

Inhaltsverzeichnis

Ein Einstieg (bis Video 06)

Arduino Sketch	5
Befehlssequenzen	5
Zeitverzögerungen	5
Digitale Ein- und Ausgänge	6
Der serielle Monitor	6
Konstanten und Variablen	7
Funktionen	8
Entscheidungen (if .. else)	9
Funktionen an Funktionen übergeben	10

Analoge Eingänge (Video 07)

Eingang abfragen	11
Globale und lokale Variablen	12

Analoge Ausgänge (Video 08)

Wert ausgeben	13
---------------	----

Schleifen und Datentypen (Video 09)

For - Schleife	14
While - Schleife	14
Do .. While - Schleife	15
Break, Continue und Return	15
Zahlen und wie man damit rechnen kann	16

Arrays (Video 10)

Deklaration eines Arrays	18
Verwendung von Arrays	18
Arrays und Funktionen	19
Grösse eines Arrays ermitteln	19

Der Zufall und weitere Arrays (Video 11)

Mehrdimensionale Arrays	20
Der Zufall	20
switch .. case	21

Das richtige Timing (Video 12)

millis()	23
micros()	23
Statische Variablen	23
Universelle Blinkfunktion	24

Spielereien mit millis() (Video 13)

Datentyp bool	25
Servos (Video 14)	
Bibliothek, Library	26
Verwendung von Klassen aus Libraries	26
Die Dokumentation der Klasse Servo	26
Die Funktionen der Klasse Servo	27
Nur wechselnde Werte an den seriellen Monitor ausgeben	27
Ab jetzt mit Display (Video 15)	
LiquidCrystal Library	28
Einen konstanten String einer Funktion übergeben	28
Ausgabe von Umlauten auf das Display	29
Der Ultraschall-Sensor (Video 16)	
pulseIn()	30
Die New Ping Library	30
Frühere Videos	30
Der Bewegungssensor (Video 17)	
Anschluss und Programmierung	31
Sensoren als Module (Video 18)	
Ein Modul für die Abfrage der Lichtstärke via LDR	32
Mit LED - Ausgabe (Video 19)	
Default - Parameter bei Funktionsaufrufen	33
Die String - Klasse	33
Das Erstellen von Modulen	34
Vom Modul zur Klasse (Video 20)	
Deklaration und Definition im .h - File	36
Implementation der Klasse	36
Verwenden der Klasse	37
War das alles?	37
Temperatur und Luftfeuchte (Video 21)	
Die Hardware	38
Die Software	39
Speicheradressen und Zeiger (Video 22)	
Deklaration eines Zeigers	40
Adresse einer Variablen	40
Wert der Variablen, auf die der Zeiger zeigt	40
Funktionen (Video 23)	
Einfache Funktionen	41

Übergabe als Wert	42
Übergabe als Pointer	42
Übergabe als Referenz	42
Spezialfall Array und Zeichenkette	43
Auch Funktionen können übergeben werden	43

Lektion 24: EEPROM, das Gedächtnis des Arduinos

Die Ansteuerung des EEPROMs	44
Speichern der Einstellungen	44
Laden der Einstellungen	45

Der Highspeed - Arduino (Video 25)

Direkter Registerzugriff	46
--------------------------	----

Von der Idee zum Projekt (Video 26)

Die Idee oder das Pflichtenheft	47
Die Hardware	47
Das Grundgerüst der Software	48
Servo und Potentiometer	48
Blinken der gelben Led und weitere Tests	49
Wird die Frequenz erhöht oder vermindert?	49
Anzeigen der Richtung mit den LEDs	50
Zusammenfassung	51

Strings (Zeichenketten) 1 (Video 27)

Erstellen eines Strings	52
Umwandlung von Zahlen in Strings	53
Zwei Strings zusammenfügen	54

Die String - Klasse (Video 28)

Erstellen eines Strings	55
Strings in Zahlen umwandeln	55
Strings zusammenfügen	56
Strings vergleichen	56
Indexierung	57
Weitere Funktionen	57

... und wo bleibt das & Co.? (Video 29)

Der Präprozessor (Video 30)

#define	58
#ifdef, #ifndef	59
#if	59
#include	59
Weitere Befehle	60

Klassen und Objekte (Video 30 bis 34)

Zusammenfassung Programmierung

Eine einfache Klasse	61
Eine neue Klasse, die alle Eigenschaften der Basisklasse erbt	62
Noch eine Klasse	64
Instanzvariablen und Klassenvariablen	65
Vererbung aus der Basisklasse	66

Von der Klasse zur Bibliothek (Video 35 - 36)

Erben aller Eigenschaften von Servo	68
Der Servo bekommt eine Bremse	69
delay() - Vermeidung im Testprogramm	70
Die Dateistruktur	71
Das ZIP - File	73

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 Interrupt, 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);
```

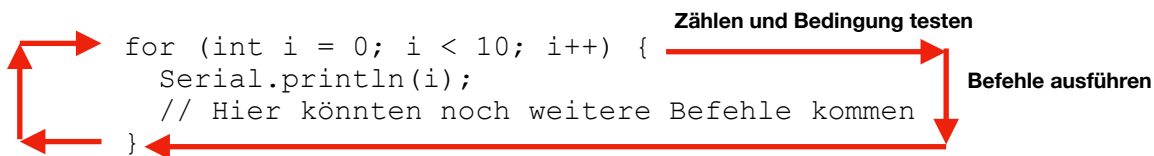
Schleifen und Datentypen (Video 09)

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

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

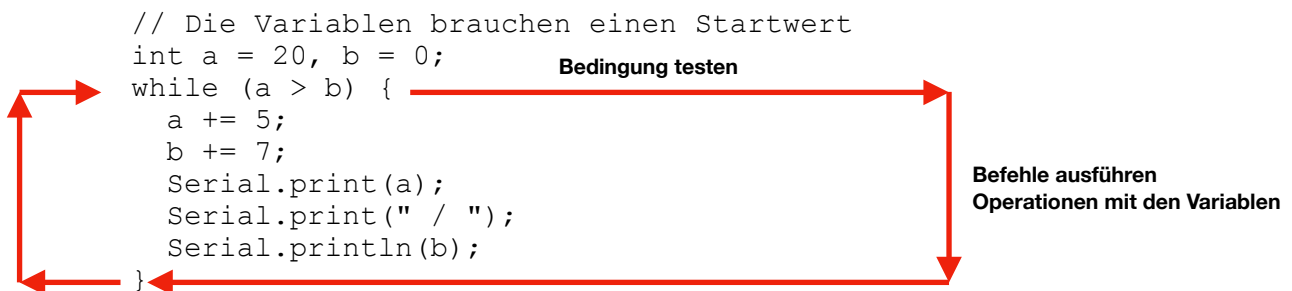


int i = 0: lokale Zählervariable mit Startwert 0
i < 10: solange diese Bedingung erfüllt ist, läuft die Schleife
i++: so wird weitergezählt

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

While - Schleife

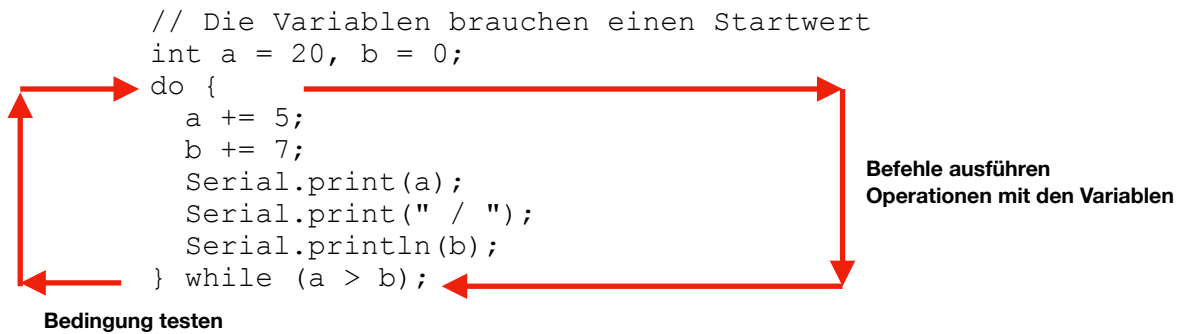
Eine While - Schleife testet eine Bedingung am Anfang der Schleife.



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

Do .. While - Schleife

Eine Do .. 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 Befehlblock 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	32 bit	-3.4028235E+38 .. 3.4028235E+38
double		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° = π = 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**.

Servos (Video 14)

Bibliothek, Library

Bibliotheken, auch Libraries genannt, sind fertige Programmstücke, die einen bestimmten Zweck erfüllen. Hier verwenden wir die Servo - Bibliothek. Sie erlaubt uns einen Servo zu steuern, ohne dass wir uns um die Details kümmern müssen.

Mit Hilfe des Präprozessor - Befehls **#include <Servo.h>** können wir die Library in unser Programm einbinden. Servo.h ist eine sogenannte Headerdatei (darum .h), wir werden das aber heute nicht näher betrachten.

Verwendung von Klassen aus Libraries

Oft werden die Funktionen einer Bibliothek in Form von Klassen bereit gestellt. Klassen sind Baupläne, die uns ermöglichen ein Objekt zu erstellen, das verschiedene Funktionen und Eigenschaften zur Verfügung stellt. In unserem Beispiel repräsentiert das Objekt das physische Bauteil Servo.

```
Servo servo1;
```

Auf diese Art erzeugen wir das Objekt servo1, das die Funktionen eines Servos zur Verfügung stellt. Man nennt dieses auch eine Instanz der Klasse Servo. Wenn mehrere Servos angesprochen werden müssen, erzeugt man einfach mehrere Instanzen der Klasse. Jede davon kann dann unabhängig gesteuert werden.

Unser Softwareobjekt ist jetzt aber noch nicht mit unserem Bauteil verbunden. Dies geschieht durch die Angabe des verwendeten Anschlusses.

