

Inhaltsverzeichnis

Ein Einstieg (bis Video 06)

Arduino Sketch	3
Befehlssequenzen	3
Zeitverzögerungen	3
Digitale Ein- und Ausgänge	4
Der serielle Monitor	4
Konstanten und Variablen	5
Funktionen	6
Entscheidungen (if .. else)	7
Funktionen an Funktionen übergeben	8

Analoge Eingänge (Video 07)

Eingang abfragen	9
Globale und lokale Variablen	10

Analoge Ausgänge (Video 08)

Wert ausgeben	11
---------------	----

Schleifen und Datentypen (Video 09)

For - Schleife	12
While - Schleife	12
Do .. While - Schleife	13
Break, Continue und Return	13
Zahlen und wie man damit rechnen kann	14

Arrays (Video 10)

Deklaration eines Arrays	16
Verwendung von Arrays	16
Arrays und Funktionen	17
Grösse eines Arrays ermitteln	17

Der Zufall und weitere Arrays (Video 11)

Mehrdimensionale Arrays	18
Der Zufall	18
switch .. case	19

Das richtige Timing (Video 12)

millis()	21
micros()	21
Statische Variablen	21
Universelle Blinkfunktion	22

Spielereien mit millis() (Video 13)

Datentyp bool

23

Ein Einstieg (bis Video 06)

Dieses Dokument dient nur als Ergänzung und Zusammenfassung zum Video. Es ist keine vollständige Dokumentation der Programmiersprache. Hier findest du ebenfalls eine Zusammenfassung von Video 1 bis 5.

Arduino Sketch

Ein Sketch ist im Wesentlichen ein C - Programm, das von der Arduino Entwicklungsumgebung übersetzt und auf den Arduino geladen werden kann.

Er besteht im wesentlichen aus zwei obligatorischen Funktionen:

```
void setup() {  
    // Das wird nur ein Mal aufgerufen  
}  
  
void loop() {  
    // Das wird in einer Endlosschleife immer wieder aufgerufen  
}
```

Diese beiden Funktionen bilden das minimale Arduino- Programm. In der Praxis wird es durch viele weitere Funktionen und Befehlssequenzen ergänzt.

Befehlssequenzen

Befehle werden nacheinander abgearbeitet. Sie enden jeweils mit einem Strichpunkt.

```
Serial.begin(9600);  
Serial.println("Ich gebe eine Zahl aus.");  
Serial.print("Die Zahl ist ");
```

Es können auch mehrere Befehle auf einer Zeile stehen.

```
Serial.begin(9600); Serial.println("Ich gebe eine Zahl aus.");  
Serial.print("Die Zahl ist ");
```

Zeitverzögerungen

Mit dem delay() - Befehl können wir das Programm eine bestimmte Zeit anhalten.

```
delay(500);                // Pause von 500 ms  
delayMicroseconds(500);    // Pause von 500 µs
```

ACHTUNG: während einer solchen Pause ist das Programm nicht aktiv und kann keine anderen Ereignisse verarbeiten. Eine Ausnahme sind Interrupts, das ist aber ein Thema für später.

Digitale Ein- und Ausgänge

```
pinMode(pinNummer, OUTPUT);           // Ausgang
pinMode(pinNummer, INPUT);             // Hochohmiger Eingang
pinMode(pinNummer, INPUT_PULLUP);      // Eingang mit Pullup - Widerstand

digitalWrite(pinNummer, HIGH);          // gibt 5V aus
digitalWrite(pinNummer, LOW);           // gibt 0V aus

int val = digitalRead(pinNummer);
// gibt HIGH oder LOW zurück
```

Der serielle Monitor

Der serielle Monitor wird aktiviert mit

```
Serial.begin(9600);
```

9600 ist die Geschwindigkeit in Bit pro Sekunde. Es sind noch wesentlich höhere Geschwindigkeiten möglich.

Geschrieben wird mit

```
Serial.print("Ohne neue Zeile");
Serial.println("Mit neuer Zeile");
```

Serial hat noch viel mehr Möglichkeiten. Für den Moment soll das aber genügen.

