt? Was passiert, wenn man eine Schaltung nachgebaut hat und sie dann erweitern möchte? Um einen guten konzeptionellen Ansatz für mein Buch zu finden, fragte ich mich, was mir am meisten dabei geholfen hat, eigene Elektronikprojekte zu bauen. Die Antwort war simpel und ernüchternd: Mir haben meine Grundkenntnisse in Elektrotechnik dabei geholfen, aus Ideen funktionierende Technikprojekte zu machen.

Ich finde es absolut beeindruckend, dass Menschen, die nichts mitbringen außer Bastellust, mit dem Arduino bereits nach einer halben Stunde eine Ampelschaltung auf dem Steckbrett betreiben können, ohne jemals zuvor einen Mikrocontroller in der Hand gehalten oder eine Zeile Code geschrieben zu haben. Es reicht dazu ein YouTube-Video und vielleicht noch eine Schritt-für-Schritt-Beschreibung, schon blinken die LEDs im gewünschten Ampelschaltungsrhythmus auf dem Schreibtisch zu Hause. Vielleicht hat man dann noch Spaß daran, den Rhythmus abzuändern, indem man im Code an der entsprechenden Stelle, die leicht zu finden ist, herumtüftelt – das war es dann aber auch schon. Um mehr daraus zu machen, braucht man ein paar Grundkenntnisse der Elektronik und der Elektrotechnik. Deshalb habe ich dieses Buch geschrieben.

# Wie ich mein Buch aufgebaut habe

Zunächst finde ich wichtig, dir möglichst viel über die Hardware und Software von Arduino mitzuteilen, damit du die gesamte Klaviatur kennenlernst, auf der hier gespielt wird. In Kapitel 1, »Arduino: Die Hardware«, stelle ich dir ziemlich detailliert das Arduino-Board mit allen Bauteilen, der Stromversorgung und den Schnittstellen vor. Daneben gibt's eine Auffrischung einiger physikalischer Grundbegriffe, die bei der Mikroelektronik eine Rolle spielen. Kannst du lesen, musst du aber nicht. In Kapitel 2, »Arduino: Die Software«, lernst du, das Board zu kontrollieren. Du erfährst, wie der Workflow zwischen PC oder Notebook und dem Arduino-Board abläuft und wie der Arduino-Code aufgebaut ist, um den Mikrocontroller steuern zu können.

In Kapitel 3, »Keine Angst vorm Programmieren – Coding Basics«, bekommst du eine geballte Packung Basic-Wissen übers Programmieren präsentiert. Auch dieses Kapitel geht wie die vorherigen Kapitel manchmal sehr in die Tiefe, aber im Zweifelsfall gilt auch hier: Kannst du lesen, musst du jetzt aber nicht. Ich empfehle dir, die ersten Kapitel als eine Art Referenzkapitel anzusehen, auf die du später im Buch – oder sogar später bei deinen eigenen Arduino-Bastelprojekten – zugreifen kannst, wenn du etwas genauer wissen möchtest.

In Kapitel 4, »Das Arduino-Discoveryboard«, stelle ich dir ein von mir entwickeltes Entwickler-Board vor. Ich habe auf einer Platine Standardelektronikbauteile verbaut, die bei den Arduino-Bastelprojekten ständig gebraucht werden. Die Bauteile habe ich auf der Platine fest verlötet, so dass ich nur noch den Arduino anschließen muss und die Projekte dann durchführen kann. Der Vorteil? Die lästige Wurschtelei auf dem Breadboard entfällt weitestgehend und der Schaltungsaufbau ist übersichtlicher. Du kannst das Arduino Discoveryboard nachbauen, dabei kannst du dich gleich im Löten üben.

Und dann geht es endlich los mit den Arduino-Bastelprojekten und deiner Bastelleidenschaft! Die Projekte fangen sehr leicht an, aber das ändert sich recht schnell, sie werden von Projekt zu Projekt komplexer. An den Stellen, an denen Grundlagenwissen notwendig ist, streue ich Texte ein, die das Bastelprojekt etwas theoretischer beleuchten. So wächst hoffentlich von Projekt zu Projekt auch dein Elektronikgrundwissen, so dass du schon bald in der Lage sein wirst, meine Projekte abzuändern oder sogar ganz eigene Bastelprojekte zu entwickeln.

# Wie ich die Bastelprojekte gestaltet habe

Wie schon erwähnt, bauen die Bastelprojekte aufeinander auf. Was du in einem Projekt gelernt hast, kannst und sollst du auch in den Folgeprojekten anwenden. Benötigst du Grundlagenwissen, dann präsentiere ich es an der Stelle, an der es benötigt wird. Die Projekte haben alle einen ähnlichen Aufbau.

Codeanalyse: Schritt für Schritt gehe ich den Programmiercode durch und erkläre genau jeden Aspekt des Programms.

Schaltplan: Der Schaltplan ist die schematische Darstellung des Bastelprojektes, sozusagen der Bauplan.

Schaltungsaufbau: Wie die Schaltung dann auf einem Breadboard aufgesteckt oder auf einer Platine aufgelötet aussieht, das stelle ich mit Fotos dar.

Troubleshooting: Die systematische Fehlersuche ist IMHO eine der besonderen Fertigkeiten, die ich gern jedem angehenden E-Bastler nahebringen möchte. Deshalb habe ich sie zum festen Bestandteil meiner Bastelprojekte gemacht.

Benötigte Bauteile: Ich gebe dir einen Überblick über die in diesem Bastelprojekt benötigten Bauteile. So kannst du auf einen Blick feststellen, ob dir die Bauteile bereits vorliegen.

Programmcode: Der für die Bastelprojekte verwendete Programmiercode wird vorgestellt, manchmal in kleinen Häppchen.

Gelegentlich biete ich *Quick-and-Dirty*-Lösungen an, die auf den ersten Blick etwas umständlich erscheinen mögen. Anschließend folgt eine verbesserte Variante, was als Anregung dienen soll, nach weiteren Lösungsmöglichkeiten zu suchen und – im Idealfall – eigene Lösungen zu entwerfen. Wenn das geschieht, habe ich genau das erreicht, was ich beabsichtigt habe. Falls nicht, auch gut. Jeder geht seinen eigenen Weg und kommt irgendwann ans Ziel.

# Meine Website mit weiteren Arduino- und Elektronikthemen

An dieser Stelle möchte ich auch auf meine Internetseite

https://erik-bartmann.de/

hinweisen, auf der du unter anderem einiges zum Thema Arduino findest. Hier kannst du auch alle Programme herunterladen, die ich in meinen Bastelprojekten verwende. Zu den schönsten Momenten für einen Autor gehört es, wenn er Feedback von Lesern erhält. Ich bin manchmal richtig gerührt, wenn mir ein Leser mitteilt, wie viel Spaß ihm mein Buch beim Lesen und Basteln gemacht hat. Darüber freue ich mich sehr und möchte jeden dazu ermuntern, mir mitzuteilen, wie er mein Buch fand, was ihm besonders gefallen hat und auch, was er für verbesserungswürdig an meinem Buch hält. Meine E-Mail-Adresse lautet:

erik.bartmann@yahoo.de.

Sie ist auf meiner Internetseite auch noch einmal aufgeführt.

# Voraussetzungen

Die einzige persönliche Voraussetzung, die du mitbringen solltest, ist Interesse am Basteln und Tüfteln. Du musst kein Elektronik-Nerd sein und auch kein Computerexperte, um die im Buch gezeigten Bastelprojekte nachvollziehen und nachbauen zu können. Da wir sehr moderat beginnen, besteht absolut keine Gefahr, dass irgendjemand auf der Strecke bleibt. Setz dich also nicht unter Druck und mach die Dinge nicht schwieriger, als sie sind. Der Spaß steht immer an erster Stelle.

# Benötigte Bauteile

Das Arduino-Board für sich alleine ist zwar ganz nett und wir können uns daran erfreuen, wie klein und schön alles konzipiert wurde, doch auf Dauer reicht das natürlich nicht aus. Wir sollten uns daher im nächsten Schritt ansehen, was wir alles von außen an das Board anschließen können. Falls du noch nie in irgendeiner Weise mit elektronischen Bauteilen (wie Widerständen, Kondensatoren, Transistoren oder Dioden, um nur einige zu nennen) in Berührung gekommen bist, ist das nicht weiter schlimm. Die benötigten Teile erkläre ich dir ausführlich, so dass du nachher weißt, wie sie einzeln und innerhalb der Schaltung reagieren. Zu Beginn eines Arduino-Bastelprojektes stelle ich, wie schon oben kurz erwähnt, eine Liste mit den erforderlichen Teilen zur Verfügung, mit deren Hilfe du dir diese Dinge besorgen kannst. Zentraler Bestandteil ist natürlich immer das Arduino-Board, das ich nicht immer explizit erwähnen werde.

Falls du dich an dieser Stelle fragen solltest, was wohl ein Arduino-Board kosten mag und ob du nach dieser Investition deinen gewohnten Lebensstil fortführen kannst, kann ich nur sagen: *Yep, du kannst!* Das Board kostet um die 25€, und das ist wirklich nicht viel. Ich verwende in allen grundlegenden Kapiteln das *Arduino Uno*-Board. Und auch die anderen Bauteile, die wir in den Bastelprojekten verwenden werden, sind bezahlbar.

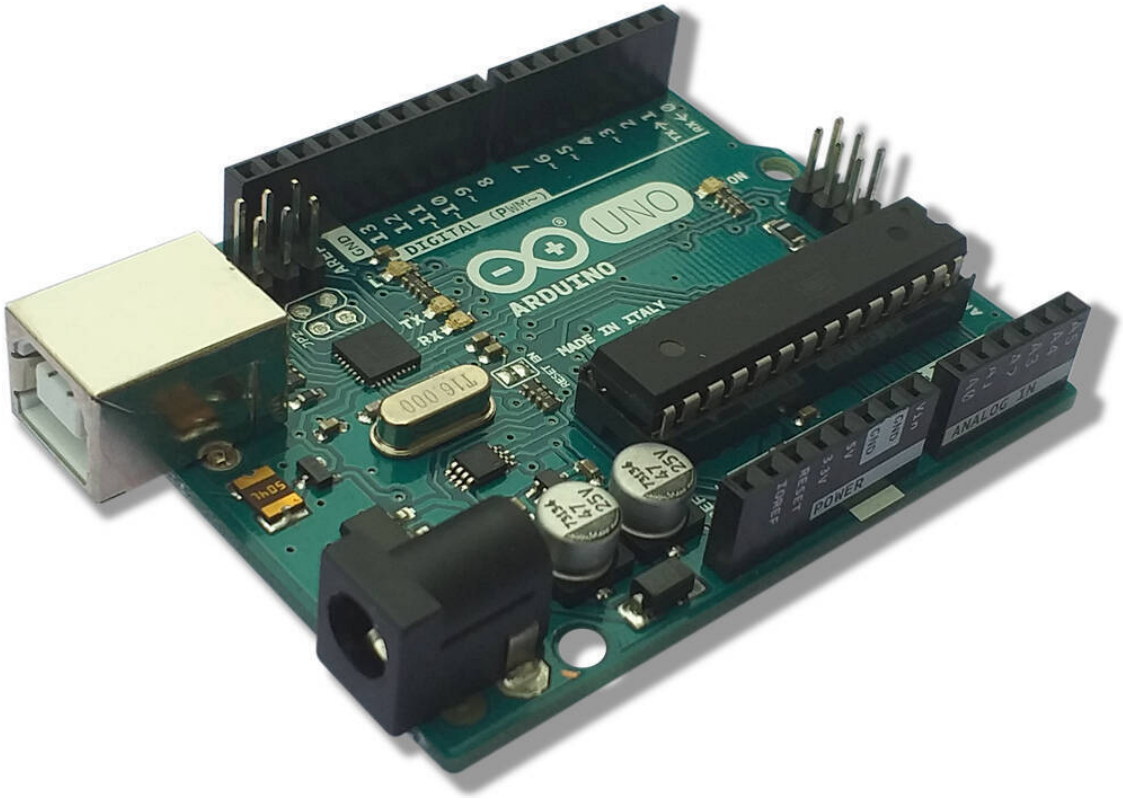



**Abb. 1:** Der Arduino Uno – das Original

# Verhaltensregeln

Wenn du dich so richtig im Brass befindest und voll konzentriert bist auf etwas, was dir unheimlich viel Spaß macht, treten folgende Effekte auf:

Verminderte Nahrungsaufnahme, die zu kritischem Gewichts- und besorgniserregendem Realitätsverlust führen kann.

Unzureichende Flüssigkeitszufuhr bis hin zu Dehydrierung und vermehrter Staubentwicklung in der Umgebung.

Vernachlässigung sämtlicher hygienischer Maßnahmen wie Waschen, Duschen, Zähneputzen, verbunden mit erhöhtem Auftreten von Ungeziefer.

Abbruch jeglicher zwischenmenschlicher Beziehungen.

Lass es nicht so weit kommen und öffne ab und zu das Fenster, um zugewanderten Insekten das Verlassen des Zimmers zu ermöglichen und Frischluft und Sonnenlicht hereinzulassen. Um den oben genannten Effekten entgegenzuwirken, kannst du den Wecker stellen, damit du in regelmäßigen Zeitabständen zu einer Unterbrechung deiner Tätigkeit aufgefordert wirst. Ich möchte mich nach der Veröffentlichung dieses Buches nicht mit einer Beschwerdewelle konfrontiert sehen, die von erbosten Partnern oder vernachlässigten Freunden über mich hereinbricht. Sag also nicht, ich hätte dich nicht vor den Risiken gewarnt. Ich wünsche dir nun eine Menge Spaß und viel Erfolg beim Basteln mit deinem Arduino-Board!

# Kapitel 1:
# Arduino: Die Hardware

Wie in der Einleitung erwähnt, wurde das Arduino-Projekt in Ivrea (Italien) an der dortigen Kunsthochschule entwickelt. In einer Kneipe nahe der Hochschule trafen sich gelegentlich Massimo Banzi und David Cuartielles, die 2005 das erste Arduino-Board entwickelten. Die Kneipe war nach Arduin von Ivrea benannt, der um das Jahr 1000 König von Italien war. *Arduino* ist seitdem die Bezeichnung sowohl für die Software als auch für die Hardware dieses Open-Source-Projektes.

In diesem Kapitel werde ich auf die Arduino-Hardware eingehen. Dabei werde ich auch wichtige Grundbegriffe und -themen erklären, die für die Mikrocontroller-Technik insgesamt von Bedeutung sind. Ich erkläre ausführlich, welche Bauteile zu einem Mikrocontroller gehören. Diese Komponenten befinden sich praktisch in jedem Mikrocontroller-Board, deshalb gehe ich ausführlich darauf ein. Und nebenher frische ich deine Physikkenntnisse ein wenig auf, indem ich einige Grundbegriffe wie Spannung oder Strom behandle.

Ich werde in diesem Kapitel – wie auch im gesamten Buch – mit dem *Arduino Uno*-Board arbeiten. Das Arduino-Uno-Board war das erste Board, das von den Arduino-Machern entwickelt und produziert wurde. Weitere Arduino-Boards kamen dann später hinzu, die technisch etwas besser wurden. Was bedeutet eigentlich besser? Wenn man sich Kennwerte wie Prozessortakt oder verfügbarer Arbeitsspeicher als Entscheidungskriterium für den Kauf eines neuen Arduino-Boards aussucht, dann gibt es sicherlich Boards, die besser geeignet sind, weil sie schneller arbeiten und größere Programme speichern und verarbeiten können. Doch das ist eben nicht immer besser. Für einen geeigneten Einstieg in die elektronische Bastlerwelt ist das Arduino-Uno-Board eben in meinen Augen die bessere Wahl, weil es sehr robust ist und sehr weite Verbreitung gefunden hat.

Der *Arduino Yún* beispielsweise ist sicherlich ein interessantes Board, das einiges an Erweiterungen wie das Betriebssystem Linux bietet. Dennoch hat sich das Board in der Bastler- und Hobbywelt nicht so richtig durchgesetzt, vermutlich auch, weil der Raspberry Pi bereits auf dem Markt war, als Arduino Yún erschien. Scheller und mehr ist eben nicht zwangsläufig auch besser.

Dennoch möchte ich einige wirklich gute Arduino-Bords anführen:

Arduino Leonardo
Arduino Mega 2560
Arduino Nano

Die genannten Boards unterscheiden sich hinsichtlich ihrer Größe und Anzahl der Buchsen, also der Anschlussmöglichkeiten, um mit der Außenwelt in Verbindung zu treten. Des Weiteren haben sie unterschiedliche Prozessoren, Taktfrequenzen und Speichervolumen. Und dennoch arbeiten sie alle nach demselben Prinzip und können durch die einheitliche Arduino-Entwicklungsumgebung angesprochen und programmiert werden. Je nach Anwendungsgebiet und Erfordernissen ist das eine Arduino-Board vielleicht besser geeignet als das andere. Die einen benötigen ein Board mit vielen I/O-Pins und entscheiden sich beispielsweise für den Arduino Mega oder den Due. Andere wählen den Arduino Micro oder Nano aus, denn diese sind recht klein und passen wunderbar in kleine Gehäuse. Sie kommen dort zur Anwendung, wo das Platzangebot beschränkt ist.

Das Universalgenie ist in meinen Augen jedoch der *Arduino Uno* und er wird es wohl noch eine lange Zeit bleiben. Er bietet eine ideale Plattform für den Einstieg in die Mikrocontroller-Welt. Für ihn finden sich im Internet auch die meisten Tutorials, Projekte und Diskussionen. Steigen die Ansprüche für deine Projekte, ist es kein Problem, sich ein weiteres Arduino-Modell zuzulegen, denn die Preise sind wirklich moderat. Viele Bastler legen sich im Laufe der Zeit mehrere unterschiedliche Boards zu, um darüber auch mehr und mehr Erfahrungen zu sammeln, was in meinen Augen ein ganz normaler Entwicklungsfortschritt ist.

Über die nachfolgenden Links bekommst du Detailinformationen zu den gezeigten Boards:

Arduino Uno: https://www.arduino.cc/en/Main/ArduinoBoardUno
Arduino Mega: https://www.arduino.cc/en/Main/ArduinoBoardMega2560
Arduino Leonardo: https://www.arduino.cc/en/Main/ArduinoBoardLeonardo
Arduino Micro: https://www.arduino.cc/en/Main/ArduinoBoardMicro
Arduino Nano: https://www.arduino.cc/en/Main/ArduinoBoardNano

Es gibt noch weitere zahlreiche Arduino-Boards und Erweiterungen, die unter den folgenden Adressen zu finden sind:

https://www.arduino.cc/en/Main/Products
https://www.arduino.cc/en/Main/Boards

# Das Arduino-Uno-Board

Zu Beginn nehmen wir das Arduino-Board unter die Lupe. Auf der folgenden Abbildung habe ich einige der wichtigsten Komponenten auf der Platine markiert und beschriftet:

Wenn ich nun im Folgenden Details über das Arduino-Uno-Board aufliste, wird dir das eine oder andere vielleicht noch nicht ganz verständlich sein, aber ich verspreche, dass ich alle Details später ausführlich erläutere.

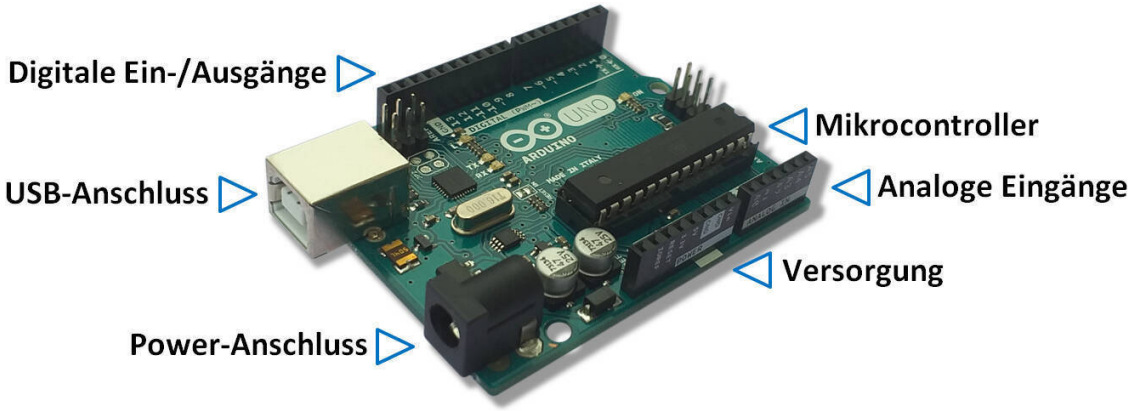



**Abb. 1:** Der Arduino Uno

# Der Mikrocontroller

Der *Mikrocontroller* ist das Herzstück des ganzen Arduino-Boards. Er ist quasi das Rechenzentrum des Arduino-Boards. Beim Arduino Uno kommt der Atmel AVR-Mikrocontroller vom Typ *ATmega328* zum Einsatz. In der vorigen Abbildung 1 ist es das große schwarze Bauteil mit den vielen Anschlüssen. Dieser Baustein wird von der amerikanischen Firma Microchip Technology Inc. hergestellt und ist in vielen Boards zu finden.

Hier eine kurze Übersicht der wichtigsten technischen Daten des Arduino Uno:

Tabelle 1: Ein paar nennenswerte Eckdaten des Arduino-Uno-Boards

| Bezeichnung | Details |
|---|---|
| Mikrocontroller | ATmega328 |
| Arbeitsspannung | 5V |
| Eingangsspannung (empfohlen) | 7V bis 12V |
| Eingangsspannung (Limit) | 6V bis 20V |
| Digitale Ein-/Ausgabe-Pins | 14 (6 stellen PWM zur Verfügung) |
| Analoge Eingänge | 6 |
| DC Strom pro Ein-/Ausgabe-Pin | 40mA |
| DC Strom für 3,3V-Pin | 50mA |
| Flash-Speicher | 32 KByte (ATmega328) davon werden 0,5 KByte vom Bootloader genutzt |
| SRAM | 2 KByte (ATmega328) |
| EEPROM | 1 KByte (ATmega328) |
| Taktrate | 16 MHz |

Weitere Detailinformationen sind unter der folgenden Adresse zu finden:

http://www.arduino.cc/en/Main/ArduinoBoardUno

# Die Ein- und Ausgänge

Um mit einem Mikrocontroller zu kommunizieren, braucht man irgendeine Form der physikalischen Verbindung. Diese Ein- und Ausgänge am Arduino-Board nennt man Ports. Wie du aus der gezeigten Liste in Tabelle 1 entnehmen kannst, steht uns zur Kommunikation mit dem Arduino-Board eine bestimmte Anzahl von Ein- und Ausgängen zur Verfügung. Sie stellen die Schnittstelle zur Außenwelt dar und ermöglichen uns, Daten an den Mikrocontroller zu senden und von ihm zu empfangen. Wirf einen Blick auf das folgende Diagramm:

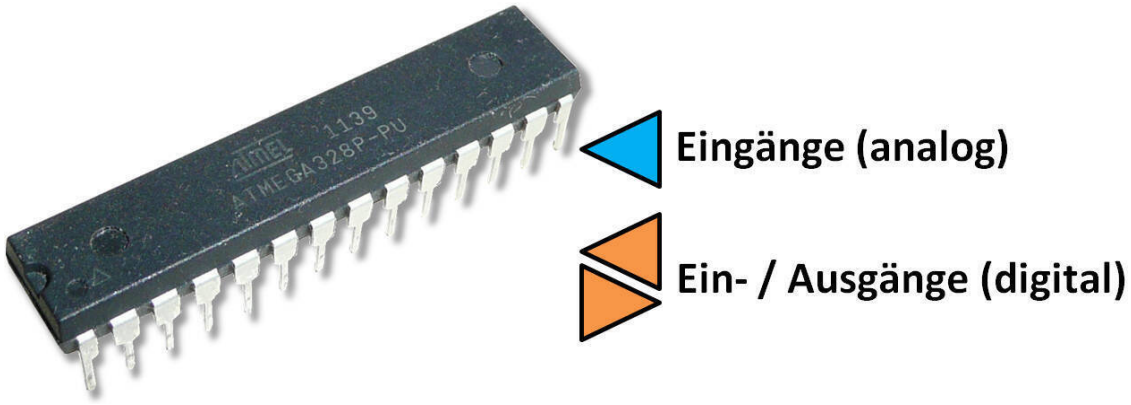



**Abb. 2:** Die Ein- und Ausgänge des Arduino Uno

Du siehst hier den Mikrocontroller, der über bestimmte Schnittstellen mit uns kommunizieren kann. Manche Ports sind als Eingänge, andere als Ein- und Ausgänge vorhanden.

Was ist ein Port?

Ein Port ist ein definierter Zugangsweg zum Mikrocontroller, sozusagen eine Tür ins Innere, die wir nutzen können, um mit ihm zu kommunizieren. Einzelne Ein-/Ausgabe-Pins werden in der Regel in Ports organisiert, die meistens einem Register in einem Mikrocontroller zugeordnet sind, wobei ein Register einen bestimmten Speicherbereich darstellt.

Wirf noch einmal einen Blick auf das Board, und du wirst an der Ober- beziehungsweise Unterkante jeweils schwarze Buchsenleisten erkennen. Du wirst dich möglicherweise fragen, wie denn zum Beispiel bei den digitalen Ein- oder Ausgängen entschieden wird, welche als Eingäge und welche als Ausgänge arbeiten. Vielleicht arbeiten ja einige nur als Eingänge und andere wiederum nur als Ausgänge. Das würde das Ganze aber recht unflexibel machen und deswegen gibt es eine viel bessere Lösung. Und wie sieht es mit den analogen Ausgängen aus? Die fehlen hier komplett.

Beginnen wir mit den digitalen Pins, wobei ein einzelner Pin ein separater Anschluss eines Ports ist, der mehrere Pins in sich vereint. Wie den Spezifikationen zu entnehmen ist, verfügt der Arduino Uno über 14 digitale Ein-/Ausgabe-Pins, die abgekürzt I/O-Pins genannt werden, wobei I/O für Input/Output steht. Über die Programmierung kann jetzt sehr flexibel entschieden werden, welcher der 14 Pins als Eingang und welcher als Ausgang arbeiten soll. Dieser Vorgang wird *Konfiguration* genannt.

Doch nun zu den analogen Pins. Unser Arduino-Board ist nicht mit separaten analogen Ausgängen versehen. Das hört sich erst einmal recht merkwürdig an, doch bestimmte digitale Pins können als analoge Ausgänge genutzt werden. Du fragst dich bestimmt, wie das funktionieren soll. Ich greife etwas vor auf das, was im Bastelprojekt 1 über die *Puls-Weiten-Modulation*, auch *PWM* genannt, kommt. Das ist ein Verfahren, bei dem ein Signal mehr oder weniger lange An- und Aus-Phasen

vorweist. Ist die An-Phase, also wenn der Strom fließt, länger als die Aus-Phase, leuchtet zum Beispiel eine angeschlossene Lampe augenscheinlich heller, als wenn die Aus-Phase länger ist. Ihr wird also mehr Energie in einer bestimmten Zeit in Form von elektrischem Strom zugeführt. Aufgrund seiner Trägheit kann unser Auge sehr schnell wechselnde Ereignisse nicht unterscheiden und auch die Lampe weist beim Hin- und Herschalten zwischen den beiden Zuständen Ein und Aus eine gewisse Trägheit auf. Dadurch hat es für uns den Anschein einer sich verändernden Ausgangsspannung. Klingt etwas merkwürdig, nicht wahr? Du wirst es ganz sicher besser verstehen, wenn wir das entsprechende Kapitel erreichen. Einen offensichtlich entscheidenden Nachteil hat diese Art der Portverwaltung schon. Verwendest du einen oder mehrere analoge Ausgänge, geht das zulasten der digitalen Portverfügbarkeit. Du hast dafür eben weniger zur Verfügung. Doch das soll uns nicht weiter stören, denn wir kommen nicht an die Grenzen, die eine Einschränkung unserer Versuchsaufbauten bedeuten würde. In den Spezifikationen wurde kurz der Begriff *Bootloader* erwähnt, was einer Erklärung bedarf.

Was ist ein Bootloader?

Ein Bootloader ist ein kleines Programm, welches auf dem Mikrocontroller ATmega328 einmalig installiert werden muss. Wird ein Arduino-Board gekauft, ist dieser Bootloader vom Hersteller schon vorinstalliert. Dieses Programm wurde in einem bestimmten Bereich des Flash-Speichers auf dem Mikrocontroller abgelegt und ist unter anderem für das Laden des eigentlichen Programms nach der Verbindung mit der Stromversorgung verantwortlich. Ist ein derartiges Programm vorhanden, wird es ausgeführt. Andernfalls wird so lange gewartet, bis es im Speicher abgelegt wird.

Der zur Verfügung stehende Flash-Speicher wird dadurch natürlich um die Größe des Bootloaders verkleinert. Normalerweise wird ein Mikrocontroller über eine zusätzliche Hardware, zum Beispiel einen *ISP-Programmer*, mit dem Arbeitsprogramm versehen. Durch den Bootloader entfällt diese Notwendigkeit, und so gestaltet sich der Upload der Software wirklich komfortabel. Nach dem erfolgreichen Übertragen des Arbeitsprogramms in den Arbeitsspeicher des Controllers wird es unmittelbar zur Ausführung gebracht. Angenommen, du müsstest deinen Mikrocontroller ATmega328 auf der Platine aus irgendeinem Grund austauschen, dann würde der neue Mikrocontroller nicht wissen, was zu tun wäre, da der Bootloader standardmäßig noch nicht geladen ist. Für diese Prozedur gibt es verschiedene Verfahren, die ich aus Platzgründen hier nicht erklären kann. Im Internet finden sich aber genügend Informationen, wie du den passenden Bootloader für den Mikrocontroller installieren kannst.

# Die Spannungsversorgung

Fangen wir mit der Spannungsversorgung an, denn ohne die geht es wirklich nicht. Es gibt verschiedene Möglichkeiten. Wenn wir mit dem Arduino arbeiten oder ihn programmieren, dann ist natürlich eine USB-Verbindung zum Rechner notwendig. Diese Verbindung hat zwei Aufgaben:

die erforderliche Spannungsversorgung von 5V herzustellen und
einen Kommunikationskanal zwischen Rechner und Arduino-Board bereitzustellen.

Beide Punkte erfolgen über die silberfarbene USB-Buchse, die in der Abbildung mit *USB-Anschluss* bezeichnet ist. Wie der Anschluss genau aussieht, von welchem Typ er ist und wie die Installation der notwendigen Software vonstattengeht, sehen wir im nachfolgenden Kapitel. Für das Entwickeln und Testen von Programmen ist dieser Weg zum Board genau richtig. Jetzt gibt es aber noch eine zweite Buchse, die sich direkt rechts daneben befindet und hier mit *Power-Anschluss* gekennzeichnet ist. Dieser Anschluss wird benötigt, um das Arduino-Board nach der Programmierung unabhängig vom Rechner zu machen, über den es zuvor programmiert wurde. Was könnte ein Grund dafür sein, diesen Weg zu beschreiten? Ganz einfach! Angenommen, wir haben ein Roboterfahrzeug gebaut und der Arduino soll jetzt per Funk Befehle von einem Sender empfangen und entsprechende Fahraktionen ausführen. Da wäre ein USB-Kabel sehr hinderlich und würde den Aktionsradius sehr einschränken. Wir benötigen lediglich eine Spannungsversorgung für das Board und gegebenenfalls eine zusätzliche für die Motoren, denn die Programmierung ist abgeschlossen. Ganz nach dem NASA-Spruch beim Raketenstart: *Guidance is internal*. Der Arduino ist auf sich allein gestellt! Auf der folgenden Abbildung 3 sehen wir die Möglichkeit der externen Spannungsversorgung über eine 9V-Blockbatterie, was jedoch auf Dauer auch keine Lösung darstellt:

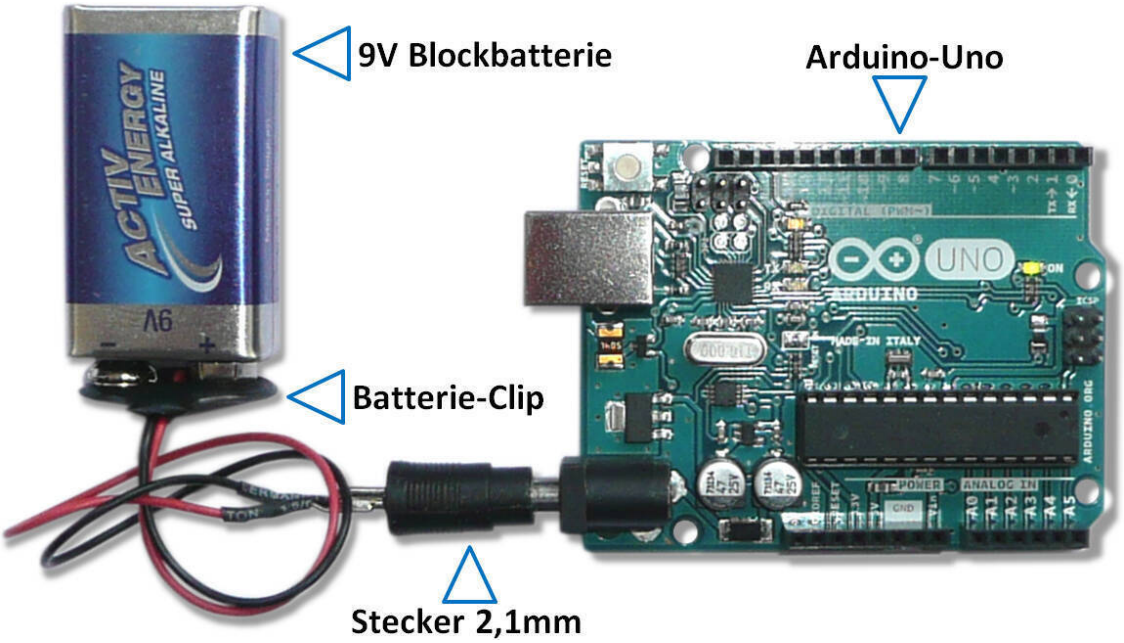



**Abb. 3:** Der Arduino Uno mit externer Spannungsversorgung über eine Batterie

Die vorgeschlagene Spannung sollte sich im Bereich von 7V bis 12V (DC = Direct Current/ Gleichspannung) bewegen. Die beste Variante hinsichtlich einer externen Spannungsversorgung stellt ein Steckernetzteil mit einem 2,1mm-Plug dar, was für unter 10€ zu bekommen ist. Also sollte es zum Beispiel 9VDC/1A liefern:

**Abb. 4:** Eine Spannungsversorgung mit einem Steckernetzteil

Außerdem gibt es noch einen Pin mit dem Namen *Vin*, der ebenfalls zur Spannungsversorgung genutzt werden kann. Detailinformationen sind unter den folgenden Adressen zu finden:

http://www.arduino.cc/en/Main/ArduinoBoardUno
http://playground.arduino.cc/Learning/WhatAdapter

Grundbegriffe verstehen

Apropos Spannungsversorgung. Was ist eigentlich Spannung und wie kommt sie zustande? Wir haben auch noch nicht über Strom gesprochen. Du kannst jetzt deine Elektronikgrundkenntnisse auffrischen, das kann dir mehr Möglichkeiten mit deinem Arduino eröffnen.

# Physikalische Größen

Da ohne eine vorhandene Spannung elektrische Geräte nichts weiter sind als nutzlose Elektrogeräte, möchte ich auf die physikalischen Größen kurz eingehen. Zu den physikalischen oder auch elektrischen Grundgrößen gehören vor allem die elektrische Spannung, der elektrische Strom, der elektrische Widerstand und die elektrische Ladung. Um mit deinem Arduino eigene Projekte bauen zu können und sie nicht nur nachzubauen, ist es wichtig, den Zusammenhang zwischen Spannung, Strom und Widerstand zu verstehen. Da dies kein Elektroniklehrbuch ist, sondern ein praxisbezogenes Arduino-Buch, kann ich dies nur kurz anreißen. Falls dahingehend Interesse besteht, verweise ich auf mein Buch *Elektronik verstehen durch spannende Experimente – Analog- und Digitaltechnik*, ISBN 978-3-946496-23-6, das die Grundlagen der Elektronik vertiefend darstellt.

# Was ist Spannung?

Alle uns umgebenden Materialien – fest, flüssig oder gasförmig – bestehen aus Atomen. Ein einzelnes Atom ist aus einem Atomkern und einer Atomhülle zusammengesetzt, wobei der Atomkern aus positiv geladenen Protonen und neutralen Neutronen besteht. In der Atomhülle flitzen die negativ geladenen Elektronen um den Kern herum. Ist die Anzahl der Elektronen gleichmäßig verteilt, dann bemerken wir hinsichtlich einer elektrischen Erscheinung eigentlich nichts. Wir merken erst etwas, wenn das elektrische Gleichgewicht in irgendeiner Weise gestört wird. Werden zum Beispiel einem Körper diese negativen Ladungsträger, die durch die Elektronen repräsentiert werden, entzogen und einem anderen Körper zugeführt, dann besteht zwischen den beiden Körpern ein elektrischer Zustand, der das vormals vorhandene Ladungsgleichgewicht aufgehoben hat. Je größer dieses Ungleichgewicht, desto größer auch der elektrische Zustand.

Das ist vergleichbar mit zwei Menschen, die unterschiedlicher Ansicht sind und zwischen denen sich aus diesem Grund eine bestimmte Spannung aufbaut. Je größer die Differenzen, desto größer auch die Spannung. Und somit wurde der erste Aspekt eines elektrischen Zustandes zur Sprache gebracht: die *elektrische Spannung*.

Bei der genannten Trennung der Ladungen handelt es sich um einen ruhenden, also statischen Zustand, denn es bewegt sich nach der Trennung der Ladungsträger nichts. Die vorherrschende Spannung kann gemessen werden. Die Einheit für die Spannung lautet *Volt* und wird mit dem Buchstaben *V* abgekürzt.

So weit so gut, doch was nützt einem dieser Ladungsunterschied, der eine bestimmte Spannung repräsentiert? Der statische Zustand ändert sich schlagartig, wenn zwischen den beiden Körpern eine mehr oder weniger leitende Verbindung hergestellt wird, die die Elektronen überbrücken können. Das kann durch die unterschiedlichsten Materialien wie zum Beispiel Kupfer, Aluminium oder Silber erfolgen.

Auf der folgenden Abbildung 5 ist auf der linken Seite eine gleichmäßige Verteilung von negativen und positiven Ladungsträgern zu sehen. Beide liegen in einem ausgewogenen Verhältnis vor und aus diesem Grund gibt es keinen Potentialunterschied und keine Spannung. Im Gegensatz dazu sind auf der rechten Seite die Ladungsträger getrennt. Die linke Seite wird von den negativen, die rechte von den positiven Ladungsträgern dominiert. Dieser Potentialunterschied führt zu einer Spannung.

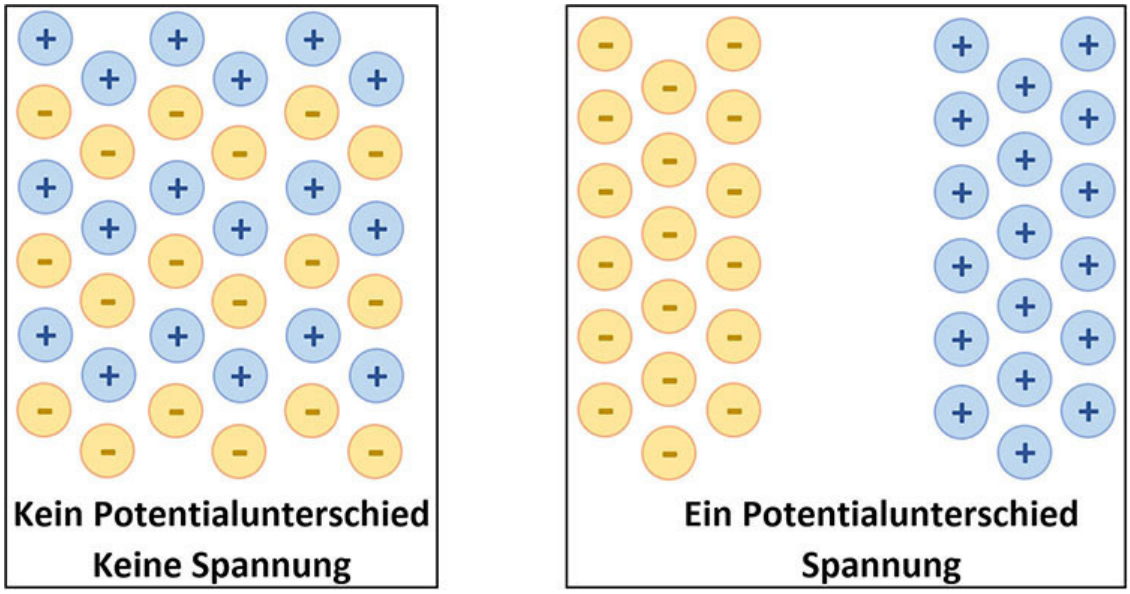


**Abb. 5:** Wann liegt eine Spannung vor?

Ein etabliertes Ladungsungleichgewicht hat stets das Bestreben, einen Ausgleich herbeizuführen und die Elektronen wandern bei einer Verbindung von einem zum anderen Körper, wobei die Wanderung vom Elektronenüberschuss zum Elektronenmangel erfolgt. Erst, wenn diese Möglichkeit gegeben ist, kommt es zu einem Stromfluss, wie das auf der folgenden Abbildung 6 zu sehen ist:

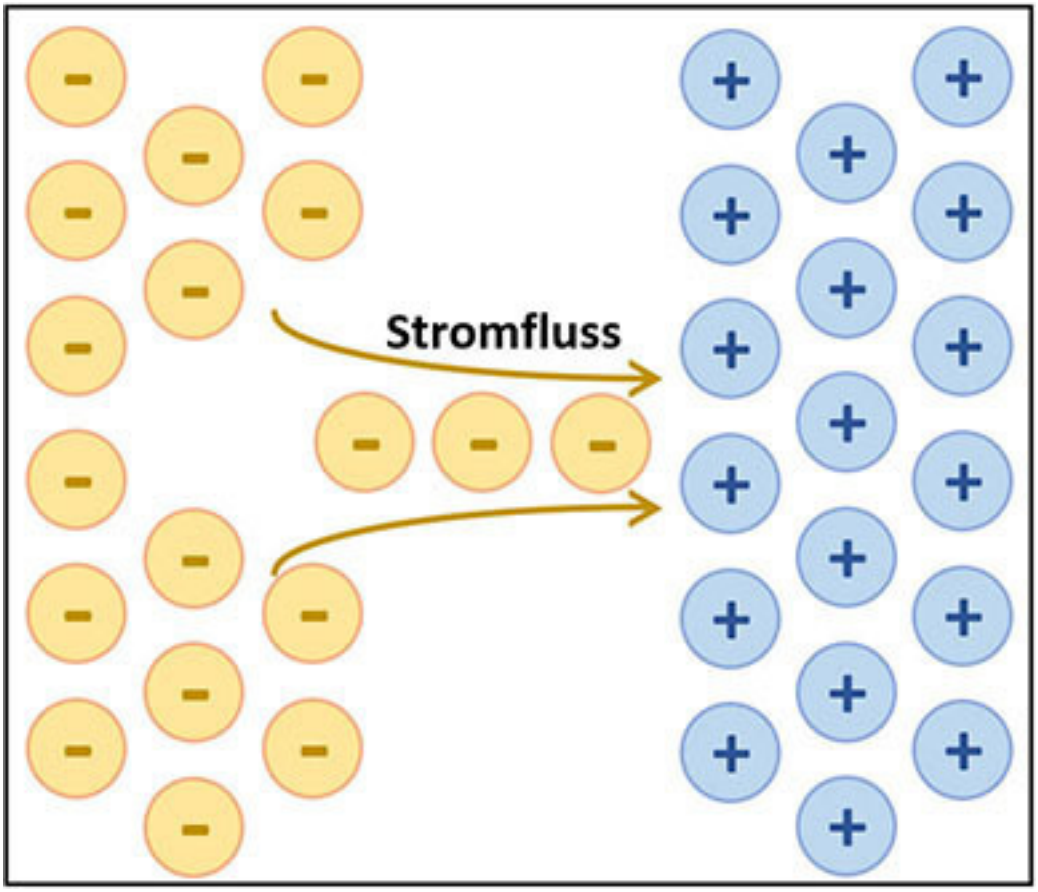



**Abb. 6:** Ein Stromfluss kommt zustande.

Sicherlich hast du schon einmal davon gehört, dass es unterschiedliche Spannungsformen gibt, die sich *Gleich-* und *Wechselspannung* nennen. Auf der folgenden Abbildung 7 sind diese Spannungsformen im zeitlichen Verlauf zu sehen, wobei das Arduino-Board mit Gleichspannung betrieben wird. Diese Spannungsform – es wird auch von Stromform gesprochen – hat die Eigenschaft, dass sich Stärke und Richtung nicht ändern. Gleichspannung beziehungsweise Gleichstrom wird mit den Buchstaben *DC* für *Direct Current* bezeichnet. Im Gegensatz dazu gibt es noch den Wechselstrom, der mit *AC* für *Alternating Current* bezeichnet wird. Die Rockgruppe ACDC hat daher übrigens auch ihren Namen. Beide Formen sehen wir auf der folgenden Abbildung 7:

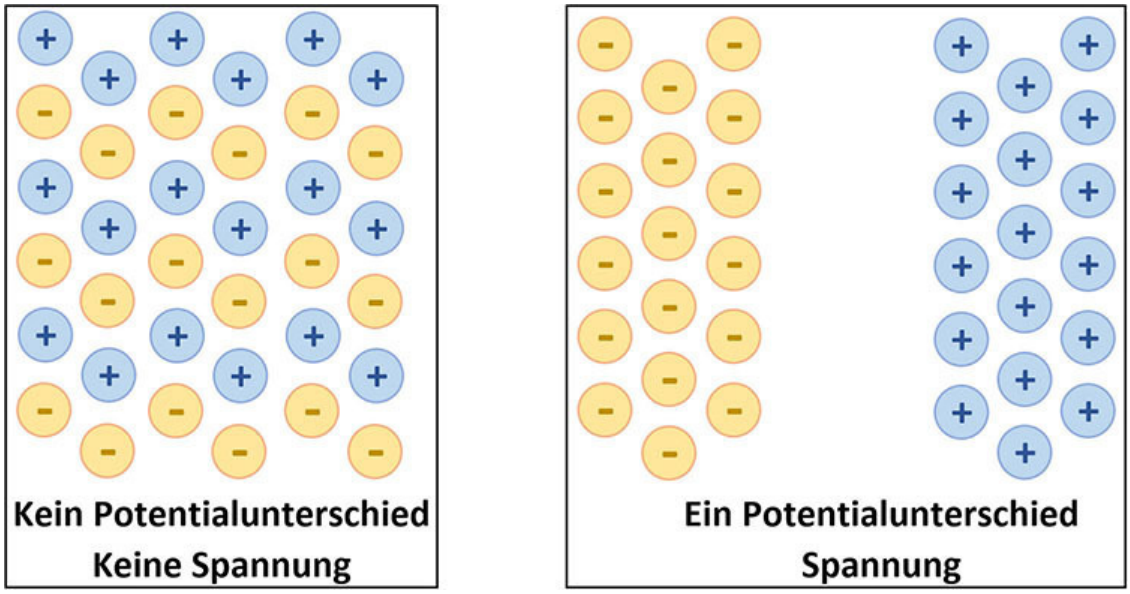



**Abb. 7:** Gleich- und Wechselspannung

Bei Gleichstrom ändert sich die Höhe der Spannung U nicht, wobei der Wechselstrom die Charakteristik hat, dass die Spannung U zwischen einem positiven und negativen Grenzwert hin- und herpendelt. Es handelt sich hierbei um einen sinusförmigen Verlauf.

# Was ist Strom?

Nun hast du schon gehört, dass bei einer vorhandenen Spannung ein elektrischer Strom fließen kann. Doch was ist elektrischer Strom überhaupt? Was passiert in einem leitenden Material wie zum Beispiel Kupfer, wenn Strom hindurchfließt? Durch das Anlegen einer Spannung und dem damit einhergehenden Potentialunterschied werden die auf der äußersten Schale von Atomen vorhandenen Elektronen herausgelöst und wandern als freie Elektronen wie eine Wolke durch das Material. Das passiert deswegen, weil durch das Herauslösen der freien Elektronen ein Atom zu einem sogenannten *Ion* wird. Ein Ion ist dabei ein Atom mit einem Ungleichgewicht zwischen Elektronen und Protonen. Das Atom ist dann nach außen hin nicht mehr neutral. Ein herausgelöstes Elektron hinterlässt also eine Lücke im Atomverbund, die dann durch ein anderes freies Elektron wieder gefüllt werden kann.

Dieser Prozess des Herauslösens und Wiederauffüllens der Lücken durch die Elektronen wird als elektrischer Strom bezeichnet. Auf den beiden folgenden Abbildungen ist der Stromfluss durch einen Leiter zu sehen. Dabei spielen der Querschnitt des Leiters und der zu beobachtende Zeitabschnitt zur Strommessung eine entscheidende Rolle. In der ersten Abbildung sind pro markierten Abschnitt ganze sechs Elektronen zu sehen. Nicht viel, wie später noch zu sehen ist, aber das spielt auch im Moment eine untergeordnete Rolle.

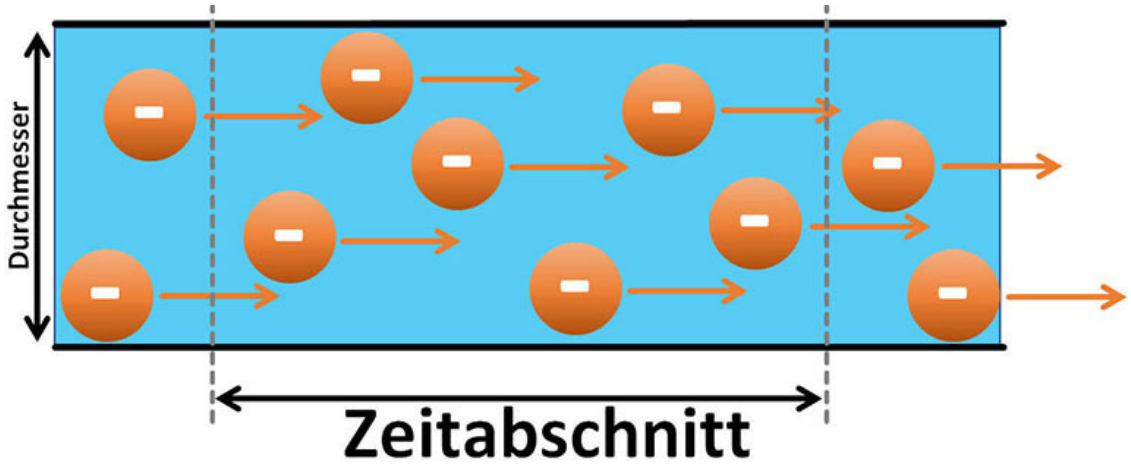



**Abb. 8:** Zahlreiche Elektronen auf dem Weg durch einen Leiter

In der zweiten Abbildung sind pro markierten Abschnitt im Vergleich dazu weniger Elektronen, nur vier, zu sehen. Der Stromfluss ist also geringer.

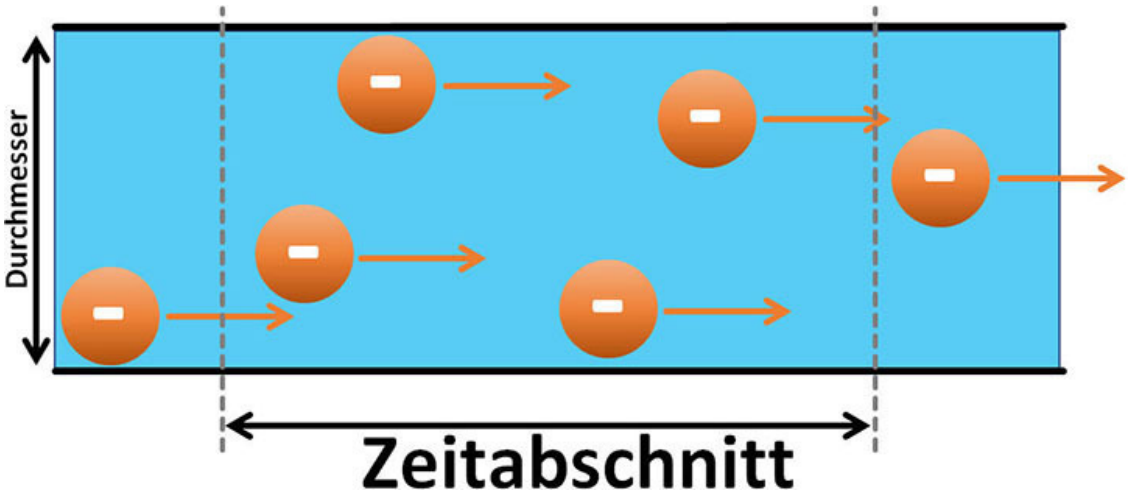


**Abb. 9:** Wenige Elektronen auf dem Weg durch einen Leiter

Der elektrische Stromfluss kann also über das folgende Verhältnis definiert werden:

$$Stromstärke = \frac{Anzahl\ der\ Elektronen}{Zeitabschnitt}$$

Natürlich spielt auch der Durchmesser des Leiters eine Rolle, denn je enger der Durchflusskanal für die freien Elektronen ist, desto größer ist auch der Widerstand und desto geringer der Stromfluss. Da die Elektronen eine Ladung besitzen, kann die Summe der zu betrachtenden Elektronen pro Zeitabschnitt auch als sogenannte *Ladungsmenge* angesehen werden. Eine derartige Ladungsmenge besitzt ebenfalls einen Formelbuchstaben und wird mit *Q* bezeichnet. Jetzt kann das Ganze mathematisch schon etwas präziser ausgedrückt werden:

$$I = \frac{\Delta Q}{\Delta t}$$

Der Buchstabe *I* steht für den Strom und das kleine Dreieck Δ – auch Delta genannt – wird in der Mathematik für Änderungen oder Differenzen eingesetzt. Es bedeutet also, dass die Stromstärke gleich der Änderung der Ladungsmenge *Q* pro Zeit *t* ist. Für das oben gezeigte Beispiel von sechs beziehungsweise vier Elektronen pro Zeitabschnitt kann festgestellt werden, dass das sehr, sehr, sehr wenig ist und die folgende Rechnung kommt der Realität etwas näher. Es gilt, die Anzahl der Elektronen zu ermitteln, die bei einer Stromstärke von 1A (A steht für Ampere, das ist die Maßeinheit des elektrischen Stroms) im Zeitabschnitt von 1 Sekunde durch den Leiter flitzen. Wie kann das aber berechnet werden? Es wird dazu die gezeigte Formel nach der Ladungsmenge umgestellt, so dass sie wie folgt lautet:

$$\Delta Q = I \cdot \Delta t$$

Mit den eingetragenen Vorgaben sieht das dann wie folgt aus:

$$\Delta Q = 1As$$

Wie schon erwähnt, ist die Ladungsmenge *Q* die Anzahl *n* der Elektronen und nun ist es zur Lösung der gestellten Aufgabe gut zu wissen, wie denn die sogenannte Elementarladung eines Elektrons ist. Diese wird mit dem Formelbuchstaben *e* gekennzeichnet und hat den Wert von $1{,}602176 \cdot 10^{-19}$ C. Das *C* (Coulomb) ist dabei die Bezeichnung beziehungsweise die Maßeinheit einer elektrischen Ladung. Nun kann in der letzten Formel das ΔQ durch die Anzahl der Elementarladungen ersetzt werden, so dass sie wie folgt lautet:

$$n \cdot e = 1As$$

Hier ist auch sehr gut zu sehen, dass *1C* (Coulomb) auch die Maßeinheit *As* besitzen kann, was gleich beim Kürzen in der Formel zum Tragen kommt. Umgestellt nach *n* und mit konkreten Werten versehen lautet die Formel inklusive des Ergebnisses dann folgendermaßen:

$$n = \frac{1As}{1{,}602176 \cdot 10^{-19}As} = 6{,}24151 \cdot 10^{18}$$

Das Ergebnis ist aufgrund der sehr hohen Anzahl der Elektronen (6 Trillionen) schon beeindruckend.

Nun habe ich die elektrischen Größen wie die elektrische Spannung und den elektrischen Strom kurz angesprochen und es fehlt noch der elektrische Widerstand. Damit das jedoch nicht zu trocken wird und ein Herunterbeten von puren Fakten ist, wird der elektrische Widerstand in den Kapiteln besprochen, in denen er wichtig ist. Das ist zum Beispiel im Bastelprojekt 1 über die Ansteuerung einer Leuchtdiode der Fall, denn dieses elektronische Bauelement, das wie eine kleine Lampe arbeitet, darf nicht ohne Begrenzung des Stromflusses betrieben werden. Und diese Funktion der Strombegrenzumg übernimmt ein bestimmtes elektrisches Bauteil. Doch dazu später mehr.

# Unterschiedliche Signalarten

Ich habe ja bereits weiter oben in diesem Kapitel die Ein- und Ausgänge am Arduino Uno gezeigt. Diese werden über sogenannte *Header* zur Verfügung gestellt, wie das auf der folgenden Abbildung zu erkennen ist:

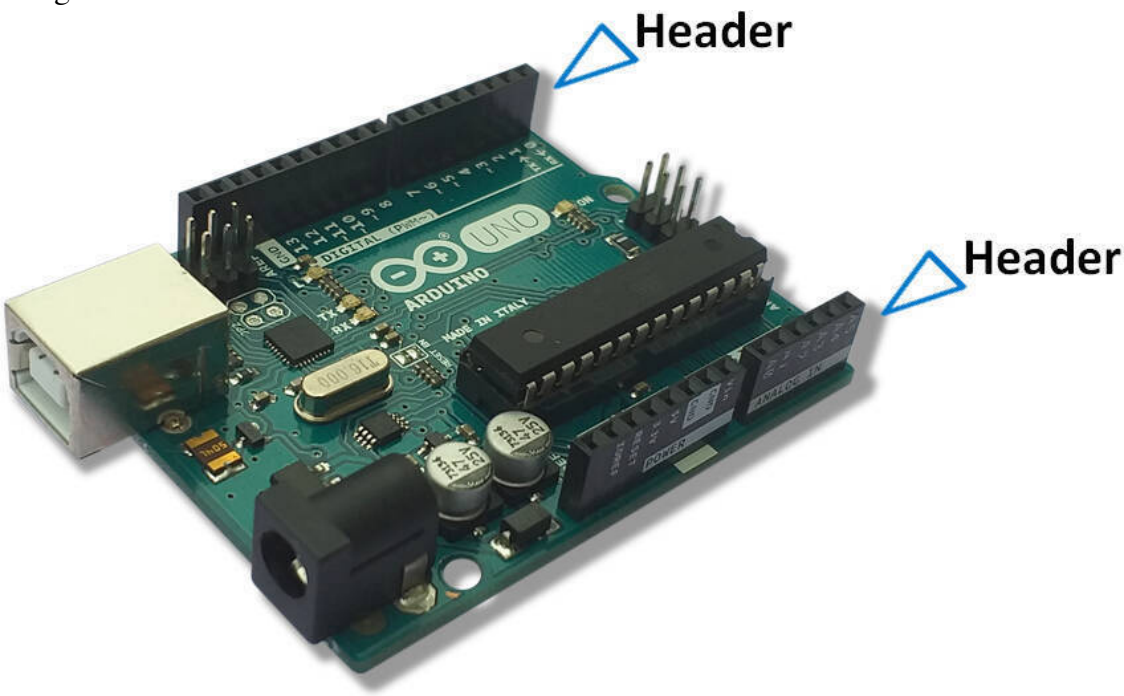



**Abb. 10:** Die Header des Arduino Uno

In die Header können entweder kleine sogenannte Patchkabel oder direkt Bauteile eingesteckt werden. Es gibt unterschiedliche Kategorien von Anschlüssen, auf die ich jetzt ein wenig genauer eingehen möchte. Zuvor muss ich jedoch noch etwas ausholen, damit der Unterschied zwischen analogen und digitalen Signalen verständlicher wird.

In der Natur sind sogenannte stufenlose Signalformen mit stufenlosen Verläufen wie zum Beispiel Temperatur oder Helligkeit zu finden. Wenn wir uns die Signalkurve eines Tons auf einem Oszillographen ansehen, dann hat er im Falle einer Sinuskurve folgenden Verlauf:

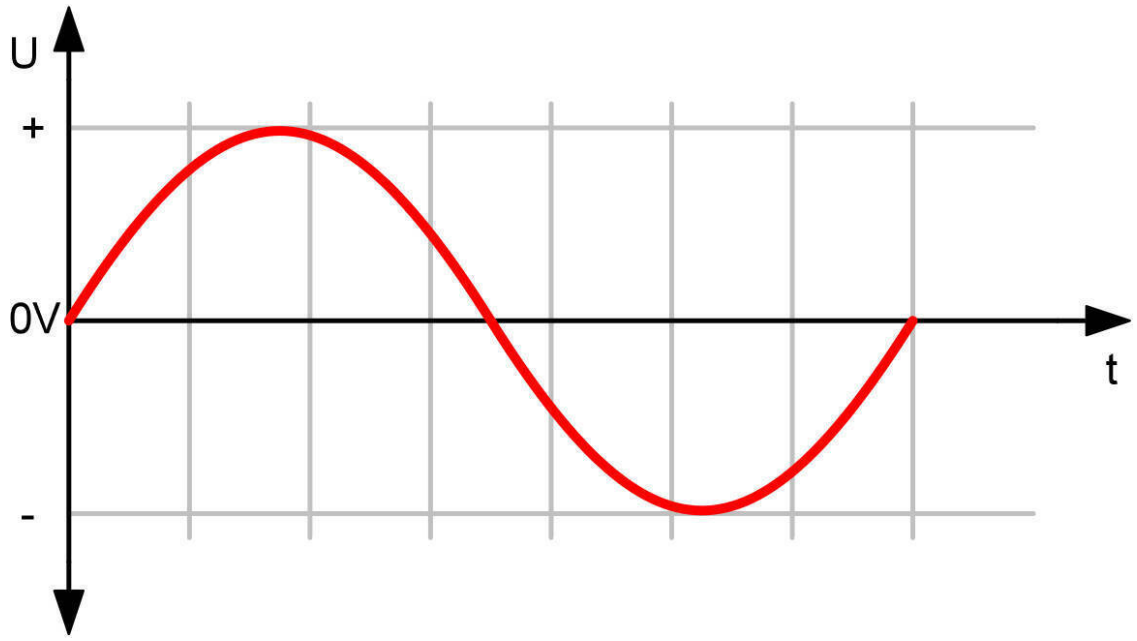



**Abb. 11:** Ein sinusförmiger Kurvenverlauf

Wir erkennen entlang der horizontalen Zeitachse *t* keine Unterbrechung und die Werte gehen stufenlos ineinander über. Wollten wir ein derartiges Signal in einem Computer abbilden, hätten wir Probleme, denn ein Computer kennt lediglich *An* oder *Aus*, was Spannungspegeln von HIGH (5V TTL-Logik) beziehungsweise LOW (0V) entspricht. Ein derartiger Verlauf sieht dann beispielsweise wie folgt aus:

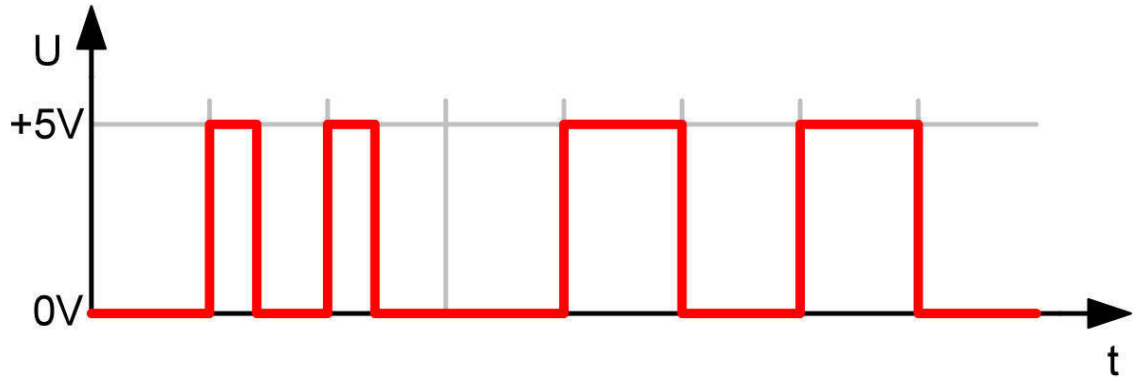



**Abb. 12:** Ein digitaler Kurvenverlauf

Hinsichtlich des analogen Spannungsverlaufs sind beim Arduino und auch anderen Mikrocontrollern Einschränkungen vorhanden, auf die ich später noch zu sprechen komme.

