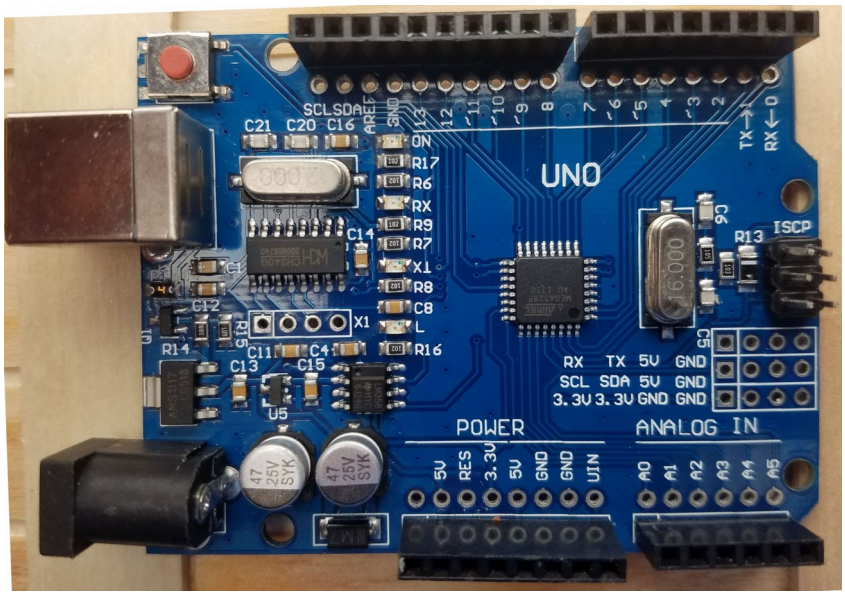
# Arduino4Kids

## 3x3x3 Cube




Wilfried Klaas

Stand: 31. Mrz 2019

# Inhalt

# Einleitung

## Was ist ein Arduino?

Arduino ist ein OpenSource Projekt (Quelloffenes Projekt), wobei alle Bereiche, sowohl Hardware wie auch Software offen sind. D.h. jeder kann und darf an dem Projekt mitarbeiten, es verwenden, Produkte dafür entwerfen… Arduino ist nicht nur die Hardware sondern auch die Software und die IDE. (Entwicklungsumgebung, also da wo du dein Programm rein schreibst).

Du kannst die verschiedensten Produkte von unterschiedlichen Händler kaufen und bist nicht auf einen Hersteller begrenzt. Das Board z.B. von Olimex läuft genau so gut mit der Arduino-IDE zusammen wie das originale Arduino Uno Board.

Der Arduino hat festgelegte Anschlussreihen. Auf diese kann man dann verschiedene sog. Shields stecken. Diese erweitern dann das Grundboard um gewisse Funktionen. Shield wie auch Arduinos werden von den verschiedensten Herstellern angeboten.

Die Anschlussreihen kannst du aber auch direkt benutzen, um deine Erweiterung z.B. auf deinem Steckbrett mit dem Arduino Board zu benutzen.

## Und was kann man damit machen?

Vieles! Du kannst den Arduino für dich rechnen lassen. Du kannst LEDs (Leuchtdioden) zum Leuchten bringen. Du kannst einen Roboter dein Zimmer aufräumen lassen. (Naja, ok, dafür ist der Arduino nun doch zu klein.) Das wichtigste Bauteil auf dem Arduino ist der kleine schwarze Kasten in der Platinenmitte. Das ist ein sog. Mikrocontroller, das Herz des Arduinos. Das ist ein Computer, der Befehle, die du ihm gibst, ausführt. Diese Befehle gibst du ihm in Form von sog. Programmen. (Beim Arduino Sketches genannt)

## Und wie funktioniert das?

Ersteinmal musst du wissen, dass ein Computer eigentlich sehr unintelligent ist. Denn er kann gar nicht viel. Eigentlich kann er von Hause aus gar nichts. Man muss ihm zunächst einmal sagen, was er tun soll. Diese Anweisungen nennt man Programm. Nun hat der Computer seine eigene Sprache. Diese Sprache nennt man für gewöhnlich Maschinensprache und besteht einfach aus ganz vielen Zahlen, die für den Computer eine Bedeutung haben. Diese Zahlen werden im Binärsystem ausgedrückt. D.h. ein Computer kennt eigentlich nur 2 Ziffern, 0 und 1, oder auch True (Wahr) oder False (Falsch) Größere Zahlen setzt er dann, genau wie wir, aus diesen Ziffern zusammen. So ist z.B. eine 2 → 10 im Binärsystem, eine 3 wäre dann 11, 4 → 100 usw.

Ein Computer kann z.B. sehr schnell rechnen. z.B. kann unser kleiner Computer hier die Aufgabe 00110011 + 00001100 innerhalb von 62,5 ns ( also eine 0,0000000625 Sekunde) ausrechnen. Oder auch 16Millionen Additionen pro Sekunde. Tatsächlich gibt man die Geschwindigkeit eines

Prozessors auch gerne in MIPS an (Millionen Instruktionen pro Sekunde). Der Arduino hat 16 MIPS. Ein Raspberry hat z.B. 2.440 MIPS, ein PC, so wie ihr ihn wahrscheinlich zu hause habt, hat ca. 50.000 MIPS. Natürlich kannst du deinem Kleinen jetzt das Programm in Maschinensprache schreiben. Aber das ist eine sehr mühselige Arbeit. Du musst jeden Befehl mit der Befehlstabelle dann in einen ausführbaren Code umsetzen. Deswegen kam man schon sehr früh auf den Gedanken, diese stupide Übersetzungsarbeit den Computer selber machen zu lassen. Also fing man an das erste Programm (noch in Maschinensprache) zu schreiben, was für Menschen lesbaren Text in Maschinensprache übersetzt. Diese Programme heißen Assembler. Für jeden Befehl gab es dann ein sog. Mnemonik, also eine Abkürzung und textuelle Darstellung eines Befehls des Computers. Beispielsweise wird unsere Addition zweier Zahlen als ADD dargestellt. Unser Programm zum Addieren 2er Zahlen sähe in Assembler dann so aus:

```
ldi  r16, 0b00110011  ; erste Zahl
ldi  r17, 0b00001100  ; zweite Zahl
add  r16, r17         ; Addition der Register r16 und r17. Das Ergebnis wird
                      ; im Register r16 abgelegt
```