Konstanten und Variablen

Einer **Konstanten** wird zu Beginn ein Wert zugewiesen. Dieser kann später nicht mehr verändert werden.

```
const int led_Pin = 12;
```

led_Pin erhält den Wert 12 und behält diesen während der ganzen Programmausführung.

const definiert eine Konstante

int ist der Typ, das ist eine Abkürzung für Integer, also eine ganze Zahl.

Eine Variable funktioniert ähnlich. Ihr Wert kann aber später verändert werden.

```
int zahl = 25;
```

int steht hier ebenfalls für eine ganze Zahl.

Um den Wert zu verändern, kann später einfach ein neuer Wert zugewiesen werden:

```
zahl = 12;
```

int darf hier nicht mehr angegeben werden, da der Typ bereits festgelegt ist.

Eine Variable kann auch ohne Startwert deklariert werden.

```
int zahl;
```

Es können auch mehrere Variablen desselben Typs gemeinsam deklariert werden.

```
int zahl1, zahl2;
```

Auch bei einer gemeinsamen Deklaration kann ein Startwert angegeben werden. Allerdings muss für jedes Element ein eigener Startwert angegeben werden.

```
int zahl3 = 7, zahl4 = 12;
```

Weitere Datentypen werden erst später angeschaut.

Funktionen

Funktionen erlauben Codeteile, die mehrfach gebraucht werden, in einer Funktion zusammenzufassen und von mehreren Stellen im Programm aufzurufen. Zusätzlich erreicht man dadurch eine gewisse Strukturierung, die der Übersichtlichkeit dient.

setup() und **loop()** sind Beispiele solcher Funktionen.

```
void schreibeWas() {  
    Serial.println("Hallo");  
}
```

void sagt, dass die Funktion keinen Wert zurückgibt.

Der Aufruf erfolgt mit

```
schreibeWas();
```

ACHTUNG: die **()** dürfen nicht vergessen werden!

Einer Funktion können auch Werte übergeben werden. Diese Werte bezeichnet man auch als Parameter.

```
void rechne(int a, int b) {  
    Serial.println(a*b);  
}
```

Aufruf:

```
rechne(3,5);
```

Eine Funktion kann aber auch Werte zurückgeben.

```
int mult(int a, int b) {  
    return a * b;  
}
```

Aufruf:

```
Serial.println(mult(3,5));
```

schreibt 15 auf den Monitor;

Entscheidungen (if .. else)

Die Möglichkeit auf Grund einzelner Werte das Programm auf verschiedenen Wegen zu durchlaufen, ist ein wesentlicher Bestandteil eines Computers. Dazu wird eine if - Abfrage verwendet.

```
zahl = 12;
if (zahl == 12) {
    Serial.println("Das Dutzend ist voll");
}
Serial.println("Das wird immer geschrieben");
```

Nach if steht eine Bedingung in Klammern. Nur wenn diese zutrifft, wird der nachfolgende Code ausgeführt. Betroffen ist der Code, der nach der Bedingung zwischen den beiden geschweiften Klammern {} steht.

Für die Bedingung stehen verschiedene Vergleichsoperatoren zur Verfügung:

```
==  ist gleich   (WICHTIG: '==' NICHT '=')
!=  ist ungleich
>=  ist gleich oder grösser
<=  ist gleich oder kleiner
>   ist grösser
<   ist kleiner
```

Es kann auch Code angegeben werden, der nur ausgeführt wird, wenn die Bedingung falsch ist.

```
zahl = 12;
if (zahl > 10) {
    Serial.println("Die Zahl ist grösser als 10");
    // Hier könnten noch beliebig viele Befehle stehen
} else {
    Serial.println("Die Zahl ist 10 oder kleiner");
}
Serial.println("Das wird immer geschrieben");
```

Zwischen den geschweiften Klammern {} können mehrere Befehle stehen. Sie bilden einen Block, der ausgeführt wird, wenn die Bedingung wahr ist. Wenn nur ein Befehl in einem Block vorhanden ist, kann {} normalerweise weggelassen werden. Ich würde das aber für den Moment nicht empfehlen.

