

Mikrocomputertechnik für Applikationsentwickler

Stefan Mühlebach

Frühjahr 2025

1 Einführung

Im letzten Jahr steht bei beiden Fachrichtungen (Systemtechniker und Applikationsentwickler) das Fach «Mikrocomputertechnik» auf dem Programm – zum letzten mal, wie mir gesagt wurde. In welchem Gefäss zukünftige Studierende einen Einblick in die Hardware und hardwarenahes Programmieren erhalten werden, ist mir aktuell nicht bekannt.

Das Fach ist auf 9 Nachmittage a 4 Lektionen (macht 36 Lektionen) aufgeteilt und dauert damit bloss ein halbes Semester. Im offiziellen Stoffprogramm der TEKO wird das Fach wie folgt umrissen:

Unterrichtsziel Die Studierenden können ein Mikroprozessor- resp. ein Mikrocontrollersystem mit geeigneter Programmiersprache programmieren.

Kurstag 1–2: Mikroprozessorprogrammierung mit Assembler

- Befehlsstruktur, -darstellung und -liste
- Adressierungsarten und Befehlssatz
- Unterprogrammtechnik
- Assemblerprogrammierung mittels Simulationsprogramm (Modellprozessor MOPS)

Kurstag 3–9: Mikrocontroller (Arduino)

- Übersicht über die Mikrocontroller-Familie
- Inbetriebnahme Funduino
(Software-Installation und Einstellungen)

- Programmierung des Mikrocontrollers
(Struktur, Datentypen, Arithmetik, Konstanten, Kontrollstrukturen, analoger und digitaler Input / Output, Zeit, serielle Kommunikation)
- Praktische Anwendung

Gemäss Lehrplan verfügen die Studierenden über Kenntnisse in Digitaltechnik und der objektorientierten Programmierung in C#.

Der Kurs deckt mit dem gezeigten Stoffprogramm zwei unterschiedliche Bereiche ab. Einerseits vermittelt er sehr knapp und rudimentär, wie ein *idealisierte Prozessor eines Von-Neumann-Rechners* arbeitet. Dies mit Hilfe des Simulationsprogramms (MOPS), welches auch im Informatik-Unterricht auf Sekundarstufe I zum Einsatz kommt.

2 Der Koffer

Um hardwarenah Programmieren zu können, braucht es zum einen die zu programmierende Hardware, aber auch eine vielfältige Sammlung von Hardwarekomponenten (wie Lämpchen, Taster, Drehschalter, Sensoren, Displays, Anzeigen, Motoren, etc), die man ansteuern kann und mit denen vielfältige Projekte realisiert werden können.

Uns steht mit dem «Fundino Starter Kit Education» ein sehr brauchbares Werkzeug zur Verfügung. Neben dem «Arduino UNO» (dem eigentlichen Mikrocontroller) enthält es eine brauchbare Menge an Komponenten aber auch eine brauchbare Menge der oben beschriebenen Komponenten enthält.¹



Abbildung 1: «Fundino» ist der Name eines Deutschen Arduino-kompatiblen Produktes, welches sich besonders für Schulen und andere Bildungsinstitutionen eignet.

Die Koffer sind nicht absolut identisch bestückt und sie haben auch nicht das gleiche Dienstalter, was sich z.B. in unterschiedlichen Abnutzungszuständen einiger Objekte zeigt. Meine Empfehlung ist, einen bestimmten Koffer zu wählen und an jedem Kursnachmittag den gleichen Koffer zu verwenden.

Im Folgenden werden die wichtigsten Komponenten des Koffers vorgestellt und im Abschnitt ?? zu einer ersten Schaltung zusammengebaut. Die meisten Komponenten werden jedoch erst im Rahmen einer konkreten Aufgabe vorgestellt.

2.1 Das Arduino-Board

Im Koffer befindet sich das Board mit dem Mikrocontroller selber. In unserem Fall ist dies ein *ATmega328P* der Firma *Microchip*. Im Anhang findet ihr einen Link auf die technische Dokumentation dieses Chips. Dieser Chip kommt auf allen Boards zum Einsatz, welche kompatibel mit «Arduino UNO» sind. Arduino ist eine Bezeichnung für das ganze Ökosystem um diesen Mikrocontroller oder ein vergleichbares Bauteil.

¹ Bitte nach jedem Kurstag die gebauten Schaltungen in ihre Einzelteile zerlegen und alles ordentlich im Fundino-Koffer verstauen. Defekte Elemente *nicht* in den Koffer zurücklegen, sondern der Kursleitung abgeben.

Mittlerweile sind auch Controller ein Teil der Arduino-Familie, die mit einem ARM-basierten Core ausgerüstet sind.

Abbildung 2 zeigt das Board aus der Vogelperspektive. Mit **A** bis **F** sind für die Programmierung relevante Bereich hervorgehoben und in Tabelle 1 detaillierter beschrieben. Wie die Pins ganz genau verwendet und konfiguriert werden, ist bei den jeweiligen Aufgaben beschrieben.

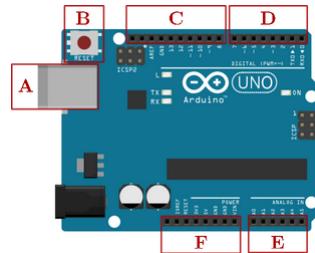


Abbildung 2: Das Layout von «Arduino UNO»

GRUPPE	BESCHREIBUNG
A	USB-Anschluss. Wird vorwiegend zum Hochladen der Programmen aus der IDE auf eurem PC verwendet und liefert auch gleich den Strom für den Betrieb.
B	Reset-Taste. Setzt alle Einstellungen auf ihren Default-Wert zurück und beginnt mit der Ausführung des im EEPROM abgelegten Programmes.
C	Digitale Pins 8 bis 13 (In- und Outputs, die Pins 9/10/11 sind ausserdem PWM-fähig, dazu später mehr) sowie die Pins für den SPI- (Pins 10/11/12/13) und den I ² C-Bus (Pins SCL/SDA).
D	Digitale Pins 0 bis 7 (In- und Outputs, die Pins 3/5/6 sind ausserdem PWM-fähig und die Pins 2/3 können für Interrupts verwendet werden).
E	Analoge Pins A0 bis A5 (In- und Output)
F	Pins für die Stromversorgung von externen Komponenten wie bspw. Schrittmotoren, Displays oder des Steckbrettes. Relevant sind 3.3V, 5V und GND (für <i>Ground</i> oder <i>Masse</i> – doppelt vorhanden, mit dem Anschluss in Gruppe C sogar dreifach).

Tabelle 1: Die GPIO-Stecker beim Arduino UNO und ihre Verwendung

2.2 Das Steckbrett

Die eigentliche Experimentier- oder Arbeitsplattform beim Elektro-Basteln ist das *Steckbrett* oder «Brotbrett» (engl. *Breadboard*). Es gibt sie in unterschiedlichen Größen und Ausführungen und sie ermöglichen den Aufbau komplexer Schaltungen ohne Löten und der Möglichkeit eines vollständigen Rückbaus.

In Abbildung 3 wird die Variante gezeigt, die sich mit grosser Wahrscheinlichkeit in eurem Koffer befindet. Die Löcher in den Randreihen (mit roter und blauer Farbe markiert) sind jeweils horizontal verbunden während die Löcher im eigentlichen Arbeitsbereich jeweils vertikal verbunden sind.

