

Arduino Programmier-Handbuch

Inhalt

.....	1
Einleitung	2
1. Struktur	3
2. Datentypen	8
3. Arithmetik	10
4. Konstanten	11
5. Ablaufsteuerung	12
6. Digitaler Input - Output	15
7. Analoger Input - Output	16
8. Zeit	17
9. Mathematisch	18
10. Zufallsfunktionen	18
11. Serielle Kommunikation	19
12. Anhang	20

Ref.:
<http://www.freeduino.de/de/books/arduino-programmier-handbuch>
Bearbeitet von DK2JK (2009, 2014)

Einleitung

[Arduino](#)

Arduino ist eine sogenannte "physical computing" Plattform, die auf einem einfachen Microcontrollerboard basiert und eine Entwicklungsumgebung der C-Programmiersprache darstellt. Mit dem Arduino können interaktive autarke Objekte entwickelt werden oder das Microcontrollerboard kann mit auf dem Computer laufender Software verbunden werden. Derzeitig verfügbare Versionen des Arduino können vorgefertigt bezogen werden. Es stehen auch sämtliche Hardware Design Informationen zur Verfügung, mit dessen Hilfe der Arduino selbst hergestellt werden kann.

Das Arduino Projekt wurde mit einer ehrenhaften Erwähnung in der Kategorie "Digitale Communities" beim Prix Ars Electronica 2006 ausgezeichnet.

Implementierungen

Hardware

Ein Arduino Board besteht aus einem Atmel AVR Microcontroller (z.B. ATmega328) , einem Quarz und einigen zusätzlichen Komponenten. Das Board kann allein vom USB versorgt werden. Der Microcontroller ist vorprogrammiert mit einem [Bootloader](#), so dass ein externes Programmiergerät nicht notwendig ist.

Alle Boards werden über eine serielle Verbindung programmiert; die Hardware hierfür unterscheidet sich je nach Board. Aktuelle Boards werden über die USB-Schnittstelle programmiert; ein entsprechender Chip übernimmt die USB-nach-Seriell- Umwandlung. Einige Varianten wie der Arduino Mini benötigen zur Kommunikation mit dem Computer einen separaten USB-Seriell-Adapter.

Das Arduino Board stellt die meisten der Input/Output Pins des Mikrocontrollers auf Buchsenleisten zur Verfügung. Das Board verfügt über 14 digitale I/O Pins, 6 davon können analoge Pulsweitenmodulationssignale generieren und es gibt 6 analoge Eingänge. Alle Pins sind auf der Oberseite des Boards angebracht und mit Buchsenleisten im 0.1 Zoll Raster versehen. Es gibt zahlreiche kommerzielle "[Shields](#)". Das sind Steckkarten für verschiedene Anwendungen, die auf die Anschlusspins des Arduino montiert werden können.

Software

Die Arduino IDE ist eine Java Applikation, die alle notwendigen Tools zur Verfügung stellt wie Editor, Compiler, Programmer etc.

Die Entwicklungsumgebung basiert auf der Entwicklungsumgebung „Processing“, die entwickelt wurde, um Leuten , die normalerweise nicht viel Kontakt mit Programmierung haben, das Programmieren näher zu bringen. Die Programmiersprache ist C mit wesentlichen C++ -Erweiterungen.

Hardware Version

Arduino Uno verwendet einen Atmega328.

Viele zum Arduino kompatible Produkte vermeiden den Namen 'Arduino' durch eine Abwandlung mit der Endung 'duino'. Dies ist auf eine Restriktion der Nutzung der Marke 'Arduino' zurückzuführen.

Open Hardware und Open Source

Das Arduino Hardware Referenz Designs wird unter einer Creative Commons Attribution Share-Alike 2.5 Lizenz veröffentlicht und kann auf der Arduino Website heruntergeladen werden. Dateien für Layout und Produktion für einige Versionen der Arduino Hardware sind ebenfalls verfügbar. Der Quellcode für die IDE und die on-board Libraries sind ebenso verfügbar und stehen unter der GPLv2 Lizenz.

1. Struktur

Der grundlegende Aufbau der Arduino Programmiersprache ist relativ einfach und teilt sich in mindestens zwei Teile auf. Diese zwei benötigten Teile oder Funktionen umschließen Blöcke von Anweisungen.

```
void setup()  
{  
  anweisungen;  
}  
  
void loop()  
{  
  anweisungen;  
}
```

Hierbei ist setup() die Vorbereitung und loop() ist die Ausführung. Beide Funktionen sind notwendig, damit das Programm ausgeführt werden kann.

Noch vor der Setup Funktion sollte die Definition von Variablen erfolgen. Setup wird als erste Funktion in einem Programm nur einmal durchlaufen. Es dient z.B. zum Setzen von PinMode oder der Initiierung der seriellen Kommunikation.

Nach der setup() Funktion folgt die loop() Funktion. Sie beinhaltet den Programmcode, der in einer unendlichen Schleife ausgeführt wird - Eingänge auslesen, Ausgänge triggern, etc. Diese Funktion ist der Kern von allen Arduino Programmen und erledigt die Hauptarbeit.

setup()

Die setup() Funktion wird einmalig aufgerufen, wenn das Programm startet. Benutze diese Funktion um PinModes zu setzen oder die serielle Kommunikation zu starten. Die setup() Funktion muss in jedem Programm vorkommen, auch wenn sie keine Anweisungen enthält.

```
void setup()
{
  pinMode(pin, OUTPUT);      // 'pin' als Ausgang definieren
}
```

loop()

Nach dem Durchlaufen der setup() Funktion macht die loop() Funktion genau das was ihr Name vermuten lässt und läuft in einer endlosen Schleife.

```
void loop()
{
  digitalWrite(pin, HIGH);   // schaltet 'pin' ein

  delay(1000);               // Pause für eine Sekunde
  digitalWrite(pin, LOW);   // schaltet 'pin' aus
  delay(1000);               // Pause für eine Sekunde
}
```

Funktionen

Eine Funktion ist ein Block von Programmcode, der einen Namen hat und eine Reihe von Anweisungen, die beim Aufruf der Funktion ausgeführt werden. Die Funktionen void setup() und void loop() wurden bereits erklärt. Es gibt noch weitere eingebaute Funktionen, die später behandelt werden.