```
servo1.attach(8);
```

Jetzt ist unser Softwareobjekt mit dem Servo verbunden, der an Pin 8 angeschlossen ist. attach() ist eine Funktion, die vom Servo zur Verfügung gestellt wird. Man spricht in der objektorientierten Programmierung normalerweise von einer Methode.

Die Klasse Servo ist eigentlich ein schlechtes Beispiel zur Erklärung von Klassen, da hier nur Methoden, nicht aber Eigenschaften zur Verfügung gestellt werden. Trotzdem belassen wir es dabei, da wir später noch Gelegenheit haben werden, komplexere Klassen kennenzulernen.

Die Dokumentation der Klasse Servo

Um mit einer Bibliothek oder Klasse arbeiten zu können, brauchen wir eine Dokumentation. Diese findet man für Standardbibliotheken auf <https://www.arduino.cc/en/Reference/Libraries> .

Die Dokumentation von Servo findet man unter <https://www.arduino.cc/en/Reference/Servo> . Nebst der Beschreibung der Funktionsweise der Library finden wir auch die Funktionen der Klasse Servo. Ebenfalls ist ein Hinweis auf verfügbare Demoprogramme enthalten.

Die Funktionen der Klasse Servo

<code>attach(8;</code>	Verbindet das Objekt mit dem Servo, der an Pin 8 angeschlossen ist.
<code>attach(8,600,2000);</code>	Verbindet das Objekt mit dem Servo, der an Pin 8 angeschlossen ist. Die PWM-Pulsbreiten werden dabei auf 600 bis 2000 Mikrosekunden beschränkt. Standard wären 544 bis 2400. Wir verwenden dieses Befehl in der Lektion nicht. Es wäre aber gut, einmal selbst damit zu experimentieren.
<code>detach(;</code>	Die Verbindung zum Servo wird wieder gelöst.
<code>attached();</code>	Gibt den Pin zurück, mit dem der Servo verbunden ist.
<code>write(70);</code>	Der Servo fährt die Position 70 Grad an. Der Bereich erstreckt sich von 0 bis 180 Grad.
<code>read();</code>	Es wird zurückgegeben, welche Position zuletzt mit <code>write()</code> angefahren wurde.
<code>writeMicroseconds(1300);</code>	Gibt an den Servo Pulse mit der Breite 1300 Mikrosekunden weiter.

Nur wechselnde Werte an den seriellen Monitor ausgeben

Wenn wir Werte an den seriellen Monitor ausgeben, laufen die oft einfach sehr schnell durch, obwohl sie beinahe immer identisch sind. Es ist aber einfach, nur einen Wert herauszuschreiben, wenn er geändert hat.

```
void ausgabe(int aktuell) {  
    static int alt = -1; // Initialisierung auf nicht vorkommenden Wert  
    if (aktuell != alt) {  
        Serial.println(aktuell);  
        alt = aktuell;  
    }  
}
```

Den alten Wert speichern wir in der Variable `alt`. Diese muss `static` sein, da sie den Wert behalten soll, wenn wir die Funktion verlassen. Falls der aktuelle Wert gegenüber dem alten geändert hat, schreiben wir den aktuellen Wert auf den Monitor und setzen `alt` auf den soeben herausgeschriebenen Wert.

Ab jetzt mit Display (Video 15)

LiquidCrystal Library

Laden der Bibliothek:

```
#include <LiquidCrystal.h>
```

Objekt lcd erzeugen:

```
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // Pins für rs, en, d4, d5, d6, d7
```

Start des Displays:

```
lcd.begin(16, 2); // 16 Zeichen, 2 Zeilen
```

Schreiben eines Textes an der aktuellen Cursorposition:

```
lcd.print("Hallo Arduino");
```

Cursor positionieren:

```
lcd.setCursor(0,1); // Spalte, Zeile
                    // erstes Zeichen der zweiten Zeile
                    // Die Nummerierung der Zeichen
                    // und Zeilen beginnt bei 0
```

Code auf Display schreiben:

```
lcd.write(0x41); // schreibt ein A
```

Weitere Befehle der Library findest du auf <https://www.arduino.cc/en/Reference/LiquidCrystal>

Einen konstanten String einer Funktion übergeben

Definition der Funktion:

```
void printX(const char a[])
```

Aufruf:

```
printX("Hallo");
```

Zeichenweises Verarbeiten des Strings in der Funktion:

```
void printX(const char a[]) {
    int i = 0;
    while (true) {
        char ch = a[i++];
        if (ch == 0) return;
        lcd.write(ch);
    }
}
```

Die Adresse des Strings "Hallo" wird an die Funktion weitergegeben. Das Array a[], beziehungsweise das erste Element (a[0]) zeigt auf das erste Zeichen des Strings (H).

Danach wird Zeichen um Zeichen ausgelesen und auf das lcd geschrieben.

```
ch = a[i++];
ist eine Kurzschreibweise von
ch = a[i];
i = i + 1;
```

Ausgabe von Umlauten auf das Display

Dazu wurde eine Funktion definiert:

```
void printX(const char a[]) {
    int i = 0;
    while (true) {
        char ch = a[i++];
        if (ch == 0) return;
        if (ch == '&') {
            ch = a[i++];
            switch (ch) {
                case 'a': lcd.write(0xE1); // ä
                           break;
                case 'o': lcd.write(0xEF); // ö
                           break;
                case 'u': lcd.write(0xF5); // ü
                           break;
                case 'g': lcd.write(0xDF); // °
                           break;
                case 'O': lcd.write(0xF4); // Ω
                           break;
                case 'U': lcd.write(0xF3); // ∞
                           break;
                case 'S': lcd.write(0xF6); // Σ
                           break;
                default: if (ch < 128) lcd.write(ch);
            }
        } else if (ch < 128) {
            lcd.write(ch);
        }
    }
}
```

Aufruf:

```
printX("&a &o &u &g &O &U &S"); // ä ö ü ° Ω Σ
```

Der Ultraschall-Sensor (Video 16)

pulseIn()

Um mit den Ultraschallsensor HC-SR04 eine Distanz zu messen, muss der Arduino die Plusbreite eines Eingangssignals ermitteln können. Dazu ist der pulseIn() - Befehl vorgesehen.

Aufruf:

```
unsigned long pulseIn(pin, wert, timeout);
```

Rückgabewert: Pulsbreite in Mikrosekunden.

pin: Pinnummer des Eingangs

wert: HIGH oder LOW, je nach dem der Puls gegen HIGH oder gegen LOW geht

timeout: Timeout in Mikrosekunden. Sobald diese Zeit abgelaufen ist, wird 0 zurückgegeben

Um die Distanz in cm zu erhalten, muss der Rückgabewert durch 58 dividiert werden.

Ein timeout - Wert von 5000 ergibt etwa 70 cm Maximaldistanz.

Bei grossen Distanzen muss mit einem grossen timeout - Wert gearbeitet werden. Dadurch kann das Programm über längere Zeit blockiert werden. Falls kein timeout-Wert angegeben wird, beträgt dieser 1000 ms. Das ist in den meisten Fällen nicht akzeptabel.

Die New Ping Library

Diese Library erleichtert die Distanzmessung. Es ist eine von vielen Libraries, die versprechen eine bessere Variante zu bieten. Sie kann über die Bibliotheksverwaltung installiert werden.

Initialisierung:

```
const int trigger_Pin = 7;  
const int echo_Pin = 8;  
const int maxDistanz = 80; // 80 cm
```

```
NewPing sonar(trigger_Pin,echo_Pin,maxDistanz);
```

Die Abfrage ist dann sehr einfach:

```
int cm = sonar.ping_cm();
```

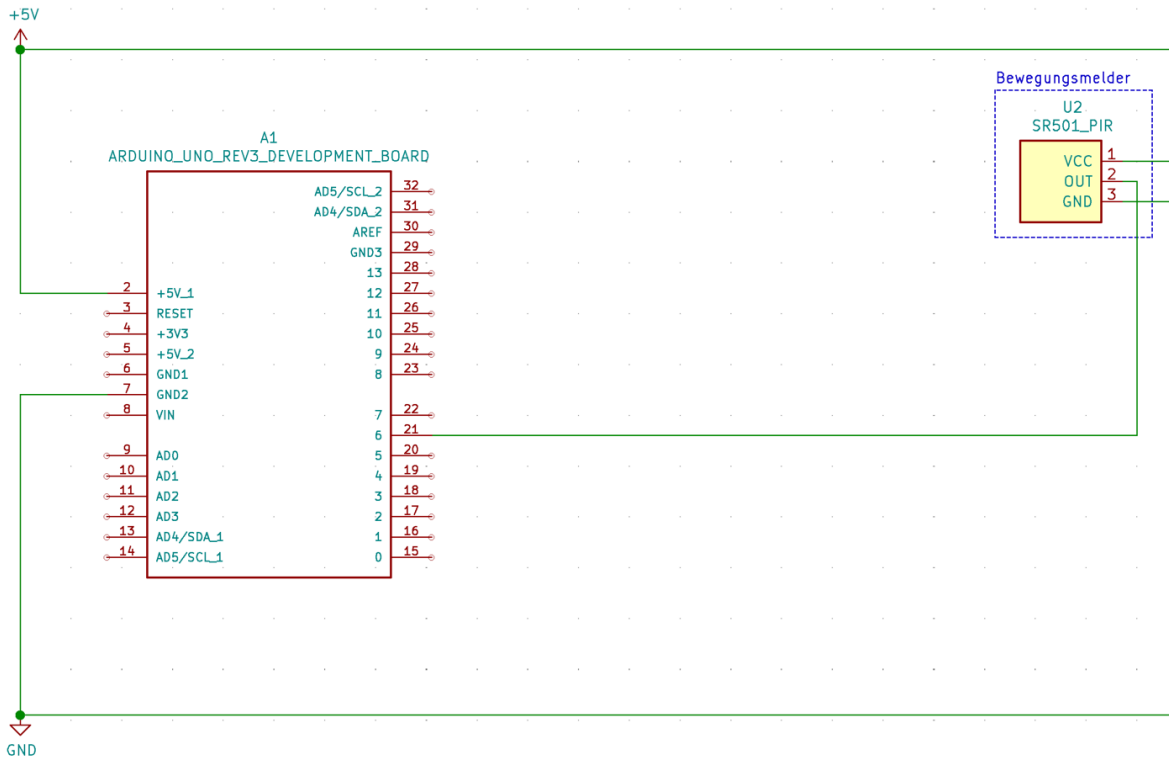
Frühere Videos

Artikel und Video auf hobbyelektroniker.ch: <https://www.hobbyelektroniker.ch/roboter/ultraschall/>
Der Ultraschallsensor und das Oszilloskop: <https://youtu.be/HQJazRhYMdE>

Der Bewegungssensor (Video 17)

Wir verwenden einen Bewegungssensor vom Typ HC-SR05. Generelle Informationen über PIR - Sensoren findet man auf Wikipedia: https://de.wikipedia.org/wiki/Pyroelektrischer_Sensor

Anschluss und Programmierung



Der Signalpin gibt HIGH aus, wenn eine Bewegung festgestellt wurde. Das werden wir mit der Funktion

```
boolean bewegung() {  
    return (digitalRead(pir_Pin));  
}
```

aus. Diese Funktion gibt den Datentyp **boolean** zurück. Eine detaillierte Beschreibung dieses Datentyps und seiner möglichen Operationen erfolgt später. Hier nur einige grundsätzliche Anmerkungen.

boolean kann nur zwei Werte annehmen: **true** oder **false**. Das kann als wahr oder falsch, ja oder nein interpretiert werden. Ein solcher Wert kann direkt als Bedingung in einer if - Abfrage verwendet werden.