Hier gibt‘s dann gleich noch einen Begriff, das Register. Das Register ist einfach eine Speicherzelle in dem Prozessor selber.
Und was ist jetzt wieder der Prozessor?
Das ist die Einheit in einem Computer, die tatsächlich rechnet. Ein Computer besteht aus vielen verschiedenen Teilen, die wichtigsten davon sind:

**Prozessor**: das ist der Teil der das Programm verarbeitet und z.B. rechnet

**Speicher**: Darin liegen das Programm und die Daten. Speicher gibt es in unterschiedlichster Form, RAM, EEPROM, FLASH, aber auch USB Stick, SD Karte, Festplatte, CD, DVD das ist alles Speicher.

**E/A Einheit**: Hier werden Befehle vom Computer zur Außenwelt gegeben und umgekehrt, dazu zählen z.B. die Pins an unserem Arduino, oder auch die serielle Schnittstelle, bei einem PC z.B. die Grafikkarte oder auch der Uhren-Baustein oder der Baustein, der Tastatur und Mausbefehle entgegennimmt. USB Anschlüsse usw.

Der Assembler übersetzt dann das o.g. „Programm“ in die entsprechenden Maschinenbefehle, die der Computer, nachdem man ihn damit gefüttert hat, ausführt. Läuft der Assembler auf dem gleichen Computer, ist das recht einfach. Aber unser Arduino hat keinen eingebauten Assembler, er hat auch keine Tastatur und Maus und auch keine Möglichkeit einen Monitor anzuschließen. Also musst beim Arduino der Assembler auf einem anderen Computer ausgeführt werden. (Das nennt man dann Crossassembler) Ein kleines Programm ist aber im Arduino schon enthalten. Das wird bei einem Start oder Reset automatisch ausgeführt und schaut auf der seriellen Schnittstelle, ob da ein neues Programm geladen werden möchte. Und so kommt dann das im Crossassembler übersetzte Programm auf deinen Arduino.

Im Laufe der Zeit war aber auch diese maschinennahe Programmierung sehr unkomfortabel. Also entwarf man andere Sprachen. Heutzutage gibt es 1000de verschiedene Programmiersprachen für

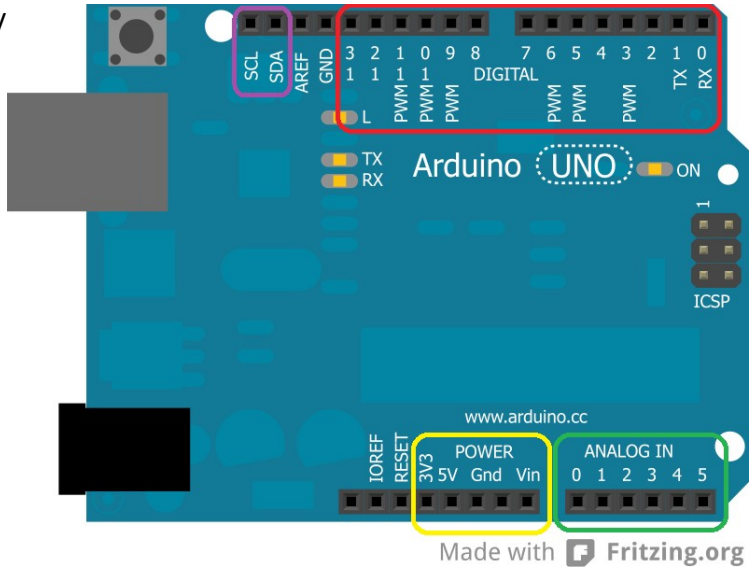
jeden Zweck. Manche können gut rechnen, manche sind gut für KI (künstliche Intelligenz), mache Sprachen gibt es für fast jeden Computer auf der Welt, manche nur für einen bestimmten. Die zur Zeit wichtigsten und bekanntesten Programmiersprachen sind C++, Java, Javascript. Unser kleiner Arduino versteht mit Hilfe der Arduino IDE z.B. C++. Und das ist die Sprache, mit der du die Programme für den Arduino schreibst. Und du wirst sehen, dass klingt viel komplizierter als es ist. Denn für viele Dinge, die man immer wieder braucht, gibt es beim Arduino bereits fertige Bausteine, die du nur benutzen musst. Um das in C++ geschriebene Programm in Maschinensprache zu übersetzen, brauchst du allerdings einen Übersetzer, einen sog. Compiler. Dieser ist in der Arduino IDE bereits enthalten. (Und da das Programm auf einem anderen Computer als dem Arduino selber übersetzt wird, nennt man das wieder Crosscompiler) Übrigens der Arduino-C++-Compiler versteht auch Assembler, weswegen man den Arduino auch direkt in Assembler programmieren kann. Für manche Dinge, vor allem bei zeitkritischen Aufgaben, ist das sehr wichtig. Für diesen Kurs brauchst du das nicht.

Jetzt aber nochmal zurück zur Hardware und im Speziellen zu den E/A Bausteinen, also den Teilen, mit denen der Arduino mit seiner Umwelt „spricht“.

## Wo findest du was auf dem Arduino?

Der Arduino kann mit seiner Umwelt auch interagieren. Dazu diesen die verschiedenen Anschlüsse.

