Skript zum Programmablauf des Tic-Tac-Toe Workshops

Der Arduino - Ein Mikrocontroller zum Ansteuern

Was ist ein Mikrocontroller?

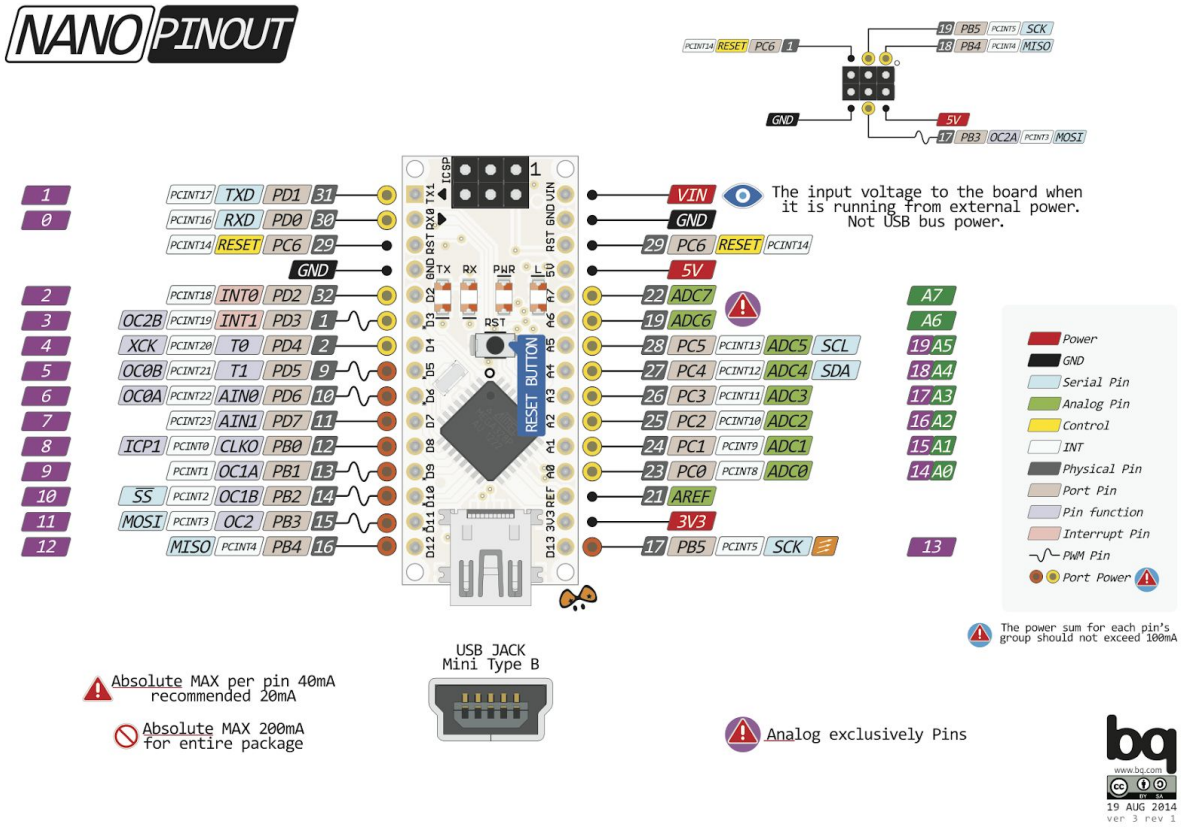
Das Herz der Platine bildet der Mikrocontroller: ein Arduino. Ein Mikrocontroller besitzt einen kleinen Mikroprozessor, der Befehle ausführt, und einen Speicher, auf den ein Programm geladen werden kann. Jedes mal, wenn der Arduino mit Strom betrieben wird, macht er nichts anderes, als das Programm auszuführen, das sich im Speicher befindet. Mit dem Mikroprozessor kann er komplexe Berechnungen durchführen und das in einer überragend kurzen Zeit. Dies liegt daran, dass der Mikroprozessor viel, viel schneller arbeitet als ein Mensch. Mit seinen GPIO Pins kann der Mikrocontroller auch nach außen hin kommunizieren und somit seine Berechnungen Teilen.

Was können seine GPIO Pins?

Ein Mikroprozessor alleine bringt einem nicht wirklich etwas, wenn er nur intern Aufgaben erledigen kann. Man könnte mit einem Taschenrechner beispielsweise nichts anfangen, wenn dieser in der Lage ist, jede beliebige Rechnung auszurechnen, aber keine Möglichkeit hat, das Ergebnis dieser Rechnung auszugeben. Dafür hat der Arduino seine GPIO Pins. Diese Pins sind quasi Alleskönner.

Bevor die Funktionen der GPIO Pins erklärt werden, ist es aber wichtig zu wissen, dass der Arduino intern mit nur 2 Zuständen arbeitet: Strom an oder Strom aus. Diese werden auch logische 1 oder logische 0 genannt; HIGH oder LOW.

Eigentlich haben die GPIO Pins nicht wirklich überragende Funktionen. Ein GPIO Pin kann entweder als Eingang oder als Ausgang genutzt werden. Wenn man ihn als Eingang verwendet, kann man ablesen, ob an dem Pin der Zustand HIGH oder LOW anliegt. HIGH bedeutet in diesem Fall eine Spannung von +5V und 0 bedeutet, dass an diesem Pin 0V anliegen. Wenn der Pin als Ausgang gesetzt ist, kann man an ihm die Zustände HIGH oder LOW ausgeben. Mit diesen GPIO Pins kann der Arduino nun mit seiner Außenwelt kommunizieren.
Einige Pins haben noch die besondere Funktion zu bieten, dass sie mit einem Analog-DIgital-Wandler verbunden sind. Der AD-Wandler wird im folgenden Abschnitt genauer erläutert.
Alle GPIO Pins haben eine Nummer, die ihnen zugeordnet ist. Damit kann man die 22 GPIO Pins auf dem Arduino Nano genau ansprechen. Die Nummer, die der jeweilige Pin trägt, steht auf dem Arduino, bzw. kann in einer sogenannten Pinout nachgeschaut werden.

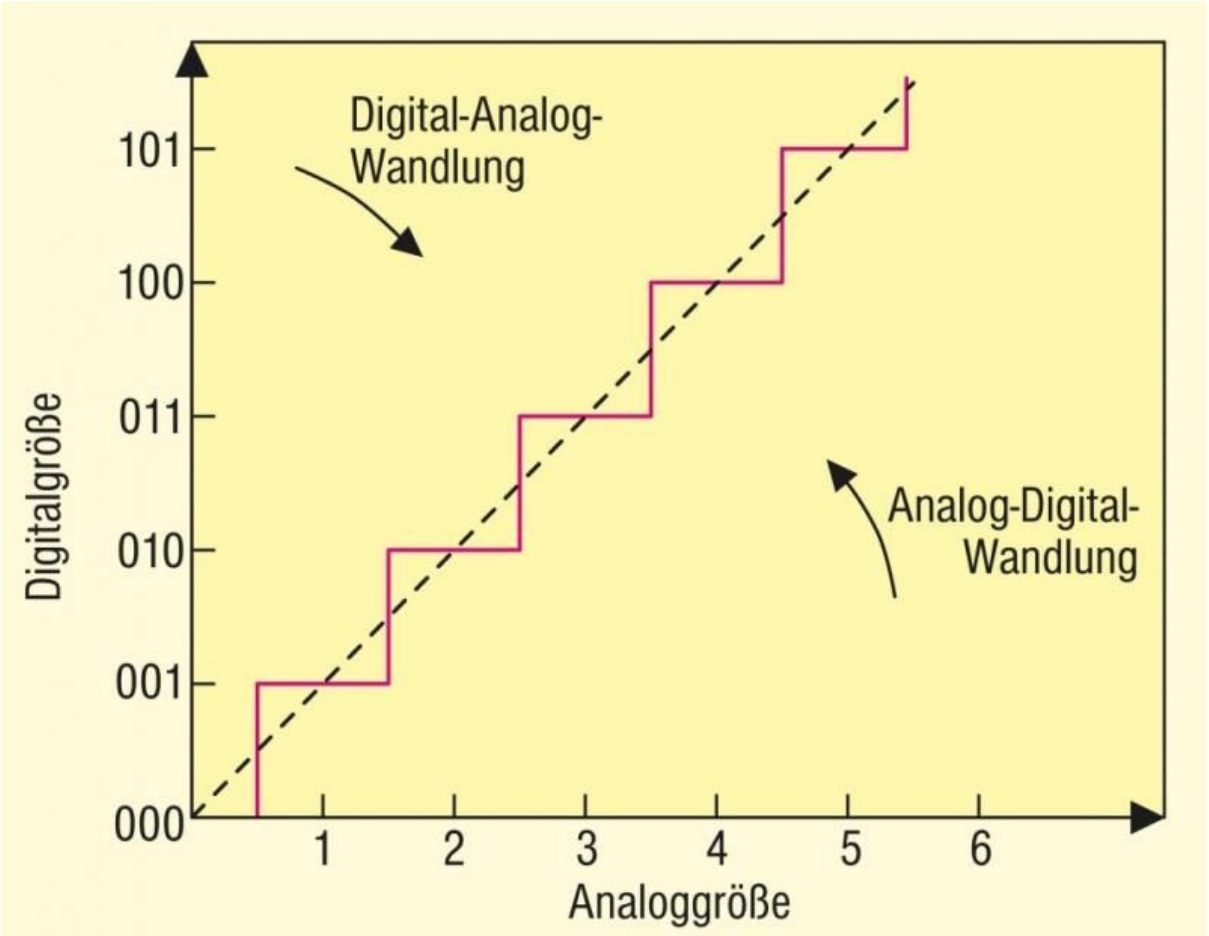


http://www.pighixxx.com/test/wp-content/uploads/2014/11/nano.png

Die Pinout verrät einiges mehr über die Pins am Arduino, denn einige Pins haben Zusatzfunktionen, die in diesem Workshop aber nicht benötigt werden. Wichtig ist, dass der Arduino die GPIO-Pins 0 bis 19 als Eingang und als Ausgang nutzen kann. Die Pins 14 bis 15 können außerdem noch als AD-Wandler mit den Nummern A0 bis A5 genutzt werden. Die Pins A6 und A7 können nur als AD-Wandler genutzt werden. Es ist wichtig, beim Programmieren die Namen der Pins mit Mehrfachbelegung nicht zu verwechseln. Möchte man einen Pin als Eingang oder Ausgang verwenden, dann muss man die Ziffer verwenden, die ihm zugeordnet ist. Möchte man ihn aber als AD-Wandler verwenden, muss man die Ziffer mit dem vorgestellten A verwenden.