Auch kompliziertere Varianten sind möglich:

```
zahl = 12;
if (zahl > 10) {
    Serial.println("Die Zahl ist grösser als 10");
} else if (zahl == 10) {
    Serial.println("Die Zahl ist genau 10");
} else {
    Serial.println("Die Zahl ist 10 oder kleiner");
}
Serial.println("Das wird immer geschrieben");
```

Funktionen an Funktionen übergeben

Das ist eine fortgeschrittene Funktion. Du wirst sie im Moment nicht brauchen. Da diese Möglichkeit vielen Arduino - Programmierern nicht bekannt ist, möchte ich sie hier ohne nähere Erklärung erwähnen:

```
int mult(int a, int b) {
    return a * b;
}

int plus(int a, int b) {
    return a + b;
}

void rechne(int (*operation)(int, int), int z1, int z2) {
    Serial.print(z1);

    if (operation == mult) {
        Serial.print(" x ");
    } else {
        Serial.print(" + ");
    }

    Serial.print(z2);
    Serial.print(" = ");
    Serial.println(operation(z1, z2));
}

void setup() {
    Serial.begin(9600);
    Serial.println("Ich rechne mit zwei Zahlen.");
    rechne(plus, 3, 5);
    rechne(mult, 7, 8);
}
```


Analoge Eingänge (Video 07)

Analoge Eingänge werden entweder gar nicht initialisiert oder mit ***pinMode(pinNummer,INPUT)*** in den richtigen Zustand gebracht.

Eingang abfragen

analogRead(pinNummer) gibt einen Wert zwischen 0 (entspricht 0 V) und 1023 (entspricht der Betriebsspannung) zurück. Zur bequemen Umrechnung kann

map(wert, vonMinumum, vonMaximum, zuMinimum, zuMaximum);

verwendet werden.

Beispiel:

```
int val = analogRead(A1);  
int res = map(val,0,1023,0,500);  
Serial.print(res / 100); Serial.println(" V");
```

gibt z. Bsp. 3.75 V auf den seriellen Monitor aus.

Globale und lokale Variablen

```
int oldVal = 0;
int val;
int volt;

const int inputPin = A0;

void messung() {
    val = analogRead(inputPin);
    volt = map(val,0,1023,0,485);
    if (val != oldVal) { // Ausgabe nur wenn geändert
        Serial.println(val);
        oldVal = val;
    }
}
```

Hier arbeiten wir mit den globalen Variablen **oldVal**, **val** und **volt**.

Diese werden aber nur innerhalb der Funktion **messung()** verwendet. Es wäre doch schön, wenn die Variablen auch nur dort sichtbar wären.

Das erreicht man mit lokalen Variablen.

```
const int inputPin = A0;

void messung() {
    int oldVal = 0;
    int val = analogRead(inputPin);
    int volt = map(val,0,1023,0,485);
    if (val != oldVal) { // Ausgabe nur wenn geändert
        Serial.println(val);
        oldVal = val;
    }
}
```

Jetzt sind die drei Variablen nur noch innerhalb der Funktion sichtbar. Leider wird unser Programm jetzt aber nicht mehr richtig funktionieren. Die Variable **oldVal** wird bei jedem Aufruf der Funktion auf 0 gesetzt, sie verliert also ihren früheren Wert. Muss jetzt **oldVal** doch global angelegt werden? Nein, dafür gibt es die **statische Variable**. Die Deklaration muss einfach so erfolgen:

```
static int oldVal = 0;
```

Beim ersten Mal wird die Variable auf 0 gesetzt, danach behält sie immer den aktuellen Wert, auch wenn die Funktion neu aufgerufen wird.

Analoge Ausgänge (Video 08)