Eigene Funktionen zu schreiben macht Sinn, um sich wiederholende Aufgaben zu vereinfachen und um die Übersichtlichkeit der Programmstruktur zu fördern. Funktionen werden erstellt, indem zuerst der Type der Funktion definiert wird. Dieser ist identisch mit dem Datentyp des zurückgegebenen Wertes, so wie zum Beispiel 'int' für den Integer Typen. Wenn kein Wert zurückgegeben werden soll, so wird der Funktionstyp 'void' verwendet. Nach der Definition des Types wird der Name festgelegt und in Klammern alle Parameter, die der Funktion übergeben werden sollen.

```
Typ FunktionsName(parameter)
{
  anweisungen;
}
```

Die folgende Integer Typ Funktion "delayVal()" wird genutzt, um einen Delay Wert, also eine Verzögerung in ein Programm durch Auslesen eines Potentiometers einzubauen. Zuerst wird eine lokale Variable 'v' erstellt. Als nächstes wird 'v' mit der Stellung des Potentiometers gleichgesetzt, das einen Wert zwischen 0 und 1023 haben kann. Der Wert wird dann durch 4 dividiert, um auf einen Bereich von 0 bis 255 zu kommen und am Ende wird das Ergebnis der Funktion zum Hauptprogramm zurückgegeben.

```
int delayVal()
{
  int v;           // erstelle temporäre Variable 'v'
  v = analogRead(pot); // lese Potentiometer Wert
  v /= 4;         // Umrechnen von 0-1023 auf 0-255
  return v;      // errechneten Wert zurückgeben
}
```

{ } geschweifte Klammern

Geschweifte Klammern definieren den Anfang und das Ende von Funktions- und Anweisungsblöcken so wie bei der 'void loop()' Funktion und auch bei der 'for' und 'if' Anweisung.

```
typ funktion()
{
  anweisungen;
}
```

Eine öffnende geschwungene Klammer '{' muss immer von einer schließenden geschwungenen Klammer gefolgt werden '}'. Hier spricht man oft davon, dass die Anzahl der Klammern ausgeglichen sein müssen („matched“). Unausgeglichene Klammern führen oft zu kryptischen, undurchschaubaren Fehlern im Compiler, die manchmal schwer zu finden sind, vor allem in großen Programmen.

Die Arduino Programmierumgebung hilft dabei, die Ausgeglichenheit der geschwungenen Klammern zu prüfen. Dafür muss man einfach eine Klammer auswählen oder kurz hinter dieser klicken, und das logisch verbundene Gegenstück wird optisch hervorgehoben.

; Semikolon

Ein Semikolon muss am Ende einer Anweisung verwendet werden und dient zur Trennung der Elemente eines Programmes.

Ein Semikolon wird auch verwendet, um die Elemente einer "for" Schleife zu trennen.

```
int x = 13; // deklariert Variable 'x' als Integer mit Wert 13
```

Bemerkung: Das Vergessen eines Semikolons am Ende einer Zeile führt zu einem Fehler im Compiler. Die Fehlerbeschreibung kann dabei sehr eindeutig sein und auf das fehlende Semikolon direkt hinweisen; das muss aber nicht der Fall sein. Wenn eine undurchschaubarer oder scheinbar unlogischer Fehler gemeldet wird, sollte als erstes nach fehlenden Semikolons in der Nähe des gemeldeten Fehlers gesucht werden.

/*... */ Block Kommentare

Block Kommentare oder mehrzeilige Kommentare sind Textbereiche, die vom Programm ignoriert werden. Sie werden für längere Beschreibungen oder Kommentare verwendet und helfen anderen Autoren Programmteile zu verstehen. Sie fangen mit /* an und enden mit */ und können mehrere Zeilen umfassen.

```
/* Dies ist eine eingefügter Block Kommentar  
   bitte den schliessenden Kommentar nicht vergessen -  
   Diese müssen ausgeglichen sein  
*/
```

Weil Kommentare vom Programm ignoriert werden und damit keinen Speicherplatz verbrauchen, sollten sie großzügig verwendet werden. Mit Block Kommentaren kann man auch ganze Programmteile zum Testen ungültig machen.

Bemerkung: Man kann einzeilige Kommentare in einen Block Kommentar integrieren. Aber es ist nicht möglich, einen zweiten Block Kommentar zu einzuschließen.

// Einzeilige Kommentare

Einfache einzeilige Kommentare werden mit einem // am Anfang der Zeile definiert und enden mit dem Ende der Zeile. Sie werden vom Programm ignoriert und verbrauchen keinen Speicherplatz.

```
// dies ist ein einzeiliger Kommentar
```

Einzeilige Kommentare werden oftmals nach Anweisungen verwendet, um mehr Informationen über die verwendete Anweisung zu vermitteln oder um Notizen für zukünftige Änderungen am Code festzuhalten.

Variablen

Eine Variable ist die Benennung eines numerischen Wertes mit einem Namen und belegt Speicherplatz für die spätere Verwendung in einem Programm. Wie der Name schon vermuten lässt, kann der Wert der Variablen kontinuierlich verändert werden, im Gegensatz zu einer Konstanten, deren Wert im Programmablauf unveränderbar bleibt. Eine Variable muss deklariert werden und kann optional mit einem Wert zugewiesen werden. Das folgende Beispiel deklariert eine Variable 'inputVariable' und ordnet ihr dann den Wert vom analogen Pin 2 zu:

```
int inputVariable = 0;           // deklariert eine Variable und  
                                // setzt ihren Wert auf 0  
inputVariable = analogRead(2); // setzt den Wert der Variable gleich  
                                // mit dem Wert von Analog Pin 2
```

'inputVariable' ist die Variable selber. Die erste Zeile erklärt, dass ihr Datentyp 'int' ist; das ist der Kurzausdruck für Integer. Die zweite Zeile gibt der Variablen den Wert des analogen Pin 2. Damit wird der Wert des Pins überall im Code verfügbar.

Wenn der Wert einer Variablen zugeordnet oder verändert wird kann man seine Wert testen, um zu sehen, ob er bestimmte Bedingungen erfüllt. Ein Beispiel, wie man mit Variablen sinnvoll arbeiten kann, zeigt der folgende Code. Hier wird getestet, ob 'inputVariable' kleiner als 100 ist. Ist dies der Fall, so wird der 'inputVariable' der Wert 100 zugeordnet und das Programm verwendet diesen Wert als Pause (delay). Der minimale Wert von 'inputVariable' ist somit in jedem Fall 100.

```
if (inputVariable < 100) // prüft, ob Variable weniger als 100
{
  inputVariable = 100; // wenn wahr, ordne Wert von 100 zu
}
delay(inputVariable); // benutzt Variable als Verzögerung
```

Bemerkung: Variablen sollten immer möglichst deutlich beschreibende Namen erhalten, um den Code besser lesbar zu machen.

Variablen Namen wie 'tiltSensor' oder 'pushButton' helfen dem Programmierer und dem Leser besser zu verstehen, was die Variable bewirkt. Namen wie 'var' oder 'value' sagen wenig aus und wurden hier nur als Beispiel verwendet. Als Namen kann alles verwendet werden, das nicht bereits ein Schlüsselwort in der Arduino Sprache ist.

Variablen Deklaration

Alle Variablen müssen vor der Verwendung deklariert werden. Einer Variablen wird als Erstes ein Typ zugeordnet (wie z.B. int, long, float), dann wird ein Name vergeben und optional einen Anfangswert zugewiesen. Dies muss nur einmal im Programm vorgenommen werden. Danach kann der Wert zu jedem Zeitpunkt durch Berechnungen oder verschiedenste Zuweisungen geändert werden.