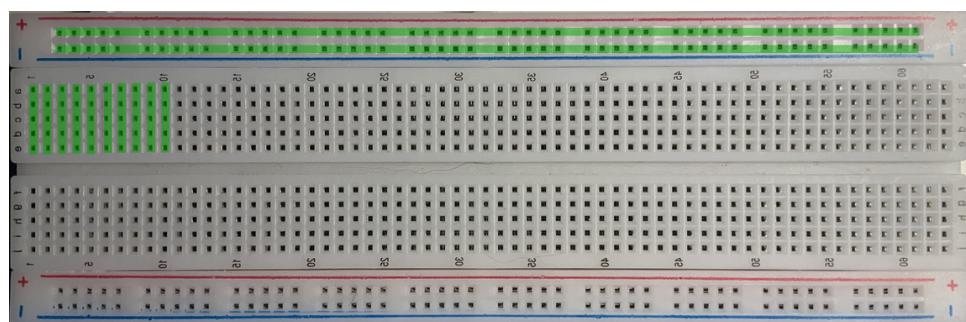


Abbildung 3: Steckbrett normaler Grösse. Mit grünen Geraden sind die zusammenhängenden Löcher markiert.

Die Randreihen werden üblicherweise mit 5 V (+) und GND (-) verbunden. Damit stehen diese beiden Potentiale mit wenig Aufwand und kurzen Drahtbrücken überall auf dem Steckbrett zur Verfügung.

Meine praktischen Erfahrungen mit Steckbrettern lassen mich meine Projekte jeweils nach folgendem Raster aufbauen (siehe auch Abbildung 4): die hinteren Randreihen (im Bild oben) werden mit 5 V und GND belegt, die Anschlüsse befinden sich ganz am Ende der Randreihen. Dann baue ich die konkrete Schaltung von hinten nach vorne (im Bild von oben nach unten) auf und verwende kurze rote, resp. kurze schwarze Drahtbrücken², um die Schaltung mit 5 V resp. GND zu versorgen.

2.3 Die Kabel

Der Koffer enthält eine stattliche Anzahl von farbigen Kabeln, mit welchen die Pins vom Arduino-Board oder eines Sensors mit dem Steckbrett verbunden werden können. Welche Farben für welche Signal-Arten verwendet werden, ist nicht vorgeschrieben und kann durch jede Gruppe selbständig bestimmt werden – mit zwei Ausnahmen: rot sollte wenn möglich immer für +5 V und schwarz für *Ground (GND)* verwendet werden.

Falls ihr einen neuen, resp. noch ungebrauchten Koffer vor euch habt, dann findet ihr die Kabel jeweils noch untereinander zu breiten Flachbandkabel verbunden von denen

²Diese kleinen Helferlein gehören leider nicht zum Standardumfang des Koffers und müssen extra besorgt, resp. hergestellt werden.

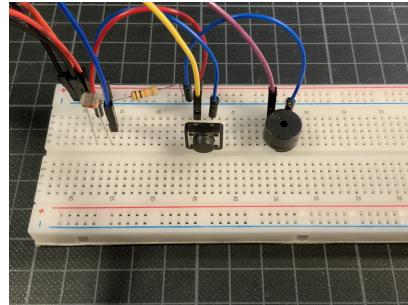


Abbildung 4: Der Aufbau von hinten nach vorne ermöglicht besseren Zugang zu den Bauteilen.

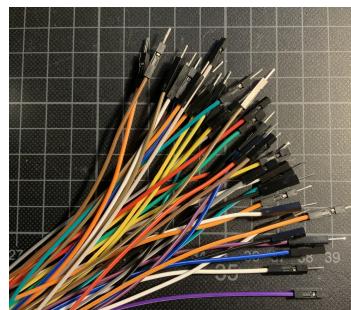


Abbildung 5: Eine grosse Anzahl von Kabel mit unterschiedlichen Farben und zwei Arten von Kabelenden.

sich die einzelnen Kabel einfach ablösen lassen. Lasst unbedingt ein 8 Kabel umfassendes Flachbandkabel intakt – dies erlaubt es, Daten mit einer Breite von 8 Bit etwas geordneter zu managen, als wenn man dafür 8 einzelne Drähte verwendet.

Es gibt drei Sorten von Kabel bezogen auf die Konfiguration der Kabelenden:
Stecker:Stecker: für Verbindungen zwischen dem Arduino-Board und dem Steckbrett
Buchse:Stecker: für die Anbindung von entsprechenden Modulen (siehe Abbildung 6)
Buchse:Buchse: wer findet einen Einsatzort für diesen Typ?

2.4 Und, und, und ...

Je nach Modell enthält euer Koffer eine grosse Anzahl von elektronischen Bauteilen die es noch zu entdecken gilt. Einige werden wir im Rahmen dieses Unterrichtes genauer unter die Lupe nehmen. Aber aus Zeitgründen werden wir uns nicht mit allen Modulen beschäftigen können.

3 Schemazeichnungen

Wenn man sich mit elektronischen Schaltungen beschäftigen will, dann kommt man um die sog. *Schemazeichnungen* nicht herum. Obwohl es regionale und fachgebietsspezifische

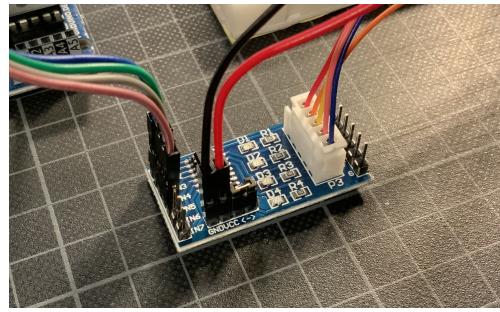


Abbildung 6: Kabel mit Buchsen am einen Ende werden für Bauteile verwendet, deren Anschlüsse als Pfosten ausgelegt sind – wie hier im Bild der Controller des Schrittmotors.

Unterschiede gibt, haben sich diese Zeichnungen über Jahrzehnte hinweg nur unwesentlich verändert und dienen vorwiegend der Dokumentation und der Kommunikation von elektr. Schaltungen.

Ein elektronisches Schaltschema ist eine abstrakte, bildliche Darstellung der relevanten Komponenten und ihren Verbindungen einer bestimmten Schaltung oder eines Ausschnittes davon. Damit ihr euch eine Vorstellung machen könnt, wie sowas in der Realität aussieht, habe ich in Abbildung 7 das Schema der Stromversorgung eines Teils meiner Stereoanlage kopiert.

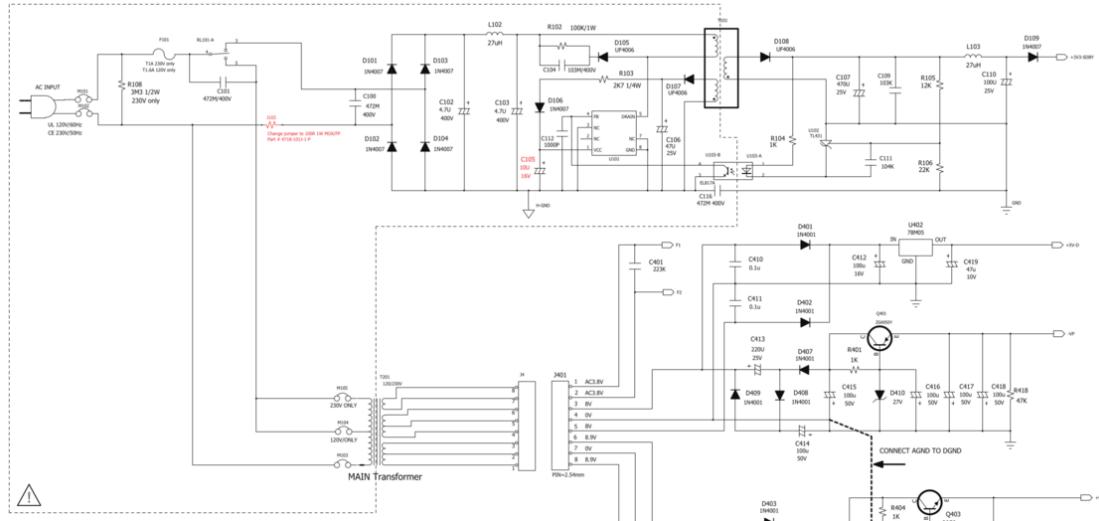
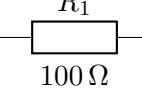
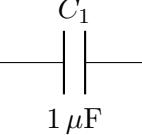
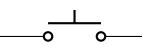
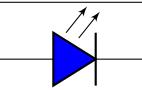
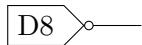
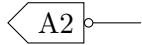
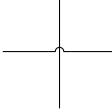
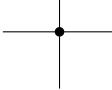
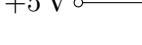


Abbildung 7: Transformer-Schaltung im NAD C 426 Tuner

3.1 Die wichtigsten Symbole und ihr reales Pendant

Ich habe in der folgenden Liste einige Symbole der Komponenten aus dem Koffer zusammengetragen.

