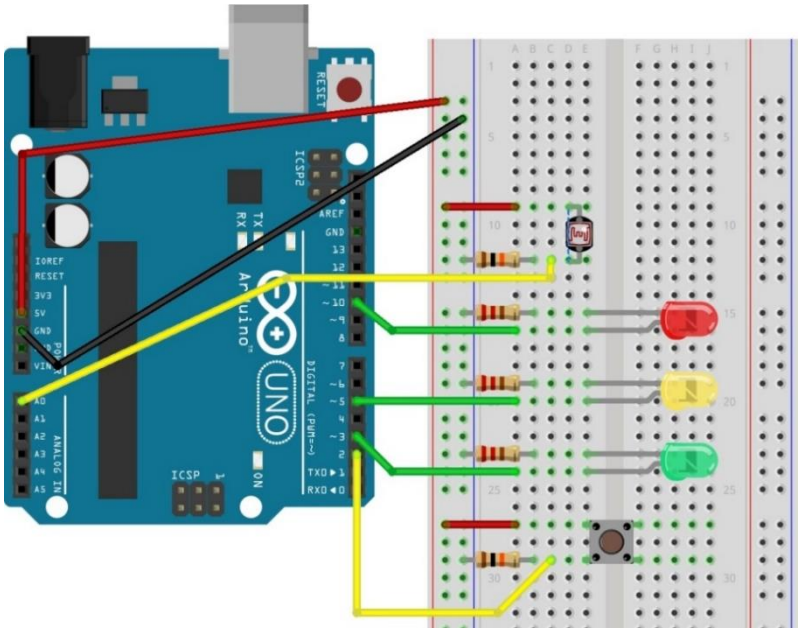
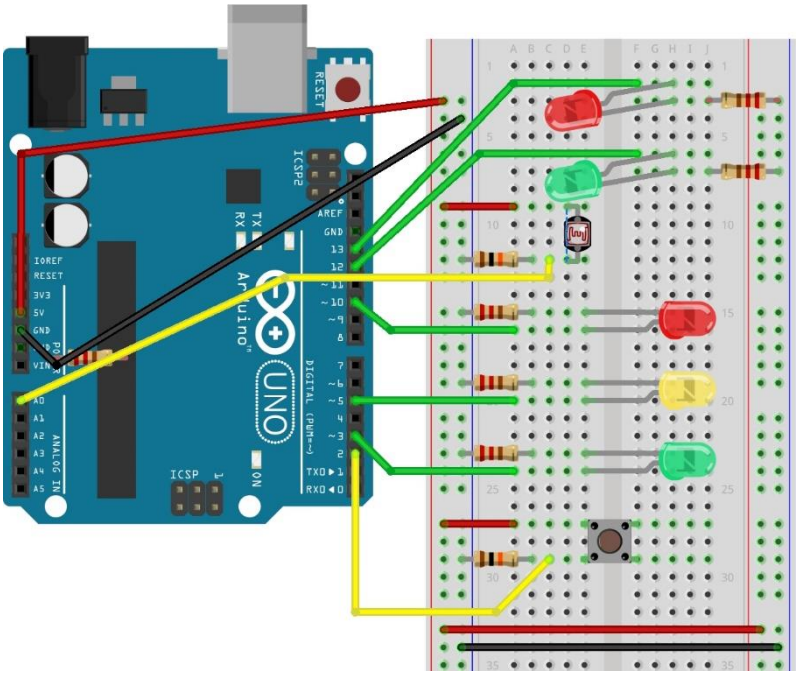
Arduino – Programmierpraxis mit Schall

In diesem Arbeitspaket werden die Grundlagen des Programmierens, die du im zweiten Arbeitspaket erlernt hast, wiederholt und gefestigt. Zusätzlich wirst du auch weitere nützliche Programmierkonzepte und Schaltungselemente kennen lernen, zunächst den Piezo-Lautsprecher für erste Experimente mit Schall.

Dabei kannst du wieder die ursprüngliche Schaltung zur Programmierung einer Ampelsteuerung oder eines Morseapparats verwenden:



Sollte bei deiner Schaltung auch die Erweiterung mit der Fußgängerampel „gesteckt“ sein, so kann diese (insbesondere für die erste Aufgabenstellung zu „Schallerzeugung mit dem Arduino“) ebenso weiterverwendet werden…



Der **Piezo-Lautsprecher** ist der (große) schwarze Bauteil, der nebenstehender Abbildung ähnelt. Dieser Bauteil kann **direkt** – d.h. ohne einen Ohm’schen Widerstand – **in eine Schaltung eingebaut** werden, indem einer der beiden Kontakte mit einem der freien digitalen Kontakte des Arduino-Boards, und der andere mit „Erde“ („GND“, 0 Volt) verbunden wird.

