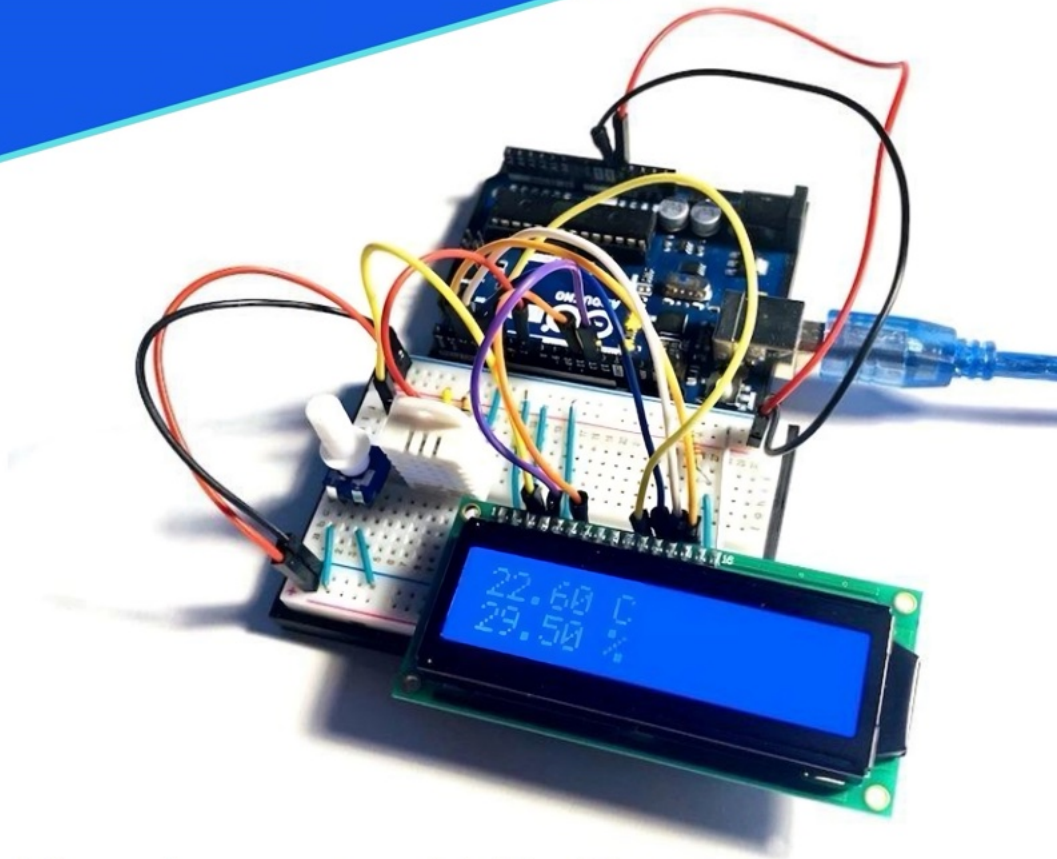
# Vom Anfänger zum Maker

MIT DEM ARDUINO & ESP8266




**Von der ersten Zeile C++ bis zum eigenen Webserver**

# VOM ANFÄNGER ZUM MAKER – MIT DEM ARDUINO & ESP8266

Frederik Kumbartzki

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



Gebhardstr. 14
76137 Karlsruhe

ISBN: 9798376245804
Independently published

# INHALT

# VORWORT

Vielen Dank, dass du dich für dieses Buch entschieden hast. Ich möchte dich gar nicht lange mit einem Vorwort aufhalten, sondern dich gleich auf die Reise durch die Welt des Arduinos schicken.

Jedoch ein paar Worte zu den Bauteilen, die du verwenden wirst. Du findest die meisten von ihnen in sogenannten Starter Kits, z.B. auf Amazon. Kits im Bereich von 50€ sollten bereits viele der hier vorgestellten Projekte abdecken.

Natürlich kannst du die Bauteile, z.B. den NodeMCU ESP8266, auch einzeln kaufen – du erhältst sie in einschlägigen Online-Shops zu günstigen Preisen.

Du findest in diesem Buch alle Sketches (Programme) für die vorgestellten Projekte. Da diese auf deinem Weg zum Maker immer länger und platzraubender werden, wurden sie teilweise auf unsere Webseite ausgelagert, wo du sie kopieren und weiterverwenden kannst:

polluxlabs.net/maker-buch

Noch ein Hinweis zu den Anleitungen in diesem Buch: Die Bibliotheken sowie die Arduino IDE werden ständig erweitert. Das bedeutet leider auch, dass die vorgestellten Programme möglicherweise nicht mehr wie abgedruckt funktionieren. Aber das ist in der Regel kein Problem, mit ein paar Anpassungen lassen sich diese Fehler meist schnell beheben.

Wir von Pollux Labs überarbeiten dieses Buch regelmäßig und prüfen die Anleitungen auf ihre Funktion. Updates findest du ebenfalls auf der oben genannten Webseite.

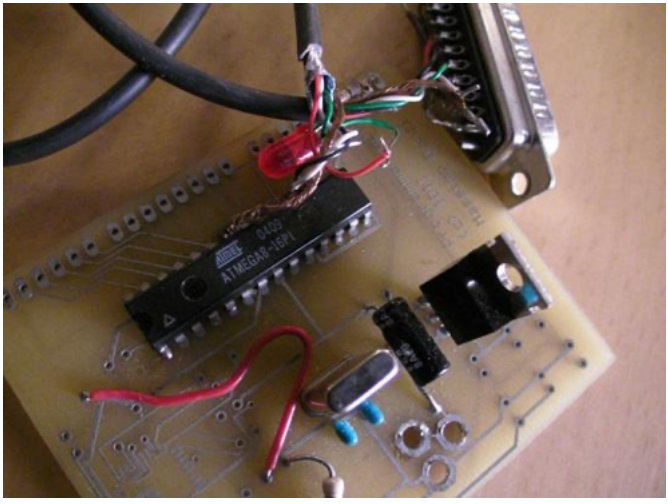
Solltest du an einer Stelle dieses Buchs oder bei einem Projekt nicht weiterkommen, schreibe uns gerne eine E-Mail an **info@polluxlabs.net**

Und nun, viel Spaß mit deinem Arduino!

# 1 DER ARDUINO UNO

Der erste Arduino Microcontroller ist auch gleichzeitig der bekannteste: der Arduino UNO. Seine Entwicklung begann im Jahr 2003 in Italien. Einige Studenten haben ihn hier zusammen mit dem Ziel entwickelt, einen günstigen und leicht programmierbaren Microcontroller für Bastler und Studierende zu entwickeln.

**Und das mit Erfolg.** Im Jahr 2005 erblickte der Arduino UNO das Licht der Welt – und es dauerte nicht lange, bis er die Herzen der Maker rund um den Globus erobert hatte.

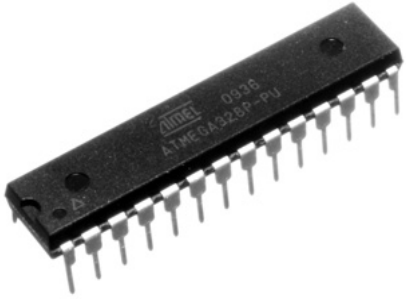



Der erste Arduino-Prototyp, Foto: Phillipptorrone, Wikipedia

Auch heute ist es für viele angehende Maker immer noch der erste Microcontroller, den sie in Händen halten. Er ist erschwinglich im Original und noch erschwinglicher in einer Kopie. Außerdem bietet er Komfort beim Aufbau von Projekten und Programmieren. **Lass uns zusammen einen Blick auf seine wichtigsten Features und Komponenten werfen.**

## DAS GEHIRN: ATMEGA328P

Kein Microcontroller ohne Microchip. Beim Arduino UNO ist dies der ATmega328P, der zentral in einem Sockel auf dem Arduino Board sitzt.




Microchip ATmega328P

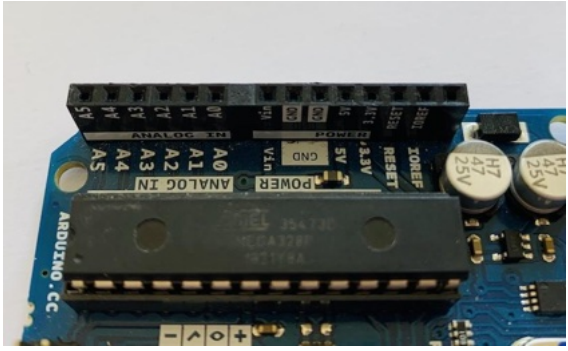
Auf diesem Chip landen deine Programme, die du hochlädst – wofür dir 32KB zur Verfügung stehen. **Gemessen an modernen Speichermedien klingt das nach nichts, du wirst aber sehen, dass du damit schon unheimlich viel anfangen kannst.**

Wenn du dir den ATmega328P genauer anschaust, siehst du, dass er 28 “Beinchen” – sogenannte Pins – hat. **Von diesen Pins kannst du 23 programmieren.** Wenn du deinen Arduino UNO umdrehst, erkennst du die Leiterbahnen, die vom Chip ausgehend unter anderem zu den Buchsen an den Seiten des Boards führen.

Wenn du also zum Beispiel eine LED in die Buchse A5 steckst, verbindest du sie dadurch mit dem entsprechenden Pin am Microchip.

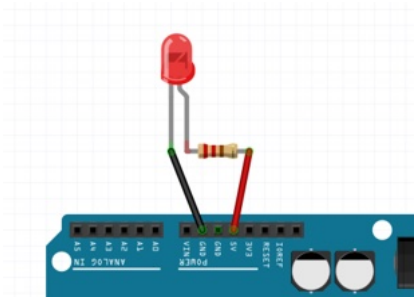
## DIE WICHTIGSTEN PINS DES ARDUINO UNO

Als nächstes werfen wir einen Blick auf die Pins, die du später benötigen wirst.




Power- und Analog-Pins

Hier haben wir zunächst Pins, die mit **Power** gekennzeichnet sind. Die beiden Pins **GND** dienen als “Minuspol” oder “Erde”. Hier kannst du beispielsweise die Kathode (Minus, kurzes Bein) einer LED anschließen, um sie zum Leuchten zu bringen.



Eine LED am Arduino UNO

In der Skizze oben siehst du, dass die Anode (Plus, langes Bein) der LED über einen Vorwiderstand an **5 Volt** angeschlossen ist. Ihr Minuspol führt direkt zu **GND**. Über LEDs und Widerstände lernst du später mehr.

Das führt uns direkt zu den beiden Pins **5V** und **3V3**. Hierüber kannst du Bauteile – wie die LED oben – mit Strom versorgen. Je nachdem, wie viel Spannung (Volt) sie benötigen, stehen dir hierfür entweder 5 Volt oder 3,3 Volt zur Verfügung. **Die meisten Hobby-Bauteile wie Servo-Motoren, Sensoren etc. benötigen eine dieser beiden Spannungen.**

Fehlt nur noch der Pin **VIN**. Hierüber kannst du den Arduino UNO mit Strom versorgen – das schauen wir uns aber in Kürze genauer an, wenn wir über die Stromversorgung sprechen.

## DIE ANALOG-PINS

Zunächst betrachten wir die **Analog-Pins A0 bis A5**. Mit diesen sechs Pins kannst du Signale “messen”, die sich kontinuierlich verändern können. Tatsächlich sind das sich verändernde Spannungen zwischen 0 und 5 Volt. Ein Beispiel ist ein Temperatursensor, der sein Ausgangssignal (also die Spannung) verändert, wenn es wärmer oder kälter wird.

Diese Spannung wird im Arduino von einem sogenannten **Analog-Digital-Konverter (ADC)** in eine entsprechende Zahl von 0 bis 1023 umgewandelt.

**Die Analog-Pins sind also immer dann nützlich, wenn du ein veränderbares Signal messen möchtest – und nicht nur wie bei digitalen Signalen entweder eine Null oder eine Eins.** Allerdings kannst du über die Analog-Pins kein veränderbares Signal ausgeben, das geht nur über einige der Digital-Pins – was uns direkt auf die andere Seite des Arduino UNO führt.

## DIE DIGITAL-PINS

Hier findest du insgesamt 13 Digital-Pins. Allerdings stehen dir für deine Projekte nur die Pins 2 bis 13 zur Verfügung – Pin 0 und 1 sind für etwas anderes reserviert und können nicht programmiert werden.

Mit den Digital-Pins kannst du die Signale Null (Aus) und Eins (An) messen und ausgeben. Das bedeutet zum Beispiel, dass

du eine LED über einen der Pins an- und ausschalten kannst. Oder messen kannst, ob eine Taste gedrückt wurde.

Einige der Digital-Pins sind mit einer Tilde (~) gekennzeichnet. Über diese Pins kannst du das erreichen, was du vielleicht eher von den Analog-Pins erwartest hättest: Du kannst hierüber ein analoges, also sich veränderndes Signal ausgeben. **Damit kannst du zum Beispiel eine LED mit unterschiedlicher Helligkeit leuchten lassen.** Der Fachbegriff hierfür heißt Pulsweitenmodulation (PWM).

## STROMVERSORGUNG

Um seine Arbeit aufnehmen zu können, benötigt dein Arduino UNO vor allem eines: Strom. Hierfür stehen dir mehrere Möglichkeiten zur Verfügung. Zunächst die USB-Buchse – diese wirst du sicherlich am häufigsten verwenden, da du hierüber auch deine Programme auf den Arduino hochlädst. **Der Arduino erhält seinen Strom dann von deinem PC oder Laptop.**

Neben der USB-Buchse befindet sich noch ein weiterer Anschluss. Hierüber kannst du einen 9V-Block anschließen. Damit machst du deinen Arduino unabhängig von einem Computer und kannst ihn transportieren.

Eine weitere Möglichkeit ist der Pin **VIN**, den wir oben schon kurz angeschaut haben. Auch hierüber kannst du Batterien anschließen. Ihren Pluspol verbindest du mit **VIN**, den Minuspol mit **GND**.

## DEN ARDUINO NEU STARTEN

Zuletzt werfen wir noch einen Blick auf einen Button, der sich auf deinem Arduino UNO neben der USB-Buchse befindet: den Reset-Knopf. Wenn du deinen Arduino neu starten möchtest, genügt ein Drücken dieses Buttons und das Programm in seinem Speicher beginnt von vorne.

Und das soll es vorerst gewesen sein. Du kennst nun die wichtigsten Komponenten deines Arduino UNO, die du auch im weiteren Verlauf verwenden wirst.

# 2 DIE ARDUINO IDE

Für Einsteiger ist die Arduino IDE (Integrated Development Environment) meist die erste Wahl – und das zu Recht. Du kannst zahlreiche Microcontroller mit ihr programmieren sowie Bibliotheken für Sensoren, Displays etc. verwalten. Außerdem besitzt sie den “Seriellen Monitor”, in dem du Daten ausgeben und auf Fehlersuche gehen kannst.

**Einsteigerfreundlich ist auch die Tatsache, dass die Arduino IDE vieles verbirgt, was für Anfänger nicht relevant ist.** So fällt der Einstieg in das eigentlich recht komplexe Thema Hardware-Programmierung leicht.

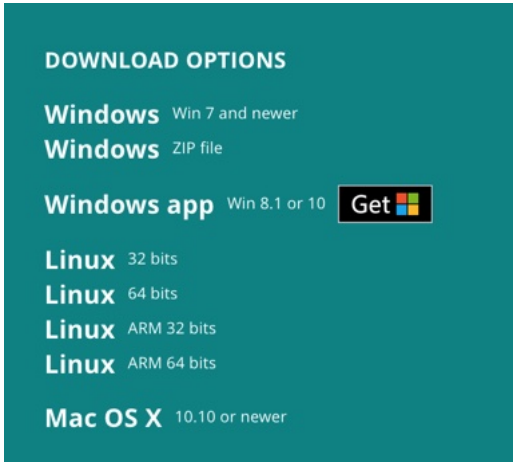
In den folgenden Abschnitten schauen wir uns an, wie du die Arduino IDE installierst. Anschließend folgt eine kleine Tour durch die wichtigsten Funktionen, um mit den ersten Projekten starten zu können.

**Hinweis:** Die Arduino IDE befindet sich aktuell (Herbst 2022) im Übergang von der Version 1.8.x zur Version 2.0 – die neue Version hat den Beta-Status zwar verlassen und ist veröffentlicht, viele Maker verwenden allerdings noch die Version 1.8.x. Deshalb findest du hier zunächst die Einführung zur “alten” IDE und anschließend mehr darüber, was sich nach dem Update geändert bzw. was in der neuen Version hinzugekommen ist.

# INSTALLATION

Die Arduino IDE gibt es für Windows, macOS und Linux und ist mit ein paar Klicks auf deinem Rechner installiert.

Auf der offiziellen Download-Seite unter www.arduino.cc/en/software findest du die jeweils aktuelle Version. Ein Klick auf dein Betriebssystem startet den Download. Wenn du die Version 1.8.x installieren möchtest, scrolle etwas nach unten bis zum Abschnitt **Legacy IDE (1.8.X).**




Sobald die Datei auf deinem Rechner ist, starte sie (bzw. öffne die ZIP-Datei) und folge den weiteren Anweisungen. Anschließend kannst du die Arduino IDE öffnen.

## ALTERNATIVEN

Ebenso offiziell ist auch der **Arduino Web Editor**. Hiermit kannst du deinen Arduino direkt im Browser programmieren. Ganz ohne Download geht das allerdings auch nicht, denn du benötigst das Plugin **Arduino Create Agent** für die Kommunikation zwischen Rechner und Arduino. Außerdem musst du ein kostenloses Benutzerkonto anlegen, um den Editor verwenden zu können.

Erfahrene Nutzer können auch die **Beta-Version** und **Nightly Builds** verwenden. Hierbei handelt es sich um Versionen, die bereits künftige Features enthalten. Allerdings können hier noch Bugs vorhanden sein, weswegen Einsteiger sie eher nicht verwenden sollten.

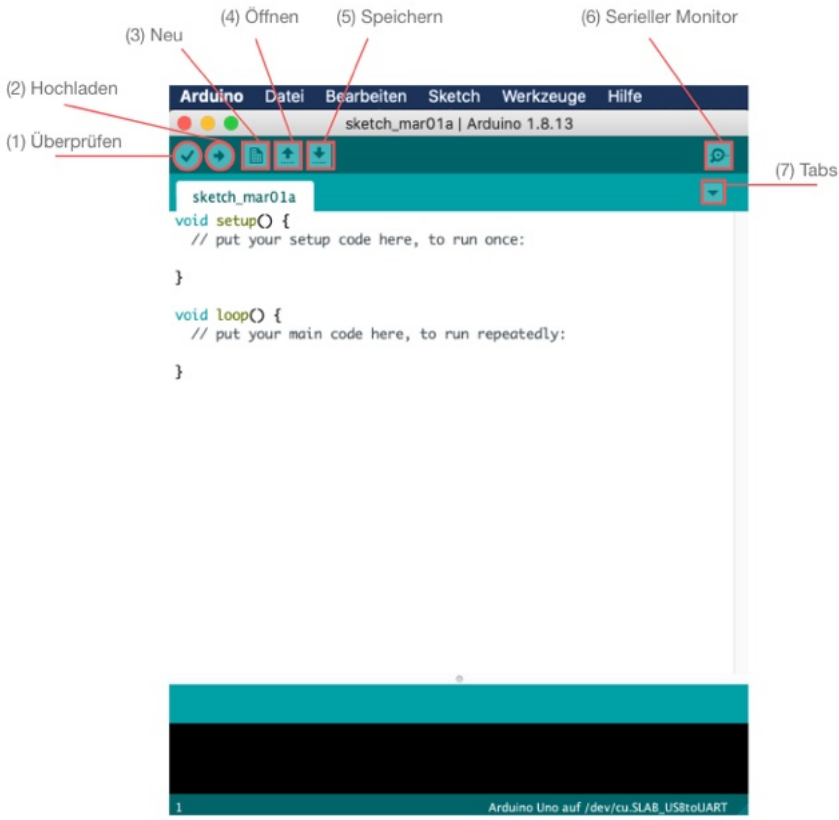
## LERNE DIE IDE KENNEN

Wenn du die Arduino IDE zum allerersten Mal startest, passiert nichts außergewöhnliches. Es gibt keinen Willkommens-Screen, kein Tutorial und keine Tour durch das Programm. Deshalb übernehmen wir das jetzt.

In diesem und den folgenden Abschnitten arbeiten wir mit der Mac-Version der Arduino IDE. **Das Programm ist jedoch auf anderen Betriebssystemen weitestgehend identisch.**

## DAS CODE-FENSTER

Nach dem Programmstart öffnet sich ein Fenster für den Code – **in der Arduino-Welt auch Sketch genannt.** Diesen schauen wir uns später genauer an, jetzt werfen wir zunächst einen Blick auf die Buttons, die sich im oberen Bereich des Fensters befinden. Hier findest du wichtige Funktionen, die du beim Programmieren deines Arduinos immer wieder benötigst.



Von links nach rechts sind das:

1. **Überprüfen:** Mit einem Klick auf diesen Button überprüfst du deinen Code auf Fehler. Findet die Arduino IDE einen Fehler, wird die entsprechende Zeile rot markiert und im unteren Bereich des Fensters die Fehlermeldung und oft auch ein Hinweis eingeblendet.
2. **Hochladen:** Mit diesem Button lädst du deinen Sketch auf den Arduino hoch. Vor dem Upload findet auch immer zunächst eine Überprüfung des Codes statt.
3. **Neu:** Dieser Button öffnet ein neues Fenster.
4. **Öffnen:** Hiermit kannst du einen gespeicherten Sketch öffnen.
5. **Speichern:** Richtig, mit diesem Button kannst du deinen Sketch auf einem Datenträger speichern.
6. **Serieller Monitor:** Dieser Button öffnet den Seriellen Monitor, sofern du deinen Arduino am Rechner angeschlossen hast. Mit dem Seriellen Monitor beschäftigen wir uns später genauer.
7. **Tabs:** Du benötigst nicht für jeden Sketch auch ein neues Fenster, sondern kannst diese auch nebeneinander in einem Fenster öffnen – so wie du es von deinem Browser kennst. Mit diesem Button kannst du deine Tabs organisieren.

## DIE MENÜLEISTE

Neben den Funktionen innerhalb des Fensters für deinen Code gibt es oben auch eine Menüleiste. Hier findest du ebenfalls die oben aufgeführten Funktionen – aber auch noch einiges mehr.


Zunächst werfen wir allerdings einen Blick in das Menü **Werkzeuge**. Hier findest du neben **Board:** den Microcontroller, der gerade in der Arduino IDE verwendet wird.

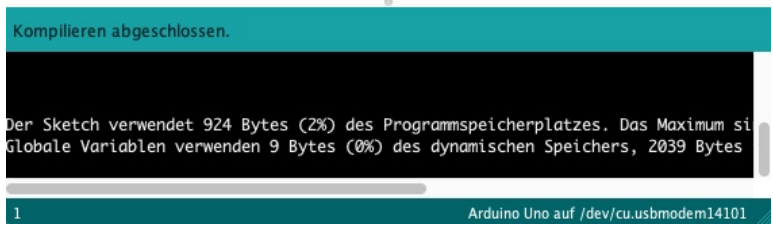

Wenn bei dir hier noch nicht **Arduino Uno** steht, klicke auf den Menüpunkt und wähle zunächst **Arduino AVR Boards**. Jetzt öffnet sich ein weiteres Untermenü, in dem du dann den **Arduino Uno** auswählen kannst.

**Hast du deinen Arduino bereits mit deinem Rechner per USB verbunden? Wenn nicht, hole das jetzt nach.**

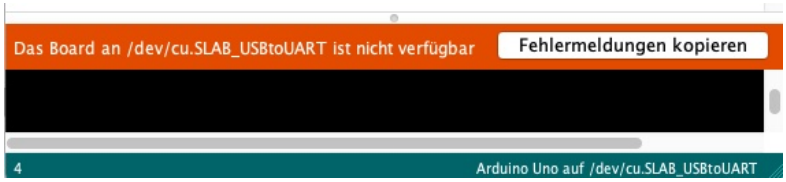
Ebenfalls im Menü **Werkzeuge** findest du den Eintrag **Port**. Wähle hier den USB-Port, an dem dein Arduino angeschlossen ist. Das ist zunächst alles, was du machen musst, um deinen Arduino mit der IDE programmieren zu können.

## INFORMATIONEN UND FEHLERMELDUNGEN

Zuletzt schauen wir noch auf den unteren Bereich des Fensters. Hier findest du einen Bereich, in dem dir Informationen, Hinweise und Fehlermeldungen angezeigt werden. Wenn du zum Beispiel einen Sketch erfolgreich auf deinen Arduino geladen hast, siehst du etwas in dieser Art:




Eine Fehlermeldung sieht hingegen beispielsweise folgendermaßen aus.



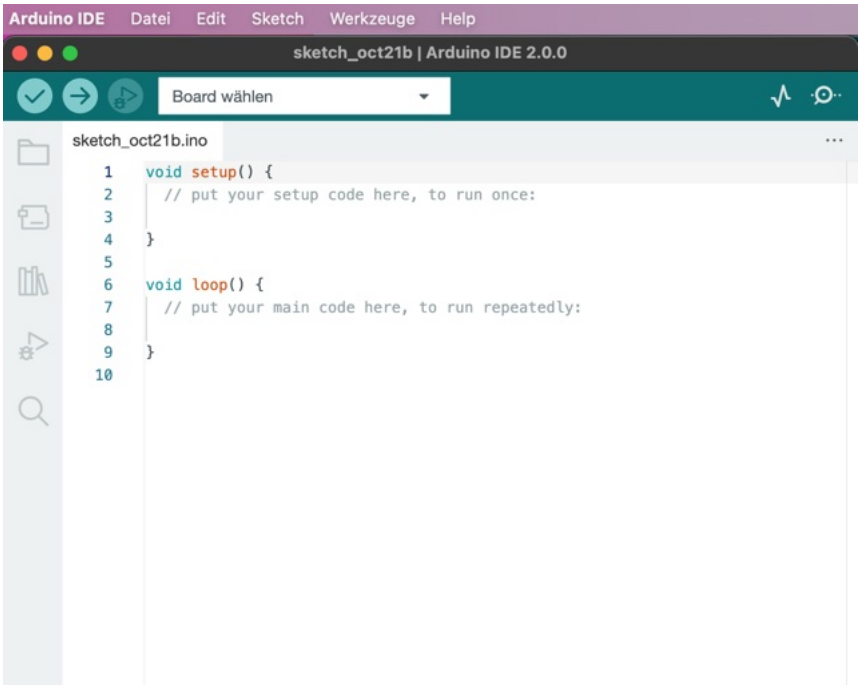
Hier ist das Problem ein nicht mit dem Rechner verbundener Microcontroller. Der Button **Fehlermeldung kopieren** ist oft praktisch. Damit kopierst du die Meldung in den Zwischenspeicher und kannst sie gleich darauf in einer Suchmaschine verwenden. Du wirst sehen, **Suchmaschinen sind oft der beste Freund beim Programmieren** – nicht nur als Anfänger.

Im nächsten Abschnitt wenden wir uns dem Code zu – und du wirst deinen ersten Sketch schreiben.

## DIE ARDUINO IDE 2.0

Die neue IDE hat optisch das Rad nicht neu erfunden, es befinden sich lediglich einige Optionen, Tools und Einstellungen an anderen Orten. Ebenso macht die neue Version einen etwas frischeren Eindruck als die Oberfläche der Version 1.8.x

Wenn du die Arduino IDE 2.0 öffnest, erwartet dich dieses Fenster:




Auch hier findest du direkt die Buttons, um deinen Sketch zu überprüfen und auf den Arduino zu übertragen. Ebenso findest du rechts den Seriellen Monitor und daneben den Seriellen Plotter – ein Tool, das dir Daten, die dein Arduino sendet, grafisch anzeigen kann.

Allerdings kannst du oben über ein Dropdown-Menü direkt das angeschlossene Board auswählen und musst dafür nicht mehr das Menü öffnen. Linker Hand befindet sich eine Menüleiste, mit der du auf deine Dateien sowie die installierten Microcontroller und Bibliotheken. Über die Lupe kannst du im

geöffneten Sketch suchen – eine Funktion, die du wie in vielen anderen Programmen auch mit Strg+F aufrufen kannst.

## WAS IST NEU?

Zwei Dinge haben in die Version 2.0 Einzug gehalten, die es vorher in der Arduino IDE noch nicht gab: **automatische Code-Vervollständigung und ein Debugger.**

Ersteres musst du evtl. zunächst aktivieren. Öffne dazu die Einstellungen und setze neben **Schnelle Editor Vorschläge** einen Haken. Damit schlägt dir die IDE schon beim Tippen z.B. Variablen vor, die du bereits deklariert hast, wie auf diesem Screenshot zu sehen:


Das kann äußerst hilfreich sein und verhindern, dass du die gleiche Variable versehentlich unterschiedlich schreibst.

Mit dem Debugger (du findest ihn in der Menüspalte links – der Play-Button mit dem Käfer bzw. englisch “Bug”) kannst du deinen Code Zeile für Zeile ausführen und dich damit auf Fehlersuche begeben. Dieses Tool steht dir leider nicht für den Arduino UNO zur Verfügung, weswegen wir es in diesem Buch

nicht ausführlicher behandeln. Mehr Informationen zum Debugger und mit welchen Boards er funktioniert, findest du auf der offiziellen Arduino-Seite hierzu:

**docs.arduino.cc/software/ide-v2/tutorials/ide-v2-debugger**

# 3 DEIN ERSTER SKETCH

Jetzt wird es Zeit für die Praxis! In den folgenden Abschnitten lernst du das Grundgerüst eines Arduino-Sketchs kennen und schreibst deine eigenen Programme. Los geht's!

## SETUP UND LOOP

Sobald du einen neuen Sketch in der Arduino IDE anlegst und sich das Fenster öffnet, siehst du immer folgendes Grundgerüst:


Aber was verbirgt sich dahinter? Wie du vielleicht schon weißt, verwendest du in der Arduino IDE die Programmiersprache C++.

In dieser Sprache lassen sich – so wie übrigens in allen anderen auch – sogenannte **Funktionen** definieren.

Diese gehören eigentlich zum fortgeschrittenen Programmieren. Du kommst aber aufgrund der Struktur eines Arduino-Sketchs nicht drumherum, Funktionen zumindest in ihren Grundzügen zu verstehen.

**Mit Funktionen kannst du deinen Code strukturieren, indem du ihnen voneinander getrennte Aufgaben zuweist.** Hierbei **definierst** du zunächst die Funktion nach einem festen Muster. In C++ sieht dieses folgendermaßen aus:

```
Typ Name() {
  Code;
}
```

Der Typ gibt an, um welche “Sorte” von Funktion es sich handelt. In unserem Fall ist dies der Typ **void**. Das bedeutet, dass diese Funktion nichts weiter tut, als den Code zwischen den geschweiften Klammern **{ }** auszuführen.

Der Name ist – richtig, der Name der Funktion. In unserem Fall also **Setup** und **Loop**. Innerhalb der beiden geschweiften Klammern kommt dann alles, was du diese Funktionen ausführen lassen möchtest.

**Hier gleich ein Hinweis:** Achte immer darauf, dass jede öffnende Klammer **{** auch ein schließendes Gegenstück **}** hat. Ansonsten wird dein Sketch nicht ausgeführt. **Aber keine Angst, solltest du eine Klammer vergessen, weist dich die Arduino IDE vor dem Hochladen des Sketchs darauf hin.**

Aber wozu sind diese beiden Funktionen gut?

## DIE SETUP-FUNKTION

In dieser Funktion bestimmst du, was dein Arduino nach dem Start **einmal** ausführen soll. Hier kannst du zum Beispiel festlegen,

- ob ein Pin Signale sendet oder empfängt.
- ob eine LED zu Beginn des Programms aus- oder eingeschaltet sein soll.
- dass der Serielle Monitor in einer bestimmten Geschwindigkeit laufen soll.

All das muss dein Arduino nur einmal zu Beginn des Sketchs wissen – es ist also, wie der Name der Funktion schon sagt, das **Setup**.

Später wirst du die Setup-Funktion mit Code füllen. Zunächst jedoch ein Blick auf den **Loop**.

### DIE LOOP-FUNKTION

Im Gegensatz zum Setup wird der gesamte Code innerhalb des Loops ständig wiederholt. Dein Arduino führt ihn Zeile für Zeile aus, bis er am Ende angekommen ist – und beginnt dann wieder von vorne.

Das machst du dir gleich zunutze, indem du mit der Loop-Funktion die interne LED deines Arduino blinken lässt.

## LASS DIE LED DES ARDUINOS BLINKEN

Zeit für dein erstes Programm! Auf deinem Arduino UNO befinden sich mehrere kleine LEDs – eine davon wirst du nun immer wieder ein- und ausschalten bzw. blinken lassen.

Öffne zunächst die Arduino IDE und erstelle einen leeren Sketch. Wähle hierfür im Menü **Datei** den Punkt **Neu**. Nun siehst du die beiden leeren Funktionen **void setup()** und **void loop()**.

### DIE SETUP-FUNKTION

Zunächst widmen wir uns der Setup-Funktion. Trage hier zwischen die beiden geschweiften Klammern **{ }** folgende Zeile Code ein:

```
pinMode(LED_BUILTIN, OUTPUT);
```

Die Setup-Funktion sieht dann so aus:

```
void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
}
```

Schauen wir uns diese Zeile genauer an. Zunächst gibt es hier eine weitere Funktion: **pinMode()**. Auch sie führt eine Aktion aus – sie legt für einen bestimmten Pin deines Arduinos eine Richtung bzw. ihren **Modus** fest: **Entweder ein Signal senden (OUTPUT) oder ein Signal empfangen (INPUT).**

Im Gegensatz zur Setup- und Loop-Funktion “erwartet” diese Funktion jedoch etwas zwischen den runden Klammern **( )**, nämlich sogenannte **Parameter.**

Der erste dieser Parameter bestimmt den Pin, dessen Richtung festgelegt werden soll. In unserem Fall ist das kein Analog- oder Digital-Pin an den Seiten des Arduinos, sondern die interne LED. Diese wird im Sketch mit **LED_BUILTIN** bezeichnet.

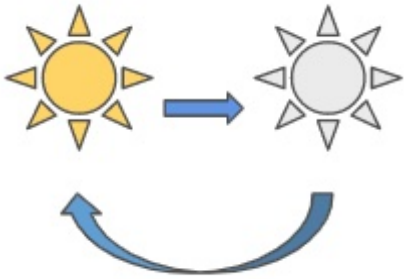
Der zweite Parameter bestimmt dann die Richtung – entweder OUTPUT oder INPUT. Wenn du eine LED steuern möchtest, muss der Arduino ihr entsprechende Signale senden. Vom Arduino aus gesehen ist das also ein OUTPUT. Wenn jedoch zum Beispiel ein Sensor Messdaten an den Arduino sendet, ist das wiederum ein eintreffendes Signal – also ein INPUT.

*Hinweis: Achte darauf, dass du jede Zeile, die nicht mit einer geschweiften Klammer endet, mit einem Semikolon **;** abschließt. Ansonsten kann dein Sketch nicht hochgeladen werden. Von dieser Regel gibt es Ausnahmen, die jedoch jetzt noch nicht wichtig sind.*

Die Richtung des Pins musst du nur einmal zu Beginn deines Programms festlegen. Deshalb befindet sich die Funktion **pinMode()** in der Setup-Funktion.

## DIE LOOP-FUNKTION

Kommen wir zum Loop. Du weißt, dass sich der Code innerhalb des Loops ständig wiederholt. Das können wir uns für das Blinken der LED zunutze machen, denn **eigentlich ist Blinken nichts anderes als 1) das Licht anschalten und 2) das Licht ausschalten – und wieder von vorne:**



## DIE LED EINSCHALTEN

Trage zunächst in deiner Loop-Funktion zwischen den geschweiften Klammern **{ }** folgende Zeile ein:

```
digitalWrite(LED_BUILTIN, HIGH);
```

Hierbei handelt es sich wieder um eine Funktion, diesmal also **digitalWrite()**. Diese Funktion kann über einen Digital-Pin entweder das Signal **HIGH** oder **LOW** senden. HIGH bedeutet so viel wie “an” bzw. 1 – LOW hingegen steht für “aus” bzw. 0.

**Am Beispiel einer LED ist das recht leicht zu verstehen: HIGH schaltet die LED ein, LOW schaltet sie aus.**

Auch die Funktion **digitalWrite()** erwartet zwei Parameter: Den Pin, den sie ansteuern soll, und das Signal. In unserem Fall ist das also der Pin **LED_BUILTIN** und zunächst das Signal "Einschalten" – also **HIGH**.

Du hast nun also eine Zeile, um die LED einzuschalten. Aber das ist natürlich noch kein ordentliches Blinken.

## WARTE KURZ

So ein Arduino ist erstaunlich schnell. Würdest du die LED einschalten und sie sofort danach wieder ausschalten, würdest du vom Blinken nichts mitbekommen. Der Wechsel von An und

Aus wäre so schnell, dass es so aussähe, als würde die LED durchgängig leuchten.

Wir müssen also kurz warten.

Im Sketch funktioniert das mit der Funktion **delay()**. Diese Funktion verzögert sozusagen den weiteren Ablauf deines Programms. **Hierfür erwartet sie einen einzigen Parameter, nämlich die Dauer der Verzögerung – in Millisekunden**.

Trage als nächstes folgende Zeile in der Loop-Funktion ein:

```
delay(1000);
```

Damit verzögerst du die weitere Ausführung um 1.000 Millisekunden, was genau einer Sekunde entspricht. **Die Verzögerung sorgt dafür, dass die LED, die du in der Zeile davor angeschaltet hast, eine Sekunde lang brennt. Erst dann wird die nächste Zeile ausgeführt**.

## DIE LED AUSSCHALTEN UND WIEDER WARTEN

In den nächsten zwei Zeilen Code drehst du den Spieß um – zunächst schaltest du die LED aus und sorgst dafür, dass das für eine Sekunde so bleibt:

```
digitalWrite(LED_BUILTIN, LOW);
delay(1000);
```

Die Funktion **digitalWrite()** kennst du ja bereits. Diesmal sendet sie an die interne LED (LED_BUILTIN) jedoch das Signal LOW. Sie schaltet sie also aus – und zwar ebenfalls für eine Sekunde.

Die vollständige Loop-Funktion sieht dann folgendermaßen aus:

```
void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

## ALLES WIEDER VON VORNE

Jetzt hast du alles, was du für deine blinkende LED brauchst: Du schaltest das Licht kurz ein und schaltest es wieder kurz aus. **Den Rest, nämlich die Wiederholung, erledigt die Loop-Funktion von ganz alleine.**

Sobald dein Programm das zweite Mal **delay(1000);** abgearbeitet hat, ist es am Ende des Loops angekommen und

springt wieder zu dessen Anfang – also in die Zeile **digitalWrite(LED_BUILTIN, HIGH);**

Probiere es gleich aus! Kopiere den folgenden Sketch in deine Arduino IDE und lade ihn auf deinen Arduino. Blinkt die LED? Spiele nun etwas mit den Parametern in der Delay-Funktion herum und verändere sie. Vielleicht kannst du zur Übung ja sogar etwas morsen, indem du die LED unterschiedlich lang aufleuchten lässt.

```
void setup() {
  // put your setup code here, to run once:
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {
  // put your main code here, to run repeatedly:
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

## FINDET DEINE IDE DEN ARDUINO UNO NICHT?

Dann fehlt dir noch der passende Treiber für den Chip auf deinem Arduino UNO. Aber das ist kein Problem, du kannst den Treiber ganz einfach nachinstallieren. Auf dieser Webseite findest du die aktuelle Version für dein Betriebssystem: **sparks.gogo.co.nz/ch340.html**

# 4 DER SERIELLE MONITOR

Kommen wir nun zum Seriellen Monitor – einem Feature der Arduino IDE, das du so gut wie bei jedem deiner Projekte verwenden wirst.

## ÜBERBLICK

Erinnerst du dich an die Lupe oben rechts in deinem Sketch-Fenster? Mit einem Klick hierauf öffnest du den Seriellen Monitor – **sofern dein Arduino UNO am Rechner angeschlossen ist und du den richtigen USB-Port ausgewählt hast.**


Der Serielle Monitor ist so etwas wie die Kommunikationsschnittstelle deines Arduino. Er kann hier Messwerte und andere Daten ausgeben, sodass du sie überprüfen kannst. Aber diese Schnittstelle funktioniert auch in die andere Richtung. Auch du kannst deinem Arduino über den Seriellen Monitor Befehle senden.

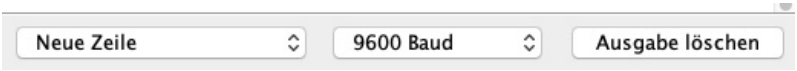
Werfen wir zunächst einen Blick auf die wichtigsten Teile des Monitors.


Ganz oben in der Titelleiste findest du den USB-Port, über den der Serielle Monitor mit dem Arduino kommuniziert.

Darunter befindet sich ein Eingabefeld samt Button zum **Senden**. Hierüber kannst du deinem Arduino Daten und Befehle senden. Zunächst bleiben wir jedoch beim Lesen.