BEZEICHNUNG	SCHEMASYMBOL	OBJEKT AUS DEM KOFFER
Widerstand mit einem Wert von $100\ \Omega$ (Ohm)	R_1 	
Kondensator mit einem Wert von $1\ \mu\text{F}$ (Mikro-Farad)	C_1 	
Taster schliesst die Verbindung nur so lange man drückt		
LED, Light Emitting Diode Leuchtdiode, blau		
Digitaler Pin als Output-Pin konfiguriert		
Analoger Pin als Input-Pin konfiguriert		
Kabelkreuzung ohne Verbindung		
Kabelkreuzung mit Verbindung		
Stromversorgung mit $+5\text{ V}$	$+5\text{ V}$ 	
GND, Ground oder auch <i>Masse</i> , 0 V		

4 Fiat Lux – Es werde Licht!

In diesem ersten Kapitel möchte ich euch alles Wichtige und Notwendige im Umgang mit Arduino beibringen. Wir beginnen mit einer ganz einfachen, manuellen und noch Arduino-freien Schaltung um eine LED (Leuchtdiode) in Szene zu setzen. In mehreren Etappen realisieren wir anschliessend Idee um Idee, neue Aspekte von Arduino kennen bis wir zum Schluss eine ausgefeilte, komfortable und dimmbare Steuerung für unsere LED haben – die sich ausserdem über nur einen Knopf bedienen lässt.

4.1 Vorbereitung

Ihr braucht für dieses Kapitel das Arduino-Board mit zugehörigem USB-Kabel, das Steckbrett und vorderhand zwei Kabel in den Farben Schwarz und Rot. Board und Steckbrett nebeneinander legen, mit dem schwarzen Kabel GND und die blaue Seitenreihe, mit dem roten Kabel 5V und die rote Seitenreihe verbinden (siehe auch Abbildung 8). Erst ganz am Schluss das USB am Computer anschliessen.

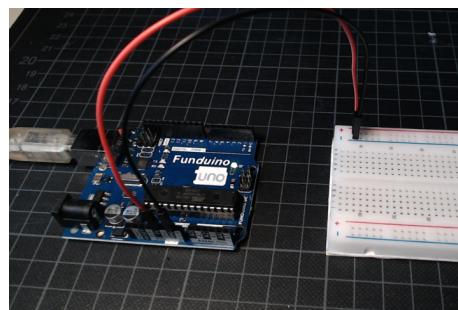


Abbildung 8: Grundaufbau für alle zukünftigen Experimente.

4.2 Direktes Ansteuern einer LED

Als erstes bauen wir uns auf dem Steckbrett die Schaltung auf, deren Schema in Abbildung 9 zu finden ist. Wem in der ersten Phase die Schemas noch zu abstrakt sind, dem oder der habe ich von den ersten zwei Aufgaben auch Photos der Schaltung erstellt.

Da dies wohl für die meisten die allererste Schaltung ist, enthält dieses Kapitel einige Hinweise und Warnungen allgemeiner art. Bitte unbedingt lesen und bei allen zukünftigen Experimenten daran denken und berücksichtigen

Kurzschlüsse Die Höhe der Spannung bei unseren Gerätschaften ist max 5 V und damit für Menschen absolut ungefährlich, ihr könnt euch keinen elektrischen Schlag holen. Dies kann dazu verleiten, Auf- und Abbau der Einfachheit halber unter Spannung zu machen, d.h. das USB-Kabel nicht abzuziehen. Auch wenn die Spannung *uns* nichts tut – ein Kurzschluss oder ein Verpolen (Plus und Minus vertauschen) bedeutet für die betroffene Komponente meistens das sichere Ende.

Merke

Umbauten (vor allem grössere) immer ohne Spannung, d.h. ohne USB-Verbindung zum Computer durchführen.

LEDs und Vorwiderstände Der Widerstand in der Schaltung ist ein sog. *Vorwiderstand*, der die LED vor Überlastung schützt. LEDs haben im Gegensatz zu ihren Vorfahren, den Glühbirnen, nahezu keinen inneren Widerstand. Sobald eine bestimmte Schwellenspannung überschritten wird, schalten sie auf Durchzug und sind wie Kurzschlüsse. Ohne Vorwiderstand würde eine LED nahezu augenblicklich durchbrennen und im Gegensatz zu Glühbirnen kann man defekte LEDs nicht so einfach visuell erkennen.

Merke

LEDs *nie* direkt sondern immer mit Vorwiderstand betreiben.

Grün:	100 Ω
Blau:	100 Ω
Rot:	200 Ω
Gelb:	200 Ω
Weiss:	200 Ω

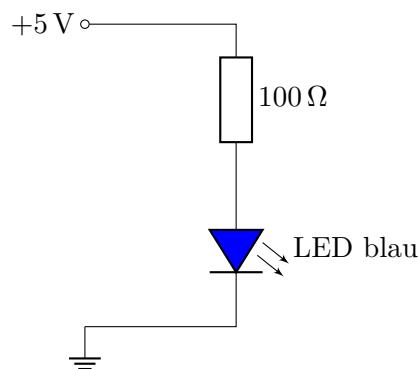


Abbildung 9: Kein unnötiger Schnick-Schnack: nur eine leuchtende LED

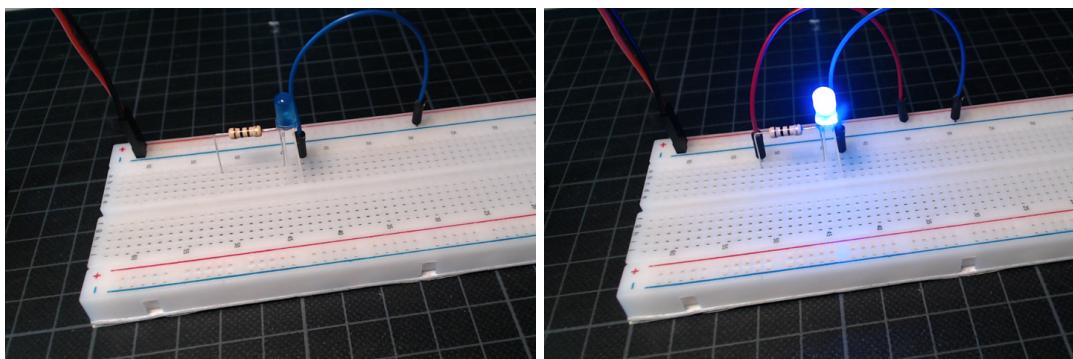


Abbildung 10: Direkte Ansteuerung einer LED

4.3 Manuelles Ansteuern via Taster

Die LED brennt nun also – aber auch nicht mehr. Um sie ein- oder auszuschalten muss man sich mit der Schaltung auskennen und die richtigen Drähte umplatzieren. Das ist nicht geraade das, was man unter einer komfortablen Licht-Steuerung versteht.