Für die Programmierung des Piezo-Lautsprechers werden zwei Varianten vorgestellt. Dabei wird davon ausgegangen, dass der Lautsprecher am **digitalen Pin mit der Nummer ~6** angeschlossen wurde:

- Codefragmente zur Programmierung mit `digitalWrite(…)`:

```
int piezoPin;

void setup() {
  piezoPin = 6;
  pinMode(piezoPin,OUTPUT);
}

void loop() {
  digitalWrite(piezoPin, HIGH);
  delay(50);
  digitalWrite(piezoPin, LOW);
  delay(50);
}
```

```
int piezoPin;

void setup() {
  piezoPin = 6;
  pinMode(piezoPin,OUTPUT);
}

void loop() {
  digitalWrite(piezoPin, HIGH);
  delayMicroseconds(50000);
  digitalWrite(piezoPin, LOW);
  delayMicroseconds(50000);
}
```

  Analog zu Leuchtdioden kann der Piezo-Lautsprecher mit dem Befehl `digitalWrite(pin,HIGH)` ein- bzw. mit dem Befehl `digitalWrite(pin,LOW)` wieder ausgeschaltet werden.

  Die Zeitspanne, während der am Piezo-Lautsprecher Spannung anliegt (bzw. nicht anliegt), kann entweder mit dem bekannten Befehl `delay(Anzahl der Millisekunden)` oder alternativ mit dem, für die Programmierung des Lautsprechers bisweilen geeigneteren, Befehl `delayMicroseconds( Anzahl der Mikrosekunden)` festgelegt werden. Dabei gilt:

  – 1 Sekunde hat 1000 Millisekunden;
  – 1 Millisekunde hat 1000 Mikrosekunden.

  Bei den beiden Codefragmenten beträgt dieser Zeitraum also jeweils 50 Millisekunden.

  Laut Arduino-Onlinehilfe (https://www.arduino.cc/reference/en/language/functions/time/delaymicroseconds/) sind beim Befehl `delayMicroseconds` für die Anzahl der Mikrosekunden Werte zwischen 3 Mikrosekunden (µs) und 16383 µs sinnvoll.

- Codefragmente zur Programmierung mit `tone(…)`:

```
int piezoPin;

void setup() {
  piezoPin = 6;
  pinMode(piezoPin,OUTPUT);
}

void loop() {
  tone(piezoPin,440);
  delay(2000);
  noTone(piezoPin);
  tone(piezoPin,396);
  delay(1000);
  noTone(piezoPin);
}
```

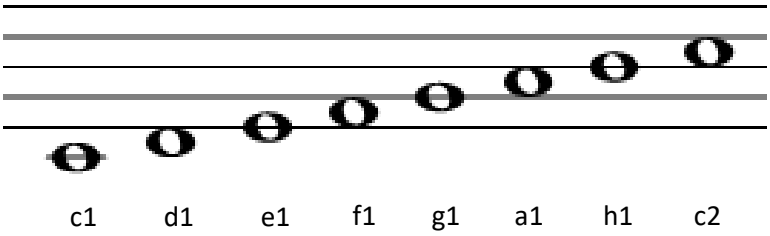
  Für das Programmieren von „Melodien“ ist es zweckmäßiger, den Befehl `tone(pin, Frequenz des Tons)` zu verwenden. Die Frequenz eines Tons legt dabei die Tonhöhe fest und wird in Hertz (Hz) angegeben.

  Im dargestellten Codefragment wird zunächst für zwei Sekunden der Kammerton, das „eingestrichene a“ (auch als a1 bezeichnet) mit der Frequenz 440 Hertz, sodann das „eingestrichene g“ (g1) mit der Frequenz 396 Hertz für eine Sekunde „gespielt“. Nach jedem der Töne wird der Piezo-Lautsprecher dann mit dem Befehl `noTone`(pin) wieder abgeschaltet.