- **Rot** gekennzeichnet sind die digitalen Ein/ Ausgänge. Diese werden als D0..13 oder auch als Pin0..13 bezeichnet.
  Ports, wie in der Atmel-Dokumentation, gibt es im Arduino nicht. Jeder Ausgang oder Eingang wird einzeln benutzt.
- **Grün** sind die analogen Eingänge.
  Wenn es mal wirklich eng wird mit den digitalen Eingängen, kann man diese Pins auch als digitale Ein/Ausgänge benutzen. Je nach Arduinotyp haben diese dann eine andere Bezeichnung. Im UNO sind das die Pins 14..19.




- **Gelb** sind die Stromversorgungsanschlüsse. Hier kannst du deine Bauteile mit Strom versorgen. Aber Vorsicht: Über den 5V Pin solltest du nicht mehr als 250mA verbrauchen. Und der 3,3V Pin kann gerade mal 50mA. $V_{in}$ ist parallel zu dem dicken schwarzen DC Anschluss.

Auf den neueren Arduinoboards befindet sich ein Taster. Mit diesem kann man einen Reset im Arduino auslösen, falls mal gar nichts mehr geht. Hier auf der schematischen Darstellung ist der oben links.

Darunter befindet sich der USB Anschluss. Also die Nabelschnur zum PC. Das ist der kleine silberne Kasten. Hierüber unterhält sich der Arduino mit dem PC.

Und darunter ist immer der Strom-Anschluss zu finden. Den brauchst du nur, wenn du den Arduino nicht mit dem USB Anschluss mit Strom versorgen möchtest.

Übrigens: der kleine schwarze quadratische Kasten in der Mitte der Platine ist unser Mikrocontroller. Er enthält die wichtigsten Bausteine, den Prozessor, 3 verschiedene Sorten Speicher (RAM, Flash und EEPROM) und ein paar E/A Einheiten, wie z.B. die digitalen Pins, einen A/ D Wandler mit 6 Eingängen, 6 analoge Ausgänge, eine serielle Schnittstelle und noch ein paar andere Schnittstellen.

## Bauteile für den Kurs

1 Arduino Uno (oder ähnlich)

1 Die Cube Platine

30 LEDs

3 2N7000 FET

10 Widerstände 180 Ohm

1 Stiftleisten

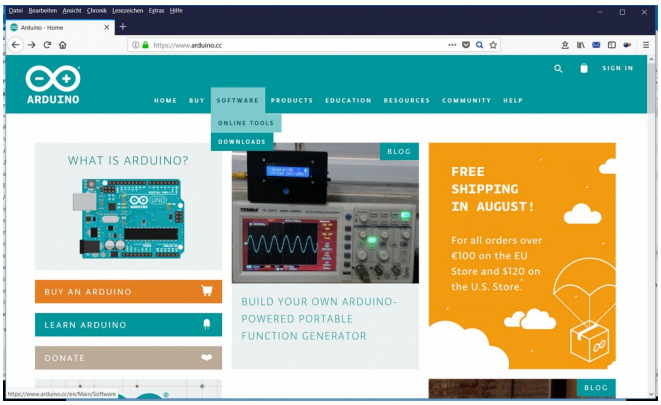
1 Buchsenleiste

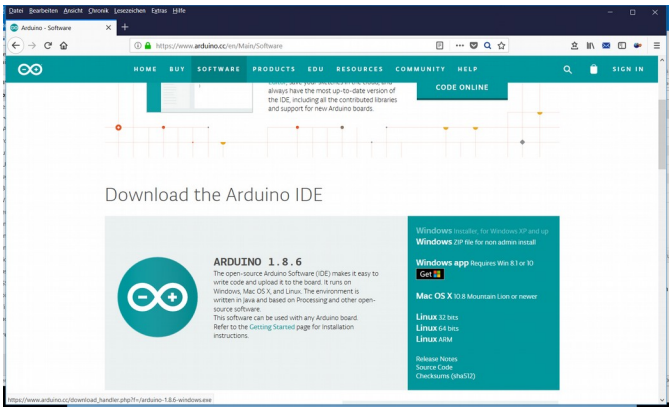
# Vorbereitung

## Installation der Software

Bevor wir anfangen, müssen wir zunächst einmal die Software installieren. (Auf den Geräten der MCS Akademie ist bereits alles vorinstalliert)

Dazu ladet ihr von der Seite Arduino.cc im Bereich Software/Downloads die aktuelle Version der Arduino IDE (derzeit 1.8.7) herunter und installiert diese. Auf dem Desktop eures Computer erscheint dann automatisch ein Icon, mit dem ihr die Entwicklungsumgebung starten könnt.