Analog-Digital-Wandlung

Wie schon erwähnt hat der Arduino mehrere Analog-Digital-Wandler, Kurz: AD-Wandler. Ein AD-Wandler liest die Spannung ein, die am Pin anliegt und gibt je nach Spannung eine Zahl aus. Liegt die Spannung an diesem Pin zwischen 0V und 5V, so gibt der AD-Wandler eine Zahl zwischen 0 und 1023 aus. Die ausgegebene Zahl verhält sich proportional zu der Spannung, die anliegt.



http://www.wissen.de/sites/default/files/styles/lightbox/public/wissensserver/jadis/incoming/227111.jpg?itok=O66xlAJq ; 07.11.2017

Kurz gesagt: mit dem AD-Wandler wird die Spannung eingelesen, die an dem GPIO Pin anliegt. Der AD-Wandler wird im Verlauf des Workshops verwendet, um die gedrückte Taste einzulesen.

Grundlegende Syntax von C++ Programmen für den Arduino

Ein Programm für einen Arduino wird in der Programmiersprache C++ geschrieben. Die Programmiersprache beschreibt, was das Programm machen soll und ist nur für Menschen lesbar. Soll das Programm auf einem Arduino ausgeführt werden, wird es durch den Compiler in die Maschinensprache übersetzt und auf den Arduino hochgeladen.

Zu Beginn haben wir ein Arduino-Programm, das nichts macht. Füllen wir es doch mit etwas inhalt, der auch nichts macht: Kommentaren. Überall im Programmcode können Kommentare stehen. Alles, was innerhalb dieser Kommentare steht, wird vom Compiler ignoriert. Ein Kommentar beginnt entweder mit zwei schrägstrichen beginnen und bis zum Zeilenende gehen oder zwischen /* und */ über mehrere Zeilen gehen.

```
// Dies ist ein Kommentar
```

```
/* Dies ist auch ein Kommentar
 *  Der über mehrere Zeilen geht
 *  Und in der nächsten Zeile endet
 */
```

Kommentare haben eine ganz wichtige Funktion im Programmcode. Wenn man z.B. längere Zeit nicht mehr an diesem Code gearbeitet hat und dann weiter machen möchte, wird man sich fragen, warum der Code das tut, was drinnen steht. Um also den Überblick zu behalten, schreibt man Kommentare rein, die den Code erklären. Wichtig ist, dass in den Kommentaren nicht steht, was der Code macht, sonder warum er es macht.

Das Programm für den Arduino besteht aus 2 Teilen: dem Setup- und dem Loop-Teil. Der Setup-Teil wird nur ein einziges mal ausgeführt, wenn der Arduino eingeschaltet wird. Der Loop-Teil hingegen wird nach dem Setup-Teil endlos oft wiederholt.In der Programmiersprache C++ werden dem Arduino Anweisungen gegeben, die dieser der Reihe nach ausführt. Jede Anweisung endet mit einem Semikolon.
Ein kleines Beispiel: Für die ansteuerung der LEDs werden die GPIO-Pins mit den Nummern 2 bis 15 als Ausgang verwendet. Die Pins müssen nur ein einziges mal als Ausgang gesetzt werden. Dies wird mit der Anweisung pinMode erledigt. Um alle Pins zu Beginn des Programms auf Ausgang zu setzen, sieht die Setup-Funktion folgendermaßen aus:

```
void setup() {
  pinMode(2, OUTPUT);
  pinMode(3, OUTPUT);
  pinMode(4, OUTPUT);

  ...

  pinMode(14, OUTPUT);
  pinMode(15, OUTPUT);
}
```

Die Arduino IDE (das Programm, mit dem die Programme für den arduino geschrieben werden) markiert bestimmte Schlüsselwörter farbig. pinMode und OUTPUT sind also Schlüsselwörter. Noch einmal ein Zusammenfassung, was pinMode so macht: zuerst steht in der Klammer die Ziffer des Pins, der angesprochen werden soll. Danach folgt entweder das Schlüsselwort INPUT oder OUTPUT. Bei Input wird der Pin als Eingang gesetzt, bei OUTPUT als Ausgang.

Wie schon einmal erwähnt kann ein als Ausgang gesetzter Pin verwendet werden, um an ihm die Zustände HIGH und LOW (+5V und 0V) auszugeben. Dieser Zustand wird mit der Anweisung digitalWrite gemacht.
Als Beispiel werden hier mal die GPIO Pins 2, 3 und 4 auf HIGH, bzw. die Ping 5, 6 und 7 auf LOW gesetzt, um zu zeigen, wie das funktioniert:

```
  digitalWrite(2, HIGH);
  digitalWrite(3, HIGH);
  digitalWrite(3, HIGH);

  digitalWrite(5, LOW);
  digitalWrite(6, LOW);
  digitalWrite(7, LOW);
```

Ein Pin kann während das Programm läuft jederzeit seinen Zustand ändern. Er muss nicht bis zum Programmende fest in einem Zustand verharren. Das folgende Beispiel demonstriert, wieder Pin zwischen möglichen Zuständen in einer Endlosschleife hin und her wechselt.

```
void loop() {
  pinMode(2, OUTPUT);

  digitalWrite(2, HIGH);

  digitalWrite(2, LOW);

  pinMode(2, INPUT);
}
```

dadurch, dass dieser Programmcode im Loop-Teil steht, wird er immer wieder durchlaufen.

Ein wichtiger Bestandteil in C++ sind Variablen. Eine variable ist ein kleiner Datencontainer, der etwas speichern kann. In diesem Workshop werden nur 2 Typen von Variablen verwendet: Integer- und Boolean Variablen. Eine Integer-Variable kann eine Zahl speichern. Um eine Integer-Variable zu erzeugen, braucht man für sie einen Namen. Sagen wir mal, wir möchten die Variable “AnzahlSchuesse” in einem Videospiel verwenden, dann erzeugen wir diese, indem wir *int AnzahlSchuesse;* in den Programmcode schreiben. Warum “ue” anstelle von “ü”? - Weil nur Groß-und Kleinbuchstabe, Ziffern und der Unterstrich (_) im Namen einer Variable vorkommen darf. Wichtig ist auch, dass der Variablenname etwas über die Funktion der Variable aussagt. “Zahl1”, Zahl2” und “Zahl3” eignen sich als Namen nicht so gut wie “ersterSummand”, “zweiterSummand” und “Summe”.
Mit der Variable können wir nun machen, was wir wollen. Weil Variablen ja Datencontainer sind, brauchen diese einen Wert. Mit *AnzahlSchuesse = 0;*
bekommt die Variable den Wert 5 und behält diesen, bis sie mit einem neuen Wert überschrieben wird.
Variablen eignen sich auch hervorragend, um mit ihnen zu rechnen. Wenn in der Variable “ersterSumand” der Wert 5 gespeichert ist und in der Variable “zewiterSummand” der Wert 10, wie bekommen wir das Ergebnis dann in die Variable “Ergebnis”? Das Programm dazu sieht so aus:

```
int ersterSummand = 5;
int zweiterSummand = 10;
int Summe = ersterSummand + zweiterSummand;
```

Mit dem + kann man die Werte zweier Variablen addieren. Dieses Beispiel zeigt auch, dass eine Variable direkt bei ihrer Erzeugung einen Wert zugewiesen bekommen kann. Alle 4 Grundrechenarten lassen sich mit den Variablen realisieren und auch bei Variablen gilt: Klammer vor Punkt vor strich!
Zu dem gerade gezeigten Beispiel gibt es noch eine ganz wichtige Anmerkung: Variablen existieren erst, nachdem sie erzeugt wurden. Würde der Programmcode Abschnitt so aussehen:

```
int Summe = ersterSummand + zweiterSummand;
int ersterSummand = 5;
int zweiterSummand = 10;
```

dann wäre das ein Fehler. Programmcode wird immer von oben nach unten gelesen. In diesem Fall sollen 2 Variablen addiert werden, die zu diesem Zeitpunkt noch nicht existieren. Der Compiler gibt dafür eine Fehlermeldung aus.

Es gibt noch einen weiteren Typ von Variablen: bool. Eine Variable vom Typ Bool kann nur 2 Zustände annehmen: true oder false. Mit ihnen kann nicht gerechnet werden, wie mit den Integer-Variablen. Sie finden aber vor allem in Kombination mit Zugriffsbeschränkungen häufig Verwendung.

Wichtig: Variablen haben einen bestimmten Gültigkeitsbereich. Wo dieser liegt, wird im Verlauf des Workshops noch genauer geklärt.