Wert ausgeben

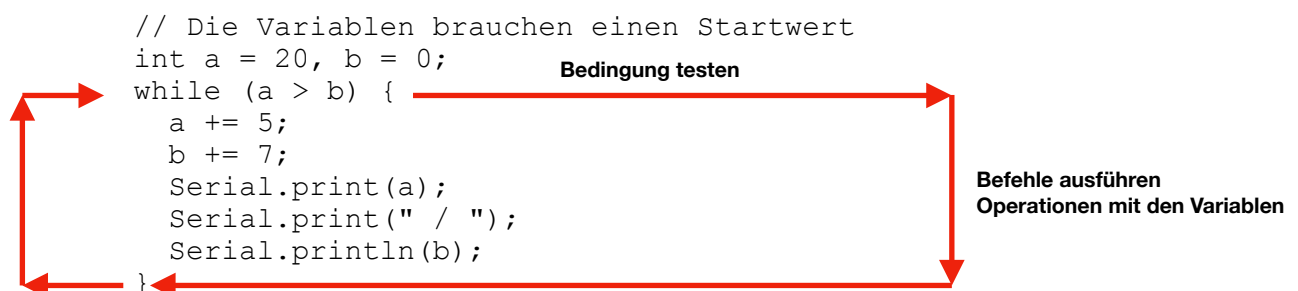
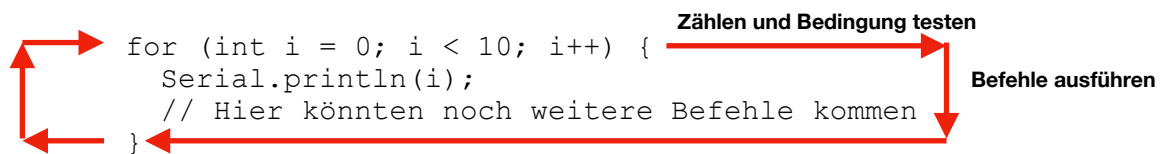
Analoge Ausgänge müssen nicht initialisiert werden. Sie dürfen aber mit ***pinMode(pinNummer, OUTPUT)*** gesetzt werden.

Mit ***analogWrite(pinNummer,wert)*** können Werte zwischen 0 und 255 ausgegeben werden.

Die Werte von den Eingängen können sehr einfach auf Ausgangswerte umgerechnet werden:

Beispiel:

```
analogWrite(3, analogRead(A0) / 4);
```



Schleifen und Datentypen (Video 09)

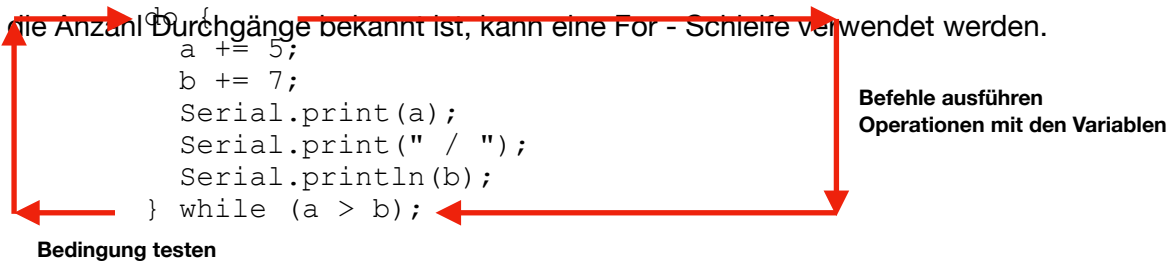
Wir haben ja gesehen, dass alles was in setup() steht genau einmal ausgeführt wird. Alles in loop() wird in einer Endlosschleife immer wieder ausgeführt. Was ist nun, wenn ich etwas genau 10 mal ausführen möchte oder bis eine bestimmte Bedingung erfüllt ist?

Genau zu diesem Zweck gibt es die Schleifen. Es gibt drei Typen, die für verschiedene Zwecke geeignet sind.

For - Schleife

Die Variablen brauchen einen Startwert
`int a = 20, b = 0;`

Wenn die Anzahl Durchgänge bekannt ist, kann eine For - Schleife verwendet werden.



int i = 0: lokale Zählervariable mit Startwert 0
i < 10: solange diese Bedingung erfüllt ist, läuft die Schleife
i++: so wird weitergezählt

i++ ist eine Kurzschreibweise für `i = i + 1`

While - Schleife

Eine While - Schleife testet eine Bedingung am Anfang der Schleife.

Bei dieser Art Schleife ist es möglich, dass sie gar nicht durchlaufen wird!

Do .. While - Schleife

Eine While - Schleife testet eine Bedingung am Ende der Schleife.

Diese Art Schleife wird immer mindestens einmal durchlaufen!