# Die digitalen Ein- und Ausgänge

Der Arduino Uno verfügt über diverse digitale Ein- und Ausgänge. Auf der folgenden Abbildung erkennen wir neben den Buchsen entsprechende Bezeichnungen:

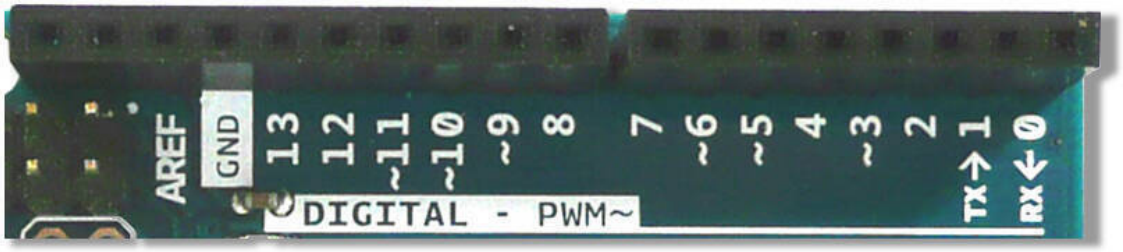



**Abb. 13:** Die digitalen Ein- und Ausgänge

Es gibt von rechts nach links eine Durchnummerierung von 0 bis 13, wobei manchen Nummern eine Schlangenlinie – auch *Tilde* genannt – voransteht. Auf diese Besonderheit gehe ich gleich noch genauer ein. Ebenso gibt es zwei Pins, an denen sich die Zusatzinformationen *RX* und *TX* befinden. Sie sollten im Normalfall nicht zur Ansteuerung verwendet werden, da sie von der seriellen Schnittstelle standardmäßig belegt sind. Die Buchse mit der Bezeichnung *GND* (Ground) stellt die Masse zur Verfügung. Vielleicht stellt sich der eine oder andere nun die Frage, welche der genannten Anschlüsse zur Kategorie Ein- beziehungsweise Ausgänge gehört. Die Antwort ist einfach. Jeder der digitalen Anschlüsse kann über die Programmierung frei konfiguriert werden und sowohl als Ein- oder als Ausgang arbeiten. Wenn ein Anschluss als Eingang arbeiten soll, muss auf jeden Fall darauf geachtet werden, dass die Spannung 5V nicht überschreitet! Andernfalls nimmt der Mikrocontroller solchen Schaden, dass er entsorgt werden muss.

# Die analogen Ein- und Ausgänge

Kommen wir nun zu den analogen Verbindungen. Auf der folgenden Abbildung 14 sind die analogen Eingänge zu erkennen:

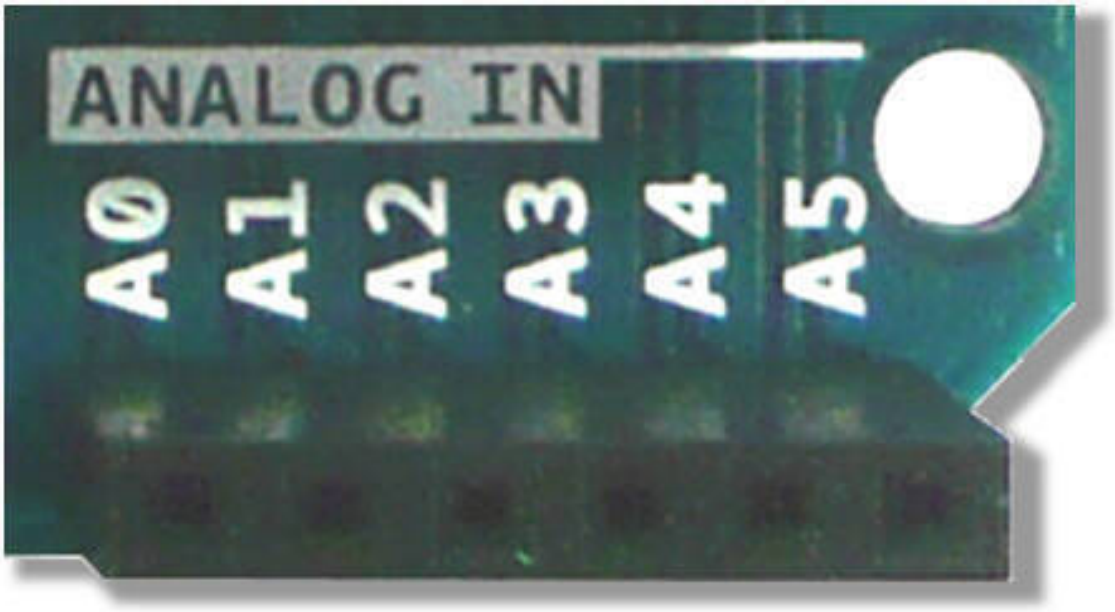



**Abb. 14:** Die analogen Eingänge

Die Bezeichnungen von A0 bis A5 ergeben insgesamt sechs analoge Eingänge. Auch hier besteht die Einschränkung, dass nur mit Spannungswerten zwischen 0V und 5V gearbeitet werden darf. Wer jetzt jedoch Ausschau nach den analogen Ausgängen hält, wird auf dem Arduino-Board lange suchen und dann vielleicht entmutigt feststellen, dass sie vergessen wurden. Na ja, nicht ganz. Sie befinden sich an einer anderen Stelle und haben ein Verhalten, das sich von dem unterscheidet, was sich die meisten vielleicht darunter vorstellen.

Nun sind wir bei den digitalen Anschlüssen mit den Tilden angelangt. Das wären dann die Anschlüsse D3, D5, D6, D9, D10 und D11. Sie können als analoge Ausgänge konfiguriert werden. Digitale Ausgänge sollen als analoge Ausgänge umfunktioniert werden?! Das hört sich schon etwas merkwürdig an.

Jetzt kommt *PWM* ins Spiel. Diese Abkürzung steht für *Puls-Width-Modulation*, was übersetzt Puls-Weiten-Modulation heißt. Wir haben es hier jedoch mit einem digitalen statt mit einem analogen Signal zu tun. Das scheint auf den ersten Blick nicht logisch, doch sehen wir uns das genauer an. Ein PWM-Signal besitzt eine konstante Frequenz mit einer konstanten Spannung. Was jedoch variieren kann, ist der sogenannte *Tastgrad*.

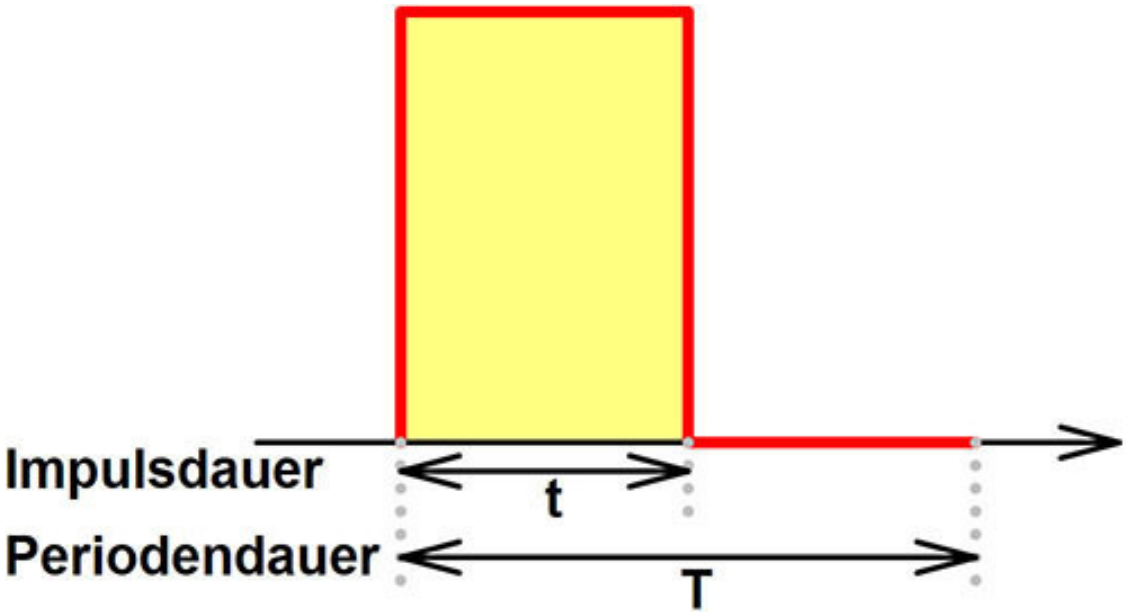



**Abb. 15:** Impuls- und Periodendauer im zeitlichen Verlauf

Wenn die Frequenz *f* gleich bleibt, bedeutet dies, dass die Periodendauer *T* ebenfalls konstant ist. Das einzige, was sich ändern kann, ist die Impulsdauer *t*. Je breiter der Impuls – quasi größere Fläche –, desto größer ist auch die Energie, die an den jeweiligen Verbraucher übertragen wird. Sehen wir uns vier markante Möglichkeiten an.

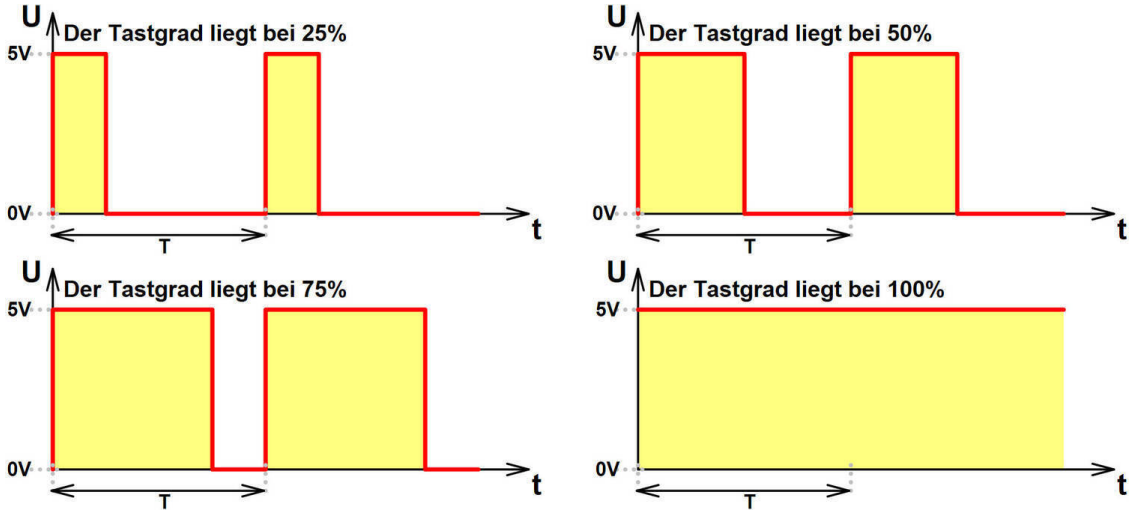



**Abb. 16:** Einige PWM-Beispiele

Wie die Ansteuerung funktioniert, werden wir noch im Detail besprechen.

# Die interne Stromversorgung

An der Buchsenleiste liegen die folgenden Pins und Funktionen:

Tabelle 2: Spannungsversorgungs-Pins

| Bezeichnung | Funktion |
|---|---|
| VIN | Eingangsspannung für das Arduino-Board für externe Spannungsversorgung |
| 5V | Regulierter 5V-Ausgang |
| 3V3 | Regulierter 3,3V-Ausgang. Maximaler Strom beträgt 50mA. |
| GND | Masse |
| IOREF | Die Referenzspannung für die I/O-Ports beziehungsweie des Mikrocontrollers |

# Der Reset-Taster

Über den *Reset-Taster* erfolgt ein Neustart des Mikrocontrollers. Der geladene Sketch wird nicht gelöscht, sondern lediglich neu gestartet.

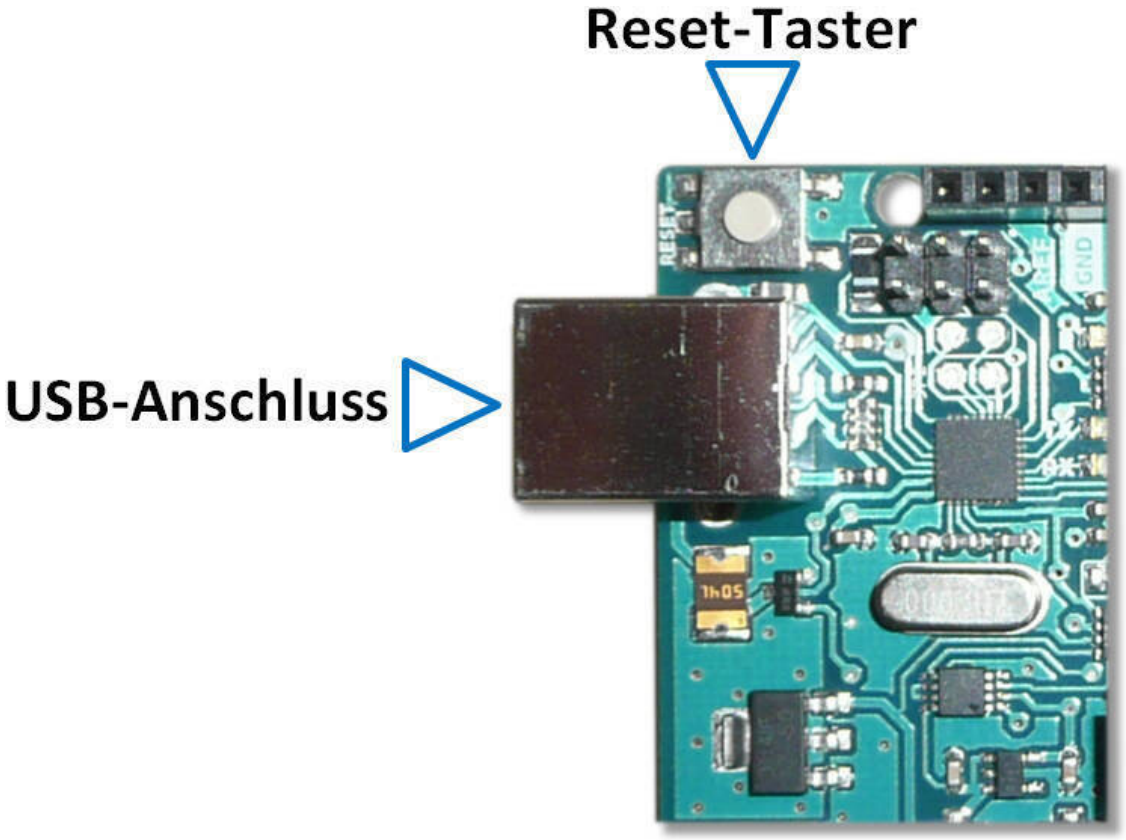



**Abb. 17:** Der Reset-Taster

# Die serielle Schnittstelle

Eine weitere Möglichkeit der Kommunikation mit dem Arduino-Board ist die über die serielle Schnittstelle. Diese Schnittstelle wird ja über den USB-Port zur Verfügung gestellt und kann über jedes Terminal-Programm wie beispielsweise PuTTY abgefragt werden. Doch eine Zusatzsoftware ist in unserem Fall nicht erforderlich, denn die Arduino-Entwicklungsumgebung stellt den sogenannten *Serial-Monitor* bereit, der sehr nützlich ist. Mit ihm können zur Laufzeit eines Sketches Werte angezeigt werden, was einerseits sehr gut zur Überwachung bestimmter Sensorwerte ist und andererseits bei einer eventuellen Fehlersuche sehr hilfreich sein kann. Doch auch weiterreichende Funktionen wie beispielsweise die Bereitstellung von Messwerten, die von anderen Programmen empfangen und optisch aufbereitet werden, sind über die serielle Schnittstelle problemlos zu realisieren. In diesem Zusammenhang ist die Programmiersprache *Processing* sehr geeignet.

# Details zur seriellen Schnittstelle und zum USB-Port

Vielleicht sind beim Erwähnen der seriellen Schnittstelle und der Kommunikation darüber ein paar Stirnrunzler aufgetreten. Wir haben den Arduino doch lediglich über den USB-Anschluss mit dem Rechner verbunden und sollen jetzt über die serielle Schnittstelle mit ihm in Kontakt treten. Wie funktioniert das denn? Nun, die Sache hat folgenden historischen Hintergrund: Der erste Arduino wurde über die serielle Schnittstelle RS232 mit dem Rechner verbunden, denn ein USB-Anschluss war nicht vorhanden. Nachfolgende Modelle bekamen dann einen sogenannten FTDI-Chip (Future Technology Devices International), der es ermöglichte, eine serielle Schnittstelle über USB verfügbar zu machen. Somit wird nach erfolgreicher Treiberinstallation ein zusätzlicher COM-Port für den Arduino angeboten. Der Arduino Uno hat nun anstelle eines FTDI-Chips (FT232RL) einen zusätzlichen Mikrocontroller mit der Bezeichnung *ATmega8u2* erhalten. Der Vorteil dieses Chips, der natürlich frei programmierbar ist, ist die universelle Einsatzbarkeit als USB-Device wie zum Beispiel einer Tastatur oder Maus. Viele weitere Informationen zu diesem Thema und anderen interessanten Aspekten sind unter der folgenden Adresse zu finden:

https://learn.adafruit.com/arduino-tips-tricks-and-techniques/arduino-uno-faq

# Die unterschiedlichen Speicher

Eingangs habe ich unterschiedliche Speicher genannt, die da lauten:

Flash
SRAM
EEPROM

Es ist wichtig zu wissen, worin die Unterschiede bestehen und für welche Anwendungsgebiete diese Speicher zum Einsatz kommen. Weiterführende Informationen sind unter der folgenden Adresse zu finden:

https://www.arduino.cc/en/Tutorial/Memory

# Der Flash-Speicher

Ein Programm, das im Arduino-Umfeld *Sketch* genannt wird, muss irgendwo innerhalb des Mikrocontrollers abgelegt beziehungsweise gespeichert werden. Ein Sketch teilt dem Mikrocontroller mit, was er zu tun hat und welche Aufgaben zu erledigen sind. Ein Programmierprojekt wird in einzelne Programmschritte (Befehle beziehungsweise Kommandos) unterteilt, die in einer bestimmten Reihenfolge abgearbeitet werden. Der Flash-Speicher übernimmt diese Aufgabe der Ablage. Wird der Arduino von der Spannungsversorgung getrennt und ist somit stromlos, dann bleibt der Sketch, der auf den Mikrocontroller übertragen wurde, resistent im Speicher. Nach erneuter Verbindung mit der Spannungsversorgung stehen diese Informationen wieder zur Verfügung.

# Das SRAM

Die Abkürzung von SRAM lautet *Static Random Access Memory*. Wird ein Sketch zum Beispiel zur Verarbeitung von Messwerten benötigt, dann müssen diese Werte in irgendeiner Form innerhalb des Mikrocontrollers abgelegt werden. Dazu nutzt man sogenannte *Variablen*, die als Platzhalter fungieren. Es handelt sich um spezielle Speicherbereiche für den Datenaustausch und zur Datenmanipulation. Diese Speicherbereiche sind jedoch flüchtig, was bedeutet, dass sie ihre Informationen nach dem Abschalten der Spannungsversorgung verlieren. Das ist jedoch kein Problem, denn ein Sketch nutzt diese Daten nur zur Laufzeit und sie werden nach dem erneuten Starten wieder hergestellt. Da der Speicherbereich je nach Modell sehr begrenzt ist, ist es möglich, Daten statt im SRAM im Flash-Speicher abzulegen. Nähere Informationen hierzu sind unter der folgenden Adresse zu finden:

https://www.arduino.cc/en/Reference/PROGMEM

# Das EEPROM

Die Abkürzung EEPROM steht für *Electrically Erasable Programmable Read-Only Memory*. Es handelt sich wie beim Flash-Speicher um einen nichtflüchtigen Speicher, der einmal gespeicherte Daten auch nach dem Verlust der Versorgungsspannung behält. Er kann dazu genutzt werden, wichtige Daten wie zum Beispiel Messwerte permanent zu speichern. Es ist jedoch zu beachten, dass die Schreibzugriffe auf diesen Speicherbereich begrenzt sind. Die maximalen Schreib- und Löschvorgänge sind offiziell mit einem Wert von 100.000 angegeben. Dieser Speicherbereich sollte also nicht für recht kurze zyklische Messvorgänge zur Speicherung von Messdaten verwendet werden.

Das war jetzt eine geballte Ladung Informationen, die ich dir über die Arduino-Hardware gegeben habe. Du solltest jetzt in etwa wissen, wo sich welche Bauteile auf dem Arduino Uno befinden. Du hast Informationen über das Herzstück des Arduino-Boards bekommen, dem ATmega328-Mikrocontroller. Wie das Board mit Strom versorgt wird, hast du ebenfalls gelernt, außerdem was die digitalen und anologen Ein- und Ausgänge sind. Du hast etwas über die Speicher gelesen, die beim Arduino genutzt werden. Und obendrauf hast du gelernt, was Strom und Spannung eigentlich ist und wie man das mithilfe von mathematischen Formeln darstellen kann.

Im folgenden Kapitel 2 zeige ich, wie du das Arduino-Board mit seiner Software-Entwicklungsumgebung steuern kannst.

# Kapitel 2:
# Arduino: Die Software

Wenn ich von Arduino spreche, dann meine ich damit sowohl das Arduino-Board als auch die Arduino-Software, die man braucht, um den Mikrocontroller steuern zu können. In diesem Kapitel wirst du erfahren, wie du die notwendige Software auf deinem Rechnersystem installierst und worauf du alles achten musst, bis das erste Arduino-Programm korrekt läuft. Ich erkläre, wo du was in der Arduino-Entwicklungsumgebung findest. Außerdem erfährst du in diesem Kapitel auch, was im Hintergrund abläuft, wenn du ein Programm auf deinem Arduino ans Laufen gebracht hast. Außerdem sage ich dir, was an Grundausstattung zu deiner Bastelwerkstatt gehören sollte.

Grundsätzlich gibt es zwei Möglichkeiten, dein Arduino-Board anzusprechen: Entweder nutzt du dafür ein Softwareprogramm, das du auf deinem Computer installierst, es ist die Arduino-Entwicklungsumgebung. Oder du verwendest Arduino Create, das ist ein webbasierter Editor, den du über deinen bevorzugten Browser aufrufen kannst.

Alle notwendigen Schritte zur Installation sowohl der lokal installierbaren als auch der Onlinevariante sind im Download-Verzeichnis meiner Interseite zu finden:

die Installation der Arduino-Entwicklungsumgebung in der aktuellen Version
die Einrichtung zur Verwendung von Arduino Create in der aktuellen Version

https://erik-bartmann.de/?Downloads___Arduino

Hinsichtlich des zu installierenden Plug-ins bei der Verwendung des webbasierten Arduino Create handelt es sich nicht um ein Browser-Plug-in, sondern um einen Agenten, der im Hintergrund arbeitet und quasi browserunabhängig seinen Dienst verrichtet. Jeder andere Browser nutzt diesen Agenten und sollte dann darüber eine Verbindung zur Cloud beziehungsweise zum Arduino herstellen können. Dieser Agent wird übrigens nach jedem Neustart des Systems gestartet. Du kannst die Informationen über den Task-Manager einholen, in dem du in den Tab-Reiter *Autostart* wechselst. Dort ist ein Eintrag mit dem Namen *Arduino_Create_Bridge* zu sehen:

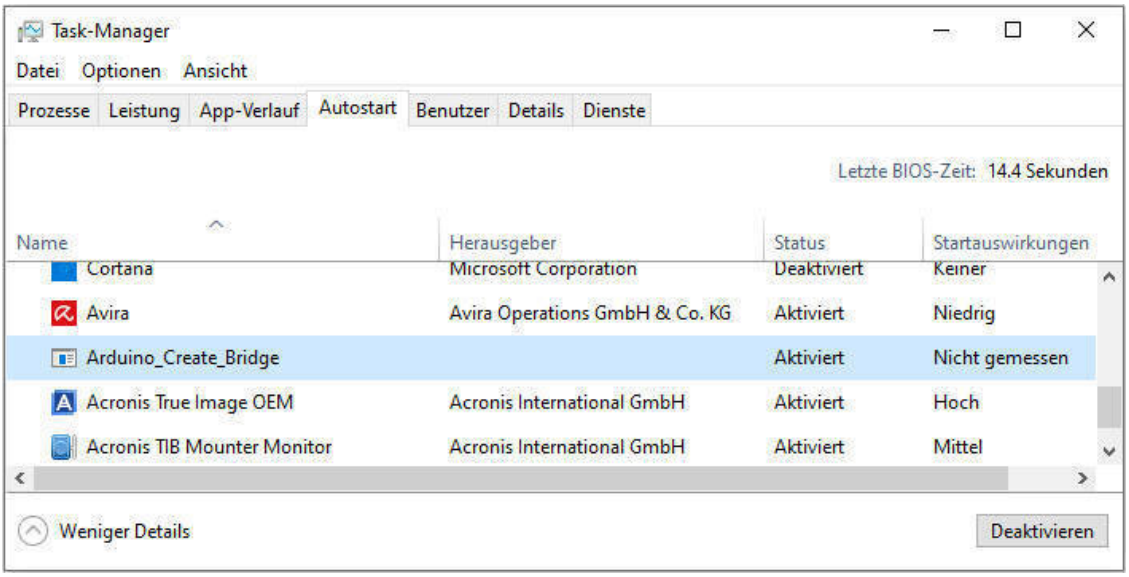


**Abb. 1:** Die Konfiguration der Startprogramme unter Windows

Falls du nicht möchtest, dass dieser Agent mit Windows gestartet werden soll, setze den Status auf *Deaktiviert*.

# Arduino-IDE oder Arduino Create?

Wenn du dich für die lokale Installation der Arduino-Entwicklungsumgebung entschieden hast, dann befindet sich dein gesamter Code, inklusive der Bibliotheken, auf deinem Rechner. Du hast also alles vor Ort und kannst direkt darauf zugreifen, ohne dass eine Internetverbindung erforderlich ist. Falls es neue Bibliotheken – das sind bereits erstellte Arduino-Programme – oder neue Versionen von bestehenden Bibliotheken gibt, liegt es in deiner Verantwortung, diese auf den neuesten Stand zu bringen oder sie zu installieren. Du erhältst jedoch Informationen über eventuell vorliegende Updates.

Im Gegensatz dazu bietet die Arduino-Create-Plattform die Speicherung deines Codes in der sogenannten *Arduino IOT Cloud*, was einen Speicher darstellt, der sich irgendwo in den Weiten des Internets befindet. Aus diesem Grund ist auch eine ständige Internetverbindung erforderlich, da alles über einen webbasierten Zugriff des Browsers abgewickelt wird. Der Vorteil dieser Variante ist natürlich der weltweite Zugriff auf alle Arduino-Ressourcen, also Code und Bibliotheken. Angebotene Bibliotheken werden auf diese Weise zentral auf einem aktuellen Stand gehalten und es ist nicht erforderlich, sich um eine Aktualisierung zu kümmern. Ich schlage vor, dass du beide Varianten testest und dich für die Version entscheidest, die dir am meisten zusagt. Sowohl mit der Arduino-IDE als auch mit Arduino Create ist ein praktisches Arbeiten möglich. Ich verwende in diesem Buch die Arduino-IDE, weil sie mir persönlich besser gefällt und ich nicht gezwungen sein möchte, immer auf eine bestehende Internetverbindung zurückgreifen zu müssen, obwohl das heutzutage ja fast immer gewährleistet ist. Ich bin kein Freund von Cloud-Lösungen, doch das musst du für dich selbst entscheiden.

# Was befindet sich wo?

Nach dem Starten der Arduino-Entwicklungsumgebung, die du auf deinem Computer installiert hast, öffnet sich ein Fenster, das aussieht wie in Abbildung 2. Die Arduino-Entwicklungsumgebung ist in einzelne Bereiche unterteilt. Es gibt einen Ein- und einen Ausgabebereich, Steuerelemente, Informationen über die Verbindungsparameter und einiges mehr. Schauen wir uns das im Detail an.

# Der Editor

Der Bereich, in dem du deinen Programmiercode eintippst, wird *Editor* genannt. Er beherrscht das sogenannte *Syntax-Highlighting*, was besagt, dass erkannte Schlüsselwörter der Programmiersprache meist farbig dargestellt werden.

## Statusinformationen

Unterhalb des Editors befindet sich ein schmaler Bereich, der Statusinformationen anzeigt. Das können Hinweise darüber sein, was gerade passiert oder Meldungen der letzten ausgeführten Aktion. Du kannst in Abbildung 2 erkennen, dass ich gerade einen Sketch erfolgreich auf das Arduino-Board hochgeladen hatte.

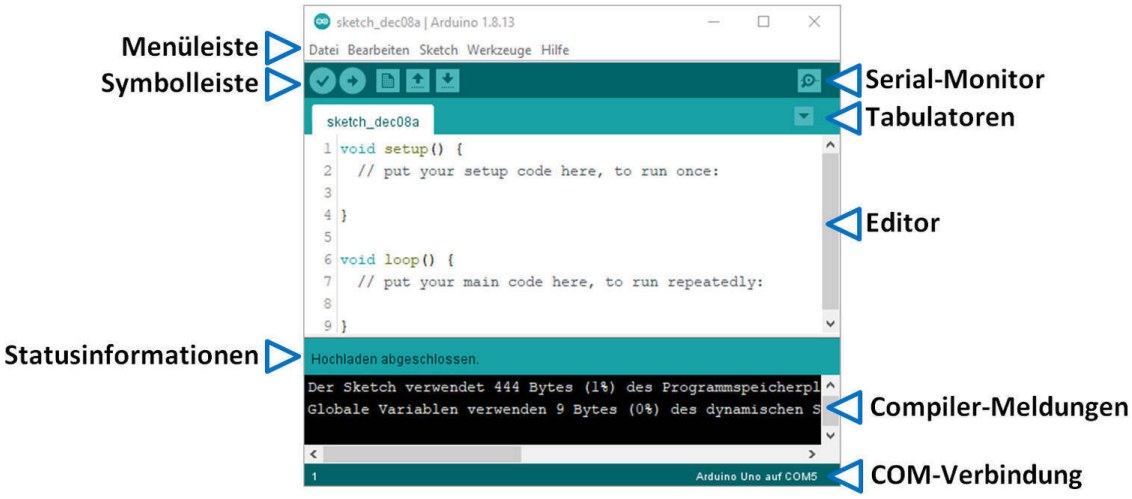
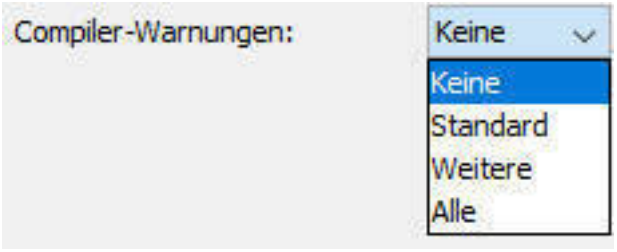



**Abb. 2:** Die Bereiche der Arduino-Entwicklungsumgebung

# Compiler-Meldungen

Wenn der Compiler, also das interne Arduino-Programm, das deinen Programmcode in eine Maschinensprache übersetzt, die der Mikrocontroller versteht, seine Arbeit zur Übersetzung des Quellcodes verrichtet, erscheinen im Bereich, der schwarz hinterlegt ist, die Meldungen über den Fortschritt oder aufgetretene Probleme und abschließend auch Hinweise über den Speicherverbrauch. Damit der Compiler etwas gesprächiger wird, kann über den Menüpunkt *Datei* | *Voreinstellungen* der Redefluss angepasst werden.




Standardmäßig ist hier der Punkt *Keine* ausgewählt.

# COM-Verbindung

Am unteren Rand der Entwicklungsumgebung sehen wir die Verbindungsinformationen, also welches Board sich an welchem COM-Port befindet. Das kann bei Verbindungsproblemen und zur Fehlersuche sehr hilfreich sein.

# Tabulatoren

Im Bereich der Tabulatoren befindet sich die Umschaltmöglichkeit zwischen Quellcodes desselben Sketches. Das wird dann interessant, wenn wir später eigene Bibliotheken und Klassen programmieren. Es kann jedoch immer nur ein einzelner Sketch je Entwicklungsumgebung verarbeitet werden. Mehrere unterschiedliche Sketches können nicht in einer einzigen Instanz der Entwicklungsumgebung Platz finden. Das Prinzip gleicht nicht den Tabulatoren innerhalb eines Browsers, der darüber mehrere unterschiedliche und eigenständige Verbindungen verwaltet.

# Serial-Monitor

Über das Lupensymbol (es sieht jedenfalls so aus) kann der Serial-Monitor geöffnet werden, der auf die serielle Schnittstelle zugreift, um dann Informationen anzuzeigen oder auch zu versenden.

# Menüleiste

Über die Menüleiste können die unterschiedlichsten Anpassungen zur Verwaltung der Anwendung vorgenommen werden, wie man das auch von anderen Programmen kennt.

# Symbolleiste

In der Symbolleiste befinden sich einige wichtige Schaltflächen, die mit Aktionen hinterlegt sind, die immer wieder benötigt werden, beispielsweise das Übersetzen (Kompilieren) des Quellcodes, das Hochladen des übersetzten Codes zum Mikrocontroller oder das Laden und Speichern von Sketches.

# Der Anschluss des Arduino-Boards

Sehen wir uns nun den Anschluss des Arduino-Boards mit dem Computer genauer an. Der Arduino Uno besitzt auf dem Board den folgenden USB-Anschluss:

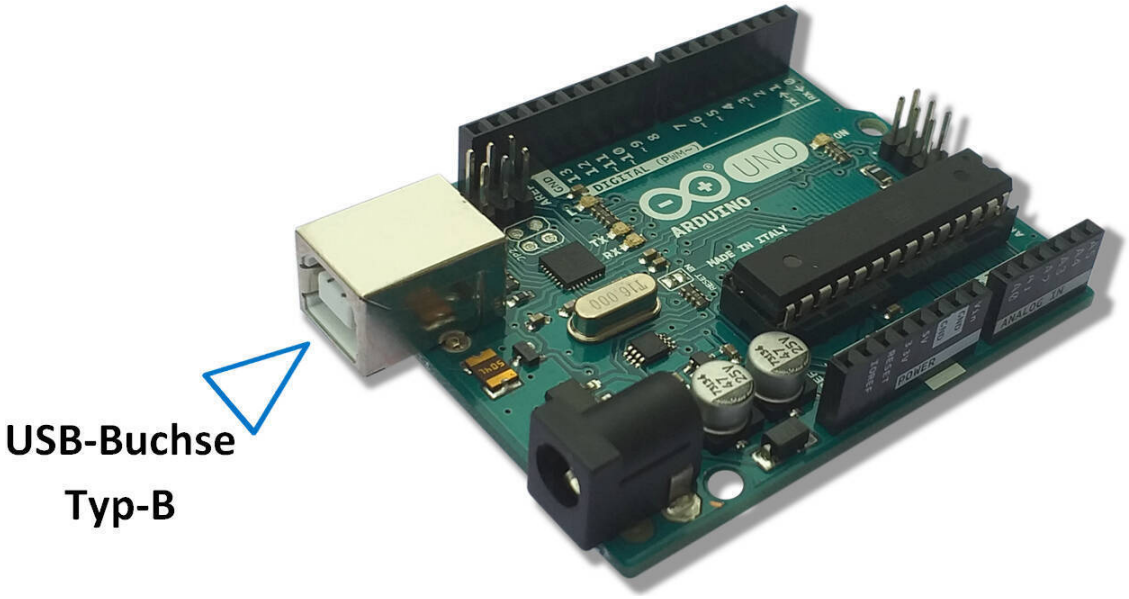



**Abb. 3:** Die USB-Buchse des Arduino

Auf Computerseite wird ein Standardanschluss *Typ-A* benötigt. Ein entsprechendes Kabel ist überall erhältlich, ich habe davon ganze Kisten voll. Ist die Verbindung hergestellt worden, so übernimmt dieser USB-Anschluss zwei Aufgaben:

Es wird das Arduino-Board mit Spannung versorgt.
Es ermöglicht die Kommunikation zwischen Computer und Board.

Detaillierte Informationen über die verschiedenen USB-Anschlüsse sind unter der folgenden Internetadresse zu finden:

http://de.wikipedia.org/wiki/Universal_Serial_Bus

Wird ein fabrikneues Arduino-Board mit Spannung versorgt, blinkt die auf dem Board befindliche LED (gekennzeichnet mit *L*) im Sekundentakt.

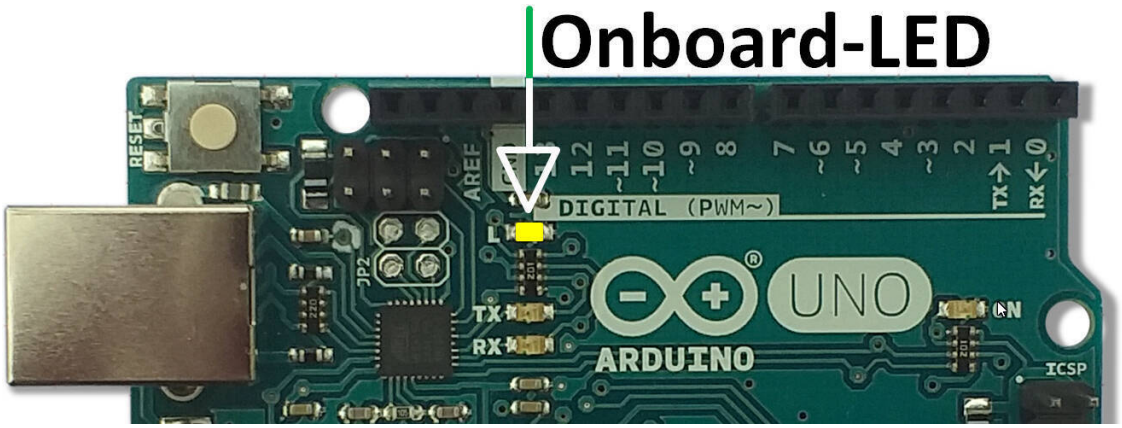



**Abb. 4:** Die Onboard-LED L

Das passiert, weil immer ein Basic-Sketch vorinstalliert ist. Wir können die erfolgreiche Kommunikation mit dem Arduino-Board mithilfe dieses Sketches überprüfen, denn die IDE verfügt von Haus aus über zahlreiche vorinstallierte Sketches.

# Wir testen die Kommunikation zwischen Computer und Arduino

Wenn das Arduino-Board von der IDE erkannt wurde, wollen wir im nächsten Schritt die Kommunikation zwischen Rechner und Arduino testen. Wenn ich von Kommunikation rede, dann meine ich im Moment die Übertragung eines Sketches (wir erinnern uns: Im Arduino-Umfeld wird ein Programm *Sketch* genannt) – zum Mikrocontroller-Board. Am besten laden wir uns den Blink-Sketch in die IDE und passen die Blinkrate etwas an. Dann kompilieren und übertragen wir den Sketch zum Arduino und sehen, was passiert. Hier die einzelnen Schritte, wobei ich voraussetze, dass die Installation erfolgreich war und der Arduino von der IDE nach deren Start erkannt wurde.

# Schritt 1: Beispiel-Sketch laden

Über den Menüpunkt *Datei* | *Beispiele* | *01.Basics* | *Blink* wird der Sketch in die IDE geladen und stellt sich wie folgt dar, wobei ich alle führenden Kommentarzeilen mit Zusatzinformationen weggelassen habe:

void setup() { pinMode(13, OUTPUT); } void loop() { digitalWrite(13, HIGH); // turn the LED on (HIGH is the voltage level) delay(1000); // wait for a second digitalWrite(13, LOW); // turn the LED off by making the voltage LOW delay(1000); // wait for a second }

Was die einzelnen Codezeilen im Detail bewirken, werden wir noch ausführlich im Bastelprojekt 1 besprechen. Die aktuelle Aufgabe ist jedoch die Ansteuerung einer auf dem Board befindlichen Leuchtdiode, die im Sekundentakt blinken soll. Jetzt wollen wir erst einmal sehen, ob der Sketch von der IDE zum Arduino-Board übertragen wird. Ich erwähnte schon, dass ein fabrikneues Board über einen fertig hochgeladenen Blink-Sketch verfügt und wir wollen diesen Sketch jetzt so anpassen, dass die LED mit der Bezeichnung *L* anfängt zu blinken. Dazu müssen die beiden Codezeilen, in denen sich der *delay*-Befehl befindet, leicht modifiziert werden.

# Schritt 2: Den Beispiel-Sketch modifizieren

Im Moment befindet sich ein Wert innerhalb des runden Klammerpaares, der festlegt, wie lange die Verarbeitung an dieser Stelle pausieren soll. Der Wert 1000 gibt die Wartezeit in Millisekunden (ms) an, wobei 1000 ms = 1 Sekunde ist. Demnach wird beim Erreichen des *delay*-Befehls für eine Sekunde gewartet, bevor es mit dem Befehl in der nächsten Zeile weitergeht. Wenn wir jetzt den Wert von 1000 auf 100 ändern, erreichen wir damit eine zehnmal höhere Blinkrate. Die beiden Zeilen, in denen der Befehl vorkommt, müssen wie folgt angepasst werden:

```
delay(100);
```

Nun kann der modifizierte Sketch – fast – auf den Arduino übertragen werden. Ich sage *fast*, weil es im Folgenden noch eine Kleinigkeit zu beachten gilt.

# Schritt 3: Das richtige Board und den richtigen COM-Port auswählen

Da von Arduino – und ich rede jetzt von der Firma und nicht von dem Board – etliche unterschiedliche Boards entwickelt wurden, müssen wir der Entwicklungsumgebung mitteilen, um welches Board es sich denn handelt, das wir verwenden. Des Weiteren ist die Angabe des korrekten COM-Ports wichtig. Diese beiden Unterschritte werden über die folgenden Menüpunkte erledigt:

Werkzeuge | Board und
Werkzeuge | Port

In meinem speziellen Fall also *Arduino Uno* und *COM5*.

# Schritt 4: Den Sketch kompilieren und hochladen

Kommen wir zu einem Schritt in der Programmierung, der sich *kompilieren* nennt. Doch eins nach dem anderen. Um ein Computerprogramm in einer für den Menschen mehr oder weniger lesbaren Form abzubilden, wurden die sogenannten Hochsprachen wie C/C++, Java und C# – um nur einige wenige zu nennen – entwickelt. Die Befehle aus dem vorliegenden Sketch sind allesamt auf Englisch, was für fast alle anderen Programmiersprachen ebenfalls gilt. Jeder einzelne Befehl beschreibt in mehr oder weniger deutlicher Weise, welche Funktion dahinter verborgen ist. Unser *delay*-Befehl bedeutet übersetzt *Verzögerung* oder *Aufschub* und darunter kann sich sicherlich jeder etwas vorstellen. Ein Mikrocontroller kann damit jedoch zunächst nicht das Geringste anfangen und deshalb müssen wir ihm auf die Sprünge helfen. Warum? Nun, ein Mikrocontroller und jede andere CPU kann lediglich Befehle in Form der ganz persönlichen Muttersprache, auch Maschinensprache oder *Native Language* genannt, verstehen und diese ist für jeden Mikrocontroller-Typen meist anders definiert. Somit wird eine Übersetzungsinstanz benötigt, die unsere Hochsprache, beim Arduino ist das C++, in eine für den Mikrocontroller verständliche Maschinensprache übersetzt. Diese Arbeit übernimmt ein sogenannter *Compiler*.

Was ist ein Compiler?

Ein Compiler ist ein Übersetzungsprogramm, das den Quellcode einer Hochsprache wie zum Beispiel C++ in eine Sprache übersetzt, die der jeweilige Prozessor beziehungsweise Mikrocontroller versteht.

Die Arduino-Entwicklungsumgebung besitzt – wie oben bereits erwähnt – mehrere kleine Symbole mit dahinter verborgenen Aktionen. In der folgenden Liste sind die beiden wichtigsten Symbole und deren Funktion zu sehen. Natürlich sind die anderen auch wichtig, doch diese hier sind eben essentiell wichtig:

Tabelle 1: Ein paar wichtige Icons für den Anfang

| Symbol | Funktion |
|---|---|
|  | Überprüfung des Quellcodes durch Kompilierung |
|  | Übertragung zum Mikrocontroller starten, wenn Kompilierung erfolgreich war |

Das erste Symbol überprüft den *Quellcode* – so nennt sich der Text im Editor einer Entwicklungsumgebung – auf Richtigkeit. Was bedeutet aber Richtigkeit? Nun, jede Sprache verfügt über eine gewisse Grammatik. Wird bei der Kommunikation nicht korrekt auf diese Regeln zurückgegriffen, versteht der andere Kommunikationsteilnehmer die Absicht des Sprechenden nicht. Ebenso verhält es sich bei der Programmierung. Werden die Befehle nicht genau so geschrieben beziehungsweise formuliert, wie der Compiler sie versteht, kommt es unweigerlich zu einem (Syntax-) Fehler und der Compiler bricht seinen Übersetzungsvorgang mit einer entsprechenden Fehlermeldung ab. In der folgenden Abbildung 5 ist der Ablauf nach Anklicken des *Überprüfen*-Symbols zu sehen:

Zu Beginn startet die Kompilierung mit der Anzeige des Fortschritts über den grünen Balken. Ist kein syntaktischer Fehler gefunden worden, wird dieser Vorgang mit der Meldung beendet,

dass die Kompilierung abgeschlossen ist. Ein Upload des Maschinencodes zum Mikrocontroller des Arduino Uno erfolgt dabei nicht. Wurde jedoch ein Fehler im Sketch-Code gefunden, so bricht der Vorgang des Kompilierens mit einer entsprechenden Fehlermeldung ab.

Das zweite Symbol analysiert ebenfalls den *Quellcode* und es werden die gleichen Prozeduren wie beim Überprüfen durchgeführt. Jedoch erfolgt hier nach einer fehlerfreien Kompilierung das Hochladen des Maschinencodes zum Mikrocontroller des Arduino Uno. Auf der nachfolgenden Abbildung 6 ist der Ablauf zu sehen:

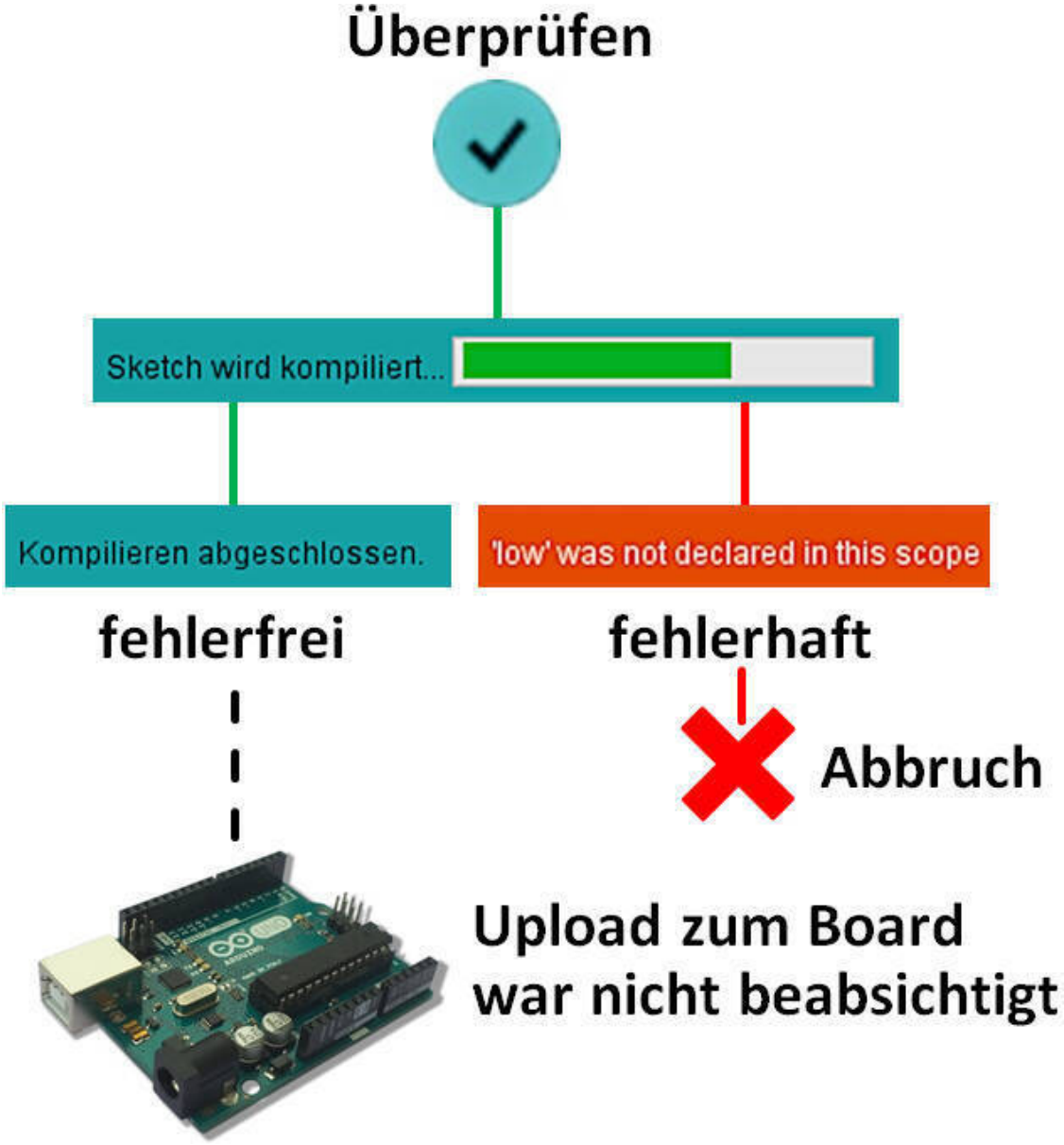



**Abb. 5:** Der Ablauf nach Anklicken des Überprüfen-Symbols

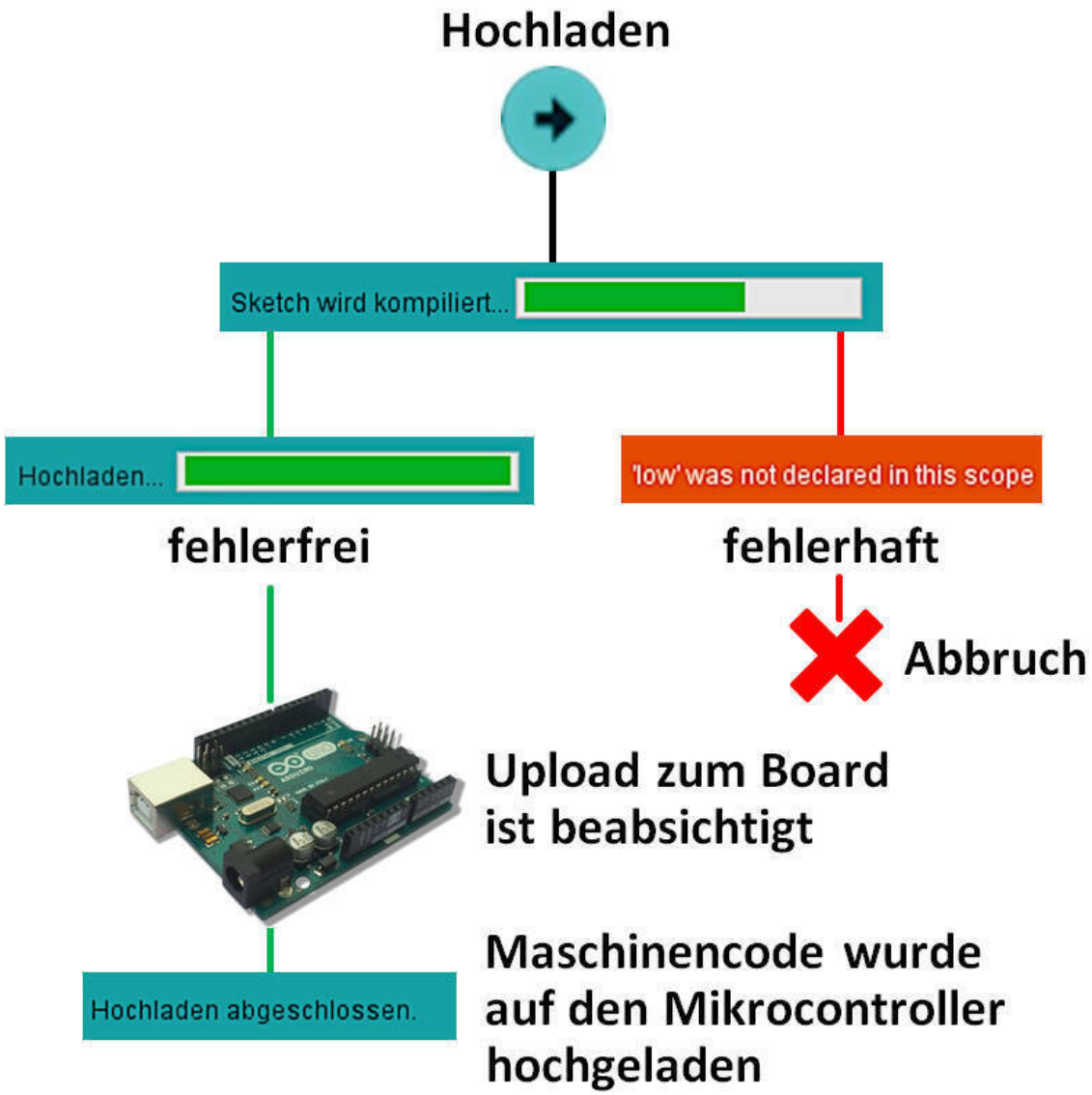



**Abb. 6:** Der Ablauf nach Anklicken des Hochladen-Symbols

Zu Beginn startet die Kompilierung mit der Anzeige des Fortschritts über den grünen Balken. Ist kein syntaktischer Fehler gefunden worden, wird der Prozess des Hochladens zum Mikrocontroller gestartet, der ebenfalls eine Fortschrittsanzeige auf einem grünen Balken bietet. Nach erfolgreichem Hochladen erscheint die Meldung, dass das Hochladen abgeschlossen wurde. Bei einem Fehler im Sketch-Code bricht der Vorgang des Kompilierens mit einer entsprechenden Fehlermeldung ab, wie das auch bei der Überprüfung der Fall war.

Es kann auch ohne Umweg über das vorherige Kompilieren sofort das Hochladen gewählt werden, weil eine vorherige Kompilierung dort ebenfalls ausgeführt wird. Während des *Uploads* – so wird das Hochladen im Englischen genannt – leuchten einige LEDs, die sich auf dem Arduino-Board befinden. Schauen wir genauer hin:

**Abb. 7:** Drei wichtige LEDs auf dem Arduino

Links neben dem Arduino-Schriftzug finden wir drei LEDs. Eine davon ist unsere LED mit der Bezeichnung *L*, die den Zustand am digitalen Pin 13 widerspiegelt. Etwas darunter sehen wir zwei weitere LEDs mit den Bezeichnungen *TX* (Senden) und *RX* (Empfangen). Es handelt sich um die Statusanzeigen der seriellen Schnittstelle, über die das Arduino-Board mit dem Computer verbunden ist. Und beim gerade angesprochenen Upload werden die Maschinensprachinformationen über diese Schnittstelle versendet. Das Board quittiert diesen Vorgang mit einem unregelmäßigen Blinken der beiden genannten LEDs. Wurde der Upload beendet – aus welchen Gründen auch immer – erlöschen die beiden LEDs wieder. Somit haben wir eine sehr gute optische Kontrolle über den Upload-Vorgang. Nach dem erfolgreichen Upload blinkt die LED *L* in kurzen Zeitabständen und wir können sicher sein, dass unser modifizierter Sketch funktioniert hat. Auf der rechten Seite der Abbildung sehen wir eine weitere LED mit der Bezeichnung *ON*. Sie leuchtet, wenn das Board mit Spannung versorgt wurde.

Der Maschinencode des Arduino

Der folgende Abschnitt hat zwar nicht unmittelbar etwas mit der Programmierung des Arduino-Boards zu tun, aber es ist wichtig, dass du die Schritte verstehst, die im Hintergrund ablaufen. Ich habe ja schon ein wenig über die Entwicklungsumgebung, den Compiler und die Programmiersprachen C/C++ erzählt. Wie läuft die Kompilierung ab und was wird eigentlich zum Mikrocontroller des Arduino-Boards übertragen?

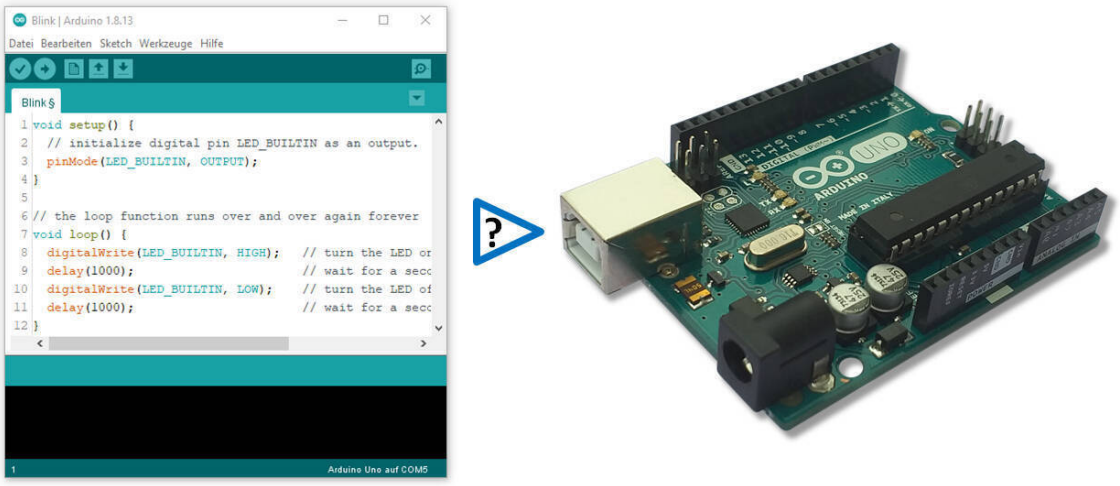



**Abb. 8:** Was passiert im Hintergrund bei der Übertragung des Sketches zum Arduino-Board?

Wir können den Ablauf in einzelne logische Schritte unterteilen:

# Schritt 1

Es findet die Überprüfung des Sketch-Codes durch die Entwicklungsumgebung statt, um sicherzustellen, dass die C/C++-Syntax korrekt ist.

## Schritt 2

Danach wird der Code zum Compiler (der Compiler heißt übrigens AVR-GCC) geschickt, der daraus eine für den Mikrocontroller lesbare Sprache, die *Maschinensprache*, erstellt.

# Schritt 3

Im Anschluss wird der Code mit einigen Arduino-Bibliotheken, die grundlegende Funktionalitäten bereitstellen, zusammengeführt und als Ergebnis eine *Intel-HEX* Datei erstellt. Es handelt sich dabei um eine Textdatei, die binäre Informationen für Mikrocontroller speichert. Hier zeige ich einen kurzen Ausschnitt aus dem ersten Sketch:

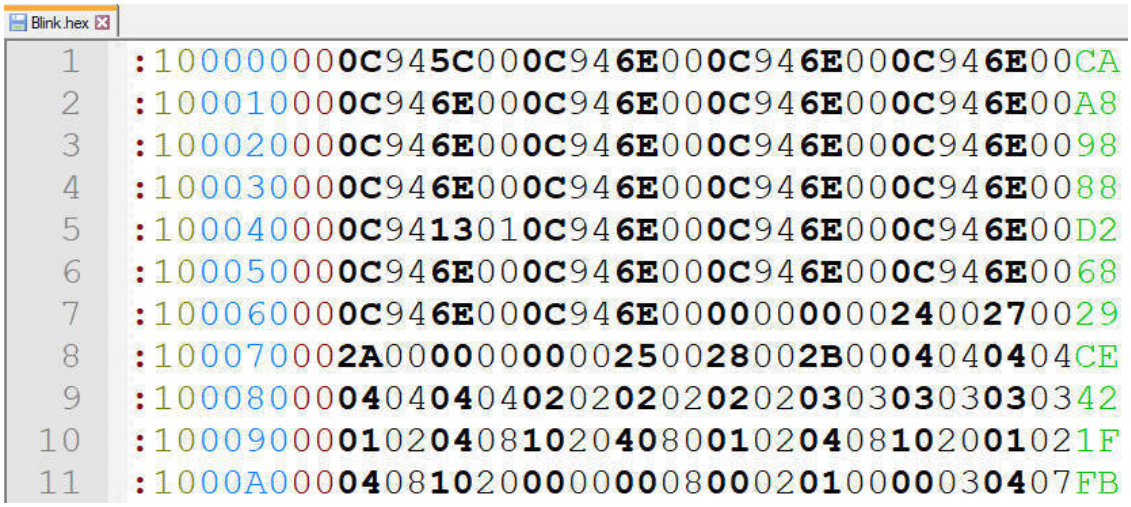



**Abb. 9:** Ausschnitt aus einer Intel-HEX Datei

Dieses Format versteht der Mikrocontroller, denn es ist seine *Native Language* oder *Muttersprache*.

# Schritt 4

Der Bootloader überträgt die Intel-HEX-Datei über USB in den Flash-Speicher des Mikrocontroller-Boards. Der sogenannte *Upload-Prozess*, also die Übertragung zum Board, erfolgt mit dem Programm *avrdude*. Es ist Bestandteil der Arduino-Installation und befindet sich unter *Arduino\hardware\tools\avr\bin*.