Das nächste wichtige Werkzeug in C++ sind Zugriffsbeschränkungen. Es gibt Gründe, warum bestimmte Programmcode-Abschnitte nur unter bestimmten Bedingungen ausgeführt werden dürfen. Ein paar Beispiele wären:

- Vor einer Division ist es sinnvoll, zu prüfen, ob man aus versehen durch 0 dividiert
- Es dürfen keine weiteren Lotto-Zahlen gezogen werden, wenn schon 6 gezogen wurden
- Das Raumschiff darf nicht mehr schießen, wenn die eigene Munition leer ist oder alle eigenen Kanonen getroffen wurden

Deshalb hat man die Zugriffsbeschränkungen erfunden. Eine Zugriffsbeschränkung sieht folgendermaßen aus:

```
if( AnzahlSitzplaetze < AnzahlPassagiere )
{
  // Es wird eng
}
```

Hier wird gezeigt, dass der if-Block nur dann ausgeführt wird, wenn die Bedingung in der Klammer erfüllt ist. Falls die Bedingung nicht erfüllt wird, kann ein zusätzlicher else-Block angehängt werden:

```
if( Koerpergroesse < Mindestgroesse )
{
  // Such dir eine andere Achterbahn
}
else
{
```

```
    // Los, steig ein
}
```

Hier wird genau das gezeigt, was für alle Achterbahnen in einem Freizeitpark gilt: Man kann sie entweder fahren oder nicht. Je nachdem, ob die Bedingung wahr ist oder nicht, wird nur der if-Block oder der else-Block ausgeführt.
Nun aber mal zu der Bedingung: Man kann Variablen mit Variablen oder Werten unterschiedlich Vergleichen. Im vorherigen Beispiel wurde gezeigt, wie man die Bedingung formulieren muss, wenn eine Variable kleiner sein soll als die andere. Die folgende Tabelle listet alle möglichen Vergleichsoperatoren auf, die es gibt:

Operator	Bedeutung
<	kleiner als
<=	kleiner oder gleich
>	größer als
>=	größer oder gleich
==	genau gleich
!=	ungleich

Warum ist der Vergleichsoperator nicht nur ein einfaches, sondern ein doppeltes Ist-Gleich Zeichen? Das einfache “=” ist schon als Zuweisungsoperator reserviert, um Variablen einen Wert zuzuweisen. Deshalb ist der Operator zum Vergleichen das “==”. Diese beiden Operatoren dürfen nicht miteinander verwechselt werden.

Eins noch zur Bedingung: Man kann mehrere Teilbedingungen zu einer Gesamtbedingung verknüpfen. Wenn ich sage, dass ich die Achterbahn fahren werde, wenn ich groß genug dafür bin und die schlange davor nicht zu lang ist, wäre das hier eine mögliche Lösung:

```
if( Koerpergroesse > Mindestgroesse && schlangeZuLang == false)
{
  // Jetzt geht's los!
}
```

In diesem Beispiel werden 2 Teilbedingungen zu einer Gesamtbedingung verknüpft. Die gesamtbedingung ist erst wahr, wenn beide Teilbedingungen wahr sind. Der doppelte senkrechte Strich (||) steht für ein logisches Oder und das Ausrufezeichen (!) invertiert das Ergebnis einer Bedingung. Die folgende Wahrheitstabellen liefern eine Übersicht über die verschiedenen Verknüpfungsmethoden:

Bedingung A	Bedingung B	A && B
Falsch	Falsch	Falsch

Falsch	Wahr	Falsch
Wahr	Falsch	Falsch
Wahr	Wahr	Wahr

Bedingung A	Bedingung B	A || B
Falsch	Falsch	Falsch
Falsch	Wahr	Wahr
Wahr	Falsch	Wahr
Wahr	Wahr	Wahr

Bedingung A	! A
Falsch	Wahr
Wahr	Falsch

Nun geht es zum Schluss des C++-Teils dieses Workshops: Schleifen. Mit Schleifen kann man Programmabschnitte wiederholen, bis eine Bedingung eintritt. Die einfachste Schleife ist die While-Schleife. Solange die Bedingung erfüllt ist, wird der Programmabschnitt ausgeführt. Hier ein kleines Beispiel: Es soll solange gewartet werden, bis die Schlange vor der Achterbahn kleiner geworden ist.

```
while(schlangeZuLang == true)
{
  // Warte
}
```

Diese Schleife prüft nach jedem Durchlauf, ob die Bedingung stimmt und entscheidet dann, ob ein erneuter Durchlauf getätigt werden soll, oder nicht. Sollte die Bedingung schon vor dem ersten Durchlauf nicht stimmen, wird die Schleife einfach übersprungen.
Ein anderer schleifentyp ist die for-Schleife, die zum Zählen gedacht ist. Sagen wir mal, wir möchte nicht länger als 1000 mal in der Schlange in der Achterbahn warten, weil wir sonst zu ungeduldig werden, dann zählen wir einfach mit, wie oft diese Schleife durchlaufen wurde:

```
for(int i = 0; i < 1000; i++)
{
  // warte
}
```

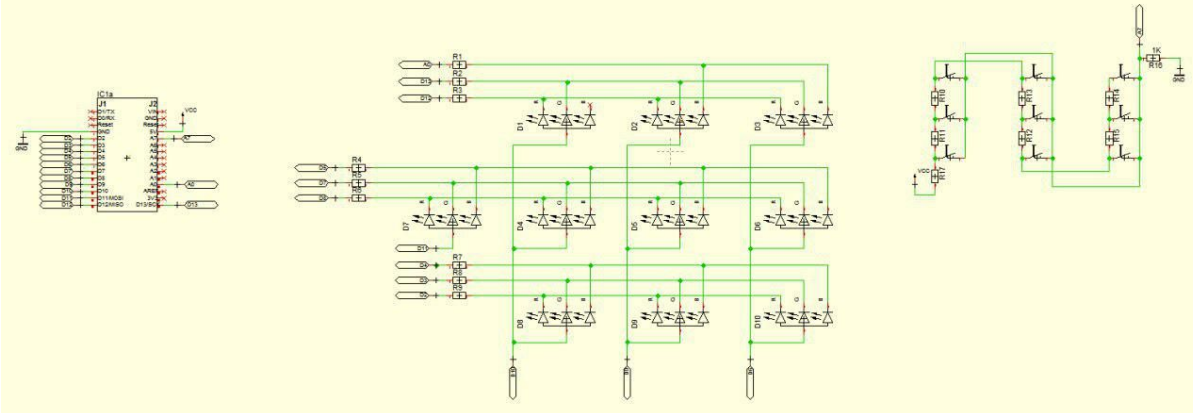
Die Variable i ist in diesem Fall die Zählervariable. Dahinter folgt di Bedingung, die für das Durchlaufen der Schleife essentiell ist. Zum Schluss folgt die Anweisung *i++*, welche eine verkürzte Schreibweise für den Befehl *i = i +1;* ist. Nach jedem Durchlauf der Schleife wird i um 1 erhöht. Warum heißt die Variable überhaupt i ? - Naja, es hat sich im Laufe der Zeit

eingebürgert, diesen Namen für die Variable zu verwenden. Man sollte natürlich immer Variablennamen verwenden, die die Funktion der Variable erklären. Wenn die Variable aber nicht besonders wichtig ist und einfach nur einen zähler darstellen soll, dann kann man sie auch ruhig “i” nennen.

Zum Schluss noch die heiß ersehnte Anweisung, die in den vorherigen Beispielen gefehlt hat: die Warte-Anweisung. Wenn ich den arduino zum Warten bringen möchte, kann ich das mit einem *delay* machen. delay(5) hält das Programm für 5 Millisekunden an. In diesem Zeitraum wartet der Arduino für 5 Millisekunden und macht nichts anderes. Um eine Sekunde zu warten, lautet der Befehl delay(1000), weil 1000 Millisekunden einer Sekunde entsprechen.

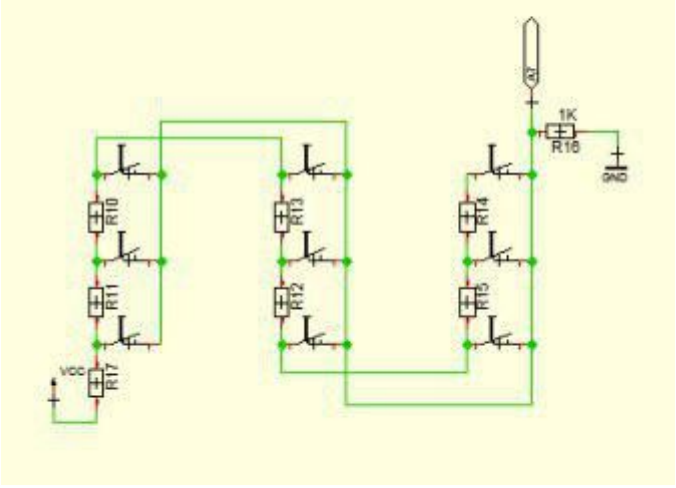
Aufbau der Platine

Die Platine besteht aus 2 Komponenten: Der LED-Matrix und den Tastern. Beide sind mit den GPIO Pins des Arduino verbunden. Die komplette Schaltung der Platine sieht folgendermaßen aus:



Spannungsteiler

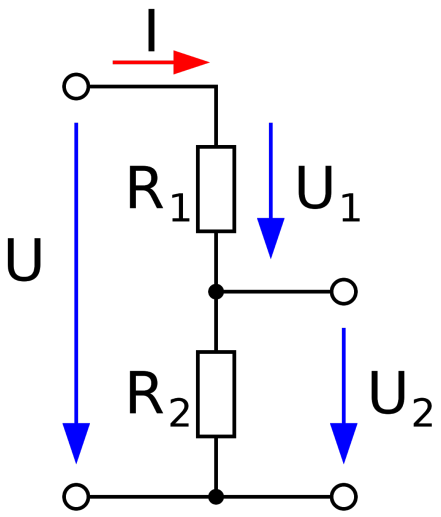
Auf der Platine wird der gedrückte Taster mit einem AD-Wandler eingelesen. Dies liegt daran, dass die Taster mit den Widerständen einen Spannungsteiler bilden. Die Schaltung der Taster sieht folgendermaßen aus:



Zu erkennen sind 9 Taster. Vor jedem Taster befindet sich ein 200Ω Widerstand. Alle Taster sind mit dem AD-Wandler verbunden, der dann noch über einen 1kΩ Widerstand mit dem 0V-Pin verbunden ist. Wenn kein Taster gedrückt ist, dann ist der AD-Wandler nur mit dem 0V-Pin verbunden und eine Zahl von 0 wird eingelesen. Wenn aber ein Taster gedrückt ist, dann zieht der

Spannungsteiler die anliegende Spannung an dem Pin nach oben. Für jeden taster wird ein eigener Wert eingelesen.