Break, Continue und Return

Es gibt noch andere Möglichkeiten Schleifen zu beeinflussen:

```
void irgendwas() {  
    Serial.begin(9600);  
    int a = 20, b = 0;  
    do {  
        a += 5;  
        b += 7;  
        if (a == 60) continue;  
        if (a == 50) break;  
        if (a == 30) return;  
        Serial.print(a);  
        Serial.print(" / ");  
        Serial.println(b);  
    } while (a > b);  
    Serial.println("Fertig");  
}
```

continue: Der Befehlsblock wird abgebrochen und mit dem nächsten Schleifendurchgang weiter gemacht.

break: Die Schleife wird sofort beendet und "Fertig" herausgeschrieben.

Return: Die ganze Funktion irgendwas() wird beendet und nicht einmal "Fertig" herausgeschrieben. **return** wird normalerweise benutzt um eine Funktion zu verlassen und einen Rückgabewert zu übergeben. Bei void - Funktionen (Funktionen ohne Rückgabewert) wird einfach die Funktion abgebrochen.

Zahlen und wie man damit rechnen kann

Je nach Zahlengrösse gibt es verschiedene ganzzahlige Datentypen

byte	8 bit	0 .. 255
word unsigned int	16 bit	0 .. 65535
short int	16 bit	-32768 .. 32767
unsigned long	32 bit	0 .. 4'294'967'295
long	32 bit	-2'147'483'648 .. 2'147'483'647

Bei Arduinos ohne Atmel - Prozessor können *int* und *unsigned int* andere Bitbreiten haben!

Datentypen wie *char* und *bool* entsprechen ebenfalls dem Typ *byte*. Diese werden aber später besprochen.

Es gibt ebenfalls verschiedene Typen mit Nachkommastellen

float double	32 bit	-3.4028235E+38 .. 3.4028235E+38 auf 6 - 7 Stellen genau
-----------------	--------	--

Bei Arduinos ohne Atmel - Prozessor kann double eine andere Bitbreite und Genauigkeit haben!

Mit diesen Zahlentypen sind verschiedene Rechenoperationen möglich

+	Addition	$i = i + 3$; $i += 3$; $i++$ entspricht $i = i + 1$
-	Subtraktion	$i = i - 3$; $i -= 3$; $i--$ entspricht $i = i - 1$
*	Multiplikation	$i = 3 * i$; $i *= 3$;
/	Division	$i = i / 5$; $i /= 5$; Wenn beide Teile ganzzahlig sind, wird das Resultat auch ganzzahlig. $10 / 4 ==> 2$; $10.0 / 4 ==> 2.5$
%	Rest	$10 \% 4 ==> 2$ dividend und Divisor müssen vom Typ <i>int</i> sein, das Resultat ist ebenfalls ein <i>int</i> .

Es sind zusätzlich noch einige mathematische Funktionen möglich

abs()	Absolutwert entfernt das Vorzeichen	abs(-5) ==> 5
sq()	Quadrat	sq(3) ==> 3 * 3 ==> 9
sqrt()	Quadratwurzel	sqrt(9) ==> 3
pow()	Exponent	pow(2,3) ==> 2 * 2 * 2 ==> 8
min()	Minimum	min(3,5) ==> 3
max()	Maximum	max(3,5) ==> 5
constrain()	Limitiert einen Wert	constrain(wert, min, max) Das Resultat entspricht wert , wenn wert zwischen min und max liegt. Sonst wird min oder max zurückgegeben.
map()	Bereichsumrechnung	map(wert, vonMin, vonMax, zuMin, zuMax)
cos()	Cosinus	Die Winkelfunktionen rechnen im Bogenmass, nicht mit Grad.
sin()	Sinus	$180^\circ = \pi = 3.14159$
tan()	Tangens	

Datentypen können konvertiert werden

Es stehen dazu die Funktionen byte(), int(), long(), word() und float() zur Verfügung.

Zahlenkonstanten können durch einen Zusatz im Typ festgelegt werden

100 ist ein int
100u oder 100U ist ein unsigned int
100l oder 100L ist ein long
100ul oder 100UL ist ein unsigned long
100.0 ist ein float
 $2.34E5$ entspricht $2.34 * 10^5$

Zahlen können auch in anderen Zahlensystemen angegeben werden

13	13 als Dezimalzahl
B00001101	13 als Binärzahl
015	13 als Oktalzahl
0x0D	13 als Hexadezimalzahl

Arrays (Video 10)

Arrays sind einfache Listen von Werten des gleichen Typs.