Ziel dieses Kapitel ist es, die bestehende Schaltung durch einen *Taster* zu erweitern, über welchen die LED manuell ein- oder ausgeschaltet werden kann. Das Schema mit dieser Erweiterung sehr ihr in Abbildung 11. Beachte: das neue Bauteil ist kein *Schalter*, sondern ein sog. *Taster*, der nach dem Loslassen wieder in seine Ursprungsposition (in diesem Fall in den Zustand «Aus») kehrt.

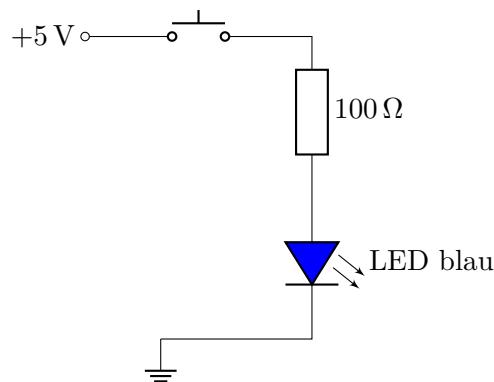


Abbildung 11: Handbetrieb für unsere Funzel.

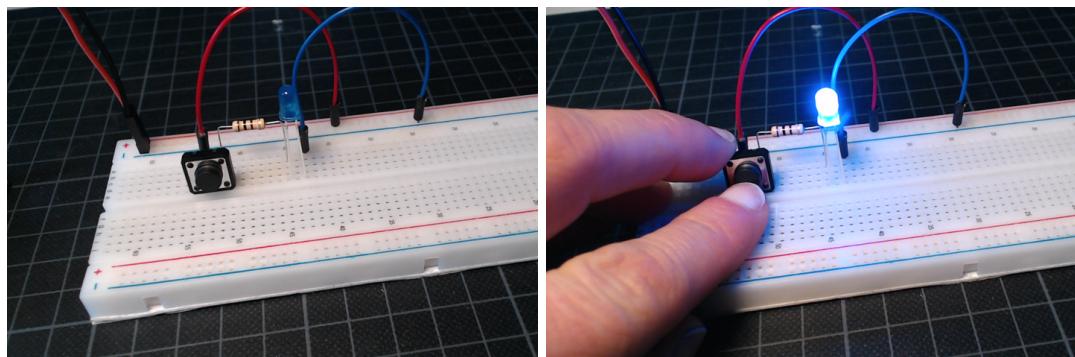


Abbildung 12: Die LED gehorcht unseren Befehlen.

4.4 Automatische Ansteuerung via Arduino

Für die Schaltungen der letzten Kapitel hätten wir den Arduino nicht gebraucht – höchste Zeit, ihn endlich einzubinden. Als erstes muss die Schaltung geringfügig verändert werden

(siehe Abbildung 13). Das Label «D5» im Schema steht für den Digital-Pin Nr. 5, welcher im Stecker-Block D auf dem Arduino-Board zu finden ist.

Anschliessend öffnet ihr eure jeweilige Arduino IDE, wählt für den Board-Typ und die Verbindung die korrekten Werte und ruft im Menu *Tools* den Eintrag *Get Board Info*. Ein kleines Dialog-Fenster mit langen Zahlen sollte als Antwort erscheinen – damit wäre der Verbindungstest auch schon fertig.

Im Editor der IDE könnt ihr nun euer erstes Programm erstellen, resp. der Einfachheit halber mit meinem Code auf Seite 12 starten. Kompiliert/Verifiziert wird das Programm mit *Sketch*→*Verify/Compile*. Wenn dieser Schritt erfolgreich war, dann könnt ihr mit *Sketch*→*Upload* euer Programm auf den Arduino laden und dort ausführen.

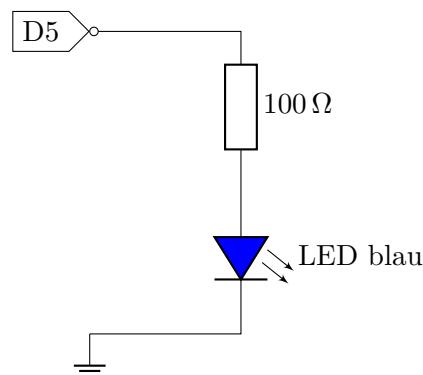


Abbildung 13: Arduino übernimmt die Steuerung.

```
1 const int ledPin = 5;
2
3 void setup() {
4     pinMode(ledPin, OUTPUT);
5 }
6
7 void loop() {
8     digitalWrite(ledPin, HIGH);
9     delay(1000);
10    digitalWrite(ledPin, LOW);
11    delay(1000);
12 }
```

Datei 1: led-blink.ino

Struktur eines Arduino-Programmes Ein Arduino-Programm (oder *Sketch*) muss also nicht viele Zeilen und Befehle umfassen, um etwas Sichtbares zu produzieren. Ich möchte am Code-Beispiel `led-blink.ino` das Wichtigste erläutern:

- Der Compiler ist ein C++ Compiler, folglich kann euer Code in C++ sein und das ganze Arsenal von Sprachbesonderheiten aufweisen.
- Jeder Arduino-Sketch muss mindestens die Funktionen `setup()` und `loop()` enthalten. Diese werden von der Laufzeitumgebung aufgerufen. `setup()` wird nur einmal zu Beginn des Programmes aufgerufen und dient dazu, die Hardware so zu konfigurieren, wie wir es für unser Programm erwarten. `loop()` dagegen wird *laufend* aufgerufen und ist mit dem *Main-Event-Loop* aus anderen Sprachen oder Umgebungen zu vergleichen.
- Einer besseren Lesbarkeit zuliebe sollten Einstellungen, wie z.B. die Pin-Nummer, über welche die LED gesteuert wird, als Namen und nicht als Zahlen im Code stehen.

Merke

Mit den Möglichkeiten von C++ lassen sich auf dem Arduino anspruchsvolle Projekte realisieren.

Befehle aus der Arduino-Standard-Library Per Default stehen einem Sketch eine Anzahl von Funktionen und Prozeduren zur Verfügung, welche die Steuerung der Hardware vereinfachen oder erst ermöglichen – im Beispiel `pinMode()`, `digitalWrite()` oder `delay()`.

Unter <https://docs.arduino.cc/language-reference/> sind alle Funktionen dieser Library dokumentiert, daher verzichte ich in diesem Dokument auf redundante Erläuterungen.

Merke

Dokumentationen lesen - eine Fähigkeit, die auch heute jedem Informatiker/jeder Informatikerin gut steht.

Viel mehr Licht Wem die eine LED zu wenig ist und er:sie sich ganze Reihen von LEDs wünscht, dem:der kann ich hier zeigen, wie sich LEDs parallel schalten lassen. Die einzige Voraussetzung ist, dass alle LEDs vom gleichen Typ sein müssen (also bspw. nur blaue oder nur gelbe). Ein Beispiel mit 7 LEDs seht ihr in Abbildung 14. Das Interessante dabei ist, dass es nur einen Vorwiderstand braucht und zwar von der gleichen Sorte wie wenn ich eine LED ansteuern würde.

Merke

Parallelenschaltung von LEDs ist möglich, jedoch nur mit typengleichen LEDs.