Das folgende Beispiel deklariert 'inputVariable' als 'int', also Integer Datentyp und setzt den Anfangswert auf 0. Dies nennt man eine 'einfache Zuordnung'.

```
int inputVariable = 0;
```

Eine Variable kann an vielen Orten im Programm deklariert werden. Der Ort der Deklaration bestimmt, welche Programmteile Zugriff auf die Variable haben.

Variablen Geltungsbereich

Eine Variable kann am Anfang des Programmes vor 'void setup()' deklariert werden, lokal innerhalb von Funktionen und manchmal auch innerhalb eines Anweisungsblocks, wie zum Beispiel einer Schleife. Wo die Variable deklariert wird, bestimmt ihren Geltungsbereich, oder die Fähigkeit anderer Programmteile, auf den Wert der Variablen zuzugreifen.

Eine globale Variable kann von jeder Funktion und Anweisung des Programmes gesehen und benutzt werden. Diese Variable wird zu Beginn des Programmes deklariert, noch vor der setup() Funktion. Wenn eine solche Variable keine Voreinstellung hat, ist automatisch der Wert Null eingestellt.

Eine lokale Variable wird innerhalb einer Funktion oder Schleife definiert. Sie ist nur sichtbar und nutzbar innerhalb der Funktion, in der sie deklariert wurde. Deswegen ist es möglich, dass Variablen mit dem selben Namen in verschiedenen Teilen des Programms unterschiedliche Werte enthalten. Durch die Sicherstellung, dass nur die Funktion selber Zugriff auf seine eigenen Variablen hat, wird das Programm vereinfacht und man reduziert das Risiko von Fehlern.

Das folgende Beispiel zeigt anschaulich, wie man Variablen auf verschiedene Weisen deklarieren kann und zeigt ihre Geltungsbereiche.

```
int value;           // 'value' ist sichtbar
                    // für jede Funktion

void setup()
{
  // kein Setup erforderlich
}

void loop()
{
  for (int i=0; i<20;) // 'i' ist nur sichtbar
  {                   // innerhalb der for-Schleife
    i++;
  }
  float f;           // 'f' ist nur sichtbar
                    // innerhalb der for-Schleife
}
```

2. Datentypen

byte

Byte speichert einen 8-bit numerischen, ganzzahligen Wert ohne Dezimalkomma. Der Wert kann zwischen 0 und 255 sein.

Der Datentyp 'byte' ist identisch mit „unsigned char“.

```
byte someVariable = 180; // deklariert 'someVariable'
                        // als einen 'byte' Datentyp
```

int

Integer sind der verbreitetste Datentyp für die Speicherung von ganzzahligen Werten ohne Dezimalkomma. Sein Wert hat 16 Bit und reicht von -32.768 bis +32.767.

```
int someVariable = 1500; // deklariert 'someVariable'
                        // als einen 'integer' Datentyp
```

Bemerkung: Integer Variablen werden bei Überschreiten der Limits 'überrollen'. Zum Beispiel wenn $x = 32767$ und eine Anweisung addiert 1 zu x , $x = x + 1$ oder $x++$, wird 'x' dabei 'überrollen' und den Wert -32,768 annehmen.

long

Datentyp für ganze Zahlen mit erweiterer Größe, ohne Dezimalkomma, gespeichert in einem 32-bit Wert in einem Spektrum von -2,147,483,648 bis +2,147,483,647.

```
long someVariable = 90000; // deklariert 'someVariable'
                           // als einen 'long' Datentyp
```

float

Ein Datentyp für Fließkomma Werte oder Zahlen mit Nachkommastelle. Fließkomma Zahlen haben eine bessere Auflösung als Integer und werden als 32-bit Wert gespeichert; Der Zahlenbereich geht von -3.4028235E+38 bis 3.4028235E+38.

```
float someVariable = 3.14; // deklariert 'someVariable'  
                        // als einen 'float' Datentyp
```

Bemerkung: Flieskommazahlen sind nicht immer genau und führen manchmal zu merkwürdigen Resultaten, wenn sie verglichen werden. Ein Vergleich wie z.B.

```
if (someVariable == 1.0)...
```

sollte vermieden werden, da der vermeintliche Wert auch intern als 1.0000001 dargestellt werden kann.

Berechnungen mit 'float' sind sehr viel langsamer als mit Integer Datentypen. Dieser Datentyp sollte deshalb nur selten verwendet werden.

arrays

Ein Array ist eine Sammlung von Werten, auf die mit einem Index zugegriffen wird. Jeder Wert in dem Array kann aufgerufen werden, indem man den Namen des Arrays mit dem Index abfragt. Der Index fängt bei einem Array immer bei 0 an. Ein Array muss deklariert werden und optional mit Werten belegt, werden bevor es genutzt werden kann.

```
int myArray[] = {wert0, wert1, wert2...}
```

Hier geht der Index von 0 bis Anzahl der Elemente -1 .

Genau so ist es möglich ein Array zuerst mit Datentyp und Größe zu deklarieren und später einer Index Position einen Wert zu geben.

```
int myArray[5]; // deklariert Datentyp 'integer' als Array mit 6  
               // Positionen  
myArray[3] = 10; // gibt dem 4. Index den Wert 10
```

Um den Wert eines Arrays auszulesen kann man diesen einfach einer Variablen mit Angabe des Arrays und der Index Position zuordnen.

```
x = myArray[3]; // x hat nun den Wert 10
```

Arrays werden oft für Schleifen verwendet, bei dem der Zähler der Schleife auch als Index Position für die Werte im Array verwendet wird. Das folgende Beispiel nutzt ein Array um eine LED zum flickern zu bringen. Mit einer for-Schleife und einem bei 0 anfangenden Zähler wird eine Indexposition im Array ausgelesen, an den LED Pin gesendet, eine 200ms Pause eingelegt und dann dasselbe mit der nächsten Indexposition durchgeführt.

```
int ledPin = 10; // LED auf Pin 10  
byte flicker[] = {180, 30, 255, 200, 10, 90, 150, 60};  
                // Array mit 8 verschiedenen Werten  
void setup()  
{  
  pinMode(ledPin, OUTPUT); // Setzt den OUTPUT Pin  
}  
void loop()  
{  
  for(int i=0; i<7; i++) // Schleife gleicht der Anzahl  
  { // der Werte im Array  
    analogWrite(ledPin, flicker[i]); // schreibt den Indexwert auf die LED  
    delay(200); // 200ms Pause  
  }  
}
```

3. Arithmetik

Arithmetische Operatoren umfassen Addition, Subtraktion, Multiplikation und Division. Sie geben die Summe, Differenz, das Produkt oder den Quotienten zweier Operatoren zurück.

```
y = y + 3;
x = x - 7;
i = j * 6;
r = r / 5;
```

Die Operation wird unter Beibehaltung der Datentypen durchgeführt. 9 / 4 wird so zum Beispiel zu 2 und nicht 2,25, da 9 und 4 Ganzzahlen sind und keine Nachkommastellen unterstützen. Dies bedeutet auch, dass der Zahlenbereich überlaufen kann, wenn das Resultat größer ist, als der Datentyp zulässt.