Deklaration eines Arrays

```
int liste[5];
```

Das erstellt eine Liste von 5 Integer-Werten. Die Werte sind noch undefiniert, es wird nur der notwendige Speicherplatz reserviert.

```
int liste[5] = {2,4,3};
```

Es wird eine Liste mit 5 Elementen erstellt. Die ersten 3 Elemente enthalten einen Wert, die restlichen 2 sind undefiniert.

```
float liste[] = {1.5, 2.1, 3.7};
```

Es wird eine Liste mit 3 Elementen erstellt. Alle Elemente enthalten einen Wert.

Ein Array kann später nicht erweitert werden. Es müssen bei der Deklaration des Arrays genügend Elemente reserviert werden. Diese Aussage bezieht sich nur auf einfache C - Arrays. C++ hätte da noch Alternativen, die aber hier kein Thema sind.

Verwendung von Arrays

Die Möglichkeiten beschränken sich auf das Auslesen und setzen der einzelnen Werte. Ein einzelnes Element kann über seinen Index angesprochen werden. Die Nummerierung der Indices beginnt immer mit 0!

```
int liste[5] = {1,2,3,4,5};
```

```
int a = liste[0];
```

Gibt den Wert 1 zurück.

```
int a = liste[5];
```

Gibt einen zufälligen Wert zurück. Da die Nummerierung bei 0 beginnt, umfasst die Liste nur den Bereich liste[0] bis liste[4].

```
liste[1] = 10;
```

Setzt den 2. Wert auf 10.

```
liste[5] = 10;
```

Schreibt den Wert 10 in einen Speicherplatz, der nicht zum Array liste[] gehört. Das kann zu schweren Störungen oder einem Programmabsturz führen.

Arrays und Funktionen

Arrays können als Parameter einer Funktion übergeben werden. Dabei muss aber eine Spezialität berücksichtigt werden.

```
int liste[] = {1,2,3,4,5};
```

```
void print(int lst[]) {  
    Serial.println(n[1]);  
    n[1] = 10;  
}
```

```
print(liste);  
print(liste);
```

Gibt

2
10

aus. Das Array wird der Funktion als Zeiger auf das erste Element übergeben. Daher verändert die Zuweisung innerhalb der Funktion das Originalarray.

Grösse eines Arrays ermitteln

Die totale Anzahl der Elemente in einem Array kann einfach ermittelt werden:

```
anzahl = sizeof(liste) / sizeof(liste[0]);
```

Das funktioniert aber nur mit der Original - Variablen. Ein als Parameter übergebenes Arrays besteht nur noch aus der Adresse des ursprünglichen Arrays!

Der Zufall und weitere Arrays (Video 11)

Mehrdimensionale Arrays

Anstelle die 5 Arrays einzeln zu deklarieren:

```
int eins[] = {0,0,0,0,1};
int zwei[] = {1,0,0,1,0};
int drei[] = {1,0,0,1,1};
int vier[] = {1,1,1,1,0};
int fuenf[] = {1,1,1,1,1};
```

ist es auch möglich, alle zusammen in einem einzigen mehrdimensionalen Array zusammenzufassen:

```
int anzeige[][5] = {
    {0,0,0,0,0},
    {0,0,0,0,1},
    {1,0,0,1,0},
    {1,0,0,1,1},
    {1,1,1,1,0},
    {1,1,1,1,1}
};
```

Die Arraygrösse darf nur bei maximal einer Dimension offen gelassen werden. Bei allen anderen Dimensionen muss sie angegeben werden.

Der Zugriff auf den Zustand einer Led kann dann mit `anzeige[zahl][ledNummer]` abgefragt werden. Nach diesem Muster können Arrays mit beliebig vielen Dimensionen erstellt werden.