Der größte Teil des Fensters ist für die Ausgabe reserviert – hier wirst du bald einen Text anzeigen lassen.

Unten links findest du zwei Checkboxen: Einmal **Autoscroll** – damit bewegt sich der Serielle Monitor mit den Ausgabedaten mit. Und **Zeitstempel anzeigen** – damit werden alle Daten, die du mit dem Arduino an den Monitor sendest, mit der Uhrzeit versehen, an der sie dort eintrafen.




Unten rechts findest du ein Dropdown, in dem standardmäßig **Neue Zeile** steht. Dieses bezieht sich wieder auf das Senden von Daten an deinen Arduino. **Daneben befindet sich jedoch noch eine wichtige Einstellung, die oft eine Fehlerquelle ist:**

## DIE BAUDRATE

Die Baudrate (oder auch Symbolrate) bezeichnet die Menge an Zeichen, die pro Sekunde übertragen werden. **Oft wird in Projekten mit einem Arduino UNO eine Rate von 9.600 verwendet.**

Wenn du das Dropdown-Menü öffnest, siehst du, dass es hier noch viele weitere Baudraten zur Auswahl gibt. Wenn du später einmal Projekte mit dem ESP8266 baust, werden diese für dich interessant, da dieser Microcontroller deutlich schneller als ein Arduino UNO ist.

**Eine Sache ist wichtig:** In diesem Dropdown-Menü muss immer die gleiche Baudrate wie in deinem Sketch eingestellt sein (Wie du sie dort einstellst, erfährst du demnächst). Wenn sich die Baudraten im Sketch und im Monitor unterscheiden, kann **nichts kaputtgehen** – aber du siehst oft einen ziemlichen Zeichensalat.

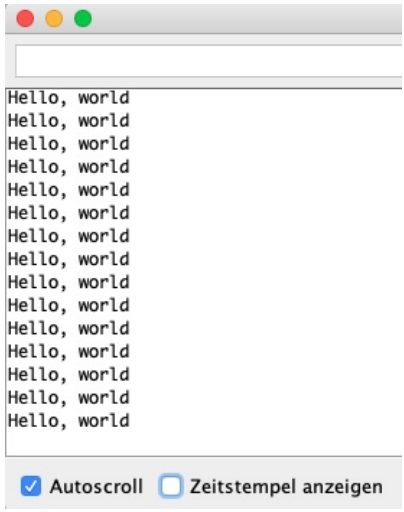
Bleibt noch ein einziger Button übrig: **Ausgabe löschen**. Wie der Name schon sagt, löschst du damit den Inhalt des Ausgabefensters

*Noch ein Hinweis: Dein Arduino kann nicht nur über USB kommunizieren, sondern auch über die Pins 0->RX und 1<-TX. Deshalb solltest du an diese Pins nichts anschließen, außer du möchtest sie später einmal für die serielle Kommunikation verwenden.*

Jetzt steigst du in die Praxis ein, aktivierst den Seriellen Monitor und gibst deine ersten Zeichen dort aus.

# 5 HELLO, WORLD

Zeit für den Klassiker der Programmierung – **hello, world**. Du schreibst nun ein kleines Programm, das in deinem Seriellen Monitor immer wieder die Zeile “hello, world” ausgibt.




## DIE VERBINDUNG ZUM SERIELLEN MONITOR

Wie du den Seriellen Monitor im Sketch-Fenster öffnest, weißt du bereits (mit der Lupe oben rechts). Bevor dort jedoch Daten erscheinen können, musst du ihn auch in deinem Sketch starten – eine einmalige Angelegenheit, weshalb der Code hierfür in die Setup-Funktion kommt:

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

Die Funktion **Serial** erweiterst du hier um den Befehl **begin()**, um die Verbindung einzurichten. **Achte unbedingt auf den Punkt zwischen Serial und begin.**

Als Parameter zwischen den runden Klammern gibst du der Funktion die Baudrate mit – in unserem Fall also 9600. Wie du bereits weißt, muss diese mit der Einstellung im Seriellen Monitor übereinstimmen.

## ZEICHEN IM SERIELLEN MONITOR AUSGEBEN

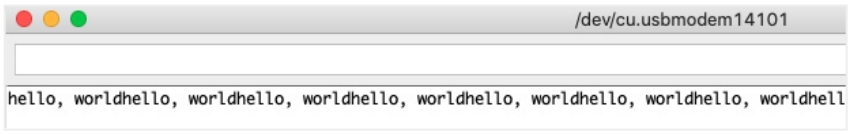
Nun kommt der interessante Teil. Trage im Loop folgenden Code ein:

```
Serial.print("hello, world");
Serial.println();
delay(1000);
```

In der ersten Zeile siehst du den Befehl **.print()**. Zwischen den Klammern steht die Zeichenkette (String) **hello, world**. Wie du siehst, steht dieser kleine Gruß zwischen Anführungsstrichen – **das ist besonders wichtig und eine häufige Fehlerquelle.**

Wenn du die Anführungsstriche testweise entfernst, wirst du eine Fehlermeldung erhalten. Das liegt daran, dass dann **hello** als Variable interpretiert wird – und nicht als String. Dem Thema Variablen wenden wir uns in einem späteren Abschnitt zu.

In der Zeile darunter steht **Serial.println();** – hiermit erzeugst du im Seriellen Monitor eine neue Zeile. Würdest du das nicht tun, würde dein Arduino die Zeichen einfach nur hintereinander schreiben:




Du kannst diese beiden Befehle – also den Gruß und die neue Zeile – jedoch auch kombinieren und dir so eine Zeile Code sparen:

```
Serial.println("hello, world");
```

Zuletzt findest du im Loop noch einen Delay von einer Sekunde. Hast du den Sketch schon ausprobiert? Erscheint der Gruß in deinem Seriellen Monitor?

Als nächstes behandeln wir eines der wichtigsten Themen: Variablen.

# 6 VARIABLEN

Um Variablen kommt kein Programmierer herum. In diesem Abschnitt lernst du, welche Typen es gibt und was du damit anstellen kannst.

## WOZU SIND SIE GUT?

Variablen sind – wie ihr Name schon sagt – variabel, also veränderlich. Genauer gesagt ist es ihr Inhalt, der veränderlich ist.

Du kannst zum Beispiel die Zahl 10 in deinem Sketch hinterlegen, um sie im Seriellen Monitor auszugeben – so wie du es mit dem Text **hello, world** gemacht hat.

```
Serial.print(10);
```

Du kannst aber auch eine Variable verwenden und die Zahl dort speichern:

```
int zahl = 10;
```

Wenn du den Inhalt dieser Variablen dann in einem Seriellen Monitor ausgeben möchtest, ersetzt du diese “feste” Zahl durch die Variable:

```
Serial.print(zahl);
```

Im Seriellen Monitor siehst du keinen Unterschied – das Ergebnis ist dasselbe. **Beim Programmieren macht das jedoch einen großen Unterschied!** Stell dir vor, du verwendest die Zahl an mehreren Stellen in deinem Sketch. Eines Tages möchtest du sie jedoch durch eine 11 ersetzen. Das würde bedeuten, dass du jede einzelne Stelle im Sketch, an der sie eingetragen ist, ändern müsstest. Wenn du jedoch eine Variable verwendest, musst du das nur einmal tun.

```
int zahl = 11;
```

Überall wo du dann die Variable eingesetzt hast, verwendet sie deine neue Zahl. In C++ gibt es eine Vielzahl von Variablentypen, zum Beispiel für Texte (Strings), ganze Zahlen und Kommazahlen. Schauen wir uns diese Typen genauer an.

## WELCHE TYPEN GIBT ES?

Bleiben wir zunächst beim Text **hello, world**. Dieser Text soll nun in einer Variablen für Texte gespeichert werden.

Am einfachsten funktioniert das in der Arduino IDE mit dem Typ **String**. Und damit machen wir einen kurzen Abstecher zur Deklaration von Variablen.

## STRINGS DEKLARIEREN

Die Deklaration von Variablen funktioniert recht einfach und immer nach dem gleichen Prinzip:

```
Typ Name = Wert;
```

In unserem Fall wäre das also

```
String text = "hello, world";
```

Lass uns kurz über den Namen von Variablen sprechen. Im Prinzip sind hier deiner Phantasie keine Grenzen gesetzt. Es gibt jedoch ein paar Namen, die du nicht verwenden kannst, da sie schon von anderen Funktionen in C++ besetzt sind.

Dazu gehören zum Beispiel: **if, else, continue, class** und **true.**

Die Liste der verbotenen Wörter ist lang – **du musst sie dir jedoch zum Glück nicht merken.** Immer wenn du einen Variablennamen bei der Deklaration verwendest, der schon anderweitig besetzt ist, färbt sich der Name ein (erlaubte Namen bleiben schwarz) und spätestens beim Hochladen des Sketchs erhältst du eine Fehlermeldung.

**Außerdem dürfen Variablennamen nicht mit einer Zahl oder einem Sonderzeichen beginnen.**

Dafür kannst du die Namen jedoch groß oder klein schreiben. Es ist jedoch üblich, einen Variablennamen klein zu beginnen. Wenn der Name aus zwei oder mehr Wörtern besteht, kommt der sogenannte **Camel Case** zum Einsatz. Hier beginnst du das erste Wort klein und schließt alle weiteren Wörter mit einem Großbuchstaben an:

```
meineKamelVariable
```

Mit viel Phantasie erkennst du hier die Höcker eines Kamels, naja… **Noch ein letzter Tipp:** Versuche, Variablen Namen zu geben, die ihren Zweck gut beschreiben. Das hilft dir und anderen Lesern deines Codes zu verstehen, was darin gespeichert wird.

## WEITERE TYPEN VON VARIABLEN

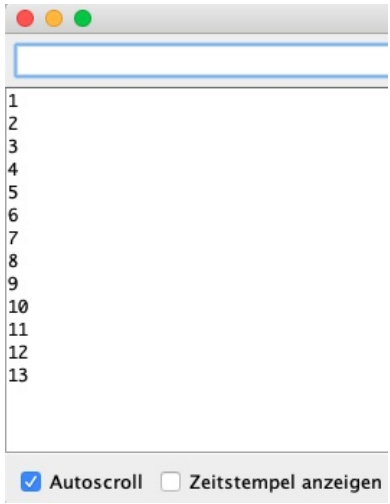
Neben Texten gibt es natürlich auch Zahlen. Hierfür gibt es gleich mehrere Typen, je nachdem um welche Zahl es sich handelt:

| Typ | Geeignet für | Zahlenbereich |
|---|---|---|
| int | Ganze Zahlen | -32.768 bis 32.767 |
| long | Ganze Zahlen | -2.000.000.000 bis 2.000.000.000 |
| float | Fließkommazahlen | -3.4028235E+38 bis 3.4028235E+38 |

Im weiteren Verlauf dieses Buchs wirst du noch Variablen des Typs **int** – für ganze Zahlen – verwenden. Und noch ein ganz besonderer Typ: **bool**. Variablen des Typs **bool** können nur zwei Werte, bzw. Zustände annehmen: True (wahr) oder False (falsch).

## VARIABLEN IN AKTION

Als nächstes baust du dir einen Zähler im Seriellen Monitor, der beginnend bei Eins jede Sekunde eins hochzählt.




Schauen wir uns zunächst an, wie du eine Variable für ganze Zahlen deklarierst. Hierfür benötigst du den Variablentyp **int** – was übrigens für **integer**, also Ganzzahl steht. Angenommen die Variable soll **zahl** heißen und zunächst den Wert **1** besitzen:

```
int zahl = 1;
```

Das war einfach. Du fragst dich jedoch vielleicht, an welcher Stelle im Sketch diese Zeile stehen soll – und hier kommt ein weiterer wichtiger Aspekt von Variablen ins Spiel: **Scope**.

## GLOBALE UND LOKALE VARIABLEN

Prinzipiell gibt es zwei Zustände, die Variablen hinsichtlich ihres Scopes einnehmen können: **global** und **lokal**. Global bedeutet,

dass die Variable für alle Funktionen im Sketch zur Verfügung steht und überall verändert werden kann. Lokale Variablen können hingegen nur in der Funktion verwendet werden, in der sie deklariert wurden – und auch nur dort gelesen und verändert werden.

Zwei dieser Funktionen kennst du ja bereits aus jedem Sketch, den du neu anlegst: **void setup()** und **void loop()**. Würdest du die Variable nun in der Setup-Funktion anlegen, wäre sie auch nur dort verfügbar, da sie **lokal** wäre:

```
void setup() {
  int zahl = 1;
  Serial.begin(9600);
}

void loop() {
  Serial.println(zahl);
}
```

Das bedeutet konkret, dass der Loop die Variable **zahl** nicht "sehen" – und sie deshalb auch nicht im Seriellen Monitor ausgeben könnte. Das heißt also, dass wir eine **globale** Variable brauchen. **Diese deklarierst du ganz zu Beginn des Sketchs, also noch vor dem Setup:**

```
int zahl = 1;

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

void loop() {
  Serial.println(zahl);
}
```

So stellst du sicher, dass der Loop auf sie zugreifen kann. Kehren wir zurück zum Zähler. Hierfür möchten wir zunächst die Variable **zahl** im Seriellen Monitor ausgeben. Anschließend soll die Zahl in der Variablen um eins erhöht werden. Das funktioniert ganz einfach:

```
zahl = zahl + 1;
```

Irritiert dich diese Schreibweise? Dann versuche die Zeile Code einmal so zu lesen: *“Nimm die Variable **zahl**, weise ihr den bisherigen Wert in der Variable zu und addiere hierzu die Zahl 1.”*

**Es gibt noch eine weitere, kürzere Schreibweise** – von der die Sprache C++ übrigens auch ihren Namen hat:

```
zahl++;
```

Diese Schreibweise ist auf den ersten Blick zwar weniger intuitiv, spart dir aber etwas Zeit und spart Platz.

Übrigens, wenn du statt der Addition andere Rechenarten verwenden möchtest, funktioniert das so:

```
zahl = zahl -1; //Subtraktion
zahl--; //Auch Subtraktion
zahl = zahl * 2; //Multiplikation
zahl = zahl / 2; //Division
```

Und das ist alles, was du für den Zähler benötigst. Der vollständige Sketch sieht dann folgendermaßen aus:

```
int zahl = 1;

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

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

**Auch hier machst du dir wieder den Loop zunutze:** Er gibt die Variable **zahl** aus, addiert eins hinzu, wartet eine Sekunde – und beginnt diese Prozedur wieder von vorne.

Im nächsten Kapitel wenden wir uns der Hardware zu – beginnend mit einer LED.

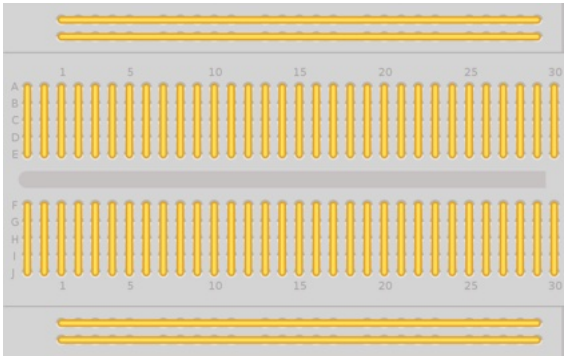
# 7 EINE LED STEUERN

Jetzt wird es Zeit für etwas mehr Hardware! In den nächsten Abschnitten schließt du eine LED an deinem Arduino an. Diese schaltest du dann in deinem Sketch an und aus. Danach baust du dir einen Dimmer, mit dem du die Helligkeit der LED regelst.

## SO SCHLIESST DU EINE LED AN

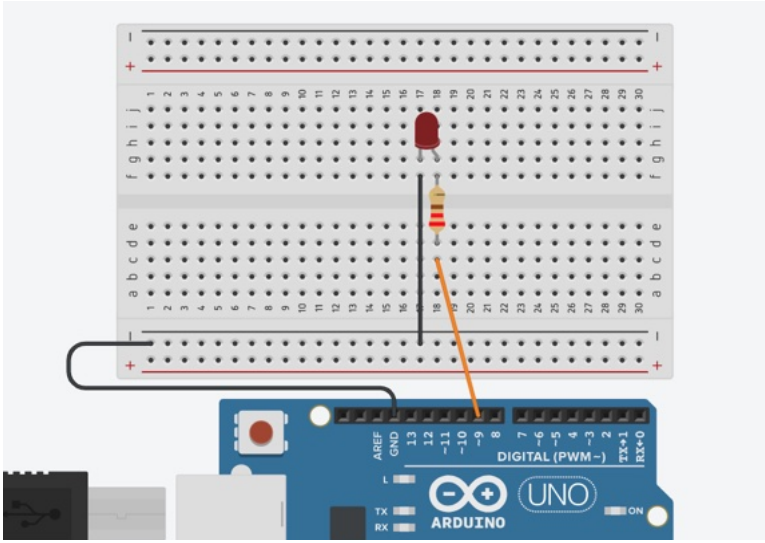
Bevor es losgehen kann, musst du zunächst deine LED mit dem Arduino UNO verbinden. Hierfür benötigst du neben der LED einen Widerstand mit 220Ω, ein Breadboard und drei Kabel.

**Zunächst ein paar Worte zum Breadboard (Steckplatine).** Hierauf kannst du Bauteile platzieren, indem du sie einfach hineinsteckst. Unter den Löchern verlaufen leitfähige Leisten, die den Strom zwischen den Löcher fließen lassen:



Wie du siehst, verläuft der Strom oben und unten auf zwei Leisten quer über das Breadboard. Diese sind auf den meisten Boards mit **+** und **–** gekennzeichnet. Diese Leisten verbindest du zum Beispiel mit den Pins **5V** und **GND** deines Arduinos, sodass du dann von den Leisten aus Strom für deine Bauteile ableiten kannst.

Unter den Löchern im Zentrum sitzen kleinere Leisten, die quer dazu angeordnet sind. Hier läuft der Strom immer unterhalb der fünf Löcher in einer Reihe. In der Mitte des Breadboards befindet sich eine "Brücke", die es in zwei voneinander getrennte Bereiche aufteilt, über die kein Strom hinweg fließt. **Zurück zur LED:** Orientiere dich beim Aufbau an folgender Skizze:



Sprechen wir zunächst über die LED. Diese hat zwei verschieden lange Beine – das längere heißt **Anode** und ist der Pluspol. Das kürzere ist die **Kathode**, der Minuspol. Damit die LED leuchtet, muss Strom durch sie fließen.

Allerdings benötigst du für jede LED einen sogenannten **Vorwiderstand**. Das liegt daran, dass eine LED immer so viel Strom von der Quelle zieht, wie sie bekommen kann – **sie kann ihren Verbrauch nicht selbst regulieren.** Je länger sie brennt, desto leitfähiger wird sie. Das bedeutet, dass auch ihr Stromverbrauch immer weiter ansteigt. Hier kommt der Widerstand ins Spiel, der selbst eine gewisse Menge Strom verbraucht und der LED nur die Menge Strom übrig lässt, mit der sie zurechtkommt.

**Würdest du keinen Widerstand zwischen Arduino und LED einbauen, würde sie mit der Zeit immer mehr Strom ziehen und damit heißer werden – bis sie schließlich durchbrennt.**

Die Anode verbindest du – samt dazwischen geschaltetem Widerstand – mit dem Digital-Pin 9 deines Arduinos. Die Kathode verbindest du hingegen mit der Erde – also mit **GND**.

Und das ist auch schon alles, was du tun musst. Weiter geht es mit der Steuerung der LED.

## DIE LED MIT EINEM SKETCH STEUERN

Jetzt wo deine LED am Arduino angeschlossen ist, möchtest du sie sicher auch einschalten. Kein Problem!

Für einen ersten Test lässt du die LED wieder blinken. Diesmal lässt du sie jedoch anfangs ganz schnell blinken und sie dann immer langsamer werden.

## VARIABLEN UND DIE SETUP-FUNKTION

Zunächst deklarierst du zwei Variablen zu Beginn deines Sketchs – damit sie global, also überall im Sketch, verfügbar sind:

```
int ledPin = 9;
int pause = 0;
```

Die erste Variable **ledPin** legt den Pin fest, an dem die LED angeschlossen ist – in unserem Fall die 9. Die zweite Variable **pause** kommt im Loop ins Spiel und wird die Zeitspanne bestimmen, die die LED nicht leuchtet. Da diese immer größer werden soll, benötigst du hierfür eine Variable, in der du immer größere Zahlen abspeichern kannst.

In der Funktion **void setup()** musst du nichts weiter tun, als den **pinMode** des **ledPin** festzulegen. Da du Signale vom Arduino aus senden möchtest, ist das also **OUTPUT.**

```
void setup()
{
  pinMode(ledPin, OUTPUT);
}
```

## DER LOOP

Hier verwendest du zunächst die Funktion **digitalWrite()**, die du bereits von der letzten blinkenden LED (der internen) kennst.

Du sendest ein **HIGH** an die LED, wartest dann 500 Millisekunden und schaltest sie mit einem **LOW** wieder aus:

```
digitalWrite(ledPin, HIGH);
delay(500);
digitalWrite(ledPin, LOW);
```

Jetzt wird es interessant! In der nächsten Zeile erhöhst du den Wert in der Variablen **pause** um 50. Beim ersten Durchlauf des Loops also von 0 auf 50 – im zweiten von 50 auf 100 – im dritten von 100 auf 150 – und immer so weiter.

Diesen Wert verwendest du dann in der Funktion **delay()** als Millisekunden, die die LED nicht brennen soll:

```
pause = pause + 50;
delay(pause);
```

**Übrigens:** Auch um eine Variable um einen anderen Wert als Eins zu erhöhen, gibt es eine Kurzform. Du kannst das auch so programmieren:

```
pause+= 50;
```

Lade den folgenden Sketch auf deinen Arduino. Wenn alles richtig angeschlossen ist, sollte sie anfangs sehr schnell blinken. Mit der Zeit werden die Abstände immer größer und das Blinken damit langsamer.

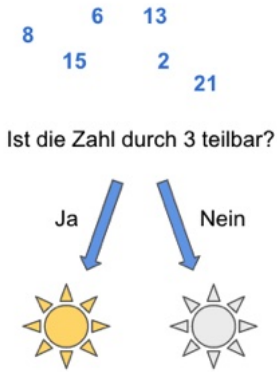
```
int ledPin = 9;
int pause = 0;

void setup()
{
  pinMode(ledPin, OUTPUT);
}

void loop()
{
  digitalWrite(ledPin, HIGH);
  delay(500);
  digitalWrite(ledPin, LOW);
  pause+= 50;
}
```

# 8 BEDINGTE ANWEISUNGEN & VERGLEICHE

Kommen wir zu einem weiteren wichtigen Konzept beim Programmieren: Bedingte Anweisungen – auch bekannt als **if…else…** Mit Hilfe dieser Anweisungen kannst du in deinem Sketch Abzweigungen für bestimmte Ereignisse einbauen.

Zum Einsatz kommt wieder die LED, die sich bereits auf deinem Breadboard befindet. Außerdem verwendest du wieder den Zähler, der jede Sekunde eine Zahl um 1 erhöht und diese im Seriellen Monitor ausgibt.

**Die LED soll diesmal immer nur aufleuchten, wenn die aktuelle Zahl durch 3 teilbar ist. Bei allen anderen Zahlen bleibt sie aus.**



Hierfür eignen sich bedingte Anweisungen. Aber eins nach dem anderen. Zunächst deklarierst du zu Beginn des Sketchs einige Variablen:

```
int ledPin = 9;
int zahl = 1;
int rest = 1;
```

Die ersten beiden kennst du schon. Die dritte Variable **rest** benötigst du, um zu testen, ob die aktuelle **zahl** durch 3 teilbar ist. Den Wert von **rest** setzt du zu Beginn auf 1.

## DIE SETUP-FUNKTION

In der Funktion **void setup()** befindet sich ebenfalls Code, den du bereits kennst. Hier startest du den Seriellen Monitor und definierst den **pinMode** des **ledPin**:

```
void setup()
{
  Serial.begin(9600);
  pinMode(ledPin, OUTPUT);
}
```

## DIE LOOP-FUNKTION

Im Loop gibst du zunächst den aktuellen Wert der Variable **zahl** im Seriellen Monitor aus – beim ersten Durchlauf also die 1.

```
Serial.println(zahl);
```

Anschließend folgt der Test, ob diese Zahl durch 3 teilbar ist. Das machst du mit dem sogenannten **Modulo**.

```
rest = zahl % 3;
```

Den Modulo schreibst du mit einem Prozentzeichen **%**. Er berechnet den Rest, wenn eine ganze Zahl durch eine andere geteilt wird. **Wenn du eine 3 durch eine 3 teilst, ist der Rest 0. Teilst du die 4 durch 3, ist der Rest 1.** Diesen Rest speicherst du in der gleichnamigen Variablen.

Jetzt kommt die bedingte Anweisung ins Spiel. Immer wenn in der Variablen **rest** der Wert 0 steckt (die aktuelle Zahl also durch 3 teilbar ist), soll die LED aufleuchten. Wenn nicht, dann soll sie ausgeschaltet bleiben. Das funktioniert so:

```
if (rest == 0) {
  digitalWrite(ledPin, HIGH);
} else {
```

```
  digitalWrite(ledPin, LOW);
}
```

In Pseudo-Code ausgedrückt, würde diese Anweisung folgendermaßen aussehen:

```
wenn rest 0 ist, dann
  schalte die LED an;
  wenn nicht, dann
  schalte sie aus;
```

In der ersten Anweisung **if** prüfst du, ob etwas **wahr** ist – in unserem Fall, ob in der Variablen **rest** die Zahl **0** gespeichert (und die aktuelle Zahl also durch 3 teilbar) ist. Falls das tatsächlich wahr ist, wird der Code zwischen den geschweiften Klammern **{ }** ausgeführt. Das ist die erste mögliche Abzweigung in deinem Code.

Anschließend gibst du mit **else** für alle anderen Fälle eine weitere Abzweigung vor, die sich in den geschweiften Klammern dahinter befindet. In diesen Fällen wird die LED ausgeschaltet.

## VERGLEICHSOPERATOREN

Immer wenn du prüfst, ob etwas **wahr** oder **falsch** ist, benötigst du die sogenannten **Vergleichsoperatoren**.

Im Code oben hast du diese Prüfung mit einem doppelt Gleichheitszeichen **==** angestellt:

```
rest == 0
```

Immer wenn in der Variablen **rest** der Wert **0** gespeichert war, ergab diese Prüfung **wahr** bzw. **true** und der Code in den geschweiften Klammern wurde ausgeführt.

In allen anderen Fällen ergab die Prüfung **false** und der Code in den geschweiften Klammern hinter **else** wurde ausgeführt.

Es gibt noch einige andere Vergleichsoperatoren, die du sicherlich schon kennst:

| | |
|---|---|
| == | ist etwas gleich? |
| != | ist etwas ungleich? |
| < | ist etwas kleiner? |
| <= | ist etwas kleiner oder gleich? |

| | |
|---|---|
| > | ist etwas größer? |
| >= | ist etwas größer oder gleich? |

Mit ihnen kannst du zum Beispiel prüfen, ob eine Zahl kleiner oder größer als eine andere ist. Etwas Besonderes ist **!=**, denn hiermit wird etwas **true**, wenn die eine Seite **nicht** der anderen entspricht:

```
3 != 4 // ist wahr, also true
```

Zurück zum Sketch. Es fehlen nämlich noch zwei Zeilen für den Zähler, die du jedoch auch schon kennst.

```
zahl++;
delay(1000);
```

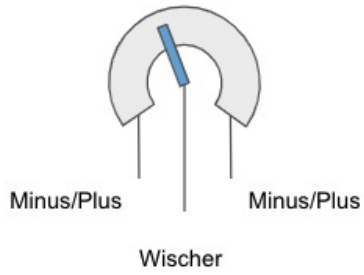
In der ersten Zeile erhöhst du den Wert in der Variablen **zahl** um 1 und in der zweiten Zeile wartest du eine Sekunde, bis du den nächsten Durchlauf der Loop-Funktion startest. Lade den folgenden Sketch nun auf deinen Arduino und probiere ihn aus.

```
int ledPin = 9;
int zahl = 1;
int rest;

void setup()
{Serial.begin(9600);
 pinMode(ledPin, OUTPUT);}

void loop()
{Serial.println(zahl);
  rest = zahl % 3;

  if (rest == 0) {
    digitalWrite(ledPin, HIGH);
  } else {
    digitalWrite(ledPin, LOW);}

  zahl++;
  delay(1000);}
```

# 9 STEUERE DIE HELLIGKEIT MIT EINEM POTI

Ein Poti (von Potentiometer) ist eine praktische Sache, wenn du die “Stärke” von etwas steuern möchtest. Du kennst sie sicherlich aus deinem Alltag, vielleicht ohne zu wissen, dass sich dahinter ein Poti verbirgt. Der Lautstärkeregler an deinem Verstärker, die Knöpfe an deiner E-Gitarre – alles Potentiometer.

Lass uns kurz einen Blick darauf werfen, wie ein Poti funktioniert.



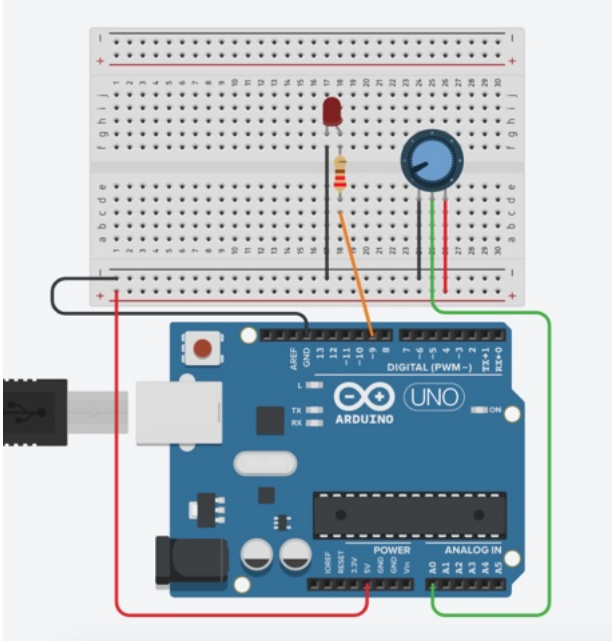

Die meisten Potis haben drei Anschlüsse: zwei Mal Minus bzw. Plus und einen für den Wischer (auch Schleifer genannt). Mit ihm kannst du einen Widerstandswert stufenlos einstellen, indem du den Wischer über einen festen Widerstand gleiten lässt. Je nach Position des Wischers verändert sich der

Stromfluss im Material – und damit die Lautstärke der Musik oder die Helligkeit des Lichts.

**Meist ist es egal, an welchen Enden du Plus und Minus anschließt** – in welche Richtung also der Strom fließt. Allerdings stellst du die Messergebnisse bei einem Wechsel auf den Kopf.

## EIN POTI ANSCHLIESSEN

Zeit also, dein Breadboard um ein Poti zu erweitern. Orientiere dich beim Aufbau an folgender Skizze.




Wenn du alles verkabelt hast, lade einen kleinen Test auf deinen Arduino:

```
int potiPin = 0;
int potiWert;

void setup()
{
  pinMode(potiPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
  potiWert = analogRead(potiPin);
  Serial.println(potiWert);
  delay(10);
}
```

Mit diesem Sketch prüfst du, ob dein Poti funktioniert, indem du seine Werte im Seriellen Monitor ausgibst. **Der integrierte Analog-Digital-Konverter deines Arduinos weist jeder Stellung des Potis einen Wert von 0 bis 1023 zu.**

Versuche einmal, Plus und Minus zu vertauschen. Die Werte im Seriellen Monitor sollten jetzt nicht mehr im Uhrzeigersinn, sondern in der entgegengesetzten Richtung steigen.

## DIE HELLIGKEIT MIT PWM STEUERN

Werte im Seriellen Monitor sind schön und gut, aber du möchtest bestimmt lieber etwas Hardware steuern. Also los – Zeit für einen Dimmer für deine LED.

Die Ausgabewerte kannst du verwenden, um sie direkt auf die Helligkeit der LED “umzulegen”. Das würde bedeuten, dass 0 eine erloschene LED bedeutet und 1023 eine mit maximaler Helligkeit.

**Aber:** Es gibt hier einen kleinen Haken. Um die Helligkeit der LED zu steuern, benötigen wir ein analoges (veränderbares) Signal. Dieses wird mit Hilfe der Pulsweitenmodulation (kurz PWM) erzeugt. Wie zu Beginn erwähnt, kannst du damit ein analoges Signal über einen Digital-Pin simulieren. Die Signalstärke bzw. Stromspannung wird durch Aus- und Anschalten des Pins in einer bestimmten Frequenz erzeugt.

So erreichst du, dass die LED nicht nur einfach (digital) an oder aus ist, sondern, dass du sie durch höhere oder niedrigere Spannungen heller oder dunkler “drehen” kannst.

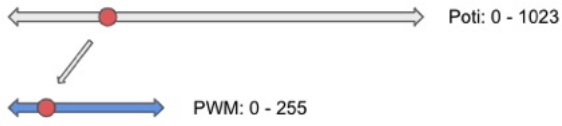
**Allerdings sind bei der Verwendung von PWM nur Werte von 0 bis 255 erlaubt. Dein Poti “liefert” jedoch 0 bis 1023.** Deshalb benötigst du eine weitere neue Funktion: **map();**

## DIE FUNKTION MAP()

Die Map-Funktion ist äußerst praktisch und wird dir zukünftig noch häufiger zur Seite stehen. Ihr Prinzip ist dabei ganz einfach:

**Du nimmst einen bestimmten Wert, der in einem Bereich X liegt. Anschließend ermittelst du, auf welcher Position dieser**

**Wert in einem anderen (größeren oder kleineren) Bereich Y liegt.**




In deinem Sketch sieht das dann folgendermaßen aus:

```
potiWert = analogRead(potiPin);
ledWert = map(potiWert, 0, 1023, 0, 255);
```

Zunächst liest du den aktuellen Wert des Potis mit der Funktion **analogRead()** aus. Das analoge Signal des Potis wird mit Hilfe des Analog-Digital-Konverters in einen Wert von 0 bis 1023 umgewandelt.

Der ermittelte **potiWert** ist der erste Parameter in der Funktion. Gefolgt vom zulässigen Bereich des Potis, also 0 bis 1023.

Zuletzt fehlt noch der Bereich, in dem der entsprechende **potiWert** gefunden werden soll. Das ist der zulässige Bereich

der PWM, also 0 bis 255. Den entsprechenden Wert speicherst du dann in der Variablen **ledWert**.

## LASS DIE LED LEUCHTEN

Nun musst du nur noch den **ledWert** zur LED bringen. Das machst du mit der Funktion **analogWrite()**

```
analogWrite(ledPin, ledWert);
```

Diese bedient sich der Pulsweitenmodulation und erzeugt ein Signal, dass die LED mit der gewünschten, am Poti eingestellten Helligkeit aufleuchten lässt. Probiere es gleich aus! Hier der vollständige Sketch:

```
int ledPin = 9;
int potiPin = 0;

int potiWert;
int ledWert;

void setup()
{
  pinMode(ledPin, OUTPUT);
  pinMode(potiPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
```

```
  potiWert = analogRead(potiPin);
  ledWert = map(potiWert, 0, 1023, 0, 255);
  analogWrite(ledPin, ledWert);
  Serial.println(ledWert);
  delay(10);
}
```

# 11 DA IST MUSIK DRIN!

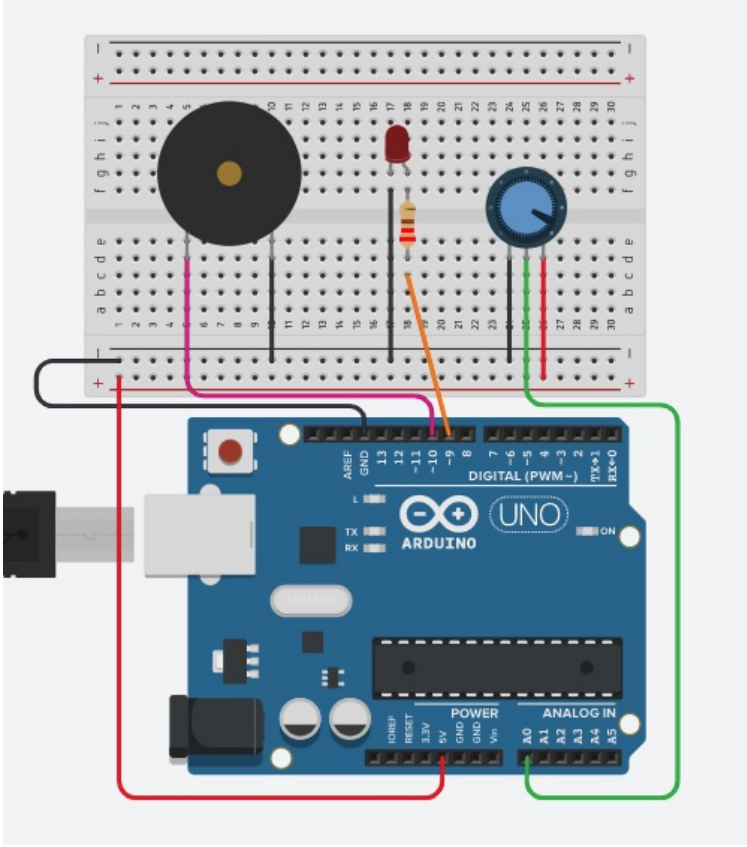
Dein Arduino kann noch viel mehr als “nur” LEDs unterschiedlich hell leuchten lassen. Zum Beispiel Musik machen. Nun lernst du, wie du einen Piezo-Summer verwendest und ihm ein paar charmante Töne entlockst.

## EINEN PIEZO ANSCHLIESSEN

Zunächst benötigst du also wieder ein neues Bauteil: den Piezo-Summer (auch Buzzer genannt). Dabei handelt es sich um ein simples Bauteil, mit dem du durch Anlegen unterschiedlicher Spannungen Töne in unterschiedlicher Höhe spielen kannst.

**Achte darauf, einen passiven Piezo-Summer zu verwenden**. Es gibt auch **aktive**, die jedoch nur einen festgelegten Ton erzeugen, sobald Strom durch sie fließt.

Montiere deinen Piezo wie folgt auf deinem Breadboard neben die anderen Teile:




Du wirst gleich dein Poti verwenden, um unterschiedlich hohe Töne zu erzeugen. Je weiter nach rechts du drehst, desto höher der Ton.

## WAS SIND TÖNE?

Zunächst müssen wir jedoch kurz einen Blick darauf werfen, was Töne eigentlich sind. Im Prinzip handelt es sich hierbei um

Schwingungen mit einer bestimmten Frequenz. Genauer gesagt, sind es Druckschwankungen, deren Schnelligkeit unterschiedlich hohe Töne erzeugen.

**Je größer die Frequenz der Schwankungen ist, desto höher der Ton.** Wenn du ein Festnetztelefon besitzt, kannst du dir hier das Freizeichen anhören. Das ist der sogenannte Kammerton A, der mit einer Frequenz von 440 Hertz (Hz) ertönt.

Um ein A eine Okatve tiefer zu erzeugen, musst du eine Schwingung erzeugen, die halb so groß ist: 220 Hz. Alle anderen Töne der Tonleiter besitzen ebenso festgelegte Frequenzen. Du kannst dir auf Github eine recht umfangreiche Liste von Tönen und den entsprechenden Frequenzen anschauen:

gist.github.com/mikeputnam/2820675

## TÖNE ERZEUGEN

Es gibt eine praktische Funktion, mit der mit deinem Piezo-Summer Töne erzeugen kannst: **tone()**

**Diese Funktion erwartet mindestens zwei Parameter:** Den Pin, an dem dein Piezo angeschlossen ist und die Frequenz. Ein optionaler dritter Parameter ist die Tonlänge.

```
tone(piezoPin, Frequenz, Länge);
```

Wenn du also den Kammerton A mit deinem Piezo an Pin 10 erzeugen willst – ohne die Tonlänge zu begrenzen – sieht der Code hierfür so aus:

```
tone(10, 440);
```

Ganz einfach, oder? Allerdings möchtest du ja eine variable Tonhöhe, die du mit Hilfe deines Potis bestimmst. Deshalb liest du zunächst das Signal des Potis und "mapst" dieses wieder auf einen bestimmten Bereich.

```
potiWert = analogRead(potiPin);
piezoWert = map(potiWert, 0, 1023, 262, 523);
```

Dieser Bereich liegt diesmal zwischen 262 Hz und 523 Hz. Das sind jeweils die beiden Töne C – in zwei unterschiedlichen Oktaven.

Der Variablen **piezoWert** weist du dann die mit der Funktion **map()** gefundene Frequenz zu. Anschließend kommt die Funktion **tone()** ins Spiel:

```
tone(piezoPin, piezoWert);
```

Hier gibst du dann also die Frequenz über den Piezo-Summer am Pin **piezoPin** aus. **Allerdings:** Der Tonbereich ist zwar von einem tiefen und einem hohen C begrenzt – die Töne dazwischen sind jedoch "stufenlos", das heißt, sie entsprechen nicht unbedingt einem D, E oder F.

Lade den folgenden Sketch auf deinen Arduino. Hörst du die Töne und kannst du sie verändern, indem du am Poti drehst? Klingt etwas nach Science Fiction, oder? Als nächstes baust du dir eine Art Spieluhr, die einen Klassiker abspielt: Beethovens "Für Elise".