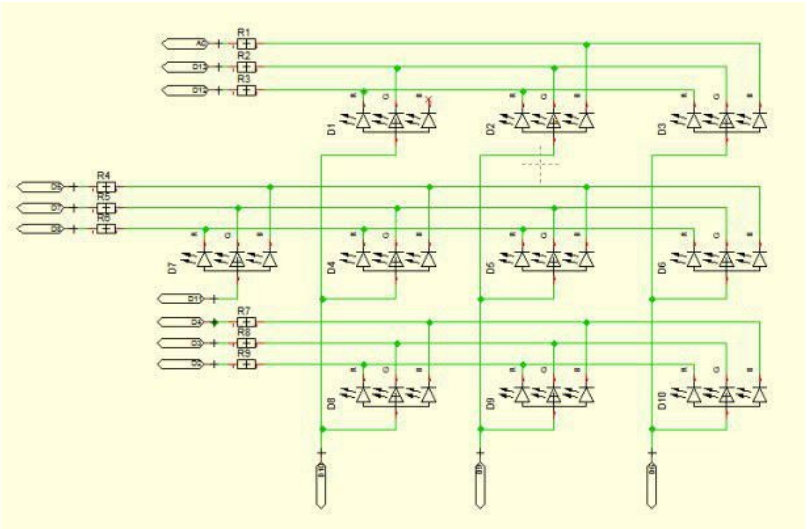
Wie funktioniert ein Spannungsteiler? Eigentlich hat ein Widerstand nur eine Auswirkung auf die Stromstärke. Erhöht man den Widerstand in einem elektrischen stromkreis, so wird die Stromstärke heruntergesetzt. Der Widerstand begrenzt die Stromstärke, die bei einer bestimmten Spannung durch den stromkreis fließen darf. Man kann Widerstände aber auch wie in dem nebenstehenden Bild anordnen. Dann ändert sich die Spannung, die zwischen den beiden Widerständen liegt, je nachdem, welche Widerstände man verwendet. Die Spannung lässt sich aus den verwendeten Widerständen berechnen. Dies ist aber nicht nötig, da die Werte für den AD-Wandler ganz simpel mit dem Arduino ausgelesen werden können. Wichtig ist aber dass die Spannung immer von dem Verhältnis der beiden Widerstände (R1 zu R2) abhängt.



https://upload.wikimedia.org/wikipedia/commons/thumb/e/eb/Einfacher-unbelasteter-Spannungsteiler.svg/2000px-Einfacher-unbelasteter-Spannungsteiler.svg.png

Wenn nun die Taste 1 auf der Platine gedrückt wird, dann berechnet sich die Spannung mit dem Verhältnis 200Ω zu 1000Ω. Wenn nun der zweite Taster gedrückt wird, dann sind 200Ω Widerstände in Reihe geschalten und haben einen Gesamtwiderstand von 400Ω. Das Verhältnis liegt also bei 400Ω zu 1000Ω. Die Schaltung sorgt dafür, dass für jeden Taster ein eigener Wert beim AD-Wandler eingelesen wird.

Multiplexer



Damit man für die Ansteuerung der LEDs möglichst wenig Pins braucht, werden diese über einen Multiplexer angesteuert. Die Schaltung zur Steuerung der LEDs wird auf dem rechten Bild gezeigt. Zuallererst muss man dazusagen, dass es sich um dreifarbige LEDs handelt. Diese haben 3 Farbkanäle, für die Farben Rot, grün und

blau. Als vierten Anschluss besitzen die LEDs einen gemeinsamen Pin. Um einen der Farbkanäle zu aktivieren, muss der dazugehörige Pin auf 0V gelegt werden. Alle 3 Farbkanäle werden aktiviert, wenn sie mit dem Minus-Pol verbunden sind. Der gemeinsame Kanal muss mit dem Plus-Pol verbunden sein. Nur dann kann die LED leuchten. Falls die LED falsch herum gepolt ist oder überall dieselbe Spannung anliegt, dann leuchtet sie einfach nicht. Wenn man nun an die Leitung, die mit Pin 6 verbunden ist, eine Spannung von 5V anlegt, dann kann man durch Steuern der Pins 2, 3, 4, 5, 7, 8, 12, 13 und 14 die 3 untersten LEDs steuern. Die Kanäle, auf denen der Pin auf LOW gesetzt wurde, leuchten; die Kanäle, bei denen der Pin auf HIGH gesetzt wurde, leuchten nicht.

Nun kann man aber nicht gezielt alle LEDs gleichzeitig leuchten lassen. Wenn ich alle LEDs außer die in der Mittel in allen 3 Farben gleichzeitig leuchten lassen möchte, dann muss ich die Pins 6, 9, 10 und 11 auf HIGH setzen und alle anderen Pins auf LOW setzen. Leider kann ich damit nicht verhindern, dass die mittlere LED auch leuchten wird. Bei einem Multiplexer wird das menschliche Auge ausgetrixt. Es ist nämlich sehr träge. Wenn wir ganz viele Bilder hintereinander sehen, sieht es so aus, als ob es sich um einen Film handeln würde. Beim Multiplexing wird nur einer der Kanäle für eine ganz kurze Zeit aktiviert. im ersten Schritt setzen wir die Leitung an Pin 6 auf HIGH und die Leitungen an den Pins 9, 10 und 11 auf LOW. Damit sorgen wir dafür, dass nur die LEDs leuchten können, die mit Pin 6 verbunden sind. Danach setzen wir die Pins 2, 3, 4, 5, 7, 8, 12, 13 und 14 so, wie wir unsere LEDs haben möchten. Die LEDs bleiben für ca. eine Millisekunde in diesem Zustand. Danach werden Alle LEDs wieder ausgeschaltet. Nun werden diese Schritte mit den anderen 2 LED-Reihen und der Status-LED wiederholt. Weil der Microcontroller sehr schnell arbeitet, sieht es für das menschliche Auge so aus, als ob alle LEDs gleichzeitig leuchten würden.

Eine wichtige Sache gibt es noch zu sagen: Der Strom, der durch die Pins 6, 9 und 10 fließen würde, wäre viel zu groß für den Pin am Arduino. Wenn bei jeder LED eine Farbe an wäre, kann nach langer Betriebszeit die Steuereinheit für den Pin durchbrennen. Deshalb befinden sich an den 3 Pins transistoren. Der Vorteil ist, dass nun viel größere Ströme fließen können. Der Nachteil ist aber, dass ein Transistor das Ausgangssignal invertiert. Wenn an den Pins nun LOW anliegt, ist die zugehörige Leitung in der LED-Matrix auf HIGH und umgekehrt genau so. Die Lösung für das Problem lautet: Man muss das Ausgangssignal also auch invertieren. Wenn man die Leitung nun auf HIGH Legen möchte, muss man die Pins auf LOW setzen. Das muss bei der Programmierung beachtet werden, sonst führt das zu Problemen. Der Pin 11 besitzt keinen Transistor, weil an ihm auch keine LED angeschlossen ist.

Auslesen der gedrückten Taster

Wie vorhin erwähnt, werden die Taster durch einen AD-Wandler eingelesen. Dabei handelt es sich um den AD-Wandler an Pin A7. Ein AD-Wandler liefert für die anliegende Spannung einen Wert von 0 bis 1023 zurück. Mit dem Befehl analogRead(A7); wird der Wert des AD-Wandlers ausgelesen. Dabei muss der Wert in einer Variable gespeichert werden. Das folgende Programm soll andauernd des Wert des AD-Wandlers auslesen:

```
void loop() {
  int ADC_Wert;
  ADC_Wert = analogRead(A7);
}
```

Der Setup-Teil bleibt bei diesem Programm leer, weil er nicht verwendet werden muss. Nun bringt es uns noch nichts, wenn wir den Wert des AD-Wandlers in einer Variable haben. Wie wir wissen, liefert der AD-Wandler für jede Taste eine eigene Zahl zurück. Dazu müssen wir die Werte vom AD-Wandler der Reihe nach einlesen und am Computer ausgeben.