Wenn die Operanden unterschiedliche Datentypen haben, wird der größere Typ verwendet. Hat zum Beispiel eine der Nummern (Operanden) den Datentyp 'float' und der andere 'int', so wird Fließkomma Mathematik zur Berechnung verwendet.

Bemerkung: Wähle für Variablen Datentypen, die groß genug sind, um die Werte zu speichern. Sei Dir bewusst, an welcher Stelle die Werte überlaufen und betrachte auch, was in der Gegenrichtung passieren könnte. z.B. bei (0 - 1) oder (0 - - 32768). Für Berechnungen, die Brüche ergeben, sollten immer 'float' Variablen genutzt werden. Allerdings mit dem Bewusstsein der Nachteile: Großer Speicherbedarf und langsame Geschwindigkeit der Berechnungen.

Nutze den Cast-Operator z.B. (int)myFloat um eine Flot-Variable in eine Ganzzahl zu verwandeln. Zum Beispiel wird `i = (int)3.6` die Variable `i` auf den Wert 3 setzen.

Gemischte Zuweisungen

Gemischte Zuweisungen kombinieren eine arithmetische Operation mit einer Variablen Zuweisung. Diese findet man üblicherweise in Schleifen, die wir später noch genau beschreiben werden. Die gängigsten gemischten Zuweisungen umfassen:

```
x ++; // identisch mit x = x + 1, oder Erhöhung von x um +1
x --; // identisch mit x = x - 1, oder Verminderung von x um -1
x += y; // identisch mit x = x + y, oder Erhöhung von x um +y
x -= y; // identisch mit x = x - y, oder Verminderung von x um -y
x *= y; // identisch mit x = x * y, oder Multiplikation von x mit y
x /= y; // identisch mit x = x / y, oder Division von x mit y
```

Bemerkung: Zum Beispiel führt `x *= 3` zur Verdreifachung von 'x' und weist der Variablen 'x' das Ergebnis der Kalkulation zu.

Vergleichende Operatoren

Der Vergleich von Variablen oder Konstanten gegeneinander wird oft in If-Anweisungen durchgeführt, um bestimmter Bedingungen auf Wahrheit zu testen. In den Beispielen auf den folgenden Seiten wird `??` verwendet, um eine der folgenden Bedingungen anzuzeigen:

```
x == y    // x ist gleich y
x != y    // x ist ungleich y
x < y    // x ist kleiner als y
x > y    // x ist größer als y
x <= y   // x ist kleiner oder gleich y
x >= y   // x ist größer oder gleich y
```

Logische Operatoren

Logische Operatoren sind normalerweise eine Methode, um zwei Ausdrücke zu vergleichen und ein TRUE oder FALSE je nach Operator zurückliefern. Es gibt drei logische Operatoren AND, OR und NOT, die oft in If-Anweisungen verwendet werden:

```
Logisch AND:
if (x > 0 && x < 5)    // nur WAHR wenn beide
                        // Ausdrücke WAHR sind
```

```
Logisch OR:
if (x > 0 || y > 0)    // WAHR wenn einer der
                        // Ausdrücke WAHR ist
```

```
Logisch NOT:
if (!x > 0)            // nur WAHR wenn der
                        // Ausdruck FALSCH ist
```

4. Konstanten

Die Arduino Programmiersprache hat ein paar vordefinierte Werte, die man auch Konstanten nennt. Sie machen den Programmcode einfacher lesbar. Konstanten werden in Gruppen unterteilt.

true/false

Diese sind Boolean Konstanten, die logische Pegel definieren. 'false' ist als 0 (Null) definiert, während 'true' als 1 definiert ist. Bei Abfragen wird alles ungleich Null als 'true' gewertet.

```
if (b == true );
{
  machEtwas;
}
```

HIGH/LOW

Diese Konstanten definieren Pin Level als HIGH oder LOW und werden beim Lesen oder Schreiben auf digital Pins verwendet. High ist als Logiclevel 1, ON, true oder 5 Volt definiert, während LOW als Logiclevel 0, OFF, false oder 0 Volt definiert ist.

```
digitalWrite(13, HIGH);
```

INPUT/OUTPUT

Konstanten, die in der `pinMode()` Funktion benutzt werden und einen digitalen Pin entweder als INPUT oder OUTPUT definieren.

```
pinMode(13, OUTPUT);
```

5. Ablaufsteuerung

if

Einfache Verzweigung.

Die If-Abfrage testet, ob eine bestimmte Bedingung wahr ist oder nicht. Bei einem analogen Wert kann dies das Erreichen eines bestimmten Levels sein. Ist dies der Fall, so werden Anweisungen innerhalb der geschweiften Klammer ausgeführt. Ist diese Bedingung nicht erfüllt, werden die Anweisungen innerhalb der Klammer übersprungen. Beispiel für eine If-Abfrage:

```
if (someVariable >= value)
{
    doSomething;
}
```

Das oben vorgestellte Beispiel vergleicht 'someVariable' mit dem Wert 'value', der entweder eine Variable oder Konstante sein kann. Wenn der Vergleich oder die Bedingung wahr ist, werden die Anweisungen innerhalb der Klammern ausgeführt, in diesem Beispiel 'doSomething'. Wenn nicht, wird der Teil übersprungen und es geht nach den Klammern weiter in der Ausführung des Programmcodes.

Bemerkung: Vorsicht vor dem versehentlichen Benutzen von '=', wie in `if(x=10)`. Hier wird hier die Variable x mit dem Wert 10 belegt, was immer wahr ist. Anstatt dessen nutze '==', wie in `if(x==10)`. Hierbei wird nur geprüft, ob x den Wert 10 hat. Am einfachsten merkt man sich das mit '=' als 'gleich' im Gegensatz zu '==' als 'ist gleich mit'. Bei Vergleich mit Konstanten merkt der Compiler die fehlerhafte Syntax, wenn man die Konstante als erstes schreibt:

```
if ( 0 = x ) // Fehler
{}

if ( 0 == x) // ok
{}

```

if... else

'if... else' erlaubt eine Entweder-Oder-Entscheidung. Zum Beispiel möchten Sie einen digitalen Eingang prüfen und im Falle von 'HIGH' andere Anweisungen ausführen als im Falle von 'LOW'. Dies kann man so im Programmcode abbilden:

```
if (inputPin == HIGH)
{
    doThingA;
}
```

```
}  
else  
{  
  doThingB;  
}
```

'else' kann auch noch weitere 'if' Abfragen enthalten, so dass mehrfache Test zur selben Zeit stattfinden können. Es ist sogar möglich, eine weitere Anzahl von diesen 'else'-Zweigen zu nutzen:

```
if (inputPin < 500)  
{  
  doThingA;  
}  
else if (inputPin >= 1000)  
{  
  doThingB;  
}  
else  
{  
  doThingC;  
}
```

Bemerkung: Eine If-Abfrage testet, ob eine Bedingung innerhalb der Klammer wahr oder falsch ist. Diese Bedingung kann jedes gültige C Statement sein, wie in unserem ersten Beispiel `if (inputPin == HIGH)`. In diesem Beispiel prüft die if Abfrage nur, ob der definierte Eingang den Logic level HIGH hat (+5 Volt).

for

Zählschleife.