# Die Bibliothekenverwaltung

Die Entwicklungsumgebung stellt eine große Anzahl von Bibliotheken zur Verfügung, die bei der Installation auf deinen Computer mit installiert werden. Sie können für die unterschiedlichsten Anwendungszwecke genutzt werden. In regelmäßigen Abständen werden Verbesserungen, sogenannte *Aktualisierungen*, zur Verfügung gestellt. Wenn die Entwicklungsumgebung derartige Updates erkennt, erfolgt ein Hinweis:

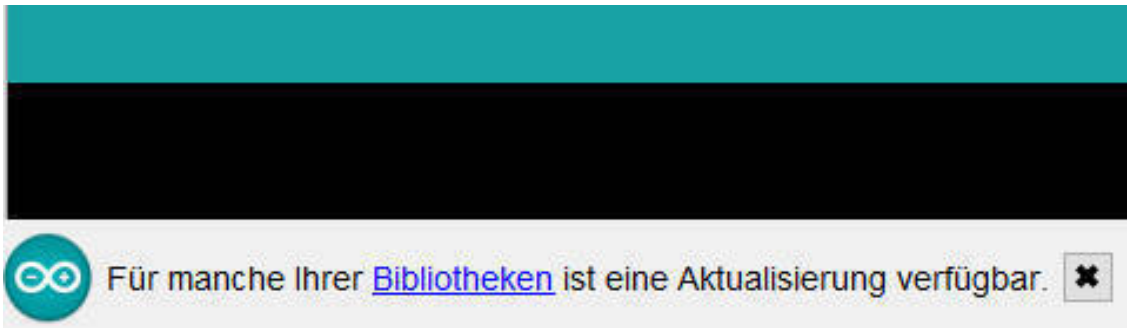



**Abb. 10:** Ein Hinweis über vorhandene Bibliotheken-Updates

Nach einem Mausklick auf den angezeigten Link *Bibliotheken* öffnet sich ein neues Fenster, in dem die neuen Updates zur Auswahl angezeigt werden und die Liste der installierten Bibliotheken wird aktualisiert, was am unten angezeigten Fortschrittsbalken zu sehen ist:

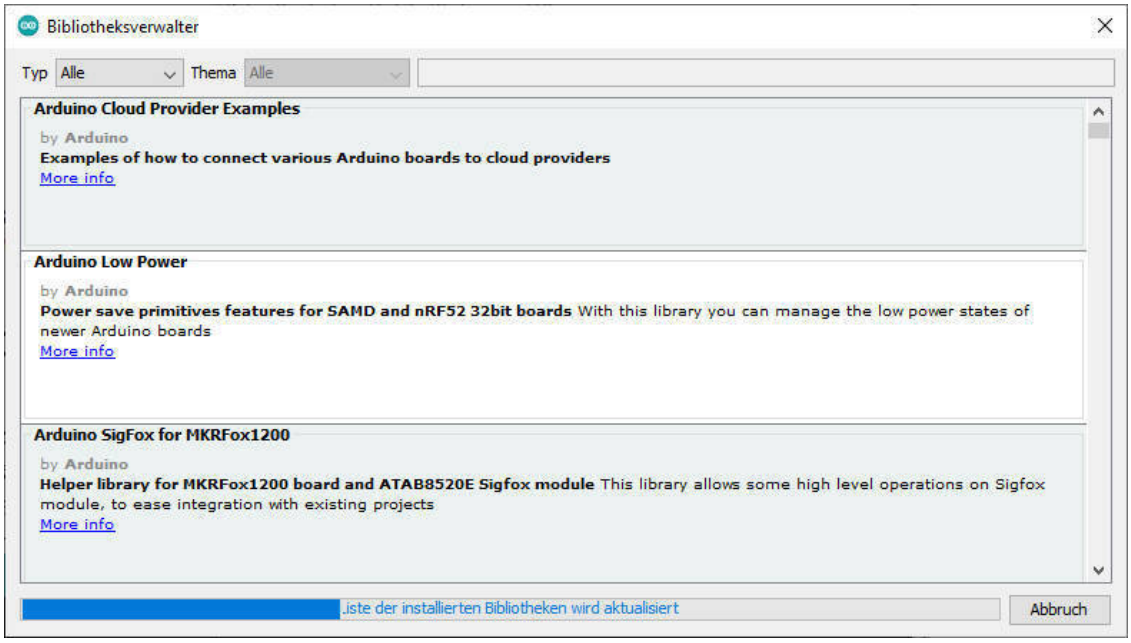



**Abb. 11:** Eine Liste der zur Verfügung stehenden Updates

Um eine neue Bibliothek zu installieren, wird der entsprechende Eintrag mit der Maus überfahren und es erscheinen in der Regel zwei Hinweise. Zum einen werden die verfügbaren Versionen zur späteren Installation als Liste angeboten und zum anderen wird die *Installieren*-Schaltfläche eingeblendet:

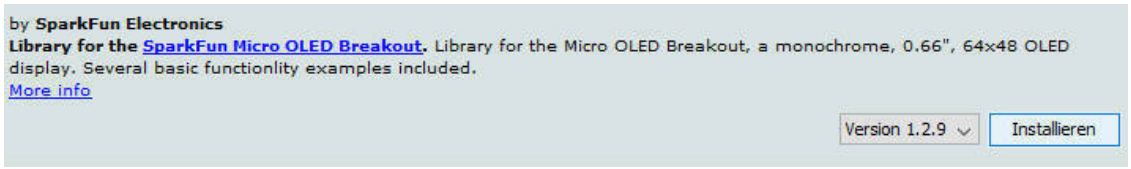



**Abb. 12:** Die Installation einer neuen Bibliothek

Ist eine Bibliothek schon installiert, ist dies am grünen Schriftzug *INSTALLED* zu erkennen und ist dafür ein Update verfügbar, erscheint nach dem Überfahren mit der Maus die *Update*-Schaltfläche:


**Abb. 13:** Die Update-Schaltfläche erscheint

Die Bibliothekenverwaltung kann aber auch über den Menüpunkt *Werkzeuge* | *Bibliotheken verwalten...* erreicht werden, sodass sich das schon gezeigte Fenster öffnet.

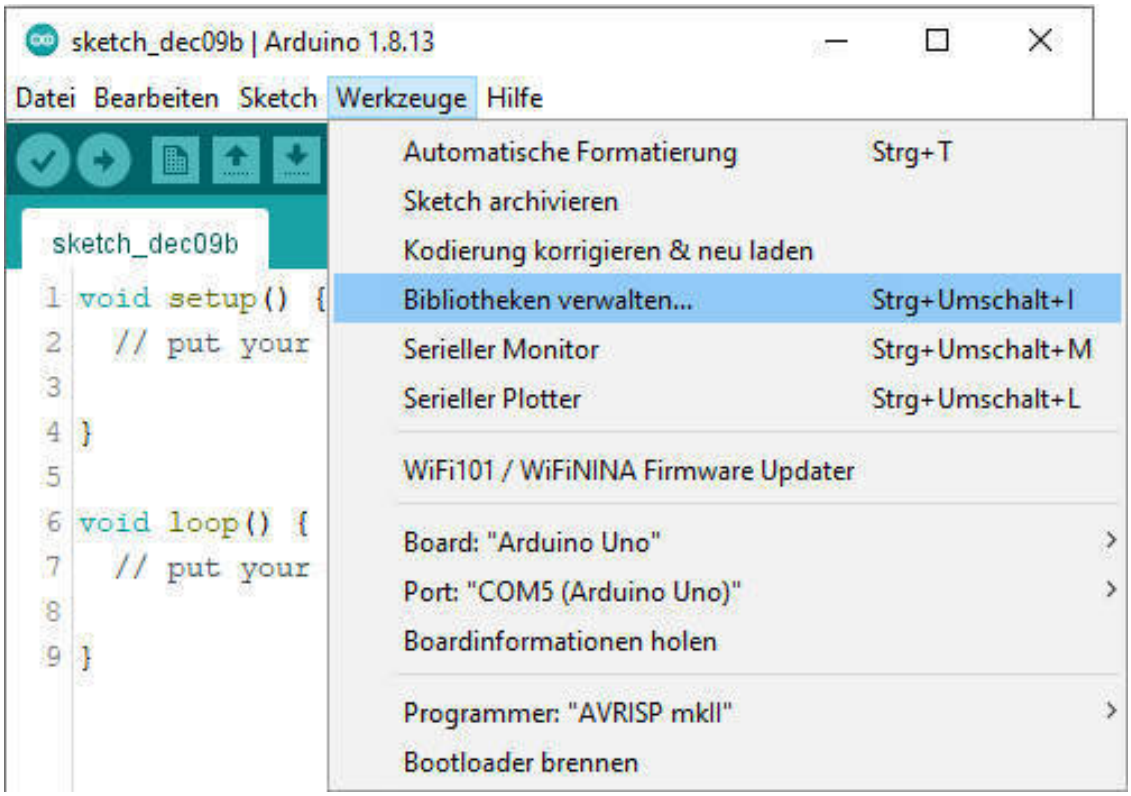



**Abb. 14:** Der Aufruf der Bibliothekenverwaltung

# Die Boardverwaltung

Die Arduino-Entwicklungsumgebung kann nicht nur Arduino-Boards programmieren und kontrollieren, sondern auch Boards von anderen Herstellern, wie zum Beispiel die sehr bekannten und beliebten ESP32 oder ESP8266. Um derartige Boards in die Entwicklungsumgebung zu integrieren, muss der Boardverwalter bemüht werden, der unter dem Menüpunkt *Werkzeuge* | *Board* | *Boardverwalter...* zu erreichen ist:

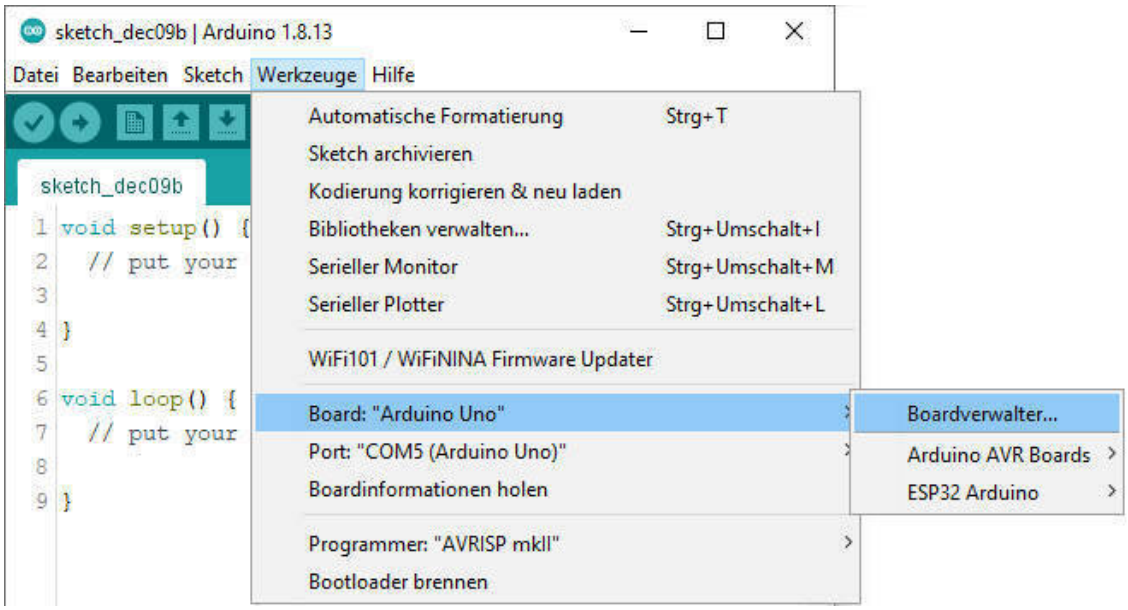



**Abb. 15:** Der Aufruf der Boardverwaltung

Möchte ich zum Beispiel die Unterstützung für das *megaAVR*-Board meiner Arduino-Entwicklungsumgebung hinzufügen, tippe ich rechts oben den Suchbegriff *megaAVR* ein und das entsprechende Installationspaket wird angezeigt. Es kann dann installiert werden:


**Abb. 16:** Das Hinzufügen eines neuen Boards

Diese Suche kann natürlich auch bei der Bibliothekenverwaltung angewendet werden. Vielleicht bist du am ESP32 und ESP8266 interessiert und hast ESP32 in das Suchfenster eingegeben. Doch leider liefert dieser Suchbegriff kein Ergebnis und die Liste bleibt leer. Keine Panik! Das hat folgenden Hintergrund: Zahlreiche Boards von Drittherstellern liefern Informationen und Installationsdateien auf einem hauseigenen Verzeichnis. Doch wie kannst du der Arduino-Entwicklungsumgebung diese Pfade mitteilen? Ganz einfach! Ruf über den Menüpunkt *Datei* | *Voreinstellungen* das entsprechende Dialogfenster auf. Das sieht dann wie folgt aus, wobei ich den benötigten Pfad zur ESP32-Unterstützung schon eingetragen und markiert habe:

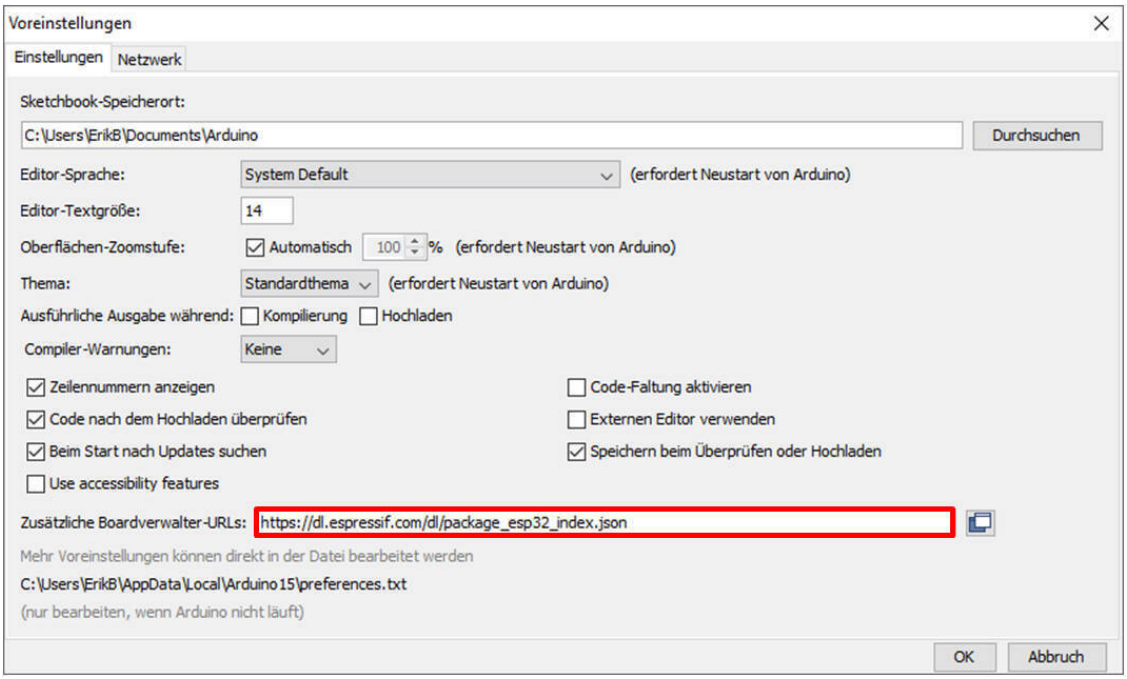



**Abb. 17:** Die Arduino-Voreinstellungen

Der dort einzusetzende Pfad lautet für die ESP32-Unterstützung:

https://dl.espressif.com/dl/package_esp32_index.json

Für eine kombinierte beziehungsweise zusätzliche Unterstützung des ESP8266-Boards ist der folgende Eintrag dem ersten – getrennt durch ein Komma – hinzuzufügen:

http://arduino.esp8266.com/stable/package_esp8266com_index.json

Im Anschluss sind beide Boards zur Installation in die Arduino-Entwicklungsumgebung bereit. Es existiert eine Quelle an inoffiziellen Erweiterungen für sogenannte 3rd-Party-Boards, die unter der folgenden Adresse zu finden ist:

https://github.com/arduino/Arduino/wiki/Unofficial-list-of-3rd-party-boards-support-urls

Dort sind sehr viele Boards aufgeführt, die mit der Arduino-IDE programmiert werden können. Kopiere die auf dieser Seite angebotenen Pfade in die Arduino-Voreinstellungen in der bereits gezeigten Zeile ein und schon kann es losgehen.

# Der Sketch-Code in der Entwicklungsumgebung

Zwar kann ich hier keinen Grundkurs in C/C++-Programmierung geben, doch möchte ich ein paar grundlegende Dinge zu dieser Sprache sagen, die in der Arduino-Entwicklungsumgebung eingesetzt wird. Die Programmiersprache C/C++ ist *case-sensitive*, was bedeutet, dass zwischen Groß- und Kleinschreibung unterschieden wird. Es ist also genau darauf zu achten, wie die Befehle geschrieben werden. Sehen wir uns dazu den Befehl *digitalWrite* an. Er setzt sich aus zwei Wörtern (*digital* und *Write*) zusammen. In der Programmierung wird das Zusammensetzen von Wortbedeutungen mit der Großschreibung des ersten Buchstaben (bis auf das erste Wort) *camel-casing* genannt. Somit würde die folgende Codierung auf jeden Fall fehlschlagen:

digitalwrite

Derartige Fehler werden aber von der Arduino-Entwicklungsumgebung erkannt und optisch dargestellt. Ein Wort, das Bestandteil eines Befehlssatzes ist, wird in der Regel farblich hervorgehoben. Wird das Wort beziehungsweise der Befehl falsch geschrieben, erfolgt keine farbliche Abhebung vom Rest des Codes, der schwarz gehalten ist. Sehen wir uns das in der Entwicklungsumgebung an:

```
7  void loop() {
8    digitalWrite(13, HIGH); // Korrekte Schreibweise
9    digitalwrite(13, HIGH); // Falsche Schreibweise
10 }
```

Es ist zu erkennen, dass der Befehl in Zeile 8 korrekt geschrieben wurde und mit einer entsprechenden Zeichenfarbe versehen wurde. Aufgrund der falschen Schreibweise des Befehls in Zeile 9 wurde dieser nicht erkannt und demnach schwarz dargestellt.

# Troubleshooting

Es kann immer wieder hier und da zu Problemen kommen, das passiert mir auch gelegentlich. Aus meinen eigenen Erfahrungen weiß ich, dass es einige typische Fehlerquellen gibt.

# Ist das Board mit Spannung versorgt?

Aufgrund eines defekten USB-Kabels kann es sein, dass das Board nicht mit Spannung versorgt wird. Dann leuchtet die LED mit der Bezeichnung *ON* auf dem Board natürlich nicht. Das hat auch zur Folge, dass das Board vom System nicht erkannt und auch nicht als weiterer COM-Port angezeigt wird.

# Stimmt die Auswahl von Board und COM-Port?

Wurde nicht das korrekte Board oder der korrekte COM-Port ausgewählt, dauert das vermeintliche Hochladen schon sehr lange, bis dann letztendlich die folgende Fehlermeldung in roter Schrift im unteren Teil der Entwicklungsumgebung erscheint:


**Abb. 18:** Die Arduino-Fehlermeldung

Der Zugriff auf den zuvor eingestellten COM5-Port ist nicht zu erreichen. Für weiterreichende Fehler gibt es von Arduino eine Troubleshooting-Seite, die unter der folgenden Internetadresse zu finden ist:

http://arduino.cc/en/pmwiki.php?n=Guide/Troubleshooting#toc1

Der richtige COM-Port

Wenn es Probleme mit der Auswahl des richtigen COM-Ports geben sollte, kannst du dir sehr einfach behelfen. Schau dir die Liste der angebotenen COM-Ports über den Menüpunkt *Werkzeuge | Port* an. Entferne nun das USB-Kabel vom Arduino-Board und sieh nach, welcher Port nicht mehr in der Liste vorhanden ist beziehungsweise stelle die Verbindung wieder her und achte darauf, welcher Port hinzugekommen ist. Die Arduino-Entwicklungsumgebung zeigt alle zur Verfügung stehenden Ports fast unmittelbar nach der erfolgten Änderung an.

Das sollte in deiner Arduino-Bastelwerkstatt vorhanden sein

Kommen wir zur Hardware, die durchgehend in allen Projekten meines Buches Verwendung findet. Damit die elektronischen Bauteile in geeigneter und beständiger Form mit dem Arduino-Board verbunden werden können, schlage ich zwei Dinge vor. Da ist zum einen das Breadboard – auch Steckbrett genannt –, auf das die Bauteile gesteckt werden. Des Weiteren benötigen wir natürlich zahlreiche Verbindungskabel, die für die elektrischen Verbindungen zwischen den einzelnen Komponenten notwendig sind. Dafür verwenden wir jedoch keine starren Kabel, sondern flexible Steckbrücken, die ohne große Mühe sowohl auf dem Breadboard als auch auf dem Arduino-Board platziert werden können:

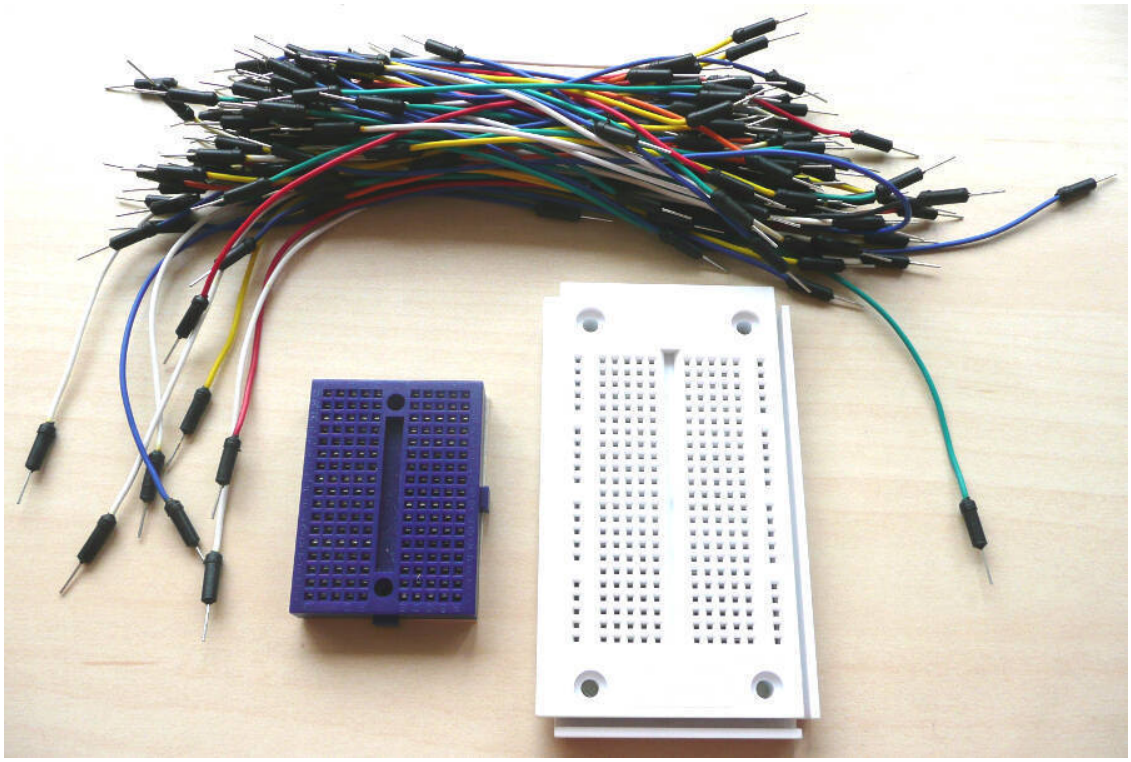


**Abb. 19:** Verschiedene Breadboards und flexible Steckbrücken

Auf Abbildung 19 sehen wir zwei handliche Breadboards, die es in unterschiedlichen Größen, Formen und Farben gibt. Ich empfehle, ein oder mehrere Breadboards in unterschiedlichen Größen zu kaufen, denn sie sind wirklich erschwinglich. Gib im Zweifelsfall lieber etwas mehr Geld für ein Breadboard aus, es lohnt sich. Die flexiblen Steckbrücken – auch Patchkabel genannt – kosten ebenfalls nicht die Welt. Es gibt auch sehr schöne Varianten mit der Kombination eines Arduino-Boards und einem Breadboard auf einer festen Unterlage als Kombination wie auf Abbildung 20. Dadurch sind das Arduino-Board und das Breadboard immer an einem Ort vorhanden, das stellt eine gewisse Erleichterung des Schaltungsaufbaus dar. Übrigens: Auf Abbildung 20 ist ein Arduino Uno zu sehen, der als Klon angeboten wird und um einiges günstiger ist als das originale Board. Klone sind Nachbauten von anderen Firmen. Und da der Arduino Uno unter einer Open-Source-Hardwarelizenz veröffentlicht wurde, steht es jedem frei, das Board nachzubauen und zu verändern. Es muss nicht immer original sein, denn Open Source ermöglicht auch derartige Boards.

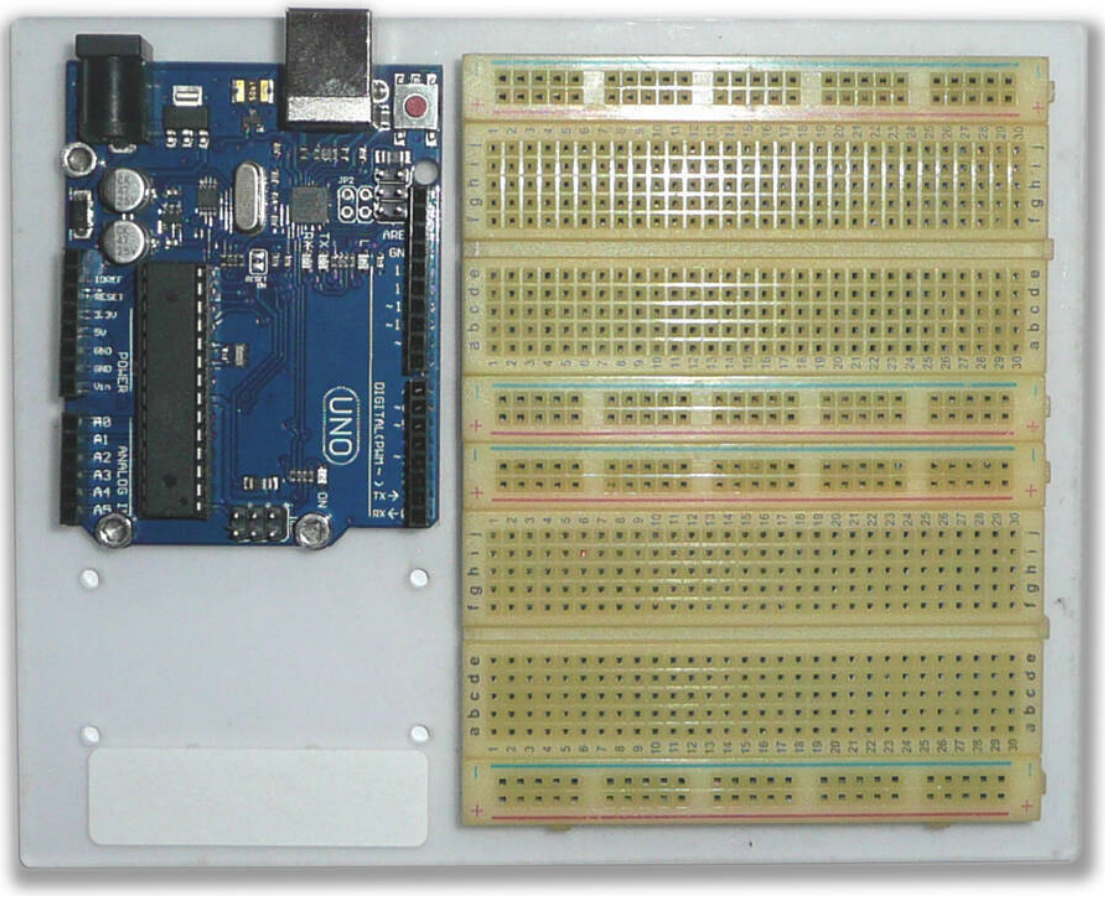



**Abb. 20:** Arduino-Board und Breadboard als Kombination

Eine sehr elegante Lösung stellt die Verwendung eines selbst hergestellten Boards dar, das ich *Arduino Discoveryboard* genannt habe. Ich stelle das Board ausführlicher in Kapitel 4 vor und zeige, wie du es selbst bauen kannst. Das Arduino Discoveryboard erspart dir viel Arbeit, die ansonsten für den Aufbau der Bastelprojekte drauf geht. Aber wie gesagt: Kannst du so machen, musst du aber nicht.

Zusätzlich ist ein Vielfachmessgerät – auch kurz Multimeter – sehr sinnvoll, denn es können damit Spannungen, Strom, Widerstände oder einfach nur elektrische Verbindungen gemessen und überprüft werden.

Es gibt natürlich noch weitere sehr interessante Messgeräte, auf die ich in den Bastelprojekten noch eingehen werde. Das bisher Gezeigte stellt ein *Muss* an Gerätschaften und Materialien dar, was wirklich nicht viel ist und was jeder Bastler immer parat haben sollte. Das Multimeter ist ein Instrument, das bei der Fehlersuche ein nicht mehr wegzudenkendes Hilfsmittel darstellt. Sei es beim Messen von Spannungen beziehungsweise Strömen, als Durchgangsprüfer oder zur Ermittlung eines Widerstandes: Das Multimeter sollte immer in Reichweite liegen.


**Abb. 21:** Ein Vielfachmessgerät – Multimeter

# Wie deine Idee in den Mikrocontroller kommt

Wir haben in diesem Kapitel gesehen, wie die Software für die Entwicklung eines Programms installiert wurde und wie ein sogenannter Sketch dann nach der Eingabe in die Arduino-Entwicklungsumgebung zum Mikrocontroller übertragen wird. Hinter dieser Prozedur steckt jedoch ein bisschen mehr, als man vielleicht auf den ersten Blick vermuten wird. Sehen wir uns die einzelnen Schritte im Detail an.

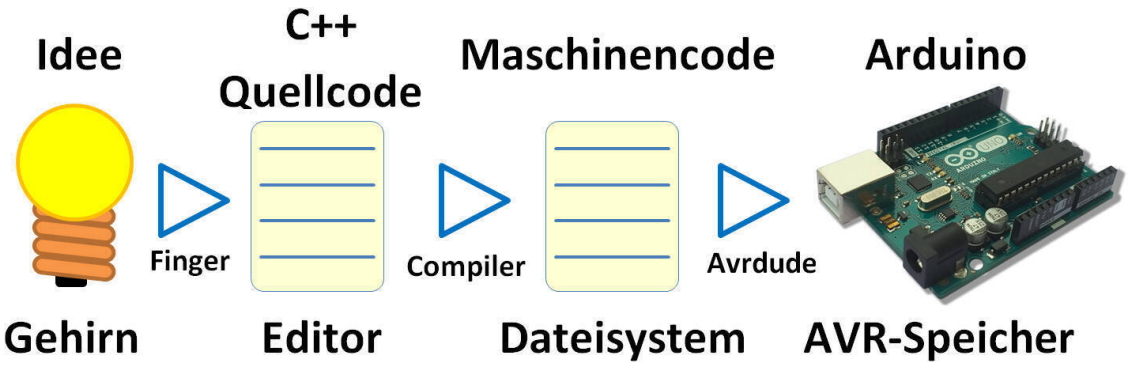



**Abb. 22:** Der Weg von der Idee zur Umsetzung

Zu Beginn steht natürlich immer eine Idee, die in deinem Gehirn entstanden und gereift ist. Natürlich kann der Mikrocontroller keine Gedanken lesen, so dass wir uns einiger Hilfsmittel bedienen müssen. In einem Texteditor, der Bestandteil der Arduino-Entwicklungsumgebung ist, werden unsere Gedanken hinsichtlich der auszuführenden Aktionen mithilfe von Befehlen aus einem Sprachvorrat definiert. Dieser Sprachvorrat enthält Elemente der Programmiersprache C++. Diese werden über einen Compiler (AVR-GCC), der eine Übersetzungsinstanz darstellt, in eine für den Mikrocontroller verständliche Sprache, Maschinensprache genannt, überführt. Die Übertragung dieser Maschinensprache wird von einem Programm mit dem Namen *Avrdude* übernommen. Dieses Kommandozeilen-Tool dient der Übertragung von Code auf den ATmega328-Mikrocontroller, der auf dem Arudino-Board verbaut wurde.

In diesem Kapitel hast du die Softwarewerkzeuge kennengelernt, die du beherrschen musst, um das Arduino-Board programmieren zu können, entweder webbasiert über den Browser oder als Standalone-Anwendung auf deinem Computer zu Hause. Ich habe gezeigt, wie du das Board an den Computer anschließt und wie sofort ein vorinstallierter Sketch eine eingebaute LED zum Blinken bringt. Mit Änderungen am Code kannst du den Blinkrhythmus nach deinen Wünschen ändern. Für den Fall, dass etwas mit dem Verbinden von Computer und Arduino nicht klappen sollte, habe ich einige typische Fehlerquellen aufgeführt.

Im folgenden Kapitel 3 geht es wieder sehr in die Tiefe: Du erfährst Grundlegendes über die Programmierung.

# Kapitel 3:
# Keine Angst vorm Programmieren – Coding Basics

Im vorherigen Kapitel 2 über die Arduino-Software hast du schon einiges über die Programmierung erfahren. Ich habe dir das erste Programm gezeigt, das im Arduino-Umfeld *Sketch* genannt wird. Doch was Programmieren wirklich bedeutet, habe ich noch nicht vertiefend dargestellt. Deshalb mache ich dich in diesem Kapitel mit Grundbegriffen der Programmierung vertraut: Algorithmus, Variablen, Konstanten und verschiedene Datentypen. Und als würde das nicht ausreichen, schiebe ich noch Erläuterungen nach über Kontrollstrukturen und Funktionen. Am Ende dieses Kapitels solltest du eine Ahnung davon haben, auf welcher Klaviatur eigentlich gespielt wird, wenn man ein Computerprogramm schreiben möchte.

Dieses Kapitel führt dich in die Programmiersprache C++ ein, allerdings nur in die Basics. C++ ist die Programmiersprache, in der die Arduino-Sketche erstellt werden. Wenn du Lust hast, dich in diese Sprache zu vertiefen oder später mal etwas nachschlagen möchtest, dann habe ich hier drei gute Online-Tutorials für dich:

https://www.c-howto.de/
http://www.c-programmieren.com/C-Lernen.html
https://www.tutorialspoint.com/cprogramming/

Klar ist, dass für die Programmierung eine Maschine benötigt wird, sei es ein PC oder ein Mikrocontroller wie das Arduino-Board. Klar sollte auch sein, dass ein derartiges Gerät keine eigene Intelligenz besitzt. Es ist ohne Befehle durch ein Programm nichts weiter als ein Stück nutzloser Hardware, das höchstens Strom verbraucht. Hardware und Software leben somit in einer Symbiose, denn keiner kann ohne den anderen auskommen. Erst die Programme »sagen« der Hardware, was sie tun soll. Und der Programmierer sagt der Software, was sie der Hardware sagen soll.

# Was ist ein Programm beziehungsweise ein Sketch?

Bei der Programmierung haben wir es in der Regel mit zwei Bausteinen zu tun.

# Programmierbaustein 1: Der Algorithmus

Der Sketch soll eigenständig eine bestimmte Aufgabe erledigen. Aus diesem Grund wird ein sogenannter Algorithmus erstellt, der eine Sammlung von Einzelschritten beinhaltet, die für ein erfolgreiches Ergebnis erforderlich sind.

Was ist ein Algorithmus?

Ein Algorithmus ist eine Handlungsvorschrift zur Lösung eines Problems und besteht aus endlich vielen Einzelschritten, die in einer gewissen Reihenfolge abgearbeitet werden.

Ein Algorithmus ist eine Rechenvorschrift, die wie ein Checkliste abgearbeitet wird. Stell dir vor, du möchtest eine kleine Holzkiste bauen, um dein Arduino-Board darin unterzubringen, damit alles etwas schöner und aufgeräumter aussieht und es auch deinen Freunden gefällt. Du kaufst dann ja auch nicht einfach Holz und baust drauf los in der Hoffnung, dass alles nachher auch irgendwie zusammenpasst. Es muss also ein Plan her, der zum Beispiel folgende Punkte beinhaltet:

Was sind die Maße der Kiste?
Welche Farbe soll sie haben?
An welchen Stellen müssen Öffnungen gebohrt werden, damit zum Beispiel Schalter oder Lampen platziert werden können?

Wenn du das Material besorgt hast, folgt die eigentliche Arbeit, die in einer ganz bestimmten Reihenfolge erledigt wird:

Holzplatten fixieren.
Holzplatten auf entsprechende Maße zuschneiden.
Kanten mit Schmirgelpapier bearbeiten.
Einige Holzplatten mit Löchern versehen, damit die Anschlüsse angebracht werden können.
Holzplatten zusammenschrauben.
Kiste lackieren.
Arduino-Board einbauen und mit Schalter bzw. Lampe verkabeln.

Es sind viele Einzelschritte nötig, um das gewünschte Ergebnis zu erreichen. Genauso verhält es sich beim Algorithmus.

# Programmierbaustein 2: Die Daten

Sicherlich hast du die Maße der Kiste sorgfältig auf dem Plan vermerkt, damit du während des Baus immer mal wieder einen Blick darauf werfen kannst. Es soll ja später alles gut zusammenpassen. Diese Maße sind vergleichbar mit den Daten eines Sketches. Der Algorithmus nutzt zur Abarbeitung seiner Einzelschritte temporäre Werte, die ihm für seine Arbeit hilfreich sind. Dazu verwendet er eine Technik, die es ihm ermöglicht, Werte abzuspeichern und später wieder abzurufen. Die Daten werden in sogenannten Variablen oder Konstanten im Speicher abgelegt und sind dort jederzeit verfügbar. Doch dazu später mehr.

Was sind Daten?

Im Allgemeinen werden erfasste Informationen oder gemessene Werte als Daten bezeichnet. Diese können in Textform oder als numerische Werte vorliegen.

# Was bedeutet Datenverarbeitung?

Unter *Datenverarbeitung* versteht man das Anwenden eines *Algorithmus*, der unter Zuhilfenahme von *Daten* diese abruft, über unterschiedliche Berechnungen verändert und später wieder ausgibt. Dieses Prinzip wird *EVA* genannt.

Eingabe
Verarbeitung
Ausgabe

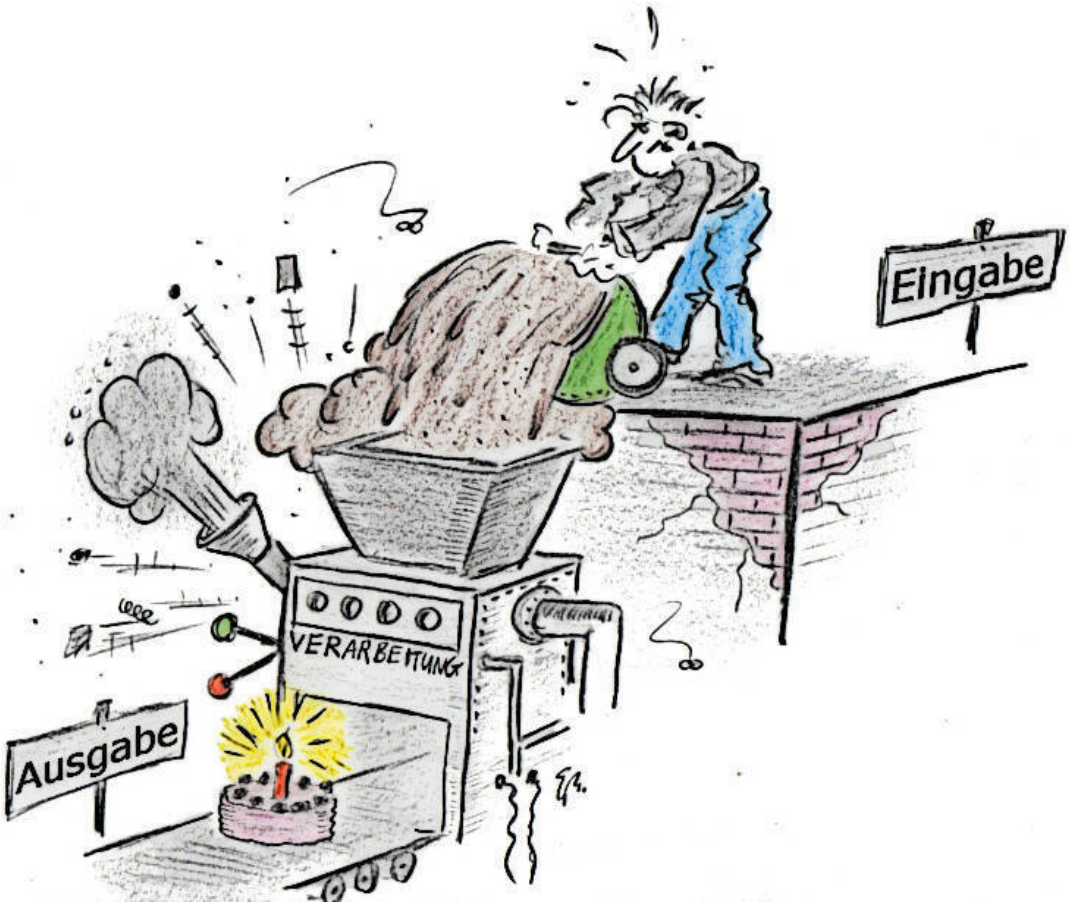



**Abb. 1:** Das EVA-Prinzip

# Was sind Variablen?

Ich hatte schon kurz erwähnt, dass Daten in Variablen abgespeichert werden. Sie spielen in der Programmierung eine zentrale Rolle und werden in der Datenverarbeitung genutzt, um Informationen jeglicher Art zu speichern.

Was ist eine Variable?

In der Programmierung ist eine Variable ein Platzhalter für eine Größe, auf die im Verlauf eines Rechenprozesses zugegriffen werden kann und die im Speicher vorgehalten wird.

Eine Variable belegt innerhalb des Speichers einen bestimmten Platz und hält ihn frei. Der Computer oder Mikrocontroller verwaltet diesen (Arbeits-)Speicher mit seinen eigenen Methoden. All dies erfolgt mittels kryptischer Bezeichnungen, die man sich als Mensch schlecht merken kann. Aus diesem Grund kannst du Variablen mit aussagekräftigen Namen versehen, die intern auf die eigentlichen Speicheradressen verweisen.

In Abbildung 2 siehst du, dass die Variable mit dem Namen *ledPin* auf eine Startadresse im Arbeitsspeicher zeigt. Du kannst sie auch als eine Art Referenz betrachten, die auf etwas Bestimmtes verweist. In Kapitel 2 habe ich einen kurzen Sketch präsentiert, der unter anderem die folgende Codezeile beinhaltet:

```
int ledPin = 13; // Variable mit Pin 13 deklarieren + initialisieren
```

Hier siehst du die Verwendung einer Variablen mit dem Namen *ledPin*, der der numerische Wert 13 zugewiesen wurde. Später im Sketch wird diese Variable ausgewertet und weiter verwendet. Das Wörtchen *int* ist eine Abkürzung für das Wort *Integer*. Integer ist ein Datentyp und wird in der Datenverarbeitung dazu verwendet, um Ganzzahlen zu kennzeichnen, womit wir schon beim nächsten Thema wären.

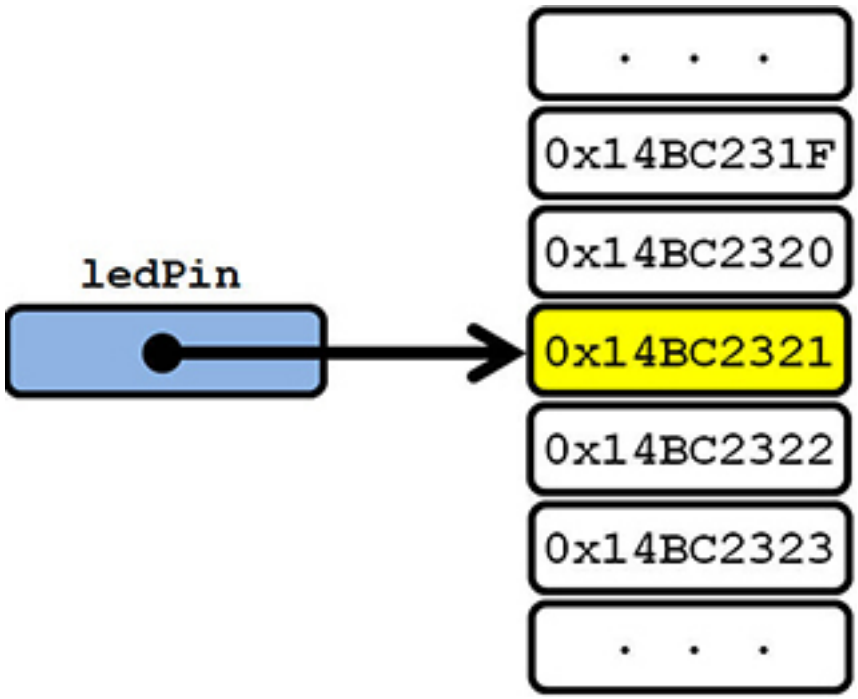



**Abb. 2:** Die Variable ledPin zeigt auf einen Speicherbereich im Arbeitsspeicher.

# Was sind Konstanten?

Irgendwo im Internet habe ich mal gelesen, dass eine konstante Variable eine Variable ist, deren Wert nach der Initialisierung nicht mehr geändert werden kann. Lies diesen Satz zweimal und bedenke, dass Folgendes gilt:

konstant: nicht veränderlich, beständig gleichbleibend.
variabel: veränderbar, nicht auf eine Möglichkeit beschränkt.

Deswegen stellt der erste Satz einen Widerspruch in sich dar. Wer denkt sich so etwas aus? Jedenfalls stimmt der zweite Teil, dass eine Konstante nach der Initialisierung zur Laufzeit des Programms nicht mehr geändert werden kann. Eine Konstante wird mit dem Schlüsselwort *const* für konstant gekennzeichnet.

```
const int a = 17;
```

Folgende Codezeilen wären demnach nicht zulässig und führen zu einer Fehlermeldung, da der Inhalt einer Konstante nicht mehr geändert werden darf:

```
const int a = 17; // Konstante void setup() { a = 18; // Nicht zulässig! } ...
```

# Die Datentypen

Wir sollten uns nun ein wenig mit den unterschiedlichen Datentypen und der Frage, was ein Datentyp überhaupt ist und warum es so viele unterschiedliche davon gibt, beschäftigen. Der Mikrocontroller verwaltet seine Sketche und Daten in seinem Speicher. Dieser Speicher ist ein strukturierter Bereich, der über Adressen verwaltet wird und Informationen aufnimmt oder abgibt, wobei Informationen in Form von Einsen und Nullen gespeichert werden.

Was ist ein Datentyp?

Ein Datentyp beschreibt eine bestimmte Menge von Datenobjekten (Variablen), die allesamt die gleiche Struktur haben. Jede Variable muss dabei einen bestimmten Datentyp haben.

Die kleinste logische Speichereinheit ist das *Bit*, das eben die zwei Zustände 1 oder 0 speichern kann. Stell es dir als eine Art elektronischen Schalter vor, der ein- und ausgeschaltet werden kann. Da du mit einem Bit lediglich zwei Zustände abbilden kannst, sind mehrere Bits zur Speicherung der Daten sinnvoll und notwendig. Der Verbund aus 8 Bits wird 1 *Byte* genannt und ermöglicht es, $2^8$ = 256 unterschiedliche Zustände zu speichern. Die Basis 2 wird verwendet, weil es sich um ein binäres System handelt, das lediglich zwei Zustände kennt. Wir können mit 8 Bits also einen Wertebereich von 0 bis 255 abdecken.

Ich liste hier für den Anfang einmal die wichtigsten Datentypen auf, mit denen du in Zukunft konfrontiert wirst:

Tabelle 1: Datentypen mit entsprechenden Wertebereichen

| Datentyp | Wertebereich | Datenbreite | Beispiel |
|---|---|---|---|
| void | keiner | null | `void setup() {}` |
| byte | 0 bis 255 | 1 Byte | `byte wert = 42;` |
| unsigned int | 0 bis 65.535 | 2 Bytes | `unsigned int sekunden = 46547;` |
| int | –32.768 bis 32.767 | 2 Bytes | `int ticks = -325;` |
| long | $-2^{31}$ bis $2^{31}-1$ | 4 Bytes | `long wert = -3457819;` |
| float | $-3.4 * 10^{38}$ bis $3.4 * 10^{38}$ | 4 Bytes | `float messwert = 27.5679;` |
| double | siehe *float* | 4 Bytes | `double messwert = 27.5679;` |
| boolean | *true* oder *false* | 1 Byte | `boolean flag = true;` |
| char | –128 bis 127 | 1 Byte | `char mw = 'm';` |
| String | variabel | variabel | `String name = "Erik Bartmann";` |

| | | | |
|---|---|---|---|
| Array | variabel | variabel | `int pinArray[] = {2, 3, 4, 5};` |

Die meisten der hier gezeigten Datentypen werden wir auch in diesem Buch verwenden.

# Was sind Funktionen?

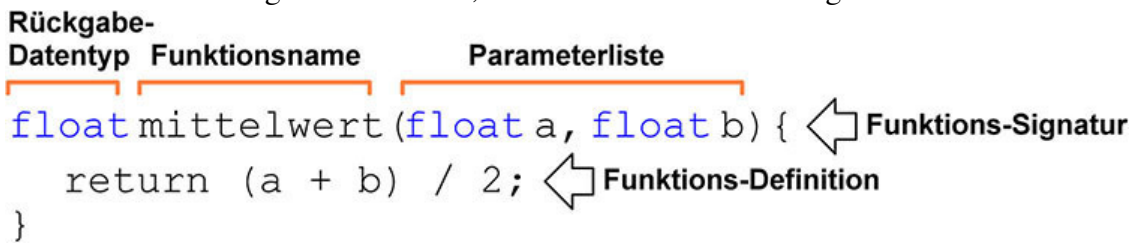
Eine Funktion ist in den meisten höheren Programmiersprachen die Bezeichnung eines Programmkonstrukts, mit dem der Quellcode strukturiert wird, so dass diese Programmteile – die quasi als Unterprogramm bezeichnet werden können – im eigentlichen Hauptprogramm wiederverwendbar sind und somit an unterschiedlichen Stellen mehrfach aufgerufen werden können. In der Objektorientierten Programmierung, auf die ich noch eingehen werde, gibt es vergleichbare Konstrukte, die dort die Bezeichnung *Methoden* besitzen. Dort kommt auch der Begriff *Kapselung* erstmalig zur Sprache, wobei diese Bezeichnung ebenfalls auf die Funktionen zutrifft. Eine Kapselung ist eine Zusammenfassung oder das Verbergen von Daten beziehungsweise Informationen, wobei der Zugriff über eine definierte Schnittstelle erfolgt. Diese Schnittstelle wird durch den Funktionsaufruf abgebildet. Möchtest du in deinem Sketch also zum Beispiel mehrfach den Mittelwert zweier Zahlen bilden, dann sind normalerweise immer die folgenden Codezeilen erforderlich, wobei ich das extra etwas umständlich formuliert habe, damit der Sinn der Funktion etwas deutlicher wird. Es gibt Funktionen, die einen Rückgabewert an den Aufrufer zurückliefern, wie das im Moment hier der Fall ist und es gibt Funktionen, die führen etwas aus, ohne dass ein Wert an den Aufrufer zurückgeliefert wird:

```
float a = 5.4, b = 7.36; float summe = a + b; float mittelwert = summe / 2;
```

Beim Mittelwert zweier Zahlen werden diese addiert und das Ergebnis durch 2 geteilt. Willst du nun an mehreren Stellen im Sketch diesen Mittelwert bilden, wären immer wieder die beiden unteren Zeilen einzufügen. Das wird jedoch durch die Definition einer Funktion erleichtert. Diese könnte wie folgt aussehen:

```
float mittelwert(float a, float b) { return (a + b)/2; }
```

Das sollten wir uns genauer ansehen, denn da kommen viele Dinge zusammen:




Die erste Zeile einer Funktion wird als Signatur bezeichnet und ist die Deklarationszeile einer Funktion. Innerhalb der geschweiften Klammern befindet sich die Definition einer Funktion. Soll eine Funktion etwas an den Aufrufer zurückliefern, steht zu Beginn der sogenannte Rückgabedatentyp, der hier *float* ist. Das macht jedoch zwingend eine *return*-Anweisung innerhalb der Definition erforderlich, denn diese ist im Endeffekt für die Rückgabe eines Wertes verantwortlich, womit die Funktion nach ihrem Aufruf auch letztendlich verlassen wird. Man kann einer Funktion keinen, einen oder mehrere Werte beim Aufruf übergeben. In unserem Fall werden zwei Werte vom Datentyp *float* erwartet, die beim Aufruf an die dort aufgeführten Parameter *a* beziehungsweise *b* übergeben werden. Diese Parameter arbeiten wie lokale Variablen, die nach dem Verlassen der Funktion wieder aus dem Speicher entfernt werden, da sie nicht weiter benötigt werden. Die beiden prominentesten Funktionen in der Arduino-Entwicklungsumgebung sind natürlich die *setup*- und *loop*-Funktion, die immer vorhanden sein müssen:

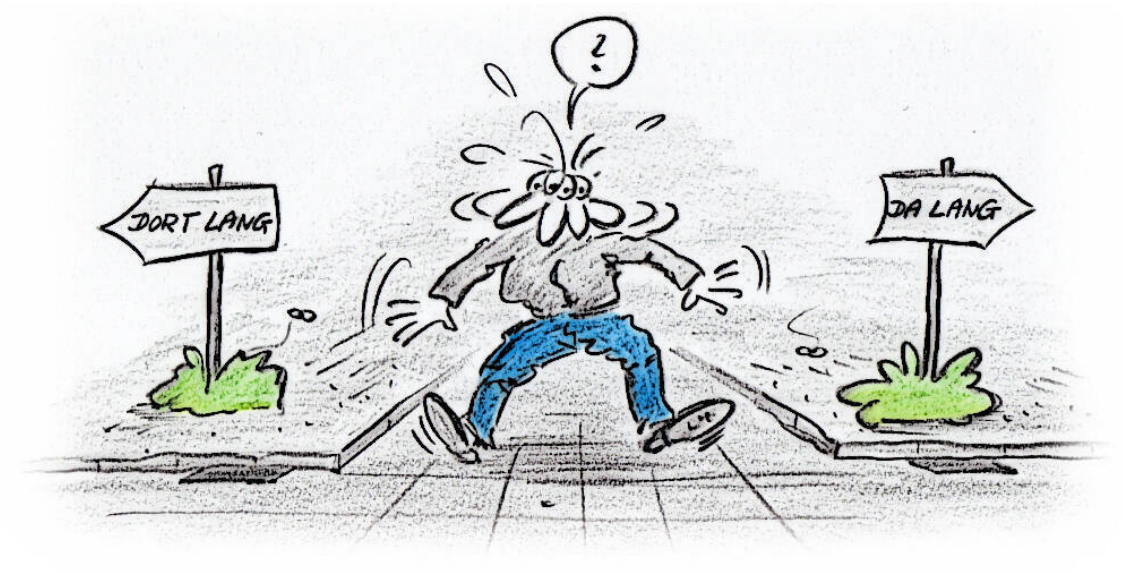
```
void setup() {/* ... */} void loop() {/* ... */}
```

Es ist zu sehen, dass beide den Rückgabedatentyp *void* besitzen, was übersetzt *leer* bedeutet, weil sie keinen Wert zurückliefern. Zudem ist keine Parameterliste zu sehen, was an den leeren runden Klammerpaaren zu erkennen ist. Es können demnach auch beim Aufruf keine Werte mit übergeben

werden. Natürlich gäbe es noch viel mehr über Funktionen zu berichten, doch das ist dann Thema von weiteren Bastelkapiteln oder Teil von C++-Tutorials und würde den Umfang dieses Buches etwas sprengen.

# Was sind Kontrollstrukturen?

In Kapitel 2 – ich hatte es erwähnt – hast du schon etwas über Befehle erfahren. Sie teilen dem Mikrocontroller mit, was er zu tun hat. Ein Sketch besteht aber in der Regel aus einer ganzen Reihe von Befehlen, die sequentiell abgearbeitet werden. Das Arduino-Board ist mit einer bestimmten Anzahl von Ein- und Ausgängen versehen, an die du diverse elektrische und elektronische Komponenten anschließen kannst. Wenn der Mikrocontroller auf bestimmte Einflüsse von außen reagieren soll, schließt du beispielsweise einen Sensor an einen Eingang an. Die einfachste Form eines Sensors ist ein Schalter oder Taster. Wenn der Kontakt geschlossen wird, soll zum Beispiel eine LED leuchten. Der Sketch muss also eine Möglichkeit haben, eine Entscheidung zu treffen: Ist der Schalter geschlossen, dann versorge die LED mit Spannung (LED leuchtet); ist der Schalter offen, dann trenne die LED von der Spannungsversorgung (LED wird dunkel).




Was ist eine Kontrollstruktur?

Kontrollstrukturen sind Anweisungen in imperativen Programmiersprachen. Sie kommen zum Einsatz, um den Ablauf eines Programms zu steuern und es in eine bestimmte Richtung zu lenken. Dabei kann eine Kontrollstruktur entweder eine Verzweigung oder eine Schleife sein. Zumeist wird ihre Ausführung über logische (boolesche) Ausdrücke gesteuert.

Wir werfen zu Beginn einen Blick auf ein Flussdiagramm, das uns zeigt, wie der Ablauf der Sketch-Ausführung in bestimmte Bahnen gelenkt wird, so dass es sich nicht mehr um einen linearen Verlauf handelt. Der Sketch steht beim Erreichen einer Kontrollstruktur an einem Scheideweg und muss sehen, wie es weitergehen soll. Als Entscheidungsgrundlage dient ihm eine Bedingung, die es zu bewerten gilt.

# Die if-Anweisung

Programmtechnisch nutzen wir die *if*-Anweisung. Es handelt sich um eine *Wenn-dann-Entscheidung*.

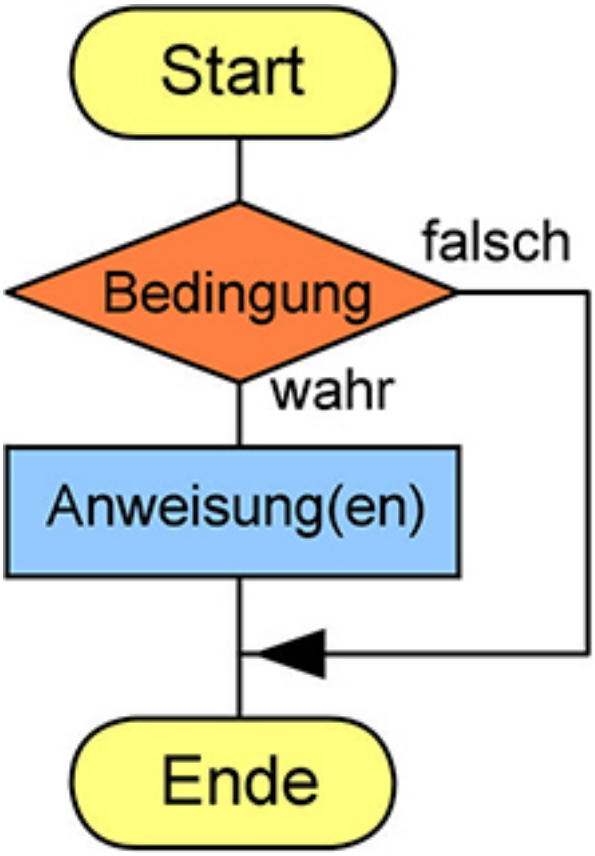



**Abb. 3:** Das Flussdiagramm einer if-Kontrollstruktur

Wurde die Bedingung als *wahr* erkannt, folgt die Ausführung einer oder auch mehrerer Anweisungen. Hier wieder ein kurzes Beispiel:

```
if(tasterStatus == HIGH) digitalWrite(ledPin, HIGH);
```

Wenn mehrere Befehle in einer *if*-Anweisung ausgeführt werden sollen, musst du einen Befehlsblock mit den geschweiften Klammerpaaren bilden. Er wird dann als komplette Befehlseinheit ausgeführt:

```
if(tasterStatus == HIGH) { digitalWrite(ledPin, HIGH); Serial.println("HIGH-Level erreicht."); }
```

# Die if-else-Anweisung

Es gibt noch eine erweiterte Form der if-Kontrollstruktur. Es handelt sich dabei um eine *Wenn-dann-sonst-Entscheidung*, die sich aus einer *if-else*-Anweisung ergibt. Das entsprechende Flussdiagramm sieht wie folgt aus:

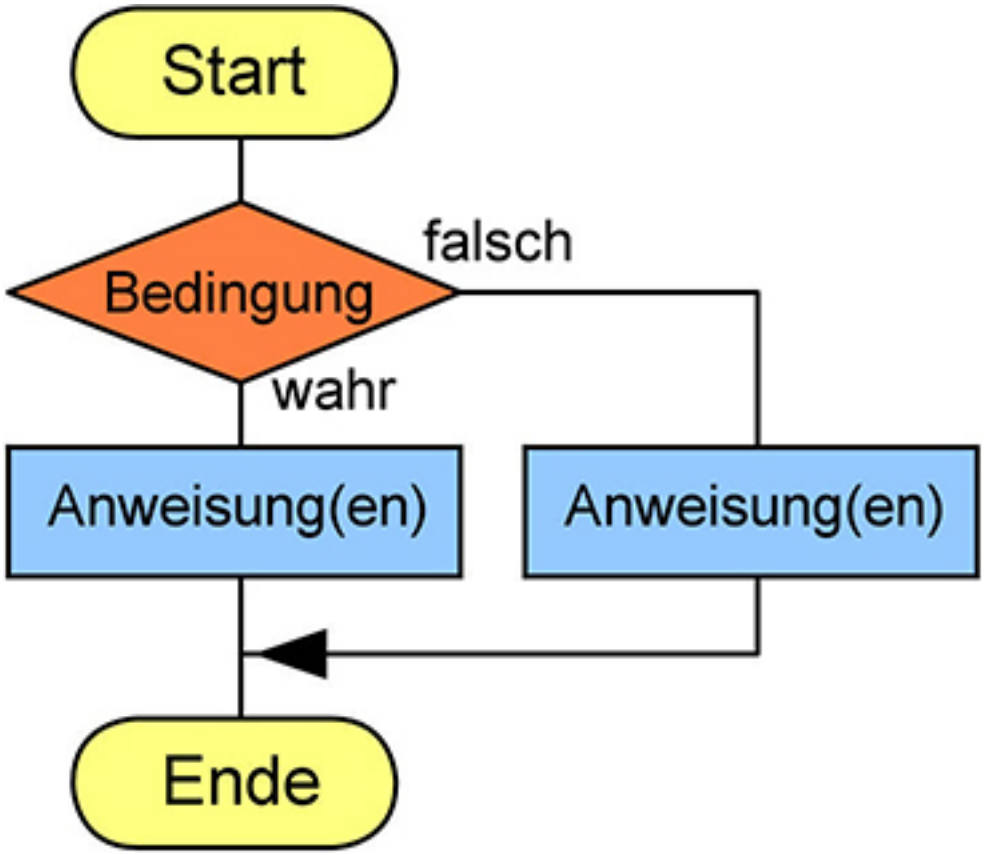



**Abb. 4:** Das Flussdiagramm einer if-else-Kontrollstruktur
Das folgende Codebeispiel zeigt dir die Syntax der *if-else*-Anweisung:

```
if(tasterStatus == HIGH) digitalWrite(ledPin, HIGH); else digitalWrite(ledPin, LOW);
```

# Die switch-case-Anweisung

Sollen mehrere Abfragen hintereinander erfolgen, kann das natürlich über ein Konstrukt mehrfacher *if-then-else*-Anweisungen erfolgen. Es gibt jedoch noch eine einfachere Variante, die vereinfacht zu schreiben und damit auch besser lesbar ist, die sogenannte *switch-case*-Anweisung (Abbildung 5).