```
int piezoPin = 10;
int potiPin = 0;

int potiWert;
int piezoWert;


void setup()
{
  pinMode(piezoPin, OUTPUT);
  pinMode(potiPin, INPUT);
  Serial.begin(9600);
}

void loop()
{
  potiWert = analogRead(potiPin);
  piezoWert = map(potiWert, 0, 1023, 262, 523);

  tone(piezoPin, piezoWert);
```

```
  Serial.println(piezoWert);
  delay(10);
}
```

## EINE SPIELUHR MIT LOOPS

Nun lässt du deinen Arduino zur Abwechslung einmal selbst arbeiten, ohne Potis bedienen zu müssen. Du verwendest deinen Piezo-Summer, um darauf mit einer ganzen Reihe von Tone-Funktionen den Anfang des bekannten Stücks “Für Elise” von Beethoven abzuspielen.

Das alleine wäre jedoch etwas einfach. Zusätzlich verwendest du einen For-Loop, um einzustellen, wie oft hintereinander diese Melodie ertönen soll.

Zunächst zur Melodie. Hier liegt ein Sketch auf Github zugrunde, der dir eine Menge Arbeit abnimmt:

gist.github.com/spara/1832855

Hier hat sich der User **spara** die Mühe gemacht, jede einzelne Note, ihre Länge und die Pausen in einen Arduino-Sketch einzutragen. Hier zum Beispiel die ersten beiden:

```
 tone(10, 329.63, 300);
  delay(350);

  tone(10, 311.13, 300);
  delay(350);
```

All diese Noten befinden sich in der Funktion **void loop()** deines Sketchs. Das bedeutet, sie werden bis in alle Ewigkeit wiederholt, sobald die letzte Note gespielt wurde.

Das ist schön, kann aber nicht zuletzt aufgrund der etwas bescheidenen Soundqualität des PIezo-Summers etwas nervig werden. Wir wäre es also, wenn die Melodie nur drei Mal hintereinander gespielt werden würde?

## DER FOR-LOOP

Hierfür gibt es sogenannte **Loops**, die genau das ermöglichen – und die du auch in anderen Projekten immer wieder benötigen wirst.

Schauen wir uns zunächst den **For-Loop** an. Dieser wird nach folgendem Schema erzeugt:

```
for (int i = 0; i < 3; i++) {
  //Code, der wiederholt werden soll
}
```

Zunächst natürlich der Befehl **for**, mit dem du den Loop einleitest. In den runden Klammern **( )** stehen dann einige weitere Parameter.

Zuerst die sogenannte **Initialisierung**. Das bedeutet im Prinzip nur, dass du eine Variable deklarierst und ihr einen Wert zuweist. Üblicherweise bezeichnet man diese Variable einfach mit dem Buchstaben **i**. **Sie ist der Zähler, der nach jedem Durchlauf des Loops erhöht wird** – solange bis eine bestimmte Bedingung nicht mehr **true** (wahr) ist.

Nun die Bedingung: Mit **i < 3** prüfst du, ob die Variable **i** kleiner als 3 ist. **Nur solange das der Fall ist, wird der Code im Loop wieder ausgeführt.** Wenn **i** den Wert 3 besitzt, stoppt der Loop.

Zuletzt der Zähler, der sich nach jedem Durchlauf erhöht: Hierfür verwendest du **i++**

Wie du schon weißt, erhöhst du damit einen Wert um 1. Sobald der Code innerhalb der geschweiften Klammern **{ }** also einmal durchgelaufen ist, erhöht dieser Befehl den Wert in der Variablen **i** von 0 auf 1.

**Werfen wir einen Blick auf die einzelnen Durchläufe:**

- Vor dem ersten Durchlauf entspricht **i** dem Wert 0 -> Der Loop beginnt.

- Nach dem ersten Durchlauf entspricht **i** dem Wert 1 -> Die Bedingung **i < 3** ist wahr -> Der Loop läuft ein zweites Mal.
- Nach dem zweiten Durchlauf entspricht **i** dem Wert 2 -> Die Bedingung ist immer noch wahr -> Der Loop läuft ein drittes Mal.
- Nach dem dritten Durchlauf entspricht **i** dem Wert 3 -> **Die Bedingung ist falsch** -> Der Loop stoppt und wird nicht weiter ausgeführt.

Mit einem For-Loop kannst du also recht gut steuern, wie oft etwas hintereinander ausgeführt werden soll. Das kann zum Beispiel auch das Blinken einer LED sein.

## DER WHILE-LOOP

Es gibt noch eine weitere Methode, einen Loop in deinem Sketch zu erzeugen: **while**

Auch hier wird geprüft, ob eine Bedingung wahr ist und der Code zwischen den geschweiften Klammern wird solange ausgeführt, bis die Bedingung falsch wird.

```
int i = 0;

while (i < 3){
  //Code, der wiederholt werden soll
i++;
}
```

**Es gibt jedoch zwei wichtige Unterschiede zum For-Loop:** Zunächst muss die Variable für den Zähler – also **i** – noch vor dem Loop erzeugt werden. Außerdem erfolgt die Erhöhung des Zählers im Code, der wiederholt wird.

Ansonsten funktioniert dieser Loop gleich. Abgesehen davon, dass du für den For-Loop weniger Platz im Sketch brauchst, gibt es noch eine weitere Überlegung, die mal für **for**, mal für **while** spricht:

**In einem For-Loop musst du bereits im Vorfeld wissen, wie oft er laufen soll. In einem While-Loop hingegen kann der "geloopte" Code selbst Einfluss darauf nehmen, ob er noch ein weiteres Mal wiederholt wird.** So kann zum Beispiel ein besonderes Ereignis den Zähler so weit erhöhen, dass der Loop stoppt.

Je mehr Erfahrung du im Programmieren sammelst, desto einfacher wird dir die Entscheidung zwischen den Loops fallen. Und dann wirst du auch irgendwann über die dritte Loop-Methode stolpern: **do…while**, die wir hier jedoch nicht behandeln.

Zurück zu Beethoven: Der Sketch ist zu lang, um ihn hier abzudrucken. Du kannst ihn auf der Webseite zum Buch herunterladen und anschließend verwenden:

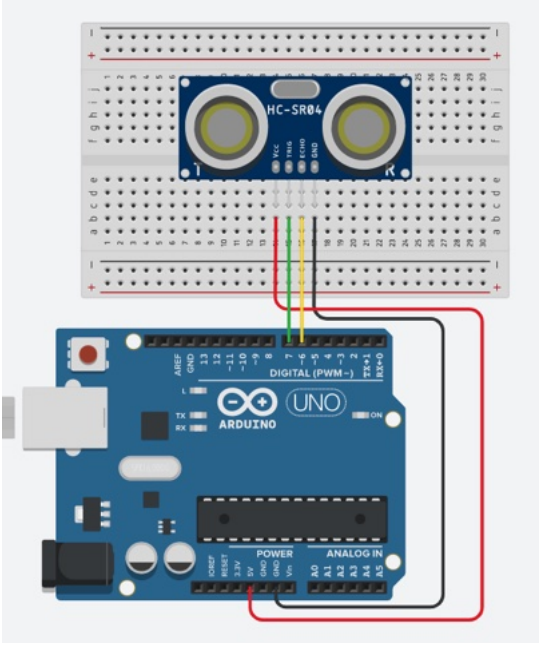
polluxlabs.net/maker-buch

# 12 EIN THEREMIN MIT ULTRASCHALL

Hast du noch Lust auf ein wenig mehr Musik? Jetzt baust du dir ein Theremin, das du mit dem **Ultraschallsensor HC-SR04** bedienst. Du bewegst deine Hand auf den Sensor zu und von ihm weg – dein Arduino berechnet aus dem Abstand die Höhe der Töne, die du dann von deinem Piezo-Summer spielen lässt.

## DEN SENSOR HC-SR04 ANSCHLIESSEN

Orientiere dich beim Aufbau an folgender Skizze:




Als nächstes machst du dich gleich an die Messung von Entfernungen.

## ENTFERNUNGEN MESSEN

Dein HC-SR04 misst den Abstand zu einem Objekt vor ihm – z.B. deiner Hand. In deinem Seriellen Monitor erscheint dann, nach einer kleinen Umrechnung, die Entfernung in Zentimetern.

Zunächst legst du zwei Konstanten fest – die Pins des Arduinos, an denen du die Pins **TRIG** und **ECHO** des HC-SR04 angeschlossen hast. Eine Konstante definierst du mit **const**, zusätzlich zum Typ. Im Gegensatz zu einer Variablen kann der Wert, der in ihr gespeichert ist, nicht mehr an einer anderen Stelle im Code verändert werden.

```
const int trig = 7;
const int echo = 6;
```

Als nächstes benötigst du zwei Variablen. Eine für die Dauer zwischen Aussenden und Empfangen des Ultraschall-Signals. Dein HC-SR04 sendet ein Signal aus, dieses wird von einem Objekt vor ihm zurückgeworfen und vom Sensor wieder empfangen – die hierbei vergangene Zeit speicherst du in der Variablen **duration**.

Die zweite Variable **distance** benötigst du später, um aus der vergangenen Zeit die Entfernung zum Objekt zu berechnen – dazu gleich mehr.

```
int duration = 0;
int distance = 0;
```

## DIE SETUP-FUNKTION

Das Setup im Sketch dürfte für dich keine Überraschungen bergen. Hier startest du lediglich den Seriellen Monitor und legst den jeweiligen **pinMode** von **trig** und **echo** fest. Über den Pin **trig** sendest du das Ultraschall-Signal aus (also OUTPUT) – der Pin **echo** empfängt dieses dann wieder (also INPUT).

```
void setup() {
  Serial.begin (9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}
```

## DER LOOP

Hier wird es nun spannend. Zunächst sendest du ein Signal vom HC-SR04 aus, indem du den Pin **trig** für 10ms auf **HIGH** setzt und anschließend wieder auf **LOW**.

```
digitalWrite(trig, HIGH);
delay(10);
digitalWrite(trig, LOW);
```

Danach misst du die Zeit, die vergeht, bis das Signal wieder am Pin **echo** eintrifft. Das machst du mit der Funktion **pulseIn()**. Die vergangene Zeit speicherst du in der Variablen **duration**.

```
duration = pulseIn(echo, HIGH);
```

Nun ist es so, dass du ja nicht die Zeit wissen willst, sondern den Abstand des Objekts zum Sensor. Hierfür benötigst du eine kleine Rechnung. Das Signal (oder die Ultraschallwelle) des Sensors bewegt sich mit Schallgeschwindigkeit durch die Luft. **Diese Geschwindigkeit beträgt bei Raumtemperatur 343,2 m/s.**

Für den Sensor benötigen wir diesen Wert jedoch in Zentimeter pro Mikrosekunde. Umgerechnet sind das 0.03432 cm/µs.

**Ein Aspekt fehlt jedoch noch:** Nicht die gesamte Zeitspanne interessiert uns, sondern nur jene bis zum Eintreffen des Signals am Objekt. Teile deshalb den Wert in der Variablen **duration** einfach durch 2.

Die Strecke **distance** ergibt sich nun also aus der Hälfte der **duration** * die Schallgeschwindigkeit in cm/µs.

```
distance = (duration / 2) * 0.03432;
```

Zuletzt gibst du die Entfernung noch im Seriellen Monitor aus und wartest mit einem kleinen Delay bis zur nächsten Messung:

```
Serial.print(distance);
Serial.println(" cm");

delay(500);
```

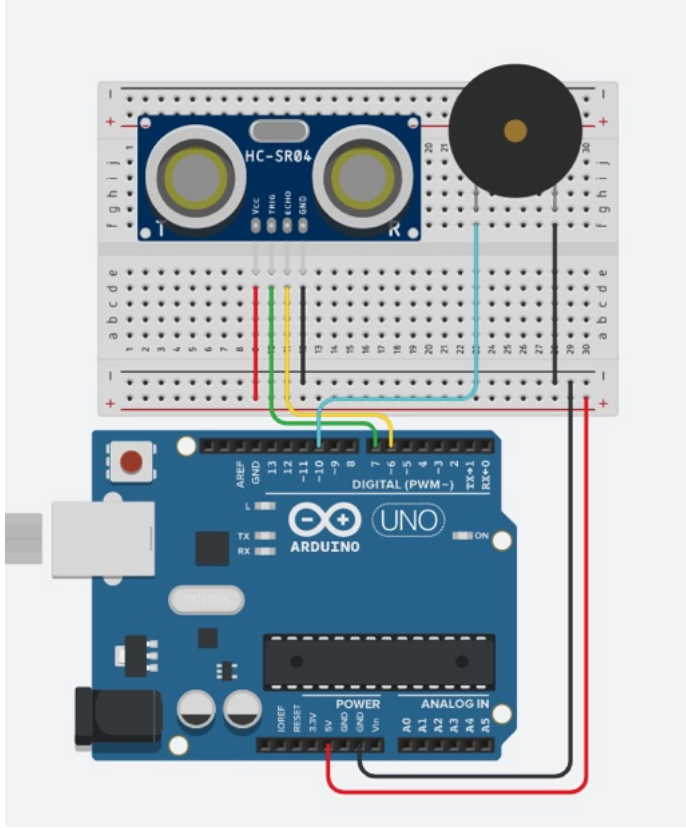
Und das war es auch schon. Lade den folgenden Sketch auf deinen Arduino und probiere deinen neuen Entfernungsmesser gleich aus!

```
const int trig = 7;
const int echo = 6;
int duration = 0;
int distance = 0;

void setup() {
  Serial.begin (9600);
  pinMode(trig, OUTPUT);
  pinMode(echo, INPUT);
}

void loop() {
  digitalWrite(trig, HIGH);
  delay(10);
  digitalWrite(trig, LOW);

  duration = pulseIn(echo, HIGH);
```

```
  distance = (duration / 2) * 0.03432;

  Serial.print(distance);
  Serial.println(" cm");

  delay(500);
}
```

## AUFBAU DES THEREMINS

Du hast den Ultraschallsensor nun auf deinem Breadboard installiert und auch schon Entfernungen gemessen. Mit dem Sensor wirst du die Tonhöhe bestimmen, indem du deine Hand davor bewegst. Fehlt nur noch ein Lautsprecher – in unserem Fall der passive Piezo-Summer. Auch diesen hast du ja bereits kennengelernt. Baue das Theremin wie folgt auf deinem Breadboard auf:




## DIE BENÖTIGTEN KONSTANTEN UND VARIABLEN

Zunächst brauchst du drei Konstanten für die Pins, an denen dein Ultraschallsensor und der Piezo angeschlossen sind.

```
const int trigger = 7;
const int echo = 6;
const int piezo = 10;
```

Dann benötigst du noch zwei Variablen für die Entfernungen. In **distance** speicherst du das aktuelle Messergebnis. Die zweite **distanceHigh** benötigst du, um zu Beginn des Sketchs die maximale Distanz deiner Hand zum Sensor abzuspeichern – dazu gleich mehr.

```
int distance = 0;
int distanceHigh = 0;
```

Wie bei der Drehorgel unterscheiden sich die Frequenzwerte von den Werten, die dein Ultraschallsensor misst. Deshalb kommt auch wieder die Funktion **map()** zum Einsatz. Doch zunächst setzt du die Variable auf Null:

```
int note = 0;
```

## DIE SETUP-FUNKTION

Hier definierst du zunächst wieder den jeweiligen **pinMode** der Pins **Echo** und **Trig** deines Sensors:

```
pinMode(trigger, OUTPUT);
pinMode(echo, INPUT);
```

Anschließend folgt ein While-Loop mit dem Zweck, deinen Sensor zu kalibrieren. Du misst in den ersten drei Sekunden nach dem Programmstart die maximale Entfernung deiner Hand zum Sensor.

Hierfür verwendest du die Funktion **millis()**, die die Anzahl der seit dem Programmstart vergangenen Millisekunden zurückgibt. Solange (while) noch keine 3000 Millisekunden – also drei Sekunden – vergangen sind, misst dein Sensor immer wieder die Entfernung und speichert sie in der Variablen **distance**:

```
while (millis() < 3000) {
 digitalWrite(trigger, HIGH);
 digitalWrite(trigger, LOW);
 distance = pulseIn(echo, HIGH);
```

Doch das reicht noch nicht, sondern du möchtest auch die maximale Entfernung in der Variablen **distanceHigh** speichern, **um dein "Spielfeld" zu begrenzen.** Das machst du mit einer bedingten Abfrage nach jeder Messung.

Immer, wenn die gerade Entfernung **distance** die maximale Entfernung **distanceHigh** überschreitet, wird diese auf den Wert von **distance** aktualisiert.

```
if (distance > distanceHigh) {
 distanceHigh = distance;
}
```

Wie gesagt, der While-Loop wird nur in den ersten drei Sekunden nach dem Programmstart ausgeführt. Sobald diese Zeit verstrichen ist, geht es sofort weiter – mit der Musik.

## IM LOOP SPIELT DIE MUSIK

Auch der Loop beginnt jedes Mal mit der Messung der Entfernung deiner Hand. Anschließend speicherst du diese Entfernung in der Variablen **distance**.

```
digitalWrite(trigger, HIGH);
delay(10);
digitalWrite(trigger, LOW);
distance = pulseIn(echo, HIGH);
```

Diese Entfernung kann natürlich immer noch die von dir kalibrierte maximale Entfernung überschreiten – was sie aber nicht soll. Deshalb prüfst du als nächstes, ob **distance** über **distanceHigh** liegt. Ist das der Fall, limitierst du den gemessenen Wert auf die maximale Entfernung:

```
if (distance > distanceHigh) {
 distance = distanceHigh;
}
```

Alles, was jetzt noch fehlt, sind Töne. Hierfür benötigst du wieder die Funktion **map()**. Du nimmst hier die Entfernung **distance**, deren Wertebereich du auf 50 bis **distanceHigh**

begrenzt. Den Frequenzbereich deiner Töne legst auf 50 bis 3000 Hz. fest. Den in der Funktion “gemappten” Wert weist du dann der Variablen **note** zu.

```
note = map(distance, 50, distanceHigh, 50, 3000);
tone(piezo, note);

delay(10);
```

Anschließend spielst du die gefundene Note mit **tone()** und schließt den Loop mit einem ganz kurzen Delay ab. Und das war es! Lade den folgenden Sketch auf deinen Arduino, kalibriere deinen Sensor und leg mit deiner Karriere am Theremin los!

```
const int trigger = 7;
const int echo = 6;
const int piezo = 10;

int distance = 0;
int distanceHigh = 0;

int note = 0;

void setup() {
  pinMode(trigger, OUTPUT);
  pinMode(echo, INPUT);

  while (millis() < 3000) {
    digitalWrite(trigger, HIGH);
    digitalWrite(trigger, LOW);
```

```
    distance = pulseIn(echo, HIGH);

    if (distance > distanceHigh) {
      distanceHigh = distance;
    }
  }
}


void loop() {
  digitalWrite(trigger, HIGH);
  delay(10);
  digitalWrite(trigger, LOW);

  distance = pulseIn(echo, HIGH);

  if (distance > distanceHigh) {
    distance = distanceHigh;
  }

  note = map(distance, 50, distanceHigh, 50, 3000);
  tone(piezo, note);

  delay(10);
}
```

## ARRAYS

Wie wäre es mit einem Theremin, das weniger nach Science Fiction klingt, sondern nur Töne einer Tonleiter spielt – so wie du das vom Klavier oder von der Gitarre kennst? Kein Problem,

allerdings musst du dich hierfür kurz noch mit einem weiteren wichtigen Programmierbaustein vertraut machen: **Arrays**.

Ein Array ist nichts anderes als eine Sammlung von Variablen oder Werten. Das können z.B. die Frequenzen von Tönen einer Tonleiter sein, aber auch jede andere Kette von Werten, die zusammen gehören – also z.B. Werte für die Helligkeit einer LED, Messergebnisse eines Sensors, Namen usw.

Bisher hast du einer Variablen einen einzelnen Wert zugeschrieben, z.B.

```
int var = 10;
```

Ein Array hingegen kann mehrere dieser Werte beinhalten. Du deklarierst es wie folgt:

```
int meinArray[] = {1, 2, 3, 4, 5};
```

Beachte hier bitte die eckigen Klammern hinter dem Namen des Arrays. Die Werte im Array stehen mit Kommas getrennt innerhalb geschweifter Klammern.

## AUF WERTE IN EINEM ARRAY ZUGREIFEN

Wie kannst du nun die Werte im obigen Array lesen? Das ist ganz einfach: Jeder Wert im Array hat einen Index, also eine Zahl, die die Stelle des Werts im Array festlegt. So wie der Ton C der erste Ton in der C-Dur-Tonleiter ist und das D der zweite. **Das ist einleuchtend, allerdings musst du in C++ (und vielen anderen Sprachen) eine Sache beachten:** Arrays sind **nullindiziert**, was nichts anderes bedeutet, als dass der erste Wert nicht an der 1. Stelle im Array steht, sondern an der "Nullten".

Wenn du im Array **meinArray[]** also auf den Wert 1 zugreifen möchtest, tust du das mit dem Index 0:

```
int ersterWert = meinArray[0];
```

Auf den zweiten Wert (also die 2) greifst du mit **meinArray[1]** zu – und so weiter. Der letzte Wert im obigen Array ist die Zahl 5 mit dem Index 4.

Aber was passiert, wenn du auf einen Wert zugreifen möchtest, der im Array nicht existiert? Wenn du also in **meinArray[]** auf den Index 5 zugreifst? Hier ist dein Sketch dann etwas verloren und wird keine sinnvollen Daten liefern – und im schlimmsten Fall abstürzen.

## WERTE MIT EINEM FOR-LOOP AUSLESEN

Wenn du alle Werte des Arrays hintereinander ausgeben möchtest, ist das ganz einfach mit einem FOR-Loop möglich:

```
for (byte i = 0; i < 5; i = i + 1) {
  Serial.println(meinArray[i]);
}
```

Hier steht der Zähler **i** zunächst auf **0** und sorgt entsprechend in **meinArray[0]** dafür, dass der erste Wert im Seriellen Monitor erscheint. Anschließend wird i um 1 erhöht – **meinArray[1]** gibt also den zweiten Wert aus usw.

## NEUE WERTE HINZUFÜGEN

Noch hat **meinArray[]** also noch keinen Index 5 – aber das kannst du ändern. Neue Werte ergänzt du einfach folgendermaßen:

```
meinArray[5] = 6;
```

Du kannst aber natürlich nicht nur neue Werte ans Ende eines Arrays setzen, sondern auch bestehende Werte ersetzen. Wähle hierfür einfach den Index der Zahl, die du ersetzen möchtest und weise ihm einen neuen Wert zu.

Als nächstes verwendest du ein Array, um darin verschiedene Töne zu speichern – auf die dann dein Theremin zugreifen kann.

## EINE FESTE TONLEITER FÜR DAS THEREMIN

Im Folgenden ersetzt du Teile des Sketchs für das Theremin, damit es nun Töne erzeugt, die auf einer Tonleiter liegen. Du kannst vieles im Sketch weiterverwenden und auch den Aufbau auf deinem Breadboard behalten.

Werfen wir nun also einen Blick auf die Programmteile, die du für das Spielen auf einer Tonleiter benötigst.

Zunächst benötigst du ein Array, das die Frequenzen der Töne, die du spielen möchtest, beinhaltet.

```
//A-Moll Pentatonik
int scale[] = {
  147, 165, 196, 220, 262, 294, 330, 392, 440,
  523, 587, 659, 784, 880, 1047, 1175, 1319, 1568,
  1760, 2093, 2349
};
```

Im obigen Array findest du Töne, die zur A-Moll Pentatonik gehören. Das sind immer wieder die Töne **A – C – D – E – G** in verschiedenen Oktaven.

Statt der A-Moll Pentatonik kannst du natürlich auch z.B. die C-Dur Tonleiter verwenden. Diese sieht dann wie folgt aus.

```
//C-Dur Tonleiter
int scale[] = {
  131, 147, 165, 175, 196, 220, 247, 262, 294,
  330, 349, 392, 440, 494, 523, 587, 659, 698,
  784, 880, 988, 1047
};
```

In diesen Arrays stecken je eine feste Anzahl an Tönen: Das erste Array hat eine Länge von 18 und das zweite besitzt 22 Werte. Die Anzahl der Werte benötigst du später in der Funktion **map()**, die du ja bereits aus dem Vorgänger-Theremin kennst. Was ist jedoch, wenn du weitere Töne hinzufügen möchtest? In diesem Fall müsstest du die Länge des Arrays auch in anderen Teilen des Sketchs anpassen.

Eleganter ist es jedoch, wenn du dein Programm selbst die Länge des Arrays ermitteln lässt. Hierfür benötigst du nur eine Variable und einen For-Loop:

```
int lengthOfScale = 0;

for (byte i = 0; i < (sizeof(scale) /
sizeof(scale[0])); i++) {
  lengthOfScale += 1;
}
```

In der Variablen speicherst du die Länge des Arrays. Im For-Loop erhöhst du den Wert in dieser Variablen in jedem Durchgang um Eins – und zwar so lange, bis der Zähler **i** das Ende des Arrays erreicht hat. Aber wo liegt das Ende? Das ermittelst du mit **sizeof()**. Dieser Befehl ermittelt die Zahl der Bytes im Array. Geteilt durch die Zahl der Bytes des ersten Werts des Arrays (mit dem Index 0) ergibt das die Gesamtzahl der Werte im Array.

Nun hast du also die Anzahl der Töne in deiner Tonleiter. Diese benötigst du nun in der Furnktion **map()**. Hier "mappst" du wieder die Entfernung deiner Hand auf einen Ton bzw. Frequenz. Diesmal ist der Bereich vom ersten Ton im Array **scale[0]** und dem letzten **scale[lengthOfScale – 1]** begrenzt. Von der Variablen **lengthOfScale** ziehst du Eins ab, weil Arrays mit dem Index 0 beginnen.

```
note = map(distance, 100, distanceHigh, scale[0],
scale[lengthOfScale - 1]);
```

In der Variablen **note** steckt nun eine Frequenz – die jedoch nicht unbedingt einem korrekten Ton entspricht. Damit dein Theremin aber die richtigen Töne spielt, benötigst du noch eine letzte Abfrage.

```
for (byte j = 0; j < (lengthOfScale); j++) {

  if (note == scale[j]) {
```

```
      tone(piezo, note);
      break;
    }
    else if (note > scale[j] && note < scale[j + 1]) {
      note = scale[j];
      tone(piezo, note);
      break;
    }
  }
```

Dieser For-Loop läuft über jeden Wert in deinem Array, also so lange wie **j < (lengthOfScale)** gilt. In jeder Iteration prüfst du zunächst, ob der Wert in der Variablen **note** zufällig genau einer Note im Array entspricht. Ist das der Fall, spielst du diese Note mit der Funktion **tone()** und verlässt den Loop mit **break**.

Wenn das aber nicht der Fall ist, liegt deine gespielte **note** offensichtlich zwischen zwei Tönen im Array: **note > scale[j] && note < scale[j + 1]**

Hier spielst du dann den Ton, der eins unter dem Wert in der Variablen **note** steckt – also den nächst tieferen korrekten Ton. Zu kompliziert? Ein Beispiel: Deine Hand hat eine Entfernung zum Sensor, die du mithilfe der Funktion **map()** auf eine Frequenz von 200 Hertz umgerechnet hast. Das ist aber keine Note der A-Moll Pentatonik, sondern liegt irgendwo **zwischen einem G und einem Gis.** Der nächsttiefere, richtige Ton ist laut deinem Array ein G mit 196 Hertz. Und den spielt dein Theremin dann. Auf diese Weise wird dein Theremin immer eine Note spielen, die in deinem Array hinterlegt ist.

Und das war es. Du findest den vollständigen Sketch auf polluxlabs.net/maker-buch

# 13 EINE ALARMANLAGE MIT DEM GERÄUSCHSENSOR

In diesem Projekt baust du dir deine eigene Alarmanlage. Diese besteht aus drei Komponenten: einem Geräuschsensor, einem aktiven Piezo-Summer und einer RGB-LED.

Mit dem Geräuschsensor misst du die Umgebungslautstärke. Sobald ein von dir bestimmter Schwellenwert unterschritten wird, leuchtet die RGB-LED Grün oder Gelb. Sobald die Lautstärke ein festgelegtes Maß überschreitet, leuchtet die LED in Rot und aus dem Piezo ertönt ein schriller Alarmton.

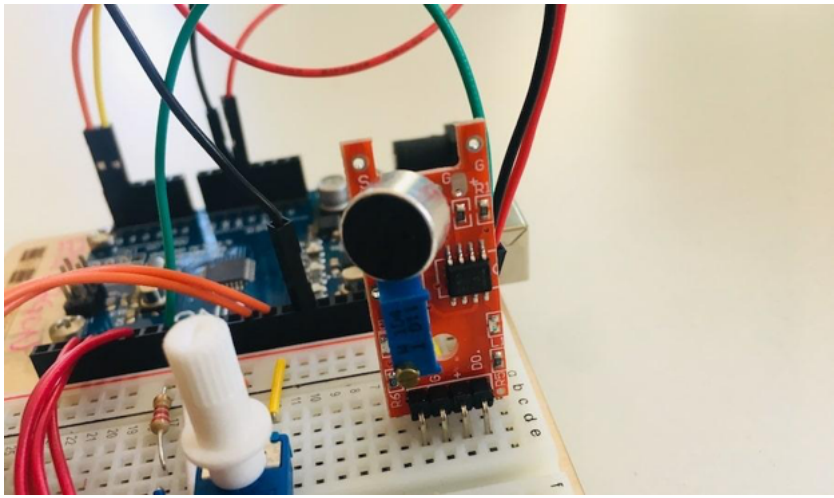
In den folgenden Abschnitten lernst du, wie du den Geräuschsensor, den Piezo-Summer und die RGB-LED anschließt. Danach erfährst du, wie alle drei Bauteile zusammen die Alarmanlage bilden.

## DER GERÄUSCHSENSOR

Zunächst erfährst du, wie du den Geräuschsensor (KY-037, auch “Sound Sensor” genannt) anschließt und damit Geräusche sowohl analog als auch digital in deinem Arduino UNO verarbeitest.

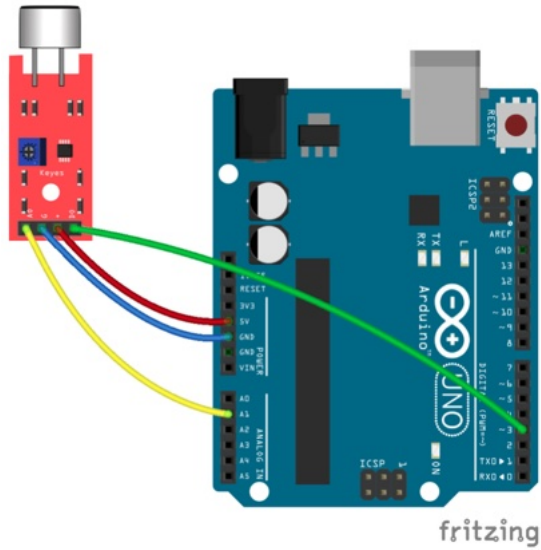
Die wichtigsten Bauteile des Geräuschsensors, den du hier kennenlernst, sind: das Mikrofon, ein Komparator (hier der LM393, der zwei Spannungen miteinander vergleicht) und ein Potentiometer (um den Schwellenwert einzustellen). Du

schließt den Sensor über mindestens 3 der 4 Pins an – Anode, Kathode, Analog- und/oder Digitalausgang.



## ANSCHLUSS AM DIGITALEN AUSGANG

Du kannst den Geräuschsensor auf zwei Arten an deinem Arduino anschließen – analog oder digital. **Der digitale Anschluss ist sinnvoll, wenn du etwas in Gang setzen möchtest, sobald ein lautes Geräusch erkannt wird.** Das könnte zum Beispiel ein Klopfen an der Tür oder ein Knall sein. So schließt du deinen Sensor an:




Versorge deinen Sensor mit Strom über die **Pins + und G** (für Ground, also Minus) und schließe den digitalen Ausgang (DO) am Arduino am digitalen Eingang 3 an. Den analogen Ausgang (AO) verbindest du als nächstes am Arduino mit dem Pin A1. Und das war es schon.

Kopiere dir nun den folgenden Sketch und lade ihn auf deinen Arduino UNO.

```
const int sensor = 3;
const int led = LED_BUILTIN;
int noise  = 0;

void setup() {
```

```
pinMode(sensor, INPUT);
pinMode(led, OUTPUT);
Serial.begin(9600);

}

void loop() {

noise = digitalRead(sensor);
Serial.println(noise);

if (noise == 1){
  digitalWrite(led, HIGH);
}else{
  digitalWrite(led, LOW);
  }
}
```

Sollte die interne LED des Arduinos jetzt dauerhaft leuchten, bedeutet das, dass der Sensor durchgehend ein Geräusch erkennt. **Nimm jetzt einen kleinen Schraubendreher zur Hand und drehe die Schraube am Potentiometer nach links – solange bis die LED ausgeht.**

Jetzt erzeuge direkt neben dem Mikrofon ein lautes Geräusch – schnippe zum Beispiel mit den Fingern. Die LED sollte kurz aufleuchten. Falls nicht – drehe die Schraube wieder leicht nach rechts. Mit etwas Fingerspitzengefühl findest du die richtige Feinjustierung. Du kannst auch im Seriellen Monitor

nachverfolgen, ob der Sensor ein Geräusch erkennt: Bei “Stille” siehst du dort eine Null, bei einem Geräusch eine 1.

## UND JETZT DER ANALOGE AUSGANG

Wenn du den Geräuschsensor analog anschließt, erhältst du in Echtzeit ein Feedback zur Lautstärke. So kannst du zum Beispiel eine LED aufleuchten lassen, sobald **eine von dir bestimmte Lautstärke** überschritten wird.

Du hast den analogen Ausgang (AO) des Sensors ja bereits mit deinem Arduino verbunden. Lade nun folgenden Sketch hoch.

```
const int sensor = A1;
const int led = LED_BUILTIN;
int noise  = 0;

void setup() {

pinMode(sensor, INPUT);
pinMode(led, OUTPUT);
Serial.begin(9600);

}


void loop() {

noise = analogRead(sensor);
Serial.println(noise);

if (noise > 200){
  digitalWrite(led, HIGH);
```

```
} else{
  digitalWrite(led, LOW);
  }
}
```

Um die Lautstärke besser verfolgen zu können, starte den Seriellen Monitor und beobachte dort die Sensorwerte. **Auch jetzt ist wieder etwas Feinjustierung nötig.** Stelle das Potentiometer so ein, dass du im seriellen Monitor Werte siehst, du etwas unter 200 liegen. Wenn du jetzt ein Geräusch machst, das laut genug ist, dass der Wert über 200 schnellt, leuchtet die interne LED des Arduinos auf.

Du siehst also, es reicht hier nicht, *dass* ein Geräusch erkannt wird. Dieses Geräusch *muss auch noch* laut genug sein, um die LED einzuschalten.

Als nächstes baust du dir mit dem Geräuschsensor eine Alarmanlage. Die Lautstärke wirst du mit einer RGB-LED darstellen und sobald eine bestimmte Lautstärke überschritten wird, ertönt ein Alarmsignal aus dem Piezo-Summer.

## DEN AKTIVEN PIEZO UND DIE RGB-LED ANSCHLIESSEN

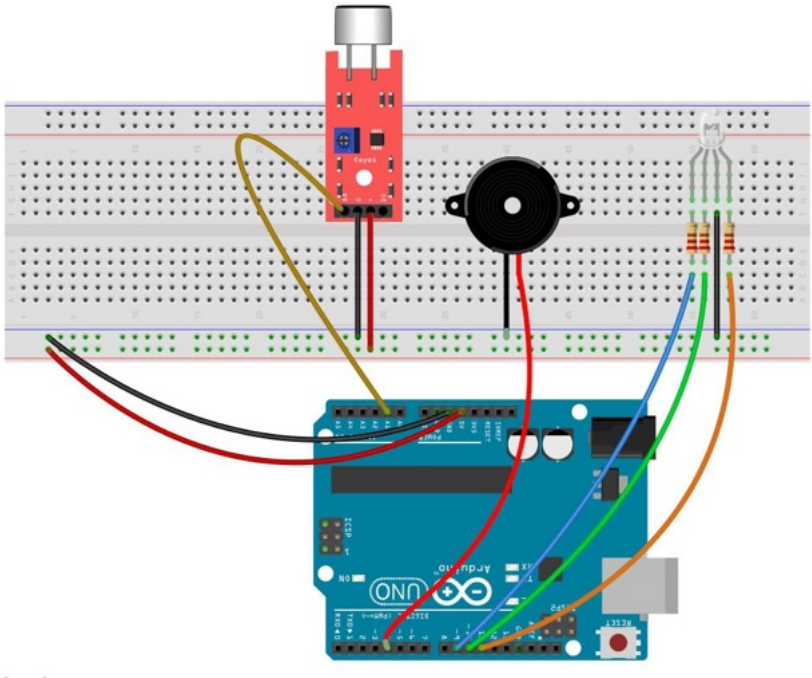
Den Geräuschsensor hast du ja bereits installiert. Allerdings benötigst du für die Alarmanlage nur den Analogausgang (AO)

des Sensors. Das Kabel am Digitalausgang kannst du entfernen.

Mit der Spieluhr und dem Theremin hast du ja bereits den passiven Piezo-Summer kennengelernt. Wie du weißt, kannst du mit diesem unterschiedliche Töne erzeugen. Nicht so mit dem aktiven Piezo: Dieses Bauteil kann lediglich einen einzigen Ton erzeugen – und macht das, sobald es mit Strom versorgt wird. Für eine Alarmanlage ist das jedoch völlig ausreichend.

Wenn du dir nicht sicher bist, ob dein Piezo aktiv oder passiv ist, lege einfach 5V Spannung direkt von deinem Arduino an. Erklingt ein Ton? Dann ist es der aktive Piezo, den du für dieses Projekt brauchst.

Der Anschluss ist ganz einfach. Verbinde das kurze Bein des Piezo-Summers mit Minus und das lange mit dem Digitalpin 4 an deinem Arduino. Wenn später der Geräuschsensor einen Alarm auslöst, leitest du über diesen Pin Strom an den Piezo, der daraufhin zu piepsen anfängt.



Etwas aufwändiger ist der Anschluss der RGB-LED. Insgesamt besitzt sie 4 Beinchen: eine Kathode, die du mit Minus verbindest, und drei Anoden – je eine für die Farben Rot, Grün und Blau. Wie du es von “regulären” LEDs gewohnt bist, musst du auch hier einen 220Ω Vorwiderstand einbauen – hier für jede Anode einen.

Die drei Anoden verbindest du mit den Digitalpins 9, 10 und 11. Diese drei Pins verfügen über Pulsweitenmodulation (PWM), die du bereits kennengelernt hast, also du die Helligkeit einer einfachen LED gesteuert hast.

Möglicherweise passen die oben gezeigten Farbkanäle und Arduino-Pins **je nach Bauart der LED** nicht zusammen. Im Beispiel unten ist der rote Farbkanal rechts von der Kathode, was bei deiner LED jedoch anders sein kann. Das kannst du jedoch leicht herausfinden und beheben, wenn du später die Farbe des Lichts umschaltest: Passe einfach die Variablen im Sketch an schreibe hinter den Farbkanal den richtigen Pin:

```
int ledRed = 10;
int ledGreen = 9;
int ledBlue = 8;
```

Wenn du alles so wie auf der Skizze oben aufgebaut und verkabelt hast, kann es mit dem Code weitergehen.

## DER SKETCH FÜR DIE ALARMANLAGE

Jetzt, wo du alles auf deinem Breadboard aufgebaut hast, wird es Zeit für das Programm.

### DIE VARIABLEN FÜR DIE ANSCHLÜSSE

Gleich zu Beginn des Sketchs legst du fest, welche Hardware du an welchen Pins angeschlossen hast. Diese kannst du mit **const** natürlich auch als Konstanten festlegen. Außerdem benötigst du ein paar Variablen für die Helligkeit der einzelnen Farbkanäle der RGB-LED (z.B. **brightnessRed = 150)** und die Lautstärke, die der Geräuschsensor misst. Diese legst du zu Beginn auf Null fest.

```
int ledRed = 11;
int ledGreen = 10;
int ledBlue = 9;

int brightnessRed = 150;
int brightnessGreen = 150;
int brightnessBlue = 150;

int noise = 0;
int sensor = A1;

int piezo = 4;
```

## DIE SETUP-FUNKTION

Hier gibt es auch nichts, das du nicht bereits schon kennst. Du legst die Pins der LED und des Piezos als **OUTPUT** fest und startest den Seriellen Monitor.