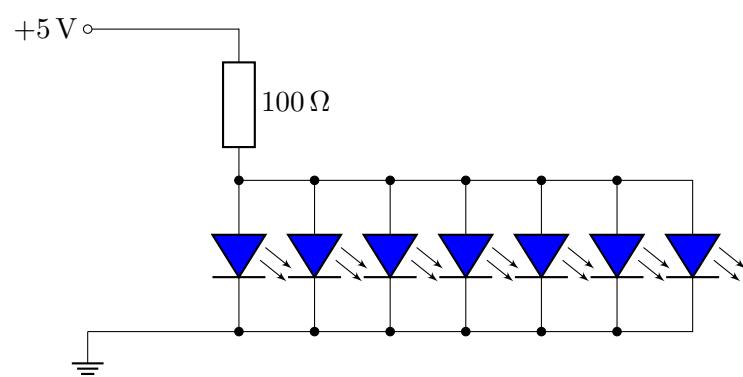


Abbildung 14: Parallelschaltung von mehreren LEDs

Aufgaben

Übung 4.1 (Timing) Wie ist die Zahl in der Funktion `delay()` zu verstehen? Welche weiteren Funktionen stehen in Zusammenhang mit dem Timing zur Verfügung und was sagt die Dokumentation betr. Zuverlässigkeit und Verwendungszweck?

Wie schnell ist unser Arduino eigentlich? Oder anders gefragt: wie lange braucht er, um die `loop()`-Funktion zu durchlaufen? Versuche ein Vorgehen zu entwickeln, mit dem du solche Fragen beantworten kannst.

Übung 4.2 (Dynamischer Blinker) Mach den Blinker dynamischer! Im Main-Loop gibt es zwei `delay()`-Aufrufe. Mit der einen Zeit wird kontrolliert, wie lange pro Blink-Zyklus die LED eingeschaltet wird, und mit der zweiten Zeit, wie lange sie pro Blink-Zyklus ausgeschaltet ist.

Unter Beibehaltung der Zeit für einen Blink-Zyklus von 2 Sekunden, könnte man das Verhältnis zwischen diesen beiden Zeiten laufend verändern und so spannende Blink-Muster generieren.

Oder die Blinkfrequenz wird über einen gewissen Zeitraum verändert: immer und immer schneller und danach wieder langsamer. Tob dich aus!

4.5 Taster-gesteuerte Ansteuerung via Arduino

Unser Projekt nimmt langsam Fahrt auf und es wird Zeit, die Schaltung selber zu finanzieren und alles Weitere dann in der Software zu realisieren. In diesem Kapitel geht es darum, den Taster (resp. den Zustand des Tasters) vom Arduino her lesen zu können und entsprechend zu reagieren. Die erste Zwischenetappe besteht darin, das Verhalten der Schaltung in Abbildung 11 nachzubilden. Die LED soll also nur so lange brennen, wie der Taster gedrückt wird.

In Abbildung 15 seht ihr, wie der Taster eingebunden wird. Zum Lesen des Taster-Zustands verwenden wir Digital-Pin Nr. 2. Auf Seite 16 findet ihr außerdem ein minimales Programm als Starthilfe.

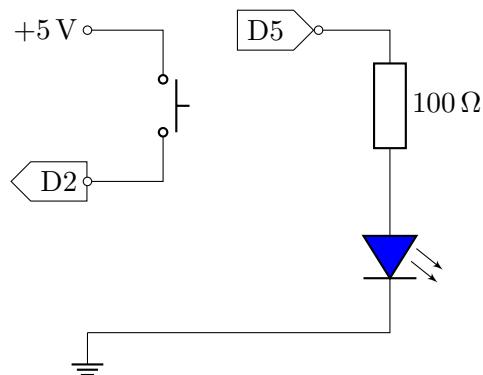


Abbildung 15: Der Status des Tasters wird über Pin D2 gelesen.

```
1 const int ledPin = 5;
2 const int buttonPin = 2;
3
4 void setup() {
5     pinMode(ledPin, OUTPUT);
6     pinMode(buttonPin, INPUT);
7 }
8
9 void loop() {
10    int state = digitalRead(buttonPin);
11    if (state == HIGH) {
12        digitalWrite(ledPin, HIGH);
13    } else {
14        digitalWrite(ledPin, LOW);
15    }
16 }
```

Datei 2: push-button.ino

Wenn ihr das Programm laufen lässt, werdet ihr verwundert feststellen, dass die LED beim Drücken des Tasters zwar angeht, aber beim Loslassen des Tasters einfach weiterbrennt. Das Phänomen dahinter ist sehr und trägt den Namen *floating pin* oder *treibender Pin*. Abbildung 16 zeigt in einem zeitlichen Verlauf, wie es dazu kommt und was sich dahinter verbirgt.

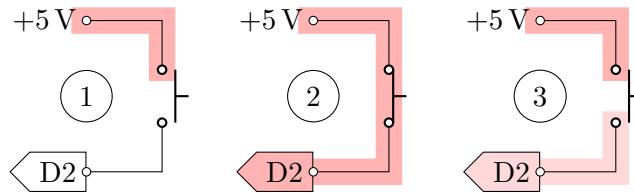


Abbildung 16: Wie ein Pin gewissermassen *davontreibt*.

1. Der Arduino wurde eben gerade zurückgesetzt oder eingeschaltet und ein neues Programm wurde geladen. Der Taster ist offen, das Potential von +5 V reicht bis zum oberen Anschluss des Tasters und der Pin D2 liest auf der ihm zugeführten Leitung *low*.
2. Der Taster wird gedrückt und das 5 V-Potential breitet sich über den Taster hinaus bis zum Arduino, wo Pin D2 nun ein *high* registriert.
3. Der Taster wird wieder losgelassen, aber die Spannung auf dem unteren Teil des Tasters, dem Kabel zu D2 und auf D2 selber wird nicht abgebaut. Das Spannungs-potential aus Schritt 2 bleibt relativ lang auf diesem Teil der Schaltung liegen und führt zum Fehler.

Man nennt dieses Problem *floating pin* und bezeichnet damit den unklaren, resp. *treibenden* Spannungszustand von D2 im Bild 3.

Zum Glück ist die Lösung recht einfach: mit einem hochohmigen Widerstand binden wir den treibenden Pin an *Ground* an (siehe Abbildung 17). Wenn der Schalter offen ist, dann wird über diesen Widerstand die Verbindung zu D2 ge-groundet, d.h. auf ein klares Spannungsniveau von 0 V gebracht.

Da man im gezeigten Fall den treibenden Pin gegen Masse verbindet (in der Schemazeichnung gewissermassen nach *unten zieht*), nennt man den neuen Widerstand auch *pull-down* Widerstand.

Übung 4.3 (Schaltung mit *pull-up* Widerstand) Analog zu einem pull-down gibt es natürlich auch einen pull-up Widerstand, resp. eine Schaltung, wo einer der Widerstände einen treibenden Pin nach oben zu 5 V zieht.

Baue die aktuelle Schaltung so um, dass dieses Verhalten abgebildet wird und (fast noch wichtiger): stelle das Programm so um, dass ein Benutzer von diesem Umbau nichts merkt, d.h. die LED soll auch in dieser Version eingeschaltet werden, wenn der Benutzer den Taster drückt und beim Loslassen soll die LED wieder erlöschen.

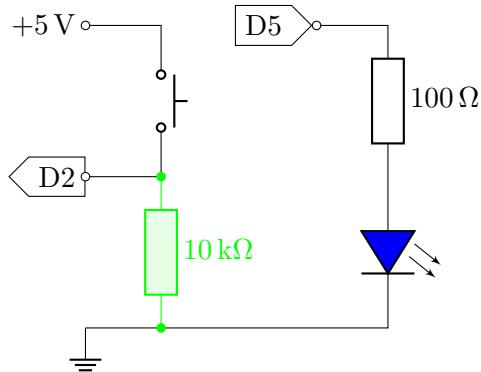


Abbildung 17: Mit einem *pull-down* Widerstand schafft man eindeutige Logik-Level.