Die Syntax dazu sieht wie folgt aus:

switch(var) { case label1: // Anweisung(en) break; case label2: // Anweisung(en) break; default: // Anweisung(en) break; }

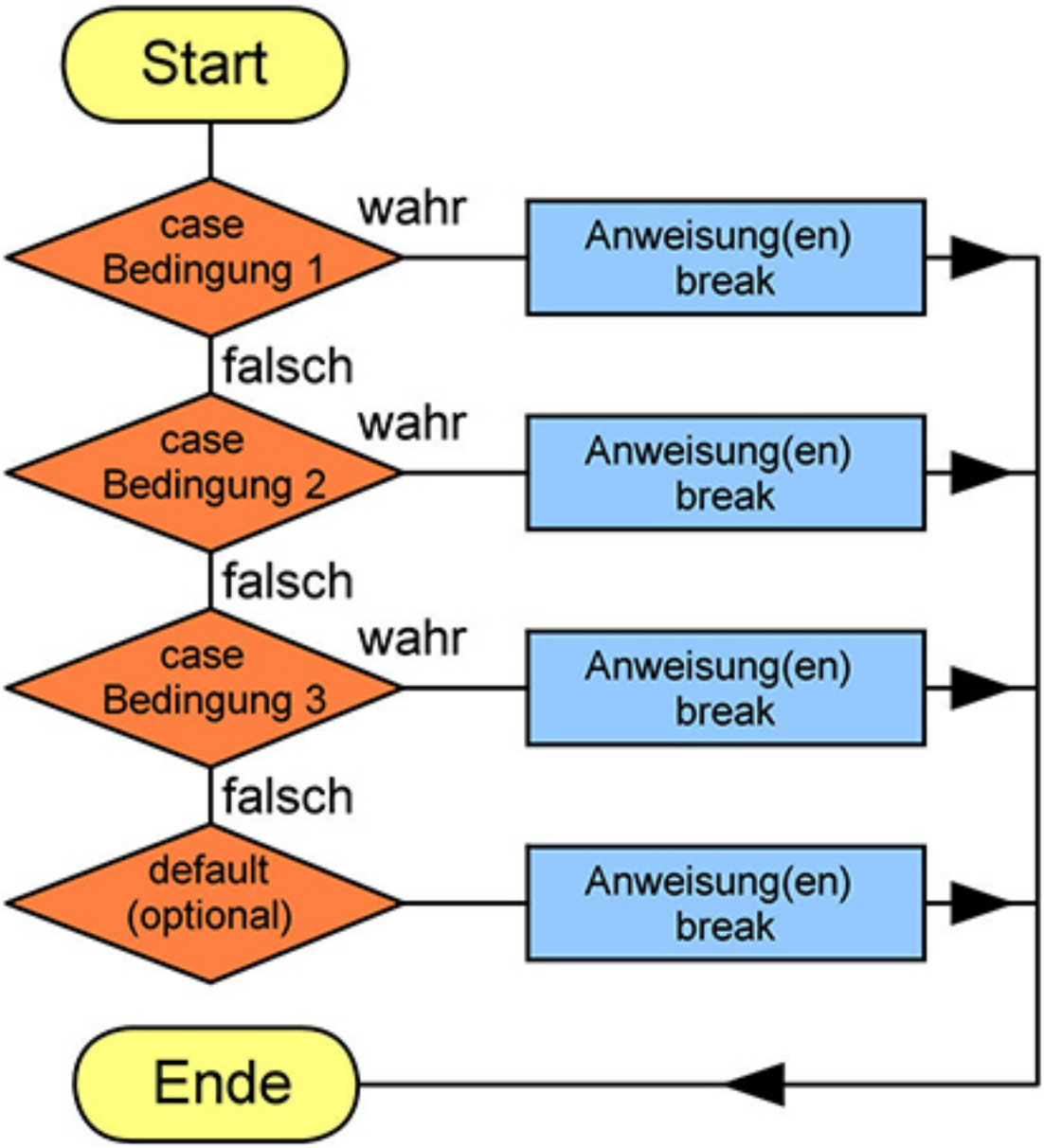



**Abb. 5:** Das Flussdiagramm einer switch-case-Kontrollstruktur

Folgende Parameter sind hierbei erlaubt:

var: eine Variable mit den erlaubten Datentypen int und char.
label1, label2: Konstanten mit den erlaubten Datentypen int und char.

Du solltest unbedingt darauf achten, dass nach der Ausführung einer Anweisung der Schleifendurchlauf mit *break* unterbrochen wird, da sonst die folgenden Sprungmarken ebenfalls geprüft und die dort aufgeführten Anweisungen vielleicht ausgeführt werden. Die letzte *break*-Anweisung Quelltext nach der default-Anweisung ist nicht zwingend erforderlich.

# Operatoren

Natürlich gibt es bei den Kontrollstrukturen und den zu testenden Bedingungen nicht nur die Prüfung auf Gleichheit. Die folgende Tabelle zeigt alle C++-Vergleichsoperatoren:

Tabelle 2: Vergleichsoperatoren

| Vergleichsoperator | Bedeutung | Beispiel |
|---|---|---|
| == | ist gleich | `if(a==b) {...}` |
| <= | ist kleiner gleich | `if(a<=b) {...}` |
| >= | ist größer gleich | `if(a>=b) {...}` |
| < | ist kleiner | `if(<b) {...}` |
| > | ist größer | `if(a>b) {...}` |
| != | ist ungleich | `if(a!=b) {...}` |

Zudem gibt es noch Verknüpfungen über sogenannte logische Operatoren, die mehrere zu testende Bedingungen zulassen:

Tabelle 3: Logische Operationen

| Logische Operation | Funktion | Bedeutung | Beispiel |
|---|---|---|---|
| ! | NICHT (NOT) | Umkehrung des logischen Zustandes. Ergebnis ist wahr (true), wenn Operand falsch (false) ist. | `if(!a) {...}` |
| && | UND (AND) | Ergebnis ist wahr (true), wenn beide Operanden wahr sind. | `if((a<=b)&&(c==5)) {...}` |
| \|\| | ODER (OR) | Ergebnis ist wahr (true), wenn einer der beiden Operanden wahr ist. | `if((a>=b)\|\| (c<=6)) {...}` |

# Was sind Schleifen?

In einem Sketch kann zur Berechnung von Daten das Ausführen vieler einzelner wiederkehrender Schritte erforderlich sein. Wenn es sich bei diesen Schritten beispielsweise immer um gleichartige Befehlsausführungen handelt, ist es weder sinnvoll noch praktikabel, diese Befehle in großer Anzahl untereinander zu schreiben und sequentiell, also hintereinander, ausführen zu lassen. Aus diesem Grund wurde in der Datenverarbeitung ein spezielles programmtechnisches Konstrukt geschaffen, das die Aufgabe hat, ein Programmstück, bestehend aus einem oder mehreren Befehlen, mehrfach hintereinander auszuführen. Wir nennen dies eine *Schleife*.

Was ist eine Schleife?

Eine Schleife – auch Loop genannt – ist eine Kontrollstruktur in einer Programmiersprache. Sie wiederholt einen definierten Anweisungsblock – den sogenannten Schleifenkörper –, solange die Schleifenbedingung logisch wahr ist. Schleifen, deren Schleifenbedingung immer wahr ist oder in denen keine Schleifenbedingung definiert wurde, sind sogenannte Endlosschleifen.

Schauen wir uns an, wie eine Schleife grundsätzlich aufgebaut ist. Es gibt zwei unterschiedliche Schleifenvarianten:

kopfgesteuerte Schleifen und
fußgesteuerte Schleifen.

Beiden Varianten ist gemeinsam, dass sie eine *Instanz* besitzen, die die Kontrolle darüber übernimmt, ob und wie oft die Schleife durchlaufen werden muss. Dieser Instanz ist ein einzelner Befehl oder ein ganzer Befehlsblock (Schleifenkörper genannt) angegliedert, der durch die Instanz gesteuert und abgearbeitet wird.

# Kopfgesteuerte Schleifen

Bei kopfgesteuerten Schleifen befindet sich die Kontrollinstanz im Schleifenkopf, der sich – wie der Name vermuten lässt – oben befindet. Das bedeutet wiederum, dass der Eintritt in den ersten Schleifendurchlauf von der Auswertung der Bedingung abhängt und gegebenenfalls nicht stattfindet. Die Schleife wird also möglicherweise überhaupt nicht ausgeführt.

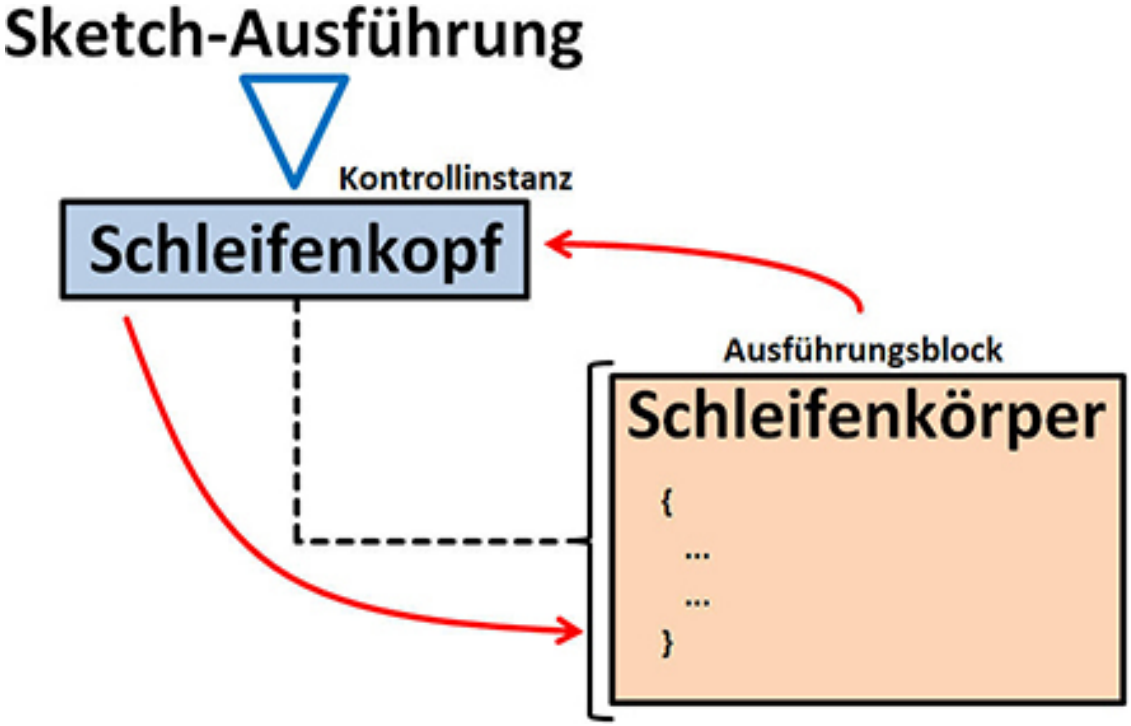



**Abb. 6:** Grundsätzlicher Aufbau einer kopfgesteuerten Schleife

Die Verwendung des Plurals kurz vorher in der entsprechenden Überschrift ist schon ein Hinweis darauf, dass es verschiedene Typen von Kopfschleifen gibt, die in unterschiedlichen Situationen zum Einsatz kommen.

## for-Schleife

Die *for*-Schleife kommt immer dann zum Einsatz, wenn vor Beginn des Schleifenaufrufs eindeutig feststeht, wie oft die Schleife durchlaufen werden soll. Werfen wir dazu einen Blick auf das Flussdiagramm, das der grafischen Wiedergabe des Programmflusses dient:

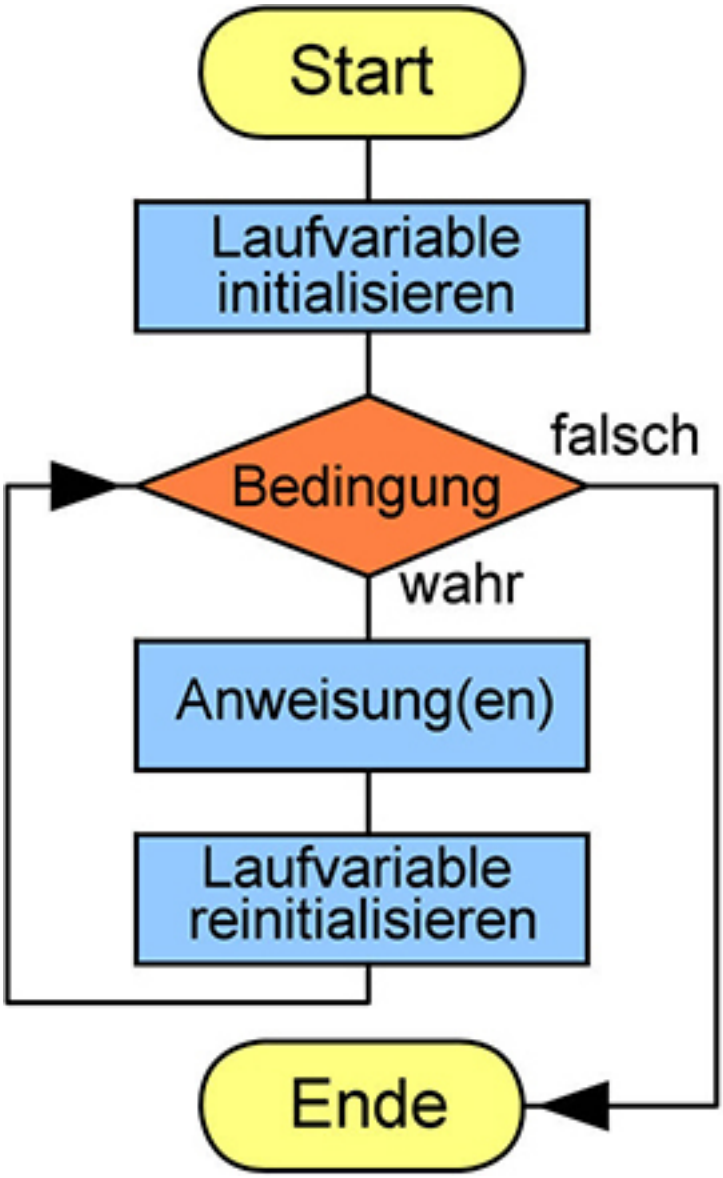



**Abb. 7:** Das Flussdiagramm einer for-Schleife

In der Schleife kommt eine Variable mit der Bezeichnung *Laufvariable* zum Einsatz. Sie wird in der Bedingung einer Bewertung unterzogen, die darüber entscheidet, ob und wie oft die Schleife durchlaufen wird. Der Wert dieser Variablen wird in der Regel im Schleifenkopf bei jedem neuen Durchlauf modifiziert, so dass die Abbruchbedingung irgendwann erreicht sein sollte, wenn du keinen Denkfehler gemacht hast. Hier ein kurzes Beispiel, das später in deinem Projekt verwendet wird:

```
for(int i = 0; i < 7; i++) pinMode(ledPin[i], OUTPUT);
```

## while-Schleife

Die *while*-Schleife wird dann verwendet, wenn sich erst zur Laufzeit der Schleife ergeben soll, ob und wie oft sie zu durchlaufen ist. Wenn während des Schleifendurchlaufs zum Beispiel ein Eingang des Mikrocontrollers kontinuierlich abgefragt und überwacht wird und bei einem bestimmten Wert eine Aktion durchgeführt werden soll, muss dieser Schleifentyp verwendet werden. Wir wollen schauen, wie das entsprechende Flussdiagramm aussieht:

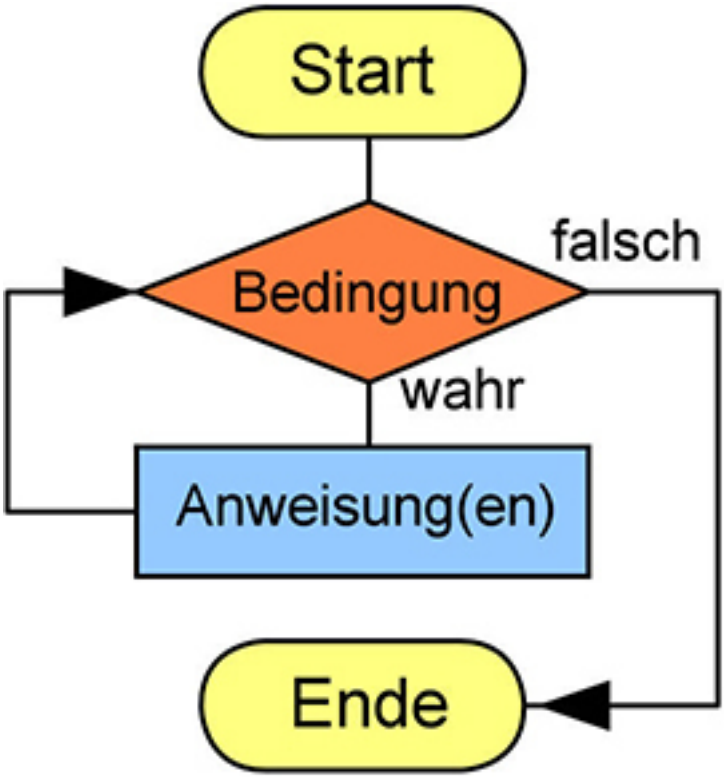



**Abb. 8:** Das Flussdiagramm einer while-Schleife

Die Abbruchbedingung befindet sich bei dieser Schleife ebenfalls im Kopf. Es wird dort jedoch keine Modifikation der in der Bedingung angeführten Variablen vorgenommen. Sie muss im Schleifenkörper erfolgen. Wenn dies vergessen wird, haben wir es mit einer *Endlosschleife* zu tun, aus der es kein Entrinnen gibt, solange der Sketch läuft. Auch hierzu ein kurzes Beispiel für eine *while*-Schleife:

```
while(i > 1) { // Kontrollinstanz Serial.println(i); i = i - 1; }
```

Wenn du in der Kontrollinstanz mit Werten und Variablen arbeitest, die beispielsweise vom Datentyp *float* sind, ist es aufgrund von Ungenauigkeiten von float sehr riskant, genau auf ein bestimmtes Ergebnis hin abzufragen. Das kann bedeuten, dass die Abbruchbedingung niemals erfüllt wird und der Sketch in einer Endlosschleife sein Dasein fristet. Verwende statt des Operators == zur Abfrage auf Gleichheit lieber die Operatoren <= oder >=.

# Fußgesteuerte Schleife

Die fußgesteuerte Schleife wird so genannt, weil die Kontrollinstanz im Schleifenfuß untergebracht ist.

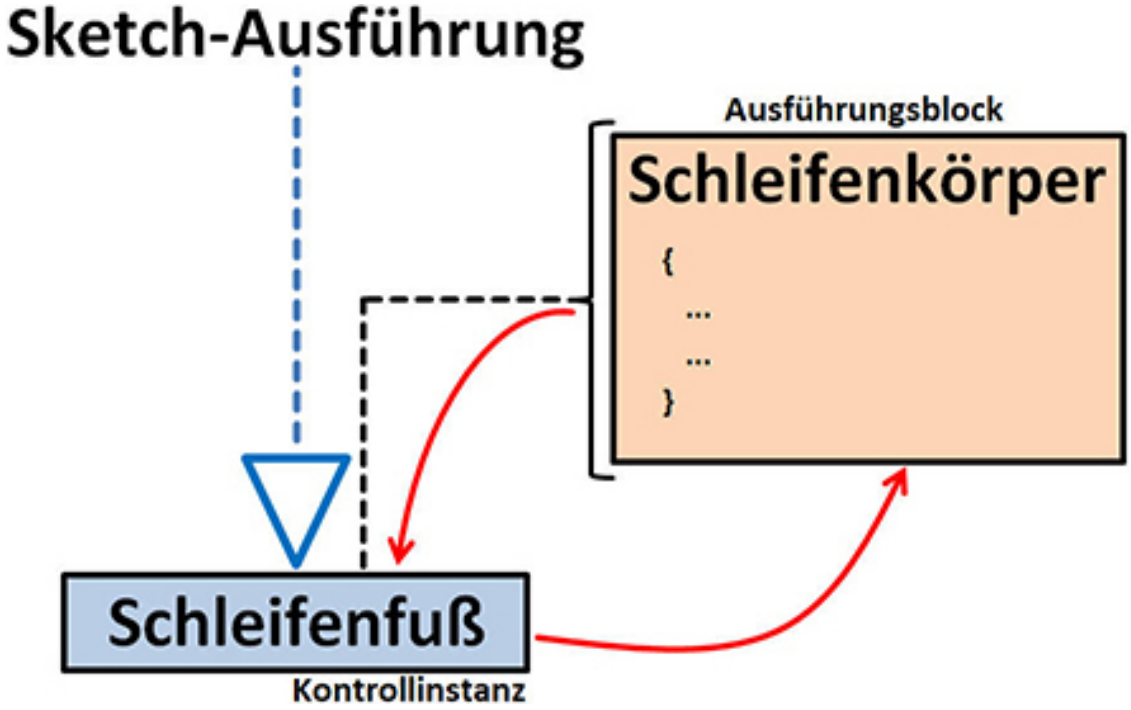



**Abb. 9:** Grundsätzlicher Aufbau einer fußgesteuerten Schleife

Hierbei handelt es sich um eine *do...while*-Schleife. Da die Auswertung der Bedingung erst am Ende der Schleife stattfindet, können wir zunächst festhalten, dass sie mindestens einmal ausgeführt wird.

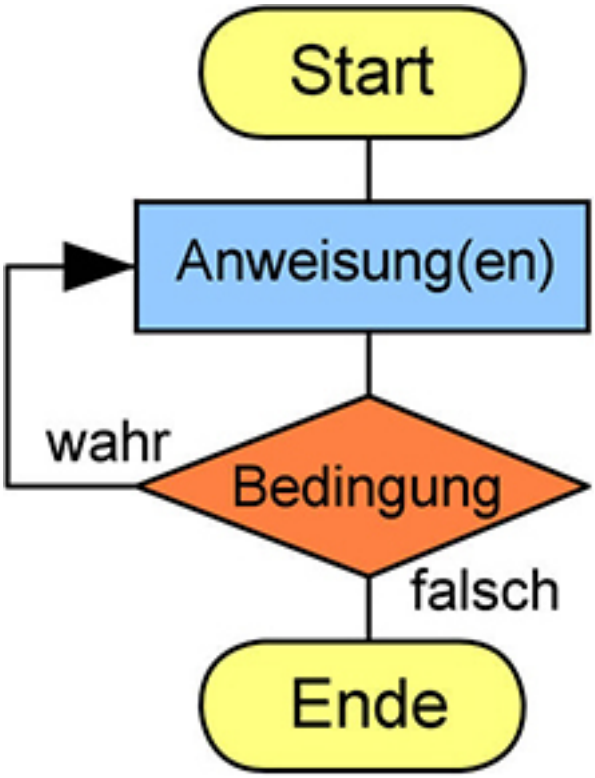



**Abb. 10:** Das Flussdiagramm einer do-while-Schleife

Diese Schleife wird recht selten Verwendung finden, doch der Vollständigkeit halber möchte ich sie dir nicht vorenthalten. Die Syntax gleicht der einer *while*-Schleife, wobei du erkennen kannst, dass die Kontrollinstanz am Fuß der Schleife untergebracht ist:

```
do { Serial.println(i); i = i - 1; } while(i > 1); // Kontrollinstanz
```

Diese unterschiedlichen Schleifenformen wirst du zukünftig verwenden, wenn du deine Sketche schreibst.

# Sei kommunikativ und sprich darüber

Wenn man sich als Programmierer eines Problems annimmt und codiert, ist es sinnvoll, sich hier und da Notizen zu machen. Manchmal hat man einen Geistesblitz oder eine geniale Idee und ein paar Tage später – mir geht es jedenfalls öfter so – fällt es dann schwer, sich an die einzelnen Gedankengänge detailliert zu erinnern. Was habe ich da bloß programmiert und warum habe ich es so und nicht anders gemacht? Natürlich kann jeder Programmierer eigene Strategien für das Ablegen geeigneter Notizen entwickeln: Collegeblock, Rückseite von Werbeprospekten, Word-Dokumente und so weiter. Alle diese Methoden haben jedoch entscheidende Nachteile:

Wo habe ich nur meine Notizen hingelegt?
Sind sie auch auf dem neuesten und aktuellsten Stand?
Jetzt kann ich nicht mal meine eigene Schrift lesen!
Wie kann ich meine Notizen meinem Freund geben, der auch an meiner Programmierung interessiert ist?

Das Problem ist die Trennung von Programmiercode und Notizen, die dann keine Einheit bilden. Wenn die Notizen verloren gehen, wird es für dich unter Umständen schwierig, alles noch einmal zu rekonstruieren. Und jetzt stell dir deinen Freund vor, der absolut keine Ahnung hat, was du mit deinem Code erreichen wolltest. Da muss eine andere Lösung her: Du kannst innerhalb deines Codes Anmerkungen beziehungsweise Hinweise hinterlegen, und das genau an der Stelle, für die sie gerade relevant sind. So hast du alle Informationen genau da, wo sie benötigt werden.

# Einzeiliger Kommentar

Schau dir das folgende Beispiel aus einem Programm an:

int ledPinRotAuto = 7; // Pin 7 steuert rote LED (Autoampel) int ledPinGelbAuto = 6; // Pin 6 steuert gelbe LED (Autoampel) int ledPinGruenAuto = 5; // Pin 5 steuert grüne LED (Autoampel) ...

Hier werden Variablen deklariert und mit einem Wert initialisiert. Zwar sind recht aussagekräftige Namen ausgewählt, doch ich denke, es ist sinnvoll, noch einige kurze ergänzende Anmerkungen anzuführen. Hinter der eigentlichen Befehlszeile wird ein Kommentar eingefügt, der durch zwei Schrägstriche (Slashes) eingeleitet wird. Warum ist das notwendig? Ganz einfach: Der Compiler versucht, alle Befehle zu interpretieren und auszuführen. Nehmen wir einmal den ersten Kommentar:

Pin 7 steuert rote LED (Autoampel)

Es handelt sich um einzelne Elemente eines Satzes, die der Compiler jedoch nicht versteht, da es sich nicht um Anweisungen handelt. Es kommt bei dieser Schreibweise zu einem Fehler beim Kompilieren des Codes. Die beiden // maskieren jetzt aber diese Zeile und teilen dem Compiler mit: *Hey, Compiler, alles, was nach den beiden Schrägstrichen folgt, ist nicht für dich relevant, du kannst es getrost ignorieren. Es handelt sich um eine Gedankenstütze des Programmierers, der sich die einfachsten Sachen nicht über einen längeren Zeitraum (> 10 Minuten) merken kann. Sei etwas nachsichtig mit ihm!* Mittels dieser Schreibweise wird ein *einzeiliger Kommentar* eingefügt.

## Mehrzeiliger Kommentar

Wenn du über mehrere Zeilen etwas schreiben möchtest, beispielsweise etwas, das deinen Sketch in groben Zügen beschreibt, kann es lästig sein, vor jede Zeile zwei Schrägstriche zu positionieren. Aus diesem Grund gibt es die mehrzeilige Variante, die folgendermaßen aussieht:

/* Autor: Erik Bartmann Scope: Ampelsteuerung Datum: 10.12.2020 HP: https://erik-bartmann.de/ */

Dieser Kommentar hat eine einleitende Zeichenkombination /* und eine abschließende Zeichenkombination */. Alles, was sich zwischen diesen beiden *Tags* (ein *Tag* ist eine Markierung, die zur Kennzeichnung von Daten benutzt wird, die eine spezielle Bedeutung haben) befindet, wird als Kommentar angesehen und vom Compiler ignoriert. Alle Kommentare werden von der Arduino-Entwicklungsumgebung mit der Farbe *Grau* versehen, um sie sofort kenntlich zu machen.

So, das also ist die Klaviatur, auf der zu zukünftig zu spielen hast, wenn du Sketche schreiben möchtest. In diesem Kapitel habe ich dir zugegebenermaßen ziemlich abstrakt vermittelt, welche Methoden zum Einsatz kommen, um einer Maschine eindeutig zu sagen, was sie tun soll. Du wirst das Gelesene nun in den Bastelprojekten anwenden. Dabei wächst dein Wissen durch den praktischen Einsatz dieser Methoden. Ich rate dir, ab und zu in diesem Kapitel noch einmal nachzuschlagen, um dein Wissen zu verinnerlichen.

# Kapitel 4:
# Das Arduino-Discoveryboard

Im vorherigen Kapitel hast du einiges nützliches grundlegendes Wissen über das Programmieren erfahren, die dir später beim Umsetzen deiner Ideen sicherlich helfen wird. In diesem Kapitel möchte ich dir eine praktische Möglichkeit vorstellen, immer wieder benötigte elektrische wie elektronische Bauteile auf einer Platine unterzubringen, die du später für deine Bastelprojekte und Arduino-Projekte gut gebrauchen kannst. Diese Platine ist kein Muss, stellt jedoch in bestimmten Situationen eine Erleichterung für den Aufbau elektronischer Schaltungen dar.

Solltest du bisher noch nicht viel Erfahrung mit dem Löten haben, dann empfehle ich dir, zunächst mit einfachen Bastelprojekten aus diesem Buch zu starten, dort einige Erfahrungen beim Löten und beim Zusammenbauen elektrischer und elektronischer Bauteile zu sammeln und dann später das Arduino-Discoveryboard zu bauen. Es bedeutet zunächst einige Arbeit, erspart dir später aber viele Stunden, weil du nicht immer wieder neu Bauteile zusammenbauen musst.

# Das Arduino-Discoveryboard

In der folgenden Abbildung zeige ich dir schon einmal die fertige Platine, die ich dir in diesem Kapitel vorstellen möchte und deren Bau ich ausführlich beschreiben werde. Sie zeigt mehrere Komponenten, die auf einer Lochrasterplatine verlötet wurden, um schnellstmöglich und sicher verschiedene Bauteile verfügbar zu haben. Ich habe diese Platine *Arduino-Discoveryboard* getauft.

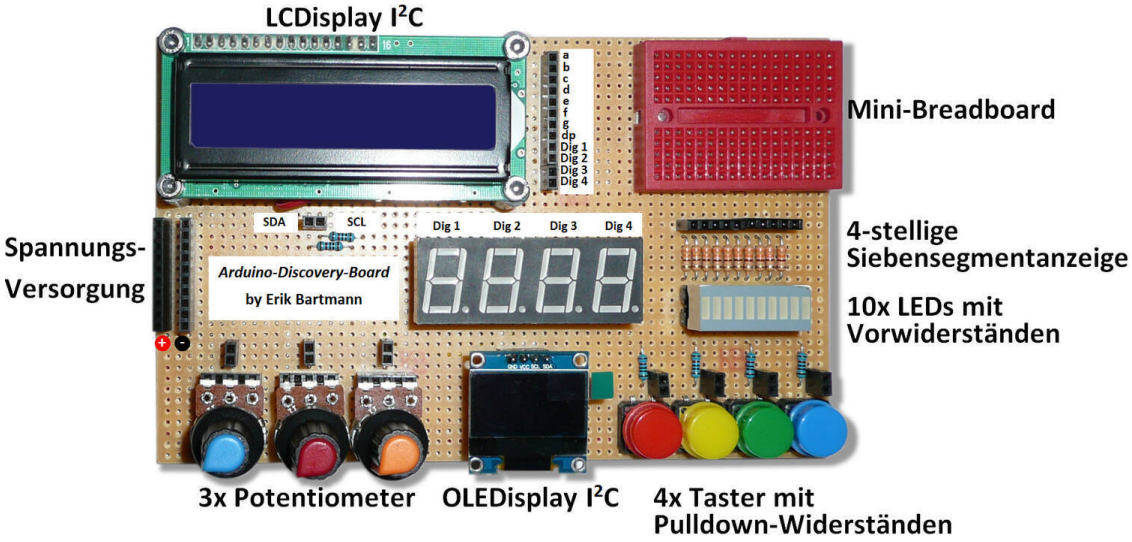
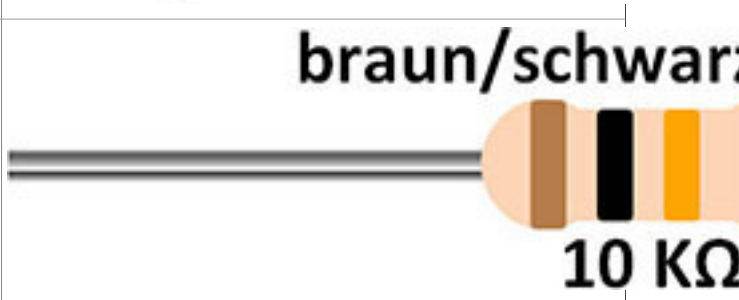
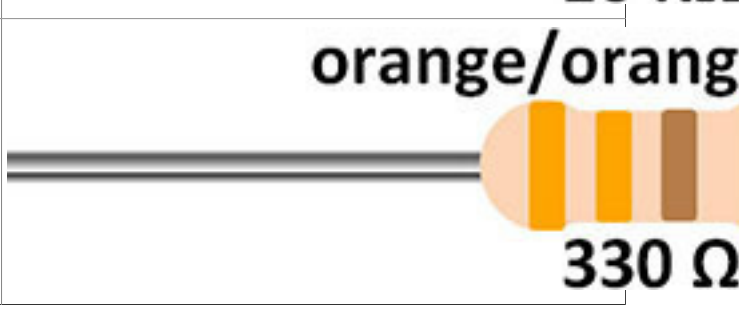



**Abb. 1:** Das Arduino-Discoveryboard

Die Idee hatte ich, als ich zum x-ten Male mehrere Potentiometer zur Ansteuerung der analogen Eingänge des Arduino benötigte, um damit den Analog-Monitor zu realisieren. Jedes Mal musste ich für diverse Bastelprojekte die erforderlichen Potentiometer aus meiner Grabbelkiste suchen, sie an die Spannungsversorgung anschließen und die Schleiferkontakte dann endgültig mit den analogen Eingängen A0 bis A5 verbinden. Das stellt eigentlich kein größeres Problem dar, doch wenn man die ganze Sache zum wiederholten Male machen muss, ist das lästig. Wenn dann noch Taster oder Schalter hinzukommen, ist es komfortabel, alles an einem Platz zur Verfügung zu haben. Alles, was du dazu benötigest, sind eine ruhige Hand und die Bauteile, die ich gleich aufliste. Versuch nicht allzu verbissen, genau das Discoveryboard zu bauen, das ich hier vorstelle. Nimm es als Anregung, dein eigenes Discoveryboard zu entwerfen, eigene Anordnungen oder andere Bauteile auf einer Platine unterzubringen.

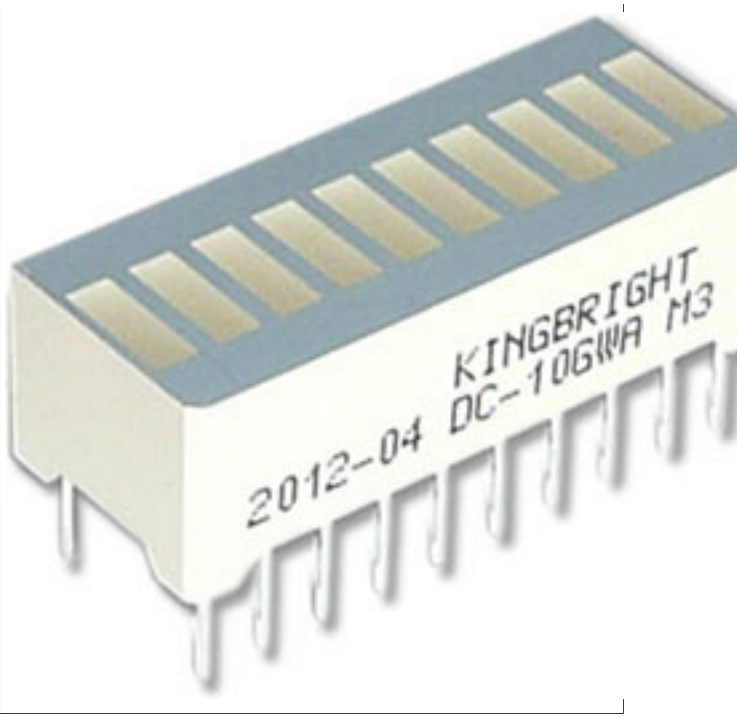
# Was wir brauchen

Für das Arduino-Discoveryboard benötigen wir die folgenden Bauteile:

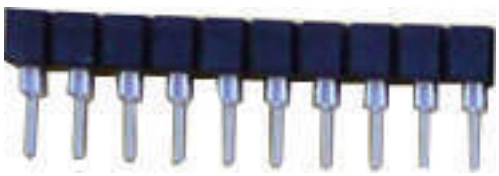
| Tabelle 1: Bauteilliste | |
|---|---|
| Bauteil | Bild |
| Mikrotaster 3x (mit farbiger Kappe) |  |
| Widerstand 10KΩ 6x |   |
| Widerstand 330Ω 14x |   |

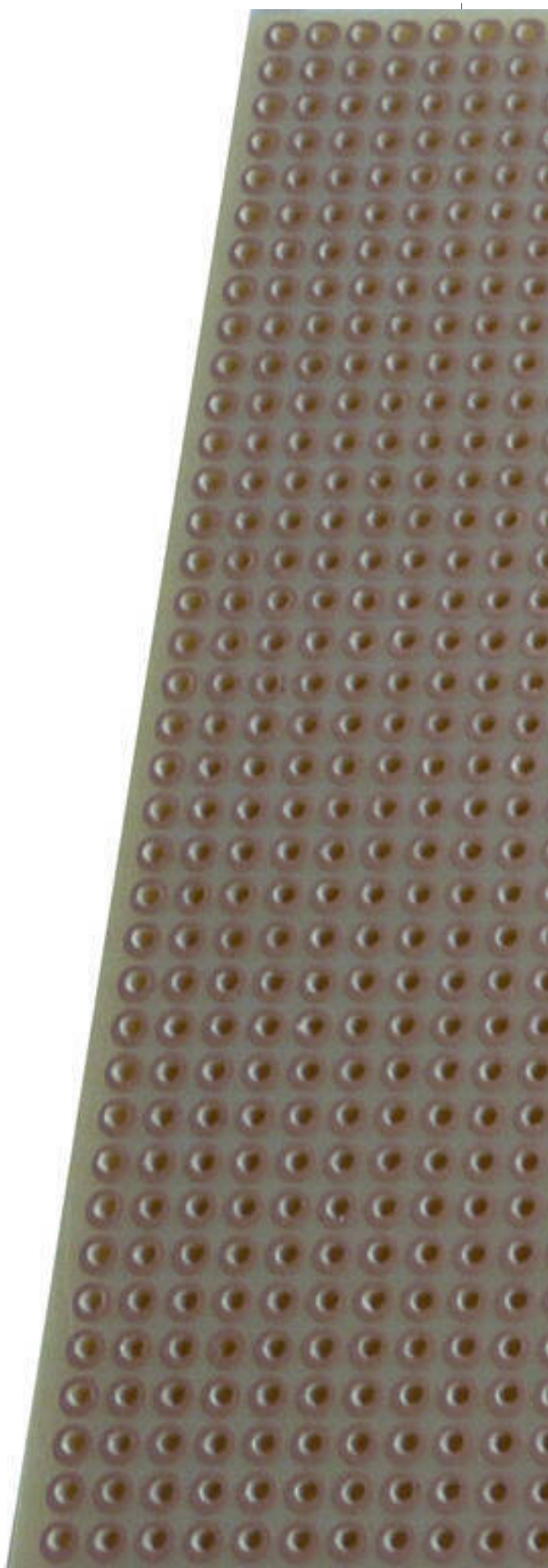
| | |
|---|---|
| Potentiometer 10KΩ 3x |  |

| | |
|---|---|
| LC-Display-I²C 1x |  |

| | |
|---|---|
| Bargraph z.B. Kingbright DC-10EWA 1x |  |

| | |
|---|---|
| 2,4 cm (0,96 Zoll) OLED-Display, 128 x 64 Pixel, LCD-Treiber 1x |  |

| | |
|---|---|
| Buchsenleisten 64-polig (RM: 2.54) 3x |  |
| 4-stellige Siebensegmentanzeige z.B. CL5641BH 1x |  |
| Breadboard-Mini 1x |  |

| | |
|---|---|
| Lochrasterplatine 160 x 100 (RM: 2.54) 1x |  |

| | |
|---|---|
| Gummifüße 4x |  |

# Der Schaltplan

Der Schaltplan ist wirklich eine einfache Sache und leicht zu verstehen.

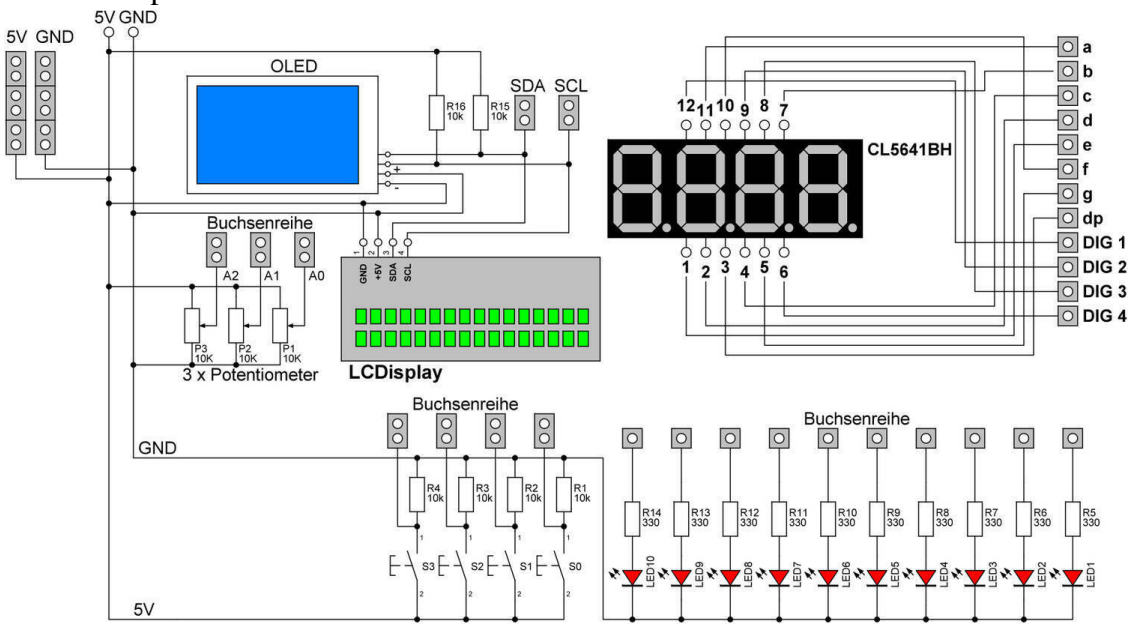



**Abb. 2:** Der Discoveryboard-Schaltplan

Beim Verlöten auf der Platinenrückseite musst du eine sehr ruhige Hand haben, doch wenn ich das hinbekommen habe, schaffst du das auch. Da die Lötpunkte sehr dicht beieinanderliegen, ist es ratsam, eine Lötpumpe griffbereit zu haben. Es passiert sehr schnell, dass zwei benachbarte Punkte einen Schluss bekommen, wenn zu viel Lötzinn verwendet wird. Kein Drama, aber nervig. Es ist mir recht oft passiert.

# Die Siebensegmentanzeige

Für die Anzeige wurde eine vierstellige Siebensegmentanzeige mit gemeinsamer Anode verwendet, die vom Typ CL5641BH ist, wobei natürlich auch andere Bauteile mit ähnlichen Spezifikationen verwendet werden können. Die Pinbelegung der Anzeige schaut wie folgt aus.

Wie diese Siebensegmentanzeige anzusteuern ist, wird in einem eigenen Bastelprojekt 17 natürlich genau beschrieben. Es ist im Schaltplan zu erkennen, dass alle Segmente der einzelnen Stellen zusammengeschaltet sind. Das würde natürlich bedeuten, dass alle Segmente auf allen Stellen gleichzeitig leuchten würden, was natürlich keinen Sinn macht. Der Knackpunkt ist die gezielte beziehungsweise intelligente Ansteuerung der einzelnen Anoden der Stellen Dig1 bis Dig 4. Diese Anode sind alle miteinander verbunden, was *gemeinsame Anode* genannt wird. Um nun aber jede einzelne Stelle separat mit unterschiedlichen Stellenwerten anzusteuern, kommt ein bestimmtes Verfahren zur Anwendung. Es wird also spannend!

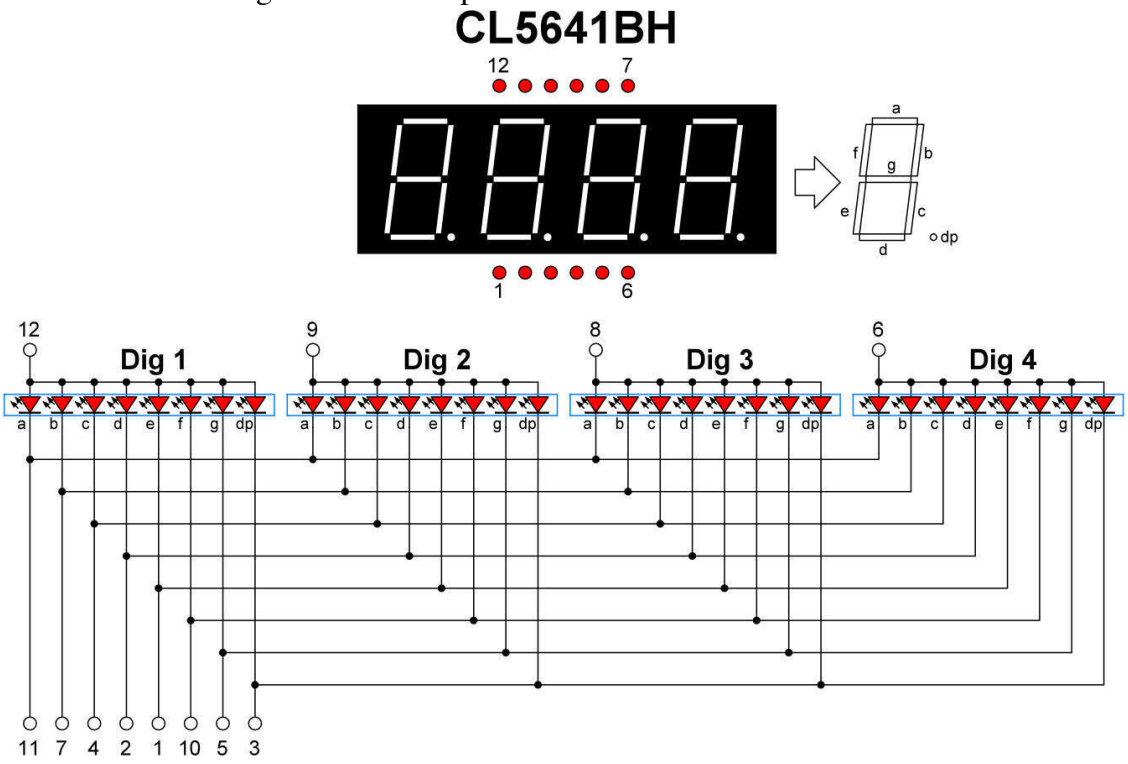



**Abb. 3:** Die Pinbelegung der Anzeige CL5641BH

Besuch bitte regelmäßig meine Webseite. Ich stelle dort im Downloadbereich gelegentlich weitere Updates zum Arduino Discoveryboard rein.

# Bastelprojekt 1:
# Hallo Welt – das Blinken einer LED

In den meisten Büchern zum Erlernen einer Programmiersprache wird zu Anfang ein sogenanntes *Hallo-Welt*-Programm präsentiert. Es soll einen Einblick in die Syntax der Programmiersprache bieten, indem es etwas recht Simples tut, und zwar nur den Text *Hallo Welt* auf dem Bildschirm ausgibt. Auf diese Weise kann ein Programmierer einen zeitsparenden Eindruck von der Programmiersprache und seiner Syntax gewinnen.

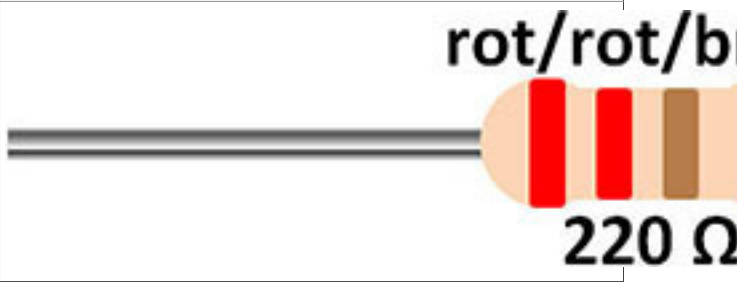
# »Hallo Welt« wird geblinkt

Wie sieht ein Hallo-Welt-Programm bei dem Arduino aus? Der Arduino hat ja in seinem Urzustand kein Display, also kein Anzeigegerät, um sich dir mitzuteilen. Was also tun? Wenn eine Kommunikation nicht in schriftlicher Form möglich ist, dann vielleicht mittels optischer oder akustischer Signale. Wir entscheiden uns für die optische Variante, denn einen Signalgeber wie eine Leuchtdiode, auch *LED* genannt, können wir ohne allzu große Probleme an einen der digitalen Ausgänge klemmen und er erregt bestimmt Aufmerksamkeit. Ich war jedenfalls sehr beeindruckt, als es bei mir auf Anhieb funktioniert hat. Schauen wir uns zuerst die Bauteilliste an.

# Was wir brauchen

Für dieses Bastelprojekt wird nicht viel benötigt und im Grunde genommen könnten wir auch ohne zusätzliche Bauteile auskommen, denn auf dem Arduino-Board befindet sich eine LED mit der Bezeichnung *L*. Dennoch möchte ich dieses Bastelprojekt mit ein paar Komponenten anreichern, die auch in weiteren Projekten Verwendung finden.

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED rot 1x |  |
| Widerstand 220Ω 1x |   |

Bevor wir jedoch einen Blick auf das Arduino-Programm – oder auch Sketch genannt – werfen, sehen wir uns den Schaltplan an. Ein Schaltplan ist übrigens eine grafische Darstellung einer elektronischen Schaltung, manche nennen das auch Schaltbild.

# Der Schaltplan

Unsere Schaltung weist lediglich eine LED mit passendem Widerstand auf.

Die Anode der LED (das längere Beinchen) wird über den Widerstand mit 220Ω an Pin 13 verbunden und das andere Ende, bei dem es sich um die Kathode handelt (das kürzere Beinchen), mit der Masse (GND) des Arduino-Boards. Beim folgenden Schaltungsaufbau verwende ich erstmalig die Kombination aus Arduino-Board und Breadboard, die ich bereits in den Einführungskapiteln erwähnt habe (Kapitel 2). In späteren Bastelprojekten werde ich das Arduino Discoveryboard einsetzen, aber du kannst selbstverständlich alle Bastelprojekte auch auf einem ganz normalen Breadboard realisieren.

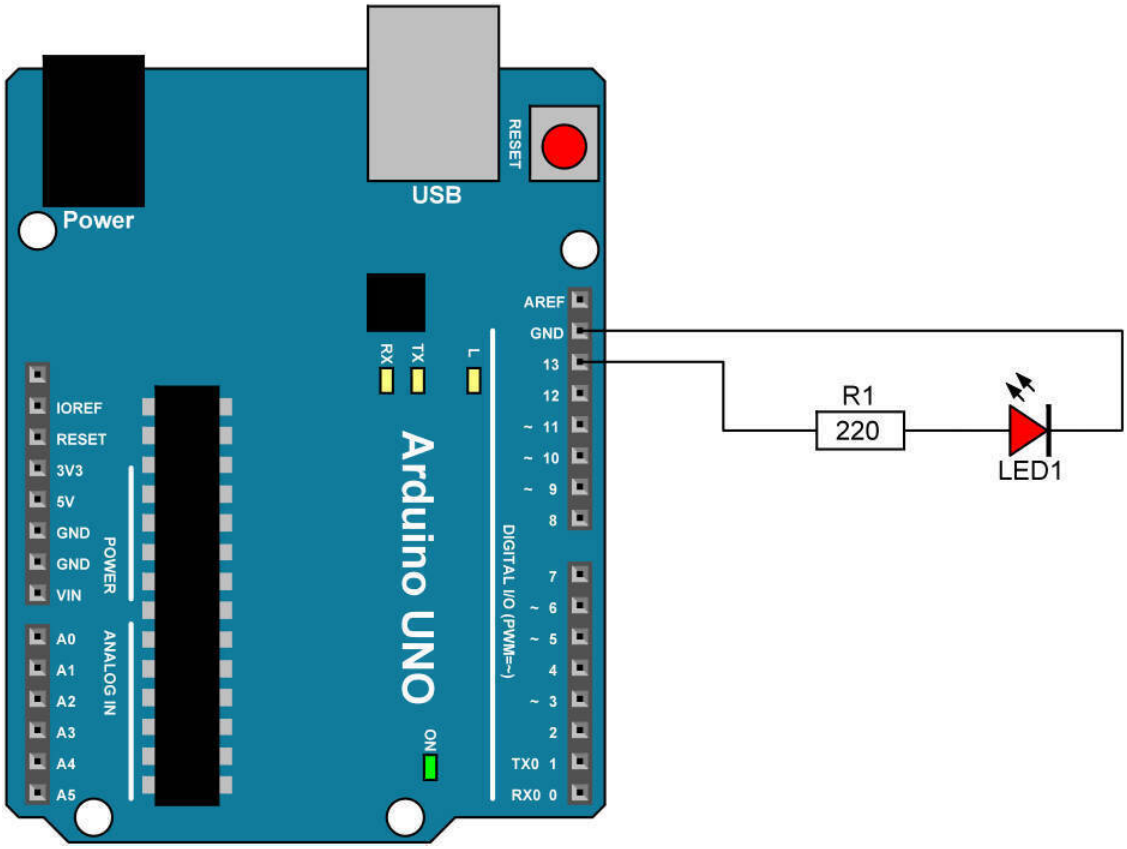



**Abb. 1:** Der Schaltplan

# Der Schaltungsaufbau

Der Schaltungsaufbau auf einem Breadboard ist leicht nachvollziehbar und übersichtlich. Nachfolgend ist der Arduino Uno mit dem Breadboard zu sehen. Beide sind über Steckbrücken verbunden. An dem Breadboard habe ich die LED und den Widerstand platziert.

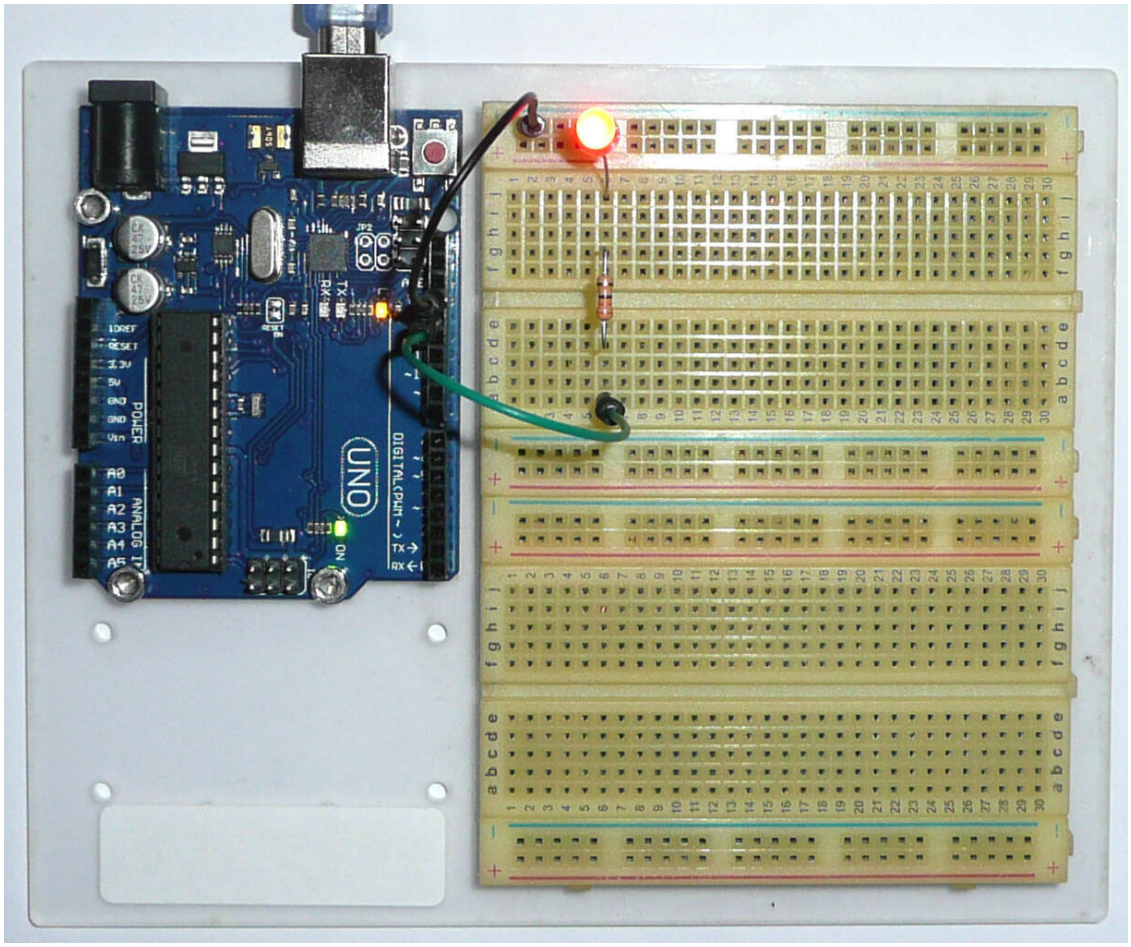



**Abb. 2:** Der Schaltungsaufbau auf einem kleinen Arduino-Combi-Board
Was ist bei der LED-Polung zu beachten?

Achte auf die korrekte Polung der LED, denn andernfalls können wir lediglich eine dunkle LED bewundern. Man läuft zwar mit einer falsch gepolten LED nicht Gefahr, etwas zu beschädigen, doch es sollte schon richtig gemacht werden.

# Der Arduino-Sketch

Der Programmcode für unseren ersten Sketch bewirkt, dass er eine über einen Widerstand angeschlossene LED im Sekundentakt blinken lässt. Das Programm dafür sieht folgendermaßen aus:

int ledPin = 13; // Variable mit Pin 13 deklarieren + initialisieren void setup() { pinMode(ledPin, OUTPUT); // Digitaler Pin 13 als Ausgang } void loop() { digitalWrite(ledPin, HIGH); // LED auf High-Pegel (5V) delay(1000); // Eine Sekunde warten (1000ms) digitalWrite(ledPin, LOW); // LED auf LOW-Pegel (0V) delay(1000); // Eine Sekunde warten (1000ms) }

Ich empfehle dringend, den Code selbst abzutippen. Aus meiner Sicht hat das einen hohen Lerneffekt. So lernst du beispielsweise, immer ein Semikolon am Ende einer Befehlszeile zu setzen. Du bekommst ein besseres Gefühl für die Feinheiten der Programmierung, wenn du selbst die Codezeilen eingibst. Das gilt besonders für den Code in den ersten Bastelprojekten. Bei späteren Projekten wird der Code manchmal so umfangreich, dass eine Eingabe per Hand zu aufwändig wäre.

Der im Buch verwendete Code steht hier zum Download zur Verfügung:

https://erik-bartmann.de/?Downloads___Arduino

Der grundlegende Aufbau eines Arduino-Sketches

Jeder Arduino-Sketch hat den gleichen Aufbau. In der folgenden Abbildung 3 wird dieser Aufbau klar:

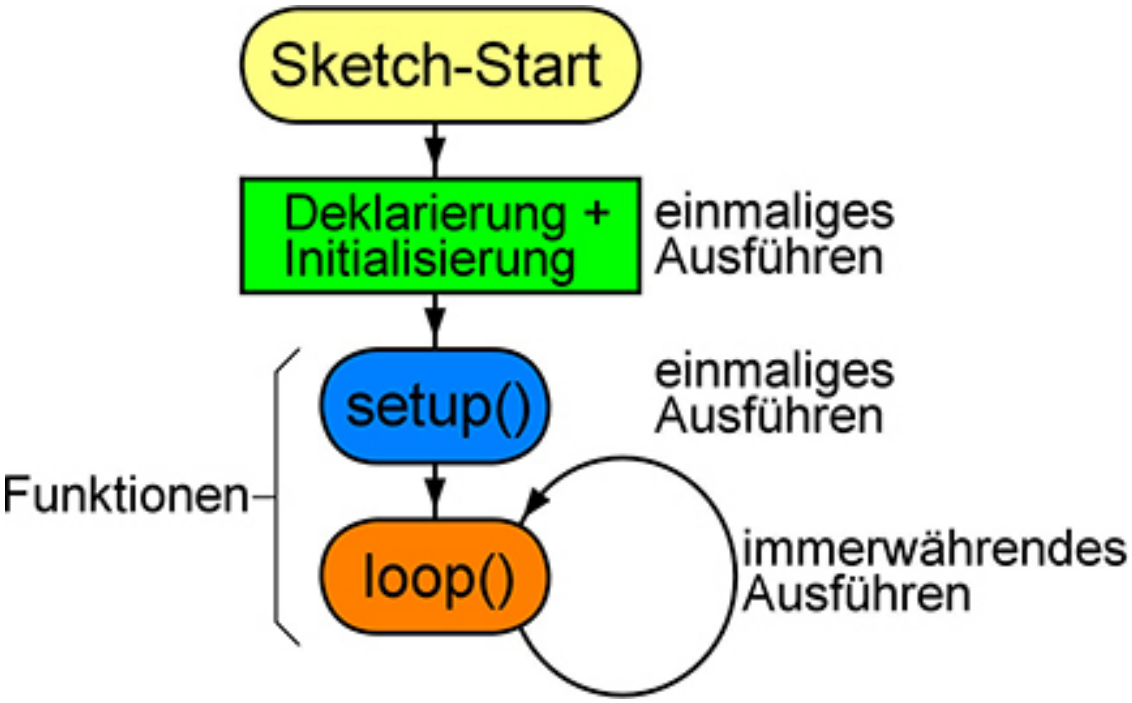



**Abb. 3:** Die grundlegende Struktur eines Sketches

Das ganze Programm ist – nach dem Start des Sketches – in drei Blöcke unterteilt, die du oben erkennen kannst. Die einzelnen Blöcke haben eine klar definierte Aufgabe in einem Sketch.

## Grüner Block: Die Deklarierung und Initialisierung

In diesem ersten Block werden beispielsweise – falls notwendig – externe Bibliotheken über die *#include*-Anweisung eingebunden. Wie das funktioniert, wirst du in späteren Bastelprojekten erfahren. Des Weiteren ist hier der geeignete Platz zur Deklaration globaler Variablen, die innerhalb des kompletten Sketches sichtbar sind und verwendet werden können. Bei der Deklaration wird festgelegt, welchem Datentyp die Variable zugeordnet sein soll. Bei der Initialisierung hingegen wird die Variable mit einem Wert versehen.

# Blauer Block: Die setup-Funktion

In der *setup*-Funktion werden meistens die einzelnen Pins des Mikrocontrollers programmiert. Es wird also festgelegt, welche der Pins als Ein- und Ausgänge arbeiten sollen. An manchen werden womöglich Sensoren wie Taster oder temperaturempfindliche Widerstände angeschlossen, die Signale von außen an einen entsprechenden Eingang leiten. Andere wiederum leiten Signale an Ausgänge weiter, um zum Beispiel einen Motor oder eine Leuchtdiode anzusteuern.

# Oranger Block 3: Die loop-Funktion

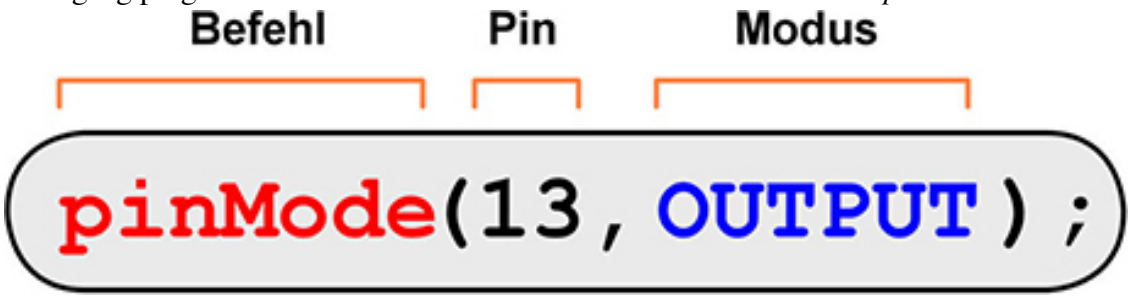
Die *loop*-Funktion bildet eine Endlosschleife. In ihr ist die Logik untergebracht, beispielsweise werden kontinuierlich Sensoren abgefragt oder Motoren angesteuert. Beide Funktionen, also die *setup*- und die *loop*-Funktion, bilden zusammen einen Ausführungsblock, der im Code durch die geschweiften Klammerpaare {} gekennzeichnet wird. Sie dienen als Begrenzungselemente, damit erkennbar ist, wo die Funktionsdefinition beginnt und wo sie aufhört.

Die beiden Funktionen *setup* und *loop* müssen genau so heißen, denn beim Start des Sketches wird nach ihnen gesucht, weil sie als Einstiegspunkte dienen, um einen definierten Start zu gewährleisten. Woher sollte der Compiler wissen, welche Funktion nur einmal ausgeführt werden soll und welche kontinuierlich in einer Endlosschleife? Diese Namen sind also unerlässlich für jeden Sketch. Ihnen wird jeweils noch das Schlüsselwort *void* vorangestellt. Es zeigt an, dass die Funktion voraussichtlich keine Informationen an die Funktion zurückgibt, von der sie aufgerufen wurde.