Der Arduino verfügt über die Fähigkeit, mit dem Computer zu kommunizieren. Im Setup-Teil des Programms muss die Serielle Verbindung zum Computer gestartet werden. Danach können wir in der Loop-Funktion die eingelesenen Werte des AD-Wandlers an den Computer senden. Das Programm dazu sieht so aus:

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

void loop() {
  Serial.println( analogRead(A7) );
}
```

Das Programm liest andauernd den Wert von A7 ein und gibt diesen an den Computer. Nun kann in der Arduino-IDE der serielle Monitor gestartet werden (Werkzeuge > Serieller Monitor), über den die Daten vom Arduino empfangen werden können. Wenn keine Taste gedrückt wird, wird die Zahl 0 angezeigt. Für jede Taste befinden sich die eingelesenen Werte in einem bestimmten Wertebereich. Dieser Wird nun in der folgenden Tabelle Zusammengefasst:

Taste	ADC_Wert
keine	0
0	360 - 370
1	560 - 570
2	635 - 645
3	390 - 400
4	505 - 515
5	725 - 735
6	420 - 430
7	460 - 470
8	845 - 860

Jetzt kommt natürlich die frage auf: Woher haben die Tasten ihr Nummer? Dabei wurde nach folgendem System durchnummeriert: Oben links befindet sich die taste nummer 0. Rechts daneben wird die Ziffer größer. In der nächsten Zeile hat der Taster ganz rechts wieder die kleinste Ziffer.

Nun muss nur noch ein Programm geschrieben werden, das die Werte des AD-Wandlers ausliest und daraus ermittelt, welche Taste gedrückt wird. Das wird mit ganz vielen If-Abfragen gemacht. An dem Programmbeispiel wird auch gezeigt, wie man testet, ob sich eine Variable innerhalb eines Zahlenbereichs befindet:

```
int Taste = 10;
int ADC_Wert = analogRead( A7 );

if ( (ADC_Wert > 360) && (ADC_Wert < 370) )
{
  Taste = 0;
}
if ( (ADC_Wert > 560) && (ADC_Wert < 570) )
{
  Taste = 1;
}
if ( (ADC_Wert > 635) && (ADC_Wert < 645) )
{
  Taste = 2;
}
if ( (ADC_Wert > 390) && (ADC_Wert < 400) )
{
  Taste = 3;
}
if ( (ADC_Wert > 500) && (ADC_Wert < 515) )
{
  Taste = 4;
}
if ( (ADC_Wert > 725) && (ADC_Wert < 735) )
{
  Taste = 5;
}
if ( (ADC_Wert > 415) && (ADC_Wert < 430) )
{
  Taste = 6;
}
if ( (ADC_Wert > 460) && (ADC_Wert < 470) )
{
  Taste = 7;
}
if ( (ADC_Wert > 845) && (ADC_Wert < 860) )
{
  Taste = 8;
}

Serial.println( Taste );
```

Damit ist das Programm fertig, mit dem die gedrückte Taste ausgelesen und an den Seriellen Monitor geschickt werden kann.

Blinkprogramm auf der Status-LED

Konfiguration der GPIO Pins

Bevor die Pins auf der LED-Matrix angesteuert werden können, müssen sie als Ausgang konfiguriert werden. Dies muss nur einmal zu Beginn des Programms gemacht werden und kann deshalb auch im Setup-Teil stehen. Alle GPIO-Pins von 2 bis 14 müssen also als Ausgang gesetzt werden. Am komfortabelsten lässt sich das mit einer for-Schleife realisieren:

```
void setup() {
  for (int i = 2; i < 15; i++)
  {
    pinMode(i, OUTPUT);
  }
}
```

Dieser Programmcode-Abschnitt wird vermutlich in anderen Programmen noch verwendet werden und kann deshalb also eine eigene Funktion ausgelagert werden. Damit ist gemeint, dass man den Programmcode-Abschnitt in eine eigene Funktionseinheit auslagert. Dann wird im Setup-Teil nur noch diese Funktionseinheit aufgerufen, die an einer anderen Stelle im Programmcode genauer beschrieben wird. Bei der Ausführung wird im Programm ein Sprung zu der Funktionseinheit vollführt. Danach wird der darin stehende Programmcode abgearbeitet. Wenn die Funktionseinheit fertig abgearbeitet wurde, wird wieder an die ursprüngliche Stelle zurück gesprungen. Das fertige Ergebnis sieht folgendermaßen aus:

```
void Init( void )
{
  for (int i = 2; i < 15; i++)
  {
    pinMode(i, OUTPUT);
  }
}

void setup() {
  Init();
}
```

So etwas kann ziemlich nützlich sein, denn wenn es Programmcode-Abschnitte gibt, die an unterschiedlichen Stellen mehrmals durchlaufen werden, dann schadet es nicht, dies in eine Funktion auszulagern. Das erhöht die Lesbarkeit von Programmcode, macht den Programmcode leichter wartbar (wenn mal was geändert werden muss, dann braucht man das nur an einer Stelle zu tun) und ist sehr zum Vorteil für schreibfaule Leute.

Alle LEDs ausschalten

Bevor wir eine LED blinken lassen, soll zuerst dafür gesorgt werden, dass alle anderen LEDs aus sind. Dazu wird eine eigene Funktion geschrieben, die nach ihrem Aufruf alle Pins so einstellt, dass die LEDs falsch gepolt sind und somit nicht leuchten. Dazu wird die Funktion *Init* vom vorherigen Abschnitt verwendet. Aktuell sieht der Programmcode so aus:

```
void Init( void )
{
  for (int i = 2; i < 15; i++)
  {
    pinMode(i, OUTPUT);
  }
}

void setup() {
  Init();

  // An dieser Stelle sind alle wichtigen Pins als Ausgang gesetzt.
  // Leider wissen wir nicht, ob diese auf HIGH oder auf LOW gesetzt sind.
  // Deshalb brauchen wir an dieser Stelle eine Funktion, die alle Pins so einstellt, dass alle
LEDs aus sind.
}

void loop() {
  // Hier soll die LED zum blinken gebracht werden.

}
```

Nun geht es darum, die Funktion *AllleAus* zu programmieren. In dieser Funktion werden genau dieselben Pins angesprochen, die auch in der Funktion *Init* angesprochen werden. Deshalb kann die For-Schleife fürs erste kopiert werden.

```
void AllesAus( void )
{
  for (int i = 2; i < 15; i++)
  {
     // Der Schleifenkörper wird aber ein anderer sein als der Bei der Funktion Init
  }
}
```

Jetzt muss nur noch der Schleifenkörper angepasst werden. Alle Pins außer der mit der Nummer 11 müssen auf HIGH geschaltet werden. Das erreichen wir, indem wir den Pin nur dann auf HIGH setzen, wenn die Zählvariable *i* nicht 11 ist.

```
if ( i != 11)
{
    digitalWrite(i, HIGH);
}
```

Und wie erreichen wir, dass der Pin 11 jetzt auf LOW gesetzt wird? Ganz einfach: mit einem else. Wenn die Schleife sich an den Pin 11 ran machen möchte, wird der If-Block übersprungen, weil die Bedingung nicht erfüllt wird. Setzen wir nun einen Else-Block hinten dran, dann wird dieser nur ausgeführt, wenn es an den Pin 11 geht. So sieht der fertige Programmcode aus:

```
void AllesAus( void )
{
  for (int i = 2; i < 15; i++)
  {
    // Der Pin 11 muss abeer auf HIGH gesetzt werden, weil er nicht mit einem Transistor
verbunden ist
    if( i != 11)
    {
      digitalWrite(i, HIGH);
    }
    else
    {
      digitalWrite(i, LOW);
    }
  }
}
```

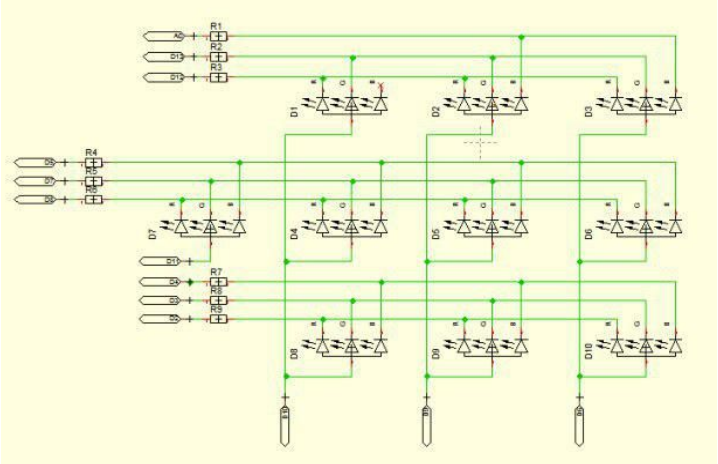
LED blinken lassen

Nun geht es darum, eine der LEDs blinken zu lassen. Dazu wird der Programmcode aus dem vorherigen Abschnitt erweitert. Als LED wird die Status-LED auf der LED-Matrix verwendet. Nach dem der Setup-Teil durchlaufen wurde, soll die LED “blinkbereit” sein. Aktuell sieht die Setup so aus:

```
void setup() {
  Init();
  AllesAus();

  // Noch sind alle LEDs in Sperrichtung geschaltet und nicht blinkbereit
}
```

Nun geht es darum, die obere LED zu aktivieren. Dazu bedarf es einen Blick in den Schaltplan:



Man erkennt, dass die gemeinsame Stromversorgung der 3 Farben von der Status-LED der Pin 11 ist. Dieser muss auf HIGH gesetzt werden, damit die LED überhaupt leuchten kann. Am besten wird das ein mal in der Setup vorgenommen, damit das in der Loop nicht mehr getan werden muss:

```
void setup() {
  Init();
  AllesAus();
  digitalWrite(11, HIGH);
}
```

Das wars dann auch schon mit dem Setup-Teil. Nun soll im Loop-Teil die LED der Reihe nach in den Farben Rot, Grün und Blau für jeweils eine Sekunde pro Farbe leuchten. Das wird folgendermaßen erreicht: Zu Beginn sind alle 3 Farben aus, da die 3 Pins zur Farbsteuerung auf HIGH gesetzt sind. Um eine Farbe zu aktivieren, muss der Pin für den farbkanal auf LOW gesetzt werden. DIe Konfigurationen für die 3 Farben lauten:

```
// Rot
digitalWrite(5, HIGH);
digitalWrite(7, HIGH);
digitalWrite(8, LOW);

// Grün
digitalWrite(5, HIGH);
digitalWrite(7, LOW);
digitalWrite(8, HIGH);

// Blau
digitalWrite(5, LOW);
digitalWrite(7, HIGH);
digitalWrite(8, HIGH);
```

In der Loop werden die 3 Zustände Rot, Grün und blau nun nacheinander aufgerufen. Mit delay(1000) kann eine Wartepause von einer Sekunde bis zum Erreichen des nächsten Zustandes erzeugt werden. Die Fertige Loop sieht folgendermaßen aus:

```
void loop() {

  // Rot
  digitalWrite(5, HIGH);
  digitalWrite(7, HIGH);
  digitalWrite(8, LOW);

  delay(1000);

  // GrÃ¼n
  digitalWrite(5, HIGH);
  digitalWrite(7, LOW);
  digitalWrite(8, HIGH);

  delay(1000);

  // Blau
```

```
  digitalWrite(5, LOW);
  digitalWrite(7, HIGH);
  digitalWrite(8, HIGH);

  delay(1000);
}
```

Was ist Multiplexing? - Erhöhung der Blinkfrequenz, um das Auge zu täuschen

Wenn nun die Wartezeit für das Delay vom vorherigen Abschnitt immer weiter heruntergesetzt wird, sieht es plötzlich so aus, als ob alle LEDs gleichzeitig leuchten würden. Dies liegt daran, dass das menschliche Auge träge ist und dieser Effekt wird zur Ansteuerung der LED-Matrix ausgenutzt.
Die LED-Matrix wird folgendermaßen angesteuert: Zuerst werden alle LEDs mit der bereits programmierten Funktion *AllesAus* ausgeschalten. Im nächsten Schritt werden die Farbkanäle für die unterste Zeile so gesetzt, wie man sie haben möchte. Danach wird die gemeinsame Stromversorgung der untersten Zeile aktiviert, eine Millisekunde gewartet und danach wieder deaktiviert. Das ganze wird nun für die anderen 2 Zeiler wiederholt. Die LEDs werden so schnell an- und ausgeschalten, dass es für das träge menschliche Auge aussieht, als wären alle LEDs gleichzeitig an.

Programmieren des Tic-Tac-Toe Spiels

Nun kommen wir endlich zu dem Wichtigen Teil: Dem Programmieren des Tic-tac-Toe Spiels. Dazu werden die 2 bereits Programmierten Funktionen aus der vorherigen Abschnitten verwendet:

```
// Alle Pins, die mit den LEDs verbunden sind, auf Ausgang setzen
void Init( void )
{
  for (int i = 2; i < 15; i++)
  {
    pinMode(i, OUTPUT);
  }
}

// Um alle LEDs aus zu schalten, müssen die damit verbundenen Pins auf LOW gesetzt
werden.
void AllesAus( void )
{
  for (int i = 2; i < 15; i++)
  {
    // Der Pin 11 muss aber auf HIGH gesetzt werden, weil er nicht mit einem Transistor
verbunden ist
    if( i != 11)
    {
      digitalWrite(i, HIGH);
    }
```

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

Nun brauche wir noch 3 globale Variablen. Diese Variablen werden ganz am Anfang des Programms erstellt und können in jeder Funktion verwendet werden. In der erste Zeile des Programms steht:

```
int Spielfeld[9];
int Spieler;
int Gewinner;
```

In der Variable Spieler wird gespeichert, welcher Spieler gerade an der Reihe ist. Die Variable Gewinner ist standardmäßig 0. Falls es ein Unentschieden gibt, wird ihr Wert zu -1. Falle Spieler 1 gewonnen hat, wird ihr Wert zu 1 und bei Spieler 2 wird ihr Wert zu 2.
Die Variable Spielfeld besitzt eckige Klammern und ist ein sogenanntes Array. Dies kann man sich wie 9 einzelne Variablen vorstellen, die alle denselben Namen tragen. Man spricht die einzelne Variable mit ihrem sogenannten Index an, welcher der Zahl in der Klammer entspricht. Dabei hat das erste Element den Index 0, das zweite Element den Index 1. das dritte Element den Index 2 und das neunte Element den Index 8. Die einzelnen Elemente werden einer LED auf dem Spielfeld zugeordnet. Dabei erfolgt die Zuordnung genau analog zur Zuordnung der Taster Nummern zu den Tastern. Die LED Oben Links entspricht *Spielfeld[0]*.

Nun geht es darum, den Setup-Teil zu erstellen. Dieser besteht aus 3 Funktiosaufrufen:

```
void setup() {
  Init();
  AllesAus();
  InitVariablen();
}
```

Dabei werden die 2 bekannten Funktionen *Init* und *AllesAus* aufgerufen. Es kommt noch eine dritte Funktion dazu: *InitVariablen*. Diese Soll den 3 globalen Variablen einen Startwert zu Beginn eines Spieles geben. Dabei muss die Funktion nur alle Elemente des Arrays auf 0 setzen. Dies kann man am Besten mit einer For-Schleife erledigen. Hinzu kommt, dass die Variable Gewinner auch auf 0 gesetzt werden muss. Zu guter letzt muss einer der Spieler 1 oder 2 beginnen. In diesem Fall beginnt Spieler 1:

```
void InitVariablen( void )
{
  // Alle Elemente des Arrays Spielfeld werden mit 0 belegt
  for (int i = 0; i < 9; i++)
  {
    Spielfeld[i] = 0;
  }

  // Spieler 1 darf mit dem Spiel beginnen
  Spieler = 1;
```

```
  // Noch hat niemand gewonnen. Deshalb wird diese Variable mit 0 belegt
  Gewinner = 0;
}
```

Schon ist der Setup-Teil fertig. Bevor wir mit dem Loop-Teil anfangen, muss noch etwas erledigt werden: aus dem Programm, in dem die Taster ausgelesen wurden, soll eine eigene Funktion zum Auslesen der Taster erstellt werden. Diese sieht so aus:

```
int TasteGedrueckt( void )
{
  // 10 bedeutet, dass keine Taste gedrückt wird
  int Taste = 10;

  int ADC_Wert = analogRead( A7 );

  // Im Folgenen werden die Werte des AD-Wandlers verglichen. daraus wird dann ermittelt,
welche taste gedrückt wird

  if ( (ADC_Wert > 360) && (ADC_Wert < 370) )
  {
    Taste = 0;
  }
  if ( (ADC_Wert > 560) && (ADC_Wert < 570) )
  {
    Taste = 1;
  }
  if ( (ADC_Wert > 635) && (ADC_Wert < 645) )
  {
    Taste = 2;
  }
  if ( (ADC_Wert > 390) && (ADC_Wert < 400) )
  {
    Taste = 3;
  }
  if ( (ADC_Wert > 500) && (ADC_Wert < 515) )
  {
    Taste = 4;
  }
  if ( (ADC_Wert > 725) && (ADC_Wert < 735) )
  {
    Taste = 5;
  }
  if ( (ADC_Wert > 415) && (ADC_Wert < 430) )
  {
    Taste = 6;
  }
  if ( (ADC_Wert > 460) && (ADC_Wert < 470) )
  {
    Taste = 7;
  }
  if ( (ADC_Wert > 845) && (ADC_Wert < 860) )
```

```
  {
    Taste = 8;
  }

  //Serial.println( Taste );

  // Der ermittelte wert für die Taste wird anschließend zurückgegeben
  return Taste;
}
```

Doch was ist das? Vor dem Funktionsnamen steht ein *int* anstelle von einem *void* und innerhalb der Funktion steht *return Taste*! Was soll das heißen? Diese Funktion ist etwas anders als die anderen Funktionen: Sie gibt einen Wert zurück. Ein Funktionsaufruf sieht deshalb so aus:

```
int Taste = TasteGedrueckt();
```

Die Funktion liefert einen Wert, der in einer Variable gespeichert werden kann. In der Variable steht dann, welche Taste gedrückt wurde.
In der Loop soll direkt zu Beginn geprüft werden, ob eine Taste gedrückt wurde und dementsprechend der Tastendruck ausgewertet werden. Noch sieht die Loop nur so aus:

```
void loop() {
  // ermittle, ob eine Tastee gedrückt wurde
  int Taste = TasteGedrueckt();
}
```

Wie kann nun mit der Variable geprüft werden, ob überhaupt eine Taste gedrückt wurde? Ganz einfach: wenn keine Taste gedrückt wurde, ist die Variable mit dem Wert 10 belegt.

```
 if (Taste != 10)
 {

 }
```

Nun muss eine passende Reaktion auf den Tastendruck getätigt werden. Die Taster und die LEDs wurden nicht ohne Grund genau gleich durchnummeriert. Wenn also eine Taste gedrückt wird, muss aus dem Spielfeld-Array das Element mit demselben Index neu beschrieben werden. Als neuer Wert wird der aktuelle Spieler verwendet, welcher in der Variable Spieler steht.

```
if (Taste != 10)
 {
   Spielfeld[ Taste ] = Spieler;
 }
```

Man darf ein Feld aus dem Spielfeld-Array nur beschreiben, wenn es vorher leer war. Ansonsten wäre es ja unfair für den Gegner, wenn man plötzlich die von ihm belegten Felder mit seiner eigenen Farbe belegen könnte. Deshalb muss geprüft werden, ob dieses Feld vor dem Beschreiben den Wert 0 beinhaltet.