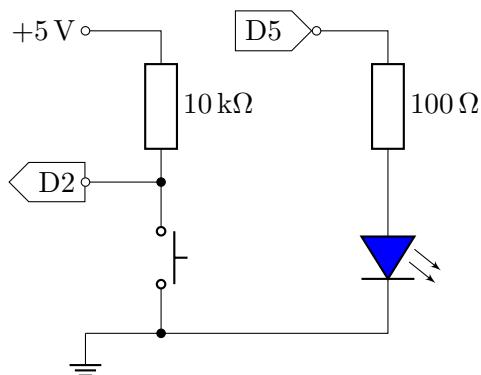


Abbildung 18: Alternative Schaltung mit einem *pull-up* Widerstand. Auf den Code hat dies einige Auswirkungen!

Pull-Up Widerstände kommen in unzähligen Schaltungen zum Einsatz. Sie sind sogar so weit verbreitet, dass die meisten Mikrocontroller bei ihren Pins einen *internen, elektronisch zuschaltbaren* Pull-Up Widerstand eingebaut haben! Bei der Konfiguration des Pins mit der Funktion `pinMode` können wir anstelle von `INPUT` die Konstante `INPUT_PULLUP` verwenden und von unserem Steckbrett den Pull-Up Widerstand wieder entfernen.

Abbildung 19 zeigt die finale Version unserer LED-Schaltung. Das bedeutet nicht, dass wir uns mit der aktuellen Variante der Ansteuerung zufrieden geben! Aber alle weiteren Schritte erfolgen in der Software.

Übung 4.4 (Doku-Studium) Sind alle Pins des Arduino mit einem internen Pull-Up Widerstand ausgerüstet? Und wie sieht es mit Pull-Down Widerständen aus? Kennt unser Arduino auch interne Pull-Down Widerstände?

Übung 4.5 (Ein- und Ausschalten der LED mit nur einer Taste) Die nächste Aufgabe besteht darin, mit Hilfe des Tasters (und natürlich etwas Software-Logik) die LED

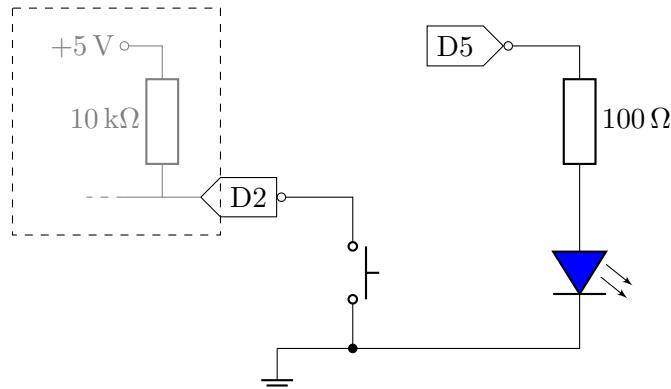


Abbildung 19: Und schliesslich die finale Version mit aktiviertem internen *pull-up* Widerstand beim Pin D2.

ein- und ausschalten zu können. Konkret soll mit jedem Druck³ auf den Taster die LED zwischen den Zuständen HIGH und LOW hin und herwechseln.

4.5.1 Helligkeit verändern

Aktuell kann unser Programm die LED nur in zwei Zuständen betreiben: *EIN* oder *AUS*. Dabei würden sich Leuchtdioden grundsätzlich stufenlos dimmen lassen – höchste Zeit also, dieses Feature auszuprobieren und in unser Programm einzubauen.

Neben den digitalen Ausgängen kennt Arduino auch eine Handvoll *analoger* Ein- und Ausgänge. Damit könnte man eine LEDs dimmend ansteuern aber es gibt eine weitaus einfachere und erst noch digitale Möglichkeit: mit *PWM* (*pulse width modulation*) oder *Pulsweitenmodulation* kann (fast⁴) jeder digitale Pin ein Signal erzeugen, mit dem sich u.a. LEDs perfekt dimmen lassen.

Dazu wechselt der Output des Pins im Mikrosekundenbereich zwischen *LOW* und *HIGH*. Mit einer Funktion kann das zeitliche Verhältnis zwischen *LOW* und *HIGH* verändert werden und damit auch die Helligkeit der LED. Das Beispiel lässt eine an Pin 5 angeschlossene LED pulsieren.

```

1 const int ledPin = 5;
2 int ledValue = 0;
3 int step = 5;
4
5 void setup() {
6   pinMode(ledPin, OUTPUT);

```

³Überleg dir gut und am besten vorher, wann deine Software den Taster als *gedrückt* interpretieren soll: wenn jemand auf den Taster drückt? Oder erst wenn man den gedrückten Taster wieder loslässt? Spielt es eine Rolle, wie lang oder kurz jemand den Taster drückt?

⁴Alle Pins, die PWM-fähig sind, haben neben ihrer Zahl auf dem Arduino-Board ein Tilde-Zeichen aufgedruckt.

```

7 }
8
9 void loop() {
10   analogWrite(ledPin, ledValue);
11   ledValue = ledValue + step;
12   delay(25);
13   if (ledValue == 0 || ledValue == 255) {
14     step = -step;
15   }
16 }
```

Datei 3: pulse-led.ino

Bemerkenswert ist bei diesem Beispiel lediglich die Funktion `analogWrite()`, die auch auf digitale Pins angewendet werden kann – sofern sie PWM-fähig sind. Als Wert akzeptiert `analogWrite()` eine Zahl $x \in [0, 255]$.

Übung 4.6 (Fade-In, Fade-Out) *Schreibt ein Programm, mit welchem die LED über den Taster ein- bzw. ausgeschaltet werden kann, aber diesmal soll die Helligkeit der LED beim Einschalten sanft von Dunkel nach Hell wechseln und beim Ausschalten ebenso. Die Dauer, welche für das Ein-, resp. Ausblenden verwendet wird, sollte als Konstante im Programm veränderbar sein. Es lohnt sich auch hier zuerst zu überlegen, wie sich das System verhalten soll: wird bereits beim Drücken des Tasters Ein-, bzw. Ausgeblendet oder erst wenn der Taster gedrückt und wieder losgelassen ist? Wie soll sich das System verhalten, wenn bereits während des Ein-/Ausblendens erneut die Taste gedrückt wird? Etc...*

4.5.2 Programmsteuerung mit Interrupts

Ein Merkmal aller bisher erstellten Sketches ist, dass die Verarbeitung von externen Signalen mittels *Polling* realisiert ist. Das bedeutet, dass unser Programm in einer Endlossschleife laufend den Zustand des Tasters (oder eines anderen Gerätes) abruft und in den meisten Fällen gar nichts zu tun hat, weil sich der Zustand nicht verändert hat.

Eine Alternative dazu ist die Verarbeitung mittels *Interrupts*. Dabei wird die Beobachtung und Kontrolle der Peripherie dem Mikrocontroller übergeben, welcher unseren Code erst dann aufruft, wenn sich tatsächlich etwas verändert hat.

Beim Arduino UNO können nur die Pins 2 und 3 für Interrupts verwendet werden. Andere Boards kennen vergleichbare Einschränkungen. Die gesamte Konfiguration wird über eine einzige Funktion geregelt: `attachInterrupt()`. Der Sketch `interrupt-led.ino` zeigt beispielhaft, wie mit Interrupts gearbeitet werden kann.

Das Programm enthält eine neue, eigene Funktion, `InterruptHandler()`, welche beim Eintreffen eines Interrupts vom Microcontroller aufgerufen werden soll. *Wichtig:* eine derartige Funktion muss genau das gezeigte Profil aufweisen (d.h. keine Parameter und keine Rückgabewerte haben), muss so kurz wie möglich gehalten werden, resp. so schnell

wie möglich terminieren und darf keine der Zeit-Funktionen (wie `delay()`, `millis()` oder `micros()`) verwenden!