## Den Code verstehen

Zu Beginn deklarieren und initialisieren wir eine globale Variable namens *ledPin*, ihr weisen wir den Wert 13 zu. Mit dem Befehl *int* (int = Integer) bestimmen wir, dass es ein ganzzahliger Datentyp ist. Somit ist sie in allen Funktionen sichtbar und es kann darauf zugegriffen werden. Die Initialisierung ist gleichbedeutend mit einer Wertzuweisung über den Zuweisungsoperator =. Die Deklaration und Initialisierung erfolgt hier in einer einzigen Zeile.


Die *setup*-Funktion wird einmalig zu Beginn des Sketch-Starts aufgerufen und der digitale Pin 13 als Ausgang programmiert. Sehen wir uns dazu noch einmal den Befehl *pinMode* an.




Er nimmt zwei numerische Argumente auf, wobei der erste für den zu konfigurierenden Pin steht und der zweite bestimmt, ob sich der Pin wie ein Eingang oder Ausgang verhalten soll. Wir wollen ja eine LED anschließen, und deswegen benötigen wir einen Pin, der als Ausgang arbeitet. Die Datenflussrichtung des zweiten Argumentes wird über eine vordefinierte Konstante festgelegt. Hinter OUTPUT verbirgt sich der Wert 1.

Ebenso verhält es sich mit dem Befehl *digitalWrite*, der ebenfalls zwei Argumente entgegennimmt.


Hier haben wir ebenfalls eine Konstante mit dem Namen `HIGH` , die als Argument bewirken soll, dass ein *HIGH*-Pegel an Pin 13 anliegt. Dahinter verbirgt sich der numerische Wert 1. In der folgenden Tabelle findest du die entsprechenden Werte:

Tabelle 2: Konstanten und ihre Werte

| Konstante | Wert | Erklärung |
|---|---|---|
| INPUT | 0 | Konstante für den Befehl *pinMode* (programmiert Pin als Eingang) |
| OUTPUT | 1 | Konstante für den Befehl *pinMode* (programmiert Pin als Ausgang) |
| LOW | 0 | Konstante für den Befehl *digitalWrite* (setzt Pin auf LOW-Level) |
| HIGH | 1 | Konstante für den Befehl *digitalWrite* (setzt Pin auf HIGH-Level) |

Der Befehl *delay* im Sketch ist für die Zeitverzögerung zuständig. Er unterbricht die Sketch-Ausführung für einen entsprechenden Zeitraum, wobei der übergebene Wert diese Zeitdauer in Millisekunden (ms) angibt.


Der Wert *1000* besagt, dass genau 1000ms, also eine Sekunde, gewartet wird, bis es schließlich weitergeht.

Die einzelnen Arbeitsschritte der Endlosschleife, die durch den *void loop*-Befehl ausgelöst wurde, sind:

1 LED an Pin 13 anschalten,
2 Warte 1 Sekunde,
3 LED an Pin 13 ausschalten,
4 Warte 1 Sekunde,
5 Geh wieder zu Punkt 1.

# Der zeitliche Verlauf visualisiert

Es ist zwar schwierig zu erkennen, doch beim genaueren Hinschauen können wir sehen, dass die Onboard-LED zur selben Zeit leuchtet wie die extern angeschlossene LED. Die LEDs sollten direkt nach der erfolgreichen Übertragung zum Board zu blinken beginnen. Ein Oszillogramm zeigt den zeitlichen Verlauf des Impulses am digitalen Ausgang etwas genauer an, wobei ich den Pegel direkt am Ausgang ohne den Widerstand aufgenommen habe. Er wechselt kontinuierlich zwischen 0V und 5V, was hier mit L (LOW) und H (HIGH) gekennzeichnet ist.


**Abb. 4:** Der Verlauf des Pegels am digitalen Ausgang Pin 13

Achtung!

Im Internet kursieren Schaltskizzen, bei denen eine Leuchtdiode direkt zwischen Masse und Pin 13 gesteckt wurde. Da die beiden Steckbuchsen auf der Seite der digitalen Pins direkt nebeneinander liegen, könnte man dort sehr einfach eine LED einstecken. Ich warne ausdrücklich vor dieser Variante, da die LED ohne Widerstand betrieben wird. Dabei mache ich mir weniger Sorgen um die LED als um den Mikrocontroller. Ich habe einmal die Stärke des Stromes gemessen: Er beträgt ganze 60mA. Dieser Wert liegt 50% über dem Maximum des erlaubten Stromflusses und ist damit definitiv zu hoch. Der maximal zulässige Strom für einen einzelnen digitalen Pin des Mikrocontrollers beträgt 40mA.

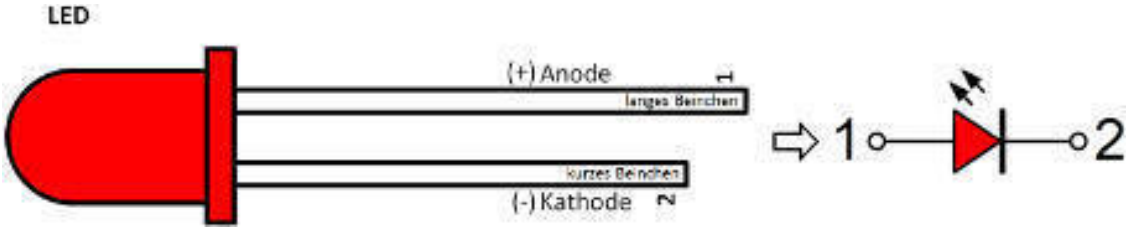
# Troubleshooting

Fehler schleichen sich schnell ein, auch wenn du glaubst, dich genau an meine Texte und Abbildungen zu halten. Aus eigener Erfahrung weiß ich, dass die Suche nach Fehlern in einem Programm oder einer Schaltung oft das Zeitaufwendigste überhaupt ist. Deshalb gebe ich dir auch immer in jedem Bastelprojekt Hinweise auf mögliche Fehlerquellen. Sollte dein Bastelprojekt also zunächst nicht das tun, was du von ihm erwartest, dann geh die Punkte durch, die ich an dieser Stelle immer aufliste.

Falls die LED nicht leuchtet, kann es mehrere Gründe dafür geben.

# Eine mögliche Verpolung?

Erinnere dich noch einmal an die beiden unterschiedlichen Anschlüsse einer LED mit der Anode und Kathode.



# Defekte LED?

Die LED ist vielleicht defekt und durch Überspannung aus vergangenen Bastelprojekten durchgebrannt. Teste sie mit einem Widerstand an einer 5V-Spannungsquelle.

# Verkabelung fehlerhaft?

Kontrolliere noch einmal die Steckleistenbuchsen, die mit der LED und dem Widerstand verbunden sind. Sind es wirklich GND und Pin 13?

## Sketch fehlerhaft?

Überprüfe noch einmal den Sketch, den du in den Editor der IDE eingegeben hast. Hast du möglicherweise eine Zeile vergessen oder falsch geschrieben und ist der Sketch wirklich korrekt übertragen worden? Hast du die geschweiften Klammerpaare zu Beginn und Ende der *setup*- und *loop*-Funktion gesetzt? Hast du auch immer das Semikolon am Ende einer Befehlszeile gesetzt? Das ist übrigens der häufigste Einsteigerfehler.

Wenn die auf dem Board befindliche LED blinkt, sollte die eingesteckte LED ebenfalls blinken, da dann der Sketch korrekt arbeitet.

Grundlagen zur Berechnung des Widerstandes

Ich habe in unserem Blink-Bastelprojekt einfach einen Widerstand von 220Ω verwendet, der für solche Schaltungen vollkommen ausreichend ist. Dennoch kommt sicherlich die Frage auf, warum das so ist und was dabei beachtet werden sollte. Die folgende Schaltung zeigt uns eine Leuchtdiode mit einem Widerstand, die an eine Spannungsquelle angeschlossen wurde.

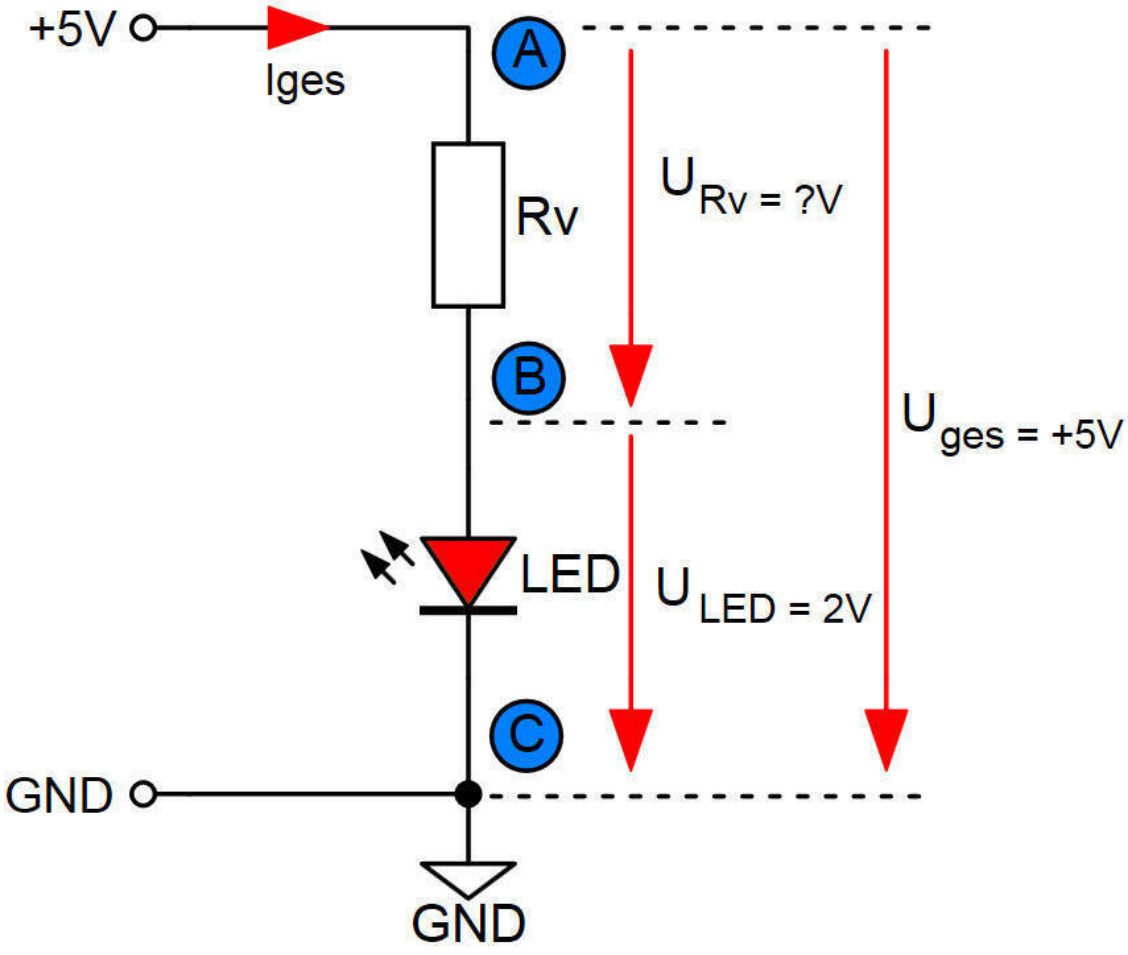



**Abb. 5:** Eine LED mit Widerstand

Zusätzlich habe ich noch ein paar Pfeile für Spannungswerte sowie den Gesamtstrom eingezeichnet. Um den Wert eines Widerstandes zu ermitteln, greifen wir auf das sogenannte

*Ohmsche Gesetz* zurück. Das Ohmsche Gesetz beschreibt die Zusammenhänge zwischen Spannung, Strom und dem Widerstand. Legen wir an einem Bauteil eine Spannung *U* an, dann verändert sich der hindurchfließende Strom *I* proportional zur Spannung. Der Quotient zwischen den beiden Größen, also Spannung und Strom, ist konstant und definiert den elektrischen Widerstand *R*. Wir können folgende Formel aufschreiben:

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

Bringe ich die Formelbuchstaben jetzt in die richtige Reihenfolge, so ergibt sich zum Bestimmen des elektrischen Widerstandes die folgende kurze Formel:

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

Doch ein wichtiger Aspekt scheint noch nicht angesprochen zu sein. Was ist überhaupt ein Widerstand? Die Elektronen, die sich durch einen Leiter bewegen, haben es mehr oder weniger leicht, ihn zu durchqueren und müssen sich gegen einen bestimmten vorherrschenden Widerstand zur Wehr setzen. Es gibt unterschiedliche Kategorien, die Aufschluss über die Leitfähigkeit eines Stoffes geben.

Isolatoren (sehr hoher Widerstand, zum Beispiel Keramik)
Schlechte Leiter (hoher Widerstand, zum Beispiel Glas)
Gute Leiter (geringer Widerstand, zum Beispiel Kupfer)
Sehr gute Leiter (Supraleitung bei sehr niedrigen Temperaturen, bei der der elektrische Widerstand auf 0 sinkt)
Halbleiter (Widerstand kann gesteuert werden, zum Beispiel Silizium oder Germanium)

Damit habe ich schon zwei entscheidende elektrische Größen ins Spiel gebracht, die in einem gewissen Zusammenhang zueinander stehen: den Widerstand *R* und die Leitfähigkeit *G*. Je höher der Widerstand, desto geringer der Leitwert und je geringer der Widerstand, desto höher der Leitwert. Folgender Zusammenhang besteht:

$$R = \frac{1}{G}$$

Der Widerstand ist der Kehrwert des Leitwertes. Ein erhöhter Widerstand ist mit einem Engpass vergleichbar, den die Elektronen überwinden müssen. Es bedeutet, dass der Stromfluss gebremst und im Endeffekt geringer wird. Stell dir vor, du läufst über eine ebene Fläche. Das Gehen bereitet dir keine großen Schwierigkeiten. Jetzt versuch, bei gleichem Kraftaufwand durch hohen Sand zu gehen. Das ist recht mühsam. Du gibst Energie in Form von Wärme ab und deine Geschwindigkeit sinkt. Ähnlichen Schwierigkeiten sehen sich die Elektronen gegenüber, wenn sie statt durch Kupfer plötzlich zum Beispiel durch Glas müssen.

Dieser zu überwindende Widerstand hat natürlich Auswirkungen. Aufgrund der verstärkten Reibung der Elektronen, beispielsweise an der Außenwand oder untereinander, entsteht Reibungsenergie in Form von Wärme, die der Widerstand nach außen abgibt.

In den meisten elektronischen Schaltungen werden spezielle Bauteile verwendet, die den Stromfluss künstlich verringern, wobei der Widerstandswert *R* in *Ohm* (*Ω*) angegeben wird. Es handelt sich dabei um extra angefertigte Widerstände (zum Beispiel Kohleschicht- oder Metallschichtwiderstände) mit unterschiedlichen Werten, die mit einer Farbcodierung versehen werden, die auf ihren jeweiligen Widerstandswert schließen lässt. Wir kommen gleich näher darauf zu sprechen.

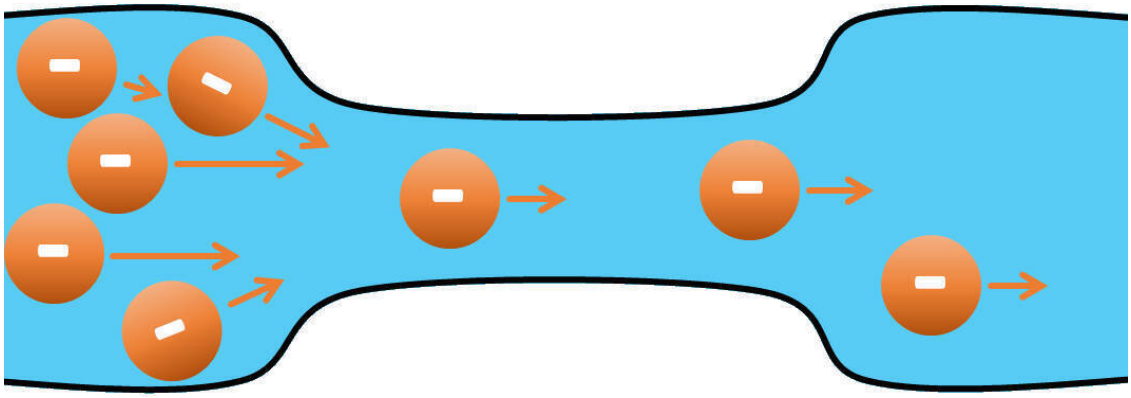


**Abb. 6:** Ein Widerstand, der den Elektronenfluss bremst

Wenn du dir Abbildung 6 mit dem Elektronenfluss ansiehst, dann könnte man zu dem Schluss kommen, dass die Elektronen auf der linken Seite eine höhere Geschwindigkeit haben als die auf der rechten Seite. Doch dem ist nicht so. Der Strom in einem geschlossenen Kreis ist immer gleich! Er wird natürlich durch den hier gezeigten Widerstand entscheidend beeinflusst, doch im Endeffekt ist der Stromfluss vor und hinter einem Widerstand immer gleich. An jeder Stelle passiert pro Zeiteinheit immer dieselbe Anzahl von Elektronen den Leiter beziehungsweise den Widerstand.

Kommen wir zurück zu unserer Schaltung, um den Widerstand zu berechnen. Wir müssen uns überlegen, wie die Werte von Spannung und Strom ermittelt werden. Das ist nicht weiter schwer. An Widerstand und LED (zwischen den Punkten A und C in Abbildung 5) liegen +5V an, denn das ist die Betriebsspannung des Arduino und sie liegt am Ausgang des jeweiligen digitalen Pins an, wenn er mit einem HIGH-Pegel angesteuert wird. An der LED, also zwischen den Punkten B und C fallen in der Regel 2V ab, was aber an der LED und deren Farbe liegt. Die Spannung am Widerstand zwischen den Punkten A und B ist demnach die Differenz von 5V und 2V, ergibt also 3V

Nun fehlt uns noch der Strom, der durch den Widerstand fließt. Wenn wir Bauteile hintereinander in einem einzigen Strompfad schalten, sprechen wir von einer Reihenschaltung, was hier offensichtlich der Fall ist. In einer Reihenschaltung ist der Strom durch alle Bauteile gleich. Ein Blick in das Datenblatt des Arduino teilt uns mit, dass der maximale Strom, den ein digitaler Pin liefern kann und darf, 40mA beträgt. Dieser Wert darf nicht überschritten werden, weil der Mikrocontroller ansonsten Schaden nimmt. Aus diesem Grund begrenzen wir den Strom mit dem eingezeichneten Widerstand Rv. Nun sollte man in der Elektronik nicht unbedingt am Limit arbeiten, sondern immer etwas unterhalb des angegebenen Grenzwertes. Zur Berechnung des Widerstandes werde ich deswegen zwei unterschiedliche Stromwerte von 5mA und 10mA zur Verdeutlichung verwenden, wobei Werte zwischen 5mA und 30mA für eine LED ausreichend sind. Sehen wir uns die entsprechenden Berechnungen und die daraus resultierenden Ergebnisse einmal an:

$$R_1 = \frac{U_{ges} - U_{LED}}{I_1} = \frac{5V - 2V}{10mA} = 300\Omega$$

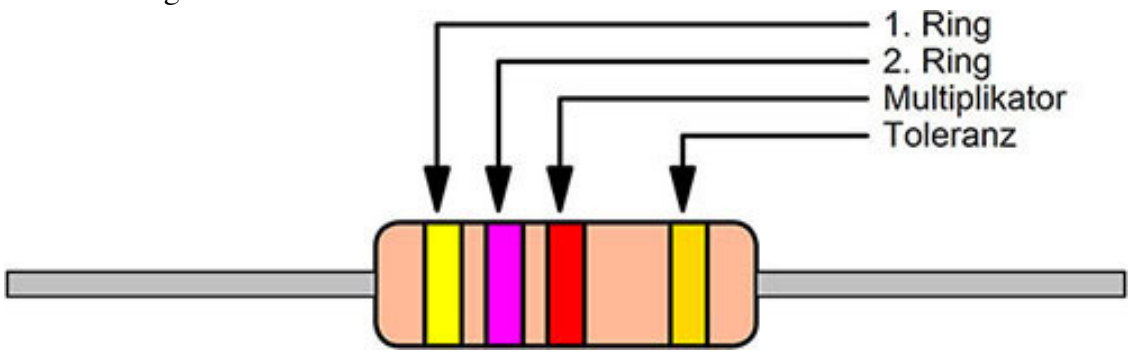
und

$$R_2 = \frac{U_{ges} - U_{LED}}{I_2} = \frac{5V - 2V}{5mA} = 600\Omega$$

Für einen passenden Widerstandswert können wir uns also Werte im Bereich zwischen 300Ω und 600Ω aussuchen. Der Ausgang des jeweiligen Arduino-Pins wird nur moderat belastet. Ein Wert von 330Ω ist für unsere Belange vollkommen ausreichend. Widerstände werden übrigens nicht in allen nur erdenklichen Größen hergestellt, sondern in unterschiedlichen E-Reihen mit bestimmten Abstufungen angeboten. Zudem sollte auf die maximale Verlustleistung geachtet werden, wobei ein Wert von ¼ Watt ok ist. Passende Sortimente sind im Handel erhältlich.

Auf der folgenden Abbildung sehen wir einen Widerstand und erkennen die unterschiedlichen Farbringe. Diese Farbringe stellen eine Farbcodierung dar.


Was haben sie zu bedeuten und wie ist der Code zu entziffern? Sehen wir uns das anhand der folgenden Grafik genauer an:




Ein Widerstand besitzt in der Regel vier Farbringe. Man hat sich für diese Form der Beschriftung entschieden, da für den Aufdruck von Zahlenwerten einfach zu wenig Platz vorhanden ist. Um die Farbringe zu dekodieren, muss der Widerstand so positioniert werden, dass die drei Ringe, die am dichtesten beieinander liegen, sich auf der linken Seite befinden. Mit welchem Widerstandswert haben wir es denn hier zu tun?

| 1. Ring | 2. Ring | 3. Ring | 4. Ring | Widerstandswert |
|---|---|---|---|---|
| Gelb: Wert 4 | Violett: Wert 7 | Rot: Wert 100 | Gold: +/- 5% | 4,7KΩ |

Wenn wir uns die ermittelten Werte hintereinander aufschreiben, ist das Ergebnis leicht abzulesen: 4700 entsprechen 4,7KΩ. Der Toleranzwert gibt Aufschluss über die Güte: Je kleiner er ist, desto genauer hält sich der Widerstandswert an seine Vorgaben. Wie bin ich aber zu diesen Werten gekommen? Ganz einfach! In der folgenden Tabelle finden wir die unterschiedlichen Farben mit den entsprechenden Werten zur Berechnung eines Widerstandswertes:

| Tabelle 3: Farbcodierung der Widerstände | | | | |
|---|---|---|---|---|
| Farbe | 1. Ring | 2. Ring | 3. Ring | 4. Ring |

| | | | | |
|---|---|---|---|---|
| Schwarz | x | 0 | $10^0 = 1$ | |
| Braun | 1 | 1 | $10^1 = 10$ | +/- 1% |
| Rot | 2 | 2 | $10^2 = 100$ | +/- 2% |
| Orange | 3 | 3 | $10^3 = 1.000$ | |
| Gelb | 4 | 4 | $10^4 = 10.000$ | |
| Grün | 5 | 5 | $10^5 = 100.000$ | +/- 5% |
| Blau | 6 | 6 | $10^6 = 1.000.000$ | +/- 0,25% |
| Violett | 7 | 7 | $10^7 = 10.000.000$ | +/- 0,1% |
| Grau | 8 | 8 | $10^8 = 100.000.000$ | +/- 0,05% |
| Weiß | 9 | 9 | $10^9 = 1.000.000.000$ | |
| Gold | - | - | $10^{-1} = 0,1$ | +/- 5% |
| Silber | - | - | $10^{-2} = 0,01$ | +/- 10% |

Die Schaltzeichen, also die Symbole, die in den Schaltplänen verwendet werden, sehen wie folgt aus, wobei sich Unterschiede zwischen der deutschen DIN- und der amerikanischen ANSI-Norm ergeben:


**Abb. 7:** Die Schaltzeichen eines Widerstandes

Auf das Ohm-Zeichen (Ω) wird bei der Darstellung in der Regel verzichtet, wobei bei Werten, die kleiner als 1Kilo-Ohm (1000Ohm) sind, lediglich die nackte Zahl genannt wird und bei Werten ab 1KΩ ein *K* für Kilo bzw. ab 1MΩ ein *M* für Mega angehängt wird. Hier ein paar Beispiele:

| Tabelle 4: Ein paar markante Widerstandswerte | |
|---|---|
| Wert | Kennzeichnung |
| 220Ω | 220 |
| 1000Ω | 1K |
| 4700Ω | 4,7K oder 4K7 |
| 2,2MΩ | 2,2M |

Die maximale Verlustleistung der Widerstände, die wir für unsere Arduino-Projekte benötigen, beträgt 1/4-Watt. Es handelt sich in allen Fällen um Kohleschichtwiderstände. Sie sind auch billiger als die Kollegen aus der Metallfilmabteilung. Widerstände gibt es in allen möglichen Größen und Farben und je größer beziehungsweise dicker sie sind, desto größer ist auch die Verlustleistung.

So, das war dein erstes Projekt, das du mit dem Arduino gemacht hast! Du hast eine LED und einen passenden Widerstand an das Arduino-Board angeschlossen und hast über den Sketch die LED in einem von dir bestimmten Rhythmus blinken lassen. Wenn du die LED erfolgreich zum Blinken gebracht hast, kannst du bereits stolz auf dich sein! Wenn du es dir zutraust, dann wage dich auch noch an folgendes Thema heran. Das Thema PWM wird dir immer wieder in deiner Bastlerkarriere begegnen. Wenn du es jetzt nicht auf Anhieb verstehst, beschäftige dich später erneut mit diesem Thema.

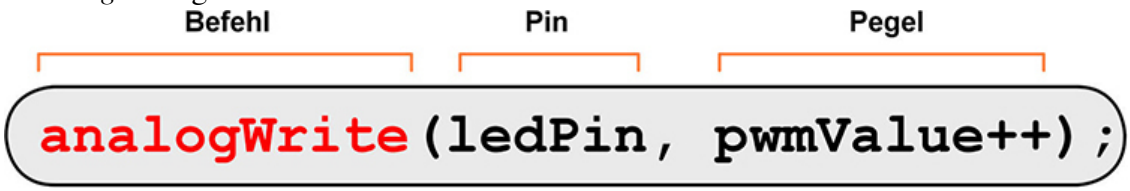
Die PWM-Ansteuerung

Kommen wir zur Ansteuerung einer LED über PWM. Was das ist, habe ich kurz in Kapitel 1 erwähnt. Nun ist es an der Zeit, es in die Tat umzusetzen. Wir wollen eine LED über eine analoge Ansteuerung langsam aufblenden und sie beim Erreichen des Maximalwertes schlagartig verlöschen lassen, bevor das Spiel von vorne beginnt. Dabei handelt es sich quasi um ein sanftes Blinken im Gegensatz zum abrupten Blinken in der vorherigen Schaltung. Die Ansteuerung erfolgt dabei über die digitalen Pins, die eine Tilde vor der Nummerierung besitzen. Das sind die Pins D3, D5, D6, D9, D10 und D11. Für unser Beispiel habe ich Pin 3 verwendet. Der Sketch-Code sieht folgendermaßen aus:

int ledPin = 3; // Variable mit Pin 3 deklarieren + initialisieren int pwmValue = 0; // Variable für PWM deklarieren + initialisieren void setup() { /* Kein Code erforderlich */ } void loop() { analogWrite(ledPin, pwmValue++); // LED mit PWM-Wert ansteuern delay(10); // Kurze Pause if(pwmValue > 255) pwmValue = 0; // Wenn PWM-Wert > 255 -> // auf 0 setzen }

# Den Code verstehen

Zu Beginn deklarieren und initialisieren wir zwei globale Variablen, die die Namen *ledPin* und *pwmValue* aufweisen und vom ganzzahligen Datentyp *int* (int = Integer) sind. Sehen wir uns den Befehl *analogWrite* genauer an.




Er nimmt zwei Argumente auf, wobei das erste den Pin bestimmt und das zweite den PWM-Wert, der sich im Bereich von 0 bis 255 bewegen darf. Über

pwmValue++

wird die Variable *pwmValue* nach der Verwendung über die beiden nachfolgenden Pluszeichen um den Wert 1 erhöht. Dieser Vorgang wird in der Programmierung *Inkrementieren* genannt. Konventionell würde man das gleiche Verhalten mit der folgenden zusätzlichen Codezeile erreichen:

pwmValue = pwmValue + 1;

Würden wir das Inkrementieren der Variablen *pwmValue* ohne nachfolgende Überwachung in der *loop*-Endlosschleife laufen lassen, wäre irgendwann ein unzulässiger Wert größer 255 erreicht. Um das zu verhindern, erfolgt über die *if*-Anweisung eine Abfrage auf Werte größer 255. Wird diese Bewertung positiv beantwortet, erfolgt die Ausführung des nachfolgenden Befehls:

pwmValue = 0;

Darüber erfolgt eine Rücksetzung auf den Anfangswert 0, was zur Folge hat, dass die LED beim nächsten Aufruf des *analogWrite*-Befehls erlischt. Ändern wir doch den Code derart ab, dass die LED nicht schlagartig erlischt, sondern langsam abblendet.

## Wichtig!

Für die PWM-Ansteuerung eines digitalen Pins mit dem *analogWrite*-Befehl ist keine vorherige Programmierung über den *pinMode*-Befehl erforderlich, um diesen als Ausgang festzulegen. Wir sehen, dass die *setup*-Funktion keinen derartigen Befehl enthält und komplett frei von Code ist.

Weitere Informationen zu PWM sind unter der folgenden Adresse zu finden:

http://www.arduino.cc/en/Tutorial/PWM
https://www.arduino.cc/en/Reference/AnalogWrite

# Gut zu wissen

Wir haben gesehen, wie man mit ein paar Codezeilen eine LED blinken lässt. Es geht jedoch noch ein wenig kürzer:

int ledPin = 13; // Variable mit Pin 13 deklarieren + initialisieren void setup() { pinMode(ledPin, OUTPUT); // Digitaler Pin 13 als Ausgang } void loop() { digitalWrite(ledPin, ! digitalRead(ledPin)); // Toggeln der LED delay(1000); // Eine Sekunde warten }

Die entscheidende Zeile lautet:

digitalWrite(ledPin, !digitalRead(ledPin));

Wir nutzen an dieser Stelle zwei neue Konstrukte. Da ist zum einen der Befehl zum Ermitteln des Status eines digitalen Pins über *digitalRead*.


Außerdem nutzen wir den sogenannten *NOT-Operator* mithilfe des Ausrufezeichens (!), der dazu verwendet wird, ein Ergebnis in das Gegenteil zu verkehren. In der Programmiersprache C/ C++ existiert kein eigener Datentyp für logische Werte wie wahr oder falsch und deshalb wird ein Wert vom Datentyp *int* verwendet. Wir haben in der Tabelle 2 über die Konstanten gesehen, dass ein LOW-Pegel den Wert 0 und ein HIGH-Pegel den Wert 1 besitzt. Über den NOT-Operator wird zwischen beiden hin- und hergewechselt, was auch *Toggeln* genannt wird. Hinweise zu booleschen Operatoren sind unter dem folgenden Link zu finden:

https://www.arduino.cc/en/Reference/Boolean

Wenn du weiter in die Thematik der Widerstände eintauchen möchtest, dann rate ich dir einen Blick in mein Buch *Elektronik verstehen durch spannende Experimente – Analog- und Digitaltechnik*, ISBN 978-3-946496-23-6. Dort werden unter anderem Reihenschaltungen, Parallelschaltungen und Gemischtschaltungen von Widerständen angesprochen und detailliert erklärt.

# Was haben wir gelernt?

Du hast die korrekte Deklaration und die korrekte Initialisierung einer globalen Variable kennen gelernt.

Es wurde die grundlegende Sketch-Struktur vermittelt.

Die Datenübertragungsrichtung eines einzelnen Pins hast du mit dem Befehl pinMode auf OUTPUT gesetzt, so dass du ein digitales Signal (HIGH bzw. LOW) über den Befehl digitalWrite zum Ausgang schicken konntest, an der die LED angeklemmt war.

Über den Befehl delay hast du eine zeitliche Unterbrechung des Sketches eingeleitet, damit die LED eine bestimmte Zeit an beziehungsweise aus war.

Mithilfe des NOT-Operators in Verbindung mit dem Befehl digitalRead haben wir eine verkürzte Variante des Codes für das Blinken der LED kennengelernt.

Wenn man eine LED betreiben möchte, ist ein entsprechend dimensionierter Widerstand – auch Vorwiderstand genannt – unerlässlich.

Wir haben gesehen, wie sich ein Vorwiderstand mit Hilfe des Ohmschen Gesetzes berechnen lässt.

Mithilfe der Farbtabelle ist es sehr einfach, einen Widerstandswert anhand der aufgedruckten Farbringe zu ermitteln.

Wir haben die Ansteuerung einer LED über einen PWM-Pin gesehen.

# Workshop zur blinkenden LED

Ich möchte dir am Ende dieses Projektes die Aufgabe stellen, den Sketch so abzuändern, dass du die Zeit, in der die LED leuchtet beziehungsweise aus ist, in zwei Variablen auslagerst. Darüber kann der sogenannte Tastgrad eingestellt werden. Er beschreibt das Verhältnis von *Impulsdauer* und *Periodendauer*. Das Ergebnis wird meist in Prozent angegeben. Auf dem folgenden Impulsdiagramm siehst du die unterschiedlichen Zeiten für *t* bzw. *T*:


**Abb. 8:** Der Verlauf des Pegels am digitalen Ausgang Pin 13 bei einem Tastgrad von 25%
t = Impulsdauer
T = Periodendauer
Die Formel zur Berechnung des Tastgrades lautet:

$$Tastgrad = \frac{t}{T}$$

Programmiere den Sketch so, dass die LED 0,5s leuchtet und 1,5s dunkel ist. Der Tastgrad würde sich wie folgt berechnen:

$$Tastgrad = \frac{t}{T} = \frac{500ms}{2000ms} = 0{,}25 \triangleq 25\%$$

Das entspricht also einem Tastgrad von 0,25 und im Hinblick auf die gesamte Periodendauer leuchtet die LED also zu 25%.

# Bastelprojekt 2: Arduino-Low-Level-Programmierung

Der Arduino Uno besitzt als Herzstück einen Mikrocontroller, der die zentrale Recheneinheit auf dem Board ist. Neben der Spannungsversorgung sind zur Kommunikation mit der Außenwelt natürlich weitere Anschlüsse vorhanden. Da gibt es die analogen und digitalen Ein- beziehungsweise Ausgänge – wir haben das schon in Kapitel 1 gesehen –, die abgefragt und auf die Einfluss genommen werden kann. Da ein Mikrocontroller auf unterster Ebene digital arbeitet, werden die Informationen als HIGH- bzw. LOW-Pegel gespeichert und verarbeitet. Diese finden in bestimmten Speicherbereichen ihr Zuhause, die man *Register* nennt. In diesem Bastelprojekt werden wir die ganze Sache auf unterster Ebene beleuchten, was zwar knifflig anmutet, doch zum Verständnis der Grundlagen sicherlich hilfreich ist. Aber keine Bange, ich erkläre es Schritt für Schritt.

Was um Himmels Willen ist denn ein Register? Wenn es um die Programmierung eines Mikrocontrollers geht, dann besitzt er zur internen Verwaltung und Speicherung von Werten bestimmte Speicherstellen. Die meisten davon haben einen Schreib- und Lesezugriff, was bedeutet, man kann dort etwas ablegen und auch wieder auslesen. Andere können lediglich ausgelesen werden und stellen somit nur einen Lesezugriff zur Verfügung. Derartige Speicherbereiche werden *Register* genannt. Wenn wir gleich die sogenannten Ports kennenlernen, dann handelt es sich um Register

# Die Zugänge des Mikrocontrollers

Um mit einem Mikrocontroller in Verbindung zu treten, muss es eine Möglichkeit geben, Signale zu senden und zu empfangen. Aus diesem Grund sind bestimmte Zugänge geschaffen worden, die *Ports* genannt werden. Port kommt aus dem Lateinischen von *porta*, was Tür beziehungsweise Zugang bedeutet. Auf dem Arduino-Board befinden sich mehrere nebeneinanderliegende Pins, die teilweise zu Funktionsgruppen zusammengefasst wurden. Sie werden *Ports* genannt und haben eine Datenbreite von acht Bits. Auf der folgenden Abbildung 1 sehen wir einige dieser Ports:

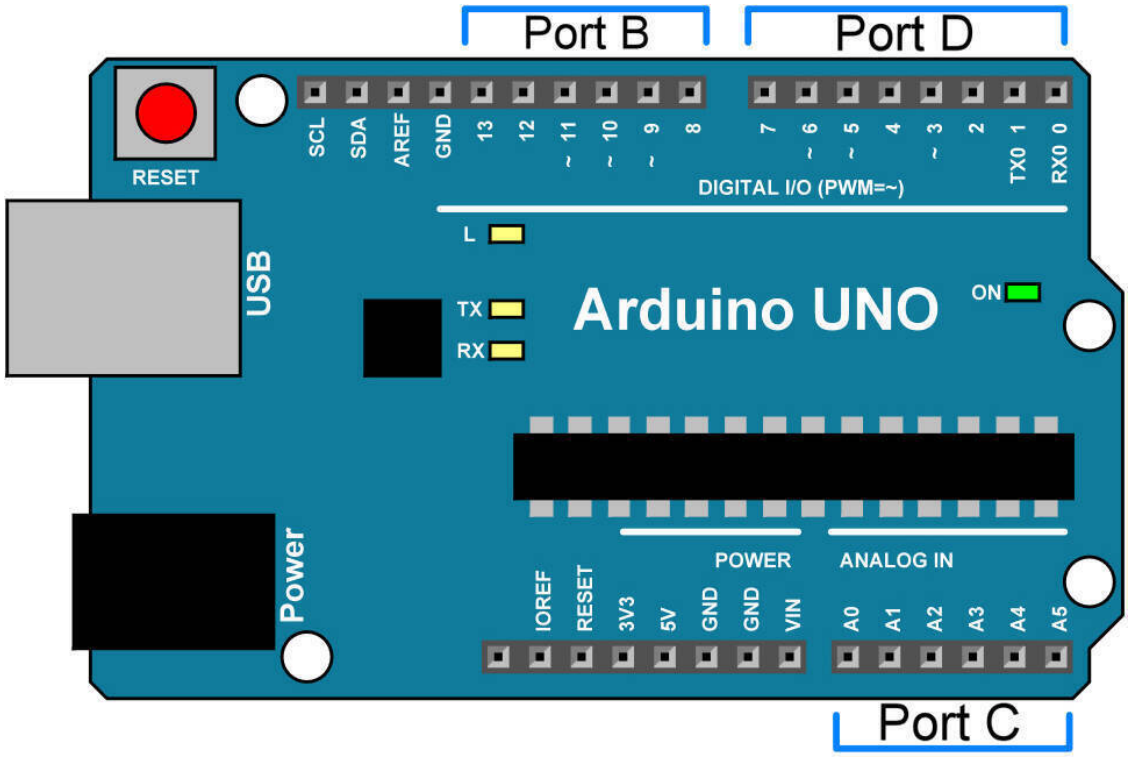



**Abb. 1:** Die Ports des Arduino Uno
Die farbig markierten Ports sind mit den folgenden Pins verbunden:

Port B: Digitale Pins 8 bis 13
Port C: Analoge Eingänge (A0 bis A5)
Port D: Digitale Pins 0 bis 7

Da jeder der drei genannten Ports individuell programmiert werden kann, sind zusätzliche Register erforderlich, die für die Konfigurationen notwendig sind. Der kleine Buchstabe *x* wird später durch den benötigten Port ersetzt. Die Ports haben ebenfalls eine Datenbreite von acht Bits und lauten wie folgt:

DDRx (Data Direction Register): Definiert einen Pin entweder als Eingang (0) oder als Ausgang (1): read/write.

PORTx (Pin Output Value): Setzt den Pin-Status auf HIGH oder LOW: write.

PINx (Pin Input Value): Liest den Wert eines Pins: read only.

http://www.arduino.cc/en/Reference/PortManipulation

# Die Programmierung eines Ports

Sehen wir uns die Sache einmal im Detail an, wobei ich nochmal erwähne, dass diese Art des Arbeitens schon in Richtung *Fortgeschrittenenprogrammierung* geht, doch zum tieferen Verständnis trägt dieses Wissen allemal bei. Werfen wir doch einmal einen Blick auf Port B mit den digitalen Pins 8 bis 13.

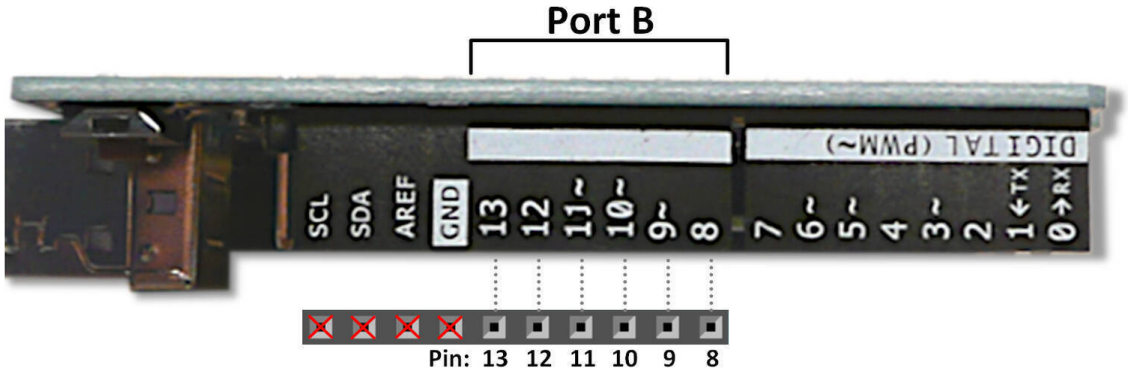



**Abb. 2:** Port B

An den Buchsen, die auch *Header* genannt werden, sind nicht alle Pins einem Port zugeordnet. Wir erkennen das hier an den kleinen diagonalen Kreuzen. Auf der Seite der Header befinden sich Beschriftungen, die Hinweis auf die Funktion geben, was in meinen Augen eine sehr nützliche und hilfreiche Sache ist. Die vier Anschlüsse von links gesehen können nicht für unsere Vorhaben verwendet werden, da sie dem $I^2C$-Bus, der Spannungsversorgung und der Masse zugeordnet sind. Damit wir auf die digitalen Pins 8 bis 13 zugreifen können, müssen entsprechende Register vorbereitet werden. Doch zuvor sollten wir uns im Klaren darüber sein, welche Pins als Ein- und welche als Ausgänge arbeiten sollen. Wir machen es uns einfach und legen Folgendes fest:

Pin 8, 9 und 10: Eingänge
Pin 11, 12 und 13: Ausgänge

Ich habe anfangs schon die Register erwähnt, die zur Konfiguration und zur Ansteuerung der Ports verwendet werden. Auf der folgenden Abbildung 3 sehen wir zwei Register mit acht Bit Datenbreite, die diese Funktion übernehmen, wobei wir später in diesem Projekt noch die Bekanntschaft mit einem weiteren Register machen:

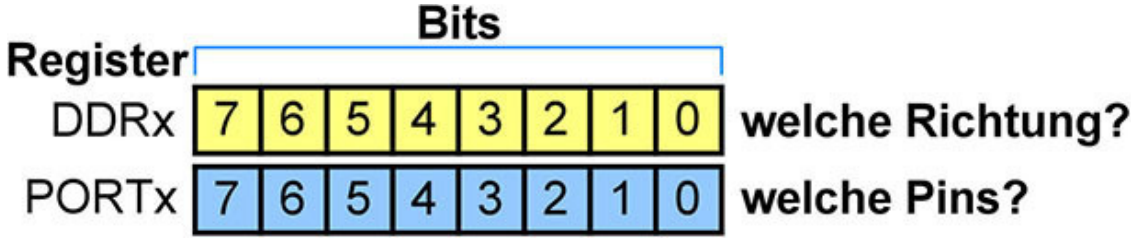



**Abb. 3:** Die Register DDRx und PORTx

Sie haben die Bezeichnung *DDRx* und *PORTx*. Fangen wir mit DDRx an, das für die Datenflussrichtung verantwortlich ist, wobei das x für den jeweiligen Port steht. In unserem Fall also DDRB. Wir wissen, dass Eingänge mit dem Wert 0 und Ausgänge mit dem Wert 1 konfiguriert werden.

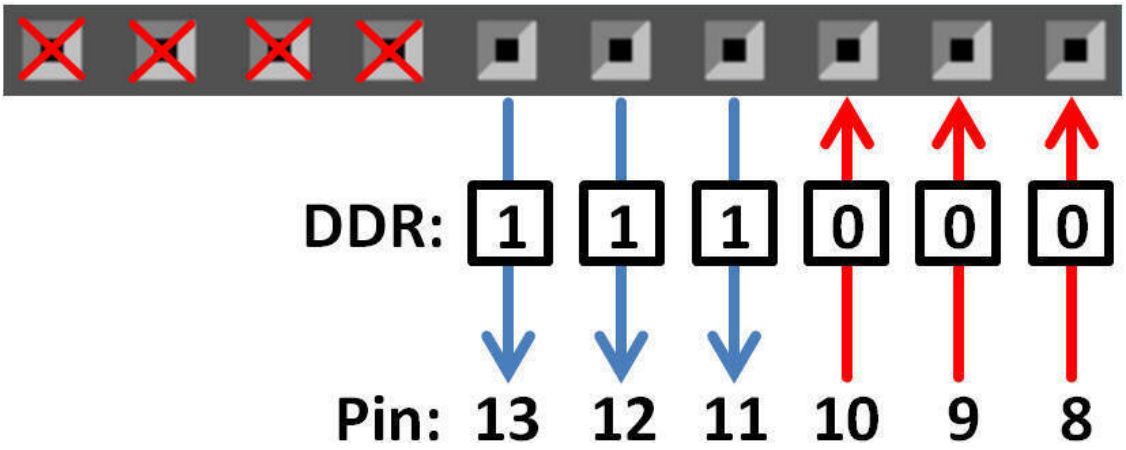



**Abb. 4:** Die Konfiguration von Port B

Die Pfeile geben die Datenflussrichtung an und wir müssen die Programmierung, wie im nachfolgenden Sketch zu sehen, in die Entwicklungsumgebung eingeben. Ich möchte noch einmal darauf hinweisen, was ich im letzten Bastelprojekt 1 bereits ausführlich dargestellt habe: Die *setup*-Funktion wird einmalig zum Sketch-Start ausgeführt und die *loop*-Funktion kontinuierlich in einer Endlosschleife:

void setup() { DDRB = 0b11111000; // Pin 8, 9, 10 als INPUT. Pin 11, 12, 13 als OUTPUT PORTB = 0b00111000; // Pin 11, 12, 13 auf HIGH-Pegel setzen } void loop() {/* leer */}

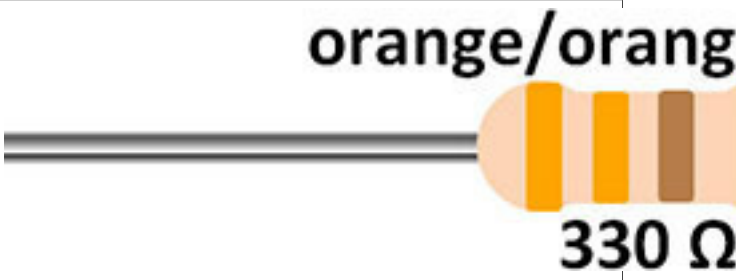
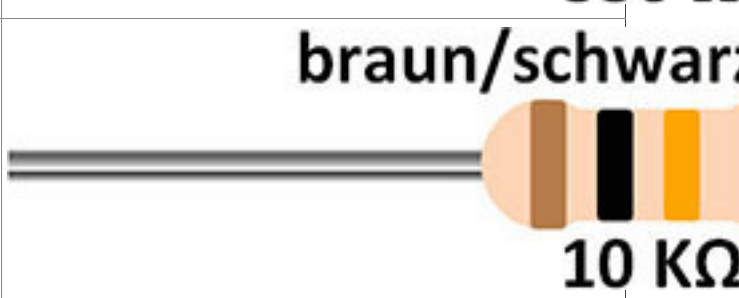
In diesem Beispiel befindet sich nur in der *setup*-Funktion der Quellcode, denn wir benötigen lediglich eine einmalige Ausführung und deswegen ist die *loop*-Funktion leer. Die Zuweisung eines Wertes an das jeweilige Register kann im Binärformat erfolgen, was die Sache in meinen Augen sehr vereinfacht, denn auf diese Weise ist sofort ersichtlich, wie die einzelnen Pins des Ports arbeiten. Durch das Vorangestellte *0b* erreichen wir die Interpretation des nachfolgenden Wertes als Binärzahl:

DDRB = 0b11111000; // Entspricht dem Dezimalwert 248

# Was wir brauchen

Für dieses Bastelprojekt benötigen wir folgende Bauteile:

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED grün 6x |  |
| Widerstand 330Ω 6x |   |
| Widerstand 10KΩ 3x |   |
| Mikrotaster 3x |  |

Der Schaltplan zur Ansteuerung der LEDs sieht wie folgt aus:

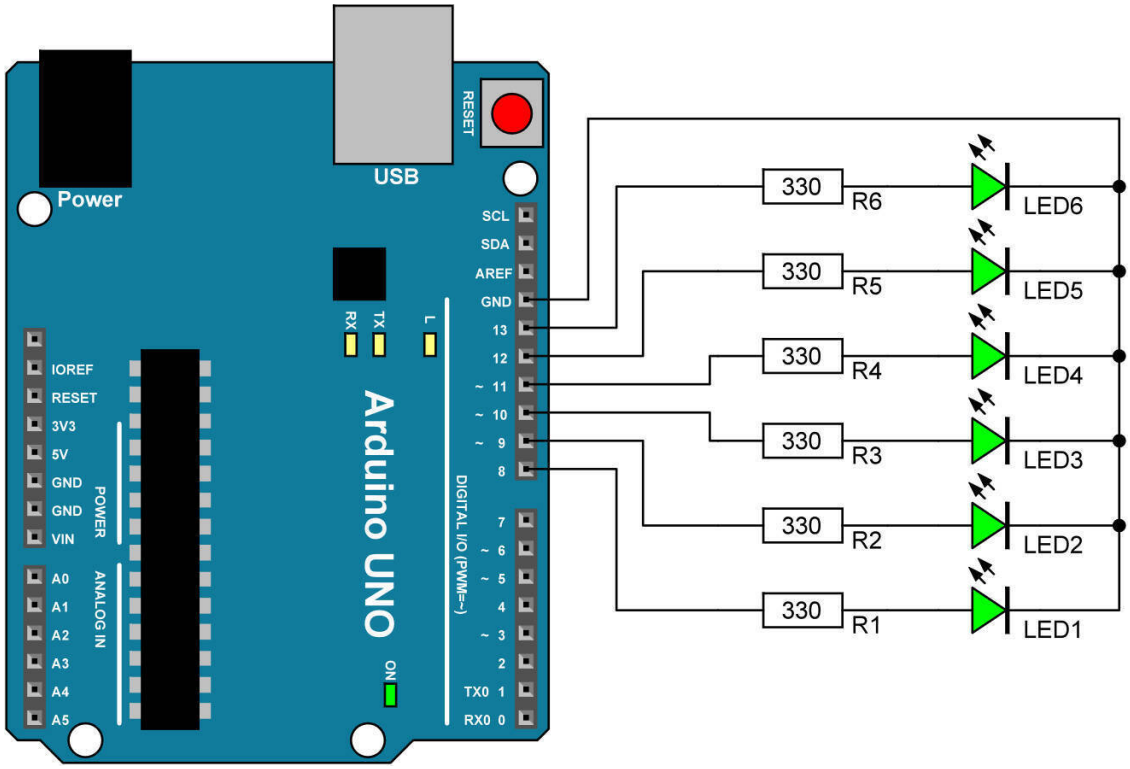



**Abb. 5:** Der Schaltplan mit 6 LEDs

Der Versuchsaufbau auf einem Arduino Discoveryboard könnte wie folgt aussehen:

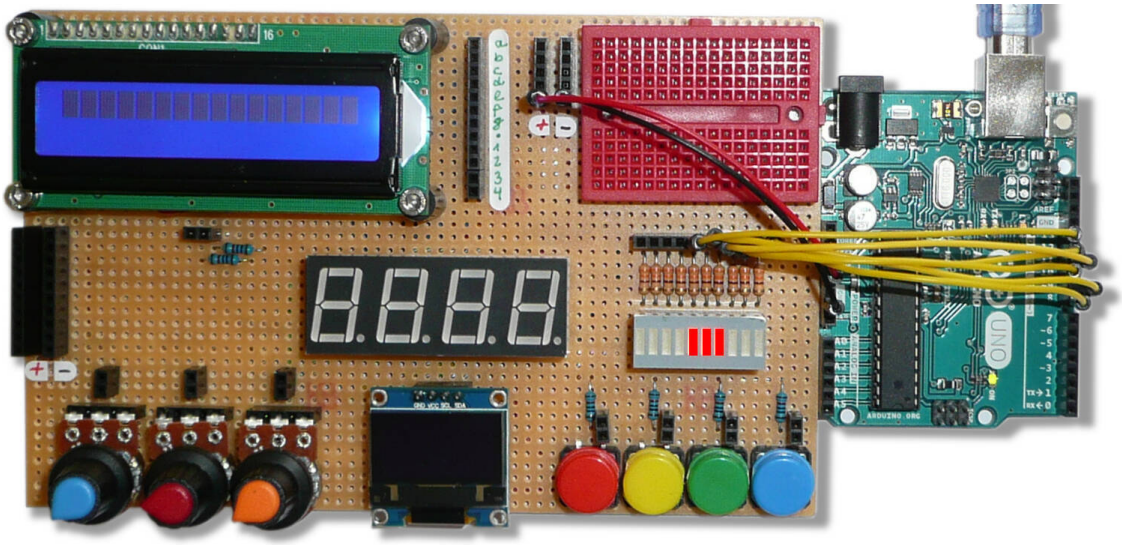


**Abb. 6:** Der Versuchsaufbau mit sechs Leuchtdioden auf dem Arduino Discoveryboard

Auf diese Weise kann mit unterschiedlichen Werten in der Ansteuerung gespielt werden und das Ergebnis ist sofort ersichtlich. Es gilt zu beachten, dass lediglich die Pins 11, 12 und 13 als Ausgänge konfiguriert wurden. Der Aufbau zeigt jedoch sechs Leuchtdioden an den Pins 8 bis 13. Was muss geändert werden, damit alle sechs Leuchtdioden angesteuert werden können? Die Antwort ist sicherlich einfach. Doch halt! Was ist überhaupt eine Leuchtdiode und was ist bei der Ansteuerung zu beachten? Eine Leuchtdiode – auch kurz *LED* (Light Emitting Diode) genannt – ist ein Halbleiterbauelement, das Licht einer bestimmten Wellenlänge abgibt und abhängig vom verwendeten Halbleitermaterial ist. Wie der Name *Diode* schon vermuten lässt, ist beim Betrieb auf die Stromrichtung zu achten, denn nur in Durchlassrichtung sendet die LED Licht aus. Bei

entgegengesetzter Polung geht die LED nicht kaputt, sie bleibt aber einfach dunkel. Es ist unbedingt darauf zu achten, dass eine LED *immer* mit einem richtig dimensionierten Vorwiderstand betrieben wird. Andernfalls leuchtet sie einmal in einer beeindruckenden Helligkeit auf und dann nie wieder. Wie du den Wert des Vorwiderstands bestimmst, hast du schon im Bastelprojekt 1 gelernt. Ein Wert von 330Ω ist da ein guter Wert. Genau wie bei einer Diode gibt es bei der Leuchtdiode zwei Kontakte, von denen einer die *Anode* und der andere die *Kathode* ist. Das Schaltzeichen sieht ähnlich aus und hat zusätzlich zwei Pfeile, die das ausstrahlende Licht andeuten:

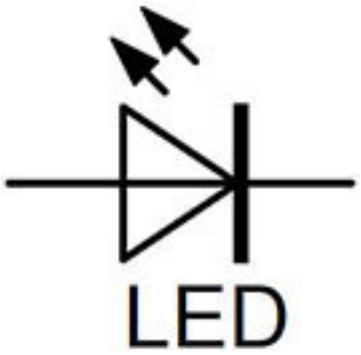



**Abb. 7:** Die Schaltzeichen einer Leuchtdiode

Kommen wir zurück zu unserer ursprünglichen Konfiguration, wo die Pins 8, 9 und 10 als Eingänge konfiguriert sind. Wie testen wir das Verhalten? Dazu muss ich ein wenig ausholen. In der Digitaltechnik gibt es nichts Schlimmeres als einen Schaltkreis, der als Eingang definiert ist und an dem nichts angeschlossen wurde. Wie ist das zu verstehen? Sehen wir uns die folgenden Schaltungen etwas genauer an:

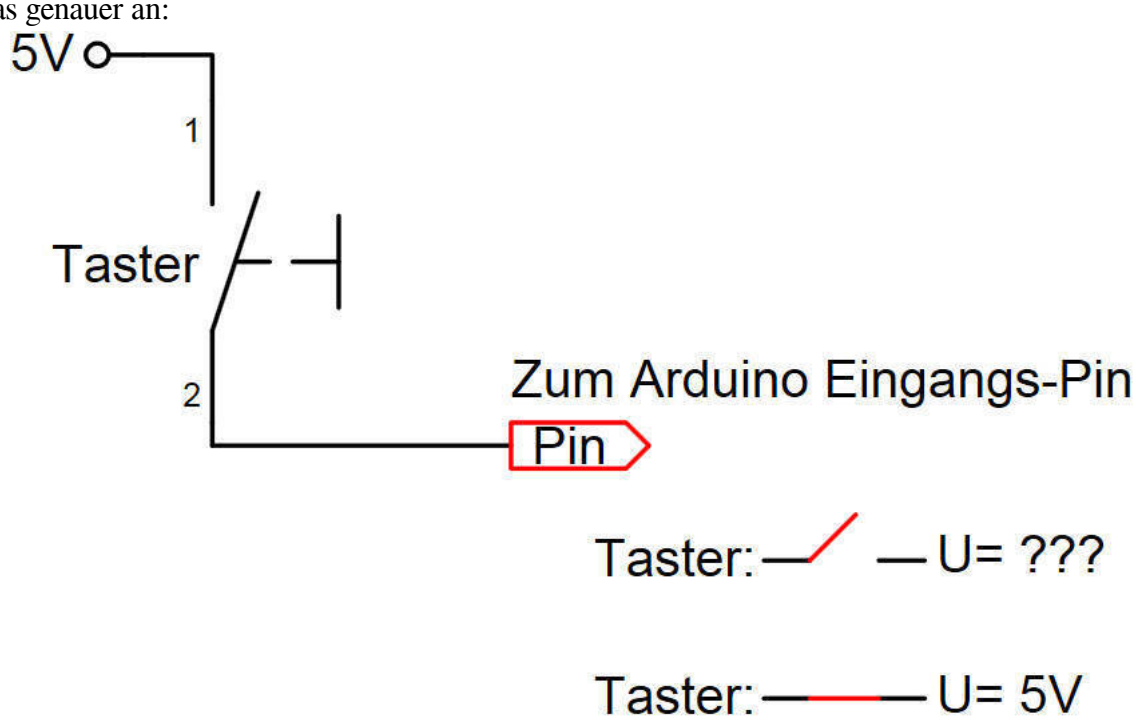



**Abb. 8:** Ein offener Eingang

Ist der Taster geschlossen, liegen +5V am Eingangs-Pin an und öffnen wir den Taster, sollten man meinen, dass nun 0V dort anliegen. Das stimmt jedoch nicht, denn über den offenen Eingang, dem kein definierter Pegel zugewiesen wurde, können Störsignale einfließen. Es reicht schon aus, wenn mit dem Finger der Eingangs-Pin berührt wird, um einen sich ständig wechselnden Pegel

zu erreichen. Aus diesem Grund verwenden wir die folgende Schaltung mit einem sogenannten *Pulldown*-Widerstand.