```
void setup() {

  pinMode(ledRed, OUTPUT);
  pinMode(ledGreen, OUTPUT);
  pinMode(ledRed, OUTPUT);

  pinMode(piezo, OUTPUT);

  Serial.begin(9600);
}
```

## DER LOOP

Hier wird es nun spannend. Als erstes misst du die Umgebungslautstärke, denn hierauf basieren später alle weiteren Aktionen im Sketch – also ob deine Alarmanlage anspringt oder nicht.

```
noise = analogRead(sensor);
Serial.println(noise);
```

Anschließend folgt die erste bedingte Anweisung. Wenn nämlich der Geräuschpegel unter einem bestimmten Wert liegt, soll die LED grün leuchten – was so viel bedeutet wie “Die Luft ist rein”. Den Wert von 200 (und die folgenden in den weiteren Anweisungen) kannst du natürlich anpassen. Achte auch darauf, dass du beim ersten Start der Alarmanlage den Geräuschsensor so kalibrierst, dass er bei Ruhe unter dem von dir festgelegten Wert liegt.

```
if(noise <= 200){
  analogWrite(ledGreen, brightnessGreen);
  analogWrite(ledRed, 0);
  analogWrite(ledBlue, 0);
  digitalWrite(piezo, LOW);
  }
```

Wie erwähnt, soll in diesem Zustand die LED grün leuchten. Das erreichst du, indem du nur den grünen Farbkanal mit der von dir

festgelegten Helligkeit **brightnessGreen** aufleuchten lässt. Die beiden anderen Kanäle für Rot (Red) und Blau (Blue) erhalten die Helligkeit **Null**, sind also aus.

Auch der Piezo soll hier nicht ertönen, weswegen du keinen Strom an ihn leitest. Das erreichst du mit dem Parameter **LOW** in der Funktion **digitalWrite()**.

## EINE WEITERE ANWEISUNG

Wenn nun der Geräuschpegel etwas ansteigt, aber immer noch nicht hoch genug für einen Alarm ist, kommt eine zweite bedingte Anweisung ins Spiel – mit **else if{}**.

```
else if(noise > 200 && noise <= 350){
  analogWrite(ledRed, brightnessRed);
  analogWrite(ledGreen, brightnessGreen);
  analogWrite(ledBlue, 0);
  digitalWrite(piezo, LOW);
  }
```

Diese Befehle werden ausgeführt, wenn die Lautstärke zwischen 201 und 350 liegt. Wie gesagt, experimentiere mit diesen Werten, um sie an deine Gegebenheiten anzupassen.

Befindet sich also die Lautstärke in diesem Bereich, wechselt das Licht der LED von Grün zu Gelb. Für diese Farbe gibt es keinen eigenen Farbkanal, weswegen du sie kurzerhand mischst. Gelb ist hier eine Mischung aus Rot und Grün. Deshalb

schaltest du diese Farbkanäle der LED an und lässt den blauen Kanal erlöschen.

## ALARM!

Wenn es nun noch lauter wird, soll der Alarm ertönen. Die LED strahlt in einem satten Rot und der Piezo-Summer fängt an zu piepsen.

```
else if(noise > 350){
  analogWrite(ledRed, brightnessRed);
  analogWrite(ledGreen, 0);
  analogWrite(ledBlue, 0);
  digitalWrite(piezo, HIGH);
  delay(10000);
  }
```

Diesmal schaltest du also nur den roten Kanal der LED ein. Außerdem leitest du nun Strom vom Arduino an den Piezo – mit der Funktion **digitalWrite(piezo, HIGH);**

Sicherlich sollte eine Alarmanlage so lange Krach machen, bis sie ausgeschaltet wird. Für ein erstes Experiment reicht jedoch vielleicht auch erst einmal eine Sekunde. Deshalb befindet sich am Ende noch ein **delay()** von 1.000 Millisekunden.

Du findest den Sketch auf **polluxlabs.net/maker-buch** – kalibriere deinen Geräuschsensor und probiere deine Alarmanlage gleich aus.

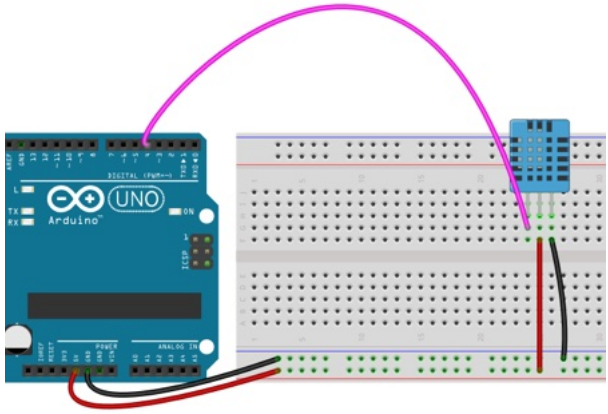
# 14 DER TEMPERATURSENSOR DHT11

Kommen wir zu einem weiteren Bauteil, das du sicherlich in vielen Projekten verwenden wirst: den Temperatursensor. In diesem Fall der beliebte DHT11, der nicht nur die Temperatur, sondern auch die Luftfeuchtigkeit messen kann.

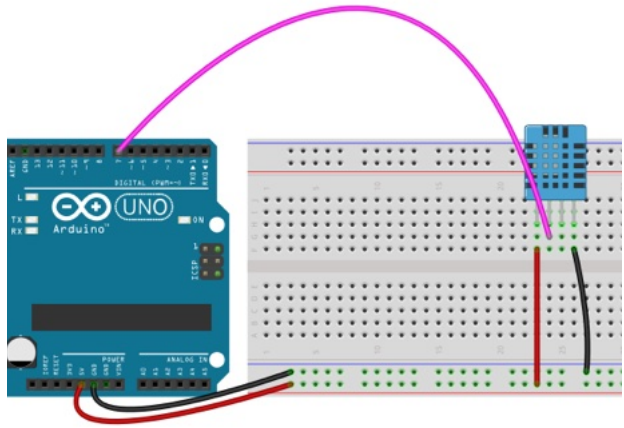
Du erfährst im Folgenden, wie du ihn anschließt, wie du die passenden Bibliotheken installierst und Daten misst. Später wirst du mehrere Projekte mit diesem Sensor bauen: von der Wetterstation bis zum temperaturgesteuerten Ventilator.

## ANSCHLUSS AM ARDUINO

Der Anschluss des DHT11 ist ganz einfach, denn hierfür benötigst du nur drei Kabel. Orientiere dich beim Aufbau an folgender Skizze.



Vergewissere dich noch einmal, dass du alle Kabel richtig angeschlossen hast. **Sollte dein DHT11 vier Pins haben, schließe ihn bitte wie folgt an:**




**In dieser Skizze ist der Sensor an Pin 7 angeschlossen. Nutze für den kommenden Sketch bitte Pin 4**

## BIBLIOTHEKEN INSTALLIEREN UND VERWENDEN

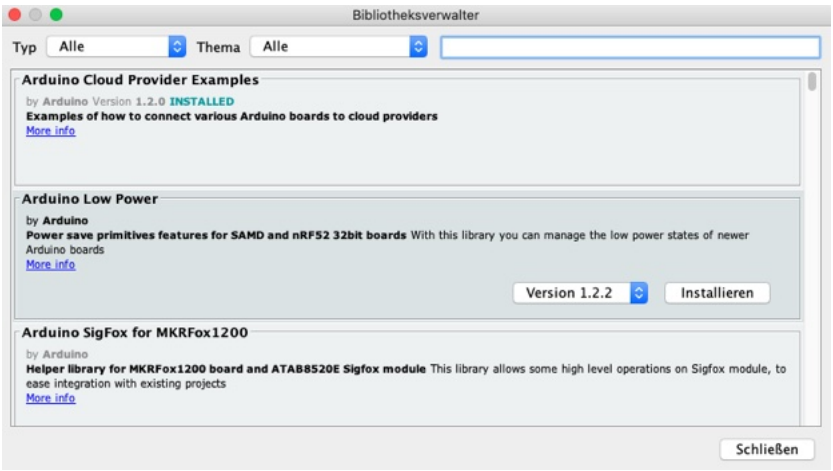
Bibliotheken sind Helfer, die du bald nicht mehr vermissen möchtest. Sie übernehmen im Hintergrund viele Aufgaben, die du ohne sie manuell programmieren müsstest. Dazu gehört z.B. die Messung von Temperatur und Luftfeuchtigkeit mit dem DHT11.

Die meisten Bibliotheken kannst du **direkt in deiner Arduino IDE** installieren und updaten. Einmal installiert, bindest du zu

Beginn deines Sketchs ein und kannst danach auf ihre Funktionen zugreifen. Doch eins nach dem anderen.

## DER BIBLIOTHEKSVERWALTER

Du findest im Menü **Werkzeuge** deiner Arduino IDE den Eintrag **Bibliotheken verwalten**. Mit einem Klick hierauf öffnet sich ein neues Fenster – dein Bibliotheksverwalter.



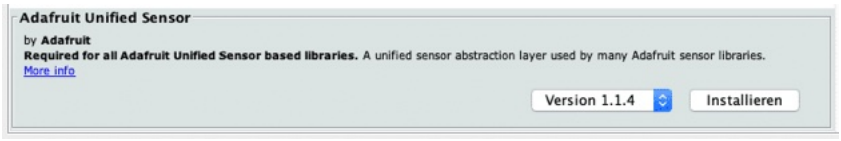

Werfen wir kurz einen Blick auf den Inhalt des Fensters.

Oben links findest du das Dropdown-Menü **Typ**. Hier kannst du z.B. nur die Bibliotheken anzeigen lassen, die du bereits installiert hast und für die ein Update verfügbar ist. Die anderen Einträge sind hingegen eher nebensächlich.

Auch das Dropdown **Thema** rechts daneben wirst du so gut wie nie brauchen. Viel wichtiger ist das Suchfeld rechts daneben. Das verwendest du gleich, um die passenden Bibliotheken für den DHT11 zu installieren.

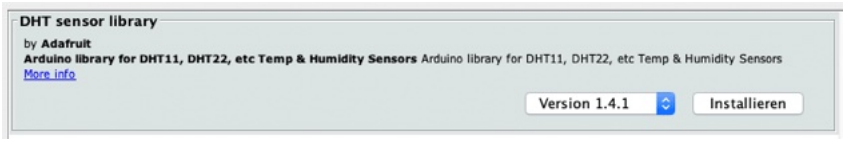
## DIE BIBLIOTHEKEN FÜR DEN DHT11

Gib ins Suchfeld **Adafruit Unified Sensor** ein – die Sucherergebnisse erscheinen nach kurzer Zeit von selbst. Scrolle etwas nach unten, bis du den richtigen Eintrag findest:




Wenn du mit der Maus über den Eintrag “fährst”, erscheint der Button **Installieren**. Links davon kannst du die Version auswählen, die du installieren möchtest. In den meisten Fällen ist das die aktuelle Version – du musst hier also nichts ändern.

Für den Sensor DHT11 benötigst du jedoch gleich zwei Bibliotheken. Suche als nächstes nach **DHT11**. In den Ergebnissen taucht nun der Eintrag **DHT sensor library** auf.



Klicke nun auf **Installieren**, um die neueste Version zu laden.

**Hinweise:** Möglicherweise wirst du vom Bibliotheksverwalter gefragt, ob du noch weitere benötigte Bibliotheken installieren möchtest. Bestätige das mit **Yes**. Lass dich bitte auch nicht von den Versionsnummern auf den Screenshots oben irritieren: Bibliotheken werden laufend aktualisiert, sodass die Nummer bei dir bereits höher sein kann.

Du hast nun alle Bibliotheken für deine Messung zusammen. Als nächstes lernst du, wie du sie in deinen Sketch einbindest und damit die Temperatur und Luftfeuchtigkeit im Seriellen Monitor ausgibst.

## TEMPERATUR & LUFTFEUCHTIGKEIT MESSEN

Jetzt wird es ernst: Mit Hilfe deiner neuen Bibliotheken misst du mit dem DHT11 die Temperatur und Luftfeuchtigkeit – und zeigst diese Werte in deinem Seriellen Monitor an. Lass uns zunächst einen Blick auf die Bibliotheken werfen.

### BIBLIOTHEKEN IM SKETCH EINBINDEN

Damit du eine Bibliothek (die du zuvor installiert hast) auch verwenden kannst, musst du sie zu Beginn deines Sketchs einbinden. Damit kann dein Programm auf die Funktionen, die in der Bibliothek hinterlegt sind, zugreifen.

Die Bibliothek für deinen Sensor bindest du ganz einfach mit einer Zeile Code ein:

```
#include "DHT.h"
```

Wie du siehst, verwendest du hierfür den Befehl **#include** gefolgt vom Namen der Bibliothek samt der Endung **.h** in Anführungsstrichen. **Wichtig:** Zeilen mit **#include** darfst du nicht mit einem Semikolon **;** abschließen – **eine Ausnahme, die ansonsten zu einem Fehler führen würde.**

Die Bibliothek **Adafruit Unified Sensor** musst du nicht einbinden, diese arbeitet nur im Hintergrund.

Oft sind die Namen der Bibliotheken im Bibliotheksverwalter und hinter dem Befehl #include identisch. In diesem Fall ist das ausnahmsweise nicht der Fall – lass dich hiervon bitte nicht irritieren.

**Übrigens:** Fast alle Bibliotheken hinterlegen nach der Installation Beispiele. Du findest sie im Menü **Datei -> Beispiele**. Für unsere Bibliothek wähle hier den Eintrag **DHT sensor library**. Dort findest du Beispiel-Sketches, in denen du mehr Informationen zum Einbinden und den Funktionen erhältst.

## ANSCHLUSS UND SENSOR FESTLEGEN

Damit die Bibliothek weiß, an welchem Pin der Sensor angeschlossen ist und um welchen Sensor (DHT11 oder

DHT22, der “größere Bruder” des DHT11) es sich überhaupt handelt, musst du diese beiden Parameter zunächst definieren:

```
#define DHTPIN 4
#define DHTTYPE DHT11
```

Auch das machst du zu Beginn des Sketchs, gleich nachdem du die Bibliothek eingebunden hast. Du verwendest hierfür die Funktion **#define** – die ebenso wie **#include** nicht mit einem Semikolon **;** abgeschlossen wird.

## VARIABLEN FÜR DIE TEMPERATUR & LUFTFEUCHTIGKEIT

Damit du deine Messungen temporär speichern kannst, benötigst du zwei Variablen – eine für die Temperatur und eine für die Luftfeuchtigkeit:

```
float temp;
float humidity;
```

Da dein Sensor hier auch die Zehntel-Grad bzw. -Prozent misst, müssen diese Variablen vom Typ **float** sein.

## DEN SENSOR STARTEN

Als nächstes erstellst du ein Objekt auf Basis der Bibliothek namens **dht**. Dieser Instanz gibst du den oben definierten Anschlusspin und den Sensortyp mit:

```
DHT dht(DHTPIN, DHTTYPE);
```

Nun folgt auch schon die Setup-Funktion. Hier startest du zunächst den Seriellen Monitor und gleich danach auch den Sensor:

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

## MESSUNG UND AUSGABE DER WERTE

Zeit für den Loop. Hier kannst du auf zwei Funktionen zurückgreifen, die dir die Bibliothek des Sensors bereitstellt. Mit diesen Funktionen speicherst du die aktuellen Messwerte in der jeweiligen Variablen:

```
temp = dht.readTemperature();
humidity = dht.readHumidity();
```

Wie du sicherlich schon erkannt hast, liest die Funktion **dht.readTemperature()** die Temperatur und **dht.readHumidity()** die Luftfeuchtigkeit. Diese Werte kannst du wie gewohnt Variablen zuweisen, die du dann später weiterverwendest.

Interessant ist die Schreibweise der Funktionen. Zu Beginn siehst du das Objekt **dht** – dieses "weiß" mittlerweile, welchen Sensor du verwendest und an welchem Pin dieser angeschlossen ist. Dahinter – getrennt mit einem Punkt – greifst du auf eine "Unterfunktion" zu, die dein Objekt bereitstellt: entweder **readTemperature()** oder **readHumidity()**.

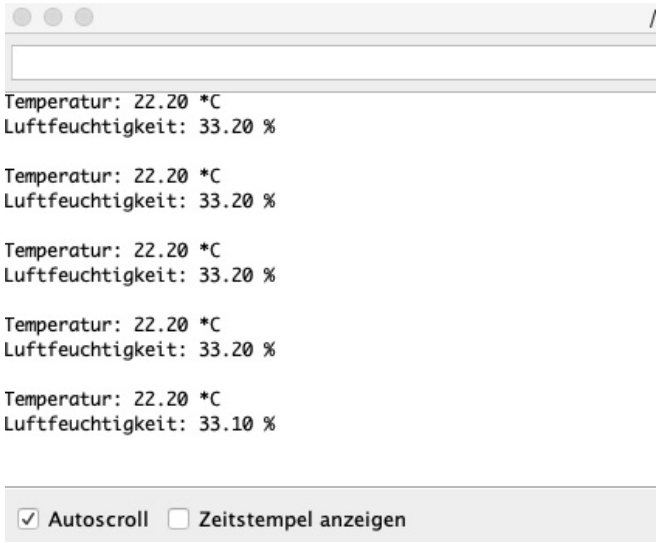
Als nächstes gibst du deine Messwerte im Seriellen Monitor aus. Wie das funktioniert, weißt du bereits aus den vorangegangenen Projekten. Wie du siehst, kommen hier nun die beiden Variablen **temp** und **humidity** zum Einsatz:

```
Serial.print("Temperatur: ");
Serial.print(temp);
Serial.println(" *C");
Serial.print("Luftfeuchtigkeit: ");
Serial.print(humidity);
Serial.println(" %");
```

Damit dein Sensor nicht von einer Messung zur nächsten hetzt, schließe den Loop mit einem Delay ab – z.B. zwei Sekunden:

```
delay(2000);
```

Und das war es! Lade den folgenden Sketch auf deinen Arduino und öffne deinen Seriellen Monitor. Du solltest nun etwas in dieser Art sehen:




```
#include "DHT.h"

#define DHTPIN 4
#define DHTTYPE DHT11

float temp;
float humidity;
```

```
DHT dht(DHTPIN, DHTTYPE);


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

void loop() {

  temp = dht.readTemperature();
  humidity = dht.readHumidity();

  Serial.print("Temperatur: ");
  Serial.print(temp);
  Serial.println(" *C");
  Serial.print("Luftfeuchtigkeit: ");
  Serial.print(humidity);
  Serial.println(" %");

  Serial.println();
  delay(2000);

}
```

# 15 DAS LC-DISPLAY

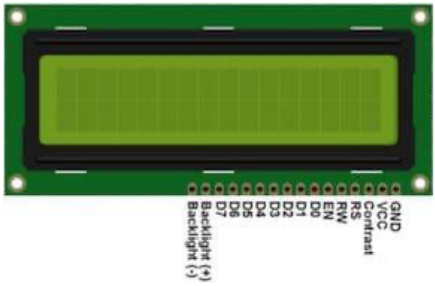
Bis hierhin hast du bereits alle möglichen Daten gemessen: Entfernungen, Temperaturen, die Luftfeuchtigkeit, etc. Diese Daten hast du oft im Seriellen Monitor ausgegeben – was bei der Entwicklung eines Projekts auch ausreichend ist. Jetzt gehst du einen Schritt weiter und stellst deine Daten auf einem Display dar.

Eines der bekanntesten ist das LCD (Liquid Crystal Display, oft fälschlicherweise auch LCD-Display genannt). Das benötigt für den Anschluss eine ganze Menge Kabel, dafür ist seine Funktion ziemlich leicht zu verstehen. In den folgenden Abschnitten erfährst du, wie du das Display anschließt und Texte sowie Zahlen darauf anzeigst.

## EIN LC-DISPLAY ANSCHLIESSEN

Ein LCD an den Arduino anzuschließen ist eigentlich nicht schwer – auch wenn einige Kabel dafür nötig sind.

Schauen wir uns zunächst die Anschluss-Pins des Displays an:



Am Display findest du insgesamt 16 Pins. Jeweils ganz außen (GND, Backlight -) legst du die Erde (GND) an und eins daneben (VCC, Backlight +) 5 Volt über einen 220Ω Widerstand. Am dritten Pin von links (Contrast, oder V0) schließt du ein Poti an, mit dem du den Kontrast des Bildschirms einstellen kannst.

**An den Pin RS sendest du von deinem Arduino Board entweder Daten oder Befehle.** Wenn vom Datenpin des Arduinos kein Strom kommt, kannst du Befehle übertragen wie z.B. “Setze den Cursor an die 1. Stelle der 2. Reihe” oder “Lösche das Display”. Wenn dein Sketch Strom anlegt, sendest du Daten, sprich die Zeichen, die du ausgeben möchtest.

Der Pin RW steht für “read – write” – ziemlich klar: Wir wollen in der Regel Daten und Befehle “schreiben”, also senden. Um den Schreibmodus zu aktivieren, verbinde diesen Pin mit GND.

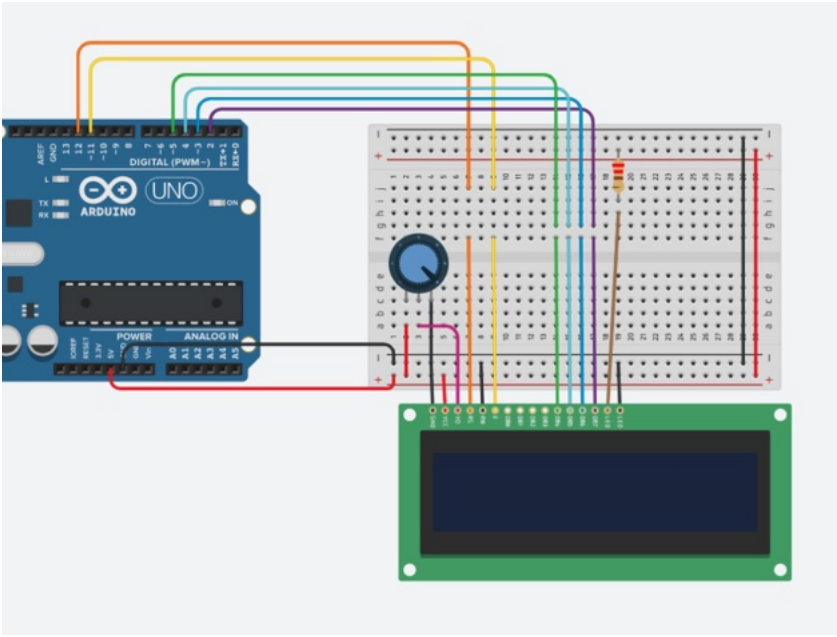
## DIE DATEN-PINS DES LC-DISPLAYS

Jetzt kommen insgesamt 9 Pins, von denen du 5 an deinen Arduino anschließen musst. Neben RW findest du EN wie “Enable” – dieser Pin ermöglicht das Schreiben auf die 8 Daten-Pins daneben: Wenn hier Strom anliegt, werden Daten angenommen. Apropos 8: Du kannst deinen Bildschirm entweder mit 8 Bit oder 4 Bit betreiben. Ein großes A sieht in 8 Bit z.B. so aus: **01000001**. Diese Bits landen in den Pins D0 bis D7.

Der Betrieb mit 4 Bit hat den **Vorteil, dass du nur die Hälfte der Pins ansteuern musst, was weniger Kabelei bedeutet.** Das bedeutet dann allerdings auch, dass man immer 2 Zugriffe

benötigt, um etwas zu übertragen, denn 2 x 4 = 8. Das muss allerdings nicht deine Sorge sein.

Orientiere dich beim Anschluss deines LC-Displays an folgender Skizze.

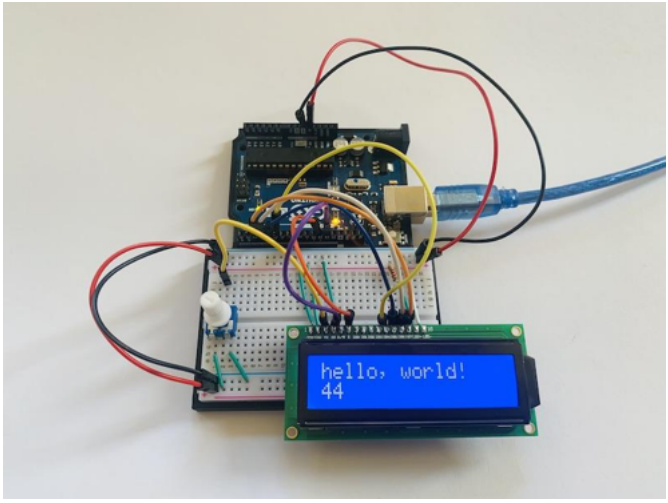



Wenn du fertig bist, überprüfe noch einmal jede Verbindung – hier schleichen sich schnell Fehler ein. Schalte deinen Arduino ein – das Display sollte nun aufleuchten. Probiere auch, die Helligkeit mit dem Poti zu regulieren.

Funktioniert? Als nächstes erfährst du, wie du Buchstaben und Zahlen auf deinem LCD anzeigst.

## TEXT UND ZAHLEN DARSTELLEN

Wenn dein Display aufleuchtet und du die Helligkeit einstellen kannst, ist das schon einmal ein gutes Zeichen. Spannend wird es jetzt – mit einem ersten Test, in dem du Buchstaben und Zahlen auf dein Display bringst.




Um dein LC-Display zu steuern, benötigst du überraschend wenig Code. Das liegt an der Bibliothek, die dir hier das Leben leicht macht. Diese Bibliothek ist sogar schon vorinstalliert – **d.h. du musst sie nicht erst über den Bibliotheksverwalter installieren.**

Binde also die passende Bibliothek zu Beginn deines Sketchs ein:

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

Anschließend erstellst du ein Objekt samt der Information an welchen Pin des Arduinos welchen Display-Pin angeschlossen hast:

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
```

Das sind folgende Verbindungen:

| **Arduino** | **LC-Display** |
|---|---|
| 12 | RS |
| 11 | Enable (E) |
| 5 | D4 |
| 4 | D5 |
| 3 | D6 |
| 2 | D7 |

In der Setup-Funktion “startest” du zunächst dein Display. Hierzu gehört auch die Angabe, wie viele Zeichen das Display hat. Für ein Display mit 16×2 Zeichen ist das diese Zeile:

```
lcd.begin(16, 2);
```

## TEXT AUF DEM DISPLAY DARSTELLEN

Zeit für ein paar Buchstaben! Glücklicherweise bekommst du diese sehr einfach auf dein Display. Alles, was du benötigst, ist die Funktion **lcd.print()**. Dieser Funktion gibst du einfach einen **String** mit auf den Weg, der dann auf deinem Display erscheint.

```
lcd.print("Hello, world");
```

Ziemlich einfach, oder? Dieser Text wird automatisch in die erste Reihe geschrieben, beginnend im ersten Feld des Displays – so wie du es oben auf dem Foto siehst. Du kannst aber natürlich auch festlegen, an welcher Stelle etwas erscheinen soll. Hierfür verschiebst du einfach den **Cursor**. Den kannst du dir vorstellen, wie den blinkenden vertikalen Strich in deiner Lieblings-Textverarbeitung.

Angenommen, du möchtest etwas in die zweite Zeile schreiben und im ersten Feld beginnen. Dann setzt du zuerst den Cursor, bevor du den Text ausgibst:

```
lcd.setCursor(0, 1);
```

Das Format dieses Befehl sieht so aus: **lcd.setCursor(Spalte, Zeile)**. Zunächst gibst du also eine **0** an für die erste Spalte – das Feld ganz links. **Beachte, dass du hier nicht bei 1 anfängst zu zählen, sondern bei der 0.**

Demnach ist der zweite Parameter für die zweite Zeile also **1**. Der Cursor befindet sich nun also in der zweiten Zeile im ersten Feld. Hier soll nun ein Counter erscheinen, der die Sekunden seit dem Programmstart hochzählt.

## ZAHLEN DARSTELLEN

Genauso einfach wie es ist, Text auszugeben, ist auch die Darstellung von Zahlen. Du trägst sie einfach in die Funktion **lcd.print()** ein:

```
lcd.print(10);
```

Um es etwas interessanter zu machen, probiere einmal einen Counter aus:

```
lcd.print(millis() / 1000);
```

Wie du siehst, ist es sogar möglich, eine andere Funktion zu nutzen, um etwas auf dem Display darzustellen – in diesem Fall **millis()**. Diese Funktion kennst du schon vom Theremin – sie

gibt die Millisekunden seit dem Programmstart zurück. Damit allerdings Sekunden erscheinen, teilst du die Millisekunden durch 1000 – auch direkt im Befehl **lcd.print()**.

Lade den folgenden Sketch auf deinen Arduino. Auf deinem Display erscheint nun **Hello, world** in der ersten Zeile. In der zweiten zählt dein Counter die Sekunden seit dem Programmstart hoch.

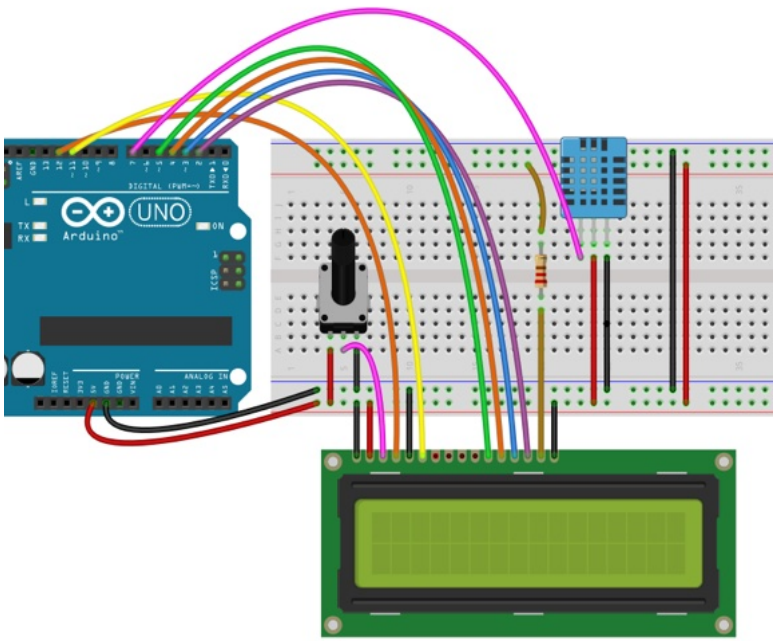
**Hinweis:** Siehst du keine Zeichen auf dem Display? Dann probiere zunächst, den Kontrast mit deinem Poti zu regulieren. Wenn das nicht funktioniert, prüfe bitte noch einmal, ob du alle Pins richtig miteinander verbunden hast.

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {

  lcd.begin(16, 2);
  lcd.print("Hello, world");
}

void loop() {

  lcd.setCursor(0, 1);
  lcd.print(millis() / 1000);
}
```

# 16 DEINE EIGENE WETTERSTATION

In diesem Projekt verbindest du deine Erfahrung mit dem DHT11 und deinem LC-Display – indem du dir deine eigene Wetterstation baust. DIE WETTERSTATION AUFBAUEN

Wenn du dein LC-Display samt Verbindung zum Arduino noch auf dem Breadboard hast – wunderbar! Ansonsten erwarten dich noch einmal ein paar Minuten intensives Verkabeln.



Wie du siehst, ist hier einiges los auf dem Breadboard. Vergewissere dich bitte, dass alle Anschlüsse richtig sind, bevor du mit dem Sketch fortfährst.

## DER SKETCH FÜR DIE WETTERSTATION

Du beginnst wie üblich mit den Bibliotheken, einigen Definitionen und Variablen und dem Erstellen von Objekten. Du kennst diese Zeilen bereits aus den vorangegangen Projekten – achte jedoch bitte auf eine Änderung: Dein Temperatursensor ist diesmal mit dem Digital-Pin 7 deines Arduinos verbunden und entsprechend im Sketch definiert:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

#include "DHT.h"
#define DHTPIN 7
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

float temp;
float humidity;
```

Auch in der Setup-Funktion erwartet dich nichts Neues. Du “startest” hier den Seriellen Monitor, das LC-Display und den Sensor:

```
void setup() {
```

```
  Serial.begin(9600);
  lcd.begin(16, 2);
  dht.begin();
}
```

## DER LOOP

Im Loop misst du zuerst die Temperatur und die Luftfeuchtigkeit:

```
temp = dht.readTemperature();
humidity = dht.readHumidity();
```

Und gibst sie danach zunächst in deinem Seriellen Monitor aus:

```
Serial.print("Temperatur: ");
Serial.print(temp);
Serial.println(" *C");
Serial.print("Luftfeuchtigkeit: ");
Serial.print(humidity);
Serial.println(" %");
Serial.println();
```

Nun möchtest du die Daten natürlich nicht in deinem Seriellen Monitor “verstecken”, sondern sie auf dem LC-Display anzeigen. Zunächst die Temperatur:

```
lcd.setCursor(0, 0);
lcd.print(temp + String(" C"));
```

Hier setzt du zunächst den Cursor des Displays in das erste Feld der ersten Zeile. Anschließend gibst du mit der Funktion **lcd.print()** die Temperatur aus, die in der Variablen **temp** gespeichert ist. Hier ist allerdings nur die Zahl gespeichert. Um dahinter noch ein **C** für Celsius anzuzeigen, benötigst du einen kleinen Trick.

Mit der Funktion **String()** konvertierst du das Zeichen **C** in den Datentyp **String**. Wenn du ausschließlich Zeichen ausgeben möchtest, musst du diese Konvertierung nicht durchführen. Wenn du allerdings, wie hier, eine Variable des Typs **float** mit Zeichen mischen möchtest, ist sie unabdingbar.

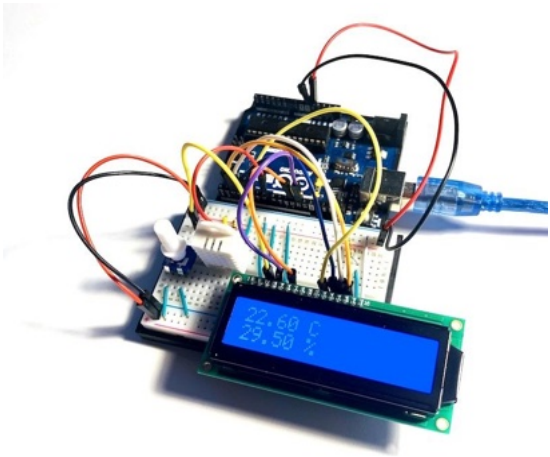
Dasselbe gilt für die Luftfeuchtigkeit. Hier setzt du zunächst den Cursor eine Zeile tiefer und gibst den Wert samt Prozentzeichen auf dem Display aus.

```
lcd.setCursor(0, 1);
lcd.print(humidity + String(" %"));
```

Jetzt fehlt nur noch ein Delay. Wähle hier die Zeitspanne, nach der die Messung aktualisiert werden soll – z.B. 10 Sekunden:

```
delay(10000);
```

Und das ist alles, was du für deine Wetterstation benötigst. Angenommen, du befindest dich einem warmen und trockenen Raum, könnte dein Projekt nur wie folgt aussehen:

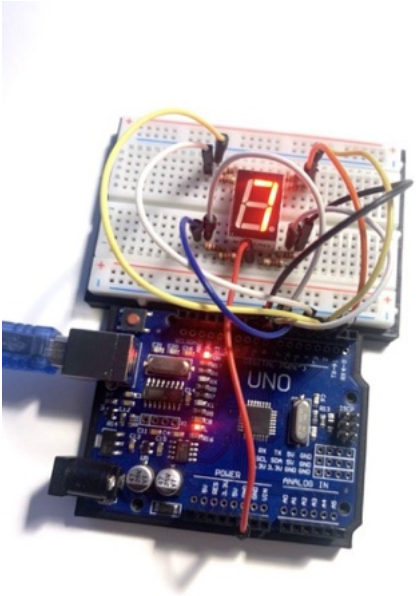



Lade den gesamten Sketch auf deinen Arduino und probiere gleich einmal aus, welche Werte bei dir erscheinen.

Wenn du möchtest, erweitere deine Wetterstation noch um eine Funktion. Wie wäre es, wenn dein Piezo anfängt zu tönen, sobald ein bestimmter Wert über- oder unterschritten wird? Du hast hierfür alle Fähigkeiten bereits gelernt. Viel Spaß beim Experimentieren! Den vollständigen Sketch findest du auf der Webseite zum Buch: [polluxlabs.net/maker-buch](polluxlabs.net/maker-buch)

# 17 DAS 7-SEGMENT DISPLAY

Soll es noch etwas mehr Retro sein? Dann ist das 7-Segment Display genau das Richtige für dich. Dieses Display erinnert unwillkürlich an einen Countdown – der bereits vor Jahrzehnten ablief. Um dieses Display zu verwenden, musst du einige Kabel und Vorwiderstände anbringen, aber die Mühe lohnt sich.

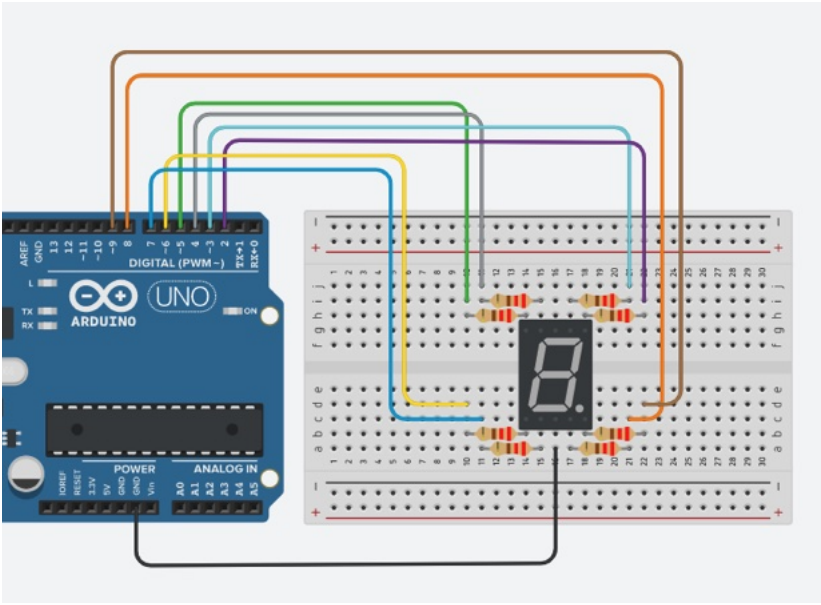



Bevor du loslegst, lass uns zunächst klären, wie solch ein Display eigentlich funktioniert. Wie du leicht nachzählen kannst, ist das Display in sieben Segmente unterteilt, mit denen sich verschiedene Zahlen und Buchstaben darstellen lassen. Jedes Segment wird von einer eigenen LED beleuchtet, die je nach Bedarf entweder aus- oder eingeschaltet wird.

**Um genau zu sein, gibt es auf diesem Display sogar acht Segmente** – das achte ist der Punkt unten rechts, der z.B. dazu dient, eine 6 von einer 9 zu unterscheiden – also anzeigt wo unten ist.

## ANSCHLUSS DES DISPLAYS

Um das 7-Segment Display anzuschließen, benötigst du insgesamt 8 Vorwiderstände à 220Ω und 9 Kabel. Orientiere dich beim Aufbau an folgender Skizze




**Ein wichtiger Hinweis:** Es gibt Displays, die du (neben den Digitalpins) mit GND verbindest – so wie das auf der Skizze oben. Es gibt jedoch auch Displays, die mit 5V verbunden werden müssen. Welches Display du hast, findest du heraus, wenn du später den Sketch auf deinen Arduino lädst. Falls du

dann nur Zeichensalat auf dem Display siehst, stecke die Verbindung um und ändere die betreffende Zeile Code im Sketch. Gleich mehr dazu.

## DER PASSENDE SKETCH

Die Verkabelung war ja schon recht kompliziert und die Steuerung des Displays wäre es auch – gäbe es nicht eine passende Bibliothek. Öffne also zunächst deinen Bibliotheksverwalter und suche dort nach **SevSeg**. Installiere dann die aktuelle Version.


Zu Beginn bindest du die Bibliothek wie gewohnt ein und erstellst das passende Objekt.

```
#include "SevSeg.h"
SevSeg sevseg;
```

In der Setup-Funktion folgen nun einige Definitionen. Z.B. stellst du ein, dass du ein Display mit nur einer Ziffer verwendest und legst die Pins deines Arduinos fest.

```
byte numDigits = 1;
byte digitPins[] = {};

byte segmentPins[] = {3, 2, 8, 7, 6, 4, 5, 9};
bool resistorsOnSegments = true;
```

Außerdem legst du fest, ob es sich um ein Display handelt, das mit GND am Arduino (COMMON_CATHODE) oder mit 5V (COMMON_ANODE) verbunden wird. Je nachdem, welches Display du hast, kommentiere die nicht benötigte Zeile einfach aus. Zuletzt stellst du die Helligkeit auf 100.

```
//sevseg.begin(COMMON_ANODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);
sevseg.begin(COMMON_CATHODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);

sevseg.setBrightness(100);
```

Im Loop erstellst du dir mit einem kleinen For-Loop einen Zähler, der jede Sekunde von Null beginnend eins hochzählt – bis er die 9 erreicht hat.

```
void loop() {
   for(int i = 0; i < 10; i++)
```

```
    {
      sevseg.setNumber(i);
      sevseg.refreshDisplay();
      delay(1000);
    }
}
```

Unten findest du den gesamten Sketch. Lade ihn auf deinen Arduino und schaue, ob alles so funktioniert wie es soll.