Die 'for' Schleife wird verwendet, um einen Block von Anweisungen in geschweiften Klammern eine festgelegte Anzahl von Wiederholungen durchlaufen zu lassen. Es gibt in der Klammer der 'for' Schleife drei Elemente, getrennt durch Semikolon ';':

```
for (Initialisierung; Bedingung; Ausdruck)  
{  
  doSomething;  
}
```

Die Initialisierung einer lokalen Zähl-Variablen passiert als erstes und nur einmalig. Bei jedem Durchlaufen der Schleife wird die Bedingung an der zweiten Stelle getestet. Solange die Bedingung erfüllt ist, wird der Ausdruck ausgeführt und die Anweisungen in geschweiften Klammern werden ausgeführt. Dies geschieht so lange, bis die Bedingung nicht mehr wahr ist.

Das folgende Beispiel startet mit einem Integer Wert 'i' bei 0, die Bedingung testet, ob der Wert noch kleiner als 20 ist und wenn dies wahr ist, so wird 'i' um einen Wert erhöht und die Anweisungen innerhalb der geschweiften Klammern werden ausgeführt:

```
for (int i=0; i<20; i++) // deklariert 'i', teste ob weniger
{
  digitalWrite(13, HIGH); // schaltet Pin 13 ein
  delay(250); // Pause fuer 1/4 Sekunde
  digitalWrite(13, LOW); // schaltet Pin 13 aus
  delay(250); // Pause fuer 1/4 Sekunde
}
```

Bemerkung: Die for-Schleife in der Programmiersprache C ist sehr flexibler. Jede oder alle der drei Header Elemente können weggelassen werden, jedoch sind die Semikolons notwendig. Zusätzlich können die Statements für Initialisierung, Bedingung und Ausdruck durch jedes gültige C Statement mit Variablen ohne Bezug zur Schleife ersetzt werden. Diese Methode ist unübersichtlich und sollte deshalb nicht verwendet werden.

while

„Bedingungsschleife“.

'while' Schleifen werden unbegrenzt wiederholt und laufen, bis die Bedingung innerhalb der Klammer falsch ist. Etwas muss die zu testende Variable ändern oder die Schleife endet nie. Dies kann im Code passieren, wie eine ansteigende Variable oder von externen Werten, wie einem Sensor Test erfolgen.

```
while (someVariable >= value)
{
  doSomething;
}
```

Das folgende Beispiel testet, ob 'someVariable' kleiner als 200 ist. Die Anweisungen innerhalb der Schleife werden solange ausgeführt, bis 'someVariable' nicht mehr kleiner als 200 ist (also z.B. gleich 200).

```
while (someVariable < 200) // testet ob weniger als 200
{
  doSomething; // führt Anweisungen aus
  someVariable++; // erhöht Variable um 1
}
```

do... while

Die 'do...while' Schleife ist eine endgesteuerte Schleife, die ähnlich funktioniert wie die 'while' Schleife. Der Unterschied ist, dass der Test der Bedingung am Ende der Schleife stattfindet. Somit läuft die Schleife immer mindestens einmal.

```
do
{
  doSomething;
} while (someVariable <= value);
```

Das folgende Beispiel ordnet readSensors() der Variablen 'x' zu, macht eine Pause für 50ms, um dann unbegrenzt weiter zu laufen bis 'x' nicht mehr kleiner als 100 ist.

```
do
{
  x = readSensors();    // ordnet den Wert von
                        // readSensors() 'x' zu
  delay (50);           // Pause für 50ms
} while (x < 100);      // Schleife laeuft weiter bis 'x' weniger als
                        // 100 ist
```

6. Digitaler Input - Output

pinMode(pin,mode)

Wird in 'void setup()' benutzt um einen speziellen Pin entweder als Eingang oder Ausgang zu konfigurieren.

```
pinMode(pin, OUTPUT);    // setzt 'pin' als Ausgang
```

Arduinos digitale Pins sind standardmäßig Eingänge, weshalb sie nicht extra als Eingänge mit 'pinMode()' festgelegt werden müssen. Als Eingang konfigurierte Pins sind hochohmig.

Es gibt im ATmega Chip auch komfortable 'Pullup' Widerstände (20...50kΩ), die per Software eingeschaltet werden können:

```
pinMode(pin, INPUT_PULLUP ); // setzt 'pin' als Eingang und
                               // schaltet den 'Pullup' Widerstand ein
```

Pullup Widerstände werden normalerweise verwendet, um Eingänge wie Schalter anzuschliessen.

Als Ausgang konfigurierte Pins können mit maximal 40 mAmpere Strom von angeschlossenen Elementen und Schaltkreisen belastet werden. Dies ist genug um eine LED aufleuchten zu lassen (seriellen Widerstand nicht vergessen), aber nicht genug um die meisten Relais, Magnetspulen oder Motoren zu betreiben.

Kurzschlüsse an den Arduino Pins ebenso wie zu hohe Ströme können den Output Pin oder gar den ganzen ATmega Chip zerstören. Deshalb ist es eine gute Idee, einen Ausgangspin mit externen Elementen in Serie mit einem 470Ω oder 1KΩ Widerstand zu beschalten.

digitalRead(pin)

'digitalRead(pin)' liest den Wert von einem digitalen Pin; das Resultat ist entweder HIGH oder LOW. Der Pin kann entweder als Variable oder Konstante festgelegt werden (0-13).

```
value = digitalRead(Pin); // setzt 'value' gleich mit
                           // dem Eingangspin
```

digitalWrite(pin,value)

'digitalWrite(pin,value)' gibt den Logiclevel HIGH oder LOW an einem Pin aus. Der Pin kann als Variable oder Konstante festgelegt werden (0-13).

```
digitalWrite(pin, HIGH); // setzt 'pin' auf high (an)
```

Das folgende Beispiel liest einen Taster an einem digitalen Eingang aus und schaltet eine LED ein, wenn der Taster gedrückt wird:

```
int led    = 13;    // LED an Pin 13 angeschlossen
int pin    = 7;    // Taster an Pin 7 angeschlossen
int value  = 0;    // Variable um den Auslesewert zu speichern

void setup()
{
  pinMode(led, OUTPUT);    // legt Pin 13 als Ausgang fest
  pinMode(pin, INPUT);    // legt Pin 7 als Eingang fest
}

void loop()
{
  value = digitalRead(pin); // setzt 'value' gleich mit
                          // dem Eingangspin
  digitalWrite(led, value); // setzt 'led' gleich mit dem
                          // Wert des Tasters
}
```