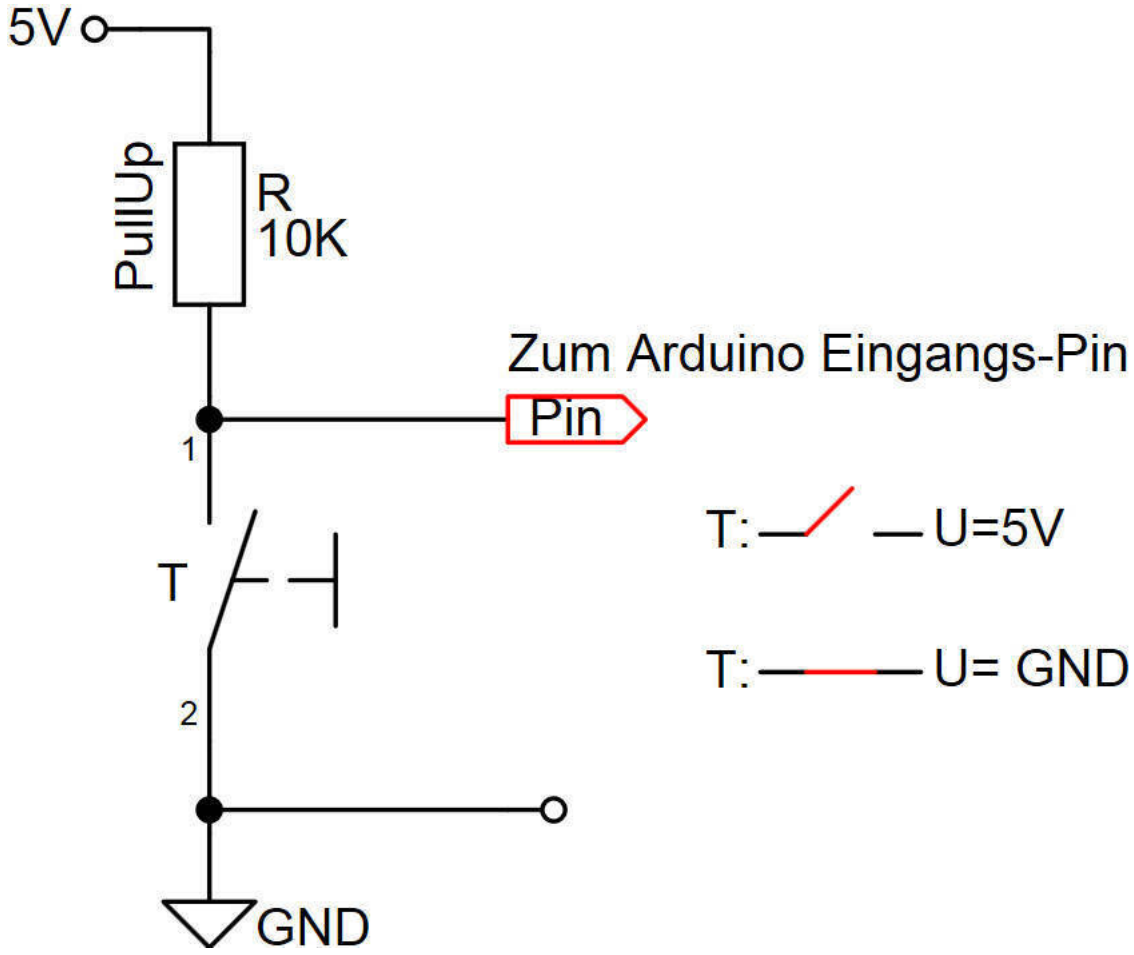



**Abb. 9:** Ein offener Eingang mit Pulldown-Widerstand

Ist der Taster geschlossen, ändert sich zur vorherigen Situation nichts und die +5V liegen am Eingangs-Pin an beziehungsweise fallen über dem Pulldown-Widerstand von 10KΩ nach Masse ab. Wird der Taster losgelassen, wird über den Widerstand das Massepotential an den Eingang geleitet, so dass dort ein LOW-Pegel von 0V anliegt. Wir haben also in beiden Tastersituationen ein definiertes Signal am Eingangs-Pin vorherrschen. Für das nun folgende Bastelprojekt wird wieder das Arduino Discoveryboard genutzt, das über fest verdrahtete Pulldown-Widerstände verfügt. Der Schaltplan für das kommende Bastelprojekt sieht wie folgt aus:

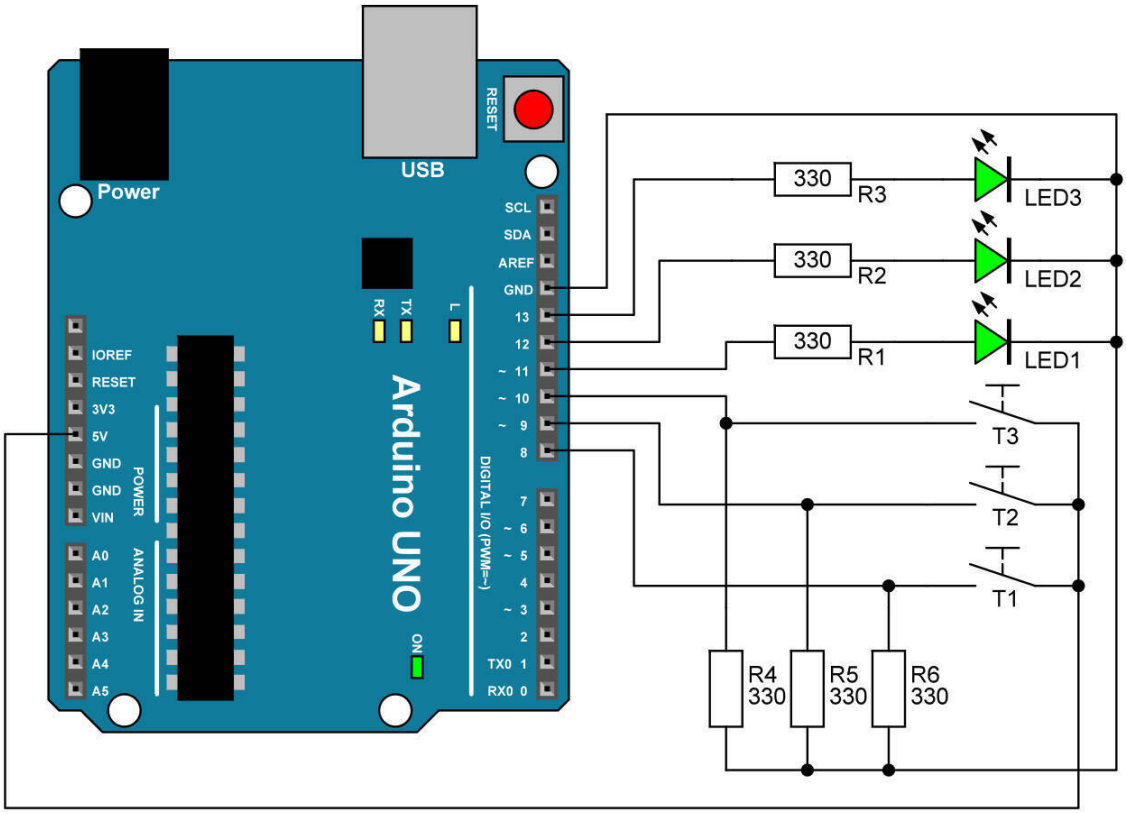



**Abb. 10:** Der Schaltplan mit drei LEDs und drei Tastern

Sehen wir uns den entsprechenden Versuchsaufbau mit den zusätzlichen drei Tastern an:

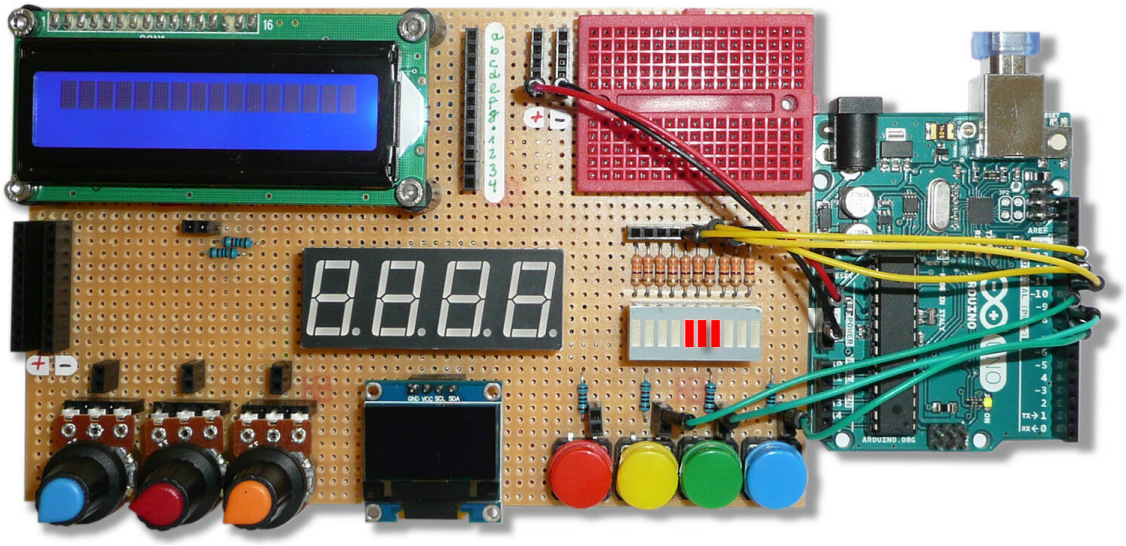


**Abb. 11:** Der Versuchsaufbau mit zusätzlichen Tastern

Natürlich müssen wir die Programmierung anpassen:

void setup() { DDRB = 0b11111000; // Pin 8, 9, 10 als INPUT. Pin 11, 12, 13 als OUTPUT PORTB = 0b00111000; // Pin 11, 12, 13 auf HIGH-Pegel setzen Serial.begin(9600); // Serielle Schnittstelle mit 9600 Baud vorbereiten } void loop() { // Binäre Ausgabe von Register PINB über // die serielle Schnittstelle Serial.println(PINB, BIN); delay(1000); // Pause von 1 Sekunde }

Bevor wir den Code besprechen, müssen wir noch ein weiteres Register besprechen. Es nennt sich PINx, wobei das *x* wieder für den entsprechenden Port steht:

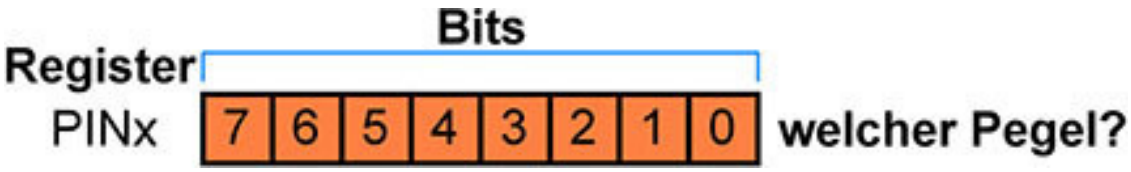



**Abb. 12:** Das Register PINx

Bei dem Register PINB (Input Pin Address) handelt es sich quasi um ein Statusregister, das den Zustand des Ports widerspiegelt. Es kann nur lesend darauf zugegriffen werden. Wir verwenden es, um den Status aller Pins auf dem sogenannten *Serial Monitor* anzuzeigen. In der *setup*-Funktion wird die serielle Schnittstelle mit ihrer Übertragungsrate von 9600 Baud initialisiert. Die spätere Ausgabe erfolgt innerhalb der *loop*-Funktion über *println* (Print Linefeed) mit dem zusätzlichen Parameter *BIN*, der eine binäre Ausgabe des Wertes erzwingt.

Der sogenannte *Serial Monitor* wird über ein Symbol in der Entwicklungsumgebung geöffnet, das ich rot umrandet habe:


**Abb. 13:** Das Öffnen des Serial Monitors

Nach einem Mausklick darauf öffnet sich der Monitor:

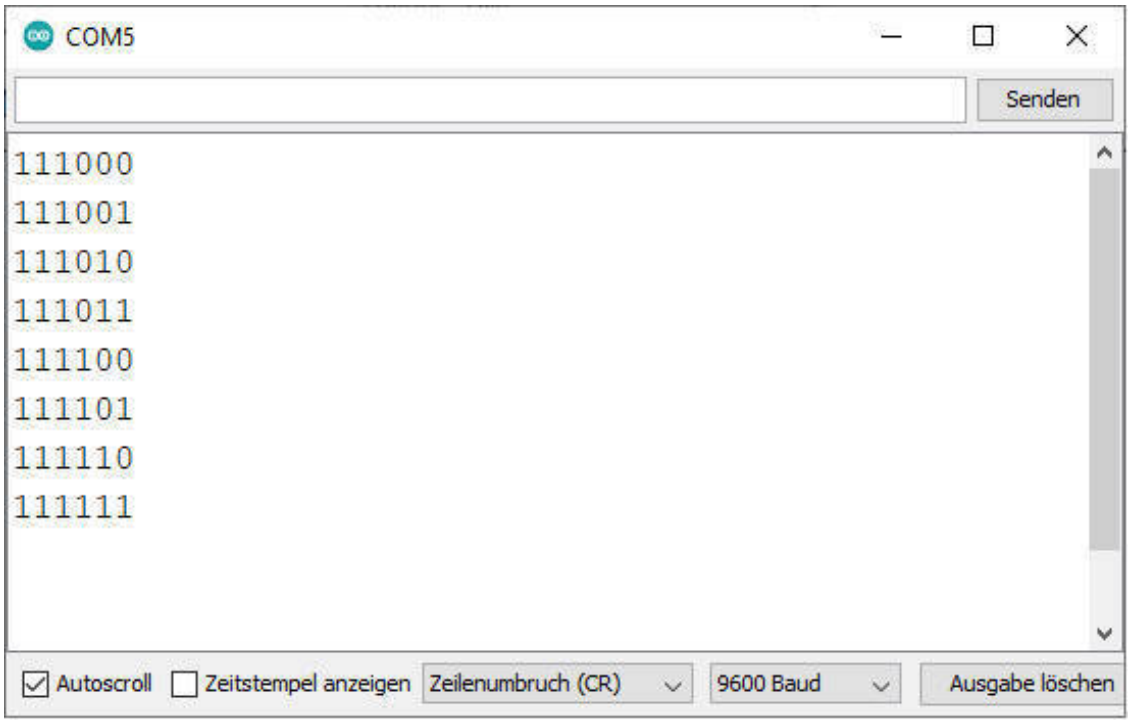



**Abb. 14:** Das Fenster des Serial Monitors

Die drei niedrigsten Bits auf der rechten Seite im Serial Monitor spiegeln den Status der drei Taster wider. Welche Taster oder welchen Taster habe ich wohl gedrückt, um die hier gezeigten Bitkombinationen zu erreichen?

Was ist beim Input-Register PINx zu beachten?

Hinsichtlich des Input-Registers PINx werden alle Statusinformationen des jeweiligen Ports auf einmal gelesen.

# Register und C++-Befehle

Zur Programmierung und Ansteuerung der verschiedenen Pins ist diese Art der Programmierung vielleicht etwas umständlich und deswegen stehen verschiedene Befehle in der Programmiersprache C++ zur Verfügung, die uns die Sache etwas vereinfachen. Wir kommen im Detail im nächsten Bastelprojekt 3 darauf zu sprechen, aber ich möchte dennoch den direkten Zusammenhang an dieser Stelle verdeutlichen:

Tabelle 2: Register und ihre Entsprechungen

| Register | C++-Befehl | Beispiel |
|---|---|---|
| DDRB | pinMode | pinMode(13, OUTPUT); |
| PORTB | digitalWrite | digitalWrite(10, HIGH); |
| PINB | digitalRead | digitalRead(8); |

Hier noch einige Informationen zu den verwendeten Befehlen:

# pinMode

Der Befehl *pinMode* programmiert die Datenflussrichtung eines digitalen Pins, wobei zwei Parameter notwendig sind. Der erste gibt den gewünschten Pin an und der zweite die Richtung, wobei OUTPUT für Ausgang und INPUT für Eingang steht.

# digitalWrite

Der Befehl *digitalWrite* beeinflusst den Zustand eines digitalen Pins, wobei zwei Parameter notwendig sind. Der erste gibt den gewünschten Pin an und der zweite den Pegel, wobei HIGH 5V und LOW 0V entspricht.

# digitalRead

Der Befehl *digitalRead* liest den Zustand eines digitalen Pins, wobei das Ergebnis entweder HIGH oder LOW sein kann.

Was ist beim Input-Register PINx zu beachten?

Hinsichtlich der beiden Pins RX und TX der seriellen Schnittstelle, die in ihren Datenflussrichtungen nicht verändert werden sollten und sich auf Port D befinden, gibt es eine sicherere Methode. Das nachfolgende Beispiel programmiert Pin 2 bis 7 als Ausgänge und verändert Pin 0 und 1 nicht: DDRD |= 0b11111100;

Warum ist das so? Wir verwenden eine binäre Oder-Verknüpfung des binären Wertes mit dem gezeigten Register. An den kritischen Stellen Pin 0 und 1 erfolgt eine Oder-Verknüpfung mit dem Wert 0, was bedeutet, dass diese beiden Bits unverändert bleiben. Nähere Informationen zur Manipulation von Bits sind unter der folgenden Adresse zu finden:

http://playground.arduino.cc/Code/BitMath

Für Detailinformationen zum Pin-Mapping ist die folgende Adresse sicherlich hilfreich:

http://www.arduino.cc/en/Reference/Atmega168Hardware

# Der Pullup-Widerstand

Wir haben gesehen, dass ein offener Eingang an einem digitalen Pin zu Problemen führen kann und deswegen eine externe Beschaltung zum Beispiel über einen Pulldown-Widerstand erforderlich ist. Nun gibt es aber auch eine Schaltung, bei der ein sogenannter Pullup-Widerstand eingesetzt wird. Es handelt sich dabei um einen ganz normalen Widerstand, der jedoch im Gegensatz zum Pulldown, der an Masse liegt, jetzt mit 5V der Versorgungsspannung verbunden ist. Bei einem offenen Eingang würden demnach 5V am betreffenden Pin anliegen. Sehen wir uns dazu die folgende Schaltung an:

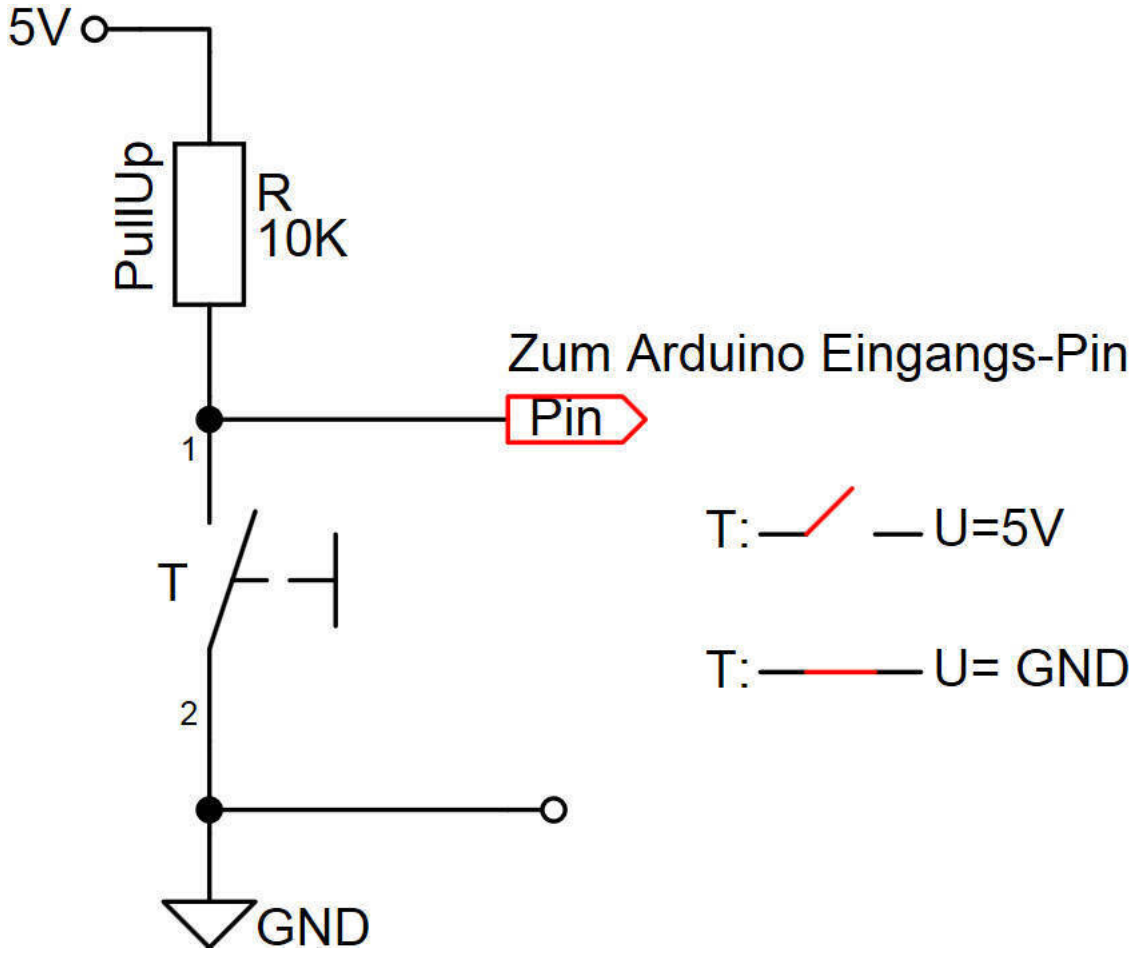



**Abb. 15:** Ein offener Eingang mit Pullup-Widerstand

Worauf ich hinaus möchte, ist die Tatsache, dass der Mikrocontroller intern über eingebaute Pullup-Widerstände verfügt, die bei Bedarf aktiviert werden können. Wie funktioniert das? Wir müssen dazu zwei Schritte durchführen, die eigentlich widersprüchlich sind:

Einen Pin als Eingang programmieren.
Den Pin mit einem HIGH-Pegel versehen.

Warum habe ich widersprüchlich gesagt? Nun, wenn ein Pin als *Eingang* programmiert wird, dann erwartet man doch, dass von außen ein Pegel anliegt, der sich auch ändern kann. Wenn wir jetzt aber im zweiten Schritt diesen Pin mit dem Wert *HIGH* versehen und so tun, als wäre der Pin als *Ausgang* programmiert, wird der interne Pullup-Widerstand aktiviert. Sehen wir uns dazu den entsprechenden Sketch an, der die Register DDB und PORTB verwendet:

void setup() { DDRB = 0b00000000; // Alle Pins als INPUT PORTB = 0b00000**111**; // Pullup für Pin 8, 9 und 10 aktiviert Serial.begin(9600); // Serielle Schnittstelle mit 9600 Baud vorbereiten } void loop() { Serial.println(PINB, BIN); delay(500); // Pause von 500 Millisekunden }

Wenn wir jetzt lediglich drei Mikrotaster ohne Widerstände an die digitalen Eingänge 8, 9 und 10 anschließen, dann funktioniert unsere Schaltung perfekt, denn die internen Pullup-Widerstände sorgen für einen sauberen und störungsfreien Pegel. Natürlich können für dieses Bastelprojekt die auf dem Arduino Discoveryboard vorhandenen Taster nicht verwendet werden, da sie über einen fest verdrahteten Pulldown-Widerstand verfügen. Das ist aber kein Problem, denn es befindet sich auf dem Arduino Discoveryboard ein kleines Breadboard, auf dem die kleinen Mikrotaster sehr gut eingesteckt werden können.

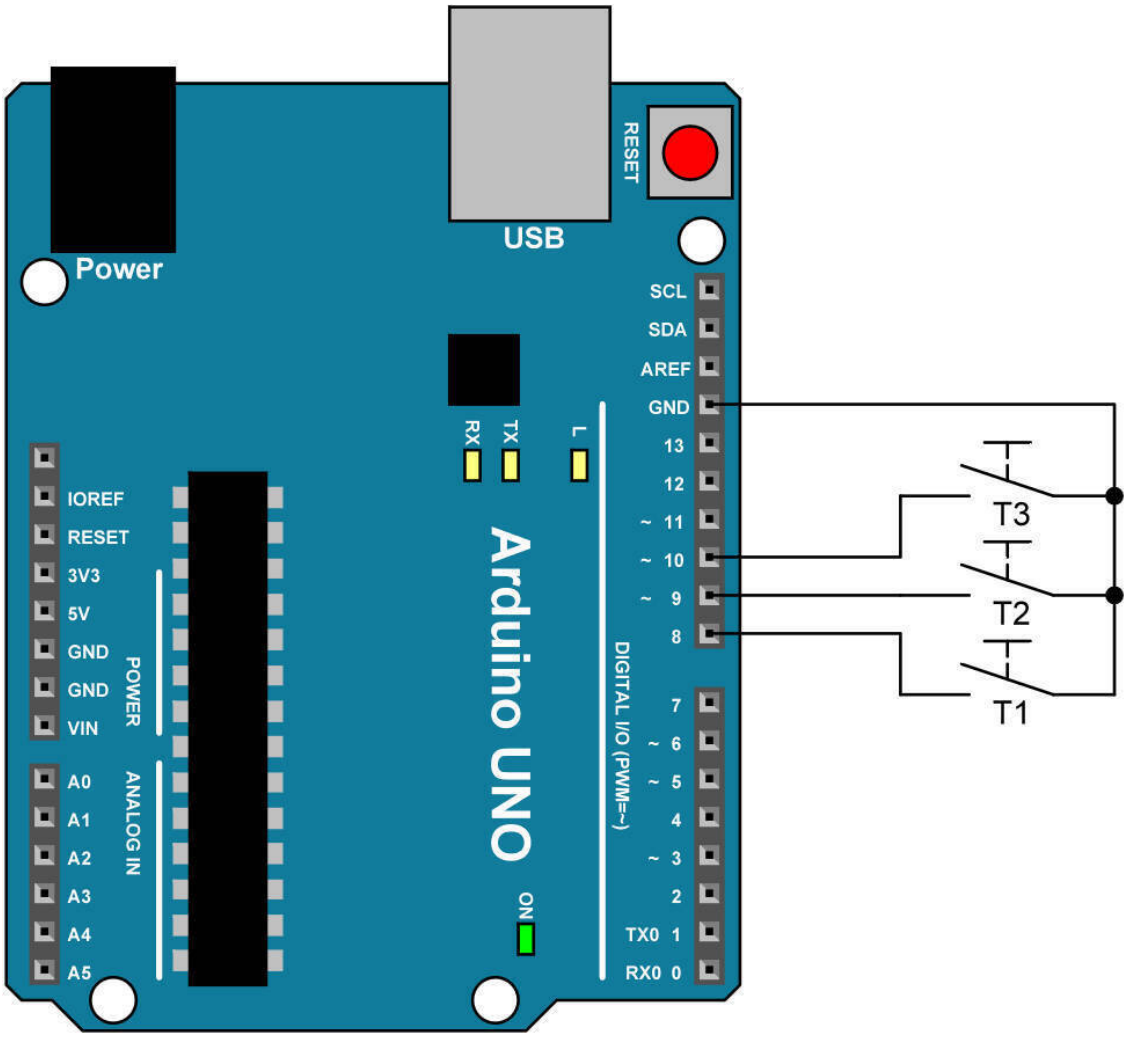



**Abb. 16:** Eine Schaltung mit drei Tastern

Wenn wir jetzt den Serial Monitor öffnen, werden wir feststellen, dass bei nicht gedrückten Tastern die jeweiligen Bits auf HIGH-Pegel liegen, was ja auch zu erwarten war. Drücken wir jetzt einen der Taster, ändert sich der betreffende Pegel des Bits von HIGH- auf LOW-Pegel. Natürlich kann man die internen Pullup-Widerstände auch mit einem konventionellen Arduino-Befehl aktivieren oder deaktivieren.

# Troubleshooting

Überprüf deine Steckverbindungen auf dem Breadboard, ob sie wirklich der Schaltung entsprechen.

# Was haben wir gelernt?

Es wurde der Zusammenhang zwischen Arduino-Pins und internen Mikrocontroller-Register hergestellt.

Über die Register DDRx, PORTx und PINx können Datenflussrichtung, logische Pegel gesetzt und abgefragt werden.

Es wurde die Leuchtdiode angesprochen und was bei der Ansteuerung zu beachten ist.

Wir haben die Problematik eines unbeschalteten digitalen Eingangs besprochen und Abhilfe über einen sogenannten Pulldown-Widerstand geschaffen.

Über geeignete Befehle wurden die internen Pullup-Widerstände aktiviert, so dass eine externe Beschaltung mit zusätzlichen Widerständen entfällt.

# Bastelprojekt 3:
# Einen Taster sicher abfragen

Warum werden in der Elektronik überhaupt Taster benötigt? Eine einfache Frage, die gar nicht so einfach zu beantworten ist, wie es auf den ersten Blick scheint. Klar, Taster werden dafür verwendet, um von außen Informationen an eine Schaltung zu leiten. Wird ein Taster gedrückt, sollen bestimmte Aktionen erfolgen, die in der Programmierung definiert wurden. Taster bilden also eine Schnittstelle zwischen Mensch und Mikrocontroller. Damit ein Tastendruck genau das bewirkt, was beabsichtigt ist, müssen wir grundlegende Dinge klären: Wann und wie oft wurde ein Taster gedrückt? In diesem Bastelprojekt wenden wir uns der Thematik zu, wann ein Taster gedrückt wurde und dass dieser Impuls auch nur dann weitergeleitet wird, wenn man den Taster schließt. In einem späteren Bastelprojekt – »Der störrische Taster« (Bastelprojekt 5) – geht es darum, wie oft ein Kontakt geschlossen wird, wenn der Taster einmal gedrückt wurde. Das sollte normalerweise auch nur einmal der Fall sein, was aber nicht immer stimmt. Wenn es also darum geht, einen Taster abzufragen, so wie es in der Überschrift steht, geht es darum zu ermitteln, wann dieser gedrückt wurde.

# Die Manipulation interner Pullup-Widerstände

Das Problem mit offenen beziehungsweise nicht beschalteten digitalen Eingängen sollte klar sein. Eine externe Beschaltung durch Pullup- bzw. Pulldown-Widerstände ist ein gängiges Verfahren, diesem Problem zu begegnen. Wir haben aber auch gesehen, dass der Mikrocontroller des Arduino Uno über interne Pullup-Widerstände verfügt, die bei Bedarf aktiviert oder deaktiviert werden können. Im zweiten Bastelprojekt, bei dem es um die Low-Level-Programmierung ging, habe ich das Aktivieren dieser Widerstände über das Setzen bestimmter Bits eines internen Registers gezeigt. Wir erinnern uns:

PORTB = 0b00000**111**; // Pullup für Pin 8, 9 und 10 aktiviert

Für geübte Programmierer, die gerne mit Bits und Bytes hantieren, ist das sicherlich eine tolle Sache, denn am meisten kann man erreichen, wenn man die Innereien eines Mikrocontrollers direkt anspricht. Doch viele – mich eingeschlossen – haben es lieber komfortabler. Ich sprach davon, dass es einen Arduino-Befehl gibt, über den ein interner Pullup-Widerstand aktiviert werden kann. Wie kann man sich einen internen Pullup-Widerstand vorstellen? Werfen wir dazu einen Blick auf das folgende Schaltbild:

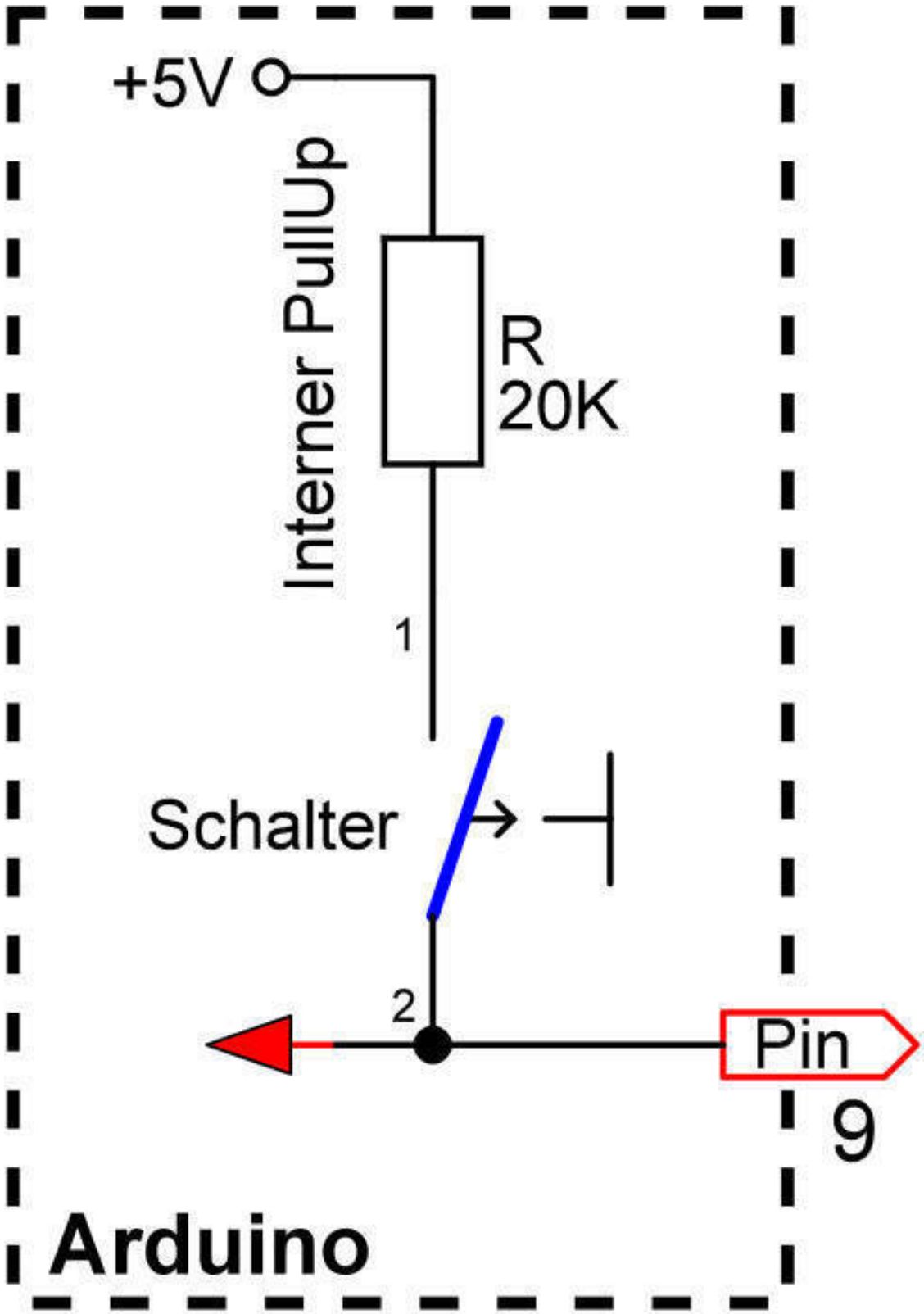



**Abb. 1:** Ein interner Pullup-Widerstand am digitalen Pin 9

In diesem Beispiel habe ich den digitalen Pin 9 ausgesucht, an dem dein Taster angeschlossen wird. Du erkennst auch den internen Pullup-Widerstand *R*, der über einen elektronischen Schalter den Pin 9 mit der Versorgungsspannung +5V – wenn er denn geschlossen wird – verbindet. Die Frage ist aber, wie du diesen Schalter schließen kannst, damit der Pin 9 bei fehlendem Eingangspegel einen *HIGH*-Pegel aufweist. Hierzu sind die folgenden Befehle erforderlich, wobei *pin* den Wert 9 besitzt:

pinMode(pin, INPUT); // Pin als Eingang konfigurieren digitalWrite(pin, HIGH); // Aktivieren des internen Pullup-Widerstandes

Sie bewirken das Schließen des besagten Schalters.

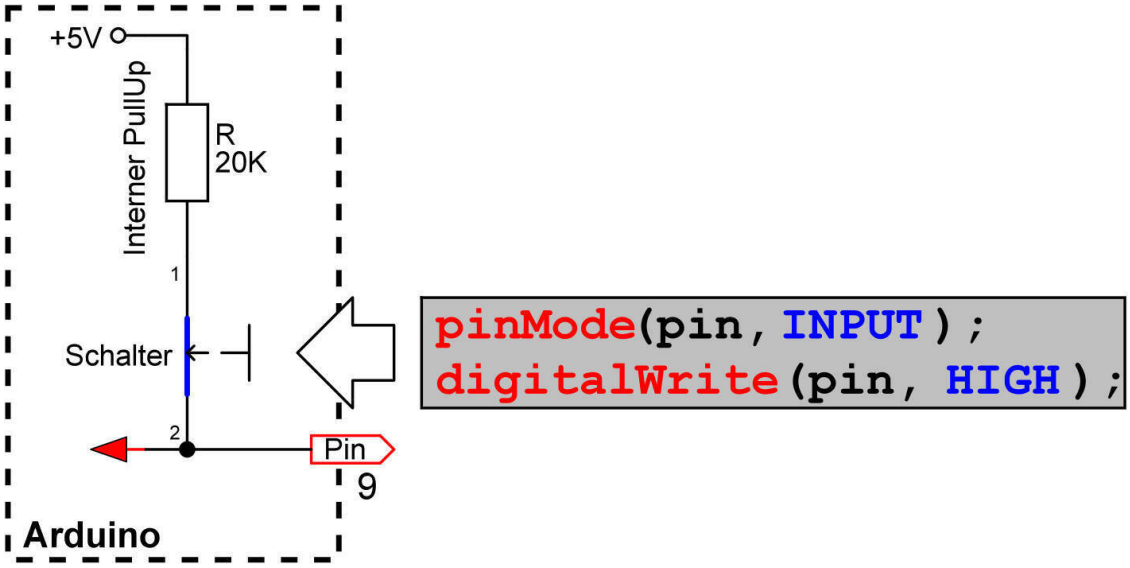



**Abb. 2:** Der interne Pullup-Widerstand wurde aktiviert

Vielleicht denkst du, dass hier etwas nicht stimmt. Du konfigurierst einen digitalen Pin als Eingang, weil wir daran einen Taster anschließen möchten. Das ist soweit noch klar. Aber dann wird versucht, mit dem Befehl *digitalWrite* an eben diesem Pin den Pegel zu verändern, der nicht als Ausgang konfiguriert wurde. Was soll das denn bedeuten? Genau das ist ja der Punkt. Über diese Befehlssequenz aktivieren wir den internen Pullup-Widerstand, der übrigens den Wert von 20KΩ besitzt. Damit zwingen wir das Potential des besagten Pins bei offenem Eingang in Richtung +5V und erhalten darüber einen definierten Eingangspegel.

Was ist bei einem Pullup- oder Pulldown-Widerstand zu beachten?

Hinsichtlich der Beschaltung eines digitalen Pins entweder über einen externen Pulldown- oder einen internen Pullup-Widerstand muss bei der Programmierung die Abfrage des Pins bei einem Tastendruck abweichend lauten.

Überlege erst einmal, bevor du hier weiterliest. Bei einem Pulldown-Widerstand liegt bei einem offenen Eingang ein LOW-Pegel an. Um eine Pegeländerung zu bewirken, müssen von außen +5V angelegt werden und das bedeutet, dass die Abfrage des Tasters wie folgt aussehen kann, wobei die Variable *tasterStatus* natürlich erst einmal initialisiert werden muss:

if(tasterStatus == HIGH) { ... }

So weit, so gut. Jetzt arbeitest du mit einem internen Pullup-Widerstand, der bei offenem Taster einen HIGH-Pegel hervorruft. Der angeschlossene Taster muss nun bei einer gewünschten Pegeländerung von außen ein LOW-Signal, also 0V erwirken, was bedeutet, dass die Abfrage des Tasters jetzt wie folgt aussehen muss:

if(tasterStatus == LOW) { ... }

Es gibt noch eine andere Variante, einen internen Pullup-Widerstand zu manipulieren. Ab der Arduino-Entwicklungsumgebung 1.0.1 ist es möglich, über einen speziellen Mode-Parameter beim *pinMode*-Befehl den Widerstand zu beeinflussen:

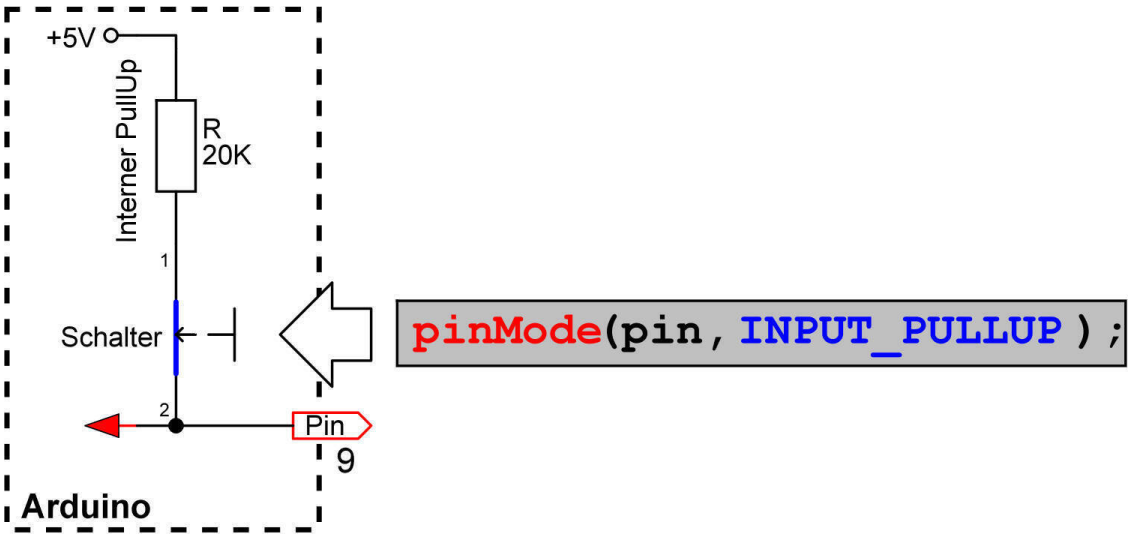



**Abb. 3:** Der interne Pullup-Widerstand wurde aktiviert

Durch einen einzigen *pinMode*-Befehl wurde der Schalter geschlossen. Die Deaktivierung erfolgt über den Modus *INPUT*, wobei der interne Schalter wieder geöffnet wird:

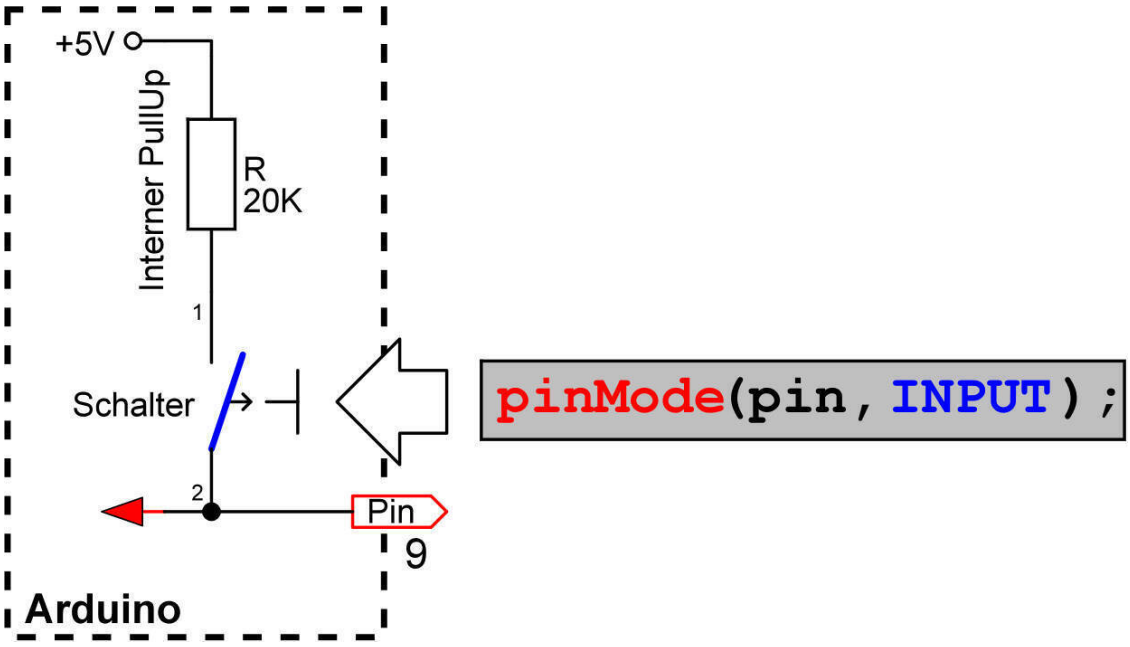



**Abb. 4:** Der internen Pullup-Widerstand wurde deaktiviert

Auch hier ist nur ein einziger *pinMode*-Befehl mit gezeigtem Modus erforderlich.

Kommen wir jetzt zu einem konkreten Beispiel, denn es ist immer noch nicht ganz klar, wie der Status eines digitalen Pins abgefragt werden kann. Es wird dazu der Befehl *digitalRead* verwendet, dessen Syntax wie folgt aussieht und den wir eigentlich schon im Bastelprojekt 1 über das Blinken der LED kennengelernt haben:


Diese Funktion wird aber nicht nur einfach aufgerufen, sondern liefert uns einen Rückgabewert, der für unsere Auswertung herangezogen werden kann. Über den Zuweisungsoperator = wird der Wert an eine Variable mit dem Namen *tasterStatus* übergeben. Die möglichen Rückgabewerte sind hierbei *HIGH* oder *LOW*, die vom System – wir erinnern uns – vordefinierte Konstanten darstellen. Bevor wir uns jedoch den Sketch, die erforderlichen Bauteile und den entsprechende Schaltplan anschauen, möchte ich den Aufbau und die Funktionsweise eines Mikrotasters erklären. Auf der folgenden Abbildung sehen wir einen derartigen Mikrotaster, der über vier Anschlüsse verfügt. Für das Schließen eines einzelnen Kontaktes werden zwei Anschlüsse benötigt, aber auch wenn wir es hier mit vier Anschlüssen zu tun haben, bedeutet das nicht, dass sich in dem Gehäuse zwei unabhängige Taster befinden:

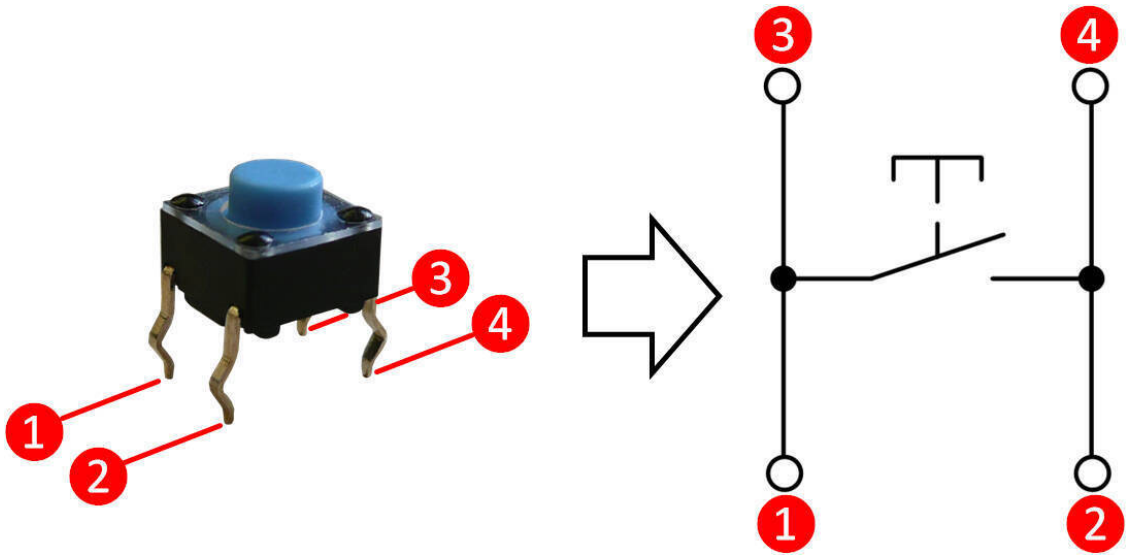


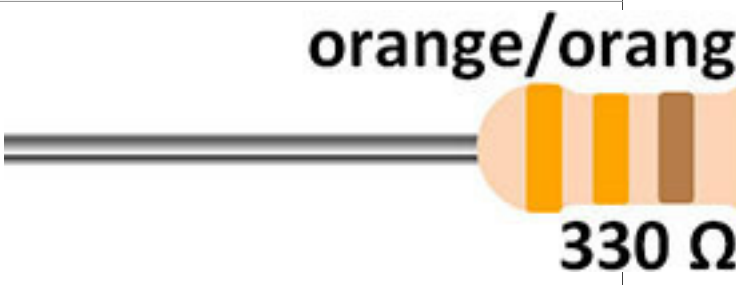
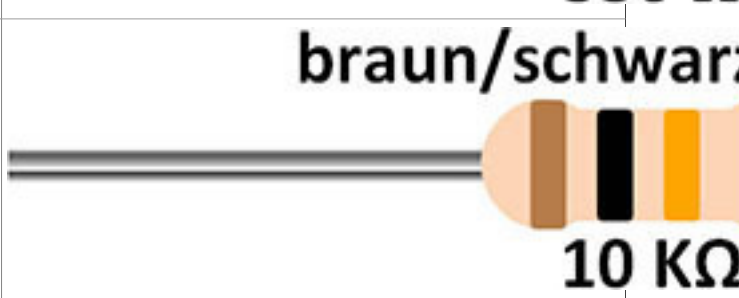

**Abb. 5:** Der Mikrotaster

Das Schaltbild rechts vom Taster zeigt die interne Verdrahtung und du erkennst, dass immer zwei Beinchen zusammengehören. Du kannst also den Taster über die Beinchen 1 und 2 oder über 3 und 4 ansprechen. Verdrehst du den Taster um 90 Grad und verwendest das gleiche Anschlussschema, dann hast du einen ewig geschlossenen Taster. Deshalb achte auf die Beinchenpaare, die zu einer Seite aus dem Gehäuse kommen. Sie werden kurzgeschlossen, wenn du den Taster betätigst. Notfalls nimmst du dir ein Multimeter zur Hand und verwendest den Durchgangstester, um die Tasterkontakte eindeutig zu identifizieren.

# Was wir brauchen

Für dieses Bastelprojekt wird nicht viel benötigt und im Grunde genommen kommen wir auch ohne zusätzliche Bauteile aus, denn auf dem Arduino-Board befindet sich eine LED mit der Bezeichnung *L*. Dennoch möchte ich dieses Bastelprojekt mit ein paar Komponenten anreichern, die auch in weiteren Projekten Verwendung finden:

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED rot 1x |  |
| Widerstand 330Ω 1x |  |
| Widerstand 10K Ω 1x |  |
| Mikrotaster 1x |  |

# Der Schaltplan

Der Schaltplan zeigt uns zur Sicherung eines sicheren Eingangspegels einen externen Pulldown-Widerstand am digitalen Pin 8.

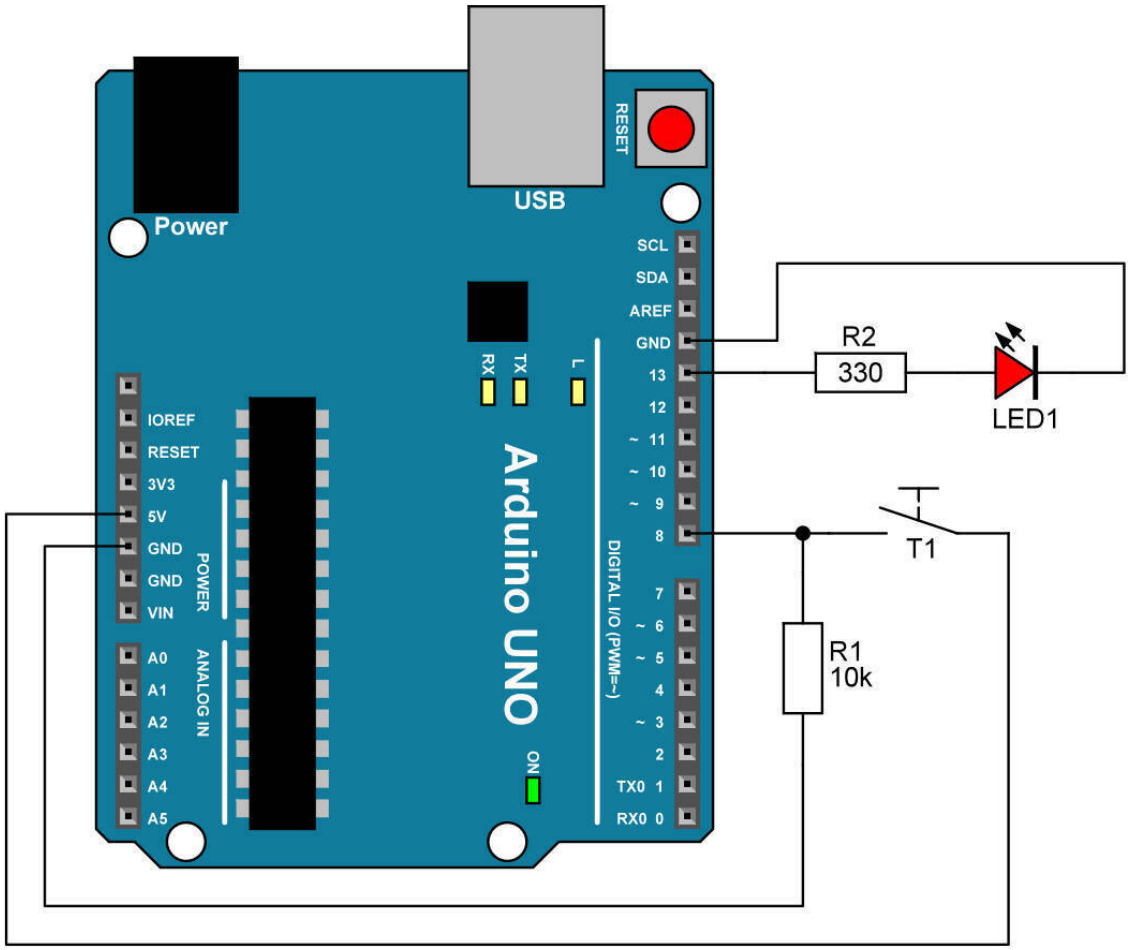



**Abb. 6:** Der Schaltplan zur Tasterabfrage

# Der Schaltungsaufbau

Der Schaltungsaufbau geht auf einem entsprechenden Breadboard leicht von der Hand und ist schnell realisiert.

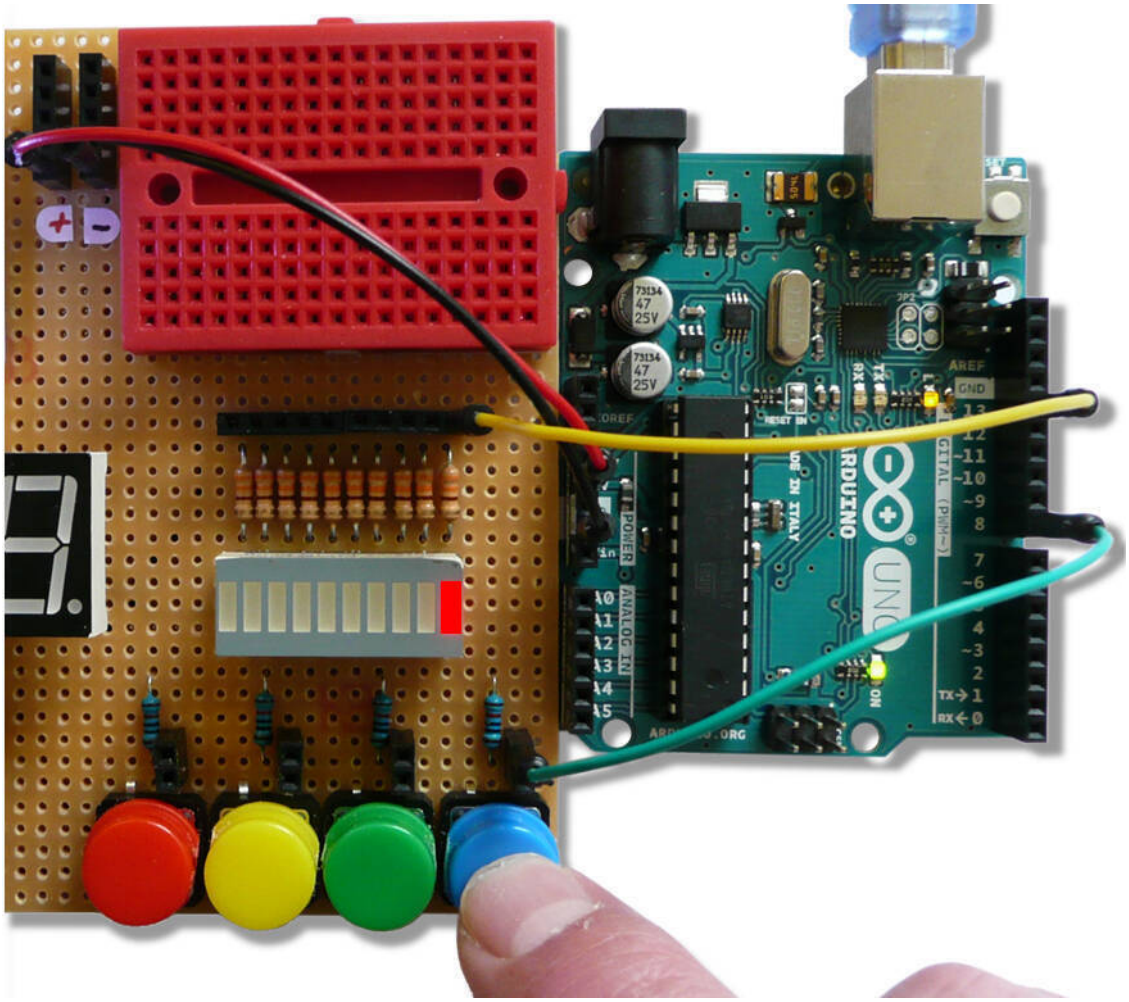


**Abb. 7:** Der Schaltungsaufbau zur Tasterabfrage auf dem Arduino Discoveryboard

Mit diesen technischen Grundlagen ausgestattet, wenden wir uns jetzt dem Sketch zu.

# Der Arduino-Sketch

Der Sketch zur Abfrage des Tasters und Ansteuerung der LED sieht wie folgt aus:

```
int ledPin = 13; // LED-Pin 13 int tasterPin = 8; // Taster-Pin 8 int tasterStatus; // Variable für den Tasterstatus void setup() { pinMode(ledPin, OUTPUT); // LED-Pin als Ausgang pinMode(tasterPin, INPUT); // Taster-Pin als Eingang } void loop() { tasterStatus = digitalRead(tasterPin); if(tasterStatus == HIGH) digitalWrite(ledPin, HIGH); else digitalWrite(ledPin, LOW); }
```

# Den Code verstehen

Es ist zu erkennen, dass in diesem Sketch mit mehreren Variablen gearbeitet wird, die am Anfang erst einmal deklariert werden müssen. Die Erklärungen dazu befinden sich hinter den Deklarationen im Sketch als Kommentare.

Was ist bei einem digitalen Pin zu beachten?

Ein digitaler Pin arbeitet standardmäßig als Eingang und benötigt deswegen keine explizite Konfiguration über den Befehl *pinMode*, wie das hier zu sehen ist. Es erhöht jedoch die Lesbarkeit, wenn es trotzdem erfolgt. Es kann weggelassen werden, wenn der Speicherplatz einmal knapp werden sollte.

Werfen wir kurz einen Blick auf das Flussdiagramm:

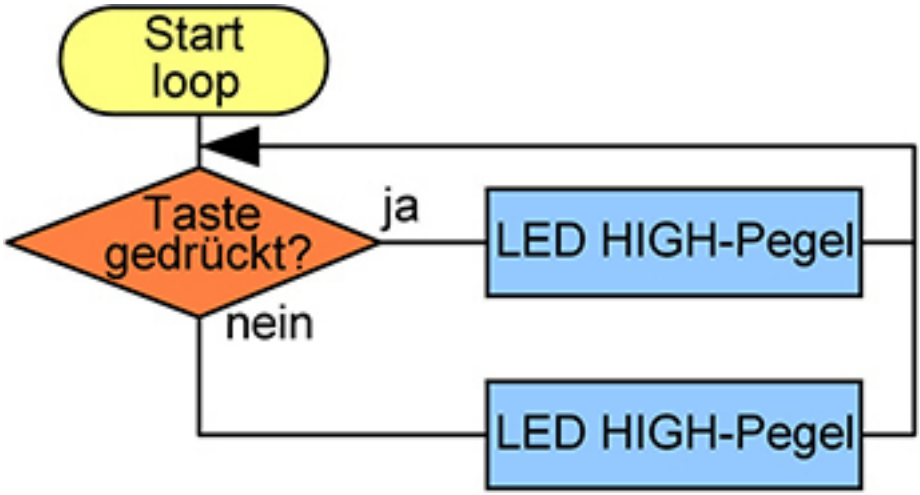



**Abb. 8:** Das Flussdiagramm zur LED-Ansteuerung

Das Diagramm liest sich recht einfach. Wenn die Ausführung des Sketches in der *loop*-Endlosschleife angekommen ist, wird der Zustand des Taster-Pins kontinuierlich abgefragt und in der Variablen *tasterStatus* abgelegt. Die entsprechende Zeile lautet:

tasterStatus = digitalRead(tasterPin);

Je nach Rückgabewert erfolgt die Auswertung über eine Kontrollstruktur in Form einer *if-else*-Anweisung (Wenn-Dann-Sonst):

if(tasterStatus == HIGH) digitalWrite(ledPin, HIGH); else digitalWrite(ledPin, LOW);

Die *if*-Anweisung bewertet den in runden Klammern stehenden Ausdruck, der umgangssprachlich ungefähr so formuliert werden könnte: »Ist der Inhalt der Variablen *tasterStatus* gleich dem Wert *HIGH*? Falls ja, dann führe den Befehl aus, der der *if*-Anweisung unmittelbar folgt. Falls nicht, fahre mit der Anweisung fort, die der *else*-Anweisung folgt.« Das folgende Flussdiagramm erleichtert das Verständnis dieser Kontrollstruktur:

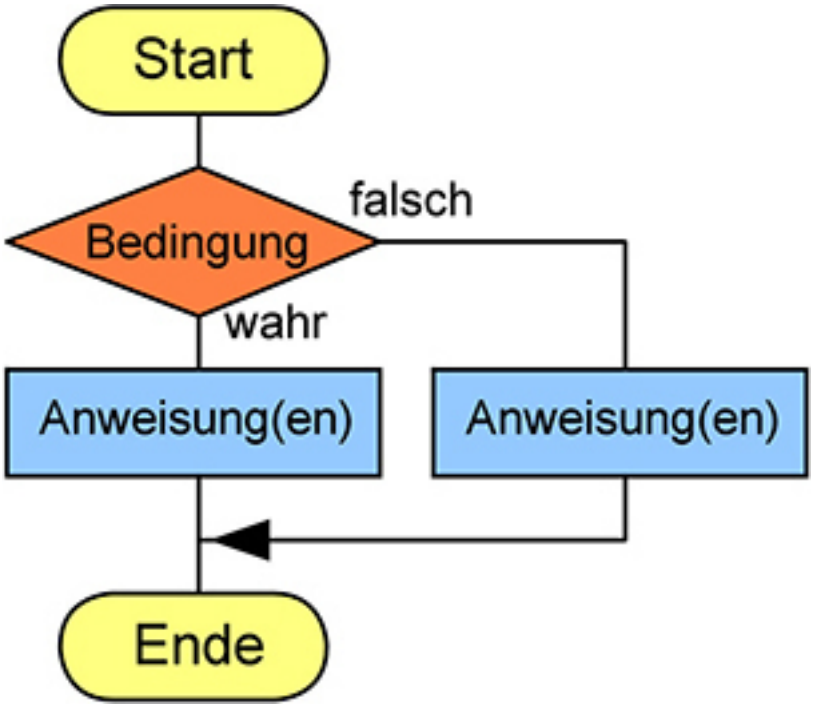



**Abb. 9:** Flussdiagramm zur if-else-Kontrollstruktur

Es gibt auch noch eine einfachere Variante dieser Kontrollstruktur, bei der der *else*-Zweig nicht vorhanden ist. Du siehst, dass ein Programmablauf nicht unbedingt geradlinig verlaufen muss. Es können Verzweigungen eingebaut werden, die anhand von Bewertungsmechanismen unterschiedliche Befehle oder Befehlsblöcke zur Ausführung bringen. Ein Sketch agiert nicht nur, sondern reagiert auf äußere Einflüsse, beispielsweise auf Sensorsignale.

Zuweisungsoperator kontra Gleichheitsoperator

Ein sehr häufiger Anfängerfehler ist die Verwechslung von Gleichheits- und Zuweisungsoperator. Der Gleichheitsoperator == und der Zuweisungsoperator = haben völlig unterschiedliche Aufgaben, werden aber häufig verwechselt. Das Heimtückische ist, dass beide Schreibweisen in einer Bedingung verwendet werden können und gültig sind.

Hier die korrekte Verwendung des Gleichheitsoperators:

if(tasterStatus == HIGH)

Nun die falsche Verwendung des Zuweisungsoperators:

if(tasterStatus = HIGH)

Aber warum erzeugt diese Schreibweise keinen erkennbaren Fehler? Ganz einfach: Es erfolgt eine Zuweisung der Konstanten *HIGH* (numerischer Wert 1) an die Variable *tasterStatus*. 1 bedeutet kein Nullwert und wird als *true* (wahr) interpretiert. Bei einer Codezeile, die *if*(*true*)... lautet, wird der nachfolgende Befehl immer ausgeführt. Ein numerischer Wert 0 wird in C/C++ als *false* (falsch) angesehen und jeder von 0 verschiedene als *true*. Derartige Fehler haben es in sich und es muss immer wieder sehr viel Zeit darauf ver(sch)wendet werden, sie ausfindig zu machen.

# Troubelshooting

Überprüf deine Steckverbindungen auf dem Breadboard, ob sie wirklich der Schaltung entsprechen.

Sind die LEDs richtig herum eingesteckt worden? Denk an die richtige Polung.

Hast du den Taster richtig angeschlossen?

# Was haben wir gelernt?

Der Mikrocontroller des Arduino Uno verfügt über interne Pullup-Widerstände, die einen Wert von 20KΩ besitzen.

Diese Widerstände können entweder über eine Sequenz von pinMode- beziehungsweise digitalWrite-Befehlen oder über einen einzigen pinMode-Befehl mit den Modi INPUT_PULLUP oder INPUT aktiviert beziehungsweise deaktiviert werden.

Über den Befehl digitalRead kann der Pegel eines digitalen Pins abgefragt werden.

Über Kontrollstrukturen wie if oder if-else kann Einfluss auf den Programmablauf genommen werden.

# Bastelprojekt 4:
# Blinken mit Intervallsteuerung

Im Bastelprojekt 1 zur Ansteuerung einer LED hast du gesehen, wie wir über die Verzögerungsfunktion *delay* eine Unterbrechung in der Ausführung des Sketches an der betreffenden Stelle hervorgerufen haben. Die angeschlossene LED an dem digitalen Pin blinkte in regelmäßigen Abständen. Eine solche Schaltung und Programmierung hat jedoch einen entscheidenden Nachteil, den wir zuerst erkennen und dann beheben wollen. Wir müssen die Blinkschaltung dazu ein wenig modifizieren und erweitern.