  Die folgende Tabelle gibt die den „eingestrichenen“ Tönen zugeordneten Frequenzen (in Hertz) wieder:

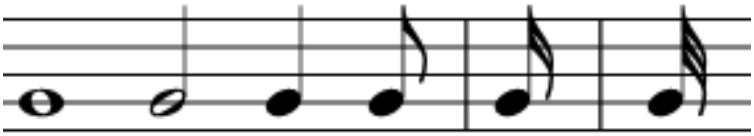
Ton:	**c**	**cis/des**	**d**	**dis/es**	**e**	**f**	**fis/ges**	**g**	**gis/as**	**a**	**ais/b**	**h**
Frequenz:	264	278,4	297	316,8	330	352	371,3	396	422,4	440	475,2	495

In der Musik werden diese Töne und ihre Tonhöhe als Noten in Notenzeilen notiert. Eine Notenzeile besteht aus fünf parallelen Linien. Die nachfolgende Abbildung zeigt die Notation für die sieben eingestrichenen Stammtöne als ganze Noten (mitsamt dem zweigestrichenen c):



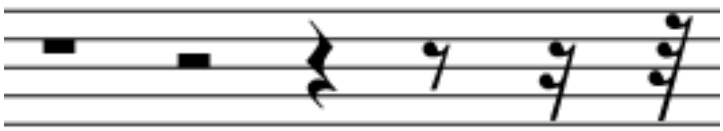
Die restlichen in der Tabelle enthaltenen Töne entstehen aus den Stammtönen mit Hilfe von sogenannten Versetzungszeichen, die vor die Note in die Notenzeile geschrieben werden und die Tonhöhe um einen Halbton erhöhen oder erniedrigen. Die Erhöhung um einen Halbton erfolgt mit dem Kreuz-Versetzungszeichen, ♯, und die entsprechenden Töne heißen dann cis, dis, fis, gis und ais. Die Erniedrigung um einen Halbton erfolgt mit dem b-Versetzungszeichen, ♭, und die entsprechenden Töne heißen dann des, es, ges, as und b (anstelle von hes).

Neben der Tonhöhe wird durch die Notenschrift auch die Dauer, für die der Ton erklingt, notiert – man unterscheidet ganze Noten, halbe Noten, Viertelnoten, Achtelnoten, Sechzehntelnoten und Zweiunddreißigstelnoten. Die Symbole für diese Notenwerte siehst Du von links nach rechts in nachfolgender Abbildung (falls die Noten auf einer der oberen Zeilen geschrieben werden, weisen die „Stiele“ nach unten):