7. Analoger Input - Output

analogRead(pin)

Liest den Wert eines analogen Pins mit einer 10 Bit Auflösung aus. Diese Funktion ist nur für Pins (0-5) verfügbar. Die resultierenden Integer Werte liegen im Bereich von 0 bis 1023.

```
value = analogRead(pin); // setzt 'value' gleich mit dem Analogwert
                          // von 'pin'
```

Bemerkung: Analoge Pins müssen im Gegensatz zu digitalen nicht zuerst als Eingang oder Ausgang deklariert werden.

analogWrite(pin, value)

'analogWrite(pin, value)' schreibt pseudo-analoge Werte mittels einer hardwarebasierten Pulsweiten Modulation (PWM) an einen Ausgangspin. Auf den Arduino Boards mit dem ATmega 168 oder 328 Chip ist diese Funktion für die Pins 3, 5, 6, 9, 10 und 11 anwendbar. Der Wert kann als Variable oder Konstante Bereich 0-255 festgelegt werden.

```
analogWrite(pin, value); // schreibt 'value' auf den analogen 'pin'
```

Ein Wert 0 generiert eine gleichmäßige Spannung von 0 Volt an einem Pin; Ein Wert von 255 generiert eine gleichmäßige Spannung von 5 Volt an eine Pin. Für Werte zwischen 0 und 255 wechselt der Pin sehr schnell zwischen 0 und 5 Volt - je höher der Wert, desto länger ist der Pin HIGH (5 Volt). Bei einem Wert von 64 ist der Pin zu dreivierteln der Zeit auf 0 Volt und zu einem Viertel auf 5 Volt. Ein Wert von 128 führt dazu, dass die Ausgangsspannung zur Hälfte der Zeit auf HIGH steht und zur anderen Hälfte auf LOW. Bei 192 misst die Spannung am Pin zu einer Viertel der Zeit 0 Volt und zu dreivierteln die vollen 5 Volt.

Weil dies eine hardwarebasierte Funktion ist, läuft die konstante Welle unabhängig vom Programm bis zur nächsten Änderung des Zustandes per analogWrite (bzw. einem Aufruf von digitalWrite oder digitalWrite am selben Pin).

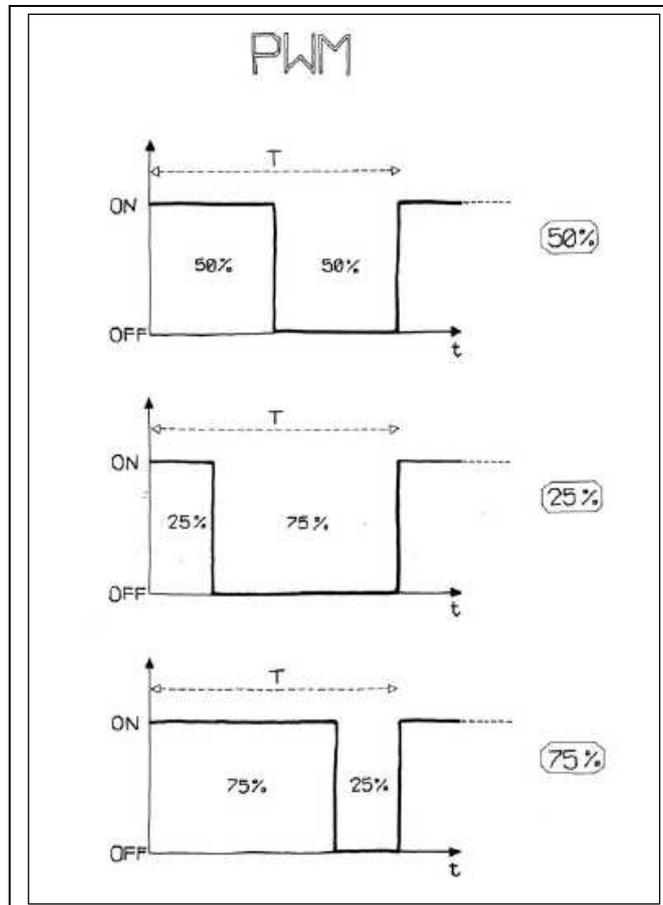
Bemerkung: Analoge Pins müssen im Gegensatz zu digitalen Pins nicht zuvor als Eingang oder Ausgang deklariert werden. Die Frequenz des PWM-Signals beträgt ungefähr 490 Hz.

Das folgenden Beispiel liest einen analogen Wert von 'pin', rechnet den Wert per Division durch 4 um und gibt ihn dann als PWM Signal an 'led' aus.

```
int led = 10;    // LED mit 220 Widerstand an Pin 10
int pin = 0;    // Potentiometer am analogen Pin 0
int value;      // Wert fuer Auslesung

void setup(){} // kein Setup benoetigt

void loop()
{
  value = analogRead(pin); // setzt 'value' gleich mit 'pin'
  value /= 4;              // wandelt 0-1023 in 0-255 um
  analogWrite(led, value); // gibt das PWM Signal an 'led' aus
}
```



8. Zeit

delay(ms)

Pausiert ein Programm für die Dauer der Zeit angegeben in Millisekunden, wobei 1000 = 1 Sekunde ist.

```
delay(1000); // wartet für eine Sekunde
```

millis()

Gibt den Wert in Millisekunden als 'long' Datentyp zurück, berechnet die Zeit, seitdem das aktuelle Programm gestartet hat.

```
value = millis(); // setzt 'value' gleich mit millis()
```

Bemerkung: Dieser Wert wird nach etwa 9 Stunden überlaufen und wieder bei Null anfangen.

9. Mathematisch

min(x, y)

Berechnet das Minimum von zwei Werten irgendeines Datentypes und gibt den kleineren Wert zurück.

```
value = min(value, 100); // setzt 'value' als kleinere  
                        // Wert oder 100 fest, damit 'value'  
                        // nie größer als 100 ist.
```

max(x, y)

Berechnet das Maximum von zwei Werten irgendeines Datentypes und gibt den höheren Wert zurück.

```
value = max(value, 100); // setzt 'value' als größeren  
                        // Wert oder 100 fest, damit 'value'  
                        // mindestens 100 ist.
```

10. Zufallsfunktionen

randomSeed(seed)

Setzt einen Wert oder 'Seed' als Ausgangspunkt für die random() Funktion.

```
randomSeed(value); // setzt 'value' als den Zufalls Seed
```

Der Arduino ist selber nicht in der Lage eine wirklich Zufallswerte zu produzieren. Mit randomSeed() kann eine Variable als 'seed' verwendet werden, um bessere Zufallsergebnisse zu erhalten. Als 'seed' Variable oder auch Funktion können so zum Beispiel millis() oder analogRead() eingesetzt werden, um elektrisches Rauschen durch den Analogpin als Ausgang für Zufallswerte zu nutzen.

random(min, max)

Die random Funktion erlaubt die Erzeugung der pseudo-zufälligen Werte innerhalb eines definierten Bereiches von minimum und maximum Werten.

```
value = random(100, 200); // setzt 'value' mit einer Zufallszahl  
                        // zwischen 100 und 200 gleich
```

Bemerkung: Benutze dieses nach der randomSeed() Funktion.