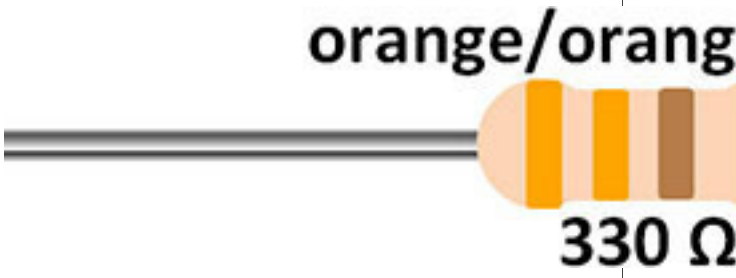
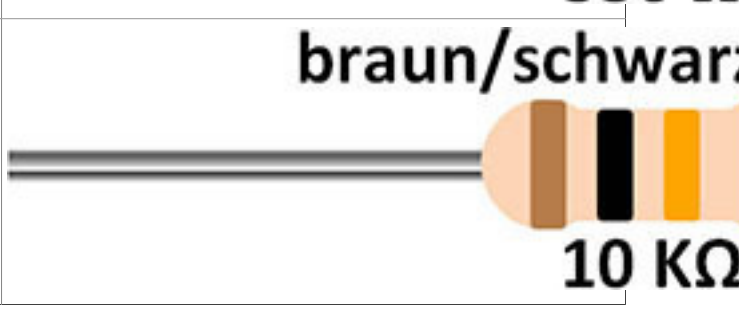
# Drücke den Taster – und er reagiert

Was geschähe wohl, wenn du an einem anderen digitalen Eingang zusätzlich einen Taster anschließen würdest, um dann seinen Zustand kontinuierlich abzufragen? Wenn du diesen Taster drückst, soll zeitgleich eine weitere LED leuchten. Vielleicht ahnst du schon, worauf ich hinausmöchte. Solange die Sketch-Ausführung in der *delay*-Funktion gefangen ist, wird die Abarbeitung des Codes unterbrochen und der digitale Eingang kann demnach nicht abgefragt werden. Du drückst also den Taster und es passiert einfach nichts.

# Was wir brauchen

Für dieses Bastelprojekt benötigen wir die folgenden Bauteile:

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED rot 1x |  |
| LED gelb 1x |  |
| Mikrotaster 1x |  |
| Widerstand 330Ω 1x |  |
| Widerstand 10KΩ 1x |  |

# Der Schaltplan

Der Schaltplan zeigt uns zur Gewährleistung eines sicheren Eingangspegels einen externen Pulldown-Widerstand am digitalen Pin 8:

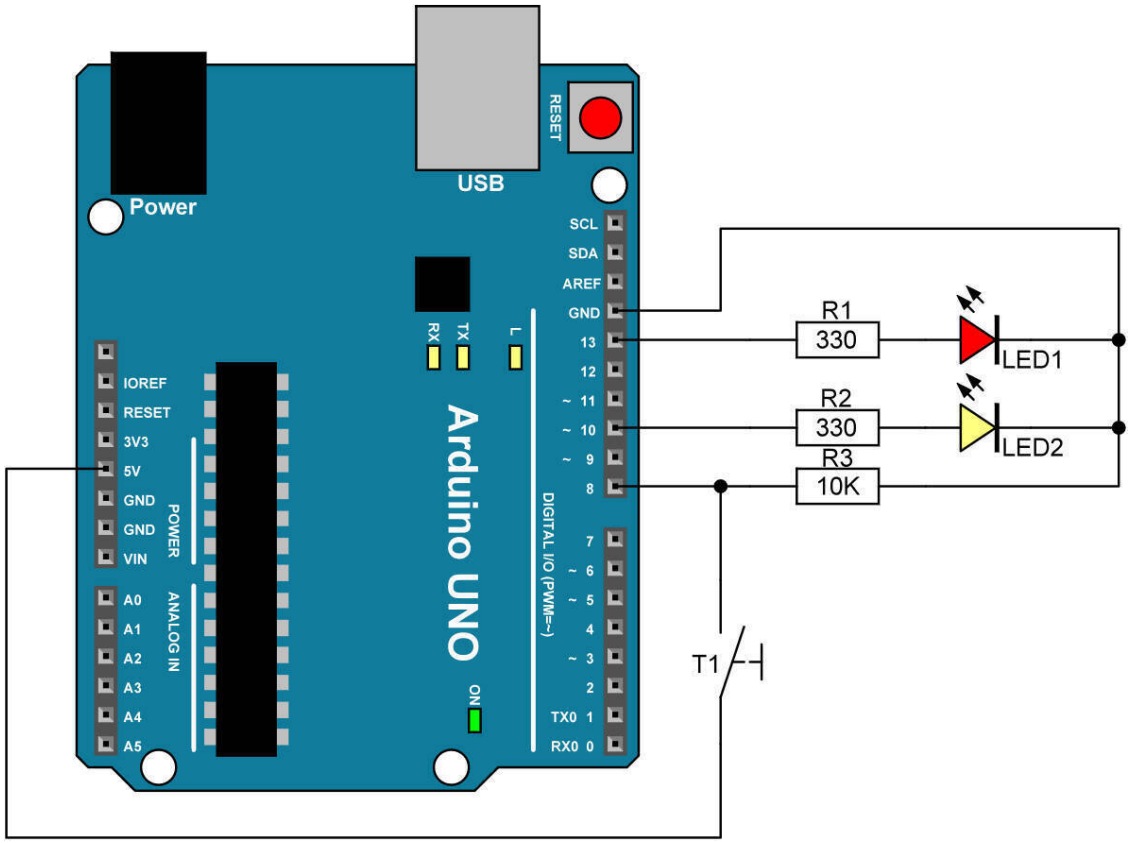



**Abb. 1:** Der Schaltplan zur Abfrage des Tasters und Ansteuerung der LEDs

# Der Schaltungsaufbau

Auf dem Arduino Discoveryboard werden in diesem Bastelprojekt ein paar Bauteile mehr zum Einsatz kommen:

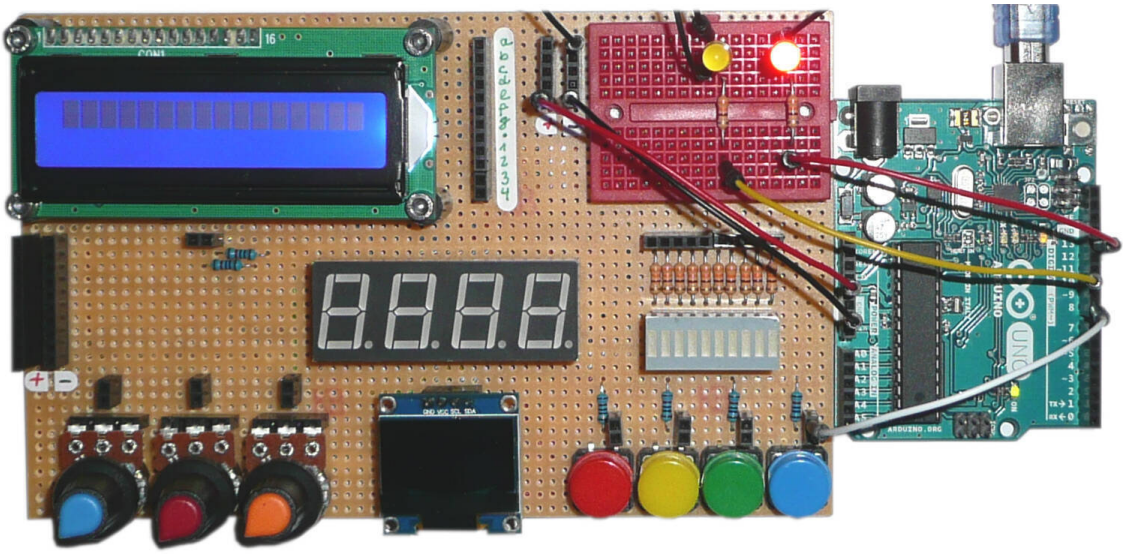


**Abb. 2:** Der Schaltungsaufbau zur Tasterabfrage
Mit diesem Schaltungsaufbau wenden wir uns dem Sketch zu.

# Der Arduino-Sketch

Vorwarnung: Der folgende Sketch funktioniert nicht so, wie wir es vielleicht erwarten.

// Der Sketch funktioniert nicht wie erhofft int ledPinBlink = 13; // Rote Blink-LED Pin 13 int ledPinTaster = 10; // Gelbe Taster-LED Pin 10 int tasterPin = 8; // Taster Pin 8 int tasterStatus; // Variable für Tasterstatus void setup() { pinMode(ledPinBlink, OUTPUT); // Blink-LED Pin als Ausgang pinMode(ledPinTaster, OUTPUT); // Taster-LED Pin als Ausgang pinMode(tasterPin, INPUT); // Taster-Pin als Eingang } void loop() { // Blink-LED blinken lassen digitalWrite(ledPinBlink, HIGH); // Rote LED HIGH-Pegel delay(1000); // 1 Sek. warten digitalWrite(ledPinBlink, LOW); // Rote LED LOW-Pegel delay(1000); // 1 Sek. warten // Abfrage des Taster-Status tasterStatus = digitalRead(tasterPin); if(tasterStatus == HIGH) digitalWrite(ledPinTaster, HIGH); // Gelbe LED HIGH-Pegel else digitalWrite(ledPinTaster, LOW); // Gelbe LED LOW-Pegel }

Warum funktioniert der Sketch denn nicht wie erwartet? Die Ausführung in der Endlosschleife kommt doch irgendwann einmal an der Zeile für die Tasterabfrage vorbei. Dann wird der Status doch korrekt abgefragt. Das entscheidende Wörtchen, das hier verwendet wird, ist *irgendwann*! Du möchtest aber sicherlich einen Sketch so programmieren, dass zu jedem Zeitpunkt der Verarbeitung auf einen Tastendruck reagiert wird und nicht nur irgendwann einmal, wenn die Ausführung des Codes gerade die betreffende Stelle erreicht. Die *delay*-Funktion behindert uns bei der Fortführung des Codes und kann hier nicht die erste Wahl sein. Ich zeige dir das Verhalten am besten anhand eines Impulsdiagramms, bei dem die relevanten Signale wie Pin 13 (Blink-LED), Pin 8 (Taster) und Pin 10 (Taster-LED) untereinander zu sehen sind:

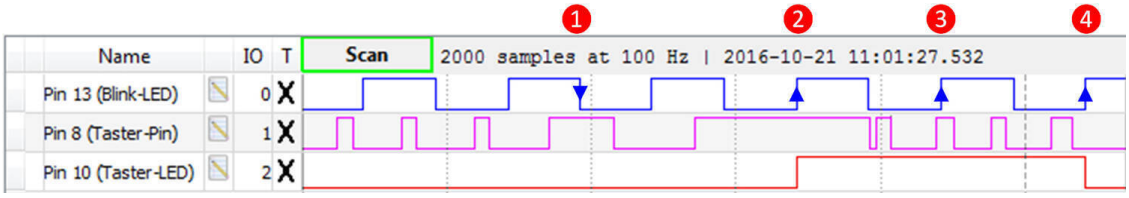



**Abb. 3:** Das Impulsdiagramm der Signale an Pin 13, Pin 8 und Pin 10

Das oberste Signal (hier blau) zeigt den Status der blinkenden LED an Pin 13, der unermüdlich im Sekundentakt zwischen *HIGH*- und *LOW*-Pegel wechselt. Das darunter liegende Signal (hier lila) zeigt den Pegel am Taster an Pin 8, der nun versucht, die Taster-LED an Pin 10 zu steuern. Das unterste Signal sollte also zeitgleich mit dem Signal am Taster den Pegel wechseln, was aber nicht der Fall ist. Ich habe vier markante rote Punkte in diesem Diagramm platziert. An Punkt 1 halte ich den Taster so lange gedrückt, dass ich die abfallende Flanke des Blink-Signals erwische. Warum reagiert die Taster-LED an Pin 10 hier nicht? Das ist ganz einfach, denn wir haben es mit zwei *delay*-Aufrufen zu tun. Bei einem Wechsel von *HIGH* zu *LOW* wird die erste *delay*-Funktion abgearbeitet und wir stecken quasi noch in der zweiten *delay*-Funktion fest. Ein Tastendruck wird also hier nicht abgefragt. Das geschieht erst an Punkt 2, wo ich die Taste beim Pegelwechsel von *LOW* zu *HIGH* gedrückt halte. Bevor es zum weiteren Pegelwechsel geht, erfolgt die Abfrage des digitalen Pins 8, an dem sich der Taster befindet. Jetzt kann der Status des Tasters ausgewertet werden und die Taster-LED an Pin 10 leuchtet. Das ist so lange der Fall, bis bei der ansteigenden Flanke der Blink-LED der Taster ein *LOW*-Signal hat, was erst bei Punkt 4 der Fall ist und nicht schon bei Punkt 3. Aus diesem Grund müssen wir auf die Verwendung der *delay*-Funktion verzichten und einen anderen Weg wählen.

Sehen wir uns den folgenden Sketch an und lassen uns nicht durch die Anzahl der Codezeilen irritieren, denn wir gehen alles Schritt für Schritt durch:

// Der Sketch funktioniert wie erhofft int ledPinBlink = 13; // Rote Blink-LED Pin 13 int ledPinTaster = 10; // Gelbe Taster-LED Pin 10 int tasterPin = 8; // Taster Pin 8 int tasterStatus; // Variable für Tasterstatus int interval = 2000; // Intervalzeit (2 Sekunden) unsigned long prev; // Zeit-Variable int ledStatus = LOW; // Statusvariable für die Blink-LED void setup() { pinMode(ledPinBlink, OUTPUT); // Blink-LED-Pin als Ausgang pinMode(ledPinTaster, OUTPUT); // Taster-LED-Pin als Ausgang pinMode(tasterPin, INPUT); // Taster-Pin als Eingang prev = millis(); // Jetzigen Zeitstempel merken } void loop() { // Blink-LED über Intervalsteuerung blinken lassen if((millis() - prev) > interval) { prev = millis(); ledStatus = ! ledStatus; // Toggeln des LED-Status digitalWrite(ledPinBlink, ledStatus); // Toggeln der roten LED } // Abfrage des Tasterstatus tasterStatus = digitalRead(tasterPin); if(tasterStatus == HIGH) digitalWrite(ledPinTaster, HIGH); // Gelbe LED HIGH-Pegel else digitalWrite(ledPinTaster, LOW); // Gelbe LED auf LOW-Pegel }

Schauen wir uns die Erklärungen zu diesem Sketch an.

# Den Code verstehen

In diesem Sketch haben wir es mit einigen Variablen zu tun. Ich beginne mit der sogenannten *Intervallsteuerung*. Das folgende Impulsdiagramm zeigt uns das Verhalten der Schaltung und es ist genau so, wie es erwünscht ist:


**Abb. 4:** Das Impulsdiagramm der Signale an Pin 13, Pin 8 und Pin 10

Jedes Mal, wenn der Taster (Taster-Pin) gedrückt wird, folgt der Pegel der Taster-LED. Und das Ganze unabhängig vom Zustand beziehungsweise Pegelwechsel an Pin 13 (Blink-LED). Für die Intervallsteuerung benötigen wir eine neue Funktion, die sich *millis* nennt und wie folgt aussieht:


Sie liefert einen Wert in Millisekunden seit dem Starten des Sketches zurück. Dabei ist etwas Wichtiges zu beachten: Der Rückgabedatentyp ist *unsigned long*, also ein vorzeichenloser 32-Bit-Ganzzahltyp, dessen Wertebereich sich von 0 bis 4.294.967.295 ($2^{32}$-1) erstreckt. Dieser Wertebereich ist so groß, weil er über einen längeren Zeitraum (maximal 49.71 Tage) in der Lage sein soll, die Daten aufzunehmen, bevor es zu einem Überlauf kommt.

Was ist ein Variablenüberlauf?

Ein Überlauf bedeutet bei Variablen, dass der maximal abbildbare Wertebereich für einen bestimmten Datentyp überschritten wurde und anschließend wieder bei 0 begonnen wird. Für den Datentyp *byte*, der eine Datenbreite von 8 Bits aufweist und demnach 28 = 256 Zustände (0 bis 255) speichern kann, tritt ein Überlauf bei der Aktion 255 + 1 auf. Den Wert 256 ist der Datentyp *byte* nicht mehr in der Lage zu verarbeiten.

Es wurden drei weitere Variablen eingefügt, die folgende Aufgabe haben:

interval: Nimmt die Zeit im ms auf, die für das Blinkintervall zuständig ist.

prev: Nimmt die aktuell verstrichene Zeit in ms auf. prev kommt von previous und bedeutet übersetzt vorher.

ledStatus: In Abhängigkeit des Status von HIGH oder LOW der Variablen wird die Blink-LED angesteuert.

Dann sehen wir mal, wie das Ganze so abläuft. Das folgende Diagramm soll den zeitlichen Verlauf der Intervallsteuerung verdeutlichen:

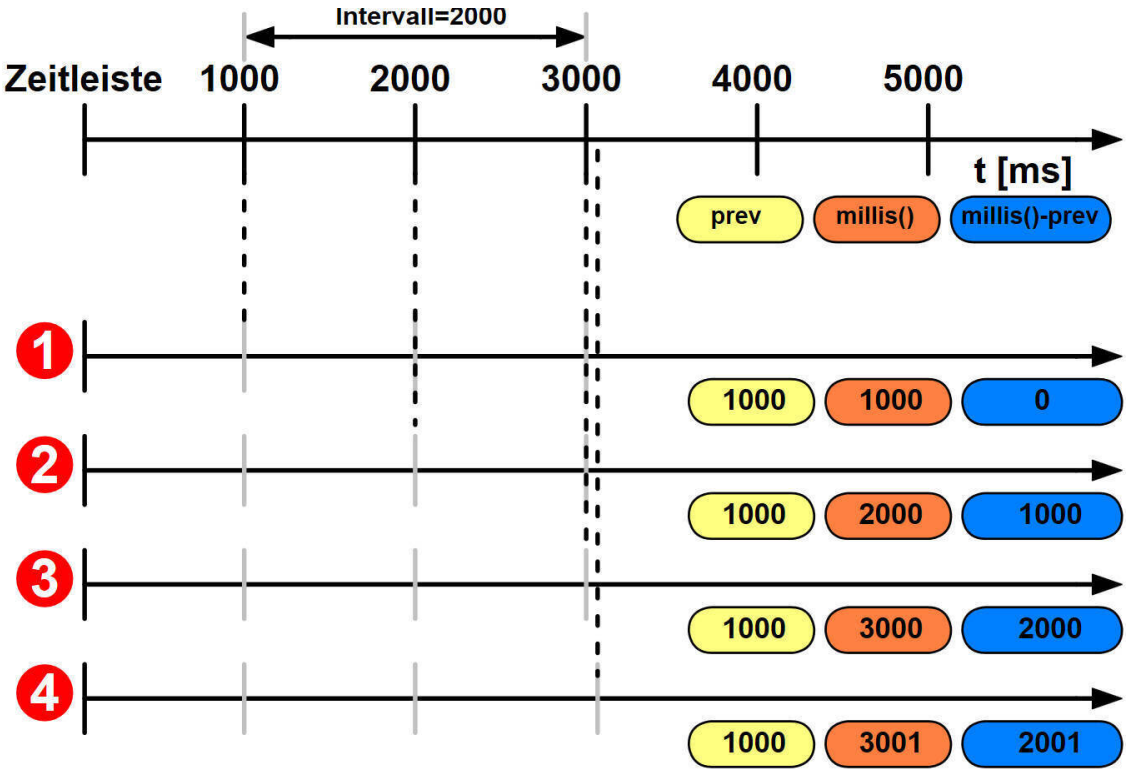



**Abb. 5:** Der zeitliche Verlauf der Intervallsteuerung

Ich analysiere einmal das Diagramm, wobei ich markante Zeitpunkte zur Verdeutlichung herausgegriffen habe. Natürlich läuft die Zeit nicht real in diesen Schritten ab:

Tabelle 2: Variableninhalte im zeitlichen Verlauf

| Zeitpunkt | Erklärung |
|---|---|
| 1 | Es wird die aktuelle Zeit (in diesem Fall 1000) in Millisekunden in die Variable *prev* übernommen. Dies erfolgt einmalig in der *setup*-Funktion. Die Differenz *millis() – prev* liefert als Ergebnis den Wert 0. Dieser Wert ist nicht größer als der Intervallwert 2000. Die Bedingung ist nicht erfüllt und der *if*-Block wird nicht ausgeführt. |
| 2 | Weitere 1000ms später wird wieder die Differenz *millis() – prev* gebildet und das Ergebnis dahingehend überprüft, ob es größer als der Intervallwert 2000 ist. 1000 ist nicht größer 2000, also ist die Bedingung wieder nicht erfüllt. |
| 3 | Nochmals 1000ms später wird erneut die Differenz *millis() – prev* gebildet und das Ergebnis darauf überprüft, ob es größer als der Intervallwert 2000 ist. 2000 ist nicht größer 2000, also ist die Bedingung wieder nicht erfüllt. |
| 4 | Nach 3001ms Laufzeit erbringt die Differenz jedoch einen Wert, der größer als der |

| | |
|---|---|
| | Intervallwert 2000 ist. Die Bedingung wird erfüllt und der *if*-Block zur Ausführung gebracht. Es wird der alte *prev*-Wert mit dem aktuellen Zeitwert aus der *millis*-Funktion überschrieben. Der Zustand der Blink-LED kann umgekehrt werden. Das Spiel beginnt auf der Basis des neuen Zeitwertes in der Variablen *prev* von vorn. |

Während des gesamten Sketch-Ablaufs wurde an keiner Stelle im Quellcode ein Halt in Form einer Pause eingelegt, sodass das Abfragen des digitalen Pins 8 zur Steuerung der Taster-LED nicht beeinträchtigt wurde. Ein Druck auf den Taster wird fast unmittelbar ausgewertet und angezeigt.

Eine Zeile im Sktech könnte vielleicht noch ein wenig Kopfschmerzen bereiten. Was bedeutet *ledStatus = !ledStatus*? Und was heißt *toggeln*, wie ich es im Kommentar hinter dem Befehl genannt habe? In der Variablen *ledStatus* wird der Pegel gespeichert, der die rote LED ansteuert oder für das Blinken zuständig ist (*HIGH* bedeutet aufleuchten und *LOW* bedeutet dunkel). Über die nachfolgende Zeile wird die LED dann angesteuert:

digitalWrite(ledPinBlink, ledStatus);

Das Blinken wird gerade dadurch erreicht, dass du zwischen den beiden Zuständen *HIGH* bzw. *LOW* hin- und herschaltest. Das wird auch *Toggeln* genannt. Ich werde die Zeile etwas umformulieren, dann wird der Sinn deutlicher:

if(ledStatus == **LOW**) ledStatus = **HIGH**; else ledStatus = **LOW**;

In der ersten Zeile wird abgefragt, ob der Inhalt der Variablen *ledStatus* gleich *LOW* ist. Falls ja, setze ihn auf `HIGH` andernfalls auf `LOW`. Das bedeutet ebenfalls ein Toggeln des Status. Viel kürzer geht es mit der folgenden einzeiligen Variante, die ich im Sketch verwendet habe:

ledStatus = !ledStatus; // Toggeln des LED-Status

Ich benutze dabei den logischen *Not*-Operator, der ja durch das Ausrufezeichen repräsentiert wird. Wie wir schon in Kapitel 3 gelernt haben, wird er häufig bei booleschen Variablen verwendet, die nur die Wahrheitswerte *true* oder *false* annehmen können. Der *Not*-Operator ermittelt ein Ergebnis, das einen entgegengesetzten Wahrheitswert aufweist wie der Operand. Es funktioniert aber auch bei den beiden Pegeln *HIGH* und *LOW*. Am Schluss wird ganz normal und ohne Verzögerung der Taster an Port 8 abgefragt:

tasterStatus = digitalRead(tasterPin); if(tasterStatus == HIGH) digitalWrite(ledPinTaster, HIGH); else digitalWrite(ledPinTaster, LOW);

# Troubleshooting

Falls die LED nicht leuchtet, wenn du den Taster drückst, oder die LED ständig leuchtet, dann geh bitte folgende Dinge durch:

Überprüf deine Steckverbindungen auf dem Breadboard, ob sie wirklich der Schaltung entsprechen.

Sind die LEDs richtig herum eingesteckt worden? Denk an die richtige Polung.

Achte auf den Taster mit zwei beziehungsweise vier Anschlüssen. Mach gegebenenfalls einen Durchgangstest mit einem Multimeter und überprüfe damit die Funktionsfähigkeit des Tasters und der entsprechenden Beinchen.

Haben die beiden Widerstände die korrekten Werte und wurden auch nicht vertauscht?

Überprüfe noch einmal den Sketch-Code auf Richtigkeit.

# Was haben wir gelernt?

Du hast die Verwendung mehrerer Variablen gesehen, die für die unterschiedlichsten Zwecke genutzt wurden (Deklaration für Eingangs- und Ausgangs-Pin und Aufnahme von Statusinformationen).

Der Befehl delay unterbricht die Ausführung des Sketches und erzwingt eine Pause, so dass alle nachfolgenden Befehle nicht berücksichtigt werden, bis die Wartezeit verstrichen ist.

Du hast über die Intervallsteuerung mittels der millis-Funktion einen Weg kennengelernt, dennoch den kontinuierlichen Sketch-Ablauf der loop-Endlosschleife aufrechtzuerhalten, sodass weitere Befehle der loop-Schleife ausgeführt wurden und damit eine Auswertung weiterer Sensoren, wie zum Beispiel der angeschlossene Taster, möglich waren.

Du hast verschiedene Impulsdiagramme kennen und lesen gelernt, die grafisch unterschiedliche Pegelzustände im zeitlichen Verlauf sehr gut darstellen.

# Bastelprojekt 5:
# Der störrische Taster

In diesem Bastelprojekt wirst du erkennen, dass sich ein Taster oder ein Schalter nicht immer so verhält, wie du es dir erwünschst. Nehmen wir für dieses Bastelprojekt einen Taster, der – so die Theorie – eine Unterbrechung des Stromflusses aufhebt, solange er gedrückt bleibt, und die Unterbrechung wiederherstellt, wenn du ihn loslässt. Das ist nichts Neues und absolut einfach zu verstehen. Doch bei elektronischen Schaltungen, deren Aufgabe beispielsweise im Ermitteln der exakten Anzahl von Tastendrücken liegt, um sie später auszuwerten, bekommen wir es mit einem Problem zu tun, das zunächst überhaupt nicht augenfällig ist.

# Ich wurde geprellt!

In der Elektromechanik gibt es einen Störeffekt, den man *Prellen* nennt. Wenn du einen ganz normalen Taster drückst und gedrückt hältst, sollte man meinen, dass der mechanische Kontakt im Taster dauerhaft geschlossen wird. Das ist jedoch meistens nicht der Fall, denn wir haben es mit einem Bauteil zu tun, das innerhalb einer sehr kurzen Zeitspanne – im Millisekundenbereich – den Kontakt mehrfach öffnet und wieder schließt. Die Kontaktflächen eines Tasters sind in der Regel nicht vollkommen glatt, und wenn wir sie uns unter einem Elektronenmikroskop ansähen, sähen wir viele Unebenheiten und Verunreinigungen. Das führt dazu, dass die Berührungspunkte der leitenden Materialien bei Annäherung nicht sofort und nicht auf Dauer zueinanderfinden. Eine weitere Ursache für den hier angeführten Effekt, den man Prellen nennt, kann im Schwingen oder Federn des Kontaktmaterials liegen, wodurch bei Berührung kurzzeitig der Kontakt mehrfach hintereinander geschlossen und wieder geöffnet wird.


Diese Impulse, die der Taster liefert, werden vom Mikrocontroller registriert und korrekt verarbeitet, nämlich so, als ob du den Taster absichtlich ganz oft und schnell hintereinander drücken würdest. Das Verhalten ist natürlich störend und muss in irgendeiner Weise verhindert werden. Dazu sehen wir uns das Impulsdiagramm in Abbildung 1 einmal etwas genauer an.

Ich habe einen Taster einmal gedrückt und dann gedrückt gehalten, doch bevor er den stabilen Zustand des Durchschaltens erreicht hatte, zickte er ein wenig und unterbrach die gewünschte Verbindung mehrfach. Das Ein- und Ausschalten, bis der endgültige gewünschte *HIGH*-Pegel erreicht ist, wird also *Prellen* genannt. Das Verhalten kann auch in entgegengesetzter Richtung auftreten. Auch wenn ich den Taster wieder loslasse, werden unter Umständen mehrere Impulse generiert, bis ich endlich den gewünschten *LOW*-Pegel erhalte. Das Prellen des Tasters ist für das menschliche Auge kaum oder überhaupt nicht wahrnehmbar, und wenn wir eine Schaltung aufbauen, die bei gedrücktem Taster eine LED ansteuern soll, stellen sich die einzelnen Impulse aufgrund der Trägheit der Augen als ein *HIGH*-Pegel dar. Ich schlage eine Schaltung mit einem entsprechenden Sketch vor, bei dem die einzelnen Impulse eines Prellens gezählt werden, denn das menschliche Auge kann diese Impulse nicht differenzieren.

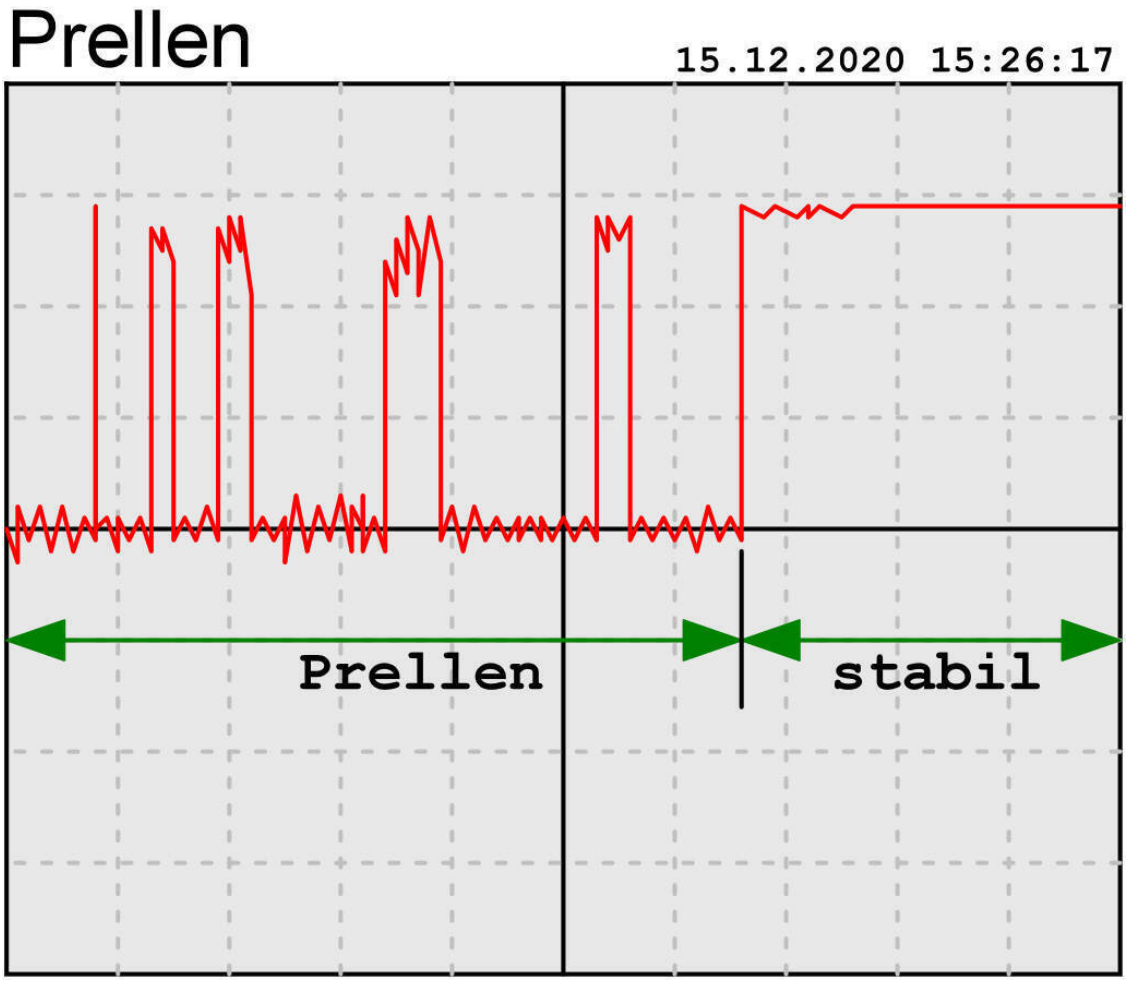
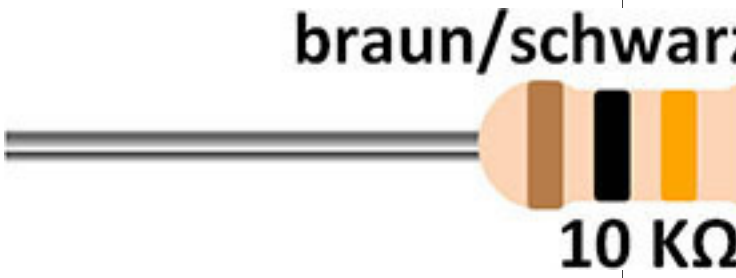



**Abb. 1:** Ein prellender Taster – Schlimmer Finger!

# Was wir brauchen

Für dieses Bastelprojekt benötigen wir die folgenden Bauteile:

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| Widerstand 10KΩ 1x |   |
| Mikrotaster 1x |  |

# Der Schaltplan

Der Schaltplan und der Schaltungsaufbau dazu ist recht einfach und sieht wie folgt aus:

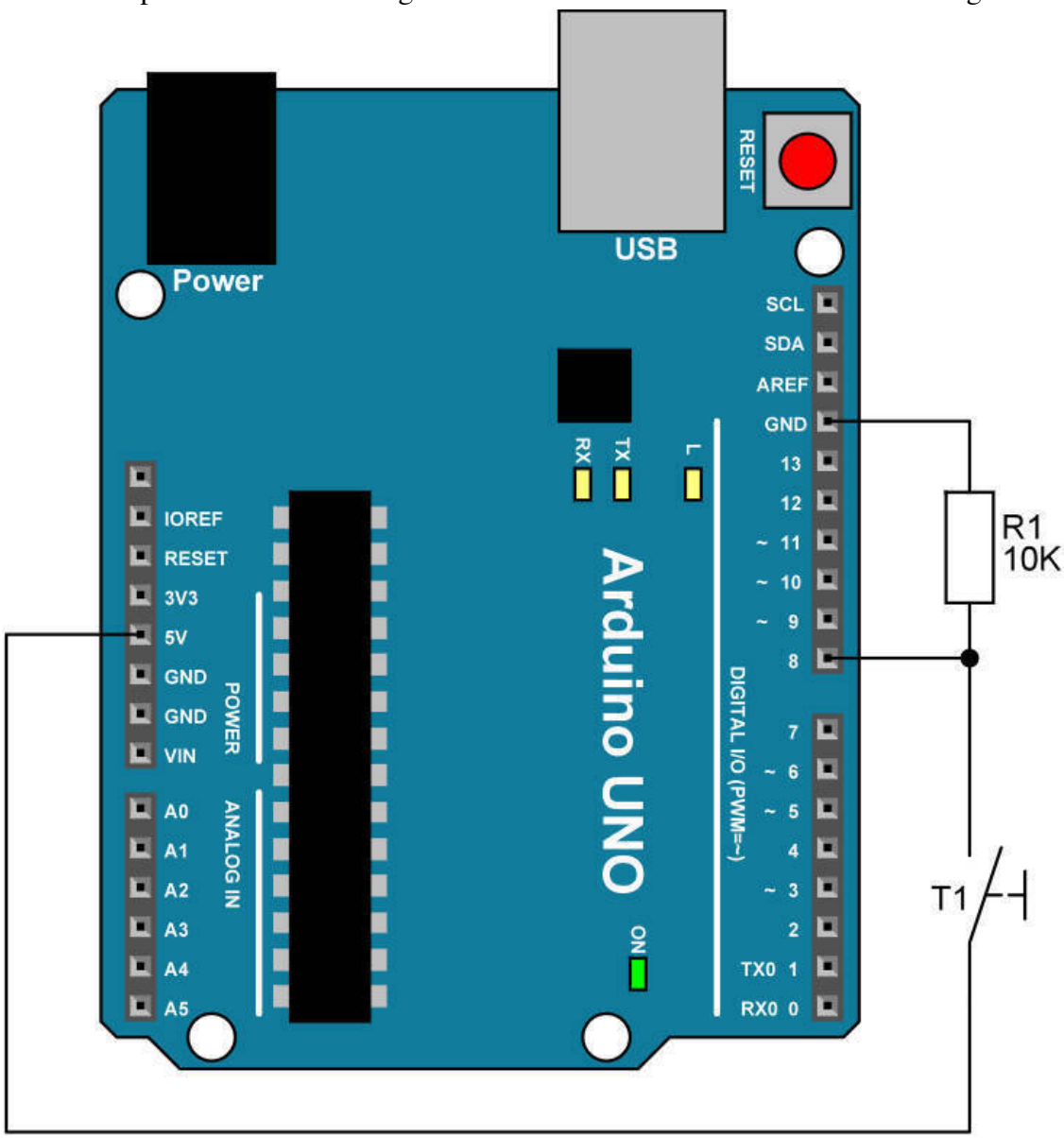



**Abb. 2:** Die Abfrage des Tasterstatus an Pin 8
Der entsprechende Schaltungsaufbau auf dem Arduino Discoveryboard sieht so aus:

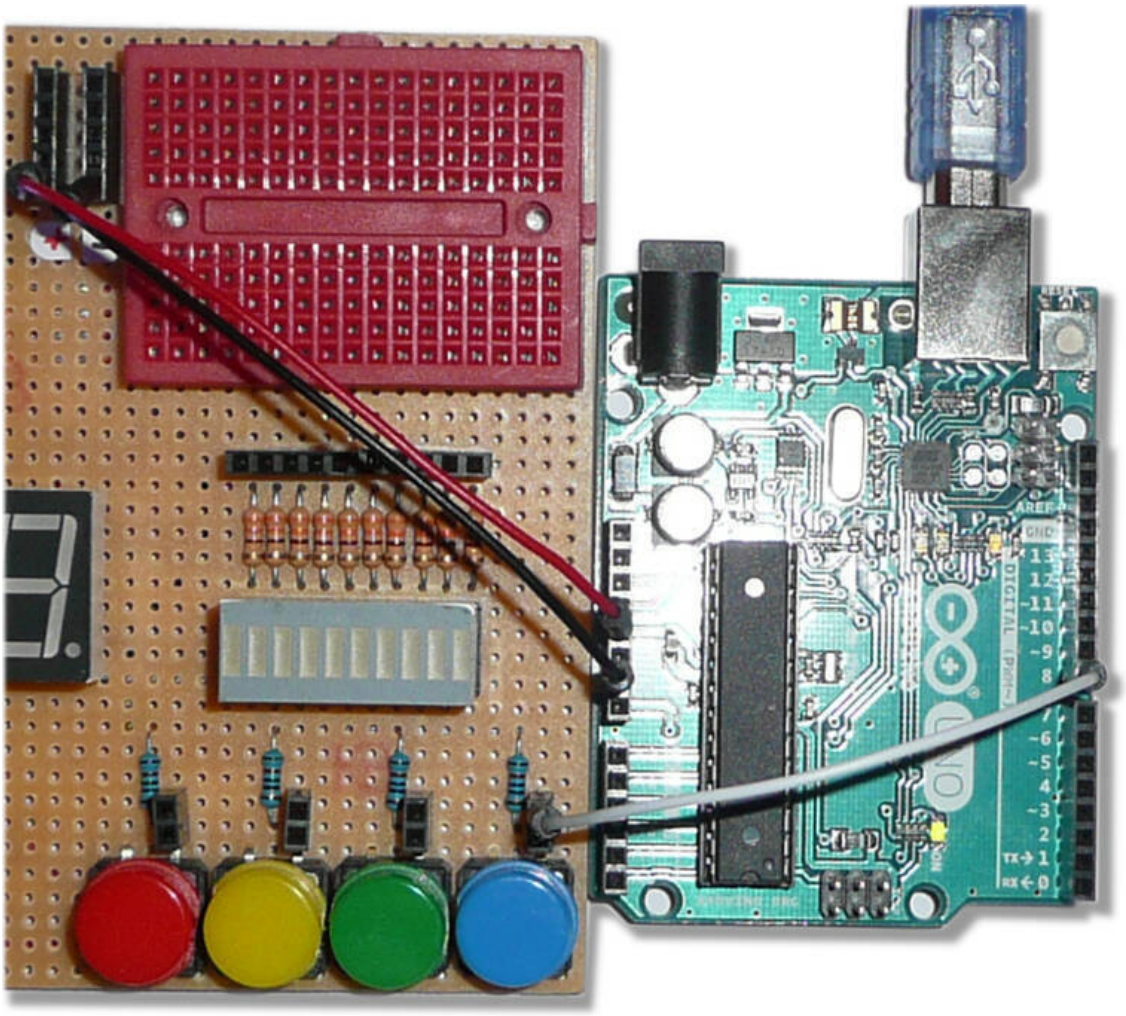


**Abb. 3:** Die Abfrage des Tasterstatus an Pin 8

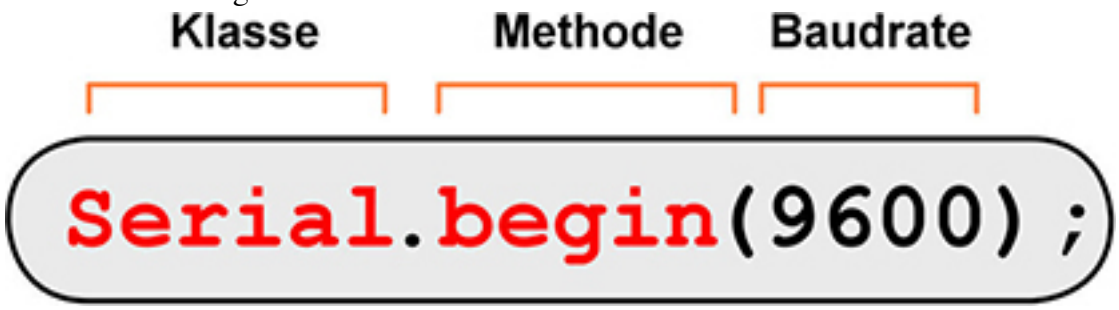
# Der Arduino-Sketch

Der folgende Code bringt die einzelnen Impulse zur Anzeige:

```
int tasterPin = 8; // Taster-Pin int impulse = 0; // Zähler boolean prevTasterState; // Änderung erkennen void setup(){ Serial.begin(9600); // Serielle Schnittstelle } void loop() { boolean tasterStatus = digitalRead(tasterPin); if(tasterStatus != prevTasterState){ // Änderung wurde erkannt prevTasterState = tasterStatus; if(tasterStatus == HIGH) { impulse++; // Impulse zählen Serial.println(impulse); } } }
```

## Den Code verstehen

Die Variable *tasterPin* wird mit dem Wert 8 initialisiert, da dort der Mikrotaster angeschlossen ist. Um später die Anzahl der Impulse zu speichern, wird eine Variable mit Namen *impulse* deklariert und mit dem Wert 0 initialisiert. Die Initialisierung der seriellen Schnittstelle erfolgt über ein entsprechendes Objekt innerhalb der *setup*-Funktion. Zuerst wird die Klasse genannt und im Anschluss – durch einen Punkt voneinander getrennt – die Methode mit dem Parameter. Die genauere Erläuterung erfolgt im Bastelprojekt über die objektorientierte Programmierung, wo es um Objekte, Klassen und Methoden geht.




Wichtig ist hier die Baudrate, die mit 9600 Baud angegeben wurde. Wird von einem Terminalprogramm wie dem Serial Monitor darauf zugegriffen, muss die Baudrate dort ebenfalls mit 9600 Baud angegeben werden, was dort in der linken unteren Ecke über eine Liste mit unterstützten Baudraten erfolgt:


Innerhalb der *loop*-Funktion wird nun die Variable *prevTasterState* genutzt, die dazu da ist, um immer den letzten Tasterstatus zu speichern. Sie ist notwendig, da immer nur ein Wechsel des Status berücksichtigt werden soll und auch nur dann, wenn ein HIGH-Pegel am Taster vorherrscht. Wenn du nun den Taster betätigst, sollte je nach Tastendruck die Variable *impulse* immer um den Wert 1 erhöht werden. Im Serial Monitor kannst du die Ergebnisse ablesen. Zur Anzeige der ermittelten Impulse wird die folge Syntax der seriellen Schnittstelle verwendet, wobei die Methode *println* den angegebenen Wert versendet:

Theoretisch sollte pro einzelnem Tastendruck der angezeigte Wert um 1 erhöht werden. Das scheint auch teilweise zu stimmen, doch nicht immer. Manchmal wird der Wert je Tastendruck um mehr als 1 erhöht, was auf das angesprochene Prellen hinweist. Öffne den Serial Monitor und beobachte die Anzeige. Man sollte meinen, dass bei jedem Tastendruck der Zähler um den Wert 1 erhöht wird, was jedoch nicht immer der Fall ist.

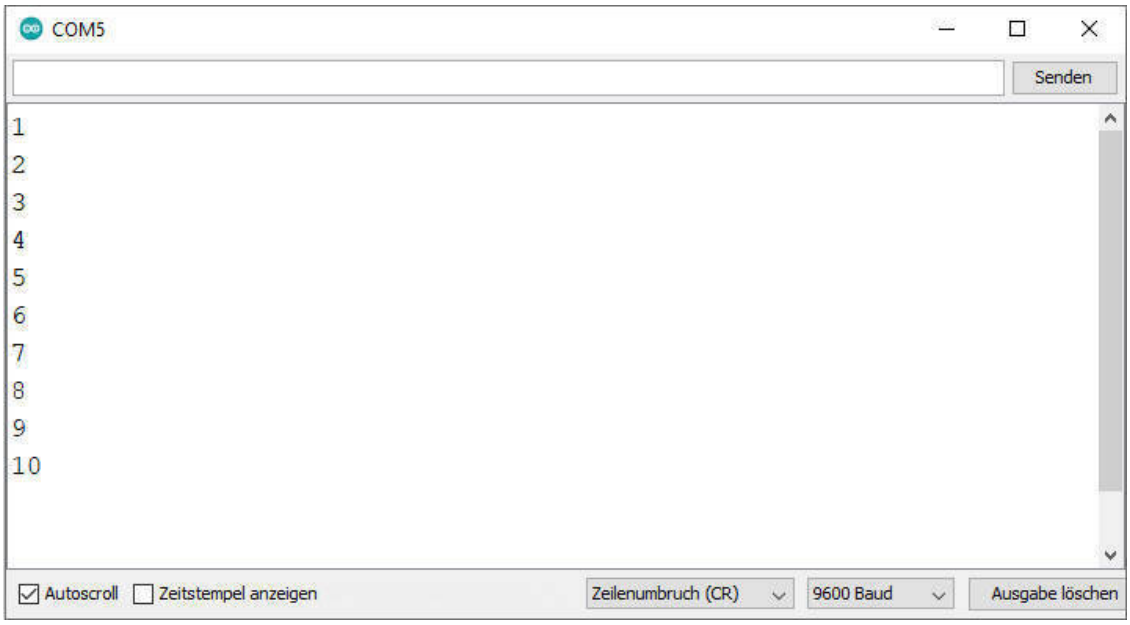



**Abb. 4:** Der Serial Monitor bringt das Prellen ans Licht

Gleich beim ersten Tastendruck zeigte die Anzeige hintereinander die Werte 1 und 2. Danach ging es stabil weiter und dann traten wieder 6 und 7 gleichzeitig in Erscheinung. Prellt ein Taster wirklich heftig, können pro Tastendruck auch mal mehr als zwei Impulse registriert werden. Probiere es einfach einmal aus. Wenn du keinen Zugang zu einem Oszilloskop hast, um dir das Prellen eines Tasters grafisch darzustellen, ist das auch kein Problem, denn die Arduino-Entwicklungsumgebung verfügt über ein interessantes Feature, den sogenannten *Serial Plotter*. Er kann anstelle des Serial Monitors den zeitlichen Verlauf von Pegeln darstellen. Verwende dazu den folgenden Sketch und öffne im Anschluss den Serial Plotter über den Menüpunkt *Werkzeuge* | *Serieller Plotter*:

int tasterPin = 8; // Taster-Pin void setup() { Serial.begin(9600); // Serielle Schnittstelle } void loop() { Serial.println(digitalRead(tasterPin)); }

Was ist eine serielle Schnittstelle?

Eine serielle Schnittstelle ist eine Einrichtung zur Datenübertragung zwischen zwei Geräten, bei denen die zu übertragende Information in Form einzelner Bits zeitlich hintereinander – also seriell – übertragen wird. Die Rate, mit der die Information über den Kommunikationskanal übertragen wird, wird Baudrate genannt. Je höher diese Baudrate ist, desto mehr Bits werden pro Sekunde übertragen.

## Den Code verstehen

Der Code ist übersichtlich, aber für das Anzeigen des Prellvorgangs vollkommen ausreichend. Es wird innerhalb der *loop*-Schleife der gemessene Pegel am Taster-Pin an die serielle Schnittstelle übertragen. Sowohl der Serial Monitor als auch der Serial Plotter können diese Daten darstellen. Bei mir sah das Prellen nach mehreren Versuchen wie in der folgenden Abbildung aus. Es ist sehr gut zu erkennen, dass der HIGH-Pegel nach dem Loslassen des Tasters nicht sofort auf einen LOW-Pegel geht, sondern erst noch einmal kurz in Richtung HIGH-Pegel zuckt, bevor er endgültig einen stabilen LOW-Pegel vorweist. Das ist der Beweis dafür, dass ein Taster nicht nur beim Schließen, sondern auch beim Öffnen prellen kann.

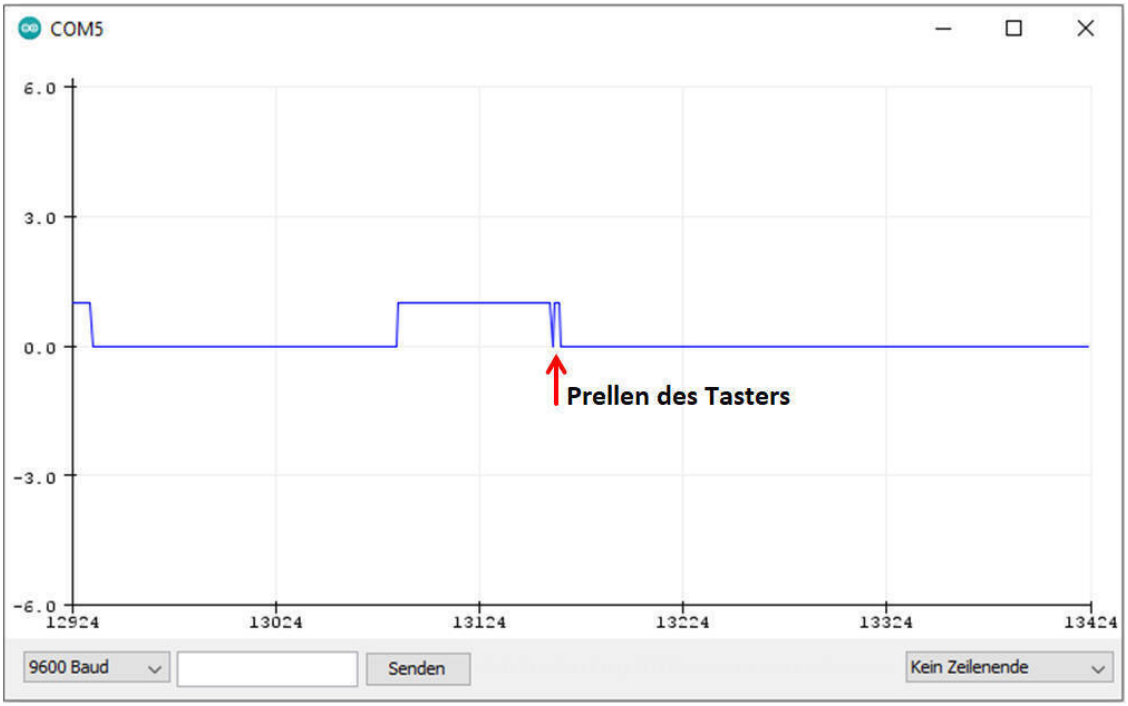



**Abb. 5:** Der Serial Plotter bringt das Prellen ans Licht

Das ist zwar jetzt alles schön und gut, doch wie kann dem unerwünschten Prellen nun begegnet werden? Es gibt diesbezüglich mehrere Ansätze, die sowohl über Soft- als auch über Hardware realisierbar sind. Da es mir in diesem Bastelprojekt vor allem um den Softwareaspekt geht, lasse ich eine Hardwarelösung außen vor.

# Anti-Prell-Lösung #1

Eine sehr einfache Lösung besteht im Einfügen einer kurzen Pause in der Verarbeitung durch einen *delay*-Befehl von zum Beispiel 10ms, wie das hier innerhalb der *loop*-Funktion zu sehen ist:

void loop() { boolean tasterStatus = digitalRead(tasterPin); if(tasterStatus != prevTasterState) { // Änderung erkannt prevTasterState = tasterStatus; delay(10); // Entprellen if(tasterStatus == HIGH) { impulse++; // Impuls zählen Serial.println(impulse); } } }

# Anti-Prell-Lösung #2

Versuchen wir es nun mit einer anderen Lösung. Was hältst du davon, wenn wir eine Schaltung aufbauen, die einen Taster an einem digitalen Eingang besitzt und eine LED an einem anderen digitalen Ausgang? Bei jedem Tastendruck soll die angeschlossene LED an- oder ausgeschaltet werden. Sie toggelt im Takt des Tastendrucks. Ohne eine Gegenmaßnahme des Entprellens würde die LED bei einem Tastendruck entweder an oder aus bleiben, weil zum Beispiel kurz hintereinander ein AN-AUS oder AUS-AN erfolgt. Wenn also mehrere Impulse beim Drücken des Tasters von der Schaltung beziehungsweise der Software registriert werden, wechselt die LED mehrfach ihren Zustand. Bei einem prellfreien Taster sollten sich die Zustände wie im folgenden Diagramm darstellen:

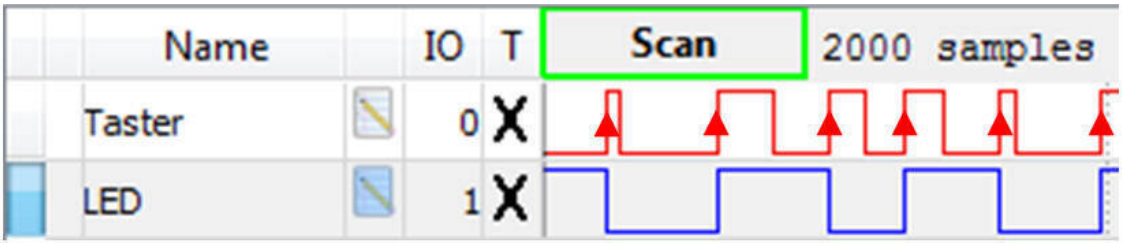


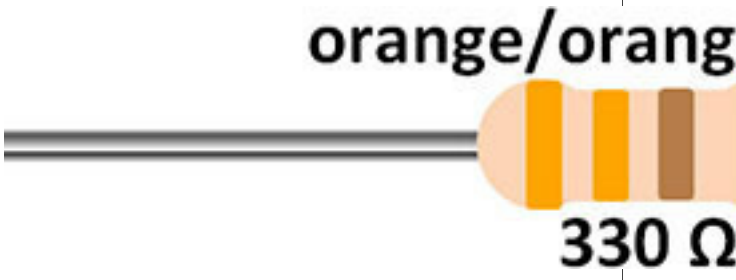
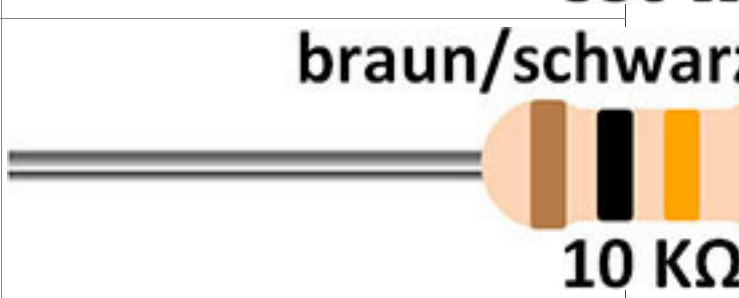

**Abb. 6:** Das Toggeln der LED durch den Taster

Es ist zu sehen, dass sich immer bei der ansteigenden Flanke des Tasterimpulses der Zustand der LED ändert.

# Was wir brauchen

Für dieses Bastelprojekt benötigen wir die folgenden Bauteile:

Tabelle 2: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED rot 1x |  |
| Widerstand 330Ω 1x |  |
| Widerstand 10KΩ 1x |  |
| Mikrotaster 1x |  |

# Der Schaltplan

Der Schaltplan gleicht dem der Abfrage eines Tasters aus einem vorherigen Bastelprojekt:

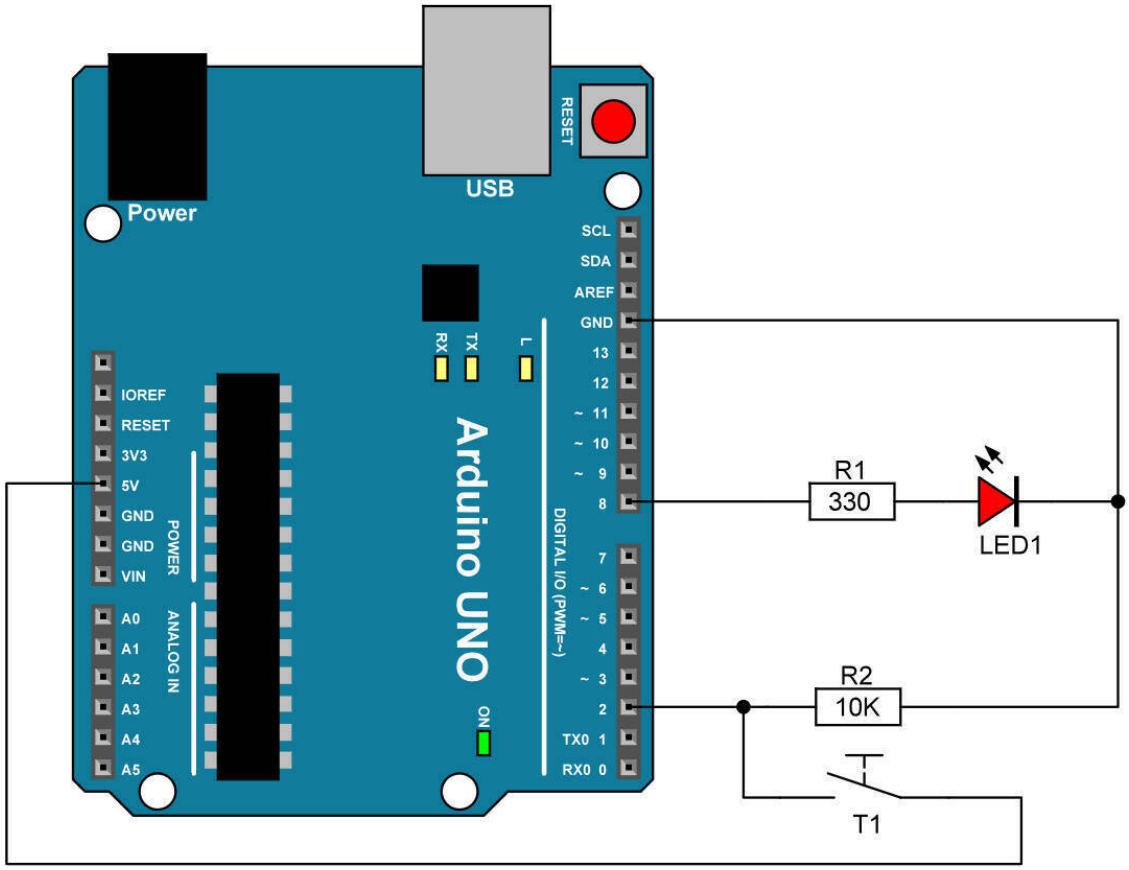



**Abb. 7:** Der Schaltplan für das Toggeln der LED über den Taster

## Der Schaltungsaufbau

Der Schaltungsaufbau gleicht dem aus dem Bastelprojekt 3 über die Abfrage des Tasterstatus. Mit diesen technischen Grundlagen wenden wir uns dem Sketch zu.

# Der Arduino-Sketch

Der folgende Sketch wird immer dann den Status der angeschlossenen LED wechseln, wenn der Taster gedrückt wird. Hält man den Taster für längere Zeit gedrückt, ändert sich der Status nach dem Wechsel nicht mehr.

```
int tasterPin = 2; // Taster-Pin 2 int ledPin = 8; // LED-Pin 8 int ledStatus; // LED-Status int tasterStatusActual; // Speichert aktuellen Tasterstatus int prevTasterStatus; // Speichert letzten Tastersstatus int debouncedTasterStatus; // Speichert debounced Tasterstatus int debounceInterval = 50; // 50ms Intervall unsigned long lastDebounceTime = 0; // Zeit, wann LED-Status sich // geändert hat void setup( ) { pinMode(ledPin, OUTPUT); // Taster-Pin als Eingang pinMode(tasterPin, INPUT); // LED-Pin als Ausgang } void loop() { tasterStatusActual = digitalRead(tasterPin); // Tasterstatus lesen unsigned long currentTime = millis(); // Debounce-Zeit lesen if(tasterStatusActual != prevTasterStatus) lastDebounceTime = currentTime; if(currentTime - lastDebounceTime > debounceInterval){ if(tasterStatusActual != debouncedTasterStatus){ debouncedTasterStatus = tasterStatusActual; // LED toggeln, wenn Tasterstatus gleich HIGH-Pegel if(debouncedTasterStatus == HIGH) ledStatus = !ledStatus; } } digitalWrite(ledPin, ledStatus); // LED ansteuern prevTasterStatus = tasterStatusActual; // Letzten Tasterstatus sichern }
```

Schauen wir uns die Erklärungen zu diesem Sketch an.

# Den Code verstehen

Ich möchte an diesem Beispiel zeigen, was ich über ein Warteschleifenverfahren realisiert habe. Zu Beginn der *loop*-Schleife wird immer der gerade vorherrschende Tastenpegel in die Variable *tasterStatusActual* eingelesen. Der gerade vorherrschende Zeitstempel seit Sketch-Start wird über die *millis*-Funktion in die Variable *currentTime* eingelesen. Wenn sich der aktuelle Tasterstatus vom vorherigen unterscheidet, wird die Debounce-Zeit auf den neuesten Stand gebracht. Diese wird in der folgenden Abfrage benötigt, um darüber zu entscheiden, ob das vorgegebene Intervall abgelaufen ist:

if(currentTime - lastDebounceTime > debounceInterval){...}

Ist das der Fall, wird der aktuelle Tasterstatus mit dem Debounced-Tasterstatus verglichen und erst wenn sie unterschiedlich sind, kommt es zur Abfrage, ob ein *HIGH*-Pegel zur Statusänderung vorliegt. Denn erst dann soll die LED in ihrem Status geändert werden, was über die Invertierung des Status in der Variablen *ledStatus* erfolgt:

if(debouncedTasterStatus == HIGH) ledStatus = !ledStatus;

Abschließend muss lediglich die LED über eben diese Variable *ledStatus* angesteuert werden:

digitalWrite(ledPin, ledStatus);

... und der aktuellen Tasterstatus in den vorherigen Status überführt werden:

prevTasterStatus = tasterStatusActual;

Das Spiel beginnt von vorn.