```
if (bewegung()) ...
```

boolean ist in der Dokumentation des Arduinos aufgeführt. Es gibt in C/C++ noch den Typ **bool**, der als identisch betrachtet werden kann.

Sensoren als Module (Video 18)

Wir erweitern unser Programm stetig. Damit wir die Übersicht behalten, teilen wir es in mehrere Dateien auf. Jeder Sensor erhält ein eigenes Modul (Dateipaar **xxx.h** und **xxx.cpp**). Dadurch kann er unabhängig von den anderen Sensoren programmiert werden.

Zusätzlich wird immer eine Datei **globals.h** erstellt, die globale Variablen und Konstanten enthält.

Eine detaillierte Anleitung zum Erstellen von Modulen kommt im nächsten Video.

Ein Modul für die Abfrage der Lichtstärke via LDR

Wir nennen das Modul **licht** und erstellen dazu die Dateien **licht.h** und **licht.cpp**.

Beginnen wir mit **licht.h**:

Hier überlegen wir und, was das Hauptprogramm von diesem Sensor wissen will. Das ist hier ganz einfach die Lichtstärke als Zahl zwischen 0 (dunkel) und 1023 (hell).

Diese Funktion nennen wir **licht_staerke()**. Der Name einer solchen Modulfunktion beginnt immer mit dem Namen des Moduls. Dadurch vermeiden wir Namenskonflikte.

```
int licht_staerke();
```

Im **.h - File** verzichten wir auf die Implementierung des Befehls. Hier wird also nicht gesagt, wie wir zu dem Wert kommen.

Die eigentliche Implementation erfolgt im **.cpp - File**. Dort programmieren wir die Abfrage des Lichtsensors. Die **.cpp - Files** 'sehen' einander nicht. Daher gibt es zwischen ihnen keine Namenskonflikte.

```
// Pin - Definitionen
const int pin = A2;

int licht_staerke() {
    return analogRead(pin);
}
```

Im Hauptprogramm laden wir die Datei **licht.h** mit

```
#include "licht.h"
```

Dann lässt sich die Helligkeit mit

```
licht_staerke()
```

abfragen.

Da gewisse Sensoren eine Initialisierung in **setup()** benötigen, wird in jedem Modul eine Init - Funktion zur Verfügung gestellt. Zusätzlich ist es sinnvoll, eine Versionsabfrage zu implementieren. Deshalb stellt unser Licht - Modul drei Funktionen zur Verfügung:

```
int licht_staerke();
void licht_init();
int licht_version();
```


Mit LED - Ausgabe (Video 19)

Default - Parameter bei Funktionsaufrufen

Im Modul **display.h** verwenden wir die Funktion

```
void display_print(String line1, String line2 = "");
```

Hier wird beim zweiten Parameter ein Standardwert angegeben. Dadurch kann der Aufruf auf zwei Arten erfolgen:

```
display_print("Erste Zeile", "Zweite Zeile");
```

oder

```
display_print("Nur eine Zeile");
```

In diesem Fall setzt das Programm für **line2** den angegebenen Standardwert ein.

Die String - Klasse

Wir verwenden hier die **String - Klasse**, die uns die Arduino Entwicklungsumgebung zur Verfügung stellt. Sie ist sehr komfortabel, aber nicht ganz unumstritten.

Ein String ist ja nichts anderes als eine Zeichenkette. Diese wird in C normalerweise als offener Array von Zeichen realisiert. Eine 0 (Wert 0, nicht das Zeichen '0') am Schluss signalisiert, dass der String zu Ende ist. Das kann sehr heikel sein und führt auch immer wieder zu gefährlichen Fehlern.

Auch die String - Klasse verwendet diesen Mechanismus, stellt aber gegenüber dem Programmierer komfortable und sichere Methoden zur Behandlung der Zeichen zur Verfügung. Das hat aber seinen Preis. Der Speicherbedarf ist relativ hoch und leider ist dieser beim Arduino oft etwas knapp. Ein Stringobjekt managt seinen Speicher selbst und reserviert dynamisch die benötigten Speicherbereiche und gibt nicht mehr benötigte frei. Da in C kein Mechanismus zum Aufräumen dieser Speicherblöcke existiert (garbage collection), können freigegebene Speicherblöcke unter Umständen nicht mehr genutzt werden. Das führt dann zu einer sogenannten Heap - Fragmentierung. Diese kann dazu führen, dass unser Speicher überläuft, obwohl eigentlich noch freier Platz vorhanden wäre. Daher sollten beim Einsatz von String einige Dinge beachtet werden.

Für den Moment machen wir uns aber keine Sorgen und freuen uns am einfachen Umgang mit Zeichenketten.

Insbesondere die Umwandlung von Zahlentypen in Strings ist sehr elegant gelöst. Ausserdem lassen sich diese String einfach mit + verknüpfen.

```
const int pin = 1;  
Serial.println(String("Wir messen an Pin ") + String(pin) + String(" einen Wert von ") + String(analogRead(pin), BIN));
```

gibt auf dem seriellen Monitor den Text aus:

Wir messen an Pin 1 einen Wert von 110010

Zusammenfassung Programmierung

String ist nicht direkt kompatibel mit `char[]`, daher muss die Übergabe eines String - Objektes an einen `char[]` Parameter etwas speziell vorgenommen werden.

```
void printX(const char a[]) {  
    ...  
}
```

```
String s = "Hallo";  
printX(s.c_str());
```

Die Methode `c_str()` gibt einen Zeiger auf das Char - Array zurück und kann daher übergeben werden. Details dazu werden in einem späteren Video folgen.

Das Erstellen von Modulen

Bei diesen Modulen handelt es sich nicht um ein normales Sprachkonstrukt. Sie wurden in diesem Kurs eingeführt, um grössere Programme zu strukturieren und übersichtlich zu halten. Als fortgeschrittener Programmierer würde man eigene Klassen erstellen. Das ist aber ein Thema, das für einen Anfängerkurs nicht geeignet ist.

1. Namen festlegen

Unser Beispielm modul soll **LEDs** heissen.

2. Dateien erzeugen

Wir erzeugen **LEDs.h** und **LEDs.cpp**

3. In LEDs.h Interface definieren

Es muss immer ein Kommentar eingefügt werden:

```
/*  
    Der leichte Einstieg in Arduino & Co.  
    Projekt Alarmanlage  
    Leuchtdioden  
  
    Version 1.00, 09.05.2019  
    Der Hobbyelektroniker  
    https://community.hobbyelektroniker.ch  
    https://www.youtube.com/c/HobbyelektronikerCh  
    Der Code kann mit Quellenangabe frei verwendet werden.  
*/
```

Danach kommen die Funktionen, die gegen aussen verfügbar sein sollen:

```
void LEDs_show(int alarmStufe);  
  
void LEDs_init();    // Led's initialisieren  
int LEDs_version(); // Version (100 --> Vers. 1.00)
```

Öffentliche Namen beginnen immer mit den Namen des Moduls und einem Unterstrich. Jedes Modul enthält eine Init - Funktion und eine Versionsabfrage.

4. In LEDs.cpp den eigentlichen Code schreiben

Auch hier kann man mit einem Kommentar beginnen.
Gleich danach wird die Version definiert:

```
#define VERS 100
```

Jetzt kommen die Standardincludes.

```
#include <arduino.h>
#include "globals.h"
#include "LEDs.h"
```

arduino.h macht dem Modul die speziellen Arduino-Befehle, Datentypen und Konstanten bekannt.

globals.h können wir selber schreiben und globale Konstanten und Variablen darin unterbringen.

LEDs.h ist die zum Modul gehörende .h - Datei.

Danach folgen weitere Includes, beliebige lokale Konstante, Variablen und Funktionen. Da diese gegen aussen nicht bekannt gemacht werden, benötigen sie auch den LEDs_ - Prefix nicht.

Zum Schluss müssen noch alle in **LEDs.h** aufgeführten Funktionen geschrieben werden.
Zum Beispiel:

```
int LEDs_version() {
    return VERS;
}
```

5. Verwendung des Moduls

Im Hauptprogramm muss die .h - Datei eingebunden werden:

```
#include "LEDs.h"
```

In setup() muss das Modul initialisiert werden:

```
LEDs_init();
```

Danach stehen alle in LEDs.h aufgeführten Elemente zur Verfügung.

Vom Modul zur Klasse (Video 20)

Deklaration und Definition im .h - File

Eine Klasse wird üblicherweise im .h - File definiert. Diese Definition ist vollständig, es wird lediglich mit den Schlüsselwörtern **public** oder **private** festgelegt, ob die Eigenschaft oder Funktion von aussen zugänglich ist. Es können Funktionen, Variablen oder Konstanten deklariert werden.

```
class Led {  
    public:  
        void init(int ledPin);  
        void ein();  
        void aus();  
        void blink();  
    private:  
        int pin;  
};
```

Die 4 Funktionen **init()**, **ein()**, **aus()** und **blink()** sind von aussen zugänglich. Mit **init(pinNummer)** wird dem Objekt gesagt, auf welcher Pinnummer es arbeiten soll. Diese Nummer wird in der privaten Variablen **pin** gespeichert. **pin** kann von aussen weder abgefragt, noch gesetzt werden.

Implementation der Klasse

Den eigentlichen Programmcode schreiben wir üblicherweise im .cpp - File. Dabei wird jede einzelne Funktion hier ausprogrammiert. Dies entspricht dem Vorgehen bei den Modulen, wir setzen aber vor jeden Funktionsnamen den Namen der Klasse, gefolgt von zwei Doppelpunkten.

Jede Funktion hat so automatisch Zugriff auf alle in der Klassendefinition aufgeführten Funktionen und Variablen.

```
void Led::init(int ledPin) {  
    pin = ledPin;  
    pinMode(pin, OUTPUT);  
    aus();  
}  
  
void Led::ein() {  
    digitalWrite(pin, HIGH);  
}  
  
void Led::aus() {  
    digitalWrite(pin, LOW);  
}  
  
void Led::blink() {  
    digitalWrite(pin, !digitalRead(pin));  
}
```

Verwenden der Klasse

Die Klasse ist ja nur ein Bauplan. Um damit zu arbeiten, müssen wir daraus zuerst ein Objekt (auch Instanz genannt) erzeugen. Aus einer Klasse können beliebig viele Instanzen erzeugt werden, die dann unabhängig voneinander arbeiten können. Das .h - File muss dazu mit #include eingebunden werden.

```
Led led1; // das Objekt led1 wird gebaut  
Led led2; // das Objekt led2 wird gebaut
```

Es werden zwei Objekte (led1 und led2) erzeugt. Diese können unabhängig voneinander arbeiten. Voraussetzung ist aber, dass jedem Objekt eine individuelle Pinnummer zugewiesen wird.