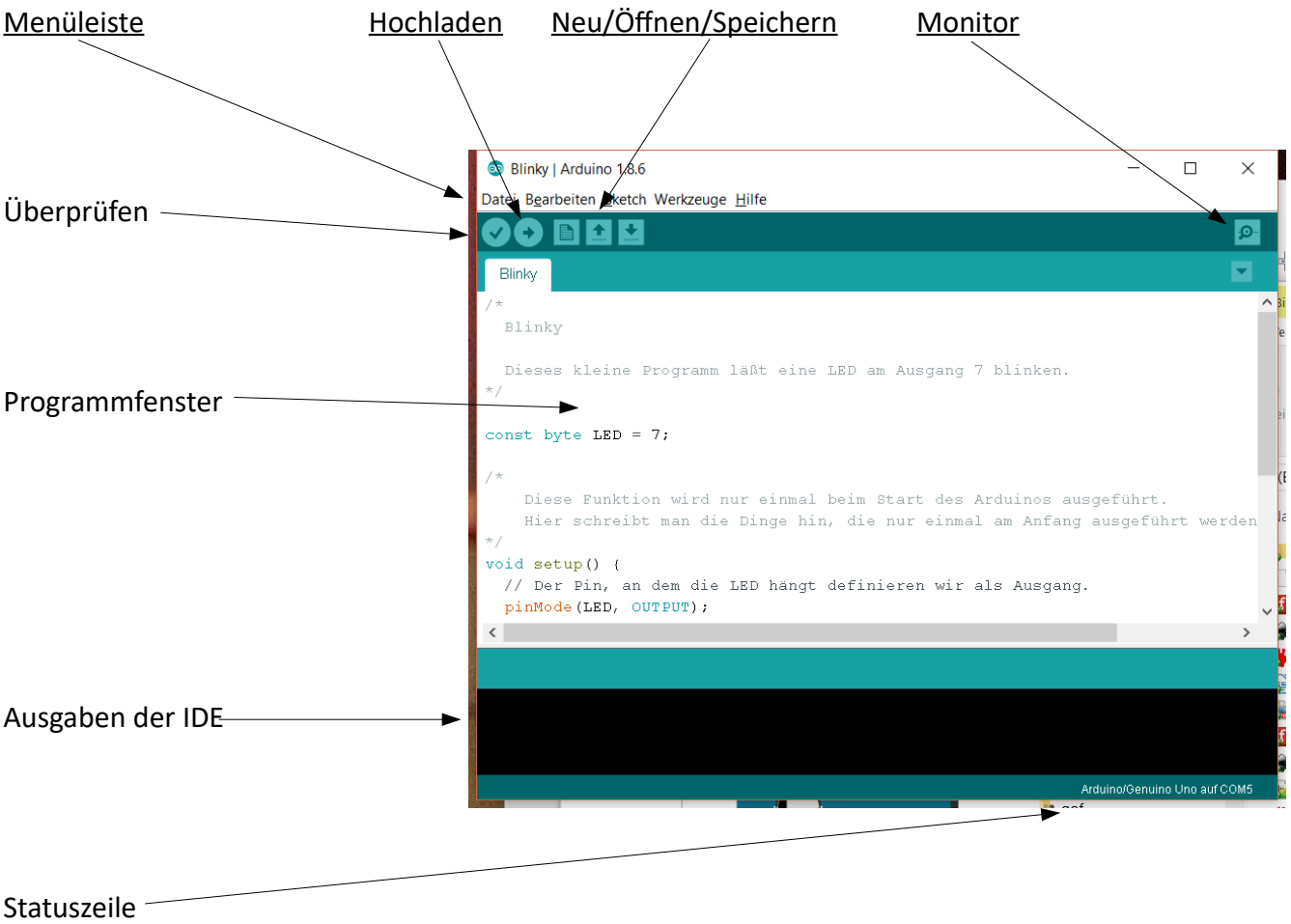




Danach ladet ihr euch die Sourcen zu diesem Workshop herunter. Auch diese installiert ihr auf dem Rechner, den ihr verwendet wollt. Jetzt steckt ihr Arduino mit dem Kabel an den Computer. Dabei installiert sich noch ein Treiber für den Arduino. Es gibt viele Seiten im Internet, die sich mit der Installation und überhaupt mit dem Arduino beschäftigen. Auch ich (Autor) hab eine Seite gemacht. Die findet ihr hier: http://rcarduino.de

Da behandele ich viele verschiedenen Themen rund um den Arduino. Wenn ihr mal Lust habt, schaut mal vorbei.

## Die Entwicklungsumgebung

Jetzt starten wir die Arduino IDE und schon kann es losgehen.




### Menüleiste

Hier befinden sich alle Befehle der IDE.

### Hochladen

Mit diesem Knopf kannst du dein Programm zu dem Arduinoboard übertragen. Wenn es bis dahin noch nicht kompiliert wurde, wird das Programm neu kompiliert.

### Monitor

Mit dem Monitor kann dir dein Arduino Texte schicken, und auch du kannst ihm eine Nachricht schicken. (Was dann passiert, steht in deinem Programm)

## Überprüfen

Mit dem Überprüfen wird dein Programm kompiliert und dabei überprüft, ob du alles richtig geschrieben hast. Der Compiler (so heißt das Programm, was das tut) überprüft aber nur, ob du dich an gewisse Regeln gehalten hast. Diese Regeln nennt man Syntax. Der Compiler prüft nicht, ob dein Programm korrekt funktioniert. Das musst du schon selber testen.

## Programmfenster

Hier schreibst du deinen Programmcode, also dein Programm rein. Die Entwicklungsumgebung gibt dir ein paar Hilfen dabei. Z.B. werden reservierte Wörter, also Wörter, die eine vorgegebene Bedeutung haben, farblich anders dargestellt. Auch vordefinierte Funktionen werden z.B. orange markiert.

## Ausgaben der IDE

Hier gibt dir die Entwicklungsumgebung (IDE) Hinweise aus. Z.B. wenn du einen Fehler gemacht hast.