Bleibt ein Segment dunkel? Dann prüfe zunächst den richtigen Anschluss und ersetze das Kabel und/oder den Widerstand. Oft liegt hier der Fehler. Falls du nur Zeichensalat siehst, verbinde das Display mit 5V und ändere die entsprechende Zeile im Sketch.

```
#include "SevSeg.h"
SevSeg sevseg;

void setup() {
  byte numDigits = 1;
  byte digitPins[] = {};

  byte segmentPins[] = {3, 2, 8, 7, 6, 4, 5, 9};
  bool resistorsOnSegments = true;

  //sevseg.begin(COMMON_ANODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);
  sevseg.begin(COMMON_CATHODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);
```

```
  sevseg.setBrightness(100);
}

void loop() {
   for(int i = 0; i < 10; i++)
   {
     sevseg.setNumber(i);
     sevseg.refreshDisplay();
     delay(1000);
   }
}
```

# 18 ELEKTRONISCHER WÜRFEL

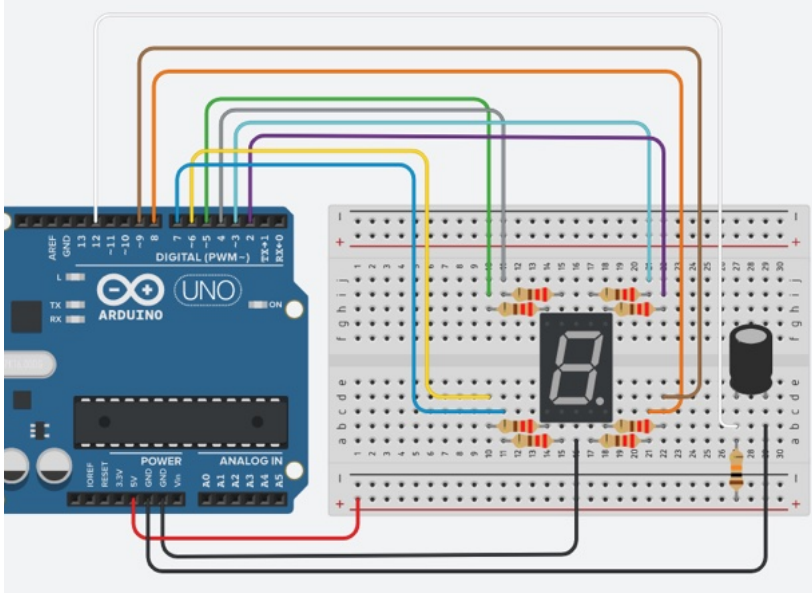
So ein Zähler ist ja ganz nett. Jetzt wird es aber Zeit für eine wirklich praktische Anwendung. In diesem Projekt installierst du neben dem 7-Segment Display einen **Tilt Switch**. In diesem Bauteil befindet sich eine kleine Kugel, die einen Stromkreis unterbricht, wenn der Tilt Switch bewegt wird. Wenn sie wieder zum Stillstand kommt, fließt der Strom wieder weiter.

Diese Unterbrechung kannst du auslesen, um damit wiederum einen Zufallsgenerator eine Zahl zwischen 1 und 6 wählen zu lassen. Das Ergebnis zeigst du dann auf dem Display an.

## DER AUFBAU DES WÜRFELS

Hoffentlich hast du das 7-Segment Display noch auf deinem Breadboard installiert. Wenn ja, baue links oder rechts daneben den Tilt Switch hinzu. Du benötigst neben den Kabeln noch einen 10kΩ Widerstand, den du mit Plus verbindest und der verhindert, dass die Werte des Tilt Switchs willkürlich von Null auf 1 springen.

Orientiere dich beim Aufbau an folgender Skizze.




Hinweis: Wenn dein Tilt Switch drei Pins besitzt, kannst du dir den Widerstand sparen. Verbinde dann den Datenausgang des Switchs einfach mit dem Digitalpin 12 an deinem Arduino.

## DER PASSENDE SKETCH

Das meiste deines Sketchs aus dem letzten Abschnitt kannst du weiterverwenden. Lege zunächst zu Beginn des Sketchs den Pin fest, mit dem der Switch verbunden ist.

```
int tiltPin = 12;
```

Die Loop-Funktion veränderst du jedoch. Statt des Zählers kommt hier eine Abfrage hinein, ob der Tilt Switch bewegt

wurde. Ist das der Fall, nutzt du den Zufallsgenerator, der glücklicherweise Teil der Bibliothek SevSeg ist. Diese Funktion benötigt zwei Parameter: die Mindestzahl (also 1) und die Höchstzahl (also 6). Da diese Funktion diese Spanne jedoch als 1 < 7 versteht, ist der zweite Parameter exklusive, wird also nicht mitgezählt. Deshalb steht hier also eine 7.

Zuletzt folgt noch ein Befehl, der die gefundene Zahl auf dein Display überträgt.

```
void loop() {
   if (digitalRead(tiltPin) == HIGH) {
    sevseg.setNumber(random(1,7));
    sevseg.refreshDisplay();
    }
}
```

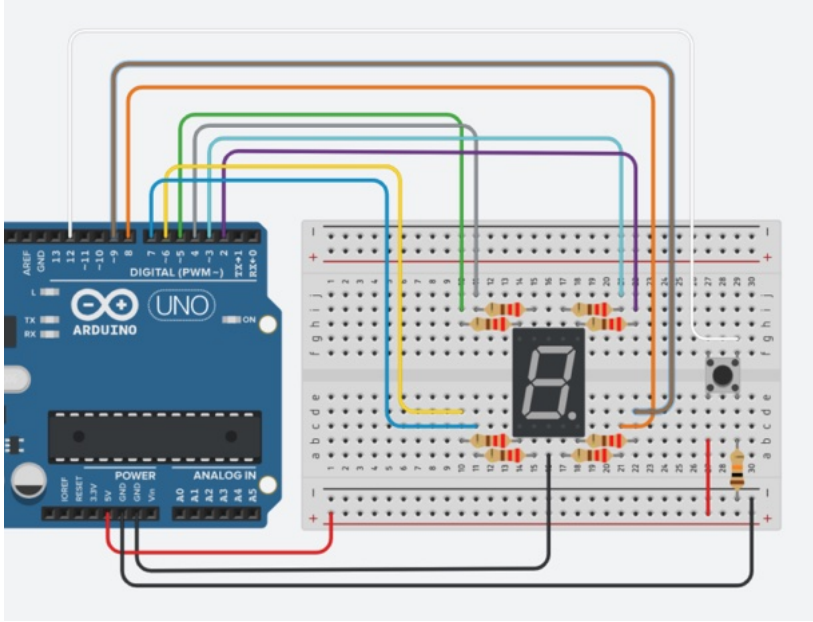
Und das war es auch schon. Lade den folgenden Sketch auf deinen Arduino und schnipse für einen ersten Test gegen den Tilt Switch. Erhältst du Zufallszahlen zwischen 1 und 6 – wie bei einem analogen Würfel?

```
#include "SevSeg.h"
SevSeg sevseg;

int tiltPin = 12;
```

```
void setup() {
  byte numDigits = 1;
  byte digitPins[] = {};

  byte segmentPins[] = {3, 2, 8, 7, 6, 4, 5, 9};
  bool resistorsOnSegments = true;

  //sevseg.begin(COMMON_ANODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);
  sevseg.begin(COMMON_CATHODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);

  sevseg.setBrightness(100);
}

void loop() {
   if (digitalRead(tiltPin) == HIGH) {
    sevseg.setNumber(random(1,7));
    sevseg.refreshDisplay();
    }
}
```

## WÜRFELN MIT EINEM PUSH-BUTTON

Wenn du keinen Tilt Switch verwenden möchtest, sondern lieber einen Push-Button, ist das mit ein paar Handgriffen getan. Entferne zunächst den Tilt Switch von deinem Breadboard und ersetze ihn durch einen Button. Auch die Kabel und den Widerstand musst du etwas versetzen. Orientiere dich an folgender Skizze:




Wie du siehst, sitzt dein Button über der Brücke des Breadboards, damit die zwei oberen Pins von den unteren getrennt sind. Die Unterseite des Buttons schließt du an 5V und über einen 10kΩ Widerstand an GND.

Hierbei handelt es sich um einen sogenannten **Pulldown-Widerstand**. Das bedeutet, du “ziehst” den Wert, den der Button ausgibt dauerhaft auf 0, also LOW. Diesen Wert liest du an der Oberseite des Buttons über das Kabel am Digitalpin 12 aus. **Erst ein Druck auf den Button schaltet das Signal auf HIGH.** Würdest du keinen Pulldown-Widerstand verwenden, würde dieser Wert ständig zwischen LOW und HIGH hin- und herspringen – und wäre für deinen Würfel unbrauchbar.

Es gibt übrigens auch den **Pullup-Widerstand**: Hier "ziehst" du das Signal auf HIGH, indem du den 10kΩ Widerstand mit 5V verbindest. Ein Druck auf den Button schaltet das Signal also auf LOW.

## DER SKETCH

Im Sketch musst du nur ein paar kleine Anpassungen vornehmen. Zunächst deklarierst du zwei Variablen für den Anschluss des Signalpins des Buttons und legst dessen Signal zunächst auf 0 fest.

```
int buttonPin = 12;
int buttonState = 0;
```

In der Setup-Funktion definierst du den **pinMode** des **buttonPin** als **INPUT**:

```
pinMode(buttonPin, INPUT);
```

Nun zum Loop. Hier fragst du per **digitalRead()** das Signal des Buttons ab und speicherst den Wert in der Variablen **buttonState**. Wenn dieser Wert = 1 ist, der Button also gedrückt wurde, startet die Ermittlung der Zufallszahl, die dann auf deinem Display angezeigt wird.

```
void loop() {
  buttonState = digitalRead(buttonPin);

  if (buttonState == HIGH) {
    sevseg.setNumber(random(1, 7));
    sevseg.refreshDisplay();
  }
}
```

Und das war auch schon alles. Hier der vollständige Sketch:

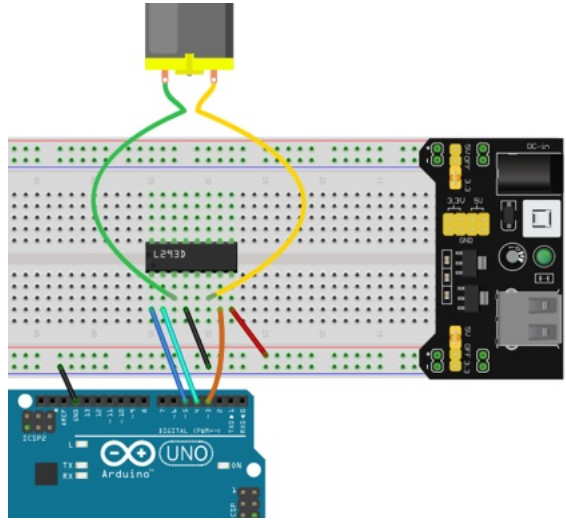
```
#include "SevSeg.h"
SevSeg sevseg;

int buttonPin = 12;
int buttonState = 0;

void setup() {
  byte numDigits = 1;
  byte digitPins[] = {};

  byte segmentPins[] = {3, 2, 8, 7, 6, 4, 5, 9};
  bool resistorsOnSegments = true;

  //sevseg.begin(COMMON_ANODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);
  sevseg.begin(COMMON_CATHODE, numDigits, digitPins,
segmentPins, resistorsOnSegments);

  sevseg.setBrightness(100);

  Serial.begin(9600);
```

```
  pinMode(buttonPin, INPUT);
}

void loop() {
  buttonState = digitalRead(buttonPin);

  if (buttonState == HIGH) {
  sevseg.setNumber(random(1, 7));
    sevseg.refreshDisplay();
  }
}
```

# 19 DER GLEICHSTROM-MOTOR

Bis hierher hat sich eigentlich noch nicht viel bewegt auf deinem Breadboard, oder? Deshalb wird es jetzt für Motoren. Wir beginnen mit dem Gleichstrom-Motor. In diesem Projekt erfährst du, wie du ihn anschließt und verwendest.

Da Gleichstrom-Motoren verhältnismäßig viel Strom benötigen, empfiehlt es sich, **hierfür nicht nur den Arduino zu nutzen, sondern eine externe Stromquelle zu verwenden.** Ebenso besteht die Gefahr, dass vom Motor auch Strom zurück in den Microcontroller fließt, was ihn beschädigen kann. Deshalb kommt hier als externe Stromquelle ein **MB102 Netzteil** für das Breadboard zum Einsatz.



Auf dem Breadboard findest du außerdem ein weiteres Bauteil, das du bisher noch nicht verwendet hast: die **H-Brücke L293D**. Das ist eine sogenannte **integrierte Schaltung** (kurz IC für integrated circuit). Im Inneren des Bauteils befinden sich zahlreiche Schaltkreise und Transistoren, die so manches Projekt vereinfachen. Ohne diese H-Brücke müsstest du den IC auf deiner Steckplatine nachbauen – wofür vermutlich der Platz des ganzen Breadbaords draufgehen würde.

Achte beim Einbau der H-Brücke darauf, dass du sie mittig über die "Brücke" deines Breadboards setzt. Auch die Richtung ist wichtig. Der IC hat auf einer Seite eine Kerbe – diese muss für den obigen Aufbau nach links schauen.

Um den Gleichstrom-Motor betreiben zu können, benötigst du also **zwei Stromquellen:** Eine für den Arduino, also z.B. ein USB-Kabel. Vom Arduino erhält der Motor seine Befehle. Betrieben wird er jedoch vom Breadboard-Aufsatz. Diesen kannst du entweder ebenfalls mit einem USB-Kabel oder einer 9V-Batterie mit Strom versorgen.

## DER PASSENDE SKETCH

Beginnen wir mit einem einfachen Versuch. Du startest den Motor mit voller Geschwindigkeit und lässt ihn laufen. Kopiere dir hierfür den folgenden Sketch und lade ihn auf deinen Arduino.

```
#define ENABLE 5
#define DIRA 3
```

```
#define DIRB 4

void setup() {

  pinMode(ENABLE,OUTPUT);
  pinMode(DIRA,OUTPUT);
  pinMode(DIRB,OUTPUT);
}

void loop() {

  digitalWrite(ENABLE,HIGH);
  digitalWrite(DIRA,HIGH);
  digitalWrite(DIRB,LOW);
}
```

Hier legst du zunächst die drei Pins fest, mit denen dein Arduino über die H-Brücke den Motor steuert. Über Pin 5 (Enable) “startest” du den Motor. Über die Pins 3 und 4 kannst du ihm die Drehrichtung vorgeben.

Im obigen Beispiel beginnt der Motor sich in eine Richtung (DIRA für Direction A) zu drehen – und ändert daran auch nichts. Du setzt **DIRA** auf HIGH und entsprechend die andere Richtung **DIRB** auf LOW.

## RICHTUNGSWECHSEL

Du kannst den Motor aber auch in die andere Richtung drehen lassen. Passe den Loop z.B. wie folgt an.

```
void loop() {
  digitalWrite(ENABLE, HIGH);
  digitalWrite(DIRA, HIGH);
  digitalWrite(DIRB, LOW);
  delay(500);
  digitalWrite(DIRA, LOW);
  digitalWrite(DIRB, HIGH);
  delay(500);
}
```

Nun wechselt der Motor alle halbe Sekunde die Richtung – zunächst dreht er in Richtung A und nach einer kurzen Pause in Richtung B.

## VERSCHIEDENE GESCHWINDIGKEITEN

Du kannst nicht nur die Richtung wechseln, sondern auch die Geschwindigkeit. Tausche den Loop in deinem Sketch für einen kleinen Test durch folgenden aus.

```
void loop() {
  analogWrite(ENABLE, 255);
  digitalWrite(DIRA, HIGH);
  digitalWrite(DIRB, LOW);
  delay(2000);
  analogWrite(ENABLE, 180);
  delay(2000);
  analogWrite(ENABLE, 128);
  delay(2000);
  analogWrite(ENABLE, 50);
  delay(2000);
```

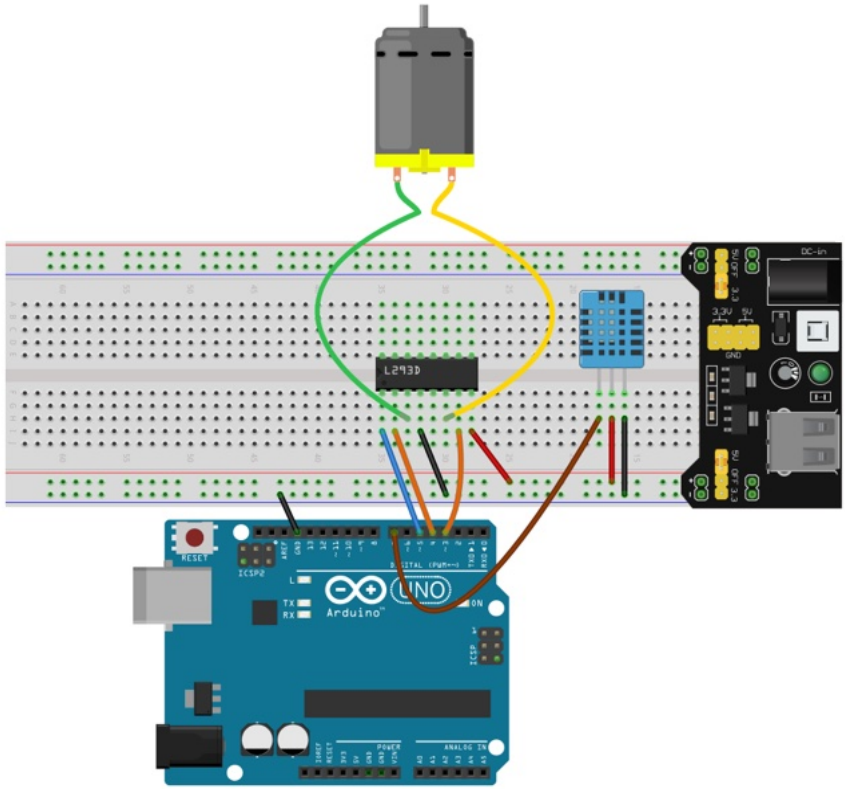
```
  analogWrite(ENABLE, 128);
  delay(2000);
  analogWrite(ENABLE, 180);
  delay(2000);
  analogWrite(ENABLE, 255);
  delay(2000);
  digitalWrite(ENABLE, LOW);
  delay(2000);
}
```

Hier “startest” du den Motor diesmal mit der Funktion **analogWrite()** und gibst ihr als Parameter die Geschwindigkeit mit. Die 255 ist hierbei die Höchstgeschwindigkeit, die nun per Pulsweitenmodulation vorgegeben wird. Im Zwei-Sekundentakt fährst du die Geschwindigkeit nun in mehreren Schritten herunter, um sie dann langsam wieder zu erhöhen. Am besten kannst du die Geschwindigkeiten erkennen, wenn du einen kleinen Rotor oder Ventilator auf den Motor setzt.

Apropos Ventilator. Als nächstes baust du dir mit dem Gleichstrom-Motor einen Ventilator, der anspringt, wenn es in deinem Zimmer zu warm wird.

# 20 TEMPERATURGESTEUERTER VENTILATOR

Wie wäre es mit einem Ventilator, den du nicht selbst einschaltest, wenn es zu warm in deinem Zimmer wird? In diesem Projekt übernimmt das der Arduino für dich. Zunächst zum Aufbau. Vielleicht hast du den Motor samt H-Brücke ja noch auf deinem Breadboard installiert. Baue nun einfach den DHT11 wie folgt daneben.



## DER PASSENDE SKETCH

Im Prinzip passiert im folgenden Sketch nichts, das du nicht schon kennst. Du bindest die Bibliothek für den DHT11 ein, definierst die Pins zur Kommunikation zwischen Sensor, Motor und Arduino und deklarierst eine Variable für die Temperatur.

```
#include "DHT.h"

#define DHTPIN 7
#define DHTTYPE DHT11

#define ENABLE 5
#define DIRA 3
#define DIRB 4

float temp;
DHT dht(DHTPIN, DHTTYPE);
```

Im Setup legst du die pinModes fest und startest den Seriellen Monitor sowie den DHT11.

```
void setup() {

  pinMode(ENABLE, OUTPUT);
  pinMode(DIRA, OUTPUT);
  pinMode(DIRB, OUTPUT);

  Serial.begin(9600);
  dht.begin();
}
```

Im Loop misst du im Sekundentakt die Temperatur und gibst das Ergebnis im Seriellen Monitor aus. Mit einer bedingten Anweisung prüfst du, ob die Temperatur über einem bestimmten Wert (hier 22) liegt und startest in diesem Fall den Gleichstrom-Motor. Liegt die Temperatur darunter, bleibt der Motor aus.

```
void loop() {

  temp = dht.readTemperature();
  Serial.print("Temperatur: ");
  Serial.print(temp);
  Serial.println(" *C");

  if (temp > 22) {
    digitalWrite(ENABLE, HIGH);
    digitalWrite(DIRA, HIGH);
    digitalWrite(DIRB, LOW);
  }

  else {
    digitalWrite(ENABLE, LOW);
  }

  delay(1000);
}
```

Und das war auch schon alles an Code. Du findest den vollständigen Sketch auf **polluxlabs.net/maker-buch**. Experimentiere nach dem Upload etwas mit dem Schwellenwert herum und freue dich auf eine frische Brise.

# 21 DER SERVO-MOTOR

Früher oder später kommt jeder Maker an den Punkt, an dem er ein Projekt mit einem Servo-Motor bauen möchte – z.B. für einen Zeiger oder für einen Roboterarm. Jetzt lernst du, wie du einen Servo am Arduino anschließt und steuerst.

## WAS IST EIN SERVO?

Zunächst ein paar Worte darüber, was ein Servo eigentlich ist: Hierbei handelt es sich um einen Motor, der nicht einfach nur anfängt zu laufen, sondern den du auf eine bestimmte Position drehen kannst. **Das heißt, du kannst ihm einen Winkel von 0° bis 180° zuweisen** und damit z.B. einen am Servo montierten Zeiger im Halbkreis drehen lassen.

## DIE BIBLIOTHEK SERVO.H

Damit die Steuerung für dich so einfach wie möglich ist, gibt es die Bibliothek **Servo.h** – die in der Regel bereits vorinstalliert ist. Mit ihr kannst du bis zu 12 Motoren an einem Arduino verwenden. Binde die Bibliothek am Anfang deines Sketchs wie folgt ein:

```
#include <Servo.h>
```

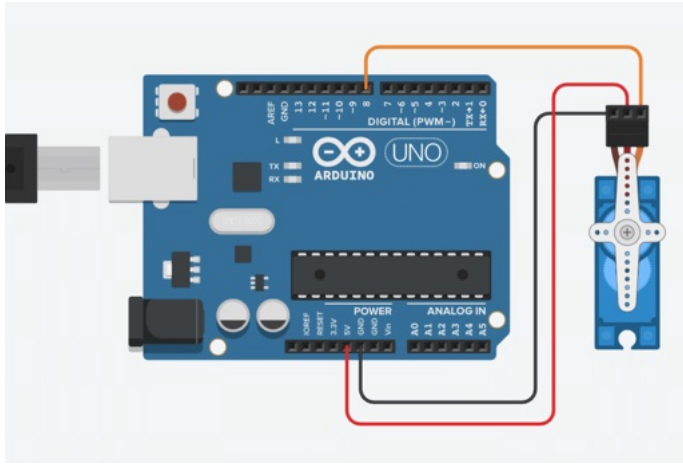
Zwei Funktionen der Bibliothek benötigst du immer wieder, wenn du mit Servos arbeitest:

- **attach(pin)** – Hier legst du einmalig den Pin fest, an dem dein Servo-Motor angeschlossen ist
- **write(angle)** – Damit "fährst" du den Servo an den gewünschten Winkel (angle)

Später schauen wir uns diese Funktionen genauer an.

## DIE MINIMALSCHALTUNG

Für die allerersten Gehversuche reicht es, wenn du deinen Servo wie folgt mit deinem Arduino verbindest. Für das Signal (Orange) kannst du natürlich einen Digitalpin deiner Wahl verwenden. In den folgenden Sketches verwenden wir jedoch immer Pin 8.



**Hinweis:** Jeder Servo hat drei Kabel: Strom (5V), Masse (GND) und Signal. Diese haben jedoch je nach Modell eine unterschiedliche Reihenfolge. Vergewissere dich immer, dass du alles richtig angeschlossen hast, bevor du ihn mit Strom versorgst.

## DER ERSTE SKETCH

Auch hier halten wir alles so einfach wie möglich. Kopiere dir den folgenden Sketch und lade ihn auf deinen Arduino.

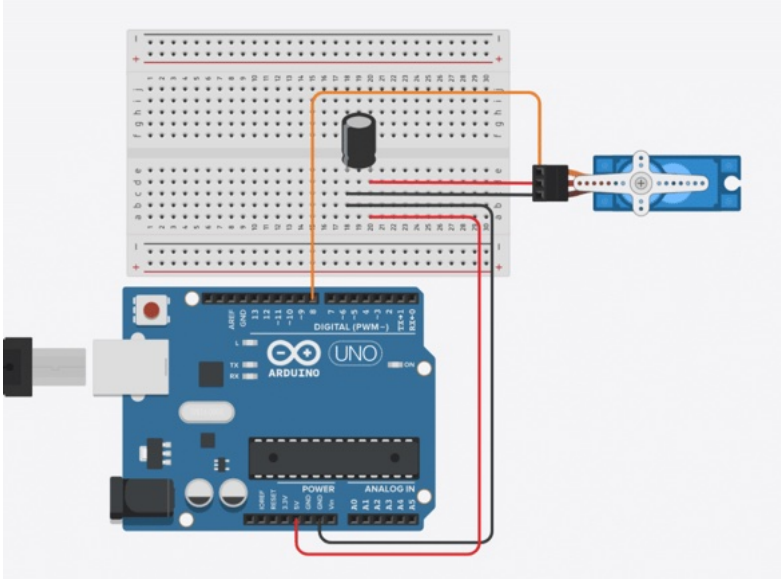
```
#include <Servo.h>

Servo myServo;

void setup() {
  myServo.attach(8);
}

void loop() {
  myServo.write(0);
  delay(2000);
  myServo.write(90);
  delay(2000);
  myServo.write(180);
  delay(2000);
  myServo.write(90);
  delay(2000);
}
```

Ganz oben im Sketch bindest du die Bibliothek **Servo.h** ein. Danach erstellst du ein Objekt mit dem Namen **myServo**.

Im Setup verknüpfst du mit Hilfe der Funktion **attach()** den Digitalpin mit diesem Objekt. Im Loop steuerst du mit deinem Motor nun verschiedene Winkel mit der Funktion **write()** an. Du beginnst bei 0° und wartest zwei Sekunden. Danach drehst du weiter auf 90°, dann auf 180°, zurück auf 90° – und wieder von vorne.

## EINEN KONDENSATOR VERWENDEN

Ein Servo benötigt recht viel Strom, vor allem zu Beginn einer Bewegung. Das kann zu einem Spannungsabfall führen, der zu ungewünschten Ergebnissen bis hin zu einem Reset des Arduinos führen kann. Um dem vorzubeugen, kannst du einen Kondensator in die Stromversorgung des Motors einbauen, wie du hier sehen kannst:




Der Kondensator mit einer Kapazität von mindestens 470µF speichert Energie, die der Servo neben jener vom Arduino nutzen kann. Damit kannst du den Spannungsabfall etwas verringern.

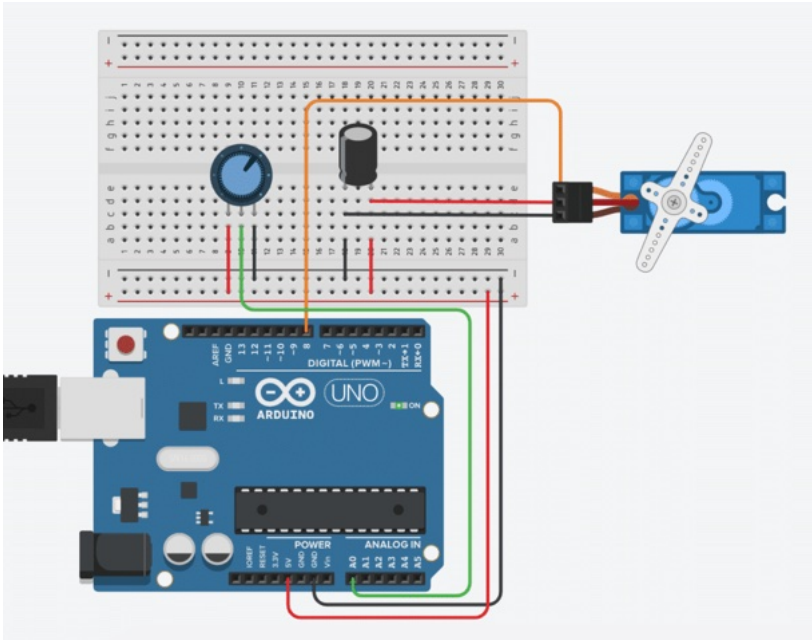
**Hinweis:** Wenn du einen Elektrolytkondensator (Elko) verwendest, achte unbedingt auf die richtige Polarität: Die Seite mit dem hellen Strich (s. Screenshot oben) muss an Erde (GND). **Wenn du die Seiten vertauschst, kann der Kondensator explodieren!**

Wenn auch ein Kondensator nicht hilft, kannst du deinen Servo an seinen eigenen Stromkreis anschließen, der vom Arduino (und weiteren dort angeschlossenen Teilen) unabhängig ist.

# DEN SERVO MIT EINEM POTI STEUERN

Jetzt gehen wir noch einen Schritt weiter: Du steuerst deinen Motor mit einem Poti. Ziel ist es hierbei, dass er sich analog zu den Bewegungen des Potentiometers dreht.

Füge also deinem Aufbau ein Poti wie folgt hinzu.




Kopiere dir anschließend den folgenden Sketch und lade ihn auf deinen Arduino:

```
#include <Servo.h>
```

```
int servoPosition;
int potiValue;
Servo myServo;

void setup(){
  pinMode(0, INPUT);
  myServo.attach(8);
}
void loop(){
  delay(10);
  potiValue = analogRead(0);
  servoPosition = map(potiValue, 0, 1024, 0, 180);
  myServo.write(servoPosition);
}
```

Neu in diesem Sketch sind zwei Variablen. In **servoPosition** speicherst du den Winkel, den der Servo ansteuern soll. In **potiValue** steht der aktuelle Wert des Potis, den du so ausliest und speicherst:

```
potiValue = analogRead(0);
```

Jetzt wird es interessant: Den Winkel des Servos bestimmst du mit der Funktion **map()**. Hier "mappst" du den Wert, den dein Poti ausgibt (potiValue) auf den entsprechenden Winkel des Servos (servoPosition). Hierfür benötigst du sowohl für den Poti als auch für den Motor den Minimal- und Maximalwert. **Für den Poti sind das 0 und 1024 – für den Servo 0 und 180 (Grad).**

```
servoPosition = map(potiValue, 0, 1024, 0, 180);
```

Zuletzt bewegst du den Motor auf die errechnete Position:

```
myServo.write(servoPosition);
```

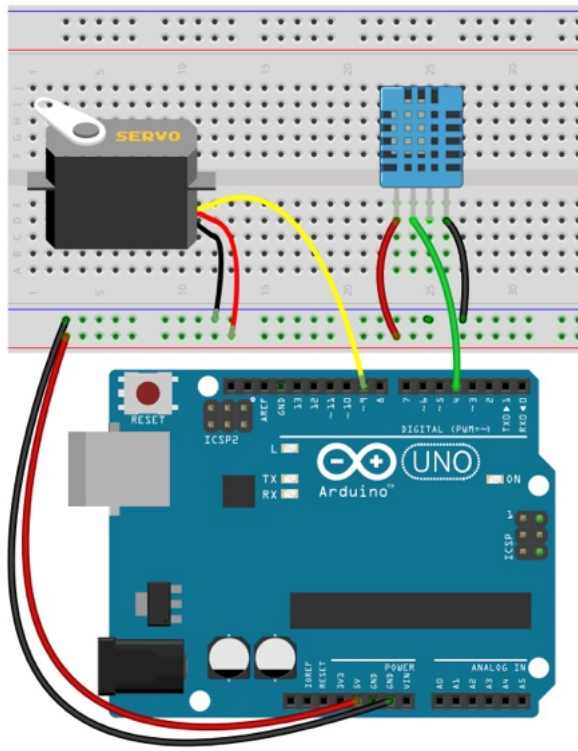
Und das soll es erst einmal gewesen sein. Du weißt jetzt, wie du einen Servo-Motor am Arduino anschließt und ihn auf verschiedene Arten und Weisen steuern kannst. Als nächstes baust du dir ein Analaog-Thermometer.

# 22 EIN ANALOG-THERMOMETER

Digitale Thermometer mit Display sind schön und gut – etwas mehr Retro-Charme erhältst du jedoch mit einem analogen. Außerdem lernst du hierbei deinen Servo-Motor besser kennen. Jetzt baust du dir ein Thermometer mit einem Zeiger, der dir auf einer Skala die Temperatur anzeigt.

## SERVO UND DHT11 AUFBAUEN

Vielleicht hast du deinen Servo ja noch auf deinem Breadboard installiert. Beachte bitte, dass das Kabel für das Signal in diesem Projekt im Digitalpin 8 steckt. Baue nun deinen Temperatursensor DHT11 neben den Sensor. In der untenstehenden Skizze haben wir den Sensor mit vier Pins verwendet – wenn du einen DHT11 mit drei Pins hast, installiere ihn entsprechend. Schließe das Signalkabel jedoch auch in diesem Fall am Digitalpin 4 an.




## DER SKETCH FÜR DEIN THERMOMETER

Den oberen Bereich kennst du schon aus dem kurzen Test und aus anderen Projekten, die den Sensor DHT11 verwendet haben:

```
#include <Servo.h>
#include "DHT.h"
Servo myServo;

#define DHTPIN 4
```

```
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);
```

Es folgen zwei Variablen, die du für die Temperatur und die passende Position des Zeigers auf dem Thermometer benötigst. Die Variable **temp** ist vom Typ **float**, da dein DHT11 die Temperatur auf das Zehntel genau, also als Kommazahl, misst. Für den Winkel reicht hingegen ein **int**.

```
float temp;
int winkel;
```

Auch das Setup des Sketchs kennst du bereits:

```
void setup() {
  Serial.begin(9600);
  dht.begin();
  myServo.attach(9);
  myServo.write(180);
}
```

Interessant wird es nun im Loop. Hier misst du zunächst die Temperatur und zeigst sie im Seriellen Monitor an.

```
temp = dht.readTemperature();
Serial.println(temp);
```

Anschließend verwendest du die Funktion **map()**, um die gemessene Temperatur auf einen Winkel zwischen 0° und 180° umzulegen und in die Variable **winkel** zu schreiben.

```
winkel = map(temp, 10, 30, 180, 0);
```

Im obigen Beispiel findest du einen Temperaturbereich von 10°C bis 30°C. Du kannst ihn aber natürlich anpassen.

Anschließend übergibst du den gefundenen Winkel an deinen Servo. Zuletzt folgt noch ein Delay von zwei Sekunden, um die Messungen zu begrenzen. Auch hier bist du natürlich frei bei deiner Wahl.

```
Serial.println(winkel);
myServo.write(winkel);
delay(2000);
```

Hier nun der vollständige Sketch:

```
#include <Servo.h>
#include "DHT.h"
```

```
Servo myServo;

#define DHTPIN 4
#define DHTTYPE DHT11
DHT dht(DHTPIN, DHTTYPE);

float temp;
int winkel;

void setup() {
  Serial.begin(9600);
  dht.begin();
  myServo.attach(9);
  myServo.write(180);
}

void loop() {
  temp = dht.readTemperature();
  Serial.println(temp);

  winkel = map(temp, 10, 30, 180, 0);

  Serial.println(winkel);
  myServo.write(winkel);
  delay(2000);
}
```

## BAUE DIR EINE SKALA

Im Prinzip ist dein Analog-Thermometer jetzt einsatzbereit. Ohne entsprechende Skala wird es allerdings schwierig herauszufinden, was dein Servo-Motor dir sagen will.

Deshalb benötigst du eine Schablone, die du am Servo anbringst. In der Mitte benötigst du ein Rechteck, um deinen Servo-Motor hindurch zu stecken.

Wenn du deine eigene Skala bastelst, achte darauf, dass der niedrigste Wert ganz links unten im gedachten Halbkreis des Servos liegt. Und der höchste Wert entsprechend rechts unten.

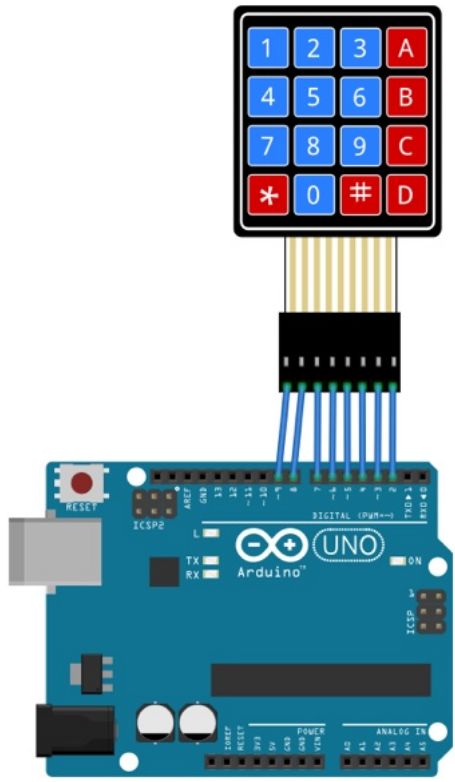
# 23 EIN CODE-SCHLOSS MIT TASTATUR UND SERVO-MOTOR

In diesem Projekt baust du dir ein Code-Schloss mit einer Folientastatur und einem Servo-Motor. Nur per Eingabe der richtigen vierstelligen Zahlenkombination wird sich der Servo-Motor in Bewegung setzen und damit eine mögliche Verriegelung lösen. Los geht's.

## DIE FOLIENTASTATUR ANSCHLIESSEN

Für ein Code-Schloss braucht es natürlich ein Gerät, über das man den Code eingeben kann. Hierfür eignet sich die 4×4 Folientastatur perfekt.

Zunächst installierst du die Verbindung zum Arduino. Die Tastatur besitzt 8 Buchsen. Stecke hier ebenso viele Kabel hinein und verbinde diese der Reihe nach mit den Digitalpins 9 bis 2 an deinem Arduino:




## DIE PASSENDE BIBLIOTHEK

Um dir die Steuerung der Tastatur so einfach wie möglich zu machen, gibt es eine Bibliothek. Diesmal nutzt du hierfür jedoch nicht den Bibliotheksverwalter, sondern bindest die Bibliothek manuell ein. Lade sie dir zunächst auf der Webseite zum Buch herunter: [polluxlabs.net/maker-buch](polluxlabs.net/maker-buch)

Wähle nun im Menü der Arduino IDE den Punkt **Sketch -> Bibliothek einbinden -> .ZIP Bibliothek hinzufügen**

Für einen ersten Test der Tastatur verwendest du nun den Beispiel-Sketch, der in der Bibliothek mitgeliefert wird. Öffne diesen unter **Datei -> Beispiele -> Keypad -> CustomKeypad**

Bevor wir einen Blick auf den Code werfen, lade den Sketch zunächst auf deinen Arduino. Öffne nun den Seriellen Monitor und drücke eine der Tasten auf der Tastatur. Erscheint das entsprechende Zeichen in der Ausgabe? Wenn ja, perfekt.

## DER BEISPIEL-SKETCH

Lass uns einen ganz kurzen Blick auf diesen Sketch werfen. Der meiste Code sind Funktionen der Bibliothek Keypad, aber zwei Dinge sind besonders interessant. Zunächst siehst du im Code die Anzahl der Tasten je Reihe und Spalte sowie eine Matrix, die die Werte bzw. Zeichen der Tastatur bestimmt:

```
const byte ROWS = 4;
const byte COLS = 4;



char hexaKeys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
```

Das sind die Zeichen, die du auch auf der Tastatur selbst findest. Solltest du einmal andere Zeichen verarbeiten wollen, könntest du diese hier eintragen und damit einer Taste auf dem Bauteil zuweisen.

Der Anschluss der Tastatur am Arduino verbirgt sich in diesen beiden Variablen:

```
byte rowPins[ROWS] = {9, 8, 7, 6};
byte colPins[COLS] = {5, 4, 3, 2};
```

Das sind die Pins, die du auch im Anschlussplan oben findest. Wenn du einmal einen dieser Pins unbedingt für ein anderes Bauteil verwenden musst, kannst du hier der Tastatur einen abweichenden Digitalpin zuweisen.