Das folgende Beispiel erzeugt einen Zufallswert zwischen 0 und 255 und gibt ihn als PWM Signal auf einem PWM Pin aus.

```
int randomNumber; // Variable um den Zufallswert zu speichern
int led = 10;     // LED mit 220 Ohm Widerstand an Pin 10

void setup() {} // kein Setup notwendig

void loop()
{
  randomSeed(millis()); // verwendet millis() als seed
  randomNumber = random(255); // Zufallsnummer im Bereich von 0-255
  analogWrite(led, randomNumber); // PWM Signal als Output
  delay(500); // halbe Sekunde Pause
}
```

11. Serielle Kommunikation

Serial.begin(rate)

'Serial.begin(rate)' öffnet den seriellen Port und setzt die Baudrate (Datenrate) für die serielle Übertragung fest. Die typische Baudrate mit dem Computer ist 9600 Baud. Andere Geschwindigkeiten werden auch unterstützt.

```
void setup()
{
  Serial.begin(9600); // oeffnet seriellen Port
} // setzt die Datenrate auf 9600 bps
```

Bemerkung: Wenn die serielle Kommunikation verwendet wird, so können die digitalen Pins 0 (RX) und 1 (TX) nicht zur selben Zeit verwendet werden.

Serial.print(data)

Schreibt Daten zum seriellen Port. Die Formatierung erfolgt passend zum Datentyp.

Serial.println(data)

Schreibt Daten zum seriellen Port, gefolgt von einem automatischen Zeilenumbruch als Carrier Return und Linefeed. Dieser Befehl hat die selbe Form wie 'Serial.print()', ist aber einfacher auf dem seriellen Monitor zu lesen.

```
Serial.println(analogValue); // sendet den Wert von
                             // 'analogValue'
```

Bemerkung: Detailliertere Informationen über die zahlreichen Varianten der Klasse 'Serial' finden Sie unter

Arduino IDE--> Hilfe--> Referenz --> Serial

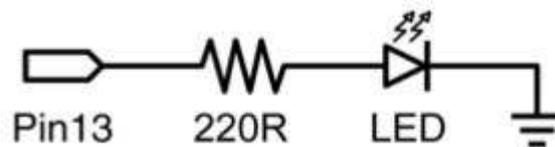
Das folgende einfache Beispiel liest einen Wert vom analogen Pin 0 aus und sendet die Daten an den Computer einmal pro Sekunde.

```
void setup()
{
  Serial.begin(9600);           // setzt die Datenrate auf 9600 bps
}

void loop()
{
  Serial.println(analogRead(0)); // sendet den Analogwert
  delay(1000);                  // pausiert fuer 1 Sekunde
}
```

12. Anhang

Digitaler Ausgang



Dies ist ein einfaches Programm und schaltet etwas ein und aus. In diesem Beispiel ist eine LED an PIN 13 angeschlossen. Die LED geht jede Sekunde an und aus.

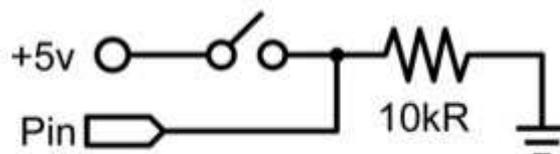
Siehe Arduino-IDE--> Beispiele--> Basics --> blink

```
int ledPin = 13;                // LED aus Digitalpin 13

void setup()                    // wird einmal druchlaufen
{
  pinMode(ledPin, OUTPUT);     // setzt Pin 13 als Ausgang
}

void loop()                     // Laufen als Endlosschleife
{
  digitalWrite(ledPin, HIGH);  // schaltet die LED ein
  delay(1000);                 // Pause fuer 1 Sekunde
  digitalWrite(ledPin, LOW);   // schaltet die LED aus
  delay(1000);                 // Pause fuer 1 Sekunde
}
```

Digitaler Eingang



Dies ist die einfachste Form eines Einganges mit nur zwei möglichen Zuständen: ein oder aus. Dieses Beispiel liest einen einfachen Schalter oder Taster an Pin 2 aus. Wenn der Schalter geschlossen ist, dann ist der Eingangspin HIGH und die blinkende LED wird aktiviert.

```

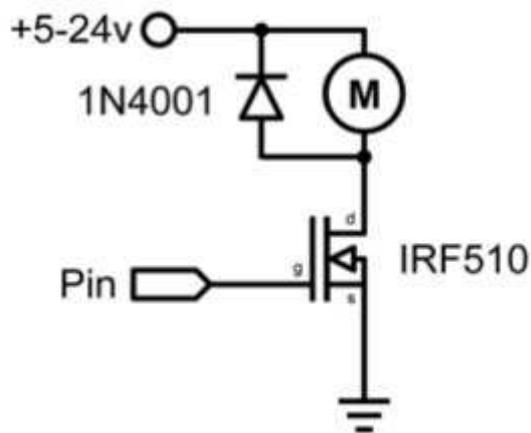
int ledPin = 13;           // Ausgangspin fuer die LED
int inPin = 2;            // Eingangspin fuer einen Schalter

void setup()
{
  pinMode(ledPin, OUTPUT); // deklariert LED als Ausgang
  pinMode(inPin, INPUT);   // deklariert Schalter als Eingang
}

void loop()
{
  if (digitalRead(inPin) == HIGH) // prüfen ob die der Eingang HIGH ist
  {
    digitalWrite(ledPin, HIGH); // schaltet die LED ein
    delay(1000);                // Pause fuer 1 Sekunde
    digitalWrite(ledPin, LOW);  // schaltet die LED aus
    delay(1000);                // Pause fuer 1 Sekunde
  }
}

```

Ausgänge mit hoher Stromstärke



Machmal ist es notwendig, mehr als die 40 Milliampere der Ausgänge des Arduino zu treiben. In diesem Fall kann ein MOSFET oder Transistor benutzt werden, um höhere Stromstärken zu schalten. Das folgende Beispiel schaltet einen MOSFET etwa 5 mal pro Sekunde ein und aus.