# Troubleshooting

Falls die LED beim Tasterdruck nicht leuchtet beziehungsweise nicht toggelt, kann es mehrere Gründe dafür geben:

Die LED ist verpolt, also falsch eingesteckt worden. Erinnere dich an die beiden unterschiedlichen Anschlüsse einer LED mit Anode und Kathode.

Die LED ist vielleicht defekt und durch Überspannung aus vorausgegangenen Bastelprojekten durchgebrannt. Teste sie mit einem Vorwiderstand an einer 5V-Spannungsquelle.

Kontrolliere noch einmal die Verbindungen auf deinen Breadboard, in die du die LED beziehungsweise die Bauteile eingesteckt hast.

Überprüfe noch einmal den Sketch, den du in den Editor der Entwicklungsumgebung eingegeben hast. Hast du eine Zeile vergessen oder hast du dich verschrieben? Und ist der Sketch wirklich korrekt übertragen worden?

Überprüfe die Funktionsfähigkeit des von dir verwendeten Tasters mit einem Durchgangsprüfer oder Multimeter.

# Was haben wir gelernt?

Du hast erfahren, dass mechanische Bauteile, Taster oder Schalter zum Beispiel, Kontakte nicht unmittelbar schließen oder öffnen. Durch verschiedene Faktoren wie Fertigungstoleranzen, Verunreinigungen oder schwingende Materialien können mehrere und kurz hintereinander folgende Unterbrechungen stattfinden, bevor ein stabiler Zustand erreicht wird. Dieses Verhalten wird vom Mikrocontroller registriert und entsprechend verarbeitet. Möchtest du zum Beispiel die Anzahl von Tastendrücken zählen, können sich solche Mehrfachimpulse als außerordentlich störend erweisen.

Du hast gelernt, was eine serielle Schnittstelle ist und wie diese beim Arduino arbeitet.

Das Prellverhalten kann durch unterschiedliche Ansätze kompensiert werden:Durch eine Softwarelösung (beispielsweise durch eine Verzögerungsstrategie beim Abfragen des Eingangssignals).Durch eine Hardwarelösung (zum Beispiel sogenannte RC-Glieder). Informationen dazu sind im Internet zu finden: http://www.mikrocontroller.net/articles/Entprellung.

# Bastelprojekt 6:
# Ein Lauflicht

Du hast jetzt schon einiges über die Ansteuerung von LEDs erfahren, so dass wir in kommenden Bastelprojekten die unterschiedlichsten Schaltungen aufbauen können, um mehrere Leuchtdioden blinken zu lassen. Das mag sich im Moment recht simpel anhören, aber lass dich überraschen, welche hübschen Sachen sich damit machen lassen. Wir wollen mit einem Lauflicht beginnen. Hierbei werden LEDs so angesteuert, dass sie nacheinander angehen und dabei der Effekt eines Lauflichts entsteht. Wenn du dich an das Bastelprojekt 2 erinnerst, haben wir so etwas bereits programmiert. Doch dabei ging es primär um die Ansteuerung der LEDs mithilfe der Ports durch Manipulation der korrespondierenden Register. In diesem Bastelprojekt möchte ich Vergleichbares zeigen, doch diesmal mit einem anderen Lösungsweg.

## Immer der Reihe nach

Die an den digitalen Pins angeschlossenen LEDs sollen nach dem folgenden Muster angesteuert werden:

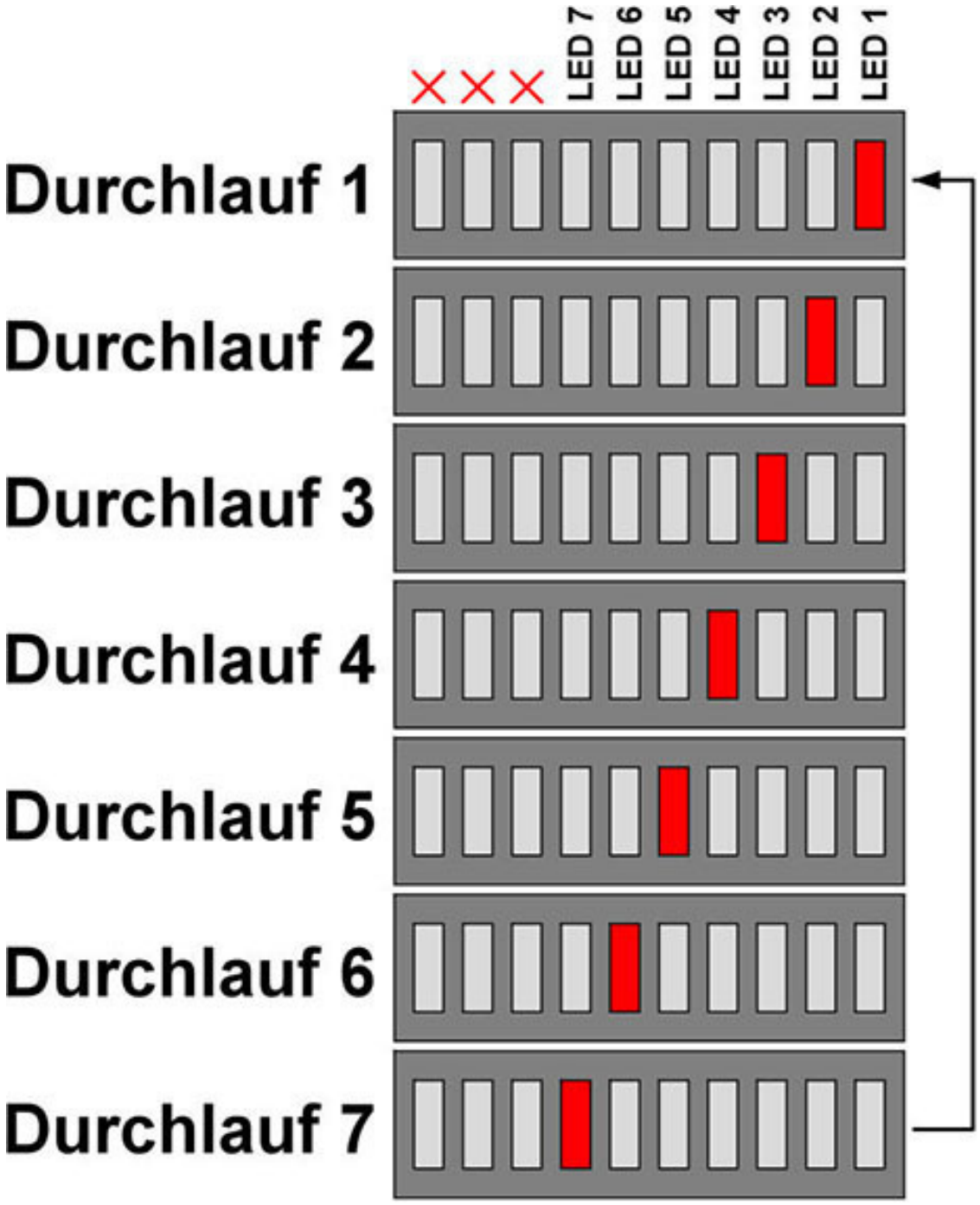



**Abb. 1:** Die Leuchtsequenz der sieben LEDs

Bei jedem neuen Durchlauf leuchtet also die LED eine Position weiter nach links. Ist das Ende erreicht, beginnt das Spiel von vorn. Du kannst die Programmierung der einzelnen Pins, die allesamt als Ausgänge arbeiten sollen, auf unterschiedliche Weise angehen. Mit dem Wissen, das du bisher hast, musst du sieben Variablen deklarieren und mit den entsprechenden Pin-Werten initialisieren. Das würde vielleicht wie folgt aussehen:

int ledPin1 = 7; int ledPin2 = 8; int ledPin3 = 9;

Und immer so weiter. Anschließend muss jeder einzelne Pin in der *setup*-Funktion mit *pinMode* als Ausgang programmiert werden, was ebenfalls eine mühsame Tipparbeit erfordert:

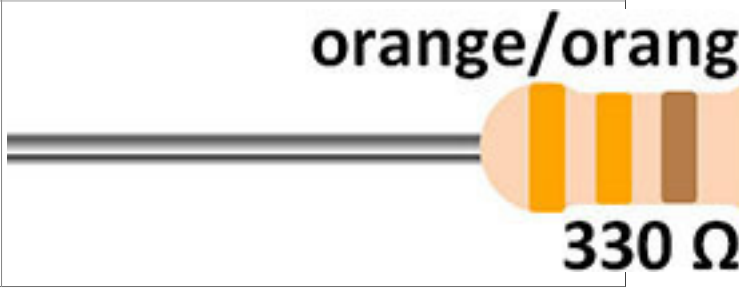
pinMode(ledPin1, OUTPUT); pinMode(ledPin2, OUTPUT); pinMode(ledPin3, OUTPUT);

Und so fort. Aber die Rettung naht! Ich möchte dir einen interessanten Variablentyp vorstellen, der in der Lage ist, mehrere Werte des gleichen Datentyps unter einem Namen zu speichern. Diese spezielle Form der Variablen nennt man *Array*. Der Zugriff darauf erfolgt nicht nur über den eindeutigen Namen, sondern eine solche Variable besitzt zusätzlich einen *Index*. Dieser Index ist eine Ganzzahl, die hochgezählt werden kann. Auf diese Weise werden die einzelnen Elemente des Arrays (so werden die gespeicherten Werte genannt) aufgerufen und geändert. Du wirst das im nun folgenden Sketch-Code sehen.

# Was wir brauchen

Für dieses Bastelprojekt benötigen wir die folgenden Bauteile:

Tabelle 1: Bauteilliste

| Bauteil | Bild |
|---|---|
| LED rot (oder auch grün) 7x |  |
| Widerstand 330Ω 7x |   |

# Der Schaltplan

Der Schaltplan gleicht dem der Abfrage eines Tasters aus dem vorherigen Bastelprojekt 2:

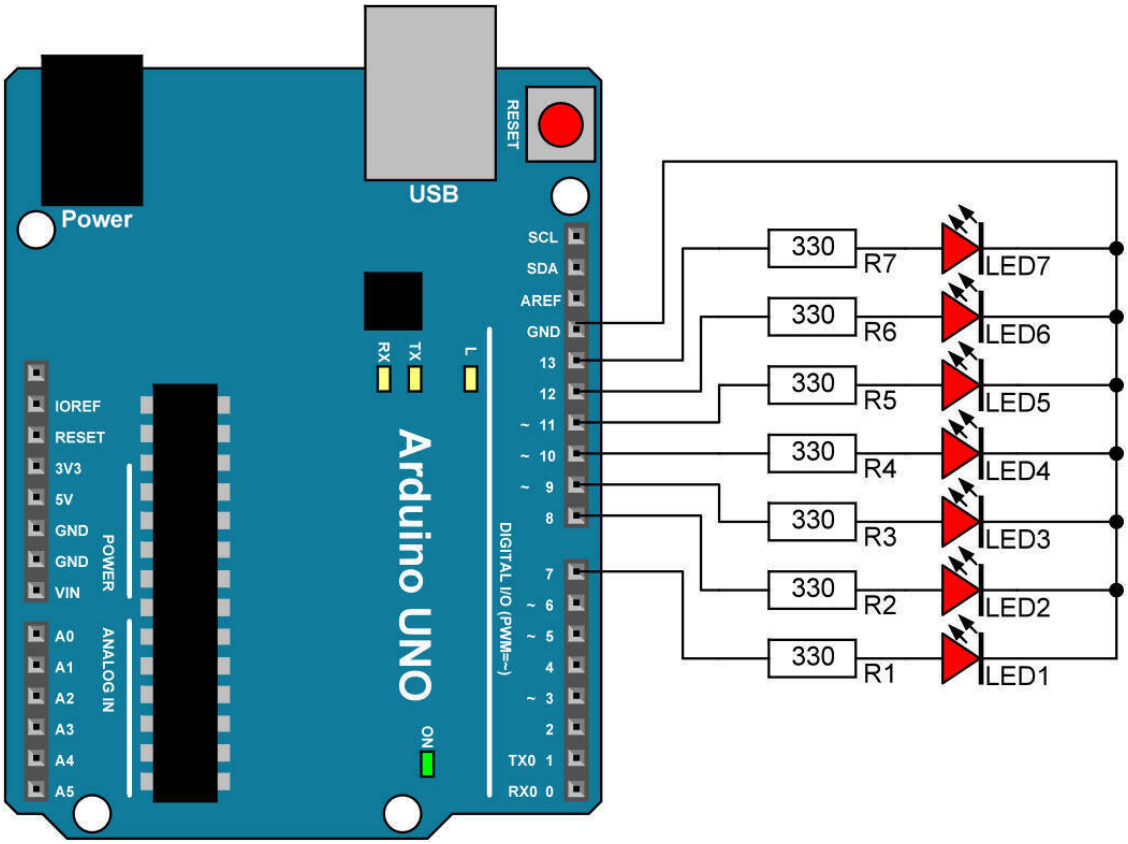



**Abb. 2:** Die Ansteuerung der sieben LEDs für das Lauflicht

# Der Schaltungsaufbau

Auch der Schaltungsaufbau ähnelt dem aus dem vorherigen Bastelprojekt und das Arduino Discoveryboard mit seinen zehn LEDs im Block leistet auch hier wieder sehr gute Dienste. Es werden hierbei nur sieben der LEDs angesteuert.

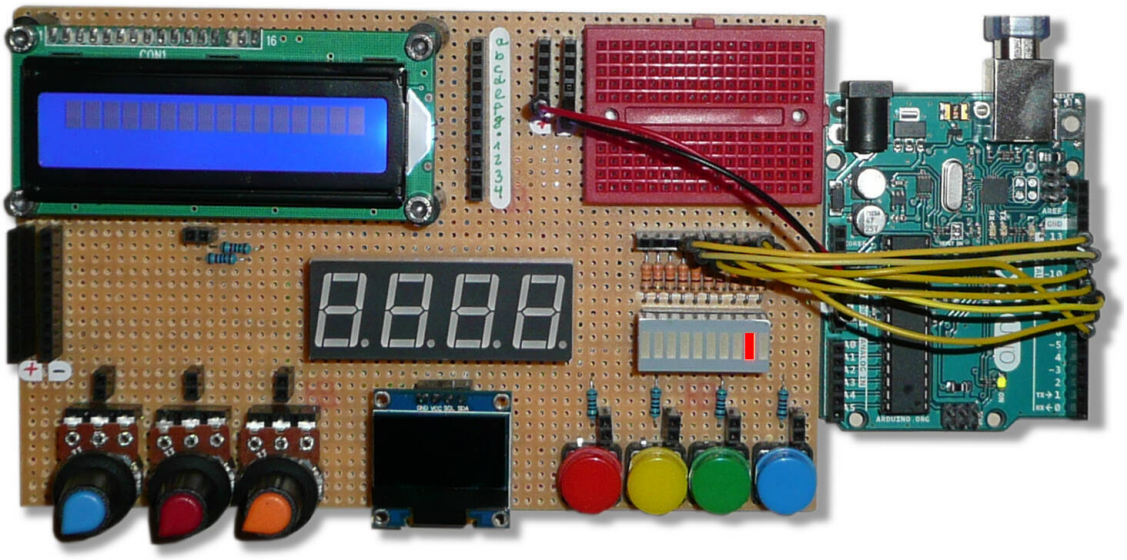


**Abb. 3:** Der Schaltungsaufbau für das Lauflicht mit sieben LEDs

# Der Arduino-Sketch

Der Sketch für das Lauflicht sieht wie folgt aus:

```
int ledPin[] = {7, 8, 9, 10, 11, 12, 13}; // LED-Array mit Pin-Werten int wartezeit = 200; // Pause zwischen den Wechseln in ms void setup() { for(int i = 0; i < 7; i++) pinMode(ledPin[i], OUTPUT); // Alle Pins des Arrays als Ausgang } void loop() { for(int i = 0; i < 7; i++) { digitalWrite(ledPin[i], HIGH); // Array-Element auf HIGH-Pegel delay(wartezeit); // Eine Pause zwischen den Wechseln digitalWrite(ledPin[i], LOW); // Array-Element auf LOW-Pegel } }
```

Schauen wir uns die Erklärungen zu diesem Sketch an, denn wir haben es mit einigen programmtechnischen Neuerungen zu tun.

## Den Code verstehen

Im Lauflicht-Sketch begegnest du zum ersten Mal einem Array und einer Schleife. Die Schleife wird benötigt, um komfortabel die einzelnen Array-Elemente über die darin enthaltenen Pin-Nummern anzusprechen. Es werden so zum einen alle Pins als Ausgänge programmiert und zum anderen die digitalen Ausgänge ausgelesen. Ich hatte erwähnt, dass jedes einzelne Element über einen Index angesprochen wird, und da die Schleife, die wir hier nutzen, einen bestimmten Wertebereich verwendet, ist dieses Konstrukt für unsere Aufgabe geeignet. Beginnen sollten wir mit der Array-Variablen. Die Deklaration ähnelt der bei einer ganz normalen Variablen, wobei aber zusätzlich das eckige Klammerpaar hinter dem Namen erforderlich ist.


**Abb. 4:** Die Array-Deklaration

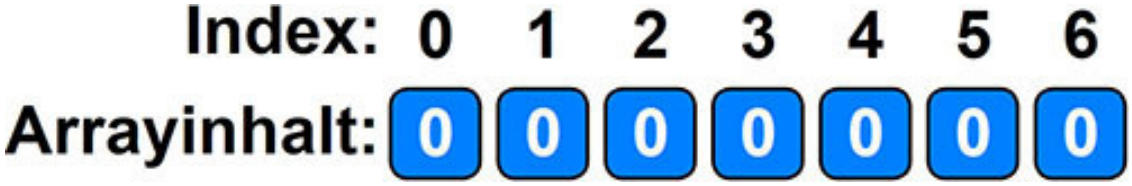
## Welche Dinge sind zu beachten?

Der Datentyp legt fest, welchen Typ die einzelnen Array-Elemente haben sollen.

Der Array-Name ist ein eindeutiger Name für den Zugriff auf die Variable.

Das Kennzeichen für das Array sind die eckigen Klammern mit der Größenangabe, wie viele Elemente das Array aufnehmen soll.

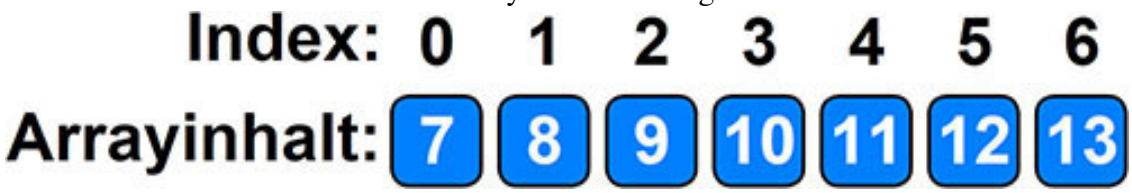
Du kannst dir ein Array wie einen Schrank mit mehreren Schubladen vorstellen. Jede einzelne Schublade trägt ein Schildchen mit einer fortlaufenden Nummer auf der Außenseite. Wenn ich dir die Anweisung gebe, doch bitte die Schublade mit der Nummer 3 zu öffnen, um zu sehen, was drin ist, dann ist das eine eindeutige Anweisung, oder? Ähnlich verhält es sich bei einem Array.




Bei dem hier gezeigten Array wurden nach der Deklaration alle Elemente implizit mit dem Wert 0 initialisiert. Die Initialisierung kann jedoch explizit auf zwei unterschiedliche Weisen erfolgen. Wir haben den komfortablen Weg gewählt und die Werte, mit denen das Array versehen werden soll, in geschweiften Klammern hinter der Deklaration, durch Komma separiert, aufgelistet.

```
int ledPin[] = {7, 8, 9, 10, 11, 12, 13};
```

Nach dieser Befehlszeile sieht der Array-Inhalt wie folgt aus:




Bei der Deklaration des Arrays ist das eckige Klammernpaar leer. Dort sollte doch die Größe des Arrays angegeben sein. Warum ist das so? In diesem Fall erkennt der Compiler anhand der mitgelieferten Informationen bei der Initialisierung, die ja in derselben Zeile erfolgt, um wie viele Elemente es sich handelt. Aus diesem Grund kannst du sie weglassen. Die etwas aufwendigere Art der Initialisierung besteht darin, die einzelnen Werte jedem Array-Element explizit zuzuweisen:

```
int ledPin[7]; // Deklaration des Arrays mit 7 Elementen void setup() { ledPin[0] = 7; ledPin[1] = 8; ledPin[2] = 9; ledPin[3] = 10; ledPin[4] = 11; ledPin[5] = 12; ledPin[6] = 13; // ... }
```

Was ist beim Index zu beachten?

Das erste Array-Element hat immer den Index mit der Nummer 0. Deklarierst du etwa ein Array mit zehn Elementen, dann ist der höchste zulässige Index der mit der Nummer 9, also immer eins weniger als die Anzahl der Elemente. Hältst du dich nicht daran, dann provozierst du möglicherweise einen Laufzeitfehler, denn der Compiler, der hinter der Entwicklungsumgebung

steckt, bemerkt das weder zur Entwicklungszeit noch später zur Laufzeit, und deshalb solltest du doppelte Sorgfalt walten lassen.

Kommen wir jetzt zur Schleife und schauen uns die Syntax ein wenig genauer an.

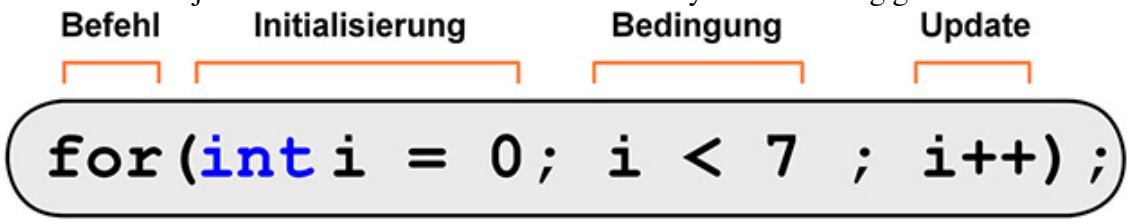



**Abb. 5:** Die for-Schleife

Die Schleife wird mit dem Schlüsselwort *for* eingeleitet und wird deswegen auch *for*-Schleife genannt. Ihr werden, in runde Klammern eingeschlossen, bestimmte Informationen geliefert, die Auskunft über folgende Eckpunkte geben:

Mit welchem Wert soll die Schleife beim Zählen beginnen (Initialisierung)?
Wie weit soll gezählt werden (Bedingung oder Test)?
Um welchen Betrag soll der ursprüngliche Wert verändert werden (Update)?

Diese drei Informationseinheiten legen das Verhalten der *for*-Schleife fest und bestimmen ihr Verhalten beim Aufruf.

Wann kommt eine for-Schleife zum Einsatz?

Eine *for*-Schleife kommt meistens dann zum Einsatz, wenn von vornherein bekannt ist, wie oft bestimmte Anweisungen ausgeführt werden sollen. Diese Eckdaten werden im sogenannten Schleifenkopf, der von runden Klammern umschlossen ist, definiert.

Aber werden wir etwas konkreter. Die folgende Codezeile deklariert und initialisiert eine Variable *i* vom Datentyp *int* mit dem Wert *0*:

for(int i = 0; i < 7; i++)

Die Angabe des Datentyps innerhalb der Schleife besagt, dass es sich um eine *lokale Variable* handelt, die nur so lange existiert, wie die *for*-Schleife iteriert, also ihren Durchlauf hat. Beim Verlassen der Schleife wird die Variable *i* aus dem Speicher entfernt. Die genaue Bezeichnung für eine Variable innerhalb einer Schleife lautet *Laufvariable*. Sie durchläuft so lange einen Bereich, wie die Bedingung (*i < 7*) erfüllt ist, die hier mit Test bezeichnet wurde. Anschließend erfolgt ein Update der Variablen durch den Update-Ausdruck. Der Ausdruck *i++* erhöht die Variable *i* um den Wert *1*.

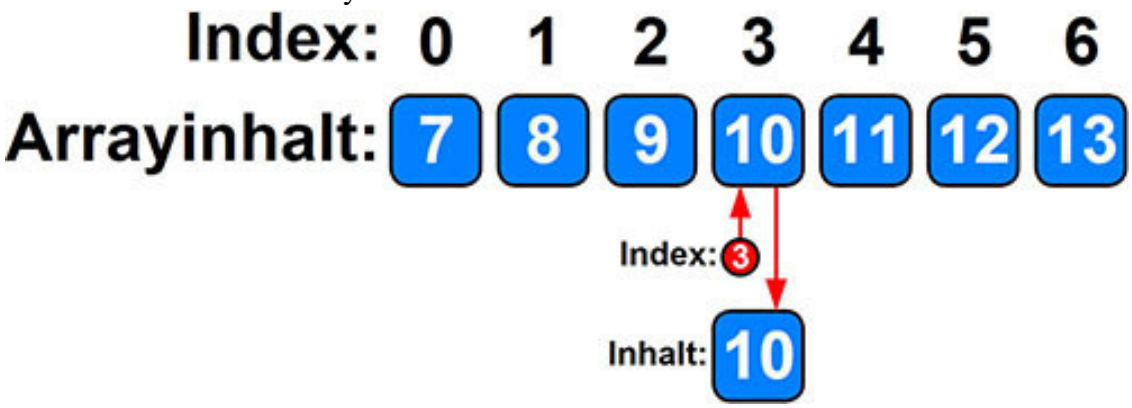
Wir haben den Ausdruck *i++* verwendet. Was bedeutet das genau? Er soll den Wert um 1 erhöhen, doch die Schreibweise ist irgendwie komisch. Bei den beiden hintereinander angeführten Pluszeichen ++ handelt es sich um einen Operator, der den Inhalt des Operanden, also der Variablen, um den Wert 1 erhöht. Programmierer sind von Haus aus faule Zeitgenossen und versuchen alles, was eingetippt werden muss, irgendwie kürzer zu formulieren. Wenn man bedenkt, wie viele Codezeilen ein Programmierer in seinem Leben eingeben muss, kommt es da auf jeden Tastendruck an. In der Summe könnte es sich um Monate oder Jahre an Lebenszeit handeln, die sich durch kürzere Schreibweisen einsparen lassen und für wichtigere Dinge, wie noch mehr Code, genutzt werden könnten. Jedenfalls sind die beiden folgenden Ausdrücke in ihren Auswirkungen vollkommen identisch:

i++;

und

i = i + 1;

Es wurden zwei Zeichen weniger verwendet, was eine Einsparung von immerhin 40% ausmacht. Doch weiter im Text. Die Laufvariable *i* wird als Indexvariable im Array eingesetzt und spricht somit die einzelnen Array-Elemente nacheinander an.




Bei diesem Snapshot eines Schleifendurchlaufs hat die Variable *i* den Wert *3* und spricht somit das vierte Element an, das wiederum den Inhalt *10* besitzt. Das bedeutet, dass mit den zwei folgenden Zeilen innerhalb der *setup*-Funktion alle im Array *ledPin* hinterlegten Pins als Ausgänge programmiert werden:

for(int i = 0; i < 7; i++) pinMode(ledPin[i], OUTPUT);

Folgendes ist noch sehr wichtig zu erwähnen: Wenn keine Blockbildung mit einer *for*-Schleife mittels geschweifter Klammern stattfindet, wie wir es gleich in der *loop*-Funktion sehen werden, wird nur die Zeile, die der *for*-Schleife unmittelbar folgt, von dieser berücksichtigt. Der Code der *loop*-Funktion beinhaltet lediglich eine *for*-Schleife, die durch ihre Blockstruktur jetzt mehrere Befehle anspricht:

for(int i = 0; i < 7; i++) { digitalWrite(ledPin[i], HIGH); // Array-Element auf HIGH-Pegel delay(wartezeit); digitalWrite(ledPin[i], LOW); // Array-Element auf LOW-Pegel }

Ich möchte dir an einem kurzen Sketch zeigen, wie die Laufvariable *i* heraufgezählt, was man auch Inkrementieren nennt:

void setup() { Serial.begin(9600); // Serielle Schnittstelle konfigurieren for(int i = 0; i < 7; i++) Serial.println(i); // Ausgabe an die serielle Schnittstelle } void loop(){ /* leer */ }

Da unser Arduino von Haus aus kein Ausgabefenster besitzt, müssen wir uns etwas anderes einfallen lassen. Die serielle Schnittstelle, an der er quasi angeschlossen ist, können wir dazu nutzen, Daten zu versenden. Die Entwicklungsumgebung verfügt über einen *Serial Monitor*, der diese Daten bequem empfangen und darstellen kann. Du kannst ihn sogar dazu verwenden, Daten an das Arduino-Board zu schicken, die anschließend dort verarbeitet werden können. Doch dazu gleich mehr. Der folgende Befehl initialisiert die serielle Schnittstelle mit einer Übertragungsrate von *9600 Baud*:

Serial.begin(9600);

Die folgende Zeile sendet dann mittels der *println*-Funktion den Wert der Variablen *i* an die Schnittstelle:

Serial.println(i);

Du musst jetzt lediglich den Serial Monitor öffnen und die Werte werden angezeigt (Abbildung 6).

Du siehst hier, wie die Werte der Laufvariablen *i* von *0* bis *6* ausgegeben werden, die wir in unserem eigentlichen Sketch zur Auswahl der Array-Elemente benötigen. Ich habe den Code innerhalb der *setup*-Funktion platziert, damit die *for*-Schleife nur einmal ausgeführt wird und die Anzeige nicht ständig durchläuft.

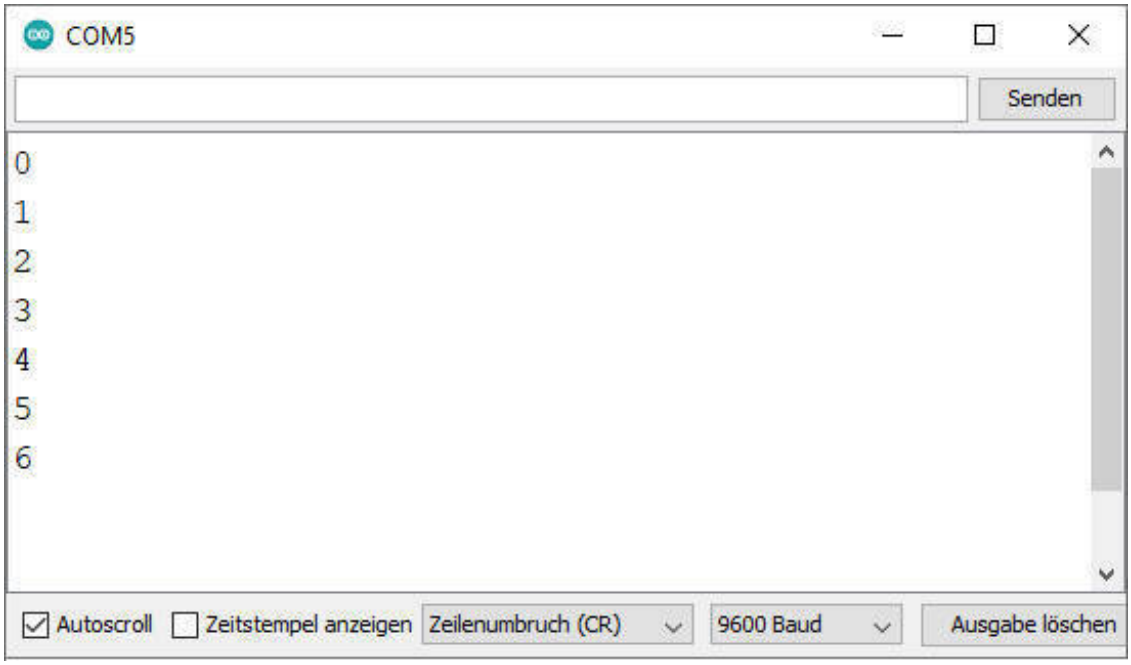



**Abb. 6:** Die Ausgabe der Werte im Serial Monitor

Die folgende Abbildung 7 zeigt dir die einzelnen Durchläufe der *for*-Schleife etwas genauer:

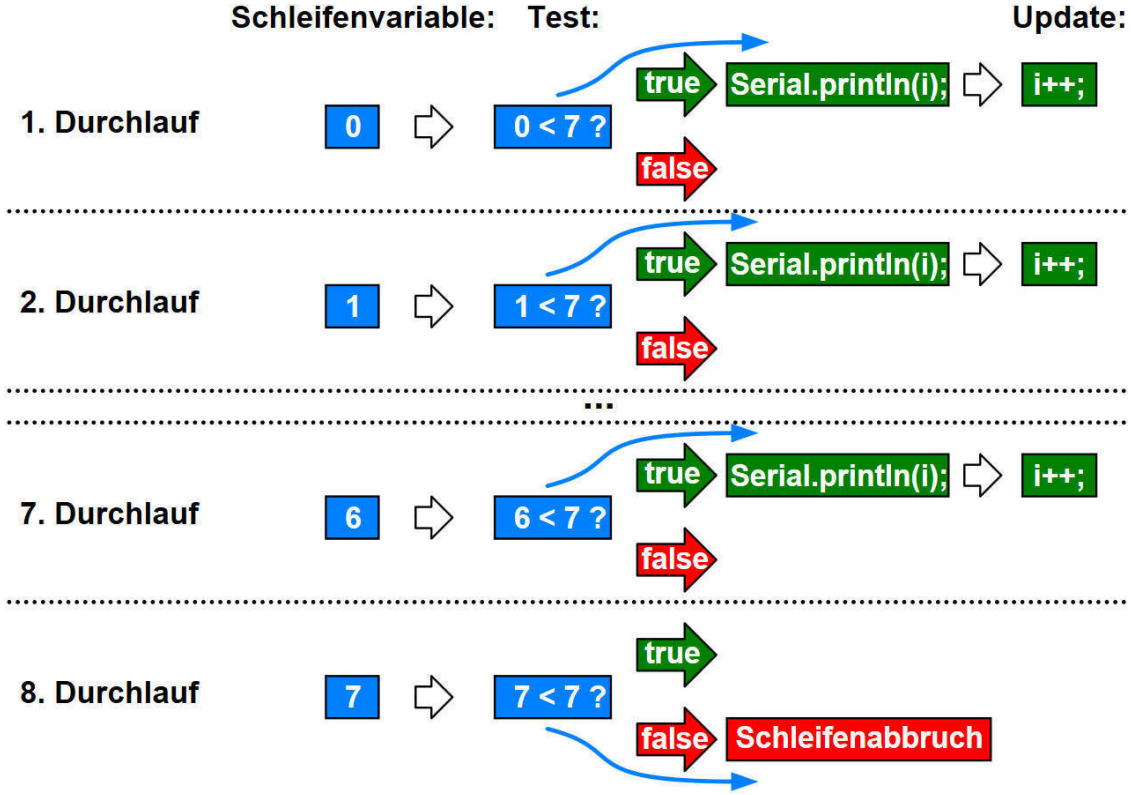



**Abb. 7:** Das Verhalten der for-Schleife

Wie die serielle Schnittstelle zu konfigurieren ist und wie man etwas dahin versendet, hast du schon gesehen. Die Methode *begin* initialisiert das Serial-Objekt mit der angeforderten Übertragungsrate und die Methode *println* (*print line* bedeutet so viel wie *Drucke und mache einen Zeilenvorschub*) gibt etwas auf der seriellen Schnittstelle aus. Das Bindeglied zwischen Objekt und Methode ist der Punktoperator (.), der beide verbindet.

Die serielle Schnittstelle bei der Fehlersuche

Du hast jetzt erfahren, wie etwas an die serielle Schnittstelle geschickt werden kann. Du kannst dir diesen Umstand zunutze machen, wenn du einen oder mehrere Fehler in einem Sketch finden möchtest. Funktioniert der Sketch nicht so, wie du es dir vorstellst, dann positioniere an unterschiedlichen Stellen im Code, die dir wichtig erscheinen, Ausgabebefehle in Form von *Serial.println(...)*; und lass dir bestimmte Variableninhalte oder auch Texte ausgeben. Auf diese Weise erfährst du, was dein Sketch treibt und warum er möglicherweise nicht korrekt abläuft. Du musst lediglich lernen, die ausgegebenen Daten zu interpretieren. Das ist manchmal nicht so einfach und es gehört ein wenig Übung dazu.

# Register direkt beeinflussen

Im Bastelprojekt 2 über die Low-Level-Programmierung des Arduino haben wir gesehen, wie einfach es ist, die digitalen Pins über die Manipulation von Registern zu beeinflussen. Hinsichtlich des Lauflichts kann man sich die Sache zunutze machen. Der nachfolgende Schaltplan verfügt lediglich über sechs LEDs mit den entsprechenden Vorwiderständen, wobei ich aber jeder LED eine kleine Zahl beigestellt habe. Wozu das ganze sinnvoll ist, werden wir gleich sehen.

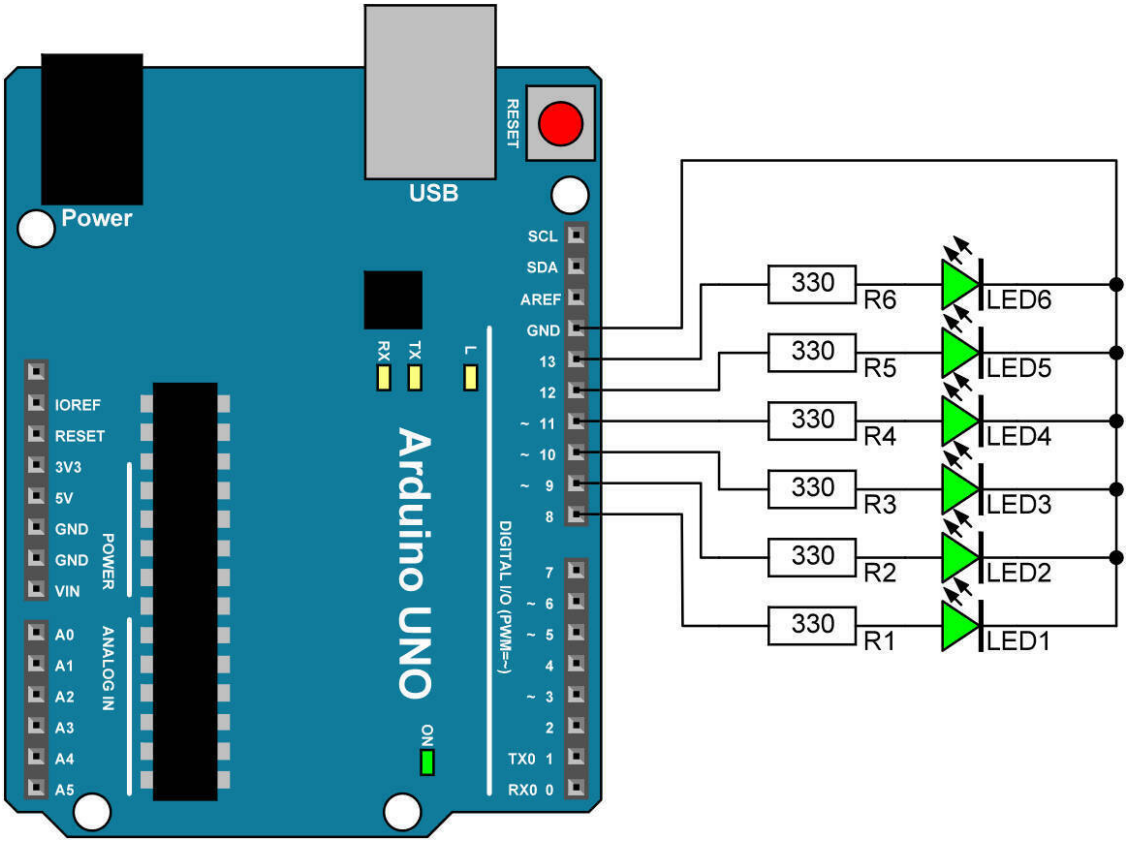
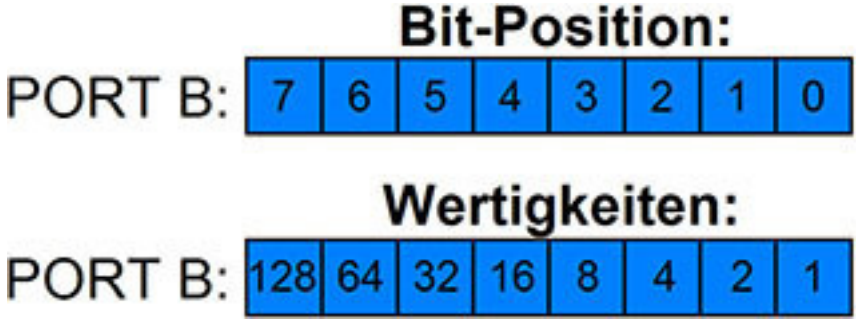



**Abb. 8:** Der Schaltplan für unser kleines Lauflicht

Werfen wir noch einmal einen Blick auf das Register PORT B.




Die acht Bits eines Ports werden zu einem sogenannten *Byte* zusammengefasst. Jedes einzelne Bit dieses Bytes besitzt eine Nummer, die von rechts nach links aufsteigend bei 0 beginnt, wie das in der oberen Reihe zu erkennen ist. Somit können wir jedes einzelne Bit eindeutig von 0 bis 7 adressieren. Jedes einzelne Bit besitzt neben seiner Position innerhalb des Bytes einen Stellenwert oder eine Wertigkeit, die eben genau von der betreffenden Position innerhalb des Bytes abhängt und

ebenfalls von rechts nach links zunimmt. Die untere Reihe zeigt die Wertigkeit jedes einzelnen Bits an. Die Frage ist nur, wie diese Werte eigentlich zustande kommen. Das ist sehr einfach! Da das binäre System lediglich die beiden Zustände 0 und 1 kennt, ist die Basis zur Berechnung der Stellenwertigkeit die Zahl 2. Wir erinnern uns, dass unser Dezimalsystem die Ziffern 0 bis 9 kennt und demnach zehn mögliche Zustände vorhanden sind. Die Basis zur Berechnung der Stellenwertigkeit ist demnach die Zahl 10.

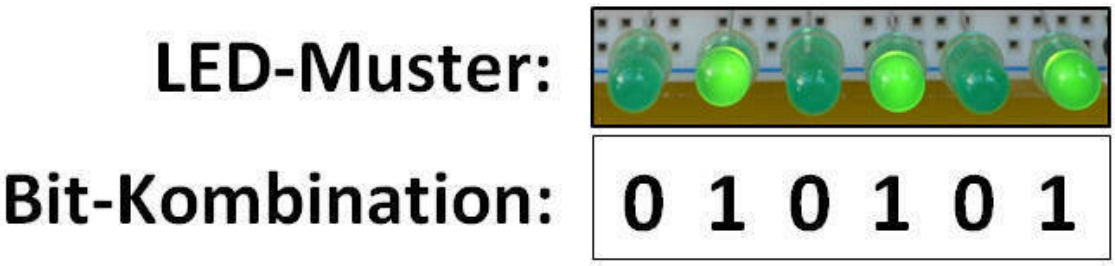
Doch kommen wir zurück zum binären System. Wie berechnet man die Stellenwertigkeit? Das erfolgt nach folgender Formel:

$$Wertigkeit = 2^{Position}$$

Wie groß ist also die Wertigkeit des Bits an der Position 4? Dann wollen wir mal sehen:

$$Wertigkeit = 2^4 = 16$$

Sehen wir uns jetzt die Variante an, die mithilfe der Register- beziehungsweise Bit-Manipulation realisiert wird. Um das gewünschte LED-Muster zu erzeugen, muss lediglich an den Stellen innerhalb des Bytes eine 1 stehen, an denen die LEDs leuchten sollen. Zum besseren optischen Verständnis habe ich hier anstelle der LED-Bar, wie sie auf dem Arduino Discoveryboard zum Einsatz kommt, richtige LEDs mit 5mm Durchmesser verwendet, was du natürlich ebenfalls machen kannst:




Der entsprechende Sketch-Code dazu sieht wie folgt aus:

```
void setup() { DDRB = 0b11111111; // PORT B komplett als OUTPUT PORTB = 0b00010101; // Erzeugung des LED-Musters } void loop() { /* leer */ }
```

Wir sehen, dass die Zeile mit der verwendeten Bit-Kombination

```
PORTB = 0b00010101;
```

genau dem LED-Muster entspricht. Natürlich muss bei der Initialisierung keine Binärzahl verwendet werden. Es funktioniert auch mit einer Ganzzahl. Machen wir ein kleines Experiment und lassen uns ein paar Bit-Kombinationen von 0 bis 63 anzeigen. Dazu muss lediglich ein Wert immer um 1 inkrementiert – also erhöht werden. Diesmal muss sich aber ein Teil des Codes innerhalb der *loop*-Funktion befinden, denn es soll ja in regelmäßigen Zeitabständen eine Änderung bewirkt werden:

```
byte pattern = 0; void setup() { DDRB = 0b11111111; // PORT B komplett als OUTPUT } void loop() { PORTB = pattern++; // Inkrementieren von pattern delay(100); // Kurze Pause von 100ms }
```

Die Variable *pattern* ist vom Datentyp *byte* und hat somit acht Bits und kann Werte von 0 bis 255 speichern. Wenn wir sie innerhalb der *loop*-Funktion inkrementieren, dann wird irgendwann einmal ein sogenannter Überlauf erfolgen, wenn der Inhalt der Variablen 255 beträgt und anschließend noch einmal der Wert 1 addiert wird. Das funktioniert natürlich nicht, denn das Fassungsvermögen ist erschöpft. Es erfolgt der genannte Überlauf und das Spiel beginnt bei 0 von vorn. Mit folgender

Zeile erfolgt das Inkrementieren der Variablen, wobei der *Inkrement-Operator*, der durch die beiden Pluszeichen repräsentiert wird, die Aufgabe des Hochzählens übernimmt:

PORTB = pattern++;

Man hätte das Gleiche auch mit den folgenden Zeilen erreicht:

pattern = pattern + 1; PORTB = pattern;

Das ist natürlich Geschmackssache, aber Programmierer lieben es, Dinge zu verkürzen und so wenig Code wie möglich zu formulieren.

# Die Bit-Manipulation

Kommen wir nun zu einem sehr interessanten Teil: der *Bit-Manipulation*. Es ist möglich, die Bits über geeignete Operatoren vielfaltig zu manipulieren, was jedoch ein wenig Übung erfordert. Aber wenn man sich diese Art der Programmierung erst einmal erschlossen ist, dann macht es richtig Spaß, die Bits in jeglicher Weise zu verbiegen. Da das Thema dieses Bastelprojektes ja ein Lauflicht ist (was später noch etwas anders als hier realisiert wird), wollen wir doch einmal sehen, wie es möglich ist, eine einzelne LED »wandern« zu lassen. Wir nutzen dazu die sogenannten *Bit-Operatoren*.

# Schiebeoperatoren

Damit eine einzelne LED von einem Bit zum anderen geschoben wird, nutzen wir einen der folgenden Schiebeoperatoren:

>> Bedeutet nach rechts schieben.
<< Bedeutet nach links schieben.

Wie soll das funktionieren? Sehen wir uns dazu die folgenden Inhalte von PORT B und was mit ihnen im Laufe der Zeit passiert:

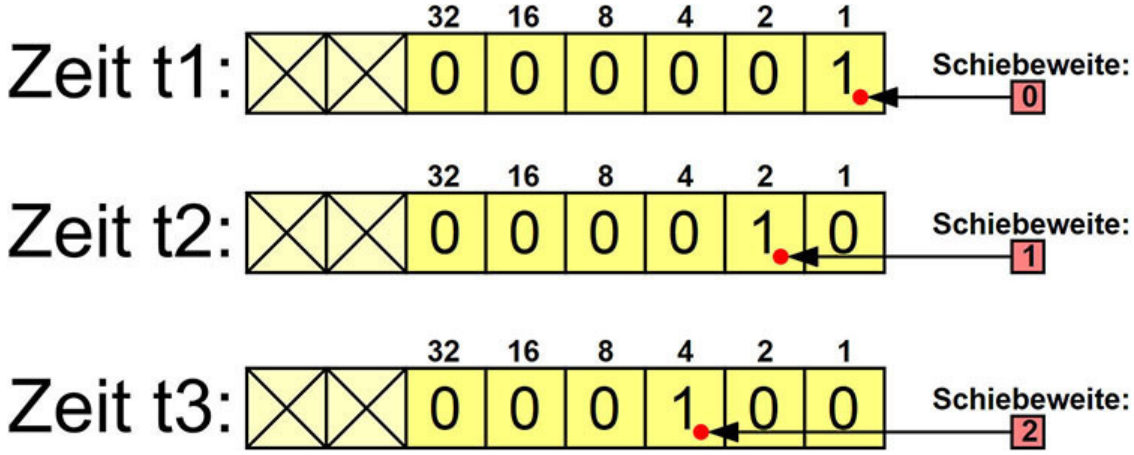



**Abb. 9:** Die Inhalte von Register PORT B zu unterschiedlichen Zeiten

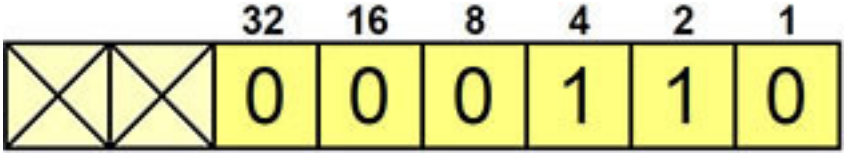
Man kann sehen, dass die 1 rechts außen schrittweise nach links wandert. Der kleine rote Punkt markiert die Endposition zur angegebenen Zeit. Wie können wir aber diese 1 von rechts nach links schieben? Der angesprochene Schiebeoperator für das nach links Schieben wird uns gute Dienste leisten. Wir gehen davon aus, dass die 1 auf der rechten Außenposition, die – wir erinnern uns – *LSB* (Least Significant Bit) genannt wird und das Bit mit dem niedrigsten Wert kennzeichnet, immer als Ausgangsposition für alle Schiebeoperationen genommen wird. Der folgende Sketch-Code übernimmt diese Funktion des Schiebens:

byte pos = 0; // Positionswert void setup() { DDRB = 0b11111111; // PORT B komplett als OUTPUT } void loop() { PORTB = 1 << pos++; // Die 1 nach links schieben if(pos > 5) pos = 0; delay(500); // Kurze Pause von 500ms }

Die Variable tritt in der Funktion als *Schiebeweitenangeber* in Erscheinung. Zu Beginn hat sie den Wert 0, was bedeutet, dass beim ersten Schleifendurchlauf die 1 auf ihrer Position bleibt, wie das auch bei Zeitmarke *t1* der Fall ist. Nach der Abarbeitung des Befehls wird die Variable *pos* um den Wert 1 erhöht, was wiederum bedeutet, dass beim nächsten Schleifendurchlauf eine Schiebeaktion nach links um eine Position bedeutet. Es muss jedoch darauf hingewiesen werden, dass durch das Schieben nach links auf der rechten Seite eine 0 eingeschoben wird. In gleicher Weise wird bei jedem erneuten Durchlauf verfahren. Ist der Wert von *pos* jedoch größer 5, was außerhalb unserer LED-Darstellungsmöglichkeit von sechs Bits liegt, wird er mithilfe der *if*-Anweisung und des nachfolgenden Befehls auf den Wert 0 zurückgesetzt und das Spiel beginnt von vorn.

# Einzelne Bits setzen

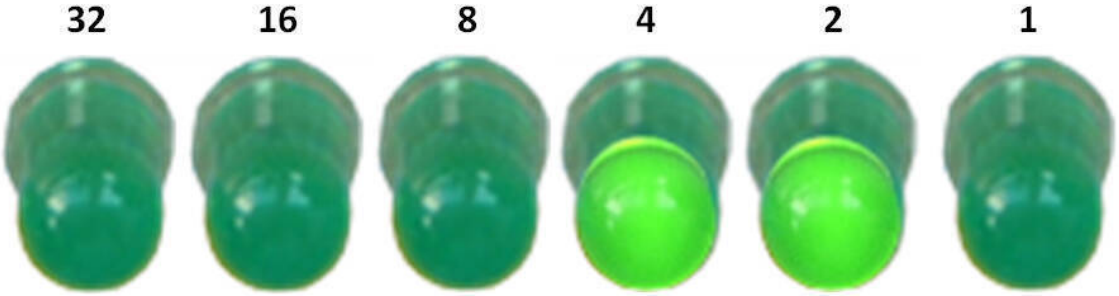
Jetzt wollen wir einmal sehen, wie es möglich ist, einzelne Bits zu setzen, also mit dem Wert 1 zu versehen, ohne bestehende Bits in ihrem aktuellen Zustand zu beeinflussen. Schauen wir uns die folgende Situation an, wobei die gezeigte Bit-Kombination die Ausgangs-Bit-Maske (Folge von Bits) darstellt:



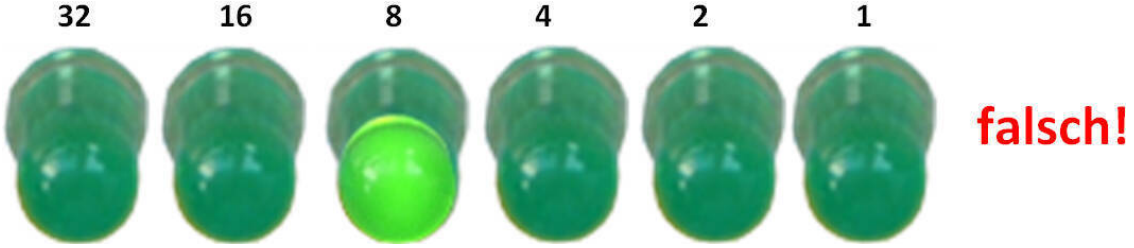

Die Aufgabe ist es jetzt, das Bit mit der Wertigkeit 8 an Position 3 zu setzen. Wir könnten den folgenden Sketch-Code schreiben, um diese Anforderung zu realisieren:

```
void setup() { DDRB = 0b11111111; // PORT B komplett als OUTPUT PORTB = 0b00000110; // Ausgangs Bit-Maske delay(500); PORTB = 1 << 3; // 3 Positionen nach links schieben } void loop() { /* leer */ }
```

Die Ausgangs-Bit-Maske, die 500ms lang zu sehen ist, sieht also wie folgt aus:




Im Anschluss soll dann die LED mit der Wertigkeit 8 zusätzlich zu den schon leuchtenden angehen. Doch was ist das Ergebnis? Schau mal:




Was ist schiefgelaufen? Nun, wir haben eine 1 rechts außen auf das LSB gesetzt und dann diese um 3 Positionen nach links geschoben. Wir erinnern uns, dass beim Schieben immer eine 0 an die vorherige Position eingeschoben wird. Für unser Vorhaben nicht so gut, denke ich. Was also tun? Die Lösung verbirgt sich hinter einem weiteren Operator, der aus der Kategorie der Bit-Operatoren entnommen wird. Es handelt sich um das binäre *ODER*. Dazu sollten wir uns jedoch die folgende Wertetabelle anschauen, damit wir erkennen, welche Auswirkungen dieser Operator hat:

| Tabelle 2: Der binäre ODER-Operator | | |
|---|---|---|
| A | B | Q |
| 0 | 0 | 0 |

| | | |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 1 |

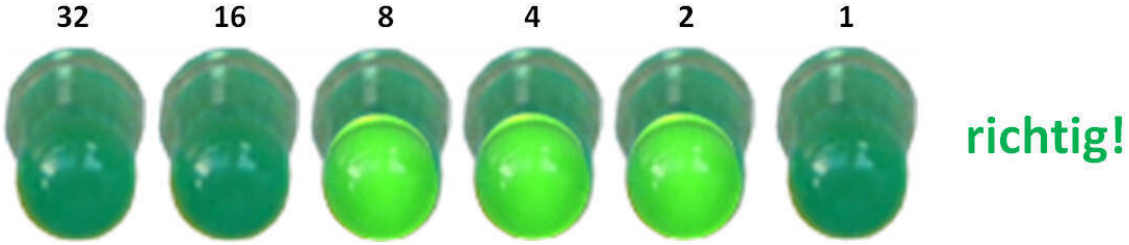
Die Spalten *A* und *B* zeigen zwei logische Ausgangswerte und *Q* das Ergebnis der binären ODER-Verknüpfung. Der binäre ODER-Operator wird durch den senkrechten Strich | (Pipe-Symbol) gekennzeichnet. Ändern wir nun die folgende Zeile

```
PORTB = 1 << 3;
```

in

```
PORTB |= 1 << 3;
```

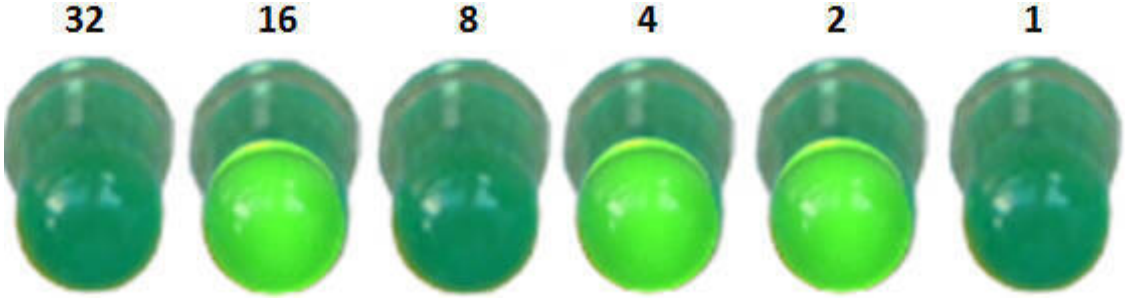
ab, dann funktioniert auch unsere Anforderung, das entsprechende Bit zu setzen, ohne die vorhandenen zu beeinflussen. Das Ergebnis sieht dann wie folgt aus:




Warum ist das aber so? Nun, wenn eine Schiebeaktion schrittweise immer wieder mit der vorherigen Bit-Maske in PORT B mit ODER verknüpft wird, dann bleiben alle Bits, die eine 0 enthalten, unverändert und alle, die eine 1 haben, werden gesetzt.

# Einzelne Bits löschen

Gesetzte Bits können natürlich auch wieder gelöscht werden. Nehmen wir an, es würde die folgende Bit-Kombination vorliegen, die mit der nachfolgenden Codezeile erreicht wird:




PORTB = 0b0010110;

Nun möchten wir das Bit mit der Wertigkeit 2 an Position 1 löschen, so dass dort die LED ausgeht. Alle anderen sollen natürlich davon unbeeindruckt ihren Zustand behalten. Dazu verwenden wir wieder einen neuen Operator aus der Kategorie der Bit-Operatoren, den binären *UND-Operator*. Zum besseren Verständnis dazu die entsprechende Wertetabelle:

| Tabelle 3: Der binäre UND-Operator | | |
|---|---|---|
| A | B | Q |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Das Ergebnis dieser Verknüpfung ist nur 1, wenn beide Eingangswerte den Wert 1 aufweisen. Der binäre UND-Operator wird durch das Kaufmanns-Und & ausgedrückt. Alle Bits, die in der Verknüpfungsmaske den Wert 0 haben, werden gelöscht, die eine 1 haben, bleiben unverändert. Der Sketch-Code dazu sieht wie folgt aus:

void setup() { DDRB = 0b11111111; // PORT B komplett als OUTPUT PORTB = 0b00010110; // Ausgangs-Bit-Maske delay(500); PORTB &= 0b11111101; } void loop() { /* leer */ }

Durch die Zeile

PORTB &= 0b11111**0**1;

ist quasi eine Maske erstellt worden und an der Stelle, an der sich die 0 befindet, gibt es kein Durchkommen für bestehende Bits.

# Troubleshooting

Falls die LEDs nicht nacheinander zu leuchten beginnen, dann trenne das Board sicherheitshalber vom USB-Anschluss und geh bitte folgende Dinge durch:

Überprüfe deine Steckverbindungen auf dem Breadboard daraufhin, ob sie wirklich der Schaltung entsprechen.

Achte auf mögliche Kurzschlüsse.

Sind die LEDs richtig herum eingesteckt worden? Denk an die richtige Polung!

Haben die Widerstände die korrekten Werte?

Überprüfe noch einmal den Sketch-Code auf Richtigkeit.

# Was haben wir gelernt?

Du hast eine Sonderform einer Variablen kennengelernt, die es dir ermöglicht, mehrere Werte des gleichen Datentyps aufzunehmen. Sie wird Array-Variable genannt. Ihre einzelnen Elemente werden durch einen Index angesprochen.

Die for-Schleife ermöglicht es dir, eine oder mehrere Codezeilen mehrfach auszuführen. Die Steuerung erfolgt über eine sogenannte Laufvariable, die innerhalb der Schleife arbeitet und mit einem bestimmten Startwert initialisiert wird. Über eine Bedingung hast du festgelegt, wie lange die Schleife durchlaufen werden soll. Damit hast du die Kontrolle darüber, welchen Wertebereich die Variable verarbeitet.

Über eine Blockbildung durch das geschweifte Klammerpaar kannst du mehrere Befehle zu einem Block zusammenfassen, die bei einer for-Schleife allesamt ausgeführt werden.

Die gerade genannte Laufvariable wird dazu benutzt, den Index eines Arrays zu manipulieren, um damit die einzelnen Array-Elemente anzusprechen.

Über die Manipulation der Register haben wir die digitalen Pins eines Ports angesteuert und so ein Lauflicht realisiert.

# Der Lauflicht-Workshop

Wenn du Lust hast, kannst du nun im Lauflicht-Workshop das Gelernte auf neue Fragestellungen übertragen und damit prüfen, ob du alles verstanden hast. In dem Workshop möchte ich dich dazu animieren, das Lauflicht in verschiedenen Mustern blinken zu lassen. Es gibt dabei unterschiedliche Varianten:

Immer nur in eine Richtung mit einer LED (das kennst du bereits).
Vor und zurück mit einer oder mehreren LEDs.
Vor und zurück zur selben Zeit (zwei LEDs, die sich aufeinander zu bewegen).
Zufallsauswahl der einzelnen LEDs.

Für eine zufällige Ansteuerung einer LED benötigst du eine weitere Funktion, die du bisher noch nicht kennengelernt hast. Sie nennt sich *random*, was übersetzt so viel wie ziellos oder zufällig bedeutet. Die Syntax dieser Funktion gibt es in zwei Varianten:

# 1. Variante

Wenn du einen zufälligen Wert in einem Bereich von 0 bis zu einer vor dir festgelegten Obergrenze generieren möchtest, verwende die nachfolgende Variante:

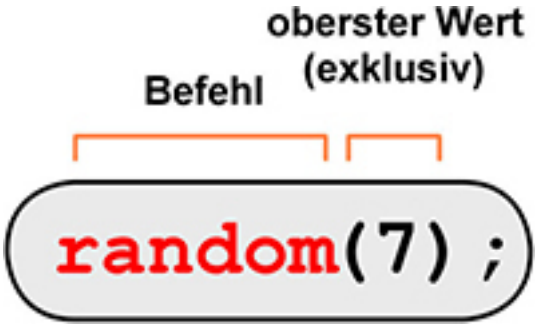



**Abb. 10:** Der Befehl random mit einem Argument

Wichtig ist jedoch, dass der oberste Wert, den du angibst, immer exklusiv ist, also nicht zu dem von dir festgelegten Zahlenbereich gehört. In diesem Beispiel generierst du also Zufallszahlen in einem Bereich von 0 bis 6.

# 2. Variante

Wenn du einen zufälligen Wert im Bereich von Untergrenze bis Obergrenze generieren möchtest, verwende die in der folgenden Abbildung dargestellte Variante:


**Abb. 11:** Der Befehl random mit zwei Argumenten

Dieser Befehl generiert Zufallszahlen im Bereich von 2 bis 5. Auch hier gilt wieder, dass der oberste Wert exklusiv ist. Dieser Umstand ist manchmal eine Fehlerquelle beim Programmieren, die man nicht so leicht findet. Die einzige Möglichkeit, diesen Fehler zu vermeiden, ist, ihn sich gut zu merken. Und nun viel Spaß bei deinem Lauflicht-Workshop. Vielleicht findest du ja auch noch weitere Möglichkeiten, was du mit dem Erlernten anstellen kannst.

# Конец ознакомительного фрагмента.