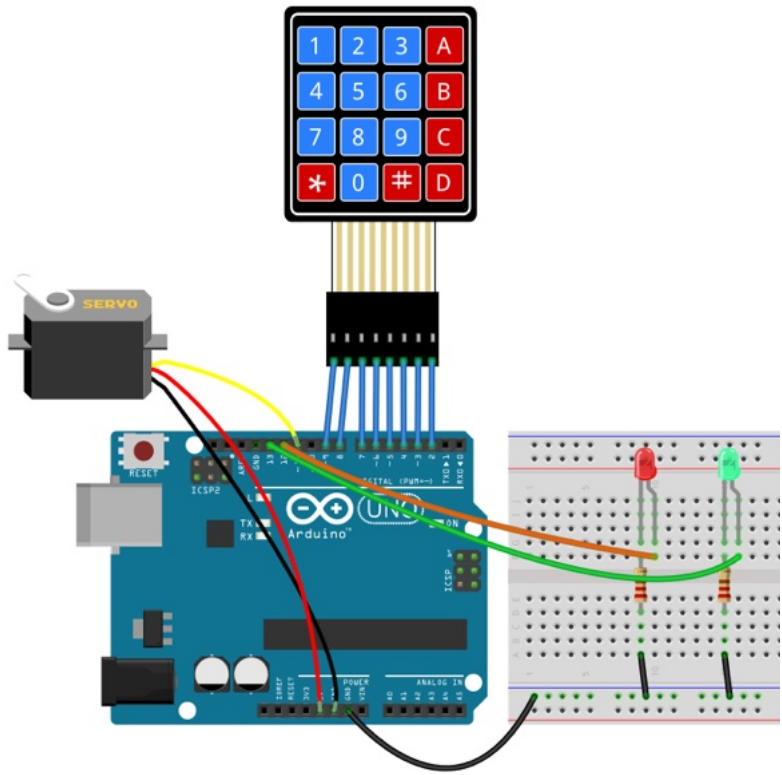
Außerdem wird mit diesen Parametern das Objekt customKeypad erzeugt:

```
Keypad customKeypad = Keypad( makeKeymap(hexaKeys),
rowPins, colPins, ROWS, COLS);
```

Und weiter geht's – als nächstes installierst du zwei LEDs sowie den Servo-Motor und sorgst dafür, dass dieser sich nur nach der Eingabe des richtigen Codes bewegt.

# DAS SCHLOSS AUFBAUEN UND PROGRAMMIEREN

Verbinde als erstes deinen Servo-Motor mit dem Arduino. Du kannst ihn entweder auf deinem Breadboard installieren und von dort aus verbinden – oder du steckst die Kabel des Servos direkt in die Pins GND, 5V und 11, so wie auf dieser Skizze.



Einen Servo-Motor hast du ja bereits in einem Projekt angeschlossen. Trotzdem noch einmal der Hinweis, dass die Reihenfolge der Anschlüsse GND, VCC und Signal sich je nach Fabrikat unterscheiden können. Bitte achte hierauf beim Anschluss.

Hinzu kommen noch zwei LEDs samt zwei Vorwiderständen mit je 220Ω. Schließe diese an die Digitalpin 12 und 13 an. Mit diesen zeigst du später an, ob der Code richtig oder falsch eingegeben wurde.

## DER SKETCH FÜR DEIN CODE-SCHLOSS

Damit kommen wir schon zum Kern des Projekts. Der Servo soll sich nur bewegen, wenn jemand den richtigen vierstelligen Code per Tastatur eingibt. Dann dreht sich der Zeiger zur Seite und gibt z.B. den Deckel einer Box frei. Wie du das Gehäuse der Box gestaltest, ist natürlich dir überlassen.

Zu Beginn des Sketchs bindest du wie gewohnt zunächst die benötigten Bibliotheken ein:

```
#include <Keypad.h>
#include <Servo.h>
```

Anschließend folgen die Informationen für das Keypad, die du schon kennst:

```
const byte ROWS = 4;
const byte COLS = 4;



char hexaKeys[ROWS][COLS] = {
  {'1', '2', '3', 'A'},
  {'4', '5', '6', 'B'},
  {'7', '8', '9', 'C'},
  {'*', '0', '#', 'D'}
};

byte rowPins[ROWS] = {9, 8, 7, 6};
byte colPins[COLS] = {5, 4, 3, 2};
Keypad customKeypad = Keypad( makeKeymap(hexaKeys),
rowPins, colPins, ROWS, COLS);
```

Nun benötigst du drei Variablen bzw. Konstanten, die die gedrückte Taste **key**, den gesamten eigegebenen Code **inputCode** sowie das von dir hinterlegte Passwort **code** beinhalten. Gegen letzteres wirst du später die Eingabe prüfen. Wenn du später deinen eigenen Code hinterlegen möchtest, kannst du das hier tun.

```
char key;
String inputCode;
const String code = "1103A"; //Der von dir festgelegte
Code
```

Zuletzt fehlen noch ein Objekt für den Servo-Motor und die beiden Variablen für die Anschlüsse der LEDs.

```
Servo servo;
int redLED = 12;
int greenLED = 13;
```

## DIE SETUP-FUNKTION

Hier gibt es sicherlich nichts Neues für dich. Du startest den Seriellen Monitor, legst die **pinMode** für die LEDs fest und den Anschluss-Pin des Servo-Motors.

```
void setup() {

  Serial.begin(9600);
  pinMode(redLED, OUTPUT);
  pinMode(greenLED, OUTPUT);
  servo.attach(11);
}
```

## DER LOOP

Als erstes benötigst du eine Variable, in der du den Wert der Taste speicherst, die gerade gedrückt wurde. Das ist hier die Variable **key** mit dem Typ **char** für Character. Den Wert speicherst du mithilfe der Funktion **customKeypad.getKey()**.

```
char key = customKeypad.getKey();
```

Nun folgt eine Reihe von bedingten Anweisungen. Das Code-Schloss verwendet zwei Tasten, um das Schloss zu verriegeln (mit der Taste *) oder die Eingabe zu prüfen (mit #). Diese Tasten darfst du also nicht in deinem hinterlegten Code verwenden. Die erste If-Abfrage “horcht” auf die Taste * und verriegelt das Schloss, indem sie den Servo auf einen Winkel von 90° stellt. Natürlich kannst du diesen Winkel nach deinen Bedürfnissen ändern.

Zusätzlich schaltet sie die rote LED an (und die grüne aus) und leert die Variable **inputCode**, damit hier für die nächste Eingabe keine “alten” Zeichen mehr zu finden sind.

```
if (key == '*') {
  inputCode = "";
  Serial.println("Schloss verriegelt.");
  delay(1000);
  servo.write(90);
  digitalWrite(greenLED, LOW);
  digitalWrite(redLED, HIGH);
```

Die zweite Abfrage folgt gleich darauf mit einem **else if** und wird angesteuert, wenn die Taste # gedrückt wurde. In diesem Fall folgen wiederum zwei weitere Abfragen – eine, die ausgeführt wird, wenn der Code korrekt eingegeben wurde und die andere, falls dem nicht so ist.

```
} else if (key == '#') {
  if (inputCode == code) {
    Serial.println("Der Code ist korrekt. Öffne das
Schloss...");
    digitalWrite(greenLED, HIGH);
    digitalWrite(redLED, LOW);
    servo.write(0);

  } else {
    Serial.println("Der Code ist falsch!");
    digitalWrite(greenLED, LOW);
    digitalWrite(redLED, HIGH);
  }
```

Falls der Code stimmt, also **inputCode == code** zutrifft, leuchtet die grüne LED und der Servo steuert auf 0°. Bei einer inkorrekten Eingabe, leuchtet entsprechend die rote LED – und nichts bewegt sich. In beiden Fällen wird die Variable für den eingegebenen Code wieder geleert:

```
inputCode = "";
```

Jetzt hast du alles, um das Schloss auf Tastendruck zu verriegeln. Außerdem um zu prüfen, ob der Code richtig oder falsch ist und entsprechend zu reagieren. Aber wie wird die Variable **inputCode** eigentlich befüllt?

Das geschieht in einem letzten **else** – die Anweisung, die bei allen Tasten außer * und # ausgeführt wird. Hier fügst du der Variablen **inputCode** einfach das zuletzt gedrückte Zeichen hinzu:

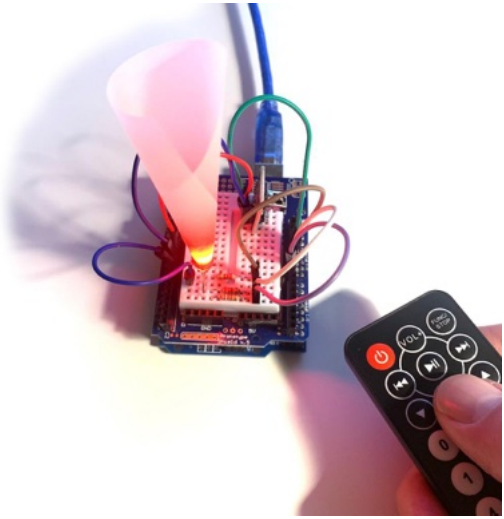
```
} else {
  inputCode += key;
  Serial.println(inputCode);
}
```

Das war es schon! Du findest den vollständigen Sketch auf **polluxlabs.net/maker-buch**

# 24 STIMMUNGSLICHT PER FERNBEDIENUNG

In diesem Projekt baust du dir mit einer **RGB-LED** und einer **Infrarot-Fernbedienung** ein Licht, dessen Farbe du auf Knopfdruck verändern kannst. Die RGB-LED hast du ja bereits kennengelernt – neu ist der IR-Empfänger und die Fernbedienung. Außerdem lernst du eine weitere praktische C++ Funktion kennen: **switch … case.**

Wenn du eine Vielzahl von Fällen hast, die du unterscheiden möchtest – wie z.B. die Tasten einer Fernbedienungen – ist das Programmieren mit einer Kette von bedingten Anweisungen (if … else if … else) recht mühsam. Hier macht dir **switch … case** das Leben leichter.



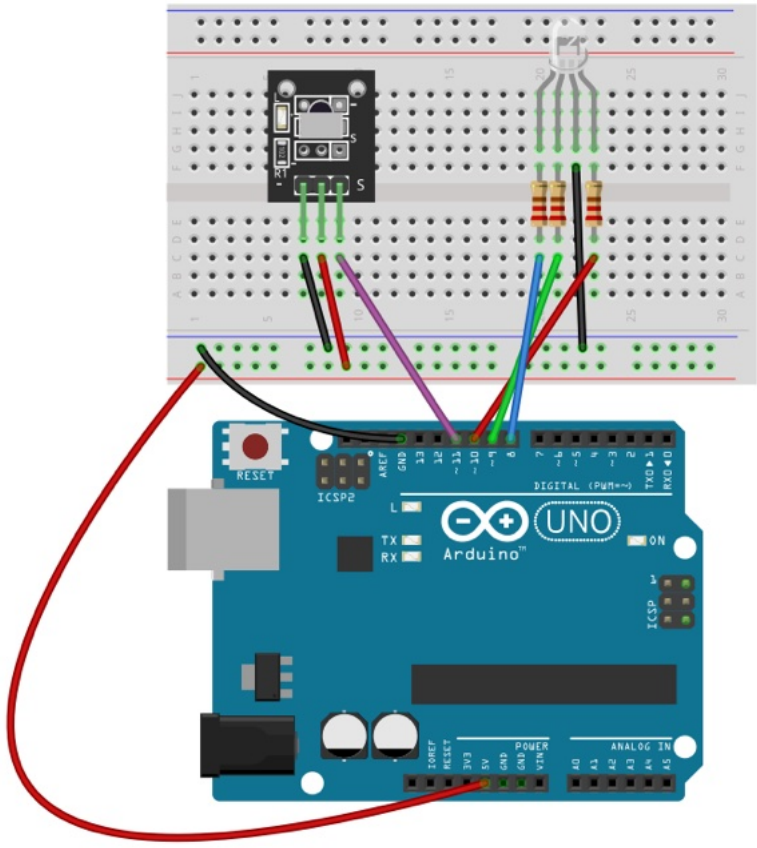
## DEN IR-SENSOR UND DIE RGB-LED ANSCHLIESSEN

Die RGB-LED hast du bereits in einem früheren Projekt angeschlossen – hier gibt es für dich sicherlich keine Überraschungen mehr. Du benötigst für jeden Farbkanal einen Vorwiderstand à 220Ω, den du zwischen der Anode der LED und dem Digitalpin, mit dem du sie ansteuerst, setzt. Das lange Beinchen ist die Kathode – diese verbindest du mit GND.

Möglicherweise passen die unten gezeigten Farbkanäle und Arduino-Pins je nach Bauart der LED nicht zusammen. Im Beispiel unten ist der rote Farbkanal rechts von der Kathode, was bei deiner LED jedoch anders sein kann. Das kannst du jedoch leicht herausfinden und beheben, wenn du später die Farbe des Lichts umschaltest: Passe einfach die Variablen im Sketch an, indem du hinter den Farbkanal den richtigen Pin einträgst:

```
int ledRed = 10;
int ledGreen = 9;
int ledBlue = 8;
```

Der Infrarot-Sensor ist ebenso schnell angeschlossen. Zwei der drei Pins dienen der Stromversorgung. Den dritten verbindest du mit dem Digitalpin 11 an deinem Arduino. Und das war es schon. Orientiere dich beim Aufbau an folgender Skizze.




## SWITCH … CASE – WELCHE TASTE WURDE GEDRÜCKT?

Bevor du mit dem Sketch für das Stimmungslicht loslegst, lernst du noch ein weiteres Feature kennen, das dir dabei hilft,

eine lange Kette von bedingten Anweisungen effizienter und übersichtlicher zu gestalten: **switch … case.**

**Grundsätzlich haben die If-Abfragen und switch … case das gleiche Ziel:** Sie fragen einen Zustand oder eine Bedingung ab und führen einen individuellen Code aus. Im Fall von switch … case fragst du den Inhalt einer Variablen ab und führst je nachdem, was in der Variablen steckt, einen entsprechenden Code aus. Der Aufbau sieht hierbei wie folgt aus:

```
switch (var) {
  case 1:
    // Mache dies, wenn "var" 1 ist.
    break;
  case 2:
    // Mache dies, wenn "var" 2 ist.
    break;
  default:
    // Mache dies im Default-Fall
    // Dieser Fall ist optional
    break;
}
```

Zunächst leitest du mit **switch (var) {** ein. In Klammern steht hierbei die Variable, auf deren Inhalt du gleich danach mit **case** reagierst. Steckt in der Variablen also z.B. der Wert 1, wird nur der Code ausgeführt, der hinter **case 1:** steht. Nach dem Code, der in diesem Fall ausgeführt werden soll, folgt der Befehl **break**, mit dem switch … case wieder verlassen wird und das Programm weiter ausgeführt wird.

```
case 1:
    Serial.println("Der Wert ist 1");
    break;
```

### DER DEFAULT-FALL

Du kennst ja von bedingten Anweisungen mit **if** die Anweisung **else**: Ein Sammelbecken für alle Fälle, die du nicht eigens abgefangen hast. So funktioniert auch **default:**

Hier kannst du Code unterbringen, der ausgeführt werden soll, wenn kein passender **case** gefunden wurde. Das ist jedoch optional – wenn im Beispiel oben also der Wert 3 in der Variablen **var** stecken würde, könnte dein Sketch keinen entsprechenden **case** finden und **switch … case** einfach verlassen und mit dem Programm weitermachen.

Als nächstes gehst du in die Praxis über. Du fragst per **switch … case** die Tasten deiner Fernbedienung ab und schaltest je nachdem welche Taste gedrückt wurde, die Lichtfarbe um.

## DAS STIMMUNGSLICHT PROGRAMMIEREN

Kommen wir also zur Praxis. Im folgenden Sketch bringst du die RGB-LED und die Fernbedienung samt IR-Empfänger zusammen. Am Ende wird ein Tastendruck von dir per **switch … case** identifiziert und das Licht der LED entsprechend umgeschaltet.

Beginnen musst du jedoch mit zwei Bibliotheken. Diese sind nicht im Bibliotheksverwalter zu finden, du kannst sie jedoch auf der Webseite zum Buch herunterladen: polluxlabs.net/maker-buch

Du findest neben dem Sketch die Datei **IR.h** – lass diese einfach an Ort und Stelle liegen, damit sie vom Sketch gefunden werden kann.

Öffne nun als nächstes den Sketch in der Arduino IDE und binde die Bibliothek **IRremote.zip** über **Sketch -> Bibliothek einbinden -> .ZIP-Bibliothek hinzufügen** ein. Und das war es auch schon. Zu Beginn des Sketch integrierst du beide Bibliotheken wie gewohnt:

```
#include "IRremote.h"
#include "IR.h"
```

Als nächstes erstellst du mithilfe der Bibliotheken zwei Objekte, die du für Empfang und Verarbeitung deiner Befehle benötigst. Ebenso deklarierst du die Variablen für die Anschlüsse der RGB-LED.

```
IRrecv irrecv(RECEIVER);
decode_results results;

int ledRed = 10;
```

```
int ledGreen = 9;
int ledBlue = 8;
```

## DIE SETUP-FUNKTION

Hier startest du den Seriellen Monitor, aktivierst den Infrarot-Empfänger und legst den jeweiligen **pinMode** für die Anschlüsse der RGB-LED fest.

```
void setup() {

  Serial.begin(9600);
  Serial.println("IR gestartet.");
  irrecv.enableIRIn();
  pinMode(ledRed, OUTPUT);
  pinMode(ledGreen, OUTPUT);
  pinMode(ledRed, OUTPUT);
}
```

## DER LOOP

Im Loop des Sketchs wartet dein IR-Empfänger auf ein Signal von der Fernbedienung und reagiert per **switch … case** entsprechend darauf. Zunächst jedoch eine If-Abfrage:

```
void loop() {

  if (irrecv.decode(&results)) {
```

Innerhalb der Klammern der If-Anweisung prüfst du, ob **irrecv.decode(&results) == true** ist – also wahr ist. Den Teil **== true** kannst du dir jedoch sparen – für diese Prüfung reicht es auch, wenn du einfach nur die Variable einträgst. Diese Prüfung bedeutet, dass dein IR-Empfänger ein Signal von der Fernbedienung empfangen hat. In diesem Fall wird der Code nach der geschweiften Klammer **{** ausgeführt, in dem geprüft wird, welche Taste gedrückt wurde.

Falls bei der Abfrage jedoch ein **false** herauskommt – also kein Signal empfangen wurde – wird weiter unten der Befehl **irrecv.resume()** ausgeführt, also weiter auf ein Signal gewartet.

## SWITCH … CASE

Nun zum Kernstück: Die Abfrage, welche Taste gedrückt wurde und die entsprechende Einstellung der RGB-LED. Zunächst leitest du mit **switch** und der Variablen, auf deren Inhalt du reagieren möchtest, ein. Diese Variable heißt hier **results.value**.

```
switch (results.value) {
```

In diesem Projekt sind nur die Tasten 0 bis 3 mit Funktionen belegt. Die Null schaltet die LED aus, die Tasten 1 bis 3 schalten die Lichtfarbe um. Also benötigst du insgesamt 4 case-Abfragen. Hier die erste:

```
case 0xFF6897:
  Serial.println("0");
  digitalWrite(ledRed, LOW);
  digitalWrite(ledGreen, LOW);
  digitalWrite(ledBlue, LOW);
  break;
```

Dies ist die Abfrage für die Taste 0 (Null). Wie du siehst, wird hier der Name der Taste im Seriellen Monitor ausgegeben und alle Kanäle der RGB-LED ausgeschaltet, also auf **LOW** gesetzt. Doch was bedeutet **0xFF6897** hinter **case**?

Hierbei handelt es sich um eine Zahl im Hexadezimalsystem, die die Taste 0 repräsentiert. Diese Zahl wird von der Fernbedienung gesendet, indem sie ein entsprechendes Signal per Pulsweitenmodulation generiert und per Infrarot sendet. Dieses Signal wird dann im Empfänger dekodiert und, wie in diesem Fall, verwendet, um verschiedene Tasten der Fernbedienung zu unterscheiden.

Im Sketch dieses Projekts findest du nur die Hexadezimal-Codes für die Tasten 0 bis 3. Weiter unten findest du jedoch eine vollständige Liste für alle Tasten deiner Fernbedienung.

Schauen wir uns noch einen weiteren Fall an: Die Taste 1.

```
case 0xFF30CF:
  Serial.println("1");
  digitalWrite(ledRed, HIGH);
  digitalWrite(ledGreen, LOW);
  digitalWrite(ledBlue, LOW);
  break;
```

Hier wird nur der rote Kanal der LED auf HIGH geschaltet, die anderen beiden verbleiben auf LOW – die LED leuchtet also in einem satten Rot. Sobald das passiert ist, folgt der Befehl **break**, mit dem der Code aus diesem **case** wieder herausspringt, sich dann wieder im Loop befindet – und auf das nächste Signal wartet.

Den Sketch findest du hier **polluxlabs.net/maker-buch**. Hier ist die vollständige Liste der Tasten und ihrer Hexadezimal-Codes:

```
switch(results.value) {
  case 0xFFA25D: Serial.println("POWER"); break;
  case 0xFFE21D: Serial.println("FUNC/STOP"); break;
  case 0xFF629D: Serial.println("VOL+"); break;
  case 0xFF22DD: Serial.println("FAST BACK"); break;
  case 0xFF02FD: Serial.println("PAUSE"); break;
  case 0xFFC23D: Serial.println("FAST FORWARD"); break;
  case 0xFFE01F: Serial.println("DOWN"); break;
  case 0xFFA857: Serial.println("VOL-"); break;
  case 0xFF906F: Serial.println("UP"); break;
  case 0xFF9867: Serial.println("EQ"); break;
  case 0xFFB04F: Serial.println("ST/REPT"); break;
  case 0xFF6897: Serial.println("0"); break;
  case 0xFF30CF: Serial.println("1"); break;
  case 0xFF18E7: Serial.println("2"); break;
  case 0xFF7A85: Serial.println("3"); break;
```

```
case 0xFF10EF: Serial.println("4"); break;
case 0xFF38C7: Serial.println("5"); break;
case 0xFF5AA5: Serial.println("6"); break;
case 0xFF42BD: Serial.println("7"); break;
case 0xFF4AB5: Serial.println("8"); break;
case 0xFF52AD: Serial.println("9"); break;
case 0xFFFFFFFF: Serial.println("REPEAT"); break;}
```

# 25 WEITERE BAUTEILE

Natürlich gibt es noch viele weitere Sensoren und Displays, die du mit deinem Arduino verbinden kannst. Auf den folgenden Seiten stellen wir dir viele wichtige Bauteile vor, die du vielleicht einmal für deine eigenen Projekte verwenden möchtest.

## TFT-DISPLAY

Wenn dir LCD zu langweilig wird, oder dein Projekt ein Farbdisplay erfordert, wird es Zeit für ein TFT-Display!

Verbinde das Display mit deinem Arduino wie folgt:

| **Pin am TFT-Display** | **Pin am Arduino** |
|---|---|
| LED | 3.3 V |
| SCK | 13 |
| SDA | 11 |
| A0 oder DC | 9 |
| RESET | 8 |
| CS | 10 |
| GND | GND |
| VCC | 5 V |

Wenn du ein anderes Arduino-Modell verwendest, benötigst du möglicherweise eine andere Belegung der Pins.

Sobald du das Display richtig angeschlossen hast, kann es auch schon mit dem Programmieren losgehen.

## DER CODE FÜR DEIN TFT-DISPLAY

Zunächst benötigst du zwei Bibliotheken, die du einbinden musst. Beide sollten bereits in deiner Arduino IDE vorinstalliert sein.

```
include <TFT.h>;
include <SPI.h>;
```

Die Bibliothek **TFT.h** benötigst du, um auf dem Display schreiben und zeichnen zu können. **SPI.h** hingegen ist für die Kommunikation zwischen Arduino und Display zuständig.

Als nächstes definierst du folgende 3 Pins. Anschließend erstellst du eine Instanz der Bibliothek **TFT** mit dem Namen **TFTscreen**.

```
#define cs 10
#define dc 9
#define rst 8

TFT TFTscreen = TFT(cs, dc, rst);
```

## DIE SETUP-FUNKTION

Hier rufst du zunächst die Instanz auf, die du gerade erstellt hast.

```
TFTscreen.begin();
```

Danach löschst du das Display, indem du seine Hintergrundfarbe auf Schwarz setzt. Die 3 Zahlen in Klammern repräsentieren die RGB-Werte der Hintergrundfarbe – Rot, Grün, Blau. **(0, 0, 0)** ergibt die Farbe Schwarz. Jede dieser Zahlen kann einen Wert zwischen 0 und 255 haben – 3x die Null ergibt Schwarz, 3x 255 Weiß. Dazwischen hast du die Möglichkeit von über 16 Millionen Farben.

```
TFTscreen.background(0, 0, 0);
```

Zuletzt setzt du die Schriftgröße auf 2. Experimentiere mit größeren und kleineren Zahlen und beobachte die Zeichen auf deinem Display.

```
TFTscreen.setTextSize(2);
```

## DER LOOP

Hier wirst du den guten alten Text “Hello, world!” auf dein Display bringen. Aber nicht nur das, du wirst ihn auch in

verschiedenen und zufällig erzeugten Farben anzeigen, um die Möglichkeiten deines Farbdisplays auszunutzen.

Kümmern wir uns zunächst um die zufälligen Farben. Wie du oben gelesen hast, besteht eine RGB-Farbe aus Rot, Grün und Blau. Jede einzelne dieser Farben kann einen Wert zwischen 0 und 255 besitzen. Diesen Wert ermittelst du mit der Funktion **random()** und weist ihn je einer Variablen zu:

```
int redRandom = random(0, 255);
int greenRandom = random (0, 255);
int blueRandom = random (0, 255);
```

Anschließend legst du mit der Funktion **TFTscreen.stroke()** die per Zufall ermittelte Farbe des Textes fest und schreibst mit **TFTscreen.text()** in die Mitte des Displays:

```
TFTscreen.stroke(redRandom, greenRandom, blueRandom);
TFTscreen.text("Hello, world!", 6, 57);
```

Zu guter Letzt baust du noch einen Delay in den Loop ein, mit dem du die Geschwindigkeit des Farbwechsels bestimmst. Also zum Beispiel alle 200 Millisekunden:

```
delay(200);
```

Und das war's! Wenn du den Code jetzt auf deinen Arduino hochlädst, sollte dein TFT-Display "Hello, world!" in wechselnden Farben anzeigen.

Hier der vollständige Code:

```
include <TFT.h>
include <SPI.h>

define cs 10
define dc 9
define rst 8

TFT TFTscreen = TFT(cs, dc, rst);

void setup() {
  TFTscreen.begin();
  TFTscreen.background(0, 0, 0);
  TFTscreen.setTextSize(2);
}

void loop() {
  int redRandom = random(0, 255);
  int greenRandom = random (0, 255);
  int blueRandom = random (0, 255);

  TFTscreen.stroke(redRandom, greenRandom,
blueRandom);
  TFTscreen.text("Hello, world!", 6, 57);
  delay(200);
}
```

# REAL TIME CLOCK (RTC)

Brauchst du für dein Arduino-Projekt die aktuelle Uhrzeit – vielleicht, weil du dir mit einem LC-Display eine Uhr bauen möchtest, oder einen Zeitstempel für deine Daten benötigst? Dann eignet sich hierfür perfekt die RTC, oder Real Time Clock.

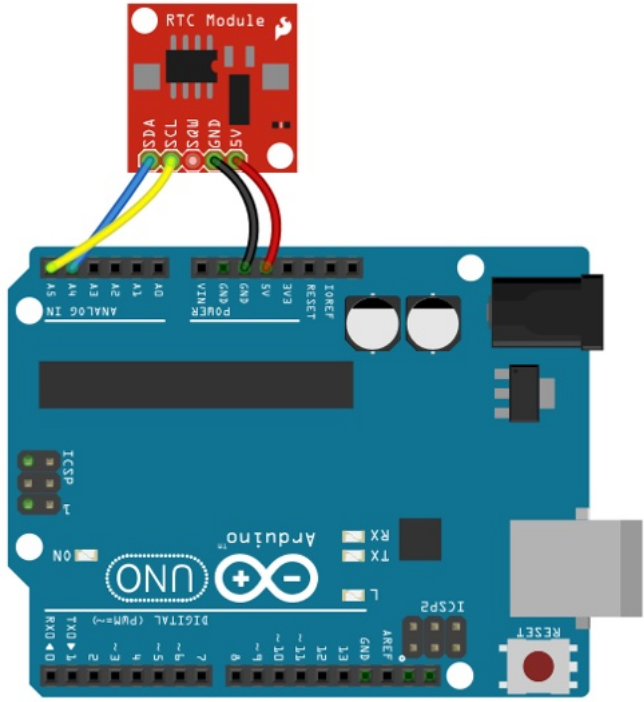
## WAS IST EINE REAL TIME CLOCK?

In den meisten Fällen erhältst du beim Kauf einer Real Time Clock ein Modul, auf dem oft ein DS1307 samt einer kleinen Batterie verbaut ist. Diese sorgt dafür, dass dein Arduino die aktuelle Zeit ermitteln kann, auch wenn es zwischenzeitlich zu einem Stromausfall gekommen ist – denn die RTC wird weiterhin mit Strom versorgt.

Diese Bauteile sind in der Regel recht preisgünstig und werden per I²C am Microcontroller angeschlossen. Die Genauigkeit ist für die allermeisten Projekte ausreichend und die Knopfbatterie hält bis zu fünf Jahre durch.

## DIE RTC AM ARDUINO ANSCHLIESSEN

Dank der Verbindung per I²C benötigst du nur vier Kabel, um die Real Time Clock an deinem Arduino anzuschließen. Wenn du einen Arduino UNO verwendest, benötigst du (neben GND und 5V) hierfür die Pins A4 und A5. Orientiere dich an folgender Skizze:




## DIE BENÖTIGTEN BIBLIOTHEKEN

Um eine RTC mit dem Chip DS1307 verwenden zu können, benötigst du zwei Bibliotheken. Öffne hierfür den Bibliotheksverwalter und suche dort nach **DS1307rtc**. Du findest neben der gleichnamigen Bibliothek auch die Bibliothek **Time**. Installiere jeweils die aktuelle Version.


Sollte deine Real Time Clock einen anderen Chip haben, suche stattdessen nach dessen Namen. **Im Folgenden arbeiten wir jedoch mit dem DS1307 weiter**.

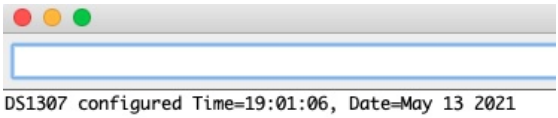
Wenn beide Bibliotheken in deiner Arduino IDE installiert sind, kann es auch schon mit dem Einstellen der Uhr weitergehen.

## DIE AKTUELLE UHRZEIT IN DER RTC EINSTELLEN

Bevor in deinem RTC-Modul die Uhr ticken kann, muss es natürlich erst einmal wissen, wie viel Uhr es ist. Hierfür benötigst du einen Sketch, der bereits in den mitgelieferten Beispielen der Bibliothek **DS1307RTC** vorhanden ist.

Den Sketch findest du in der Arduino IDE unter **Datei > Beispiele > DS1307RTC > SetTime**

Öffne ihn und lade ihn auf deinen Arduino. Mit diesem Sketch “lädt” dein Arduino die aktuelle Uhrzeit samt Datum von deinem Computer und speichert beides in der RTC.

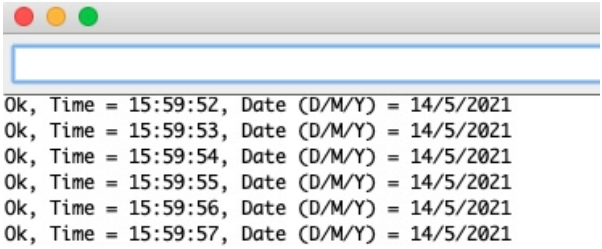



Das war es auch schon mit der Konfiguration. Den Sketch **SetTime** musst du erst wieder nach dem nächsten Batteriewechsel ausführen.

## UHRZEIT UND DATUM IM SERIELLEN MONITOR AUSGEBEN

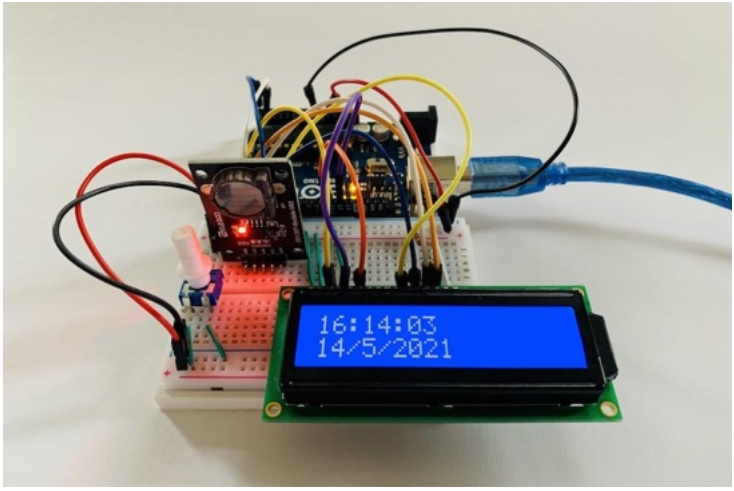
Für einen ersten Test reicht ebenfalls ein Beispiel-Sketch der Bibliothek. Öffne im gleichen Verzeichnis den Sketch **ReadTest** und lade ihn auf deinen Arduino.

In deinem Seriellen Monitor sollte nun im Sekundentakt die aktuelle Uhrzeit und das Datum erscheinen.



## UHRZEIT UND DATUM AUF EINEM LC-DISPLAY ANZEIGEN

Der Serielle Monitor ist das Eine – aber eine Digitaluhr ist etwas ganz Anderes. Schließe hierfür zunächst dein LC-Display am Arduino zusätzlich zur RTC an.




Danach kommt der Sketch an die Reihe. Um es einfach zu halten, haben wir einfach den Sketch **ReadTest** modifiziert und um das Setup des LC-Displays und die Ausgabe darauf erweitert.

Du findest im Loop nun unter jeder Ausgabe von Stunden, Minuten etc. auch den entsprechenden **lcd.print()** direkt darunter.

Ein besonderer Clou ist hierbei die Funktion **void print2digits()**, die auch schon im Sketch vorhanden ist. Diese Funktion sorgt dafür, dass jede Zahl zweistellig angezeigt wird. Ohne diese Funktion würde die Uhrzeit 16:08 als 16:8 auf dem Display dargestellt werden.

Hier nun der gesamte Sketch, du findest ihn auf polluxlabs.net/maker-buch. Achte bitte darauf, dass das LC-Display hier an andere Pins angeschlossen ist, als im Beispiel weiter oben.

## ALKOHOLSENSOR MQ-3

Mit dem Sensor MQ-3 kannst du messen, wie viel Ethanol sich in der Luft – z.B. in der Atemluft – befindet. **Allerdings:** du kannst so **nicht** feststellen, wie viele "Promille" ein Proband intus hat und ob er noch fahrtüchtig ist. Ebenso erhältst du keine absoluten Werte, sondern immer nur einen relativen Wert in Bezug auf die "Frischluft", die du vorab kalibrieren musst.

Das alles klingt komplizierter als es ist! Fangen wir mit einem einfachen Test an. Verbinde zunächst den Sensor MQ-3 wie folgt mit deinem Arduino:

| Pin am MQ-3 | Pin am Arduino |
|---|---|
| VCC | 5V |
| GND | GND |
| DO | Digitalpin 2 |
| AO | Analogpin 0 |

## DEN ANALOGPIN DES MQ-3 AUSLESEN

Um die Analogdaten des Sensors auszulesen und im Seriellen Monitor anzuzeigen, benötigst du nur sehr wenig Code. Kopiere dir den folgenden Sketch und lade ihn auf deinen Arduino. **Achte darauf, dass die Baudrate des Sketchs und deines Seriellen Monitors übereinstimmen.**

```
int analogValue = 0;

void setup() {
  Serial.begin(115200);
  pinMode(analogValue, INPUT);
}

void loop() {
  Serial.println(analogRead(analogValue));
  delay(200);
}
```

Wenn du jetzt deinen Seriellen Monitor öffnest, wirst du einen Haufen stetig fallender Zahlen sehen. Wie eingangs erwähnt, kannst du anhand dieser Zahlen noch keine Aussage über den Ethanolgehalt in der Luft treffen.

**Stattdessen musst du den Sensor einige Minuten in "frischer" Luft stehen lassen** und warten, bis sich die Zahlen auf einen Wert einpendeln. Dieser Wert zeigt dir dann an, dass der Sensor gerade kein Ethanol misst. Ist das jedoch der Fall, springt der Wert hoch – der Abstand zum Referenzwert ermöglicht dir dann eine vage Aussage über die Menge an Alkohol in der Luft.

Bei unserem Test hat sich der "Frischluft-Wert" des MQ-3 nach einigen Minuten auf circa 100 eingependelt. **Dieser Wert kann jedoch bei dir anders sein.**

Wenn du jetzt z.B. eine offene Flasche Gin an den Sensor hältst, wirst du sehen, wie der Wert in deinem Seriellen Monitor steigt. Bei uns ging er hoch bis auf knapp 300.

Über den Analogpin des MQ-3 erhältst du also den relativen Wert des Alkoholgehalts in der Umgebungsluft. In deinem Sketch kannst du einen Schwellenwert festlegen, bei dessen Überschreiten etwas passieren soll – z.B. eine LED aufleuchtet.

Du kannst diesen Schwellenwert jedoch auch direkt am Sensor einstellen und seine Überschreitung vom Digitalpin des MQ-3 auslesen.

## SO VERWENDEST DU DEN DIGITALPIN DES MQ-3

Für einen ersten Test reicht wieder ein ganz simpler Sketch aus. Lade den folgenden Code auf deinen Arduino:

```
int digitalPin = 2;

void setup() {
  Serial.begin(115200);
  pinMode(digitalPin, INPUT);
}

void loop() {
  Serial.println(digitalRead(digitalPin));
  delay(200);
}
```

In deinem Seriellen Monitor solltest du statt den Analogwerten vom ersten Test jetzt entweder eine 0 oder eine 1 sehen. **Auch beim Verwenden des Digitalpins musst du den Sensor**

**zunächst ein paar Minuten in Betrieb nehmen und ruhen lassen, bis er "weiß", was Frischluft bedeutet.**

Wenn du das getan hast, widmest du dich der Rückseite des MQ-3: Hier findest du ein Potentiometer, das du mit einem Schraubendreher einstellen kannst. Drehe die Schraube so weit, bis die Zahl in deinem Seriellen Monitor gerade auf die 1 springt und so bleibt.

Wiederhole jetzt den Test mit einer Flasche Alkohol und du wirst sehen, dass die Anzeige auf 0 springt.

## 7-SEGMENT-ANZEIGE MIT 8 STELLEN

Du hast bereits eine Vielzahl von Displays kennengelernt (auch eine 7-Segment-Anzeige mit einer Stelle), aber keines davon hat so einen Old-School-Faktor wie das 7-Segment-Display mit 8 Stellen. Back To The Future? Bitte schön! **Im Folgenden lernst du, wie du dieses Display anschließt und Zahlen darauf anzeigst.**

### DER ANSCHLUSS AM ARDUINO

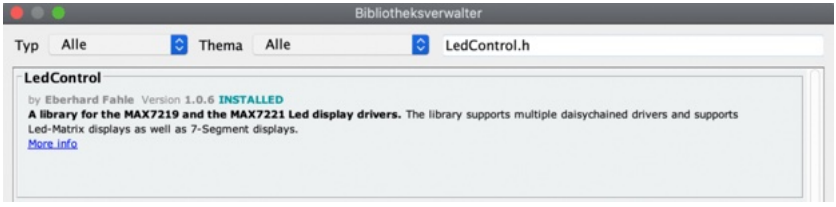
Um die 7-Segment-Anzeige anzuschließen, benötigst du drei freie Digitalpins am Arduino. Du kannst das Display wahlweise mit 3,3V oder 5V betreiben. In diesem Tutorial erfolgt der Anschluss wie folgt:

| Arduino | 7-Segment-Anzeige |
| --- | --- |
| GND | GND |
| 3,3V oder 5V | VCC |
| 10 | CS |
| 11 | CLK |
| 12 | DIN |

Meistens wird das Display mit verlöteten Pins geliefert, sodass du 5 Kabel der Sorte male-female benötigst.

## DIE PASSENDE BIBLIOTHEK FÜR DAS 7-SEGMENT-DISPLAY

Für die Steuerung der Anzeige gibt es eine passende Bibliothek, die dir das Leben erleichtert. Öffne also deinen Bibliotheksverwalter in der Arduino IDE, suche nach **LedControl** und installiere die aktuelle Version.