```
led1.init(5); // Led 1 ist an Pin 5  
led2.init(6); // Led 2 ist an Pin 6
```

Das könnte auch im Konstruktor geschehen. Da wir aber momentan den Begriff Konstruktor nicht verwenden, rufen wir dazu die Funktion init() auf.

Jetzt können wir die Objekte verwenden:

```
led1.on(); // Led 1 einschalten  
led1.off(); // Led 1 ausschalten  
led2.blink(); // Led 2 blinken
```

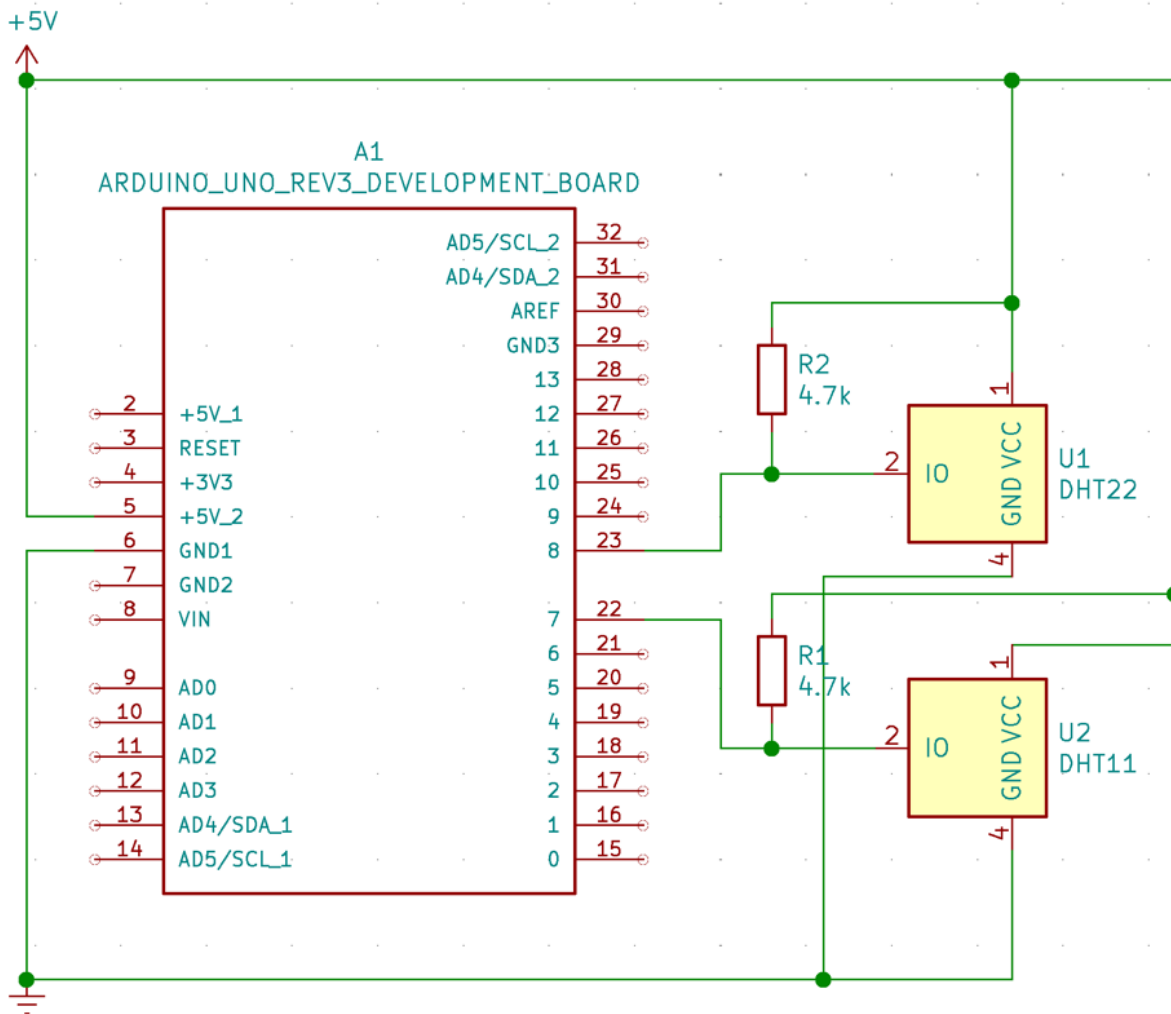
War das alles?

Für den Moment schon. Das Thema Klassen ist damit aber nicht vollständig behandelt. Da vermutlich niemand Lust hat, noch einige Lektionen mit trockener Programmiertheorie zu verbringen, lassen es wir aber dabei. Möglicherweise werde ich später doch noch auf Konstruktoren eingehen, alles Andere ist Thema für einen Fortgeschrittenen - Kurs.

Temperatur und Luftfeuchte (Video 21)

Diesmal verbinden wir Sensoren zur Messung der Luftfeuchtigkeit und der Umgebungstemperatur mit dem Arduino. Sensoren, die unter der Bezeichnung DHT 11, DHT 21, DHT 22 und AM2302 bekannt sind. Es gibt noch viele andere Sensoren, doch diese sind weit verbreitet.

Die Hardware



Hier sind 2 Sensoren gleichzeitig angeschlossen, ein DHT11 und ein DHT22 (=AM2302). Diese benötigen nur eine Datenleitung. Diese Leitung muss über einen PULL-UP - Widerstand mit + verbunden werden.

Die Software

Man muss das Rad nicht immer neu erfinden. So greifen wir auch hier auf eine fertige Bibliothek von Adafruit zu. Wie diese installiert wird, hast du ja im Video gesehen.

Das vollständige Programm findest du im Begleitmaterial unter <https://www.hobbyelektroniker.ch/resources/Lektion21.zip>

```
#include "DHT.h" // Library von Adafruit  
Die Bibliothek wird geladen
```

```
DHT dht1(8, DHT22); // weisser Sensor (AM2302) an Pin 8  
DHT dht2(7, DHT11); // blauer Sensor (DHT11) an Pin 7
```

Die Programmobjekte für die beiden Sensoren werden erzeugt. Dabei wird auch gleich angegeben, um welchen Typ es sich handelt und an welchem Pin er angeschlossen ist.

```
dht1.begin();  
dht2.begin();
```

Die beiden Sensoren werden gestartet.

```
delay(2000);  
float feuchte1 = dht1.readHumidity();  
float temp1 = dht1.readTemperature();  
if (isnan(feuchte1) || isnan(temp1)) {  
    Serial.println("Sensor 1 konnte nicht gelesen werden!");  
}
```

Die relative Luftfeuchtigkeit und die Temperatur werden abgefragt. Das darf nur alle 2 Sekunden erfolgen. Mit `isnan()` kann geprüft werden, ob kein gültiger Wert ermittelt werden konnte. So kann eine Fehlermeldung erstellt werden.

Speicheradressen und Zeiger (Video 22)

Zeiger, auch Pointer genannt, sind oft genutzte Konstrukte in der Programmiersprache C. Auch wenn du sie selbst nur selten verwendest, wirst du sie doch häufig antreffen. Aus diesem Grund solltest du wissen, um was es dabei geht. Schau also unbedingt das Video an, es erklärt dir, was du wissen musst. <https://youtu.be/cRB2T1ghvVg>

Variablen elementarer Datentypen

Was geschieht, wenn ich Variablen verschiedener einfacher Datentypen anlege?

```
// Variablen
int a = 5;
long int b = 25000000;
float c = 10.0;


// Zeiger (Pointer)
int *ptrA;
long int *ptrB;
float *ptrC;

// Zeiger (Pointer)
ptrA = &a; // 1008
ptrB = &b; // 1004
ptrC = &c; // 1000
```


// Zugriff
Serial.println(c);
Serial.println((int)ptrB);
Serial.println(*ptrA);

Ein Zeiger (auch Pointer genannt) speichert die Adresse einer Variablen. Der Zugriff auf den Wert kann über den Variablennamen oder über seinen Zeiger erfolgen.

Name	Typ	Adresse	Speicherinhalt
c	float	1000	10.0
b	long int	1004	25000000
a	int	1008	5
freier Speicher			



Der einfache Einstieg in Arduino & Co.
Speicheradressen und Zeiger



Hier nur eine Zusammenfassung der wichtigsten Punkte:

```
int a = 5;
```

Wir haben eine Variable a. Diese hat eine Adresse und diese Adresse kann in einem Zeiger gespeichert werden.

Deklaration eines Zeigers

```
int *ptrA;
```

* definiert, dass es sich um einen Zeiger handelt
int ist der Typ der Variablen, auf die der Zeiger zeigt

Adresse einer Variablen

```
ptrA = &a;
```

& ist der Adressoperator. &a gibt somit anstelle des Inhalts von a die Adresse von a zurück.

Wert der Variablen, auf die der Zeiger zeigt

```
Serial.print(*ptrA);
```

* ist der Dereferenzierungsoperator. Er sorgt dafür, dass anstelle der Adresse von a der Inhalt von a zurückgegeben wird.

Funktionen (Video 23)

Einfache Funktionen

```
int rechnung(int a, int b, int c) {  
    return a + b + c;  
}
```

Dieser Funktion müssen 3 Integer - Werte übergeben werden. Sie gibt einen Integer - Wert zurück.
Zum Beispiel: `int i = rechnung(1,2,3);`
Das ist die einfachste Art der Funktion.

In der Funktionsdeklaration gibt man also an, was übergeben wird und was man zurück erhält.
Man nennt das auch die Signatur einer Funktion. Die Signatur enthält keine Variablennamen, wäre also in diesem Fall ***int rechnung(int, int, int)***.

Es gibt die Möglichkeit mehrere Funktionen mit demselben Namen in einem Programm zu haben.
Dazu müssen sich die Signaturen unterscheiden. Allerdings genügt das nicht. Der Rückgabewert wird nicht zur Unterscheidung beigezogen.

float rechnung(int,int,int) ist dasselbe wie ***int rechnung(int,int,int)***. Die Funktionsliste müssen sich also in der Parameterliste unterscheiden.

So ist

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

eine andere Funktion, da ihr nur 2 Integer - Werte übergeben werden.
Zum Beispiel: `int i = rechnung(3,4);`

Diese Funktionsdeklaration darf nicht zusammen mit der folgenden Funktion verwendet werden:

```
int rechnung(int a, int b, int c = 10) {  
    return a + b + c;  
}
```

Dieser Funktion können 3 Werte übergeben werden. Sie funktioniert dann wie im ersten Beispiel.
Es ist aber auch möglich, nur 2 Werte zu übergeben. Dann wird für den 3. Wert einfach der angegebene Standardwert von 10 verwendet. Es können auch mehrere Parameter mit Standardwerten angegeben werden. Sie müssen aber lückenlos am Schluss der Liste stehen.

Übergabe als Wert

```
int addValues(int a, int b) {  
    int result = a + b;  
    b = 99;  
    return result;  
}
```

Aufruf:

```
int a = 5;  
int b = 8;  
println(addValues(a,b));  
println(b);
```

Hier wird der Wert übergeben. Innerhalb der Funktion haben wir eine Kopie der Variablen a und b. Aus diesem Grund verändert sich die Variable b ausserhalb der Funktion nicht, obwohl wir ihr innerhalb der Funktion den Wert 99 zuweisen.

Das gilt übrigens auch für Instanzen von Klassen. Diese werden kopiert, was unter Umständen Zeit und viel Speicherplatz beansprucht. Daher sollten solche Objekte nicht als Wert übergeben werden.

Übergabe als Pointer

```
int addPointer(int *a, int *b) {  
    int result = *a + *b;  
    *b = 99;  
    return result;  
}
```

Aufruf:

```
int a = 5;  
int b = 8;  
println(addValues(&a, &b));  
println(b);
```

Es werden Zeiger auf die entsprechenden Variablen übergeben. Der Wert wird also nicht übergeben, sondern nur die Adresse. Daher wird auch der Wert der Variablen b ausserhalb der Funktion verändert. Wir müssen aber daran denken, die Adresse mit & zu übergeben und innerhalb der Funktion die Variablen mit * anzusprechen.

Das ist etwas verwirrend. Zum Glück gibt es noch eine weitere Methode, die dasselbe bewirkt.

Übergabe als Referenz

```
int addReferenz(int &a, int &b) {  
    int result = a + b;  
    b = 99;  
    return result;  
}
```

Aufruf:

```
int a = 5;  
int b = 8;  
println(addValues(a,b));  
println(b);
```

Technisch gesehen werden hier auch Pointer übergeben. Es genügt aber, wenn du das in der Funktionsdeklaration mit & angibst. Der Aufruf und das Ansprechen der Variablen in der Funktion kann dann aber ganz normal erfolgen.

Instanzen von Klassen sollten immer als Referenzen übergeben werden, um unnötige Kopien zu vermeiden.

Spezialfall Array und Zeichenkette

```
void showValues(int intArray[], char text[]) {
    Serial.print(text);
    for (int i = 0; i < 5; i++) {
        Serial.print(", "); Serial.print(intArray[i]);
    }
    Serial.println();
    intArray[0] = 9;
    text[0] = 'h';
}
```

Aufruf:

```
int a[] = {1,2,3,4,5};
char s[] = "Hallo";
showValues(a,s);
showValues(a,s);
```

Arrays und Zeichenketten werden immer als Zeiger übergeben. Daher wirken sich Änderungen innerhalb der Funktion auch auf das Original - Array ausserhalb der Funktion aus. Ist das unerwünscht, sollte man die Parameter als **const** deklarieren:

```
void showValues(const int intArray[], const char text[]) {
```

Da hier nur die Adresse des Arrays übergeben wird, kann innerhalb der Funktion die Grösse des Arrays nicht abgefragt werden. Mit `sizeof()` erhalten wir nur die Grösse des Adresszeigers zurück. Wir wissen also nie, wieviele Elemente das Array umfasst und müssen diese Information gesondert übermitteln. Nicht notwendig ist das beim Text. Da dieser immer mit 0 endet, wissen wir, wie lange er ist.

Auch Funktionen können übergeben werden

Wir können auch die Signatur einer Funktion als Parameter angeben.

```
int calc(int f(int, int) , int a, int b) {
    return f(a,b);
}
```

Alle Funktionen mit dieser Signatur können jetzt übergeben werden. Zum Beispiel:

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

Aufruf:
`calc(plus, 2, 3);`

Lektion 24: EEPROM, das Gedächtnis des Arduinos

Das EEPROM stellt einen nicht flüchtigen Speicher von 1 KByte auf dem Arduino zur Verfügung. Es eignet sich sehr gut zum Speichern von Konfigurationsdaten.

Die Ansteuerung des EEPROMs

Wir verwenden die automatisch installierte Bibliothek EEPROM.h. Daraus verwenden wir nur zwei Befehle:

```
EEPROM.put(adresse, variable);
```

Der Inhalt der Variablen wird an die Speicheradresse **adresse** geschrieben. Wieviele Bytes geschrieben werden, hängt vom Typ der Variablen ab. Falls an der betreffenden Stelle bereits der richtige Inhalt steht, wird nicht geschrieben.

```
EEPROM.get(adresse, variable);
```

Es werden die Daten bei Speicheradresse **adresse** gelesen und in die Variable geschrieben. Wieviele Bytes gelesen werden, hängt vom Typ der Variablen ab.

Es stehen noch weitere Befehle zur Verfügung. Eine Referenz dazu ist auf der [Arduino - Webseite](#) zu finden.

Speichern der Einstellungen

Wir speichern die Helligkeit und die Pause zwischen den Blink - Zuständen.

Zuerst schreiben wir eine Signatur und eine Versionsnummer, damit wir beim Lesen feststellen können, ob die Daten gültig sind und wie sie aufgebaut sind.

```
const int eeprom_signatur = 23379;
const int eeprom_version = 2;

void eeprom_speichern() {
    int adresse = 0;
    EEPROM.put(adresse, eeprom_signatur);
    adresse += sizeof(eeprom_signatur);
    EEPROM.put(adresse, eeprom_version);
    adresse += sizeof(eeprom_version);
    EEPROM.put(adresse, pause);
    adresse += sizeof(pause);
    EEPROM.put(adresse, helligkeit);
    adresse += sizeof(helligkeit);
}
```

Laden der Einstellungen

Zuerst lesen wir die Signatur und die Version. Mit der Signatur können wir feststellen, ob die Daten zu unserem Projekt gehören. Wenn ja, dann können wir die Daten gemäss der Struktur unserer Version einlesen.

```
const int eeprom_signatur = 23379;
const int eeprom_version = 2;

void eeprom_laden() {
    int signatur;
    int version = 0;
    int adresse = 0;
    EEPROM.get(adresse, signatur);
    adresse += sizeof(signatur);
    if (signatur != eeprom_signatur) return;
    EEPROM.get(adresse, version);
    adresse += sizeof(version);
    switch(version) {
        case 1: {
            EEPROM.get(adresse, pause);
            adresse += sizeof(pause);
            break;
        };
        case 2: {
            EEPROM.get(adresse, pause);
            adresse += sizeof(pause);
            EEPROM.get(adresse, helligkeit);
            adresse += sizeof(helligkeit);
            break;
        };
    }
}
```

Der Highspeed - Arduino (Video 25)

Die bequemen Arduino - Befehle sind oft etwas langsam. Wenn die Zeit kritisch wird, helfen direkte Zugriffe auf die internen Register des Prozessors. Jeder digitale I/O - Pin entspricht einem einzelnen Bit innerhalb der internen Register des Arduino-Chips. Welchen Pin wir wo finden, können wir zum Beispiel in der [Arduino - Dokumentation](#) oder bei [Netzmafia](#) nachlesen.

Wir möchten den digitalen Pin 8 dazu bringen, ein Rechtecksignal auszugeben.
Das funktioniert einfach:

```
void loop() {  
    digitalWrite(outPin, !digitalRead(outPin));  
    delayMicroseconds(50);  
}
```

gibt 10 kHz aus.

Bei 100 kHz funktioniert das nicht mehr, da

```
digitalWrite(outPin, !digitalRead(outPin));
```

selbst etwa 5 Mikrosekunden Zeit benötigen.

Direkter Registerzugriff

Wir versuchen es daher mit einem direkten Registerzugriff. Aus der Dokumentation entnehmen wir, dass Pin 8 dem Bit 0 des Ports B entsprechen. Daher definieren wir:

```
#define outPort PORTB  
#define outBit 0
```

Wir müssen das ganze Register lesen, dann das Bit wechseln und anschliessend das ganze Register wieder zurückschreiben. In unserem Fall wäre das die Operation $PORTB = PORTB \wedge B00000001$. Die XOR - Funktion (\wedge) wechselt die Bits an den Stellen, an denen die Maske eine 1 aufweist.

gelesener Wert	Bxxxxxxx1	Bxxx0xxxx
XOR - Maske	B00000001	B0001xxxx
geschriebener Wert	Bxxxxxxx0	Bxxx1xxxx

Die Maske für den XOR - Befehl kann durch Linksschieben einer 1 erzeugt werden.

```
B00000001 << 0 ==> B00000001  
B00000001 << 4 ==> B00010000
```

So kommen wir zu folgendem Befehl:

```
port = port ^ 1 << bitNum
```

Den können wir ebenfalls mit #define festlegen:

```
#define bitToggle(port, bitNum) port = port ^ 1 << bitNum  
  
void loop() {  
    bitToggle(outPort, outBit);  
    delayMicroseconds(5);  
}
```

Damit erreichen wir eine Beschleunigung um mindestens den Faktor 20.

Von der Idee zum Projekt (Video 26)

Welche Schritte sind notwendig, um von einer Idee zum fertigen Projekt zu kommen. An einem einfachen Beispiel werden wir das durchspielen. Der Schwerpunkt liegt dabei auf der Software. Das gezeigte Vorgehen ist für Hobbyentwicklungen mit einem einzelnen Entwickler gedacht. **Das Vorgehen eignet sich nicht für professionelle Entwicklungen.**

Die Idee oder das Pflichtenheft

Zuerst müssen wir uns im Klaren sein, was wir eigentlich machen wollen. Das beginnt mit einer vagen Idee und wird dann immer exakter ausformuliert, bis im Idealfall ein vollständiges Pflichtenheft vor uns haben. Unser Beispiel ist sehr einfach, es soll ja schliesslich in einem einzigen Video realisiert werden können.

- Ein Poti gibt die Blinkrate einer gelben LED vor. Diese soll zwischen 0.5 und 5 Blinks pro Sekunde liegen.
- Die Blinkrate soll mit einem Zeiger, der an einen Servo angebracht ist, visualisiert werden.
- Wenn die Blinkrate erhöht wird, soll eine rote LED leuchten.
- Wenn die Blinkrate erniedrigt wird, soll eine grüne LED leuchten.
- Damit das einen ruhigen Eindruck macht, sollen die rote und grüne LED jeweils mindestens 1 Sekunde leuchten. Es soll aber immer nur maximal eine der beiden LEDs leuchten.

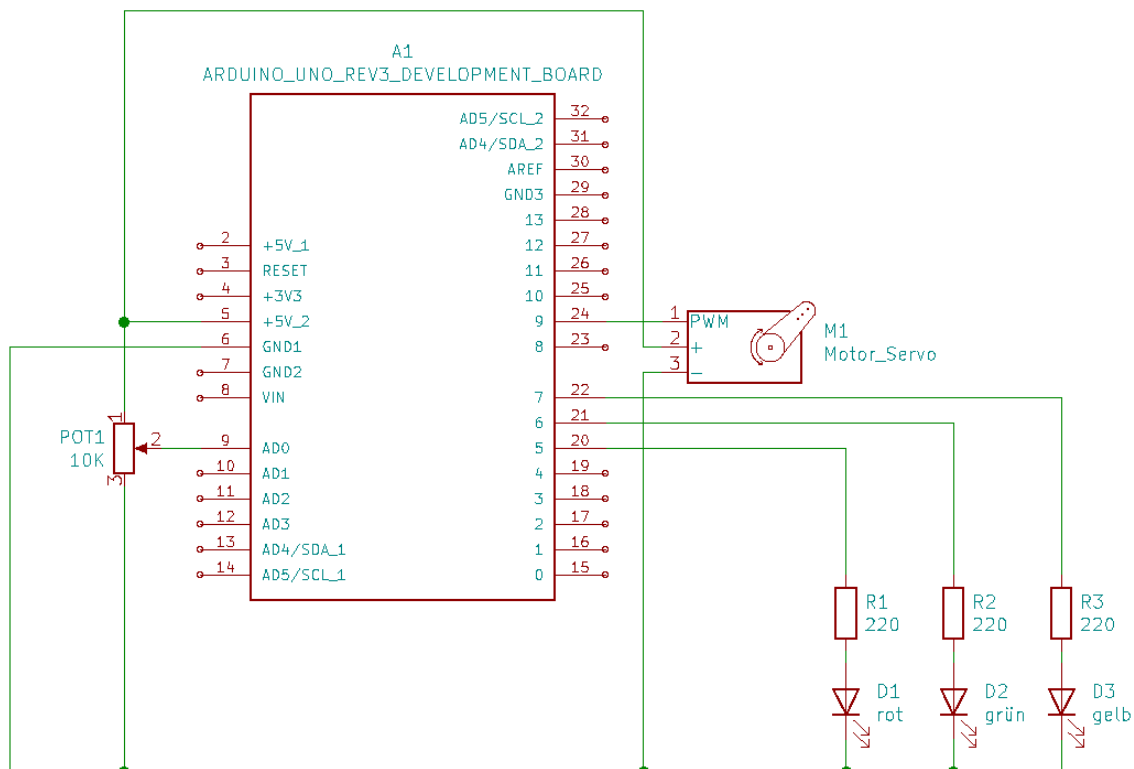
Die Hardware

Wir haben als Komponenten 1 Potentiometer, 3 LEDs und 1 Servo.

Die LEDs arbeiten alle digital, benötigen also kein PWM.

Wir verwenden einen Arduino UNO.

Wir entwerfen dieses Schema:



Das Grundgerüst der Software

Im ersten Schritt geht es nicht um Funktionalität. Wir definieren lediglich an welchen Pins die Komponenten angeschlossen sind, welche Bibliotheken verwendet werden und welche Pins initialisiert werden müssen. Ausserdem definieren wir einige wichtige Konstanten.

Serial.begin() setzen wir ein, da wir vermutlich später noch Debug-Ausgaben machen werden.

In **IdeeZuProjekt1.ino**:

```
#include <Servo.h>
Servo servo;

// Pin - Definitionen
const int servoPin = 9;
const int gelbPin = 7;
const int gruenPin = 6;
const int rotPin = 5;

const int potiPin = A0;

// Konstanten
const float minBlink = 0.5;    // Frequenz
const float maxBlink = 5.0;    // Frequenz
const int anzeigedauer = 1000; // in ms

void setup() {
  Serial.begin(9600);
  servo.attach(servoPin);
  pinMode(rotPin, OUTPUT);
  pinMode(gelbPin, OUTPUT);
  pinMode(gruenPin, OUTPUT);
}
```

Servo und Potentiometer

Jetzt lesen wir das Potentiometer ein, berechnen die Blinkfrequenz und zeigen diese mit dem Potentiometer an. Dazu schreiben wir zwei Funktionen und testen diese.

In **IdeeZuProjekt2.ino**:

```
float getBlinkFrequenz() {
  int potiValue = analogRead(potiPin); // 0 .. 1023
  return map(potiValue, 0, 1023, long(minBlink*10), long(maxBlink*10)) / 10.0;
}
```

Das Potentiometer wird mit analogRead() eingelesen und mit Hilfe von map() in eine Frequenz umgewandelt. map() arbeitet nur mit ganzzahligen Werten. Damit wir auch eine Frequenz von 0.5 Hz erhalten können, multiplizieren wir die Grenzwerte zuerst mit 10 und dividieren vor der Rückgabe wieder.

```
void zeigeMitServo(float frequenz) {
  int servoValue = map(long(frequenz*10), 0, long(maxBlink*10), 0, 180);
  servo.write(servoValue);
  delay(10); // Dem Servo etwas Zeit geben
}
```

Auch hier müssen wir beachten, dass map() nur mit ganzzahligen Werten arbeitet. Der Servo erhält noch 10ms Zeit, die neue Position anzufahren.

Zusammenfassung Programmierung

```
void loop() {  
    float blinkFrequenz = getBlinkFrequenz();  
    zeigeMitServo(blinkFrequenz);  
}
```

Mit diesem Code testen wir unser bisheriges Programm.

Blinken der gelben Led und weitere Tests

Jetzt soll die gelbe Led mit der eingelesenen Blinkfrequenz blinken.

In **IdeeZuProjekt3.ino**:

```
void blinkeGelb(float frequenz) {  
    static unsigned long start = 0;  
    unsigned long zeit = 1000 / frequenz / 2; // Pause in ms  
    if (millis() >= start + zeit) {  
        start = millis();  
        digitalWrite(gelbPin, !digitalRead(gelbPin));  
    }  
}
```

Das ist ganz normaler Blinkcode, wie wir ihn schon oft gesehen haben. Wir arbeiten mit millis() um das Einlesen des Potis nicht zu behindern.

Die Funktion muss jetzt noch in loop() aufgerufen werden.

```
void loop() {  
    float blinkFrequenz = getBlinkFrequenz();  
    zeigeMitServo(blinkFrequenz);  
    blinkeGelb(blinkFrequenz);  
}
```

Wird die Frequenz erhöht oder vermindert?

Jetzt soll festgestellt werden in welche Richtung die Frequenz verändert wird. Das Ergebnis wird für den ersten Test auf den seriellen Monitor ausgegeben.

In **IdeeZuProjekt4.ino**:

```
int getRichtung(float frequenz) {  
    static float f = 0.0;  
    if (frequenz >= 1.1*f) {  
        f = frequenz;  
        return 1; // schneller  
    } else if (frequenz <= 0.9 * f) {  
        f = frequenz;  
        return -1; // langsamer  
    } else {  
        return 0; // keine Änderung  
    }  
}
```

Wenn die Frequenz vermindert wird, erhalten wir -1, bei einer Erhöhung 1. Keine Veränderung wird mit dem Rückgabewert 0 signalisiert. Eine Änderung wird erst signalisiert, wenn die Veränderung etwa 10 % beträgt.

```
void loop() {
    float blinkFrequenz = getBlinkFrequenz();
    zeigeMitServo(blinkFrequenz);
    blinkeGelb(blinkFrequenz);
    int richtung = getRichtung(blinkFrequenz);
    if (richtung == 1) {
        Serial.println("schneller");
    } else if (richtung == -1) {
        Serial.println("langsamer");
    }
}
```

Anzeigen der Richtung mit den LEDs

Jetzt verwenden wir die Richtungsinformation um die rote und grüne Led anzusteuern. Der Debug-Code mit Serial. kann jetzt wieder entfernt werden.

In **IdeeZuProjekt5.ino**:

```
void roteLed(int r) {
    static unsigned long start = 0;
    if (r == 1) { // schneller
        start = millis();
        digitalWrite(rotPin, HIGH);
    } else if (r == -1) { // langsamer
        digitalWrite(rotPin, LOW);
    } else {
        if (millis() > start + anzeigedauer) {
            digitalWrite(rotPin, LOW);
        }
    }
}
```

Auch hier wird das Timing mit millis() gemacht, um die Potiabfrage nicht zu beeinträchtigen.

Eine ähnlich Funktion benötigen wir für die grüne Led:

```
void grueneLed(int r) {
    static unsigned long start = 0;
    if (r == -1) { // langsamer
        start = millis();
        digitalWrite(gruenPin, HIGH);
    } else if (r == 1) { // schneller
        digitalWrite(gruenPin, LOW);
    } else {
        if (millis() > start + anzeigedauer) {
            digitalWrite(gruenPin, LOW);
        }
    }
}
```

Zusammenfassung Programmierung

Jetzt müssen wir das noch in loop() aufrufen:

```
void loop() {  
    float blinkFrequenz = getBlinkFrequenz();  
    zeigeMitServo(blinkFrequenz);  
    blinkeGelb(blinkFrequenz);  
    int richtung = getRichtung(blinkFrequenz);  
    roteLed(richtung); // schneller  
    grueneLed(richtung); // langsamer  
}
```

Unser Projekt wäre damit realisiert. Jetzt kann man sich bereits Gedanken über die Version 2 machen.

Zusammenfassung

1. Idee in einem Pflichtenheft formulieren.
2. Komponenten auswählen und Verdrahtung festlegen. Idealerweise wird ein Schema gezeichnet.
3. Programmgerüst erstellen und Pinbelegungen und andere Konstanten eintragen.
4. Problem in verschiedene Teile zerlegen. Die Teile sollten möglichst voneinander unabhängig sein. Sie bekommen ihre Werte beim Aufruf und geben unter Umständen ein Resultat zurück. Globale Variablen sollten vermieden werden.
5. Einen Teil nach dem anderen realisieren. Jeder Teil wird getestet.
6. Wenn alle Teile funktionieren, das Ganze beurteilen und nach Verbesserungsmöglichkeiten suchen.

Strings (Zeichenketten) 1 (Video 27)

Die Verarbeitung von Text wird regelmässig gebraucht. Jede Programmiersprache bietet hierzu Funktionen und Datentypen an. Technisch gesehen ist ein Text nur eine Aneinanderreihung von Zeichen. Daher wird eine Zeichenkette in C als ein Array von Zeichen behandelt. Allerdings sind hier einige Spezialitäten zu beachten.

Wir müssen wissen, wann der Text zu Ende ist. Dafür gibt es im Prinzip zwei Lösungen. In einigen Programmiersprachen wird zusätzlich die Länge gespeichert. In C wurde eine andere Lösung gewählt: der String ist beendet, wenn der Wert 0 (nicht das Zeichen '0!') gefunden wird.

So wird "Hallo" so dargestellt:

Als Zeichen:	H	a	l	l	o	\0
Als Wert:	72	97	108	108	111	0

Das Wort **Hallo** hat 5 Buchstaben, belegt aber im Speicher 6 Bytes.

Erstellen eines Strings

```
char zeichen = 'A';
```

Das ist kein String, sondern nur ein einzelnes Zeichen. Daher wird 'A' verwendet und nicht "A".

```
char hallo[] = "Hallo";
```

Das ist ein normaler C-String. Er benötigt 6 Byte, 5 für den String und 1 für die abschliessende 0.

```
char hallo[6] = "Hallo";
```

Dieser String entspricht exakt dem aus dem vorhergehenden Beispiel.

```
char hallo[12] = "Hallo";
```

Das ist ein normaler C-String. Er belegt 12 Bytes, 5 für den String, 1 für die abschliessende 0 und 6 Reservebytes.

```
char hallo[5] = "Hallo";
```

Dieser String ist fehlerhaft. Der Text belegt alle 5 Bytes, so dass für die abschliessende 0 kein Platz mehr bleibt. Der Compiler zeigt eine Warnung, führt den Code aber trotzdem aus. Normalerweise treten dabei unerwünschte Effekte auf.

```
String hallo = "Hallo";
```

Das ist ein Stringobjekt. Dieses ist sehr bequem und weniger heikel, braucht aber mehr Speicherplatz. Dieses Objekt wird in der nächsten Lektion ausführlich behandelt.

Es werden im Folgenden nur die wichtigsten Operationen mit C-Strings besprochen. Sie stammen aus den Standardbibliotheken von C. Allerdings muss beachtet werden, dass in der Arduino - Entwicklungsumgebung nicht alle dort vorhandenen Funktionen unterstützt.

Umwandlung von Zahlen in Strings

Alle diese Operationen benötigen einen Buffer, der zuerst reserviert werden muss. Ist dieser zu klein gewählt, funktioniert das Programm nicht richtig oder stürzt sogar ab.

Hier einige Beispiele:

```
// Buffer erstellen (wird immer benötigt)
char buf[50]; // Hier haben maximal 49 Zeichen Platz
```

```
// Eine Integer-Zahl in einen String kopieren
itoa(i, buf, 10);
```

Die Integer-Zahl *i* wird in einen String umgewandelt und in **buf** gespeichert. Mit **10** wird angegeben, dass die Ausgabe im Dezimalsystem erfolgen soll.