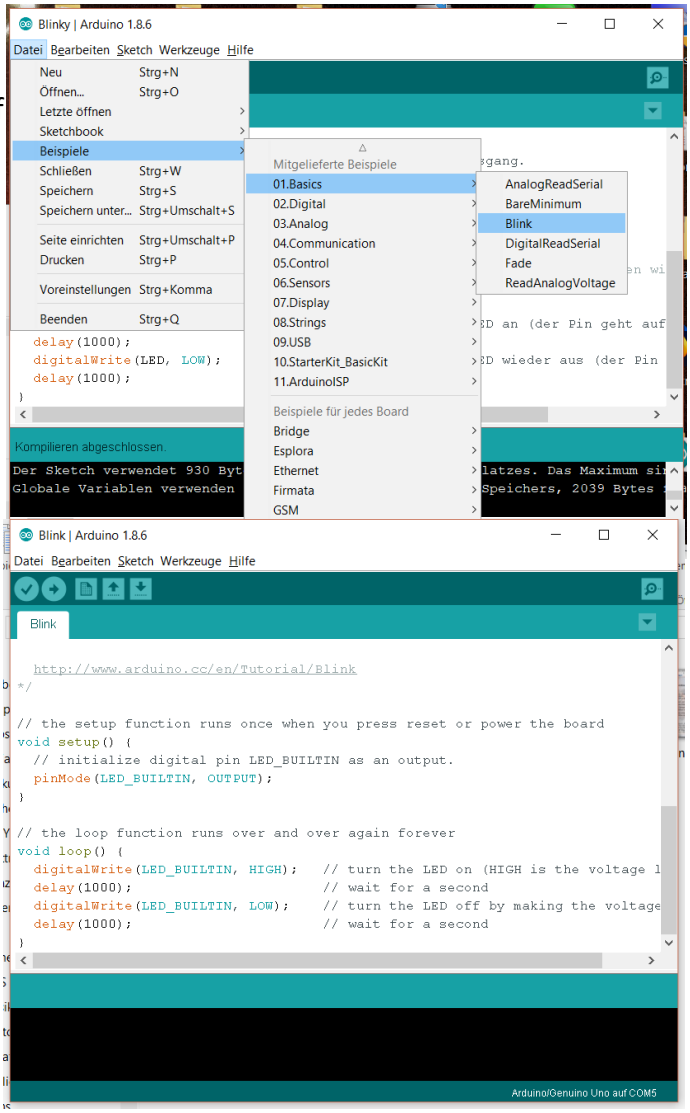
## Statuszeile

In der Statuszeile zeigt dir die IDE z.B. an, mit welchem Arduino du gerade verbunden bist.

## Das erste Programm

Wenn du das erste mal ein Arduinoboard an deinen Computer anschließt, solltest du zunächst einmal kontrollieren, ob das Board von der IDE richtig erkannt wurde. Dazu öffnest Du das Programm blink. Das findest du unter [Datei]/[Beispiele]/[01.Basics]/[Blink]

Dieses Programm lädst du nun mit dem Pfeilknopf zu deinem Arduinoboard hoch. Wenn alles funktioniert hat, sollte dann eine LED auf dem Board blinken. Hast du das ohne Fehler geschafft, geht‘s weiter. Nun schau dir mal das Programm an.




Ein Arduino Programm besteht aus mindestens 2 Funktionen. (manche sagen da auch Prozeduren zu...)
Die eine heißt `void setup() {}` und die andere nennt sich `void loop(){}`.

- `setup` wird beim Start einmal ausgeführt. Hier kann man Code schreiben, der nur einmal ausgeführt werden muss, wie z.B. Initialisierungen.
- `loop` wird immer wieder ausgeführt. Und zwar unendlich oft.

Hier noch mal das Grundgerüst eines Arduinoprogramms.

```
void setup() {

}

void loop() {

}
```

*Grundgerüst eines Arduinoprogrammes*

Im Setup befindet sich nun der Befehl

```
pinMode(LED_BUILDIN, OUTPUT);
```

Ein Pin ist einer dieser Ein und Ausgänge. Um einen Pin zu benutzen, musst du dem Controller zunächst einmal sagen, als was der Pin arbeiten soll. In dem Beispiel soll der Pin einen LED zu leuchten bringen. Also ist das ein Ausgang. Bei dem Befehl `pinMode` müssen nun 2 Parameter gesetzt werden. Der 1. Parameter bezeichnet den Ein/Ausgang der 2. dann, ob du den Pin als Eingang oder Ausgang benutzen möchtest. Viele Arduinos haben an einem Pin direkt eine LED

eingebaut. Welcher Pin benutzt wird, steht in der Anleitung. Allerdings haben es uns die Entwickler leicht gemacht. Sie haben direkt eine Konstante mit der richtigen Pinnummer belegt. Somit brauchst du die Nummer nicht nachschauen, sondern kannst direkt die Konstante mit dem Namen `LED_BUILDIN` verwenden. Somit bedeutet

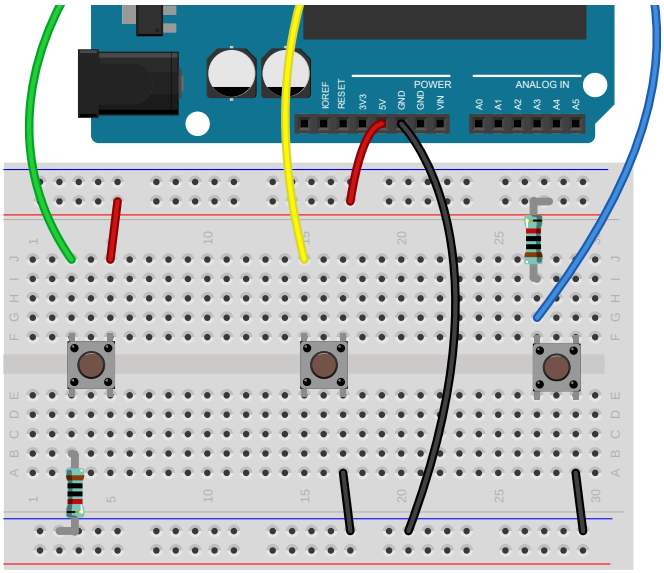
```
pinMode(LED_BUILDIN, OUTPUT);
```

stelle den Pin `LED_BUILDIN` (13) auf Ausgang. Und was bedeutet jetzt:

```
pinMode(2, INPUT);
```