## DER SKETCH FÜR DEN ERSTEN TEST

Für den allerersten Versuch soll eine einzige Ziffer auf dem Display genügen. Wie so oft ist der erste Schritt, die oben genannte Bibliothek einzubinden:

```
#include "LedControl.h"
```

Anschließend legst du fest, an welchen Digitalpins du das Display angeschlossen hast. **Hierbei ist die Reihenfolge DIN, CLK, CS entscheidend.** Das letzte Argument in der folgenden Codezeile ist die Anzahl der Displays, die du steuern möchtest. Theoretisch könntest du mit der Bibliothek so viele Ziffern darstellen, dass es für die globale Schuldenuhr der nächsten Jahrzehnte reichen würde – aber wir bleiben hier erst einmal bescheiden.

```
LedControl lc=LedControl(12,11,10,1);
```

## DIE ANZEIGE ANSCHALTEN UND EINE ZIFFER ANZEIGEN

Kommen wir zur Funktion **setup()**. Hier erledigst du zu Beginn des Sketchs drei Dinge: das Display aus dem Sleep Mode aufwecken, die Helligkeit einstellen und alle Ziffern darauf löschen, die vielleicht noch darauf zu sehen sein könnten.

```
lc.shutdown(0,false);
lc.setIntensity(0,8);
lc.clearDisplay(0);
```

Was die Helligkeit angeht, kannst du der Funktion **lc.setIntensity()** eine Zahl von 0 bis 15 mitgeben.

Kommen wir also zum entscheidenden Moment. Ebenfalls in der Setup-Funktion schreiben wir in das erste Feld der 7-Segment-Anzeige (ganz rechts) die Ziffer 9:

```
lc.setDigit(0, 0, 9, false);
```

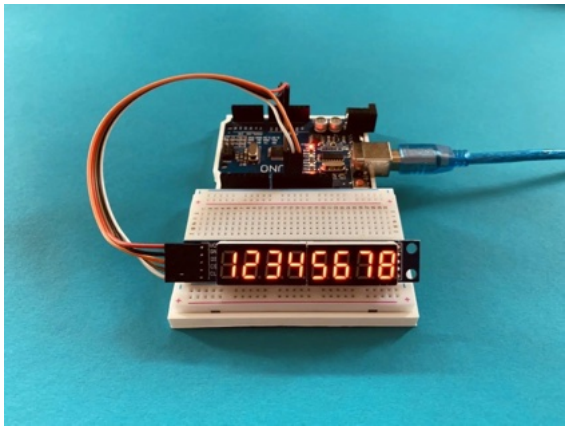
Wenn du die 9 ins erste Feld ganz links schreiben möchtest, wäre der Code hierfür folgender. Wie üblich fängst du bei der 0 an zu zählen – und zwar von rechts. Das ganz linke Feld erhält dann die Nummer 7:

```
lc.setDigit(0, 7, 9, false);
```

Was passiert, wenn du statt der 9 eine 10 einträgst? Dann wird diese Dezimalzahl im Hexadezimalsystem dargestellt, also mit dem Buchstaben A. Das geht bis zur Zahl 15, die dann entsprechend als F ausgegeben wird.

## LANGE ZAHLEN AUF DEM 7-SEGMENT-DISPLAY DARSTELLEN

Jedes Feld der Anzeige mit einer Ziffer zu belegen funktioniert also, ist unter Umständen aber recht mühselig. Was, wenn du einfach die Zahl 12345678 auf einmal ausgeben möchtest? Mit zwei Hilfsfunktionen ist das kein Problem.




Zunächst definierst du 8 Variablen, die später die einzelnen Ziffern deiner Zahl enthalten. Die Variable **a** wird die erste Ziffer und **h** die letzte Ziffer enthalten.

```
int a;
int b;
int c;
int d;
int e;
int f;
int g;
int h;
```

Anschließend benötigst du eine (auf den ersten Blick etwas komplizierte) Funktion, die deine Zahl in diese Ziffern aufteilt. Das bewerkstelligst du mit dem Operator Modulo, der den Rest der Division einer ganzen Zahl durch eine andere berechnet.

Die letzte Ziffer der Zahl 12345678 ist – die 8, die im Feld ganz rechts stehen soll. Mit folgender Rechnung erhältst du diese Ziffer und speicherst sie in der Variablen **h**:

```
h = number % 10;
```

Ausgeschrieben lautet diese Rechnung wie folgt: **h = 12345678 : 10 = 1234567,8** – der Rest hinter dem Komma ist also die 8, unsere Ziffer ganz rechts.

Weiter geht es mit der vorletzten Ziffer. Hier prüfst du zunächst, ob die darzustellende Zahl überhaupt zwei Ziffern hat (hat sie, deine Zahl hat sogar 8). Falls ja, teilst du sie zunächst durch 10 und erhältst dadurch die Zahl 1234567. **Hier fällt die Nachkommastelle weg,** da wir die Variable **number** ja mit **long** angelegt haben und bei diesem Typ die Nachkommastellen automatisch gestrichen werden, da er nur ganze Zahlen speichern kann.

Eine weitere Rechnung mit dem Modulo beschert dir dann die vorletzte Ziffer 7:

```
if (number > 9) {
  g = (number / 10) % 10;
}
```

Diese Rechnungen führst du für alle Ziffern in deiner Zahl aus, wie du unten im vollständigen Sketch sehen kannst. Sobald die Funktion durchgelaufen ist, musst du nur noch alle Ziffern auf das 7-Segment-Display bringen. Hierfür verwendest du eine weitere Funktion, die prüft, wie viele Ziffern deine Zahl hat und diese in den Variablen **a – h** gespeicherten Ziffern darstellt:

```
lc.setDigit(0, 0, h, false);

if (number > 9) {
  lc.setDigit(0, 1, g, false);
}
.
.
.
```

Auf polluxlabs.net/maker-buch findest du den Sketch, mit dem du Zahlen auf das Display bringst. Speichere zum Testen verschiedene, maximal achtstellige Zahlen in der Variable **number.**

## EFFEKTIVERER CODE

Du kannst den Code oben auch komprimieren, sodass du nur noch eine Funktion benötigst. Definiere hierfür zunächst, wie

viele Ziffern dein Display darstellen kann – in unserem Fall also 8. Anschließend erledigst du die Berechnung der einzelnen Ziffern und ihre Darstellung in fünf Zeilen Code:

```
const int NUM_DIGITS = 8;

void drawDigits(int num) {
  for (int i = 0; i < NUM_DIGITS ; i++) {
    lc.setDigit(0, i, num % 10, false);
    num /= 10;
    if (num == 0)
      return;
  }
}
```

## PIR-BEWEGUNGSSENSOR HC-SR501

Wenn du Bewegungen erkennen möchtest, um daraufhin z.B. das Licht einzuschalten, ist der Bewegungssensor HC-SR501 (auch PIR für “passiver Infrarot-Sensor” genannt) eine oft ausreichend gute und günstige Wahl.

In erster Linie reagiert dieser Sensor auf Wärme in Form von Infrarotstrahlen. Damit nicht die Heizung das Licht einschaltet, reagiert der Sensor nur auf warme Objekte, die sich bewegen. Hierfür ist eine sogenannte **Fresnel-Linse** (die “Kuppel”) über dem eigentlichen Sensor verbaut.

## DEN BEWEGUNGSMELDER ANSCHLIESSEN

Der Anschluss des Sensors ist ganz simpel. Gerade einmal drei Pins benötigst du hierfür: Plus (VCC), Minus (GND) und OUT. Letzteren schließt du an einen Digitalpin deines Arduinos an. Sobald dein Sensor eine Bewegung erkennt, sendet er ein HIGH an den Arduino, das du in deinem Sketch auslesen kannst.

An der Unterseite des Bewegungsmelders findest du zwei Potentiometer. Drehe den Sensor um, sodass die Platine nach oben zeigt. Am linken Potentiometer kannst du die Empfindlichkeit des Sensors einstellen; am rechten die Länge des Signals, das an deinen Arduino gesendet wird. **Stellst du hier eine lange Zeitspanne ein, hält der Bewegungsmelder den "Alarm", damit in dieser Zeit keine weitere Bewegung erkannt werden kann.** Das ist praktisch, wenn du nicht möchtest, dass ein und dieselbe Bewegung mehrere Signale sendet.

## DER PASSENDE SKETCH

Für diesen Sketch haben wir eine LED an den Digitalpin 9 angeschlossen. Diese leuchtet auf, sobald eine Bewegung erkannt wurde. Der Bewegungssensor ist mit dem Digitalpin 7 verbunden.

Neben den Einstellungen und Angaben für die verwendeten Pins benötigst du nur eine Variable und eine IF-Abfrage:

```
status = digitalRead(sensor);

if (status == HIGH){
digitalWrite(led, HIGH);
} else {
digitalWrite(led, LOW);
}
```

Den Wert, den der Bewegungsmelder ausgibt, speicherst du in der Variablen **status** – liegt keine Bewegung vor, ist dieser Wert 0. Wenn sich etwas vor dem Sensor bewegt, entsprechend 1.

Anschließend lässt du die LED bei einer Bewegung aufleuchten. Wenn dein Bewegungsmelder wieder auf Null umschaltet – weil er keine Bewegung erkennt – geht sie wieder aus.

Hier der vollständige Sketch:

```
int led = 9;
int sensor = 7;
int status = 0;

void setup(){
  pinMode(led, OUTPUT);
  pinMode(sensor, INPUT);
}

void loop(){
  status = digitalRead(sensor);

  if (status == HIGH){
    digitalWrite(led, HIGH);
    } else {
      digitalWrite(led, LOW);
      }
}
```

# 26 WEITER GEHT'S MIT DEM ESP8266

Kennst du dich schon gut mit dem Arduino aus? Dann wird es Zeit für die nächsten Schritte – z.B. mit dem **NodeMCU ESP8266**. Dieser Microcontroller besitzt im Prinzip die gleichen Funktionalitäten wie der Arduino, allerdings kannst du mit ihm per WLAN ins Internet. Hieraus ergeben sich viele spannende Möglichkeiten, die du in den folgenden Abschnitten kennenlernst.

Zunächst erfährst du mehr über die wichtigsten Parallelen und Unterschiede zum Arduino. Dann verbindest du den ESP8266 mit dem Internet und **findest heraus, wie viele Menschen gerade im Weltall sind**. Hierbei lernst du, wie du API-Abfragen erstellst und die Daten verarbeitest, die du von dort erhältst. Das ist der perfekte Start für viele weitere Anwendungen, wie z.B. Wetterstationen, die ihre Messdaten loggen oder die Wettervorhersage aus dem Internet laden.

## DER ESP8266 UND SEINE PINS

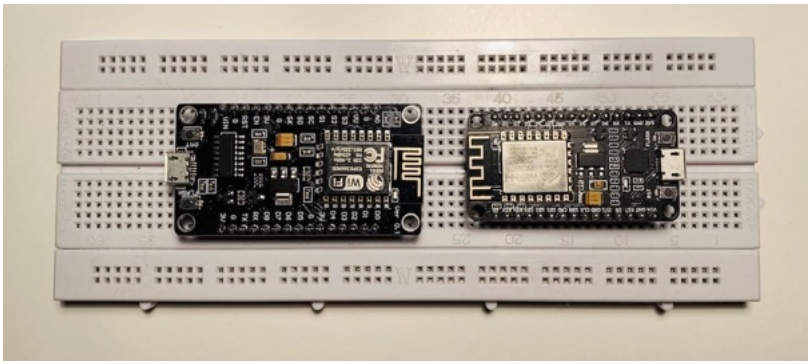
Wenn von einem ESP8266 die Rede ist, ist meistens ein ganzes Board gemeint. Das kann zum Beispiel ein NodeMCU ESP8266 sein, aber auch der kleine ESP-01. Oder auch ein Wemos D1 Mini.

Der ESP8266 selbst ist eigentlich nur der auf diesem Board verbaute Chip des Herstellers Espressif. Dieser verfügt über ein WLAN-Modul – was wiederum der Hauptgrund für seinen Einsatz in Projekten sein dürfte.

NodeMCU ESP8266 Amica

## NODEMCU AMICA VS. LOLIN

Wenn du dir einen weiteren ESP8266 kaufen möchtest, wirst du auf zwei verschiedene Modelle stoßen: **v2 Amica und v3 LoLin**. Um es kurz zu machen, die beiden Versionen unterscheiden sich in technischer Hinsicht nur marginal. **Die LoLin-Version hat jedoch einen entscheidenden Nachteil:** Sie nimmt auf deinem Breadboard die gesamte Fläche ein, sodass du keine Kabel mehr daneben stecken kannst und somit nur sehr schwer an die Pins kommst.



LoLin vs. Amica

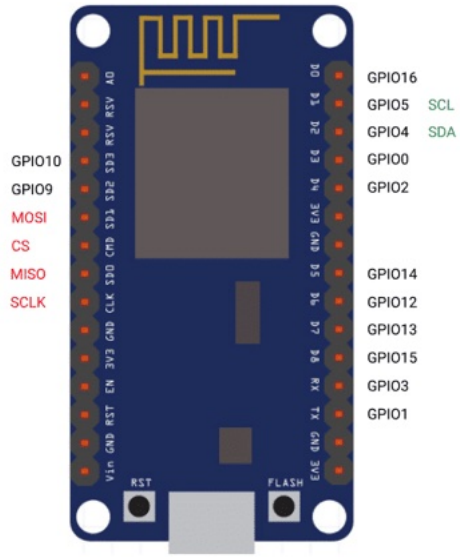
Bei der Amica-Version geht das hingegen – DAS Argument, diese Version vorzuziehen.

## DIE WICHTIGSTEN PINS

Apropos Pins: An deinem NodeMCU ESP8266 findest du insgesamt 30 Pins, die unterschiedlichen Zwecken dienen. Einige davon sind selbsterklärend, **wie zum Beispiel GND oder 3V3.** Erstere dienen als Erde, mit den letzteren versorgst du deine Sensoren, Displays etc. mit Strom (mit 3,3 Volt).

Etwas kniffliger wird es mit den Digitalpins. Diese sind zwar auf dem Board mit einem **D** und einer fortlaufenden Nummer beschriftet – diese Nummer kannst du allerdings nicht einfach in deinem Sketch verwenden.

In der folgenden Übersicht findest du die entsprechenden Nummern (GPIO, general purpose input/output), die du dort verwenden musst:




Die wichtigsten Pins des ESP8266

Ein Beispiel: Du hast ein Bauteil an Pin D0 des ESP8266 angeschlossen. Laut der Grafik oben entspricht das dem GPIO16.

Grundsätzlich hast du in deinem Sketch die Möglichkeit, beide Methoden zu verwenden – beide führen zum selben Ergebnis:

```
#define LED D0 //Beschriftung am Board
#define LED 16 //entsprechender GPIO
```

**Aber:** Das funktioniert nicht immer. Es gibt durchaus Bibliotheken, die von dir die verwendeten GPIOs erwarten.

### I²C UND SPI

Lass uns noch kurz über diese beiden Anschlüsse sprechen. Es gibt Bauteile, z.B. einige Sensoren und Displays, die du per I²C oder SPI verbinden musst. Auch hierfür die richtigen Pins am NodeMCU zu finden, ist nicht ganz leicht, da sie nicht entsprechend beschriftet sind.

In der Grafik oben findest du sie rot bzw. grün markiert.

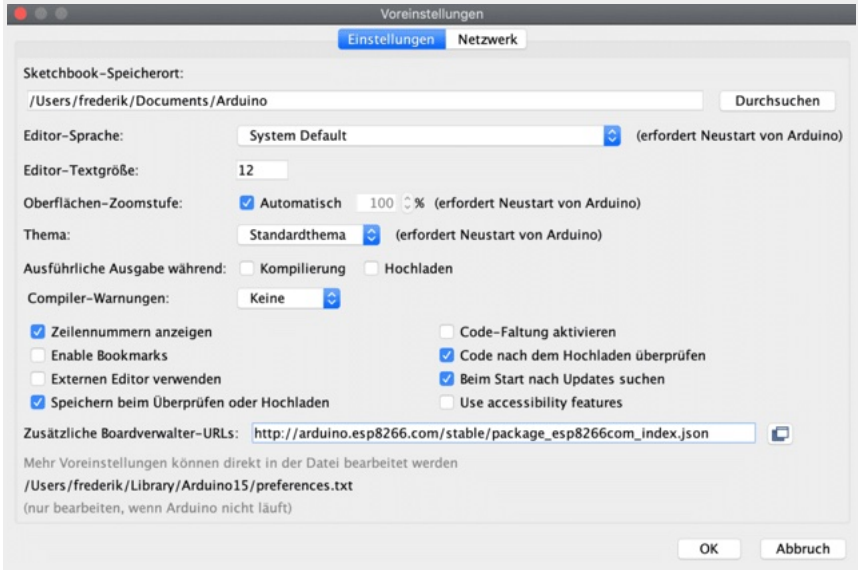
## DEN ESP8266 MIT DER ARDUINO IDE PROGRAMMIEREN

Du kannst deinen ESP8266 genauso programmieren, wie du es von deinem Arduino gewohnt bist. Hierfür musst du allerdings ein paar Vorbereitungen in der Arduino IDE treffen. Das dauert aber nicht länger als 5 Minuten.

Öffne zunächst die Einstellungen der Arduino IDE. Im unteren Bereich findest du das Feld **Zusätzliche Boardverwalter-URLs**.

Trage hier folgende Adresse ein:

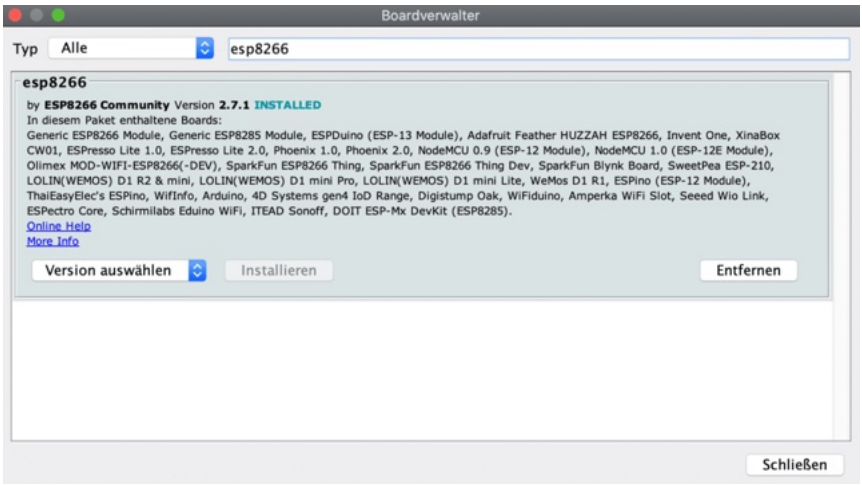
```
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```




**Tipp:** Wenn du dort später einmal schon eine andere URL – z.B. deines ESP32 – eingetragen hast, schreibe die des ESP8266 einfach mit einem Komma getrennt dahinter. Dann verfügst du in der Arduino IDE über beide.

Schließe nun das Fenster mit einem Klick auf **OK**. Öffne als nächstes das Menü **Werkzeuge** und wähle dort den Menüpunkt **Board** und anschließend **Boardverwalter**.

Suche in dem Fenster, das sich jetzt öffnet, nach **ESP8266**. Scrolle etwas nach unten, bis du den Eintrag **ESP8266 by ESP8266 Community** findest. Noch ein Klick auf **Installieren**, kurz warten und das sollte es gewesen sein.




## EIN ERSTER TEST

Lass uns gleich einmal testen, ob die Verbindung zu deinem ESP8266 funktioniert. Erstelle einen neuen Sketch und kopiere folgenden Code hinein:

```
int ledPin = 16;

void setup() {
 pinMode(ledPin, OUTPUT);
 }

void loop() {
 digitalWrite(ledPin, LOW);
```

```
  delay(500);
  digitalWrite(ledPin, HIGH);
  delay(500);
  }
```

Dieser Sketch wird die interne LED deines ESP8266 im Halbsekundentakt blinken lassen. Diese LED befindet sich an Pin D0, bzw. GPIO16.

Schließe nun deinen ESP8266 an den USB-Port deines Computers an. Wähle noch einmal im Menü den Punkt **Werkzeuge** und wähle unter **Board** den Eintrag **NodeMCU 1.0** und den Port, an dem dein ESP8266 angeschlossen ist.

Lade jetzt den Sketch hoch. Blinkt die LED?

**Hinweis:** Solltest du mit dem ESP8266 in der Arduino IDE Probleme beim Hochladen deiner Sketches haben, installiere probeweise eine frühere Version des Boards.

## WIE VIELE MENSCHEN SIND GERADE IM WELTALL?

Was wäre dein ESP8266 ohne Internetverbindung? Eben. Lass uns also schnell auf den Code schauen, den du dafür benötigst. **Du kannst das Snippet dann in all deinen weiteren Projekte verwenden.**

## DIE PASSENDE BIBLIOTHEK FÜR DIE INTERNETVERBINDUNG

Für die Internetverbindung benötigst du zunächst die Bibliotheken **ESP8266WiFi.h** und **WiFiClient.h**

Diese sollten in deiner Arduino IDE bereits verfügbar sein – sofern du dort dein Board installiert hast. **Wenn das noch nicht der Fall ist, gehe noch einmal zurück.** Du kannst die Bibliothek dann wie folgt am Anfang deines Sketchs – noch vor der Setup-Funktion – einbinden:

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
```

## DEINE ZUGANGSDATEN

Bevor sich dein Controller mit deinem WLAN-Netzwerk verbinden kann, benötigt er die passenden Zugangsdaten. Auch diese legst du am Anfang deines Sketchs zum Beispiel in unveränderlichen Konstanten fest:

```
const char* ssid = "Name deines WLAN-Netzwerks";
const char* password =  "Dein WLAN-Passwort";
```

## UND AB INS INTERNET!

Jetzt kann es losgehen. Es gibt prinzipiell mehrere Möglichkeiten, die Verbindung einzurichten und den Verbindungsaufbau im Seriellen Monitor darzustellen. Zentral

ist jedoch immer die Funktion **WiFi.begin()** und dass diese **bestenfalls im Setup deines Sketchs ausgeführt wird**, damit für den Loop alles vorbereitet ist.

Trage folgenden Code in deine Setup-Funktion ein:

```
void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Ich verbinde mich mit dem
Internet...");
  }
  Serial.println("Ich bin mit dem Internet
verbunden!");
}
```

Zunächst rufst du die Funktion **WiFi.begin()** auf, der du deine Zugangsdaten als Argumente mitgibst. Der anschließende Loop wird solange ausgeführt, wie die Verbindung zum Internet noch nicht steht **(WiFi.status() != WL_CONNECTED)** und schreibt jede Sekunde in den Seriellen Monitor, dass die Verbindung aufgebaut wird.

Sobald diese steht, erhältst du die Erfolgsmeldung im Seriellen Monitor. Und das war es auch schon! Die Funktion **void loop()** benötigst du in diesem Sketch nicht.

**Hinweis:** Da dein ESP8266 mit schnelleren Baudraten umgehen kann, beträgt diese hier 115200. Achte darauf, dass sie mit der Einstellung in deinem Seriellen Monitor übereinstimmt.

Solltest du Probleme beim Verbindungsaufbau haben, prüfe zunächst deine Zugangsdaten genau. Ein weitere Fehlerquelle ist oft die Frequenz des WLAN – dein ESP8266 funktioniert nur in einem Netzwerk mit 2,4 GHz. Solltest du also 5 GHz nutzen, erstelle am besten ein weiteres Netzwerk mit 2,4 GHz. Konsultiere hierfür die Dokumentation deines Providers bzw. Routers.

## APIS UND DATEN: WIE VIELE MENSCHEN SIND IM WELTALL?

Wenn du mit deinem ESP8266 im Internet bist, dann sicher nicht ohne Grund. Vielleicht möchtest du Daten von einer API abrufen und in deinem Projekt weiterverwenden. **Als nächstes lernst du, wie du Daten im JSON-Format lädst und mithilfe der Bibliothek ArduinoJson dekodierst (oder parst).**

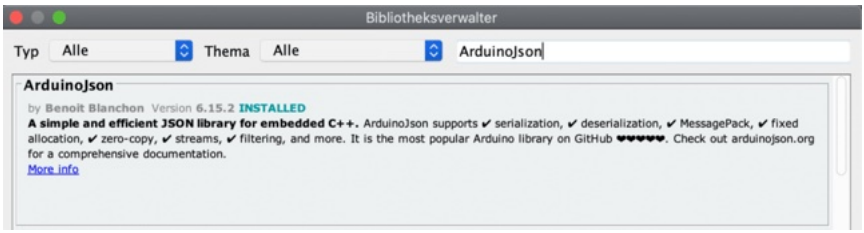
Hierfür werden wir eine API kontaktieren und JSON-Daten herunterladen, die die aktuelle Anzahl von Menschen im Weltraum enthalten. Diese Daten wirst du auf deinem ESP8266 parsen und das Ergebnis in deinem Seriellen Monitor anzeigen.

## DIE BIBLIOTHEK ARDUINOJSON

Bevor wir loslegen können, benötigen wir die aktuelle Version der Bibliothek **ArduinoJson**. Hierbei handelt es sich um eine

wirklich praktische Erweiterung, die dir die meiste Arbeit mit JSON abnimmt.

Öffne also deinen Bibliotheksverwalter in der Arduino IDE und suche dort nach **ArduinoJson**. Installiere dir die neueste Version.




Sobald die Bibliothek installiert ist, kannst du sie in deinem Sketch einbinden. Erstelle zunächst einen neuen Sketch und kopiere folgende Zeile in die erste Zeile (noch vor der Setup-Funktion):

```
#include <ArduinoJson.h>
```

## DER API CALL

Um herauszufinden, wie viele Astronauten gerade im Weltall sind, fragen wir eine API von open-notify.org ab – und zwar unter folgender URL:

```
http://api.open-notify.org/astros.json
```

Wenn du diese URL kopierst und in deinem Browser öffnest, siehst du bereits die Rohdaten im JSON-Format. Recht am Anfang findest du **“number”** und dahinter die aktuelle Anzahl Astronauten im Weltraum. Im Herbst 2022 waren das beispielsweise 7.

Dahinter findest du weitere Informationen zu den einzelnen Astronauten – ihren Namen und wo sie sich gerade befinden.

**Bevor wir jedoch loslegen können, müssen wir den ESP8266 zunächst mit dem Internet verbinden.** Binde also zunächst wieder die entsprechenden Bibliotheken und deine Zugangsdaten zu Beginn deines Sketchs ein. Von letzterer erstellst du die Instanz **wifiClient**.

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
WiFiClient wifiClient;

const char* ssid = "Name deines WLAN-Netzwerks";
const char* password =  "Dein WLAN-Passwort";
```

Außerdem benötigst du noch eine weitere Bibliothek für den API Call – **ESP8266HTTPClient.h**

Diese musst du nicht eigens installieren, da sie nach der Installation des ESP8266 in der Arduino IDE verfügbar ist. Binde

auch diese Bibliothek zu Beginn hinter den anderen Bibliotheken ein:

```
#include <ESP8266HTTPClient.h>
```

## DIE SETUP-FUNKTION

In der Setup-Funktion folgt dann der Code, um deinen ESP8266 mit deinem WLAN zu verbinden:

```
void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Ich verbinde mich mit dem
Internet...");
  }
  Serial.println("Ich bin mit dem Internet
verbunden!");
}
```

## DER LOOP

Jetzt hast du alle Vorbereitungen im Sketch getroffen und kannst nun die Daten von der API abrufen – das machst du im Loop.

Hier prüfst du jedoch zunächst einmal, ob der ESP8266 tatsächlich mit dem Internet verbunden ist:

```
if ((WiFi.status() == WL_CONNECTED)) {
```

Wenn das der Fall ist, kannst du die Abfrage starten:

```
HTTPClient http;
http.begin(wifiClient,
"http://api.open-notify.org/astros.json");
int httpCode = http.GET();
```

In der ersten Zeile erstellst du zunächst eine Instanz der Bibliothek ESP8266HTTPClient.h mit dem Namen **http**.

Danach folgt die Abfrage an der oben genannten URL. Wenn du diese Adresse im Browser öffnest, siehst du einfach die Daten. Im Hintergrund sendet der entsprechende Server jedoch einen Code. Wenn der Server erreichbar ist und deine Anfrage annehmen kann, ist das der Code 200.

Nur in diesem Fall, erhalten wir also die Daten, die wir möchten. Deshalb speicherst du in der dritten Zeile diesen Code zunächst in der Variablen **httpCode**. Anschließend erstellst du eine weitere Abfrage, die den nachfolgenden Code nur ausführt, wenn vom Server der Code 200 kommt:

```
if (httpCode == 200) {
```

Die Daten erhältst du vom Server mit der Funktion **http.getString()** und speicherst sie in der Variablen **input**:

```
String input = http.getString();
```

Jetzt befinden sich hierin die Rohdaten, die du von der API geladen hast – und zwar im JSON-Format**.** Das ist im Prinzip genau die Zeichenkette, die du auch im Browser gesehen hast.

Schön und gut, aber um diese Daten weiterverarbeiten zu können – also z.B. die aktuelle Anzahl Astronauten zu erhalten – musst du sie zunächst dekodieren bzw. parsen. Und hier kommt die Bibliothek ArduinoJson ins Spiel.

## PARSEN MIT ARDUINOJSON

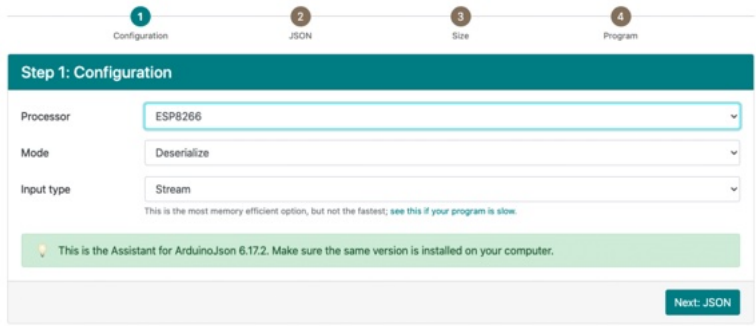
Mit dieser Bibliothek greifst du dir einzelne Daten aus dem JSON-String und speicherst sie in Variablen deiner Wahl. Damit sie jedoch ihre Arbeit erledigen kann, muss sie zunächst wissen, wie groß die Rohdaten sind, um sich selbst genügend Arbeitsspeicher auf dem ESP8266 zu reservieren.

Hierfür gibt es einen praktischen Assistenten. Öffne zunächst noch einmal die URL der API (http://api.open-notify.org/astros.json) und kopiere dir mit **Strg + A** und **Strg + C** sämtliche Zeichen.

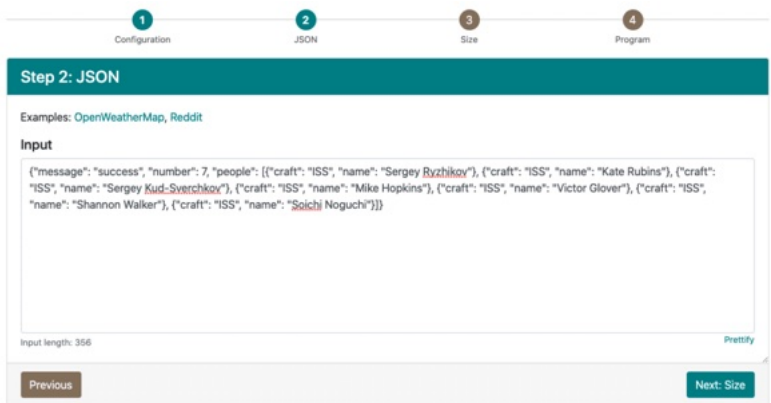
Öffne anschließend den Assistenten von ArduinoJson:

arduinojson.org/v6/assistant/

Hier erwartet dich eine Art Wizard. Wähle im ersten Schritt die folgenden Einstellungen und klicke auf **Next: JSON**.

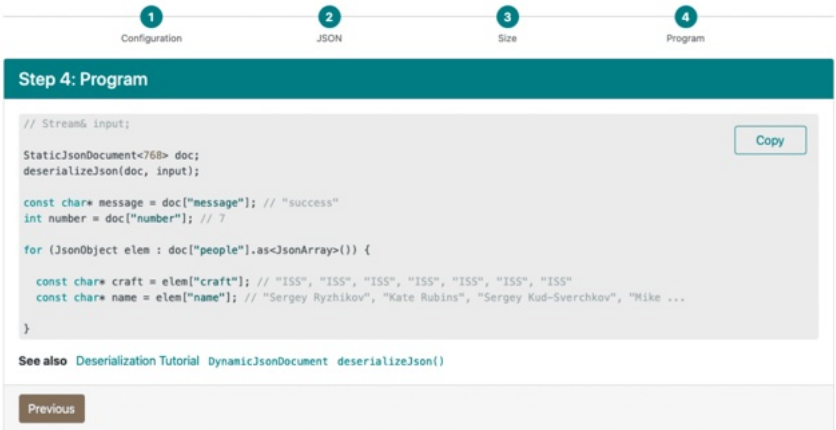



Kopiere im nächsten Schritt den JSON-String aus deinem Browser in das Textfeld und klicke dich durch zum nächsten Schritt.



In Schritt 3 erhältst du Informationen über die Größe der Daten, die wir jedoch an dieser Stelle nicht weiter beachten müssen. Je größer der JSON-String, desto größer ist auch der Speicher, den die Bibliothek zum Parsen benötigt. Gehe nun weiter zu Schritt 4.

Hier wird es jetzt interessant, denn du erhältst hier bereits den Code, den du im Sketch benötigst.




Zunächst die erwähnte Speichergröße. In obigem Beispiel sind das 768 Bytes. Bei dir können es jedoch mehr oder weniger sein – je mehr Astronauten im Weltall sind, desto größer dein Speicherbedarf.

Trage die ersten beiden Zeilen nun in deinen Sketch ein – direkt unter der Zeile, in der wir die Daten in die Variable **input** gespeichert haben.

```
StaticJsonDocument<768> doc;
deserializeJson(doc, input);
```

Die erste reserviert den benötigten Speicher und die zweite kümmert sich dann schon um das Dekodieren der Daten.

Um die Zahl der Menschen im Weltall nun herauszufinden, ist nur eine weitere Zeile Code nötig:

```
int number = doc["number"];
```

Hier speicherst du in der Variablen **number** den Wert aus dem Key-Value-Paar **“number”: 7** im Json String. Der Key, den wir in der oberen Zeile aufrufen, lautet **“number”** – und der hier hinterlegte Wert (Value) **7**. Beachte wieder: Der Wert kann bei dir anders lauten.

Jetzt musst du nur noch den Inhalt der Variablen **number** im Seriellen Monitor ausgeben:

```
  Serial.println(number);
```

Und das war es auch schon! In deinem Seriellen Monitor sollte nun die Zahl der Astronauten im Weltall erscheinen.

**Noch ein paar Hinweise:** Wenn du die Abfrage nur einmal beim Start des ESP8266 ausführen möchtest, kannst du den gesamten Inhalt des Loops auch im Setup ausführen. Die Funktion **void loop()** bleibt dann einfach leer.

Wenn du allerdings den Loop nutzt – so wie wir in diesem Beispiel – füge am Ende noch einen **delay** ein. Ansonsten würde dein ESP8266 eine Abfrage nach der anderen an die API starten. Das wäre völlig unnötig, denn wie du weißt, verändert sich die Anzahl der Menschen im Weltall eher selten.

Und das war es auch schon mit den Grundlagen deines ESP8266. Den vollständigen Sketch findest du auf **polluxlabs.net/maker-buch**

# 27 DEIN EIGENER ESP8266 WEBSERVER

Als nächstes baust du dir einen eigenen ESP8266 Webserver. Mit diesem kannst du über dein Smartphone Geräte steuern – in diesem Fall eine LED. Aber dein Server sendet auch Daten, nämlich die aktuelle Temperatur.

Du gestaltest eine einfache Webseite, auf der du die Messdaten ablesen und die LED ein- und ausschalten kannst. Aber alles der Reihe nach.

## EINEN TEMPERATURSENSOR ANSCHLIESSEN

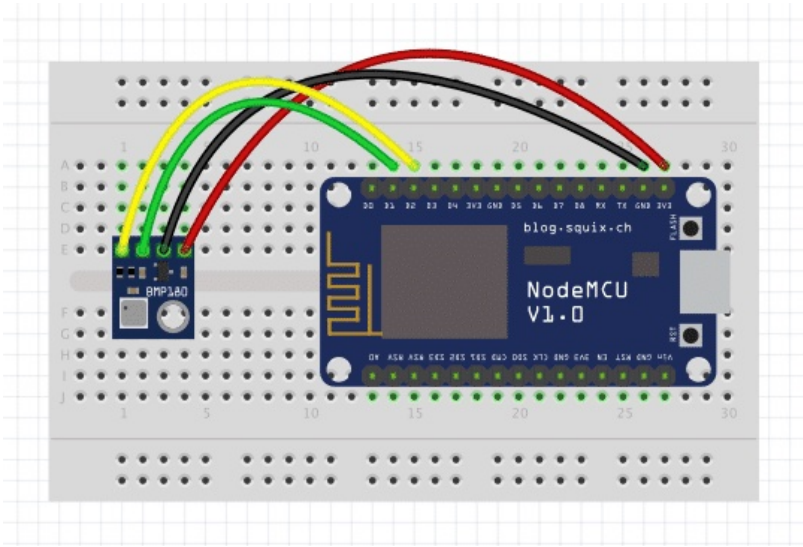
Natürlich gibt es eine Vielzahl von Sensoren, deren Daten du über einen Webserver überprüfen kannst. Wir möchten hier jedoch den wohl beliebtesten Anwendungsfall beschreiben – Temperaturdaten.

Hierfür gibt es auf dem Markt einige günstige Sensoren, die immer wieder zum Einsatz kommen und die sich bewährt haben. Du kannst sie im Handumdrehen anschließen und dank der passenden Bibliotheken ebenso einfach die Temperatur messen.

Schauen wir uns drei der bekanntesten Sensoren an: Den BMP180, den GY-906, den DHT22 und DHT11, den du ja bereits kennst.

## DEN BMP180 ANSCHLIESSEN

Diesen Sensor schließt du per I²C an deinem ESP8266 an. Hierfür, benötigst du die Pins D1 und D2. Orientiere dich beim Aufbau an folgender Skizze:




## DIE PASSENDE BIBLIOTHEK

Neben der bereits vorinstallierten Bibliothek **Wire** (für die Kommunikation per I²C), benötigst du noch eine weitere, um die Daten des Sensors problemlos auslesen zu können.

Öffne also den Bibliotheksmanager in der Arduino IDE und suche nach **BMP180**. Du findest nun eine Bibliothek namens **Adafruit BMP085 Library** – das ist die richtige, auch wenn sie ein anderes Modell im Namen trägt. Der BMP085 war das

Vorgängermodell des BMP180, was die Kommunikation angeht, jedoch mehr oder weniger baugleich.


Installiere nun diese Bibliothek und schließe den Bibltiotheksmanager.

## DIE TEMPERATUR MESSEN

Jetzt kann es mit der Messung auch schon losgehen. Kopiere dir den folgenden Sketch und lade ihn auf deinen Arduino hoch:

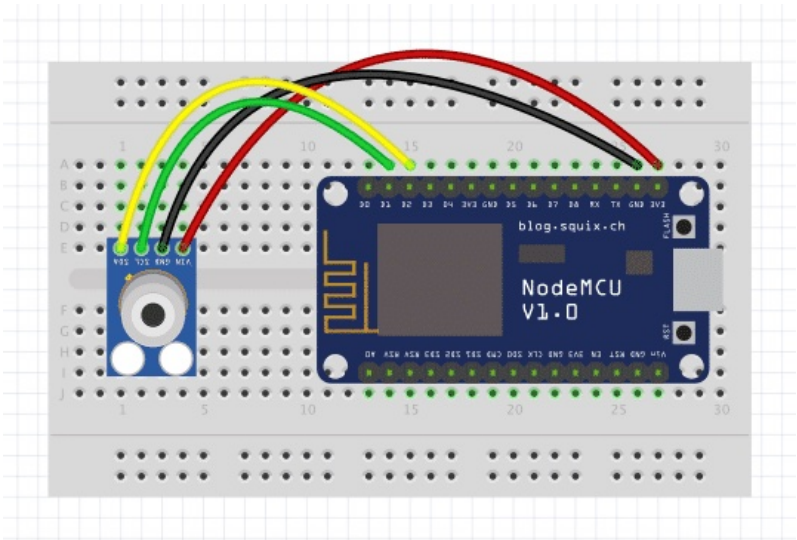
```
#include <Wire.h>
#include <Adafruit_BMP085.h>

Adafruit_BMP085 bmp;

void setup() {
  Serial.begin(115200);
  if (!bmp.begin()) {
      Serial.println("Sensor nicht gefunden!");
      while (1) {}
  }
```

```
}

void loop() {
    Serial.print("Temperatur = ");
    Serial.print(bmp.readTemperature());
    Serial.println(" *C");