```
// Eine Float-Zahl in einen String kopieren
dtostrf(f, 5, 3, buf);
```

Die Float-Zahl *f* wird in einen String umgewandelt und in **buf** gespeichert. Mit **5** wird angegeben, wie lang der String sein soll. Hier kann auch 0 angegeben werden, dann wird die notwendige Länge automatisch festgelegt. Mit **3** werden die Anzahl Nachkommastellen festgelegt.

```
// Eine Integer-Zahl in einen String einbetten
sprintf(buf, "Die Zahl %d ist vom Typ integer", i);
```

Die Zahl *i* wird in einen String umgewandelt und dieser anstelle des Platzhalters **%d** eingefügt. Der resultierende String wird dann in **buf** gespeichert.

%d kann noch mit Formatanweisungen ergänzt werden:

```
int i = 5;
%d    ==> "5"
%3d   ==> " 5"
%03d  ==> "005"
```

```
// Einen String in einen anderen String einbetten
sprintf(buf, "Mein Name ist %s", "Max");
```

Der String **"Max"** wird anstelle des Platzhalters **%s** eingefügt. Der resultierende String wird dann in **buf** gespeichert.

```
// Eine Float-Zahl in einen anderen String einbetten
// sprintf(buf, "Die Zahl %f ist vom Typ float", f); // Das geht nicht!!!
```

Es muss ein Umweg gemacht werden. Die Float-Zahl wird zuerst in einen String umgewandelt und dieser dann in den ersten String eingebettet. Dazu wird ein zusätzlicher Buffer benötigt.

```
char fl[20];
dtostrf(f, 5, 3, fl);
sprintf(buf, "Mein Name ist %s", "fl");
```

Zwei Strings zusammenfügen

Zwei C - Strings lassen sich nicht mit + zusammenfügen. Dazu muss die Funktion **strcat** verwendet werden.

```
char text1[] = "Das ist Teil 1.";
char text2[] = "Das ist Teil 2.";
strcat(text1, text2); // Das führt zu Problemen !!!
```

```
char text1[35] = "Das ist Teil 1.";
char text2[] = "Das ist Teil 2.";
strcat(text1, text2); // Das funktioniert !!!
```

Das Resultat ist dann in **text1** zu finden. Das ist auch der Grund, weshalb die erste Variante nicht funktioniert. **text1** ist dort zu klein und hat keinen Platz für die zusätzlichen Zeichen. Darum muss **text1** gross genug erzeugt werden. Die 35 Bytes der zweiten Version sollten genügen.

Eine zweite Möglichkeit ist die Verwendung von **sprintf**. Auch hier muss ein Buffer in ausreichender Grösse für das Resultat reserviert werden.

```
char text1[] = "Das ist Teil 1. %s"; // Hier den Platzhalter einsetzen
char text2[] = "Das ist Teil 2.";
char resultat[35];
sprintf(resultat, text1, text2);
```

Auch diese Möglichkeiten stehen zur Verfügung:

```
sprintf(resultat, "Das ist Teil 1. %s", "Das ist Teil 2.");
sprintf(resultat, "Das ist Teil 1. %s", text2);
sprintf(resultat, text1, "Das ist Teil 2.");
```

Die String - Klasse (Video 28)

Die C-Strings sind sehr effizient, haben aber den Nachteil, dass sie relativ umständlich zu verwenden sind und immer die Gefahr von gefährlichen Fehlern besteht.

Die Klasse String verwaltet seinen Speicher selbstständig und bietet ein umfangreiches Set an Funktionen an. Eine Dokumentation der String - Klasse findest du auf arduino.cc.

Die Klasse kann mehr Speicher verbrauchen als erwartet. Ein Effekt, der sich Heap - Fragmentierung nennt, kann zu unerwünschten Ergebnissen führen. Daher sollten Instanzen der Klasse String nie als globale Variablen angelegt werden.

Erstellen eines Strings

Beinahe jeder Datentyp lässt sich direkt in einen String umwandeln.

```
String text = "Hallo";

String zeichen = String('a');

String ganzeZahl = String(18);
String ganzeZahl = 18;
String dezimalZahl = String(18, DEC);
String hexZahl = String(18, HEX);
String binZahl = String(18, BIN);
String binZahl = String(18, OCT);
String binZahl = String(18, 8); // ebenfalls Oktal
String binZahl = String(18, 17); // beliebige Basis

String floatZahl1 = String(18.5); // Default: 2 Nachkommastellen
String floatZahl2 = String(18.5, 3); // Anzahl Nachkommastellen
```

Strings in Zahlen umwandeln

```
String s = String("125.25");
int i = s.toInt(); // ==> 125
float f = s.toFloat(); // ==> 125.25
double d = s.toDouble(); // ==> 125.25

String s = "keine Zahl";
int i = s.toInt(); ==> 0
```

Strings zusammenfügen

```
String text1 = "Anfang ";
String text2 = "Ende";
text1.concat(text2);
print(text1) ==> "Anfang Ende"
```

```
String text1 = "Anfang ";
String text2 = "Ende";
String text3 = text1 + text2;
print(text3) ==> "Anfang Ende"
```

```
String text1 = "Anfang ";
String text2 = "Ende";
text1 += text2;
print(text1) ==> "Anfang Ende"
```

Das geht auch mit anderen Datentypen:

```
String text1 = "Zahl ";
text1 += 5;
print(text1) ==> "Zahl 5"
```

Strings vergleichen

```
text1 == text2;
text1.equals(text2);
```

Gibt true zurück, wenn text1 und text2 identisch sind.

```
text1.equalsIgnoreCase(text2);
```

Bei diesem Vergleich werden Gross- und Kleinbuchstaben als identisch angesehen.

```
text1.startsWith("Hallo");
text1.endsWith("Welt");
```

Gibt true zurück, wenn text1 mit dem zweiten String beginnt, beziehungsweise endet. Gross- und Kleinbuchstaben werden unterschieden.

```
text1 != text2;
text1 < text2; // auch mit >, <=, >=
```

Test aus Ungleichheit, beziehungsweise grössere oder kleiner.

```
text1.compareTo(text2);
```

Gibt 0 zurück, wenn beide Strings identisch sind.
Bei einem negativen Resultat ist text1 kleiner als text2.
Bei einem positiven Resultat ist text1 grösser als text2.

Indexierung

Auf einzelne Zeichen eines Strings kann wie bei C-Strings über den Index zugegriffen werden.

```
String s = "Hallo";  
s[1] oder s.charAt(1) geben das Zeichen a zurück.
```

```
s[1] = 'A' und s.setCharAt(1, 'A') erlauben, ein Zeichen zu ändern.
```

```
s.length() gibt die Länge des Strings zurück.
```

```
s.substring(1) gibt den String ab Index 1 zurück ("allo")
```

```
s.substring(1, 3) gibt den String ab Index 1 bis Index 2 zurück ("al")
```

```
s = "Die String - Objekte benötigen mehr Speicher als C-Strings";
```

```
s.indexOf("String"); // gibt das erste Vorkommen zurück (4)
```

```
s.lastIndexOf("String"); // gibt das letzte Vorkommen zurück (52)
```

```
s.indexOf("String", 5); // gibt das erste Vorkommen nach Index 5 zurück (52)
```

Wenn der gesuchte Begriff nicht vorhanden ist, wird -1 zurückgegeben.

```
s = "Die String - Objekte";
```

```
s.remove(3); // Alles nach Index 3 wird entfernt ("Die")
```

```
s = "Die String - Objekte";
```

```
s.remove(4, 9); // 9 Zeichen nach Index 4 werden entfernt ("Die Objekte")
```

Weitere Funktionen

```
String s = " Hallo, das ist ein Text ";
```

```
s.trim(); entfernt vorangehende und nachfolgende Leerzeichen.
```

```
s.replace("Text", "String"); ersetzt 'Text' durch 'String'
```

```
toLowerCase(); wandelt in Kleinbuchstaben um
```

```
toUpperCase(); wandelt in Grossbuchstaben um
```

```
c_str(); gibt einen Zeiger auf den C-String zurück
```

```
getBytes(); kopiert die Zeichen in einen Buffer
```

```
toCharArray(); wie getBytes()
```

```
reserve(100); reserviert 100 Bytes für Stringmanipulationen
```

... und wo bleibt das & Co.? (Video 29)

In diesem Video geht es um die Einbindung eines ESP32 - Boards in die Arduino IDE. Schau dir das Video und experimentiere selbst mit einem ESP32 - Board!

Der Präprozessor (Video 30)

Der Präprozessor (engl. preprocessor) ist ein fester Bestandteil aller C-Compiler. Er verarbeitet Makro - Befehle und erzeugt dabei den eigentlichen Quelltext, der dann übersetzt wird. Diese Befehle sind sehr leistungsfähig und werden daher auch oft eingesetzt.

#define

Mit **#define** wird ein Makro definiert.

```
#define LED 13
```

erzeugt ein Makro mit dem Namen **LED**. Dieses hat den Wert **13**. Dabei handelt es sich nicht um eine Variable oder Konstante im Sinne der Programmiersprache. Es handelt sich um eine einfache Textersetzung. Überall im Quelltext soll der String **LED** durch den String **13** ersetzt werden. Diese Ersetzung erfolgt im ausführbaren Quelltext, nicht aber in den Kommentaren oder innerhalb von Strings.

Makros können auch leer sein:

```
#define DEBUG
```

Das Makro **DEBUG** existiert jetzt, hat aber keinen Wert.

Es ist üblich, Makros vollständig in Grossbuchstaben zu schreiben. Das ist aber reine Konvention, keine Bedingung.

In der Arduino - Umgebung existieren bereits viele fest vorgegebene Makros. Beispiele sind OUTPUT, INPUT_PULLUP, HIGH, LOW.

Es können auch komplexere Makros definiert werden:

```
#define SET_OUTPUT(pin) pinMode(pin, OUTPUT)
```

Hier wird ein Codesegment definiert. Dabei können auch Argumente übergeben werden.

Selbst mehrzeilige Makros sind möglich:

```
#define BLINK(pin, zeit) \
    digitalWrite(pin, HIGH); \
    delay(zeit); \
    digitalWrite(pin, LOW); \
    delay(zeit)
```

Dabei wird mit \ angezeigt, dass das Makro in der nächsten Zeile weitergeführt wird.

#ifdef, #ifndef

Es kann jederzeit abgefragt werden, ob ein Makro existiert.