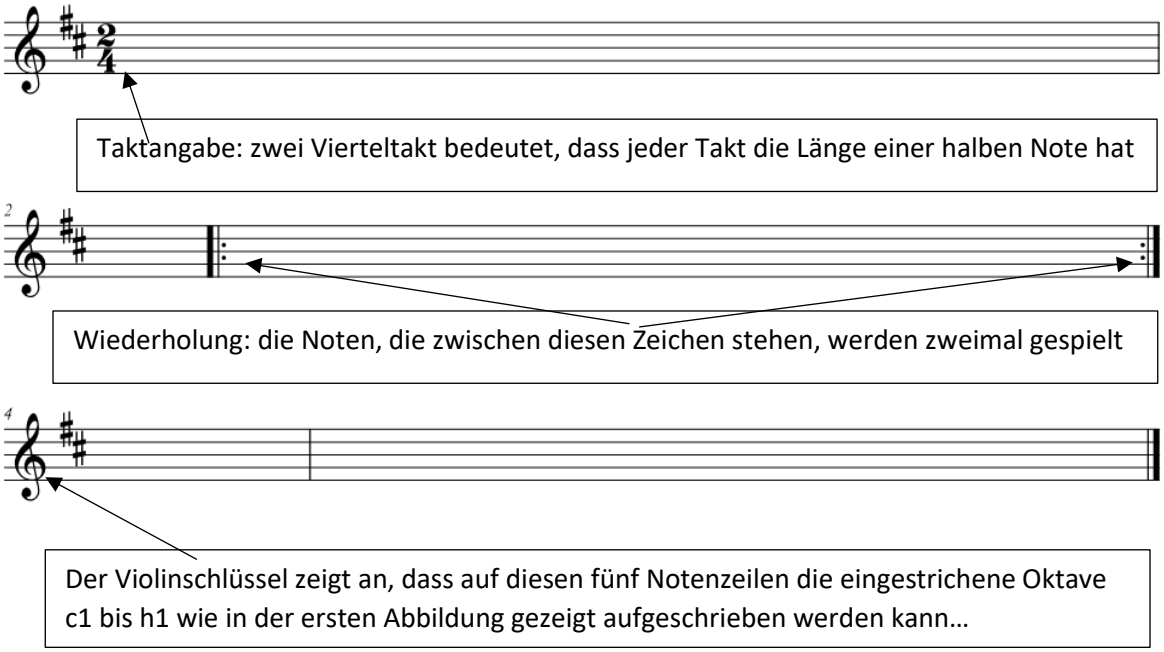


Dabei gilt: Ein Ton, der durch eine halbe Note dargestellt wird, dauert halb so lang wie einer, der durch eine ganze Note dargestellt wird, ein Ton, der durch eine Viertelnote dargestellt wird, dauert halb so lang wie einer, der durch eine halbe Note dargestellt wird usw.

Zusätzlich zu Tönen lassen sich in der Notenschrift auch Pausen notieren – so wie bei den Noten unterscheidet man auch bei den Pausen zwischen ganzer Pause, halber Pause, Viertelpause usw., wobei eine ganze Pause genauso lange dauert wie eine ganze Note. Nachfolgend sind die Pausenzeichen bis zur Zweiunddreißigstelpause von links nach rechts dargestellt:



Ein großer Vorteil der Notation von Tönen in Notenzeilen ist, dass alle Töne eines Musikstücks zusammen in einem „Behälter“ aufgeschrieben werden können – man verwendet dazu auf einem Notenblatt mehrere untereinander angeordnete Notenzeilen oder auch mehrere solcher Notenblätter. In der folgenden Abbildung siehst du ein Beispiel für drei solcher untereinander angeordneter Notenzeilen. Zusätzlich sind noch einige weitere Symbole erklärt, die für das Lesen von Notenzeilen wichtig sind:



Für das Programmieren von Melodien wäre es praktisch, auch in der Programmiersprache einen „Behälter“ zur Verfügung zu haben, in dem der Reihe nach allen Frequenzen und Längen der Töne einer Melodie gespeichert werden können – bislang kennen wir aber nur „Behälter“, in denen ein einziges Datum gespeichert werden kann, nämlich (einfache) Variable. Wollten wir mit solchen (einfachen) Variablen auch nur den ersten Takt von „Alle meine Entchen“ (vier Achtelnoten d1 – e1 – f1 – g1) codieren, bräuchten wir für die vier Noten insgesamt acht Variable:

```
int frequenz1 = 297;
int duration1 = 200;
int frequenz2 = 330;
int duration2 = 200;
int frequenz3 = 352;
int duration3 = 200;
int frequenz4 = 396;
int duration4 = 200;
```

Tatsächlich gibt es aber auch Variable, in denen mehr als ein Wert gespeichert werden kann. Man nennt solche Variablen **Feldvariable** oder **Arrays**. In der Programmiersprache C ist es zudem nur notwendig, **nach dem Namen der Variablen ein Paar eckiger Klammern** zu schreiben, um eine Feldvariable des gewünschten Datentyps zu vereinbaren und gleichzeitig zu initialisieren, z.B. also:

```
int frequenzen[] = {297, 330, 352, 396};
```

oder:

```
int durations[] = {200, 200, 200, 200};
```

Wie du siehst, können bei der Initialisierung einer solchen Feldvariable die Werte, die in der Feldvariablen gespeichert werden sollen, in geschwungenen Klammern und durch Beistriche getrennt angegeben werden.

Selbstverständlich müssen wir nun unsere **Modellvorstellung von Variablen erweitern**, um auch mit Feldvariablen gut arbeiten zu können:

Auch bei der **Deklaration einer Feldvariable** wird ein **passender Speicherbereich reserviert**, nur müssen eben vier `int`-Speicherbereiche reserviert werden, wenn in der Feldvariablen vier ganze Zahlen gespeichert werden sollen, z.B. also:

```
int frequenzen[] = {297, 330, 352, 396};
```

…weil der Feldvariablen **vier Werte** zugewiesen werden, werden **vier aufeinanderfolgende Speicherbereiche** reserviert; `frequenzen` verweist auf den ersten dieser Speicherbereiche (das ist der Speicherbereich mit der niedrigsten Adresse)