```
// Falls eine Taste gedrückt wurde und dieses Feld noch unbelegt ist, markiere es mit
dem aktuellen Spieler
 if ((Taste != 10) && (Spielfeld[ Taste ] == 0))
 {
   Spielfeld[ Taste ] = Spieler;
```

```
  Spieler++;
 }
```

Eine Sache wird noch nebenbei erledigt: es wird zum nächsten Spieler gewechselt. Das wird mit *Spieler*++ realisiert. Wie schon einmal erwähnt, ist das eine Kurzschreibweise für *Spieler* = *Spieler* + *1*. Falls Spieler 1 nun an der Reihe war, ist Spieler 2 dran. Falls Spieler 2 an der Reihe war, enthält die Variable danach den Wert 3. Moment, es gibt aber nur Spieler 1 und 2. Wir müssen nach der Erhöhung der Spielervariable den Wert für den aktuellen Spieler also korrigieren:

```
 // Nach jedem Spielzug ist der neue Spieler an der Reihe. Dafür wird der Spieler zuerst um 1 erhöht und danach auf einen Überlauf geprüft
 if (Spieler > 2)
 {
  Spieler = 1;
 }
```

Kümmern wir uns nun um die LED-Ausgabe. Zuerst machen wir uns an die Status-LED ran, die anzeigt, welcher Spieler gerade an der Reihe ist. Falls es Spieler 1 ist, soll nur der rote Farbkanal aktiviert werden; bei Spieler 2 der Grüne. Das lässt sich mit ein paar bedingten Anweisungen realisieren:

```
if(Spieler == 0)
{
 // Alle Aus
 digitalWrite(5, HIGH);
 digitalWrite(7, HIGH);
 digitalWrite(8, HIGH);
}
if(Spieler == 1)
{
 // Rot
 digitalWrite(5, HIGH);
 digitalWrite(7, HIGH);
 digitalWrite(8, LOW);
}
if(Spieler == 2)
{
 // Gruen
 digitalWrite(5, HIGH);
 digitalWrite(7, LOW);
 digitalWrite(8, HIGH);
}
```

Das kann noch weiter vereinfacht werden. In diesem Fall wird immer eine Variable mit verschiedenen Konstanten verglichen. Dafür gibt es sogenannte Switches. Mit einem Switch legt man zu Beginn die Variable fest, die man prüfen möchte und definiert danach die einzelnen Fällt, die man bearbeiten möchte. Die fertige Funktion zum Anzeigen der Status-LED sieht folgendermaßen aus:

```
void AusgabeStatusLED( void )
{
 // Je nachdem, welcher Spieler gerade dran ist, leuchtet die LED in einer bestimmten Farbe
 switch (Spieler)
 {
```

```
    case 0: // Allle Farben aus
       digitalWrite(5, HIGH);
       digitalWrite(7, HIGH);
       digitalWrite(8, HIGH);
       break;
    case 1: // Rot
       digitalWrite(5, HIGH);
       digitalWrite(7, HIGH);
       digitalWrite(8, LOW);
       break;
    case 2: // Gruen
       digitalWrite(5, HIGH);
       digitalWrite(7, LOW);
       digitalWrite(8, HIGH);
       break;
    case 3: // Blau
       digitalWrite(5, LOW);
       digitalWrite(7, HIGH);
       digitalWrite(8, HIGH);
       break;
  }

  // Setze die gemeinsame Kathode auf HIGH und schalte die LED somit an
  digitalWrite(11, HIGH);

  // Warte eine Millisekunde
  delay(1);

  // Schalte die LED aus
  digitalWrite(11, LOW);

  AllesAus();
 }
```

Der Vollständigkeit halber wurde noch der Fall mit einprogrammiert, dass die LED bleu leuchtet, wenn der Spieler 3 an der Reihe ist. Diesen gibt es im Spiel aber nicht. Dieser Fall wird hier verwendet, um zu zeigen, wie man die blaue Farbe auf der LED aktivieren kann wie man später die Farbe Blau im Spiel verwenden kann.
Was passiert nun nach dem switch? Die gemeinsame Stromversorgung wird aktiviert und nach einer Millisekunde wieder deaktiviert. Dadurch leuchtet die blaue LED ganz kurz. für das träge menschliche Auge sieht es aber so aus, als ob die LED konstant leuchten würde. Zum Schluss müssen alle LED-Farbkanäle wieder auf “aus” geschalten werden.

Gehen wir nun einen Schritt weiter und machen das Ganze einmal für die komplette LED-Matrix. Zuerst möchten wir nur die oberste Reihe ausgeben. Der zugehörige Programmcode sieht wiefolge aus:

```
switch ( Spielfeld[ 0 ] )
  {
    case 0: // alle Farben aus
```

```
    digitalWrite(2, HIGH);
    digitalWrite(3, HIGH);
    digitalWrite(4, HIGH);
    break;
  case 1: // Rot
    digitalWrite(2, LOW);
    digitalWrite(3, HIGH);
    digitalWrite(4, HIGH);
    break;
  case 2: // Gruen
    digitalWrite(2, HIGH);
    digitalWrite(3, LOW);
    digitalWrite(4, HIGH);
    break;
  case 3: // Blau
    digitalWrite(2, HIGH);
    digitalWrite(3, HIGH);
    digitalWrite(4, LOW);
    break;
}
switch ( Spielfeld[ 1 ] )
{
  case 0: // Allle Farben aus
    digitalWrite(5, HIGH);
    digitalWrite(7, HIGH);
    digitalWrite(8, HIGH);
    break;
case 1: // Rot
    digitalWrite(5, HIGH);
    digitalWrite(7, HIGH);
    digitalWrite(8, LOW);
    break;
case 2: // Gruen
    digitalWrite(5, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(8, HIGH);
    break;
case 3: // Blau
   digitalWrite(5, LOW);
   digitalWrite(7, HIGH);
   digitalWrite(8, HIGH);
   break;
}
switch ( Spielfeld[ 2  ] )
{
  case 0: // alle Farben aus
```

```
      digitalWrite(12, HIGH);
      digitalWrite(13, HIGH);
      digitalWrite(14, HIGH);
      break;
    case 1: // Rot
      digitalWrite(12, LOW);
      digitalWrite(13, HIGH);
      digitalWrite(14, HIGH);
      break;
    case 2: // Gruen
      digitalWrite(12, HIGH);
      digitalWrite(13, HIGH);
      digitalWrite(14, LOW);
      break;
    case 4: // Blau
      digitalWrite(12, HIGH);
      digitalWrite(13, LOW);
      digitalWrite(14, HIGH);
      break;
  }
```

Das ist ein relativ großer Block an Programmcode, den wir nur ungerne für alle 3 Zeilen einmal in unserem Programmcode hätten. Deshalb möchten wir mit diesem Block alle 3 Zeilen auf einmal ausgeben und nutzen einen entscheidenden Vorteil aus: Die Farbkanäle für die LEDs 0, 3 und 6 sind die selben. Am Besten Packen wir den oberen Block in eine Schleife, die alle 3 Zeilen durch geht. In jedem Durchlauf wird für jeden Switch die vorherige LED mit der Ziffer 3 erhöht ausgewertet:

```
for(int i = 0; i < 3; i++)
 {
  // Prüfe die LEDs 0, 3 und 6
  switch ( Spielfeld[ 3*i ] )
  {
    case 0: // alle Farben aus
      digitalWrite(2, HIGH);
      digitalWrite(3, HIGH);
      digitalWrite(4, HIGH);
      break;
    case 1: // Rot
      digitalWrite(2, LOW);
      digitalWrite(3, HIGH);
      digitalWrite(4, HIGH);
      break;
    case 2: // Gruen
      digitalWrite(2, HIGH);
      digitalWrite(3, LOW);
      digitalWrite(4, HIGH);
      break;
    case 3: // Blau
      digitalWrite(2, HIGH);
```

```
    digitalWrite(3, HIGH);
    digitalWrite(4, LOW);
    break;
}

// Pruefe die LEDs 1, 4 und 7
switch ( Spielfeld[ 3*i + 1 ] )
{
  case 0: // Allle Farben aus
    digitalWrite(5, HIGH);
    digitalWrite(7, HIGH);
    digitalWrite(8, HIGH);
    break;
case 1: // Rot
    digitalWrite(5, HIGH);
    digitalWrite(7, HIGH);
    digitalWrite(8, LOW);
    break;
case 2: // Gruen
    digitalWrite(5, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(8, HIGH);
    break;
case 3: // Blau
   digitalWrite(5, LOW);
   digitalWrite(7, HIGH);
   digitalWrite(8, HIGH);
   break;
}

// Pruefe die LEDs 2, 5 und 8
switch ( Spielfeld[ 3*i + 2 ] )
{
  case 0: // alle Farben aus
    digitalWrite(12, HIGH);
    digitalWrite(13, HIGH);
    digitalWrite(14, HIGH);
    break;
  case 1: // Rot
    digitalWrite(12, LOW);
    digitalWrite(13, HIGH);
    digitalWrite(14, HIGH);
    break;
  case 2: // Gruen
    digitalWrite(12, HIGH);
    digitalWrite(13, HIGH);
    digitalWrite(14, LOW);
    break;
  case 4: // Blau
    digitalWrite(12, HIGH);
    digitalWrite(13, LOW);
```

```
        digitalWrite(14, HIGH);
        break;
    }
```

Im ersten Durchlauf von der Schleife werden die LEDs 0, 1 und 2 geprüft, im 2. Durchlauf die LED 3, 4 und 5 und im dritten Durchlauf die LEDs 6, 7 und 8. Nun muss für jeden Durchlauf noch die dazugehörige Stromversorgung aktiviert werden. Wie schon erwähnt, ist zu beachten, dass der Pin auf LOW gesetzt werden muss, damit der Transistor die Leitung auf HIGH setzt. Dies geschieht nun folgendermaßen:

```
void AusgabeLEDs()
{
  // Das wird für alle 3 Zeilen wiederholen
  for(int i = 0; i < 3; i++)
  {
    // Prüfe die LEDs 0, 3 und 6
    switch ( Spielfeld[ 3*i ] )
    {
      case 0: // alle Farben aus
        digitalWrite(2, HIGH);
        digitalWrite(3, HIGH);
        digitalWrite(4, HIGH);
        break;
      case 1: // Rot
        digitalWrite(2, LOW);
        digitalWrite(3, HIGH);
        digitalWrite(4, HIGH);
        break;
      case 2: // Gruen
        digitalWrite(2, HIGH);
        digitalWrite(3, LOW);
        digitalWrite(4, HIGH);
        break;
      case 3: // Blau
        digitalWrite(2, HIGH);
        digitalWrite(3, HIGH);
        digitalWrite(4, LOW);
        break;
    }