Mit dem Aufruf von `attachInterrupt()` auf Zeile ?? 17 wird die Hardware angewiesen den Pin 2 zu überwachen und im Fall eines Wechsels von `HIGH` nach `LOW` (das wird als *fallende Flanke/FALLING* bezeichnet) die Funktion `InterruptHandler` aufzurufen. Es gibt neben `FALLING` noch weitere Ereignisse, auf die man sich per Interrupt abonnieren kann – mehr dazu in der offiziellen Doku von Arduino.

Die `loop()`-Funktion hat damit keine Relevanz mehr, muss aber aus Gründen der Syntax trotzdem vorhanden sein – halt einfach leer.

```
1 const int ledPin = 5;
2 const int buttonPin = 2;
3 bool ledIsOn = false;
4
5 void InterruptHandler() {
6     ledIsOn = !ledIsOn;
7     if (ledIsOn) {
8         digitalWrite(ledPin, HIGH);
9     } else {
10        digitalWrite(ledPin, LOW);
11    }
12 }
13
14 void setup() {
15     pinMode(ledPin, OUTPUT);
16     pinMode(buttonPin, INPUT_PULLUP);
17     attachInterrupt(digitalPinToInterrupt(buttonPin), \label{line:add-int}
18                     InterruptHandler, FALLING);
19 }
20
21 void loop() {}
```

Datei 4: `interrupt-led.ino`

Übung 4.7 (Fade-In/Fade-Out, per Interrupt initiiert) Baut euer Programm dahingehend um, dass ihr Anstelle des Pollings mit Hilfe eines Interrupts auf die Betätigung des Tasters wartet. Das Licht der LED soll wiederum langsam hoch-, resp. heruntergefahren werden – allerdings darf dies nicht im Kontext der Interrupt-Funktion erfolgen! Wie schon erwähnt ist deren Laufzeit auf ein absolutes Minimum zu reduzieren.

4.5.3 Die Unzulänglichkeiten der Hardware

Möglicherweise habt ihr es auch schon bemerkt: manchmal macht die Schaltung nicht genau das, was sie gemäss Programmierung und Schema machen sollte. Ich spreche

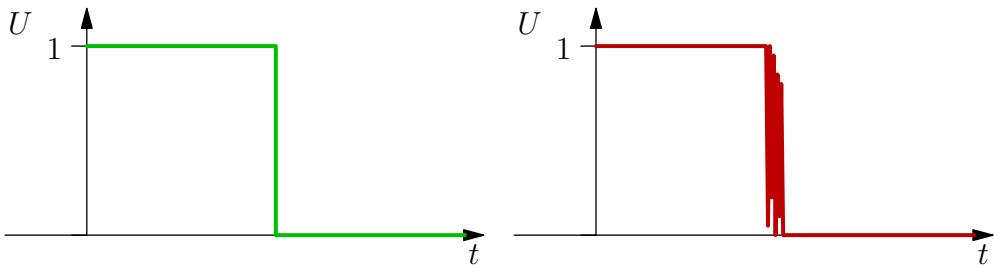


Abbildung 20: Wie wir uns den Schaltvorgang vorstellen (links) und wie er in der Realität stattfindet (rechts).

hier nicht von möglichen Fehlern in euren Programmen oder von Fehlern beim Zusammenstecken, sondern von Situationen, in denen eine (1) Betätigung des Tasters gleich mehrere Ausführungen der Interrupt-Funktion zur Folge hat. Auch beim Polling lassen sich vergleichbare Effekte beobachten – als ob Taster und Arduino etwas zu euphorisch unterwegs wären.

Dieses Problem hat mit der Funktionsweise des Tasters zu tun und mit dem Umstand, dass unsere Hardware sehr schnell und präzise arbeitet. Wenn wir uns den Schliessvorgang des Tasters in Superzeitlupe und unter dem Mikroskop vorstellen und während des Schliessens die Spannung über dem Schalter messen (Schalter offen: 5 V; Schalter geschlossen: 0 V), dann zeigt sich, dass beim Schaltvorgang mehrmals zwischen 0 V und 5 V hin- und hergesprungen wird.

Man bezeichnet diesen Effekt als *Schalterprellung* (engl. *Bouncing*) und man kann sich gut vorstellen, wie dieses Auf und Ab von der präzisen Messung des Arduino als mehrere, sehr kurz aufeinanderfolgende Schaltvorgänge interpretiert wird. Um dies in Zukunft zu unterbinden muss der Schalter *entprellt* werden – dazu gibt es sowohl Hardware-, als auch Software-basierte Lösungen.

Es ist zwar sehr spannend und lehrreich die Entprellung per Software zu realisieren aber zum einen belegt das Speicher und Ressourcen in unserem Controller und zum anderen gibt es elegante Lösungen aus der Signaltechnik welche komplett wartungsfrei und gut skalierbar sind. Die Schaltung in Abbildung 21 zeigt diese Lösung anhand zweier Beispiele: links ein Schalter mit *active high* (d.h. wenn man drückt, geht das Signal zum Eingang D2 auf 5 V) und rechts ein Schalter mit *active low* (wenn man drückt, geht das Signal auf Masse).

Übung 4.8 (Fade-In/Fade-Out, per Interrupt initiiert und entprellt) *In dieser Aufgabe geht es darum, alle bekannten Techniken zu kombinieren, um einen Schalter für unsere LED zu bauen, der:*

- ... das Licht sanft hoch, resp. herunter fährt.
- ... mit Polling oder Interrupt realisiert ist (die Entscheidung überlasse ich euch).

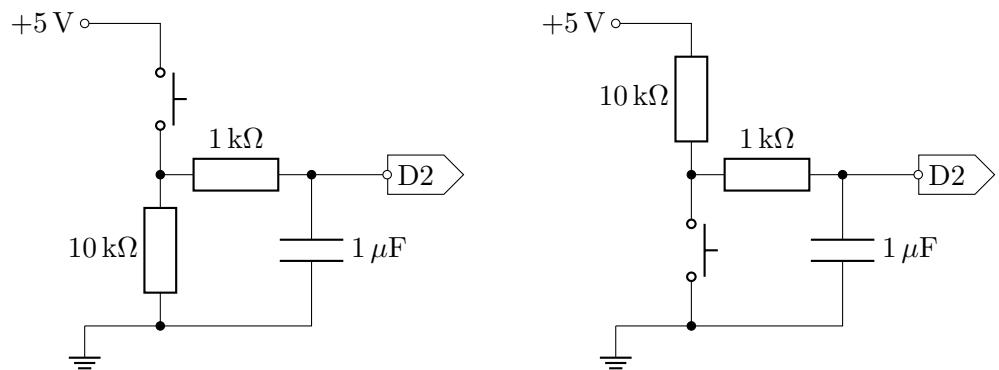


Abbildung 21: Entprellung eines Schalters mit Hilfe eines sog. RC-Glieds.

- ...per Software oder Hardware entprellt ist (auch diese Entscheidung ist euch überlassen).

Die Dauer des Fade-In, resp. Fade-Out sollen als Konstanten in der Software verankert sein. Und wieder gilt es gut zu überlegen, wie sich eure Schaltung in einer Ausnahmesituation verhalten soll und sie entsprechend zu bauen.

5 Der C Präprozessor

Wer mit C oder C++ arbeitet, dem:der sind bestimmt schon die `#include`-Zeilen aufgefallen, die sich am Anfang in praktisch allen Source-Dateien finden lassen. Diese Zeilen werden nicht vom Compiler verarbeitet, sondern von einem, dem Compiler vorgelagerten Tool: *Dem Präprozessor* oder *Makro-Prozessor*.