    Serial.println();
    delay(2000);
}
```

In deinem Seriellen Monitor sollten jetzt alle 2 Sekunden die Messwerte für die Temperatur in °C erscheinen. Sollte nichts erscheinen, **überprüfe, ob die Baudrate im Sketch und im Seriellen Monitor übereinstimmt oder du SCL und SDA versehentlich verwechselt hast.**

## DEN GY-906 ANSCHLIESSEN

Im Prinzip schließt du diesen Sensor genauso an wie den BMP180 – er kommuniziert ebenfalls über I²C. Die Skizze für den Anschluss sieht also ganz ähnlich aus:




## DIE PASSENDE BIBLIOTHEK

Auch für den GY-906 gibt es eine Bibliothek, die dir das Leben einfacher macht. Suche im Bibliotheksmanager nach **Adafruit_MLX90614** und installiere die aktuelle Version.

**MLX90614** bezieht sich auf den Sensor selbst – GY-906 ist hingegen die Bezeichnung des ganzen Bauteils.

## DIE TEMPERATUR MESSEN

Kopiere dir den folgenden Sketch und lade ihn auf deinen Arduino hoch:

```
#include <Adafruit_MLX90614.h>
#include <Wire.h>
```

```
Adafruit_MLX90614 mlx = Adafruit_MLX90614();

void setup() {
  Serial.begin(115200);
  mlx.begin();
}

void loop() {
  Serial.print("Umgebung = ");
Serial.print(mlx.readAmbientTempC());
  Serial.print("*C\tObjekt = ");
Serial.print(mlx.readObjectTempC());
  Serial.println("*C");

  Serial.println();
  delay(2000);
}
```

Wie du siehst, bindest du zu Beginn des Sketchs die Bibliotheken für den Sensor und die Kommunikation per I²C ein. Anschließend erstellst du ein Objekt der Bibliothek mit dem Namen **mlx**.

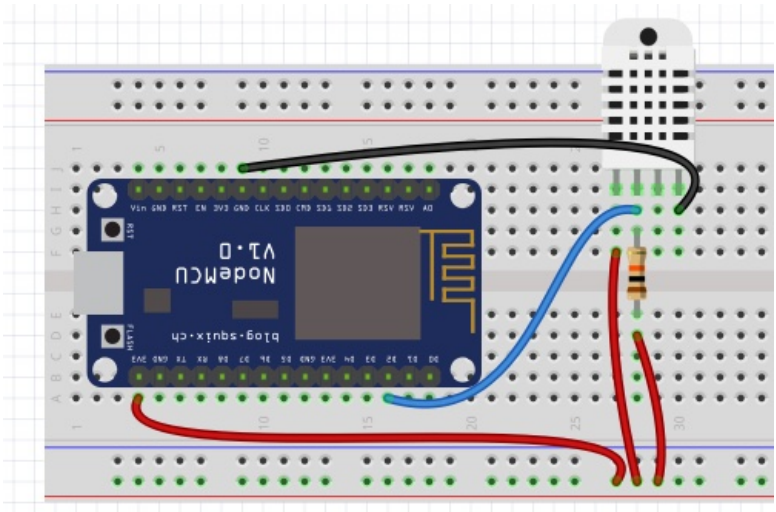
Sicherlich weißt du, dass der GY-906 eine Besonderheit hat (was ihn auch etwas teurer als andere Sensoren macht): Er misst nicht nur die Umgebungstemperatur, sondern per Infrarot auch jene von Objekten.

Deshalb befinden sich im Sketch zwei Abfragen – eine für die Temperatur um den Sensor herum – **mlx.readAmbientTempC()**

– und eine für das Objekt “vor seiner Nase” – **mlx.readObjectTempC**. Welche Temperatur später bei deinem Webserver zum Einsatz kommen soll, ist natürlich dir überlassen.

## DEN DHT22 ANSCHLIESSEN

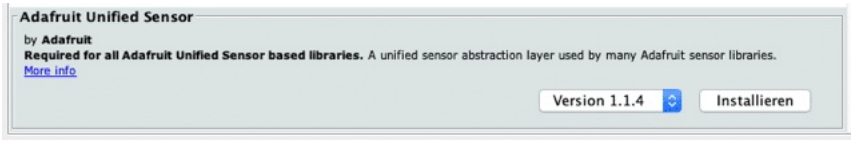
Etwas aufwändiger ist der Anschluss des Sensors DHT22, denn hierfür benötigst du einen 10 kΩ Widerstand. Orientiere dich beim Anschluss an dieser Skizze:



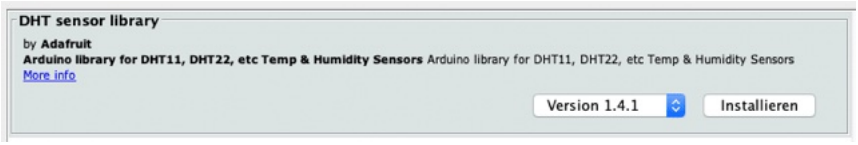

## DIE PASSENDEN BIBLIOTHEKEN

Für den DHT22 musst du zwei Bibliotheken installieren, von denen du jedoch nur eine im Sketch einbinden musst. Öffne deinen Bibliotheksmanager. Suche dort zunächst nach **Adafruit Unified Sensor** und installiere die aktuelle Version. Die

Versionsnummer in den folgenden Screenshots können abweichen.




Suche anschließend nach **DHT22** und installiere die Bibliothek **DHT sensor library**.




## DIE TEMPERATUR MESSEN

Kopiere dir den folgenden Sketch und lade ihn auf deinen Arduino hoch:

```
#include "DHT.h"

#define DHTPIN 4
#define DHTTYPE DHT22

float temp;

DHT dht(DHTPIN, DHTTYPE);
```

```
void setup() {
  Serial.begin(115200);
  dht.begin();
}

void loop() {

  temp = dht.readTemperature();

  Serial.print("Temperatur: ");
  Serial.print(temp);
  Serial.println("*C");

  Serial.println();
  delay(2000);
}
```

Nachdem du die Bibliothek eingebunden hast, legst du den Pin fest, an dem der DHT22 angeschlossen ist. In unserem Sketch ist das der **Pin 4**. Der Sensor ist allerdings am ESP8266 an **Pin D2** angeschlossen. Wieso dann die unterschiedlichen Zahlen?

Scrolle noch einmal nach oben zum Pin-Diagramm des ESP8266. **Wie du dort sehen wirst, entspricht der Pin D2 dem GPIO 4 – diese Ziffer kommt im Sketch zum Einsatz.**

In der nächsten Zeile legst du das Modell des Sensors fest – in unserem Fall also ein DHT22. Anschließend erstellst du ein

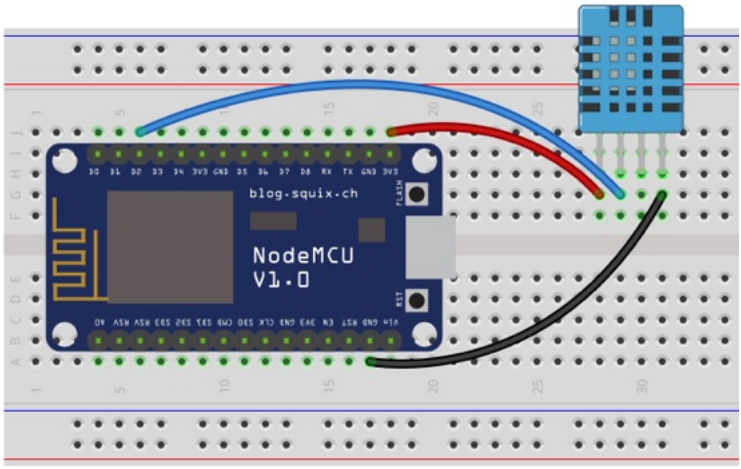
Objekt der Bibliothek names **dht**, das später bei der Messung mit der Funktion **dht.readTemperature()** zum Einsatz kommt.

Der Rest des Sketchs dürfte für dich kein Problem sein. **Achte jedoch darauf, dass die Baudrate von Sketch und Seriellem Monitor übereinstimmt.**

## DEN DHT11 ANSCHLIESSEN

Der Sensor DHT11 ist so etwas wie der “kleine Bruder” des DHT22. Er hat einen kleineren Messbereich und ist auch etwas ungenauer.

Du hast ihn ja eventuell schon am Arduino angeschlossen. Orientiere dich beim Anschluss an deinem ESP8266 an folgender Skizze:



In obigen Beispiel-Sketch musst du für den DHT11 nur eine Zeile anpassen:

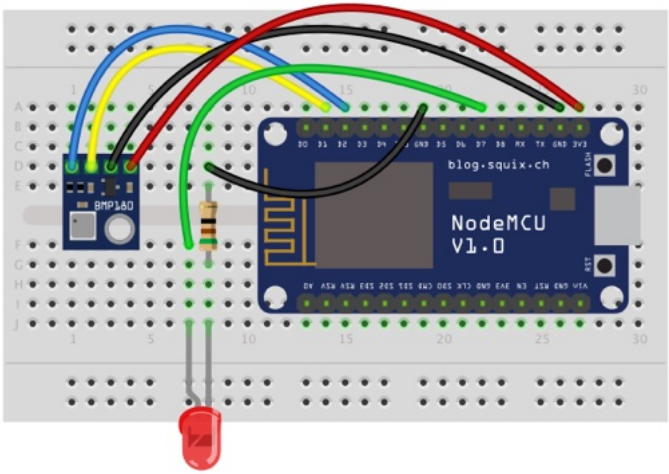
```
#define DHTTYPE DHT11
```

## EINE LED ANSCHLIESSEN

Du hast nun einen Temperatursensor, dessen Messdaten du später über deinen Webserver auslesen wirst. Aber so ein Webserver soll natürlich keine Einbahnstraße sein: Ebenso hast du die Möglichkeit, an deinen ESP8266 ein Gerät anzuschließen, das du vom Smartphone aus steuerst.

**Hier sind deiner Phantasie kaum Grenzen gesetzt.** Vielleicht möchtest du die Rollläden hoch- und herunterlassen oder die Heizung an- und ausschalten. Oder einfach nur das Licht.

In diesem Abschnitt und im weiteren Verlauf des Kurses beschäftigen wir uns mit einem “Gerät”, das du sicherlich schon oft verwendet hast: eine LED. Erweitere also zunächst dein Projekt auf dem Breadboard um eine LED samt Vorwiderstand. Orientiere dich hierbei an folgender Skizze:




Als Beispiel soll uns eine rote Standard-LED mit einer Spannung von 2,3V und 20mA Stromfluss dienen. Da dein ESP8266 3,3V “liefert”, ergibt sich hieraus ein Vorwiderstand von 51Ω.

Verbinde die Anode (langes Bein) der LED mit dem Pin D7 des ESP8266, damit du sie mit den späteren Sketches dieses Kurses ohne Bearbeitungen steuern kannst. Die Kathode kommt mit dem Vorwiderstand dazwischen an GND.

## SO STEUERST DU “GROSSE” GERÄTE

Die LED in diesem Abschnitt dient natürlich nur dem Verständnis. Sicherlich möchtest du stattdessen Geräte steuern, die mit 230 Volt versorgt werden. Auch das kannst du mit deinem ESP8266 umsetzen – allerdings benötigst du hierfür ein Relais.

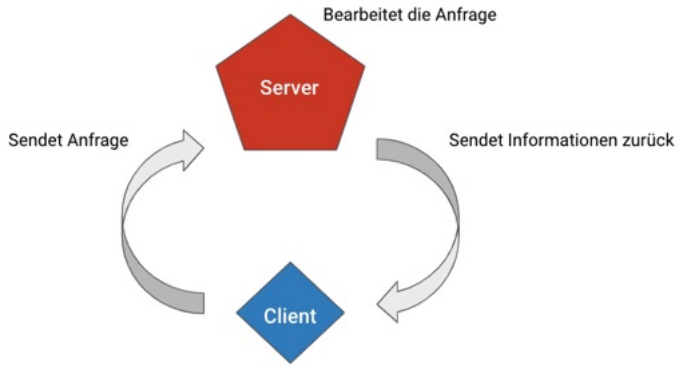
**Achte beim Kauf darauf, dass dieses mit den 3,3V des ESP8266 funktioniert.** Viele der erhältlichen Relais benötigen 5V. Solltest du einen Wemos D1 Mini verwenden, kannst du sogar ein praktisches Relais-Shield darauf montieren.

Damit hast du die Grundlagen abgeschlossen. In den nächsten Abschnitten beschäftigen wir uns mit dem eigentlichen Webserver.

## DEN WEBSERVER EINRICHTEN UND STEUERN

Was ist eigentlich ein Server und was macht er? Lass uns dieses Thema zunächst ganz kurz betrachten.

Ein Server ist ein Computer – oder in unserem Fall ein Microcontroller – der von einem anderen Computer Anfragen nach Informationen erhält. Diese Anfragen bearbeitet er und wenn er die angeforderten Informationen besitzt, sendet er sie zurück an den Client.



In diesem Projekt dient dein ESP8266 als Server. Dein Client kann dann zum Beispiel das Gerät sein, auf dem du diesen Text gerade liest – also ein PC, Laptop, Tablet oder Smartphone.

Informationen können dann zum Beispiel “bloße” Messdaten sein oder auch eine Webseite, die diese Messdaten enthält.

**Aber nicht nur das:** Dein Webserver kann ebenso Befehle erhalten, die deinen ESP8266 dann etwas steuern lassen. So kannst du zum Beispiel die im vorherigen Abschnitt erwähnte LED an- oder ausschalten. Den neuen Status der LED (also AN oder AUS) kannst du dir dann zurücksenden lassen.

In den folgenden Abschnitten tasten wir uns langsam an dieses Thema heran. Zunächst starten wir, wie man jedes ordentliche Projekt startet: Mit einem “Hello world!” Danach fragen wir die Temperatur als blanke Zahl ab. Anschließend hübschen wir diese Messdaten mithilfe von HTML auf und bauen uns ein Interface.

Zuletzt wirst du selbst aktiv und schaltest das Licht über dieses Interface an und aus.

## HELLO WORLD!

Zeit, um mit unserem Webserver zu beginnen. Zunächst lassen wir den Temperatursensor und die LED noch links liegen. Stattdessen stellt dein ESP8266 auf Anfrage eine Webseite bereit, die erst einmal nichts als ein einfaches “Hello World” enthält.


Damit du diese Webseite aufrufen kannst, teilt dir dein ESP8266 im Seriellen Monitor die IP-Adresse mit, die du in einem Browser deiner Wahl aufrufen kannst.

**Hinweis:** Beachte bitte, dass du auf deinen Webserver nur zugreifen kannst, wenn sich dein Client im gleichen WLAN-Netzwerk befindet wie dein ESP8266. Darüber, wie du von überall auf der Welt auf deinen Server zugreifen kannst – und ob du das überhaupt solltest, sprechen wir am Ende dieses Projekts.

## DER ANFANG DES SKETCHS

Du hast bereits gelernt, wie du deinen ESP8266 mit dem Internet verbindest. Ohne Netzwerk kein Server – deshalb benötigen wir auch in allen folgenden Sketches dieses Kurses eine Verbindung zu deinem WLAN.

Hier noch einmal in aller Kürze. Diesen Code benötigst du am Anfang des Sketchs:

```
//Bibliothek für die WLAN-Verbindung
```

```
#include <ESP8266WiFi.h>

//Zugangsdaten zu deinem WLAN-Netzwerks
const char* ssid     = "SSID";
const char* password = "PASSWORT";
```

Anschließend legst du den Port deines Webservers auf 80 fest. **Dieser Port ist der Teil der Netzwerk-Adresse an der dein Webserver auf eingehende Anfragen von Clients wartet.**

```
//Port des Webservers auf 80 setzen
WiFiServer server(80);
```

## DAS SETUP

Hier startest du zuerst den Seriellen Monitor. Achte später darauf, dass du dort dieselbe Baudrate eingestellt hast wie im Sketch.

```
Serial.begin(115200);
```

Anschließend stellt dein ESP8266 die Verbindung zu deinem WLAN-Netzwerk her:

```
WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Ich verbinde mich mit dem
Internet...");
  }
  Serial.println("Ich bin mit dem Internet
verbunden!");
```

Da dein ESP8266 nun mit dem Netzwerk verbunden ist, hat er dort auch eine IP-Adresse. Diese benötigst du später, um eine Anfrage an den Server zu schicken. Damit du weißt, wie sie lautet, gibst du sie im Seriellen Monitor aus und startest danach den Server:

```
Serial.println("IP-Adresse: ");
Serial.println(WiFi.localIP());

server.begin();
```

## DER LOOP

Die bisherigen Einstellungen und Funktionen waren noch recht überschaubar. Im Loop findet nun die gesamte Kommunikation zwischen Server und Client statt.

Zunächst benötigen wir eine Zeile Code, mit der der ESP8266 nach eingehenden Anfragen “horcht”:

```
WiFiClient client = server.available();
```

Ob eine Anfrage eintrifft, fragst du mit einem If-Statement ab:

```
if (client) {
```

Nur dann – also wenn du die IP-Adresse in deinem Browser aufrufst – wird der weitere Code in **void loop()** ausgeführt. Zunächst gibst du eine entsprechende Meldung im Seriellen Monitor aus und erstellst einen **String**, in dem du die ankommenden Daten der Anfrage zeilenweise speicherst und prüfen wirst, ob sie zu Ende ist.

```
Serial.println("Anfrage von Client erhalten.");
String currentLine = "";
```

Solange der Client mit dem Server verbunden ist und Daten von ihm “reinkommen”, speicherst du diese Byte für Byte in der Variablen **c** und gibst sie im Seriellen Monitor aus.

```
while (client.connected()) {

   if (client.available()) {
     char c = client.read();
     Serial.write(c);
```

Dort kannst du die Anfrage Zeile für Zeile nachverfolgen. Apropos Zeilen: Interessant sind hier die Zeilenumbrüche und Leerzeilen. Wenn dein Server das Zeichen für eine **New Line \n** erhält, prüfst du, ob die Anfrage zu Ende ist.

Das ist etwas kompliziert, denn eine Anfrage endet immer mit einer Leerzeile, was nicht nur einem **\n**, sondern der Zeichenkombination **\n\n** entspricht. Aber eins nach dem anderen. Zunächst prüfst du ob ein **\n** hereinkommt:

```
if (c == '\n') {
```

oder etwas anderes außer einem **Carriage Return \r**. All diese “anderen” Bytes (also alles außer \n und \r) speicherst du in deiner Variablen für die aktuelle Zeile **currentLine**.

```
else if (c != '\r') {
  currentLine += c;
  }
```

Wenn das aktuelle Byte **c** der Anfrage jedoch **\n** lautet, greift das oben erwähnte If-Statement. Gleich danach folgt eine weitere Prüfung, ob die Variable **currentLine** einen Inhalt hat:

```
if (currentLine.length() == 0) {
```

Ist das nicht der Fall, kam offensichtlich noch kein **\n\n** – also noch keine Leerzeile. Deshalb wird die Variable **currentLine** wieder gelöscht und der Sketch springt zurück zur nächsten Zeile der Anfrage des Clients:

```
else {
  currentLine = "";
  }
```

Wenn jetzt jedoch wieder ein **\n** kommt, haben wir unsere Leerzeile – oder anders gesagt: **Die Anfrage des Clients ist zu Ende!** Dieses Prozedere mit den verschiedenen Schleifen ist auf den ersten Blick recht kompliziert – nimm dir deshalb ruhig Zeit und verfolge die Durchgänge im gesamten Sketch.

## DER SERVER ANTWORTET

Nun ist also der Server mit seiner Antwort an den Client dran. Da mit deinem Webserver alles in Ordnung ist, lässt du ihn mit einem entsprechenden Status-Code antworten – nämlich 200:

```
client.println("HTTP/1.1 200 OK");
```

Es folgen Informationen zur Art des Contents, der gleich geliefert wird und darüber, dass der Server die Verbindung zum Client kappen wird, sobald er mit seiner Antwort fertig ist. Gefolgt von einer Leerzeile:

```
client.println("Content-type:text/html");
client.println("Connection: close");
client.println();
```

Nun kommt endlich unser **“Hello world!”**. Und zwar als HTML:

```
client.println("<!DOCTYPE html><html>");
client.println("<head><meta name=\"viewport\"
content=\"width=device-width,
initial-scale=1\"></head>");
client.println("<body><h1 align=\"center\">Hello
world!</h1></body></html>");
```

Sehr reduziertes HTML – abgesehen vom **<head>** befindet sich im **<body>** nur eine in deinem Browser zentrierte Überschrift **<h1>** mit unserer Nachricht.

Später erfährst du mehr über HTML ein und baust dir damit ein ansprechendes Interface, über das du mit deinem ESP8266 interagieren kannst.

An dieser Stelle war es das jedoch. Fast. Fehlt nur noch das Ende der Verbindung und eine Info hierüber in deinem Seriellen Monitor:

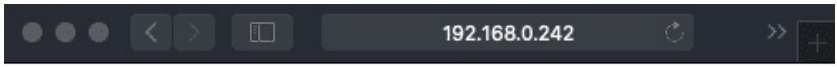
```
client.stop();
Serial.println("Verbindung beendet.");
Serial.println("");
```

Den vollständigen Sketch findest du auf polluxlabs.net/maker-buch

## WIE WARM IST ES GERADE?

Im letzten Abschnitt hast du einen Webserver in seiner einfachsten Form erstellt und ein einfaches **Hello world!** auf einer ebenso einfachen Webseite ausgegeben.

Lass uns nun einen Schritt weiter gehen. In diesem Abschnitt ermittelst du mithilfe deines Sensors die aktuelle Raumtemperatur. Wenn du die die IP-Adresse des Servers aufrufst erscheint diese dann auf der Webseite.




# Die aktuelle Temperatur

22.90 °C

## PASSE DEN SKETCH AN

Die gute Nachricht: Du kannst den Großteil des Sketchs aus dem letzten Abschnitt einfach übernehmen. Hinzufügen musst du lediglich etwas Code für die Messung der Temperatur. Ebenso bedarf es ein paar Anpassungen der Webseite, die dein Server an deinen Client übermittelt.

Zunächst zur Temperatur. In diesem Abschnitt arbeiten wir weiter mit dem Sensor BMP180. Füge zu Beginn des Sketchs folgende Zeilen hinzu:

```
#include <Adafruit_BMP085.h>
Adafruit_BMP085 bmp;
float temp;
```

Mit der ersten Zeile integrierst du die Bibliothek für den Sensor. Die zweite erstellt das Objekt **bmp**. Anschließend benötigen wir noch eine Variable für die gemessene Temperatur.

Im Setup prüfst du, ob der Sensor verfügbar ist. Ist das nicht der Fall, begibt sich dein Sketch in eine Endlosschleife und wird nicht weiter ausgeführt:

```
if (!bmp.begin()) {
  Serial.println("Sensor nicht gefunden!");
  while (1) {}
}
```

Wir gehen natürlich davon aus, dass das nicht der Fall ist. Deshalb folgt nun die eigentliche Messung. Diese integrierst du im Loop – mitten in der Antwort des Servers an den Client. **Das hat den Vorteil, dass du immer die aktuelle Temperatur erhältst, sobald du eine neue Anfrage stellst, also zum Beispiel einen Refresh im Browser machst.**

```
temp = bmp.readTemperature();
Serial.print("Temperatur = ");
Serial.print(temp);
Serial.println(" *C");
```

## ETWAS MEHR HTML FÜR DEINE WEBSEITE

Jetzt musst du nur noch die Antwort des Servers – also die Webseite – etwas anpassen. Zunächst fügst du dem **<head>** der Webseite die Zeichencodierung UTF-8 hinzu, damit das Grad-Zeichen ° korrekt dargestellt wird:

```
client.println("<meta charset=\"utf-8\"></head>");
```

Anschließend ersetzt du die Headline **Hello world!** durch **Die aktuelle Temperatur** und fügst direkt darunter einen neuen Absatz hinzu, der den Messwert deines Temperatursensors in der Variablen **temp** enthält:

```
client.println("<body><h1 align=\"center\">Die
aktuelle Temperatur</h1>")
client.println("<p align=\"center\">");
client.println(temp);
client.println("°C</p>");
client.println("</body></html>");
```

## TESTE DEINE NEUE WEBSEITE

Lade den folgenden Sketch auf deinen ESP8266 und rufe die IP-Adresse im Browser auf. Siehst du die aktuelle Temperatur?

Teste auch, ob sich die Temperatur verändert, wenn du deinen ESP8266 samt Sensor an einen anderen Ort stellst und eine neue Anfrage schickst.

Im nächsten Abschnitt integrierst du eine Funktion, die dir etwas “Arbeit” abnimmt – du musst die Webseite nicht mehr manuell aktualisieren, sondern sie macht das von alleine.

Den vollständigen Sketch findest du auf **polluxlabs.net/maker-buch**

## AUTORELOAD DER WEBSEITE

Wie wäre es, wenn du die Webseite mit der Antwort des Servers nicht selbst neu laden müsstest, um die aktuelle Temperatur zu erfahren? Kein Problem!

In diesem Abschnitt lernst du zwei Methoden kennen, die genau das für dich übernehmen.

### AUTOREFRESH MIT EINEM META-TAG

Die erste Methode erfordert nur eine kleine Anpassung im HTML der Antwort deines Servers. Erweitere einfach den **Head** des HTML um folgende Information:

```
<meta http-equiv=\"refresh\" content=\"5\">
```

Der gesamte Head sollte dann wie folgt aussehen:

```
client.println("<head><meta name=\"viewport\"
```

```
content=\"width=device-width, initial-scale=1\">");
client.println("<meta charset=\"utf-8\"><meta
http-equiv=\"refresh\" content=\"5\"></head>");
```

**Jetzt lädt deine Webseite alle 5 Sekunden neu** – und stellt somit eine neue Anfrage an deinen ESP8266, der daraufhin die Temperatur ermittelt und zurückgibt.

Das funktioniert im Prinzip ganz gut – allerdings kannst du damit einen Refresh höchstens einmal pro Sekunde ausführen. Wenn du häufigere Anfragen möchtest, um deine Daten quasi in Echtzeit zu erhalten, eignet sich ein kleines Script besser.

## AUTOREFRESH MIT JAVASCRIPT

Mit dem folgenden Script kannst du deine Webseite häufiger als einmal pro Sekunde neu laden – theoretisch sogar jede tausendstel Sekunde. Das wäre jedoch sicherlich etwas zu viel des Guten. Füge also statt des Meta-Tags von oben folgendes Script deinem Head hinzu:

```
<script>
  function refresh(refreshPeriod)
  {setTimeout('location.reload(true)',
refreshPeriod);}
  window.onload = refresh(5000);
 </script>
```

Der Head sollte dann in deinem Sketch so aussehen:

```
client.println("<head><meta name=\"viewport\"
content=\"width=device-width, initial-scale=1\">");
client.println("<meta charset=\"utf-8\">");
client.println("<script>function
refresh(refreshPeriod){setTimeout('location.reload(tru
e)', refreshPeriod);}");
client.println("window.onload =
refresh(500);</script></head>");
```

Jetzt aktualisiert sich deine Webseite alle 500 Millisekunden von alleine. Diese Information findest du in **refresh(500)** – probiere es gleich einmal aus und ersetze diesen Wert durch andere. Siehst du, wie schnell sich die Temperatur im Browser verändert (vorausgesetzt sie verändert sich wirklich um deinen Sensor herum)?

## DIE LED STEUERN

Du kannst nun also die aktuelle Raumtemperatur auf der Webseite sehen, die dein Server dir unter seiner IP-Adresse zur Verfügung stellt. Wie wäre es, wenn du von dort aus auch das Licht – also deine LED – ein- und ausschalten könntest? Genau das setzen wir jetzt um.

Zunächst benötigen wir etwas Code für die LED. Zu Beginn des Sketchs legst du den Pin fest, an dem sie angeschlossen ist. **Wir bleiben hier bei Pin D7 – den wir im Sketch mit der Zahl 13**

**ansteuern.** Ebenso benötigen wir eine Variable für den Status der LED, also ob sie an oder aus ist:

```
const int led = 13;
String ledState = "aus";
```

Im Setup des Sketchs legst du nun noch schnell den **pinMode** fest und schaltest die LED zu Beginn aus, so wie du es oben in der Variablen **ledState** auch festgehalten hast.

```
pinMode(led, OUTPUT);
digitalWrite(led, LOW);
```

Wie kannst du nun die LED von der Webseite aus an- und ausschalten? Hierfür bringst du dort zunächst einen Link unter, der deinen Befehl weiterleitet. Wenn die LED aus ist (also ledState auf “aus” steht), lautet der Ankertext des Links **EINSCHALTEN**. Wenn sie an ist entsprechend **AUSSCHALTEN**.

```
if (ledState == "aus") {
  client.println("<p align=\"center\"><a
href=\"/led/on\">ANSCHALTEN</a></p>");
} else {
  client.println("<p align=\"center\"><a
href=\"/led/off\">AUSSCHALTEN</a></p>");
  }
```

**Ein kurzer Exkurs zu HTML:** Ein Link wird hier mit dem Tag <a> begonnen und mit </a> wieder geschlossen. Zwischen diesen beiden Tags befindet sich der Ankertext, also das Wort, das du anklicken kannst.

Das Ziel des Links befindet sich im öffnenden Tag <a> hinter **href=** und wird mit Anführungsstrichen markiert. Im Code oben heißt das Linkziel also entweder “/led/on” oder “/led/off”.

Übrigens, sind dir die umgedrehten Schrägstriche (Backslashes) aufgefallen? Diese benötigst du in deinem Sketch, um die Anführungszeichen zu maskieren. Ansonsten würde z.B. im Befehl **client.println()** oben zu viele Anführungszeichen stehen, was unweigerlich zu einem Fehler führen würde.

Wenn du die IP-Adresse deines Servers nun aufrufst, sieht du folgende Webseite:

Aber noch einmal zurück zum Linkziel: Wenn du also den Link **ANSCHALTEN** klickst, wird "/led/on" hinter die IP-Adresse des Servers geschrieben, sodass sie z.B. so lautet:

```
http://192.168.0.242/led/on
```

Das können wir uns zunutze machen, wenn dein Server die Anfrage des Clients liest.

## DEINEN BEFEHL IM HEADER AUSLESEN

Wie du ja bereits weißt, sendet dein Client eine ganze Reihe Informationen an deinen Server, wenn er eine Anfrage an ihn stellt. Nach deinem Klick auf den Link auch den Befehl **/led/on** bzw. **/led/off**.

Um diese Befehle zu empfangen, benötigst du zunächst zu Beginn des Sketchs eine gleichnamige Variable für den Header:

```
String header;
```

Wenn eine Anfrage reinkommt, verwendest du ja bereits die Variable **c**, um zu ermitteln, wann die Anfrage zu Ende ist. Hier integrierst du nun auch deinen String **header**, den du mit den übertragenen Daten füllst:

```
if (client.available()) {
  char c = client.read();
  Serial.write(c);
  header += c;
  if (c == '\n') {
```

Nun befindet sich im String **header** (nach deinem Klick auf den Link) entweder der Befehl **/led/on** oder **/led/off**.

Um herauszufinden welcher, machst du eine kleine Abfrage:

```
if (header.indexOf("GET /led/on") >= 0) {
  Serial.println("Die LED ist an");
  ledState = "an";
  digitalWrite(led, HIGH);
} else if (header.indexOf("GET /led/off") >= 0) {
  Serial.println("Die LED ist aus");
  ledState = "aus";
  digitalWrite(led, LOW);
}
```

Hier benutzt du die Funktion **indexOf()** – diese prüft, ob sich ein bestimmter String (hier z.B. “GET /led/on”) in einem anderen String befindet. Ist das nicht der Fall, ist ihr Ergebnis **-1**. Wenn sie ihn jedoch findet, gibt sie seine Position zurück. Diese ist uns eigentlich egal, Hauptsache das Ergebnis ist nicht **-1**, sondern **>= 0**.

Wenn du also die LED mit einem Klick angeschaltet hast, steht im **header** der String “GET /led/on”. In diesem Fall schaltest du die LED mit **digitalWrite()** an und setzt die Variable **ledState** auf “an”.

Das wiederum verändert den Ankertext des Links: Dieser wechselt von **ANSCHALTEN** auf **AUSSCHALTEN**. Und damit wirklich klar ist, was Sache ist, schreibt dein Server noch den Status der LED über den Link:

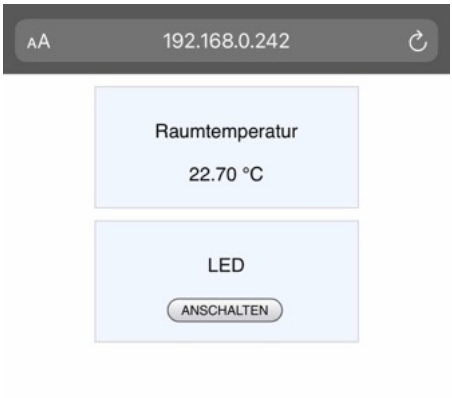
```
client.println("<p align=\"center\">Die LED ist " +
ledState + ".</p>");
```

Und das war es auch schon. Probiere es gleich mal aus. Kannst du die LED über die Webseite an- und ausschalten?

Den vollständigen Sketch findest du auf polluxlabs.net/maker-buch

## CSS UND HTML FÜR EINE SCHÖNERE WEBSEITE

Du hast nun eine einfache Webseite, auf der du die aktuelle Temperatur siehst und die LED steuern kannst. Allerdings sieht sie sehr nach den Anfängen des World Wide Webs aus. Lass uns also ein paar Minuten Zeit investieren und sie etwas aufmöbeln. Am Ende dieses Abschnitts sieht deine Webseite folgendermaßen aus:




Du siehst zwei Karten – eine für die Temperatur und eine für die Steuerung der LED. Statt eines einfachen Links befindet sich auf letzterer ein Button. Beachte bitte, dass sich das Aussehen des Buttons je nach Browser verändern kann.

## DAS CSS

Bisher haben wir kaum CSS (Cascading Style Sheets) verwendet. Das ändert sich jetzt, aber keine Angst – es hält sich trotzdem im Rahmen. Folgende Informationen benötigen wir für die Webseite:

```
<style>
    html {
      font-family: Helvetica;
    }

    h1,
    p {
      text-align: center;
    }
```

```
    a {
      color: black;
      text-decoration: none;
    }

    div {
      margin: 10px auto 10px auto;
      padding: 10px;
      height: 80px;
      width: 200px;
      background-color: aliceblue;
      border: 1px solid lightgray;
    }
  </style>
```

Allerdings ahnst du es vielleicht bereits, diese Informationen zum Styling der Webseite musst du im Sketch etwas anders unterbringen – nämlich so wie du es mit den anderen HTML-Tags auch gemacht hast: Innerhalb der verschiedenen **client.println()** Funktionen. **Achte bitte darauf, folgende Zeilen noch vor dem schließenden </head> Tag unterzubringen.**

```
client.println("<style>");
client.println("html { font-family: Helvetica;} h1, p
{text-align: center;}");
client.println("a { color: black; text-decoration:
none;}");
client.println("div { margin: 10px auto 10px auto;
padding: 10px; height: 80px; width: 200px;
background-color: aliceblue; border: 1px solid
lightgray;}");
```

```
client.println("</style>");
```

## DAS HTML

Den Head der Webseite hast du nun um das neue Styling erweitert. Nun kommt der Body dran. Folgendes HTML musst du hier integrieren:

```
<div>
  <p>Raumtemperatur</p>
  <p> (temp) </p>
</div>
<div>
  <p>LED</p>
  <button style="margin-left:50px;"><a
href="/led/on">ANSCHALTEN</a>
  </button>
  <button style="margin-left:45px;"><a
href="/led/off">AUSSCHALTEN</a>
  </button>
</div>
```

Die Variable **temp** siehst du im obigen Code fett gedruckt – dort dient sie nur als Erinnerung, dass sie dort hinein muss. In deinem Sketch sieht dieses HTML dann so aus:

```
client.println("<body><div><p>Raumtemperatur</p>");
client.println("<p>");
```

```
client.println(temp);
client.println("°C</p></div>");

client.println("<div><p>LED</p>");

if (ledState == "aus") {
  client.println("<button
style=\"margin-left:50px;\"><a
href=\"/led/on\">ANSCHALTEN</a></button>");
} else {
  client.println("<button
style=\"margin-left:45px;\"><a
href=\"/led/off\">AUSSCHALTEN</a></button>");
}

client.println("</div></body></html>");
```

Wie du siehst, befindet sich hier wieder die If-Abfrage, die je nachdem, ob die LED an oder aus ist, einen anderen Button ausspielt. Das Styling für die Buttons – genauer gesagt ihre Abstände nach links – befinden sich direkt im <button> Tag. Achte in diesem Fall bitte auf die korrekte Maskierung der Anführungsstriche mit dem Backslash \.

Und das war es schon. Lade den Sketch mit dem aktualisierten CSS und HTML auf deinen ESP8266 und schau dir deine neue Webseite gleich einmal an.

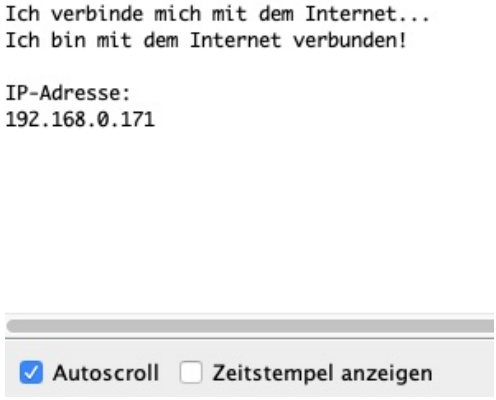
Den vollständigen Sketch findest du auf **polluxlabs.net/maker-buch**

## EINE FESTE IP-ADRESSE VERGEBEN

Wechselt die IP-Adresse deines ESP8266 nach jedem Neustart? Das kannst du leicht beheben, indem du ihm in deinem Sketch eine feste IP zuweist. Hierfür benötigst du nur ein paar Zeilen Code.

**Hinweis:** Die IP-Adresse, die du verwenden möchtest, muss in deinem Netzwerk natürlich noch verfügbar sein und sich im entsprechenden Gateway befinden.

Nehmen wir an, dein ESP8266 hat bisher die Adresse 192.168.0.242 zugewiesen bekommen und ist darunter erreichbar. Du möchtest diese nun jedoch manuell auf 192.168.0.171 festlegen.



Um diese IP-Adresse festzulegen, füge deinem Sketch noch vor der Setup-Funktion folgende Zeilen hinzu. Hier legst du die IP-Adresse 192.168.0.171 im Gateway 192.168.0.1 fest:

```
IPAddress local_IP(192, 168, 0, 171);
IPAddress gateway(192, 168, 0, 1);

IPAddress subnet(255, 255, 0, 0);
```

Innerhalb der Setup-Funktion verwendest du diese Daten nun für die Konfiguration. Achte hierbei darauf, folgende Zeile noch vor der Funktion **WiFi.begin()** unterzubringen:

```
WiFi.config(local_IP, gateway, subnet);
```

Nach dem Upload besitzt dein ESP8266 nun die oben festgelegte IP-Adresse in deinem Netzwerk.

## HINWEISE ZUR IT-SICHERHEIT

Zum Ende dieses Buchs möchten wir noch auf ein wichtiges Thema hinweisen: IT-Sicherheit.

Den Webserver, den du gebaut hast, kannst du zunächst nur aus deinem eigenen WLAN-Netzwerk ansteuern. Und das ist auch erstmal ganz gut so. Denn so können nur Menschen mit

dem passenden Schlüssel auf die Daten des Servers zugreifen – **vorausgesetzt, dein WLAN-Netzwerk ist sicher.**

Zunächst möchten wir dich darauf hinweisen, dass die gängigen ESP8266-Module **nur für Hobbyprojekte** ausgelegt sind. Das heißt im Umkehrschluss, dass du **niemals sicherheitsrelevante Informationen** damit verarbeiten solltest.

Das gilt insbesondere, wenn du auf deinen Webserver auch von außerhalb deines eigenen WLAN-Netzes zugreifen möchtest. **Dann könnte jeder auf ihn zugreifen – nicht nur du.** Natürlich gibt es auch hierfür diverse Sicherheitsmaßnahmen, mit denen du das zumindest erschweren kannst – aber dennoch, **diese solltest du nur umsetzen, wenn du genau weißt, was du tust.**

An dieser Stelle können wir dir leider keine weiteren Hinweise und Sicherheitslösungen an die Hand geben, da dieses Thema den Rahmen dieses Buchs sprengen würde und sich auch rasant weiterentwickelt.

Für den Einstieg in dieses Thema können jedoch diese zwei Artikel dienen:

**Sicherheit? Sicherheit!**
(az-delivery.de/blogs/azdelivery-blog-fur-arduino-und-raspberry-pi/entwurf-sicherheit-sicherheit)

**Sicherheit in der IoT**
(az-delivery.de/blogs/azdelivery-blog-fur-arduino-und-raspberry-pi/sicherheit-in-der-iot)

# WIE GEHT ES WEITER?

Du hast nun eine ganz schöne lange Reise mit deinem Arduino und ESP8266 hinter dir! Wenn du ohne Vorkenntnisse mit deiner ersten Zeile C++ begonnen hast und jetzt deinen eigenen Webserver vor dir stehen hast, kannst du sehr stolz auf dich sein.

Aber natürlich ist hier nicht Schluss. Sicherlich hast du bereits viele Ideen für deine nächsten Maker-Projekte entwickelt. Falls dir noch etwas Inspiration fehlt, freuen wir uns auf deinen Besuch auf polluxlabs.net

Wir wünschen dir viel Erfolg bei all deinen Projekten!