Der Zufall

Der Befehl zur Erzeugung einer Zufallszahl lautet `random()`. Bei dieser Zahl handelt es sich immer um einen ganzzahligen long - Wert.

```
random() gibt einen Wert zwischen 0 und 2'147'483'647 zurück.
random(10) gibt einen Wert zwischen 0 und 9 zurück.
random(5,10) gibt einen Wert zwischen 5 und 9 zurück.
```

Es sind auch negative Werte möglich:

```
random(-5,6) gibt einen Wert zwischen -5 und 5 zurück.
random(-100,-50) gibt einen Wert zwischen -100 und -51 zurück.
```

Bei `random()` handelt es sich um einen Pseudozufallsgenerator. Vereinfacht gesagt, gibt er bei identischer Ausgangslage immer dieselbe Sequenz zurück. Das ist vielfach unerwünscht. Daher können wir diese Ausgangslage verändern.

Dazu dient der Befehl `randomSeed()`.

Durch die Initialisierung mit `randomSeed(10)` erhalten wir eine andere Sequenz zurück als bei Initialisierung mit `randomSeed(123)`. Jede mit `randomSeed(10)` initialisierte Sequenz ist aber identisch. So müssen wir eine Initialisierung mit einer zufälligen Zahl vornehmen. Unbeschaltete analoge Eingänge geben zufällige Werte zurück. Daher kann die Initialisierung mit

```
randomSeed(analogRead(0));
```

durchgeführt werden.

switch .. case

Dieser Befehl könnte auch durch mehrere if .. else if ersetzt werden. Es gibt aber einige Fälle, in denen das switch .. case - Konstrukt wesentlich übersichtlicher ist. Es bietet aber für den Einsteiger einige Fallstricke.

```
switch (<vergleichsvariable>) {
    case <vergleichswert> :
        // beliebig viele Befehle;
        break;
    case <vergleichswert>:
        // beliebig viele Befehle;
        break;
    default:
        // beliebig viele Befehle
}
```

Die **Vergleichsvariable** muss vom Typ **int** oder **char** sein. Der **Vergleichswert** muss eine Konstante desselben Typs sein. Die Befehle nach **default:** werden ausgeführt, wenn keine andere Übereinstimmung gefunden wurde.

Soweit ist alles klar. Wozu ist aber der **break** Befehl notwendig?

Switch .. case verhält sich weniger intelligent als wir zuerst annehmen. Er vergleicht die Variable mit der entsprechenden Konstanten und überspringt die Befehle, solange er keine Übereinstimmung findet. Bei der ersten Übereinstimmung stellt er die Vergleiche ein und führt einfach alle Befehle aus, die weiter unten noch kommen. Deshalb müssen wir manuell mit **break** abbrechen, wenn wir wollen, dass er nur die Befehle der gefundenen Übereinstimmung ausführt.

Hier einige Beispiele:

```
int i = 2;
switch (i) {
    case 5:
        Serial.println("5");
    case 2:
        Serial.println("2");
    default:
        Serial.println("sonstwas");
}
```

Gibt
2
sonstwas
aus.

Das ist ein typischer Anfängerfehler: **break** wurde vergessen. Richtig wäre:

```
int i = 2;
switch (i) {
    case 5:
        Serial.println("5");
        break;
    case 2:
        Serial.println("2");
        break;
    default:
        Serial.println("sonstwas");
}
```

Zusammenfassung Programmierung

Dieses Verhalten kann man aber auch ausnützen um eine Oder - Verknüpfung zu erstellen:

```
int i = 2;
switch (i) {
    case 5:
    case 2:
        Serial.println("2 oder 5");
        break;
    default:
        Serial.println("sonstwas");
}
```

Der Text '2 oder 5' wird ausgegeben, wenn i entweder 2 oder 5 ist.

Das richtige Timing (Video 12)

Der Befehl **delay()** ist in vielen Fällen nicht geeignet, um das richtige Timing innerhalb eines Programmes sicherzustellen. Der grösste Nachteil ist der blockierende Charakter von `delay()`. Innerhalb eines Delays reagiert der Prozessor auf keine der angeschlossenen Sensoren. Mit Interrupts könnte man das Problem umgehen, doch diese stehen auch nicht unbegrenzt zur Verfügung. Hier sehen wir eine andere Lösung, die sehr vielseitig eingesetzt werden kann. Interrupts werden in einer späteren Lektion angeschaut.

Der Arduino enthält keine Echtzeituhr. Es ist also nicht möglich ohne zusätzliche Hardware die genaue Uhrzeit abzufragen. Wir können aber abfragen, wieviel Zeit seit dem Start des Programmes vergangen ist.

millis()

Das ist die Zeit in Millisekunden seit dem Start.