Die Funktionen des Präprozessors kann man für eigene Zwecke verwenden und damit einige erstaunliche Dinge anstellen, die in anderen Programmiersprachen nur schwer zu bewerkstelligen sind – und das möchte ich euch in diesem Abschnitt erkl”ren.

Ich erkläre am Ende des Abschnittes auch, wie man unter Linux/UNIX den Präprozessor eigenständig, also ohne Teil eines Compiliervorganges zu sein, verwenden kann.

5.1 Textverarbeitung

Den Präprozessor kann man sich wie eine programmierbare Textverarbeitung vorstellen. Sie funktioniert grundsätzlich unabhängig von der Programmiersprache und erlaubt es, in einer Textdatei folgende Mutationen vorzunehmen:

- Den Inhalt einer weiteren Datei an einer bestimmten Stelle in der Source-Datei einzusetzen.
- In Abhängigkeit von Schaltern bestimmte Abschnitte ein- oder ausblenden.
- Mithilfe von *Makros* Textersetzung durchführen.

Im Gegensatz zu einer klassischen Textverarbeitung macht der Präprozessor diese Mutationen nicht *in place* in der Source-Datei, sondern bei jedem Compiliervorgang.

5.2 Dateien einsetzen

Mit der Anweisung `#include` kann der Inhalt einer beliebigen Datei in die Source-Datei eingebunden werden und zwar exakt an jener Stelle, wo diese Anweisung steht.

```
#include <(_fileName_)>
#include "(_fileName_)"
```

Ob der Dateiname mit `<>` oder mit `""` angegeben wird, hat Einfluss auf die Verzeichnisse, in denen der Präprozessor nach dieser Datei sucht. `<>` wird für Dateien aus Systemverzeichnissen verwendet, `""` dagegen für Dateien aus Benutzerspezifischen Verzeichnissen.

Beispiel Eine konkrete Anwendung, welche auch im Umfeld von Arduino verwendet wird, ist die Auslagerung von sensiblen Angaben (wie Benutzernamen oder Passwörter) in eigene Dateien. Bei der Weitergabe oder der Veröffentlichung von Code müssen diese sensiblen Daten nicht jedesmal entfernt werden

```
1 #include <Arduino.h>
2 #include "Secrets.h"
```

Datei 5: Include.ino

```
1 const char *Username = "charly123";
2 const char *Password = "admin1234";
```

Datei 6: Secrets.h

5.3 Abschnitte ein-/ausblenden

Um ganze Abschnitte einer Datei in Abhängigkeit bestimmter Schalter ein- oder auszublenden, existieren die Befehle `#if`, `#else`, `#elif` und `#endif` sowie die Shortcuts `#ifdef` (steht für `#if defined(...)`) und `#ifndef` (für `#if !defined(...)`). Außerdem können die logischen Verknüpfungen `!` (für NOT), `&&` (für AND) und `||` (für OR) verwendet werden.

Im folgenden Beispiel wird die Ausgabe mit `Serial.println` nur dann im Code belassen, wenn der Schalter `VERBOSE` gesetzt wurde:

```
#ifdef VERBOSE
    Serial.println("Trace message");
#endif
```

Diese Schalter werden zum grössten Teil von der Entwicklungsumgebung gesetzt, können aber auch in der Source-Datei mit dem Befehl `#define` direkt gesetzt werden. Damit im obigen Beispiel die `println`-Funktion im Code bleibt, muss irgendwo vor diesen Zeilen der Schalter `VERBOSE` wie folgt gesetzt werden:

```
#define VERBOSE
```

Die oben gezeigten Befehle können wie ihre Pendants in C oder C++ verschachtelt werden, traditionellerweise werden sie jedoch *nicht eingerückt* – jedenfalls nicht so, wie man vermuten würde. Siehe folgendes Beispiel aus `stdio.h`

```
#if defined(__USE_XOPEN) || defined(__USE_XOPEN2K8)
# ifdef __GNUC__
# ifndef __VA_LIST_DEFINED
typedef __gnuc_va_list va_list;
# define __VA_LIST_DEFINED
# endif
# else
# include <stdarg.h>
# endif
#endif
```

Beispiel In Arduino-Programmen wird der Output von Textmeldungen über die serielle Verbindung nur während der Entwicklung verwendet. In einer «produktiven» Variante des Codes sollten diese Zeilen ausgeblendet werden. Die Initialisierung der seriellen Verbindung in `setup()` wird mit folgenden Zeilen in Abhängigkeit des Schalters `DEBUG` im Code belassen oder komplett entfernt:

```
#define DEBUG

void setup() {
#ifndef DEBUG
    Serial.begin(9600);
    while (!Serial) {}
#endif
    ...
}
```

Übung 5.1 Was ist der Unterschied zu folgender Lösung, welche komplett in C, resp. C++ realisiert ist?

```
const bool DEBUG = true;

void setup() {
    if (DEBUG) {
        Serial.begin(9600);
        while (!Serial) {}
    }
    ...
}
```

5.4 Makros verwenden

Mit `#define` lassen sich nicht nur Schalter setzen, sondern auch sog. Makros definieren, mit denen einzelne Code-Sequenzen ersetzt werden können. In C oder C++ werden sehr oft Konstanten auf diese Weise gepflegt. Die Geschwindigkeit der seriellen Verbindung könnte man wie folgt durch das Makro `BAUD_RATE`⁵ ersetzen.

```
#define BAUD_RATE 9600

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

⁵Traditionellerweise werden Kontanten, die als Makros definiert sind, mit Grossbuchstaben bezeichnet.
Dies erleichtert die Unterscheidung zwischen Makro-Konstanten und C-Konstanten im Code.

So können wichtige Einstellungen konzentriert am Anfang der Datei vorgenommen werden und sind nicht über den ganzen Code verstreut.

Ausserdem können Makros wie Funktionen definiert und verwendet werden. Dazu schreibt man bei der Definition direkt anschliessend an den Namen des Makros runde Klammern und die Namen der Parameter – wie bei einer Funktion. Auf die Angabe von Datentypen kann man verzichten, da der Präprozessor keinerlei Kenntnis solcher Dinge hat.

Folgendes Beispiel zeigt ein Makro namens `DEBUG_PRINT`, mit welchem Debugging-Meldungen vereinheitlicht und über einen Schalter bei Bedarf komplett aus dem Code entfernt werden können:

```
#define DEBUG
#define DEBUG_PREFIX "DEBUG: "

#ifndef DEBUG
# define DEBUG_PRINT(msg) Serial.println(DEBUG_PREFIX ## msg)
#else
# define DEBUG_PRINT(msg) {}
#endif
```

Mithilfe von Makros lassen sich beispielsweise auch typenübergreifende (sog. generische) Mathe-Funktionen erstellen:

```
#define MAX(x, y) ((x > y) ? x : y)
#define MIN(x, y) ((x < y) ? x : y)
#define ABS(x)      ((x > 0) ? x : -x)
```

6 Links

- Arduino; Hauptseite der Firma *Arduino*
<https://www.arduino.cc/>
- Arduino, IDE 2; Downloadpage für die integrierte Entwicklungsumgebung 2.0 von Arduino
<https://www.arduino.cc/en/software>
- Arduino, Language Reference; Dokumentation der klassischen C++ Funktionen wie `pinMode()` oder `delay()`
<https://docs.arduino.cc/language-reference/>
- Datasheet, ATmega328P von Microchip; Die Originaldokumentation des Mikrocontrollers
<https://tinyurl.com/mrxnh5t3>⁶

⁶Wo eine URL zu lang für den Satz geworden ist, habe ich sie mit tinyurl.com gekürzt.