**Wichtig:** Hinter jedem Befehl steht immer ein Semikolon (;)

Stelle Pin 2 auf Eingang. Es gibt noch einen speziellen Modus: `INPUT_PULLUP`. Dieser wird verwendet wenn man einen Schalter oder Taster (oder was ähnliches) an einem Eingang betreiben möchte. Denn der Eingang möchte immer einen definierten Spannungspegel am Eingang sehen, entweder 5V (was dann einem HIGH oder auch eine 1 bedeutet) oder 0V, das dann LOW bzw. 0 bedeutet. Wenn ich nun einen Schalter anschließe, müsstest du zwischen diesen beiden Werten 5V und 0V (oder auch GND) wechseln. Das bedeutet aber, du bräuchtest 3 Kabel für den Anschluss. Deswegen ist man auf folgenden Trick gekommen. Die Signalleitung (also die Leitung, die vom Schalter zum Eingang geht) wird mit einem Widerstand an 5V angeschlossen. Somit sieht der Pin immer 5V. Der Schalter wird nun einfach an die Signalleitung und an Masse angeschlossen. Schaltet man nun den Schalter ein, wird der Pin gegen GND kurzgeschlossen. Keine Sorge, durch den Widerstand wird der Strom auf einen ganz kleinen Wert begrenzt. Aber der Pin sieht nun GND und gibt dann eine 0 aus. Diesen Widerstand bezeichnet man als Pull-up Widerstand. (Pull = ziehen, up = hoch, also hochziehen, in diesem Fall auf 5 V) Unser kleiner Controller hat nun für jeden Pin einen entsprechenden Pullup Widerstand eingebaut. Und diesen aktivierst du eben mit dem Wert INPUT_PULLUP.




**Wichtig:** Wenn du so den Schalter anschließt, ist die Logik umgekehrt, denn ein aktivierter Schalter bedeutet eine **0** oder **LOW**. Will man die Logik umdrehen, muss man einen sog. Pulldown Widerstand einsetzte und den Schalter statt gegen GND gegen +5V arbeiten lassen. Dann verwendet man im pinMode den Wert INPUT. Ein aktivierter Schalter bedeutet dann eine **1** oder auch ein **HIGH**.

Der nächste Befehl befindet sich schon in der loop Funktion.

```
digitalWrite( LED_BUILDIN, HIGH);
```
Hier wird auf einem Ausgang ein bestimmter Pegel ausgegeben. In diesem Fall gibst du der LED eine HIGH oder 1. Das bedeutet der entsprechende Pin (LED_BUILDIN = 13) gibt 5V aus. Die LED leuchtet. Bei LOW gibt der Controller 0V (oder auch GND) aus. Die LED erlischt.

Und noch ein kleiner Befehl

```
delay(1000);
```
Dieser Befehl stoppt die Programmausführung um die in dem Parameter angegebene Anzahl der Millisekunden. (1ms = 1/1000s) Hier sind es 1000ms = 1s. Die LED leuchtet eine 1 Sekunde lang dann geht Sie für eine Sekunde aus.

Jetzt willst du bestimmt auch mal selber ein (oder auch mehrere) LEDs zum Leuchten bringen.

# Aufbau der Platine

# Aufbau des Cubes

## Aufbau der LED Cubeebenen

## Zusammenbau des Cubes

## Hochzeit

# Programmierung des Cubes

## Aufbau einer LED

Eine Leuchtdiode (LED) hat 2 Anschlüsse. Der eine Anschluss heißt Kathode, der Andere nennt sich Anode. Wenn du die LED zum Leuchten bringen willst, musst du zwischen Anode und Kathode eine Spannungsquelle anschließen. Die Anode kommt dabei an den Pluspol (+) der Batterie und die Kathode an den Minuspol (-). Wenn du mal in das Glas der Diode schaust, siehst du 2 verschiedene Metallteile. Beide sind jeweils mit einem Bein verbunden. Das eine Teil ist relativ groß und hat oben einen kleinen Trichter. Das ist die Kathode. Die Kathode ist auch durch eine kleine Abflachung am Gehäuse gekennzeichnet. In dem Trichter liegt der kleine Kristall, der leuchtet. Daneben ist ein kleiner Pin von dem ein sehr dünner Metallfaden in den Trichter (und da auf den Kristall) geht. Das ist die Anode.


**Doch Vorsicht!** Eine LED hat eine bestimmte Betriebsspannung. Die rote Leuchtdiode z.B. braucht eine Spannung von 2,2V. Und eine Leuchtdiode verträgt auch nur einen gewissen Strom, 20mA. Wenn du jetzt einfach eine Batterie anschließt, kann es entweder sein, dass die Leuchtdiode nicht leuchtet, z.B. wenn du nur eine Mignon (AA) Batterie benutzt. Benutzt du eine zu große Spannung geht die Leuchtdiode geht kaputt, z.B. bei einem 9V Block. Damit eine Leuchtdiode richtig leuchtet, brauchst du zusätzlich einen Widerstand. Dieser sorgt für die richtige Spannung und den richtigen Strom bei der Leuchtdiode. Unser kleiner Computer arbeitet mit 5V. Der richtige Widerstand der LED berechnet sich so:

$R = \frac{(U - U_{LED})}{I_{LED}}$ oder eingesetzt $R = \frac{5\,V - 2{,}2\,V}{0{,}02\,A} = 140\,\Omega$ also 140 Ohm. Einen 140 Ohm Widerstand gibt es aber so nicht. Und die LED leuchtet auch schon sehr hell bei deutlicher weniger Strom. Deswegen nimmt man gerne einen etwas höheren Wert hier z.B. 180 Ohm.