Bemerkung: Die Schaltung zeigt einen Motor mit Schutzdiode. Nicht-induktive Lasten können auf die Diode verzichten.

```

int outPin = 5;           // Ausgangspin fuer den MOSFET

void setup()
{
  pinMode(outPin, OUTPUT); // setzt Pin5 als Ausgang
}

void loop()
{
  for (int i=0; i<=5; i++) // Schleife wird 5 mal durchlaufen
  {
    digitalWrite(outPin, HIGH); // schaltet MOSFET an
  }
}

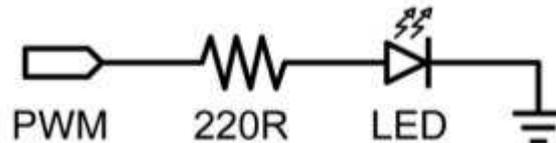
```

```

    delay(250);           // Pause 1/4 Sekunde
    digitalWrite(outPin, LOW); // schaltet MOSFET aus
    delay(250);           // Pause 1/4 Sekunde
  }
  delay(1000);           // Pause 1 Sekunde
}

```

analoger PWM Ausgang



Die Pulsweiten Modulation (PWM) ist eine Methode, um analoge Ausgänge zu simulieren, indem man die Ausgangsspannung pulsieren lässt. Damit kann man zum Beispiel eine LED heller oder dunkler werden lassen oder einen Servomotor steuern. Das folgende Beispiel lässt eine LED mit Hilfe einer Schleife langsam heller und dunkler werden.

Siehe Arduino-IDE--> Beispiele--> Analog --> fading

```

int ledPin = 9;           // PWM Pin fuer die LED

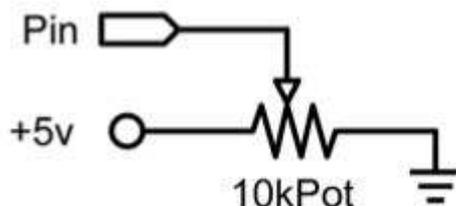
void setup(){}           // Kein Setup notwendig

void loop()
{
  for (int i=0; i<=255; i++) // aufsteigender Wert für i
  {
    analogWrite(ledPin, i); // setzt den Helligkeitswert auf i
    delay(100);             // Pause fuer 100ms
  }

  for (int i=255; i>=0; i--) // absteigender Wert fuer i
  {
    analogWrite(ledPin, i); // setzt den Helligkeitswert auf i
    delay(100);             // Pause fuer 100ms
  }
}

```

Potentiometer Eingang



Mit einem Potentiometer und einem der Analog-Digital Converter (ADC) Eingänge des Arduino ist es möglich, analoge Werte von 0-1023 zu lesen. Das folgende Beispiel verwendet ein Potentiometer, um die Blinkrate einer LED zu kontrollieren.

```

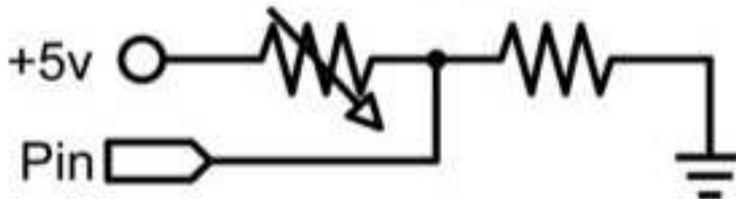
int potPin = 0;    // Eingangspin fuer das Potentiometer
int ledPin = 13;  // Ausgangspin fuer die LED

void setup()
{
  pinMode(ledPin, OUTPUT); // deklarriere ledPin als OUTPUT
}

void loop()
{
  digitalWrite(ledPin, HIGH); // schaltet ledPin ein
  delay(analogRead(potPin)); // pausiert Program um Wert des
                             // Potentiometers
  digitalWrite(ledPin, LOW); // schaltet ledPin aus
  delay(analogRead(potPin)); // pausiert Program um Wert des
                             // Potentiometers
}

```

Eingang für variable Widerstände



Variable Widerstände umfassen CdS Lichtsensoren, Thermistoren, Flex Sensoren und ähnliches.

Dieses Beispiel nutzt eine Funktion, um den Analogwert auszulesen und seiner Größe entsprechend eine Pause im Programmablauf zu definieren. Dies kontrolliert die Geschwindigkeit, nach der eine LED heller und dunkler wird.

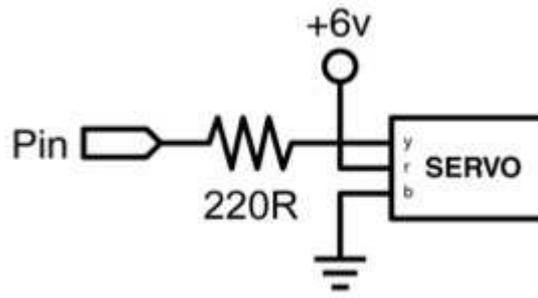
```

int ledPin    = 9;    // PWM Pin für die LED
int analogPin = 0;    // variabler Widerstand auf Analog Pin 0
void setup(){}      // kein Setup benoetigt
void loop()
{
  for (int i=0; i<=255; i++) // ansteigender Wert fuer 'i'
  {
    analogWrite(ledPin, i); // setzt die Helligkeit der LED auf Level 'i'
    delay(delayVal());     // laenge der Pause aus der Funktion delayVal()
  }
  for (int i=255; i>=0; i--) // absteigender Wert fuer 'i'
  {
    analogWrite(ledPin, i); // setzt die Helligkeit der LED auf Level 'i'
    delay(delayVal());     // laenge der Pause aus der Funktion delayVal()
  }
}

int delayVal()
{
  int v; // definert temporaere Variable
  v = analogRead(analogPin); // liest den Analogwert aus
  v /= 8; // rechnet 0-1024 auf 0-128 um
  return v; // gibt das Resultat der Funktion zurueck
}

```

Servo Output



Hobby-Servos sind eine geschlossene Motoreinheit, die sich in der Regel in einem 180-Grad-Winkel bewegen lassen. Es braucht nur einen Puls, der alle 20ms gesendet wird. In diesem Beispiel wird eine servoPulse-Funktion genutzt, um das Servo von 10-170 Grad und wieder zurück zu bewegen.

Siehe auch Arduino-IDE--> Beispiele--> Servo

```
int servoPin = 2;    // Servo mit Digital-Pin 2 verbunden
int myAngle;        // Drehwinkel des Servos ca 0-180 Grad
int pulseWidth;     // Variable der servoPulse-Funktion
void setup()
{
  pinMode(servoPin, OUTPUT); // Pin 2 als Ausgang setzen
}
void servoPulse(int servoPin, int myAngle)
{
  pulseWidth = (myAngle * 10) + 600; // bestimmt die Verzögerung
  digitalWrite(servoPin, HIGH);      // setzt den Ausgang auf HIGH
  delayMicroseconds(pulseWidth);     // Mikrosekunden Pause
  digitalWrite(servoPin, LOW);       // setzt den Ausgang auf LOW
}
void loop()
{
  // Servo startet bei 10 Grad und dreht auf 170 Grad
  for (myAngle=10; myAngle<=170; myAngle++)
  {
    servoPulse(servoPin, myAngle); // sendet Pin und Winkel
    delay(20);                     // Zyklus erneuern
  }
  // Servo startet bei 170 Grad und dreht auf 10 Grad
  for (myAngle=170; myAngle>=10; myAngle--)
  {
    servoPulse(servoPin, myAngle); // sendet Pin und Winkel
    delay(20);                     // Zyklus erneuern
  }
}
```