    // Pruefe die LEDs 1, 4 und 7
    switch ( Spielfeld[ 3*i + 1 ] )
    {
      case 0: // Allle Farben aus
        digitalWrite(5, HIGH);
        digitalWrite(7, HIGH);
        digitalWrite(8, HIGH);
        break;
    case 1: // Rot
        digitalWrite(5, HIGH);
        digitalWrite(7, HIGH);
        digitalWrite(8, LOW);
        break;
```

```
case 2: // Gruen
    digitalWrite(5, HIGH);
    digitalWrite(7, LOW);
    digitalWrite(8, HIGH);
    break;
case 3: // Blau
   digitalWrite(5, LOW);
   digitalWrite(7, HIGH);
   digitalWrite(8, HIGH);
   break;
}

// Pruefe die LEDs 2, 5 und 8
switch ( Spielfeld[ 3*i + 2 ] )
{
  case 0: // alle Farben aus
    digitalWrite(12, HIGH);
    digitalWrite(13, HIGH);
    digitalWrite(14, HIGH);
    break;
  case 1: // Rot
    digitalWrite(12, LOW);
    digitalWrite(13, HIGH);
    digitalWrite(14, HIGH);
    break;
  case 2: // Gruen
    digitalWrite(12, HIGH);
    digitalWrite(13, HIGH);
    digitalWrite(14, LOW);
    break;
  case 4: // Blau
    digitalWrite(12, HIGH);
    digitalWrite(13, LOW);
    digitalWrite(14, HIGH);
    break;
}

// Schalte die dazugehörige gemeinsame Anode an.
// Die Zustände an den Kathoden werden durch den Transistor invertiert
switch(i)
{
  case 0: digitalWrite(10, LOW); break;
  case 1: digitalWrite(9, LOW); break;
  case 2: digitalWrite(6, LOW); break;
}

// warte eine Millisekunde
delay(1);

// Schalte die LED Matrix wieder aus
AllesAus();
```

```
  }
 }
```

Die folgenden 2 Zeilen müssen noch in die Loop hinzugefügt werden:

```
  // Die Status-LED in der richtigen Zeile anzeigen lassen
 AusgabeStatusLED();

  // Der aktuelle Spielstand wird auf der LED Matrix ausgegeben
  AusgabeLEDs();
```

Nun ist die Funktion fertig, um alle LEDs anzusteuern. Kommen wir nun zu dem letzten Block in dem Workshop: Das Spiel soll einen Gewinner erkennen und eine Animation für den Gewinner anzeigen. Um den Gewinner zu ermitteln, werden die Ziffern in den Zeilen und Spalten, bzw. den Diagonalen multipliziert. Wenn das Ergebnis 1 ist, hat Spieler 1 gewonnen, bei 2 hat Spieler 2 gewonnen. In der Funktion wird jede Zeile, bzw. Spalte mit einer Schleife geprüft:

```
void PruefeGewinner( void )
{
  int Produkt = 1;

  // Pruefe zuerst, ob es ein Unentschieden gibt
  for(int i = 0; i < 9; i++)
  {
    Produkt *= Spielfeld[i];
  }
  if(Produkt != 0)
  {
    Gewinner = -1;
  }

  // Pruefe alle 3 Zeilen und Spalten
  for(int i = 0; i < 3; i++)
  {
    Produkt = Spielfeld[0+i]*Spielfeld[3+i]*Spielfeld[6+i];
    if(Produkt == 1)
    {
      Gewinner = 1;
    }
    if(Produkt == 8)
    {
      Gewinner = 2;
    }

    Produkt = Spielfeld[0+i*3]*Spielfeld[1+3*i]*Spielfeld[2+3*i];
    if(Produkt == 1)
    {
      Gewinner = 1;
    }
    if(Produkt == 8)
    {
      Gewinner = 2;
    }
```

```
  }

  // Pruefe die 2 Diagonalen
  Produkt = Spielfeld[0] * Spielfeld[4] * Spielfeld[8];
  if(Produkt == 1)
  {
    Gewinner = 1;
  }
  if(Produkt == 8)
  {
    Gewinner = 2;
  }

  Produkt = Spielfeld[2] * Spielfeld[4] * Spielfeld[6];
  if(Produkt == 1)
  {
    Gewinner = 1;
  }
  if(Produkt == 8)
  {
    Gewinner = 2;
  }
 }
```

Falls es einen Gewinner gibt, soll die Funktion *SpielGewonnen* aufgerufen und falls das Spiel unentschieden ist, wird die Funktion *SpielUnentschieden* aufgerufen. In der Loop steht also:

```
 // Es wird geprüft, ob es schon einen Gewinner gibt
  PruefeGewinner();

  //Serial.println(Gewinner);

  // Falle es einen Gewinner gibt, wird die dazugehörige Animation aktiviert
  if(Gewinner > 0)
  {
    SpielGewonnen();
  }
  // Falls das Spiel unentscheiden steht, wird die dazugrhörige Animation aktiviert
  if(Gewinner == -1)
  {
    SpielUnentschieden();
  }
```

Kommen wir nun zu den 2 Funktionen für die Animation: Zuerst wird für 2 Sekunden nur der letzte Spielstand angezeigt:

```
 bool neuesSpiel = false;

  // Zuerst bleibt die Anzeig für 2 Sekunden auf dem aktuellen Spielstand
  unsigned long Counter = millis();
  while((millis() - 2000) < Counter)
```

```
  {
    AusgabeLEDs();
  }

  // Danach wird noch zwischengespeichert, welcher Spieler als nächstes an der
Reihe wäre
  // Das ist notwendig, weil spÃ¤ter dieser Wert verloren geht
  int naechsterSpieler = Gewinner + 1;
  if (naechsterSpieler > 2)
  {
    naechsterSpieler = 1;
  }
```

DIe Variable vom Typ Bool wird später verwendet. Eine Schleife wird eine Animation ausgeben, bis eine Taste gedrückt wird. Ob eine Taste gedrückt wurde, wird in dieser Variable gespeichert. Danach muss nur noch gespeichert werden, welcher Spieler als nächstes an der Reihe wäre, weil dieser das nächste Spiel beginnen wird. Die fertige Funktion sieht folgendermaßen aus:

```
void SpielGewonnen( void )
{
  bool neuesSpiel = false;

  // Zuerst bleibt die Anzeig fÃ¼r 2 Sekunden auf dem aktuellen Spielstand
  unsigned long Counter = millis();
  while((millis() - 2000) < Counter)
  {
    AusgabeLEDs();
  }

  // Danach wird noch zwischengespeichert, welcher Spieler als nächstes an der Reihe wäre
  // Das ist notwendig, weil später dieser Wert verloren geht
  int naechsterSpieler = Gewinner + 1;
  if (naechsterSpieler > 2)
  {
    naechsterSpieler = 1;
  }

  // Solange keine Taste gedrückt wird, wird nun diese Animation angezeigt
  do
  {
    // Teil 1
    for(int i = 0; i < 9; i++)
    {
      Spielfeld[i] = Gewinner;
    }
    Spielfeld[4] = 0;

    Counter = millis();
    while((millis() - 1000) < Counter)
```

```
    {
      AusgabeLEDs();

      if(TasteGedrueckt() != 10)
      {
        neuesSpiel = true;
      }
    }

    // Teil 2
    for(int i = 0; i < 9; i++)
    {
      Spielfeld[i] = 0;
    }
    Spielfeld[4] = Gewinner;

    Counter = millis();
    while((millis() - 1000) < Counter)
    {
      AusgabeLEDs();

      if(TasteGedrueckt() != 10)
      {
        neuesSpiel = true;
      }
    }
  }while(!neuesSpiel);

  // eine Taste wurde gedrückt. Das Spiel kann von neuem beginnen
  InitVariablen();
  Spieler = naechsterSpieler;
}
```

Die Funnktion lässte im ersten Schritt alle LEDs außer die mittlere in den Farben des Gewinners leuchten und im 2. Schritt nur die mittlere. Die Funktion für ein Unentschieden funktioniert fast identisch, nur dass sie die äußeren LEDs abwechselnd rot und grün leuchten lässt:

```
// Falls das Spiel mit unentschieden endet, zeige eine dazugehörige Animation
void SpielUnentschieden( void )
{
  bool neuesSpiel = false;

  // Zuerst bleibt die Anzeig für 2 Sekunden auf dem aktuellen Spielstand
  unsigned long Counter = millis();
  while((millis() - 2000) < Counter)
  {
    AusgabeLEDs();
  }

  // Danach wird noch zwischengespeichert, welcher Spieler als nächstes an der Reihe wÃ¤re
  // Das ist notwendig, weil später dieser Wert verloren geht
  int naechsterSpieler = Spieler;
```

```
// Solange keine Taste gedrückt wird, wird nun diese Animation angezeigt
do
{
  // Teil 1
  Spielfeld[0]=1;
  Spielfeld[2]=1;
  Spielfeld[6]=1;
  Spielfeld[8]=1;
  Spielfeld[4]=0;
  Spielfeld[1]=2;
  Spielfeld[3]=2;
  Spielfeld[5]=2;
  Spielfeld[7]=2;

  Counter = millis();
  while((millis() - 1000) < Counter)
  {
    AusgabeLEDs();

    if(TasteGedrueckt() != 10)
    {
      neuesSpiel = true;
    }
  }

  // Teil 2
  Spielfeld[0]=2;
  Spielfeld[2]=2;
  Spielfeld[6]=2;
  Spielfeld[8]=2;
  Spielfeld[4]=0;
  Spielfeld[1]=1;
  Spielfeld[3]=1;
  Spielfeld[5]=1;
  Spielfeld[7]=1;

  Counter = millis();
  while((millis() - 1000) < Counter)
  {
    AusgabeLEDs();

    if(TasteGedrueckt() != 10)
    {
      neuesSpiel = true;
    }
  }
}while(!neuesSpiel);

InitVariablen();
```

```
  Spieler = naechsterSpieler;
}
```