# Anhang

## Befehlsübersicht

### Compileranweisungen

| Befehl | Beschreibung |
|---|---|
| `#define <Name>` | sagt dem Compiler, das es Name gibt. So kann man z.B. bestimmte Codebereiche mit Hilfe der bedingten Kompilierung ein bzw. ausblenden. |
| `#define <Name> <Wert>` | definiert ein Makro oder Übersetzung. Überall wo im Quelltext Name auftaucht, ersetzt der Compiler das mit Wert. |
| `#ifdef <Name>`<br>`[Anweisungen 1]`<br>`#else`<br>`[Anweisungen 1]`<br>`#endif` | bedingte Kompilierung. Die Anweisungen 1 werden nur compiliert, wenn Name existiert. Ansonsten werden Anweisungen 2 compiliert. Der Else Teil ist optional. |
| `#ifndef <Name>`<br>`[Anweisungen 1]`<br>`#else`<br>`[Anweisungen 2]`<br>`#endif` | bedingte Kompilierung. Das genaue Gegenteil von `#ifdef` |
| `#include „Dateiname“` | Weißt den Compiler an, an dieser Stelle die lokale Datei „Dateiname“ zu kompilieren. |
| `#include <Dateiname>` | Weißt den Compiler an, an dieser Stelle die globale Datei „Dateiname“ zu kompilieren. |

## Strukturen

| Befehl | Beschreibung |
|---|---|
| `void setup() {`<br>`[Anweisungen 1]`<br>`}` | Muss in jedem Arduinoprogramm vorkommen. Dieser Codeteil wird beim starten des Arduinos einmal durchlaufen. |
| `void loop() {`<br>`[Anweisungen 1]`<br>`}` | Muss in jedem Arduinoprogramm vorkommen. Dieser Codeteil wird nach dem `setup` immer wieder ausgeführt. |
| `if ([Bedingung]) {`<br>`[Anweisungen 1]`<br>`} else {`<br>`[Anweisungen 2]`<br>`}` | ist die Bedingung wahr, wird Anweisungen 1 ausgeführt, sonst wird Anweisungen 2 ausgeführt. |
| `switch (Ausdruck) {`<br>`  case Wert 1:`<br>`  [Anweisungen 1]`<br>`  break;`<br>`  case Wert 2:`<br>`  [Anweisungen 2]`<br>`  break;`<br>`  case Wert 3:`<br>`  [Anweisungen 3]`<br>`  break;`<br>`  default:`<br>`  [Anweisungen 4];`<br>`  }` | Hat ein Ausdruck (z.b: eine Variable) mehrere Möglichkeiten, das gilt für Aufzählungen oder auch für Zahlen vom Typ byte kann man hiermit auf bases des Wertes Entscheidungen treffen.<br>Hat der Ausdruck den Wert 1 werden die Anweisungen 1 ausgeführt. Gleiches gilt für Wert 2 und Anweisungen 2 und Wert 3 und Anweisungen 3. Das ganze kann man beliebig fortsetzen.<br>Stimt Ausdruck mit keinem definierten Wert überein, wird der Default Teil benutzt (Anweisungen 4)<br>Wichtig ist auch das break. Damit wird die Programmausführung nach dem switch weitergeführt. Fehlt das break, werden die anderen Anweisungen bis zum nächsten Break bzw. bis zum Ende ausgeführt. |
| `for (int i = 1; i <= 100; i++) {`<br>`...`<br>`}` | |
| `for ([init];`<br>`[Bedingung]; [loop])`<br>`{`<br>`  [Anweisungen 1]`<br>`}` | FOR-Schleife. Diese Schleife wird sooft ausgeführt wie die Bedingung wahr ist. Zunächst wird Init ausgeführt. Dann wird die Bedingung gecheckt. Ist diese wahr, wird zunächst loop ausgeführt und dann die Anweisungen 1. |
| `Beispiel`<br>`for (byte i = 1; i <= 100; i++) {`<br>`  [Anweisungen 1]`<br>`}` | Im Beispiel eine einfache schleife, die 100 Mal ausgeführt wird. |
| `while(Bedingung){`<br>`  [Anweisungen 1]`<br>`}` | While Schleife, Anweisungen 1 werden immer wieder ausgeführt, bis die Bedingung falsch ist. |
| `do {`<br>`  [Anweisungen 1]` | Do Schleife, Anweisungen 1 werden immer wieder ausgeführt, bis die Bedingung falsch ist. Unterschied zur While Schleife, da die Bedingung |

| Befehl | Beschreibung |
|---|---|
| `} while (Bedingung);` | erst am Ende der Schleife geprüft wird, werden die Anweisungen mindestens einmal durchlaufen. |

## Bedingungen

| Befehl | Beschreibung |
|---|---|
| `a > b` | ist wahr, wenn die variable a größer als b ist. |
| `a < b` | ist wahr, wenn die variable a kleiner als b ist. |
| `a >= b` | ist wahr, wenn die variable a größer oder gleich b ist. |
| `a <= b` | ist wahr, wenn die variable a kleiner oder gleich b ist. |
| `a == b` | ist wahr, wenn die variable a gleich b ist. (ACHTUNG: 2 Gleichheitszeichen) |
| `a != b` | ist wahr, wenn die variable a nicht gleich b ist. |
| `! (a)` | ist wahr, wenn a nicht wahr ist. |

## Typen

**boolean**: kann nur 2 Werte annehmen, Wahr oder Falsch (true, false) oder auch 1 oder 0. (0=false, 1=true)
**byte**: kann ein Byte aufnehmen. Also Werte von 0..255.
**char**: kann ein Zeichen aufnehmen, also z.B. 'A' oder 'ß'.
**int**: kann Werte zwischen -32.768 bis 32.767 aufnehmen. (Braucht 2 Byte Platz)
**unsigned int**: kann Werte zwischen 0 und 65.535 aufnehmen.
**word**: ist das gleiche wie unsigned int.
**long**: kann Werte zwischen -2.147.483.648 to 2.147.483.647. (Braucht 4 Byte Platz)
**unsigned long**: kann Werte zwischen 0 to 4.294.967.295. (Braucht 4 Byte Platz)
**float**: sind Fließkommazahlen. Bereich von 3.4028235E+38 bis runter nach -3.4028235E+38, werden aber nur in 4 byte gespeichert. Deswegen ist die Genauigkeit recht klein. Manchmal ist dann 6,0 / 3,0 leider nicht 2,0…
**String**: Zeichenkette, also sowas wie „willie ist da!“ (Für die Spezies: Das ist gar kein Typ sondern ein Objekt…)

Und dann gibt's da noch die Felder. (Arrays) Die definiert man einfach so: Feld mit 4 Bytes. Gezählt wird dann immer von 0 an. Hier geht's also von 0..3.