`millis()` gibt einen **unsigned long** zurück. Nach etwa 50 Tagen erfolgt ein Überlauf und die Zählung beginnt wieder bei Null.

micros()

Das ist die Zeit in Mikrosekunden seit dem Start.

Es handelt sich dabei ebenfalls um einen **unsigned long** und daher erfolgt hier der Überlauf bereits nach etwa 70 Minuten.

Der Arduino ist nicht in der Lage auf einzelne Mikrosekunden aufzulösen. Je nach Taktfrequenz erfolgt die Angabe in Schritten von 4 Mikrosekunden (Prozessoren mit 16 MHz Clock) oder 8 Mikrosekunden (8 MHz).

Statische Variablen

In der Lektion findest du Beispiele, in denen

```
static unsigned long startZeit = 0;
```

verwendet wird. Wozu ist das gut?

Grundsätzlich gibt es **globale** und **lokale** Variablen.

Globale Variablen werden normalerweise am Anfang des Programms deklariert und können an jeder Stelle des Programms verändert oder abgefragt werden.

Lokale Variablen sind zum Beispiel innerhalb einer Funktion oder eines Blocks deklariert und können auch nur dort verwendet werden. Sobald man die Funktion verlässt, verliert die Variable ihren Wert.

Statische Variablen sind ebenfalls innerhalb einer Funktion deklariert und können auch nur dort abgefragt oder verändert werden. Sonst verhalten sie sich aber wie globale Variablen. Beim ersten Durchgang werden sie auf den deklarierten Ausgangswert gesetzt (`startZeit = 0`). Beim Verlassen der Funktion behalten sie aber ihren Wert und können diesen beim nächsten Durchlauf wieder verwenden.

Genau das nutzen wir in unserer universellen Blinkfunktion aus.

Universelle Blinkfunktion

```
void blink() {  
    const int pin = 7; // Pinnummer  
    const int blinkZeit = 1000; // Zeit bis zum nächsten Wechsel  
    static unsigned long startZeit = 0; // Letzter Wechsel  
  
    if (millis() > startZeit + blinkZeit) { // Zeit zum wechseln?  
        digitalWrite(pin,!digitalRead(pin));  
        startZeit = millis(); // Aktuelle Zeit als neue Startzeit  
    }  
}
```

Wenn zusätzlich eine andere Led blinken soll, kann diese Funktion einfach kopiert werden. Es müssen nur die Konstanten ***pin*** und ***blinkZeit*** angepasst werden. Alle Blinkfunktionen können direkt hintereinander aufgerufen werden. Es erscheint dann so, als würden sie alle gleichzeitig ablaufen.

Spielereien mit millis() (Video 13)

Datentyp bool

In dieser Lektion sind nicht viele neue Sprachelemente enthalten. An einer Stelle wird aber eine Variable vom Typ **bool** deklariert.

Der Datentyp **bool** kann nur zwei Werte annehmen: **true** oder **false** (wahr oder falsch). Damit ist er in der Lage, Resultate eines Vergleichs zu speichern.

```
bool b1, b2;  
int a= 5; b= 10;
```

```
b1 = (a == 5);  
b2 = (b == 9);
```

b1 ist jetzt **true**, **b2** ist jetzt **false**

```
b1 = !b2;
```

b1 wird so auf **NICHT b2** gesetzt, wird also **true**.

bool existierte sehr lange nicht in C. Ältere Compiler kennen den Typ nicht. Seine Umsetzung ist auch sehr primitiv, eigentlich könnte auch ohne ihn gearbeitet werden.

Ein **bool** entspricht einem **byte**, belegt also 8 Bit. Dasselbe gilt übrigens auch für **char**, aber das ist eine andere Geschichte.

Jedes Byte mit dem Wert 0 entspricht einem **false**, alle anderen Werte entsprechen **true**. Aus diesem Grund kann ich Folgendes schreiben:

```
byte a = 5;  
byte b = 0;  
bool b1, b2;
```

```
b1 = (a);  
b2 = (b);
```

b1 wird dabei **true** und **b2** **false**.