147	229	244	23	166	50
178	135	135	213	175	181
255	0	196	213	230	62
255	0	196	213	230	62

…durch die Wertzuweisung an die Feldvariable werden in den reservierten Speicherbereiche auch die Werte, die ursprünglich darin gespeichert sind, durch die zugewiesenen Werte ersetzt:

147	229	244	23	166	50
178	~~135~~ 297	~~135~~ 330	~~213~~ 352	~~175~~ 396	181
255	0	196	213	230	62
255	0	196	213	230	62

Ganz wesentlich für das Arbeiten mit Feldvariablen in der Programmiersprache C ist auch, dass die **Feldvariable auf den ersten der reservierten Speicherbereiche verweist.** Dies bedeutet:

Soll auf den **Wert** zugegriffen werden,

…der **im ersten Speicherbereich** gespeichert ist, muss man **keinen Speicherbereich** weitergehen – dies schreibt man in unserem Beispiel als `frequenzen[0];`

…der im **zweiten Speicherbereich** gespeichert ist, muss man **einen Speicherbereich** weitergehen – dies schreibt man in unserem Beispiel als `frequenzen[1];`

…der im **dritten Speicherbereich** gespeichert ist, muss man **zwei Speicherbereiche** weitergehen – dies schreibt man in unserem Beispiel als `frequenzen[2];`

…der im **vierten Speicherbereich** gespeichert ist, muss man **drei Speicherbereiche** weitergehen – dies schreibt man in unserem Beispiel als `frequenzen[3]`.

Die Zahl in der eckigen Klammer, die angibt, wie weit man von der Startadresse – d.h. vom ersten für die Feldvariable reservierten Speicherbereich – weitergehen muss, um auf den gewünschten Wert zugreifen zu können, nennen wir auch den **Index** des betreffenden im Feld gespeicherten Wertes.

Statt Deklaration einer Feldvariablen und Wertzuweisung an die Feldvariable wie im Beispiel

```
int frequenzen[] = {297, 330, 352, 396};
```

„gleichzeitig“ durchzuführen, kann man auch so codieren:

```
int frequenzen[4];    // Feldgröße als Zahl in den eckigen Klammern
frequenzen[0] = 297; // Wertzuweisung an den 1.(!) Speicherbereich
frequenzen[1] = 330; // Wertzuweisung an den 2.(!) Speicherbereich
frequenzen[2] = 352; // Wertzuweisung an den 3.(!) Speicherbereich
frequenzen[3] = 396; // Wertzuweisung an den 4.(!) Speicherbereich
```

Ebenso, wie an einen der Speicherbereiche ein Wert geschrieben wird, kann auch der dort gespeicherte Wert gelesen werden – man verwendet den Namen der Feldvariable und in der eckigen Klammer die Zahl, die dem jeweiligen Speicherbereich zugeordnet ist. Beispielsweise könnte der erste Takt von „Alle meine Entchen“ nun so codiert werden:

```
tone(piezoPin, frequenzen[0]);
delay(durations[0]);
noTone(piezoPin);
delay(200);
tone(piezoPin, frequenzen[1]);
delay(durations[1]);
noTone(piezoPin); delay(200);
tone(piezoPin, frequenzen[2]);
delay(durations[2]);
noTone(piezoPin); delay(200);
tone(piezoPin, frequenzen[3]);
delay(durations[3]);
noTone(piezoPin); delay(200);
```

Selbstverständlich…

…selbstverständlich kann dieses Codefragment mit Hilfe einer **Programmschleife** noch sparsamer codiert werden. Dazu verwendet man eine **Zählvariable**, die der Reihe nach die gewünschten **Indexwerte** annimmt – einen Vorschlag für solchen Programmcode findest du nachfolgend:

```
int index;                    // Zählvariable für die Indexwerte
index = 0;                    // Start bei Indexwert 0
while (index <= 3) {          // Schleifenbedingung(*)
      tone(piezoPin, frequenzen[index]);
      delay(durations[index]);
      noTone(piezoPin);
      delay(200);
      index = index + 1;      // Indexwert um 1 erhöhen
}                             // …wegen der Schleifenbedingung (*)
                              // wird auf die Werte in den Feldern an
                              // den Indexpositionen 0, 1, 2 und 3
                              // zugegriffen
```