**byte** feld[4];

## eingebaute Befehle (nicht vollständig)

| Befehl | Beschreibung |
|---|---|
| `pinMode(Pin, Mode)` | Setzen des odus eines Pins. Modus kann sein: OUTPUT, INPUT, INPUT_PULLUP. |
| `digitalRead(Pin)` | Abfrage des aktuellen Zustandes eine Eingabe pins |
| `digitalWrite(Pin, Wert)` | Ausgabe des Wertes auf einen Pin. Wert kann nur die Werte 0 oder 1 haben. |
| `analogRead(Pin)` | Lesen eines Wertes von den analogen Eingängen. Der Wert liegt im Bereich von 0..1023. Geht nur auf den extra gekennzeichneten Pins A0..A5. Pin kann somit die Werte 0..5 annehmen. |
| `analogWrite(Pin, Wert)` | Ausgabe eines analogen Wertes auf einen Pin. Wert kann nur einen Bereich von 0..255 annehmen. Pin können nur die gekennzeichneten Pins sein. 3, 5, 6, 9, 10, 11 |
| `delay(Wert)` | Warten um Wert in msek mit der Programmausführung. |
| `millies()` | Gibt den aktuellen Zeit seit Start des Programms in Millisekunden. ACHTUNG nach ca. 50 Tagen läuft der interne Zähler über und beginnt dann bei 0 erneut. |
| `tone(Pin, Frequenz)`<br>`tone(Pin, Frequenz, Dauer)`<br>`noTone(Pin)` | gibt einen Ton mit der Frequenz auf dem Pin aus. Ist die Dauer angegeben (in Millisekunden) wird der Ton solange gespielt. Ansonsten bleibt der Ton erhalten bis `noTone()` aufgerufen wird. |
| `min(a,b)` | Die Funktion liefert den kleineren der beiden Werte x und y zurück. |
| `max(a,b)` | Die Funktion liefert den größeren der beiden Werte x und y zurück. |
| `abs(x)` | Die Funktion liefert den absoluten Wert von x zurück. Also den Wert ohne Vorzeichen. |
| `constrain(a,x,y)` | Die Funktion liefert einen Wert innerhalb von den Werten x und y zurück. Ist also a innerhalb von x und y dann kommt a zurück ist a < x dann kommt x zurück und ist a > y dann kommt y zurück. |
| `map(a, x1, y1, x2, y2)` | ist eine Funktion für faule. Oder auch der implementierte Dreisatz. a ist der Eingabewert, x1,y1 ist der Wertebereich von a, x2 und y2 ist dann der angestrebte Wertebereich. Hier mal die mathematische Formel |
| `pow(b, exp)` | Die Funktion liefert $b^{exp}$ zurück. |
| `sqrt(x)` | Die Funktion liefert die Wurzel von x. |
| `sin(x), cos(x), tan(x)` | Das sind die möglichen trigonometrischen Funktion. Alle Argumente müssen in Rad angegeben werden. Das Ergebnis ist ein Float. sin und cos |

| Befehl | Beschreibung |
|---|---|
| | liefern ein Ergebnis zwischen -1 ud 1 ab, tan natürlich zwischen -unendlich und unendlich. |
| `Serial.begin(x);` | Startet die serielle Kommunikation. x ist die Baudrate und darf folgende Werte annehmen: 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, oder 115200 |
| `Serial.print(x)` | Gibt den Wert X auf der Schnittstelle aus. X kann ein beliebiger Datentyp sein. Print macht daraus eine Textrepresentation daraus. Man kann auch einen 2. Parameter verwenden. Dieser kann folgende Werte haben:<br><br>BIN: Gibt die Binäre Repräsentation aus, aus 45 wird dann „00101101“<br>OCT: Das ganze als Oktalzahl, aus 45 wird dann „55“<br>DEC: ergibt dann „45“<br>HEX: ergibt dann „2D“<br><br>Bei Flieskommavariablen kann man mit dem 2. Parameter die Anzahl der Nachkommastellen bestimmen.<br><br>print(1.23456, 0) gibt „1“<br>print(1.23456, 2) gibt „1.23“<br>print(1.23456, 4) gibt „1.2345“<br><br>Man beachte hier die Amerikanische Schreibweise. Nicht Komma ist das Komma sondern der Punkt. |
| `Serial.println()` | Das gleiche wie print allerdings wird die Zeichenkette mit einem Zeilenende und Zeilenvorschubzeichen abgeschlossen. Also Zeichen 13 und 10. (0x0D 0x0A, oder auch \r\n) |
| `Serial.read()` | ließt ein Zeichen von der Schnittstelle. Ist nix da gibt's eine -1. |
| `Serial.parseInt()` | Liest und interpretiert die nächste Int-Variable aus dem Stream. |
| `Serial.parseFloat()` | Liest und interpretiert eine Float-Variable aus dem Stream. |
| `randomSeed(Wert);` | Initialisierung des Zufallszahlgenerator mit einem Anfangswert. Um eine echte Zufallszahl zu bekommen, sollte man den Startwert auch zufällig erzeugen. Beispielsweise könnte man einen unbenutzten analogen Eingang auslesen. Da dort Rauschen anliegt bekommt man mit analogRead(0) einen guten Startwert. |
| `random(max)`<br>`random(min, max)` | liefert den nächsten zufälligen Wert als long. Max ist die obere ausgeschlossene Schranke, das heißt Zahlen werden immer bis zu Max-1 erzeugt. Min ist die untere eingeschlossene Schranke, also kann min durchaus mal als Zufallszahl vorkommen. |

## Webverweise

Arduino Homepage

https://www.arduino.cc/

Download der IDE

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

Arduino Webseite des Autors

http://www.rcarduino.de

Workshop Unterlagen, Sourcen und mehr

http://rcarduino.de/doku.php?id=arduino:mcsworkshop