```
#ifdef DEBUG
    Serial.println("DEBUG existiert")
#else
    Serial.println("DEBUG existiert nicht")
#endif

#ifndef DEBUG
    Serial.println("DEBUG existiert nicht")
#else
    Serial.println("DEBUG existiert")
#endif
```

#if

Auch die Werte können abgefragt werden.

```
#if DEBUG > 100000
    SERIAL_PRINT(String(DEBUG) + " Baud ist schnell!");
#endif
```

#include

Dieser Befehl verfolgt einen ganz anderen Zweck als die bisher besprochenen. Um den Code übersichtlich zu halten, wird er oft in mehrere Dateien aufgeteilt. Man kann das der Arduino-IDE überlassen und mit mehreren .ino - Dateien arbeiten. Dies ist aber oft etwas intransparent und es ist oft unklar, wie die Sichtbarkeiten der Programmteile festgelegt werden.

Klarer ist es bei der Verwendung von .h - Dateien.

Der Text in der .h - Datei wird exakt an der Stelle in den Code eingefügt, an der das #include steht.

Wenn dieselbe Datei an mehreren Stellen im Projekt importiert wird, gibt es eine Fehlermeldung. Das lässt sich mit

```
#ifndef EinName_H
#define EinName_H
    // Der Programmtext
#endif
```

oder mit

```
#pragma once
```

vermeiden.

Weitere Befehle

Eine Liste aller Befehle findest du in der gcc - Dokumentation.

<https://gcc.gnu.org/onlinedocs/cpp/Index-of-Directives.html#Index-of-Directives>

#assert:		Obsolete Features
#define:		Object-like Macros
#elif:		Elif
#else:		Else
#endif:		Ifdef
#error:		Diagnostics
#ident:		Other Directives
#if:		Conditional Syntax
#ifdef:		Ifdef
#ifndef:		Ifdef
#import:		Alternatives to Wrapper #ifndef
#include:		Include Syntax
#include_next:		Wrapper Headers
#line:		Line Control
#pragma GCC dependency:		Pragmas
#pragma GCC error:		Pragmas
#pragma GCC poison:		Pragmas
#pragma GCC system_header:		System Headers
#pragma GCC system_header:		Pragmas
#pragma GCC warning:		Pragmas
#pragma once:		Pragmas
#sccs:		Other Directives
#unassert:		Obsolete Features
#undef:		Undefining and Redefining Macros
#warning:		Diagnostics

Klassen und Objekte (Video 30 bis 34)

Eine einfache Klasse

Dieses Kapitel erstreckt sich über mehrere Lektionen. Klassen wurden in diesem Kurs und auch in anderen Videos schon mehrmals angeschnitten. Diesmal werden wir aber einen genaueren Blick darauf werfen. Du bist nach Abschluss dieser Lektionen sicher noch kein OOP - Experte (OOP = Objektorientierte Programmierung), du solltest dann aber in der Lage sein, eigene Klassen zu entwerfen, zu implementieren und zu verwenden.

In der Lektion 30 behandeln wir nur Grundlagen. Möglicherweise erschliesst sich dir hier der Sinn von Klassen noch nicht. Das wird sich aber ändern, wenn du in späteren Lektionen unerwartete Möglichkeiten entdecken wirst. Bereits in Lektion 31 verwenden wir Vererbung, um eine leistungsfähigere Klasse zu erstellen.

Schaue dir das Video und das Dokument **Klassen1.pfd** an. Hier siehst du ein Beispiel einer Klasse. Das Beispiel ist sehr einfach gehalten, wir simulieren die Funktion einer Led. Du wirst den Eindruck gewinnen, dass hier ein einfaches Problem sehr kompliziert gelöst wird. In Lektion 31 kannst du dann aber erahnen, dass mit einem solchen Klassenkonstrukt ein sehr mächtiges Werkzeug zur Verfügung steht.

Fürs Erste konzentrieren wir uns auf den Aufbau einer Klasse.

Eine Klassendefinition beginnt immer mit dem Schlüsselwort **class** und dem Namen der Klasse. Danach folgt ein Block, abgeschlossen mit einem Semikolon. Achtung: hier findest du nur Codeschnipsel, für den vollständigen Code solltest du die Dateien in den Begleitunterlagen verwenden.

```
class LedFunktionen {  
    ...  
};
```

Eine Klasse kann Funktionen und Eigenschaften gegen aussen zur Verfügung stellen oder privat halten.

```
class LedFunktionen {  
    public:  
  
        void einschalten() {  
            ...  
        }  
  
        void ausschalten() {  
            ...  
        }  
  
    private:  
        int _status;  
};
```

So können die Funktionen **einschalten()** und **ausschalten()** von aussen verwendet werden, die Variable **_status** ist aber nicht zugänglich. Das wird durch die Bereiche **public:** und **private:** gesteuert.

Zur Verwendung der Klasse müssen wir Instanzen erstellen. Das sind Objekte, die an Variablen gebunden sind, über die wir ihre Funktionen aufrufen können.

```
LedFunktionen led1;
```

Zusammenfassung Programmierung

```
LedFunktionen led2;
```

erzeugt zwei Instanzen, **led1** und **led2**.
Diese können wir dann verwenden.

led1.einschalten() schaltet die Led 1 ein und **led2.ausschalten()** schaltet die Led 2 aus.

Eine neue Klasse, die alle Eigenschaften der Basisklasse erbt

Bisher haben wir nur simuliert. Jetzt wollen wir aber eine echte Led leuchten lassen.
Da wir den Code für das Ein- und Ausschalten schon geschrieben haben, wollen wir nur noch den Code für die Ansteuerung der Led hinzufügen.

Dazu verwenden wir das Konzept der **Vererbung**.

Unsere neue Klasse heisst **DigitalLed**. Sie steuert eine digitale Led an. Für diese Klasse wollen wir alle Fähigkeiten der Klasse **LedFunktionen** übernehmen.

Daher deklarieren wir **DigitalLed** so:

```
class DigitalLed : public LedFunktionen {  
};
```

Die neue Klasse heisst **DigitalLed**, sie erbt alle Funktionen und Eigenschaften von **LedFunktionen**. Mit **public** bestimmen wir, dass alle Methoden von **LedFunktionen** dem Anwender von **DigitalLed** zur Verfügung stehen.

DigitalLed kann also eine Led simulieren und kennt die Befehle **einschalten()** und **ausschalten()**.

Um eine echte Led anzusteuern brauchen wir einen **Pin**. Diesen geben wir beim Erstellen der Instanz mit.

```
DigitalLed led1(2);
```

Erstellt eine digitale Led an Pin 2.

Damit das funktioniert, benötigen wir einen **Konstruktor**. Dieser wird automatisch aufgerufen, wenn wir die Instanz erzeugen.

```
DigitalLed(int pin) : LedFunktionen() {  
    _pin = pin;  
    pinMode(pin, OUTPUT);  
}
```

Wir definieren, dass der **Konstruktor** eine Pinnummer entgegennimmt. Daher können wir **led1(2)** schreiben. Diese Nummer wird in einer privaten Variable gespeichert und der Pin wird mit **pinMode** vorbereitet.

Ein **Konstruktor** einer abgeleiteten Klasse sollte immer den **Konstruktor** der darunterliegenden Klasse aufrufen. Das machen wir mit

```
DigitalLed(int pin) : LedFunktionen()
```

In unserem Beispiel ist der **Konstruktor** von **LedFunktionen** ein leerer **Standardkonstruktor**. Dieser ist immer vorhanden, auch wenn wir ihn in der Klasse nicht explizit ausprogrammieren.

Zusammenfassung Programmierung

Die eigentliche Arbeit geschieht in **setStatus()**. In der Basisklasse setzen wir einfach den Wert der privaten Variablen **_status**. Das müssen wir jetzt erweitern. Dazu schreiben wir in unserer neuen Klasse diese Funktion neu.

```
void setStatus(int status) {
    LedFunktionen::setStatus(status);
    switch (getStatus()) {
        case eingeschaltet: digitalWrite(_pin,HIGH);
                               break;
        case ausgeschaltet: digitalWrite(_pin,LOW);
                               break;
    }
}
```

Dabei werden die Programmschritte aus **LedFunktionen** nicht nochmals geschrieben. Wir rufen einfach **setStatus** aus der darunterliegenden Klasse auf.

```
LedFunktionen::setStatus(status);
```

Danach müssen wir die Funktion nur noch um den neuen Code erweitern.

Damit **setStatus** aus der neuen Klasse verwendet wird, obwohl wir eine Funktion der darunterliegenden Klasse aufrufen (**einschalten()**), müssen wir in **LedFunktionen** die Funktion **setStatus()** als **virtuell** deklarieren.

```
virtual void setStatus(int status) {
```

Den vollständigen Code findest du wie immer in den Begleitunterlagen.

Wieso schreiben wir

```
switch (getStatus()) und nicht switch (_status)?
```

Auf die Variable **_status** haben wir aus der Klasse **DigitalLed** keinen Zugriff, da diese dort als **privat** deklariert ist. Daher verwenden wir die Funktion **getStatus()**, die als öffentlich (**public**) deklariert ist. Eine solche Funktion nennt man einen **Getter**.

setStatus() ist ein **Setter**. Er setzt eine Variable, kann dabei aber noch andere Aufgaben wahrnehmen.

public und **private** haben wir schon kennengelernt. Zusätzlich gibt es noch **protected**. Hier die Unterschiede der drei Sichtbarkeiten:

public	In allen Klassen und dem Sketch sichtbar.
protected	In der eigenen Klasse und allen abgeleiteten Klassen sichtbar.
private	Nur in der eigenen Klasse sichtbar.

Noch eine Klasse

Weil es so gut geklappt hat, wagen wir uns nochmals an eine neue Klasse. Diesmal geht es um eine analoge Led, bei der die Helligkeit mit PWM gesteuert werden kann.

Unsere neue Klasse heisst **AnalogLed**. Sie steuert eine analoge Led an. Für diese Klasse wollen wir ebenfalls alle Fähigkeiten der Klasse **LedFunktionen** übernehmen.

Daher deklarieren wir **AnalogLed** so:

```
class AnalogLed : public LedFunktionen {  
};
```

Auch um eine analoge Led anzusteuern, brauchen wir einen **Pin**. Diesen geben wir beim Erstellen der Instanz mit.

```
AnalogLed led1(3);
```

Der Benutzer muss darauf achten einen PWM - fähigen Pin anzugeben.

Auch hier erstellen wir einen **Konstruktor**. PinMode brauchen wir hier nicht, dafür setzen wir die Helligkeit auf das Maximum. **_helligkeit** muss als private Variable angelegt werden.

```
AnalogLed(int pin) : LedFunktionen() {  
    _pin = pin;  
    _helligkeit = 255;  
}
```

Da **_helligkeit** eine private Variable ist, benötigen wir einen Setter und einen Getter.

```
void setHelligkeit(int helligkeit) {  
    _helligkeit = constrain(helligkeit, 0, 255);  
    setStatus(getStatus());  
}  
  
int getHelligkeit() {  
    return _helligkeit;  
}
```

Die eigentliche Arbeit geschieht auch hier in **setStatus()**.

```
void setStatus(int status) {  
    LedFunktionen::setStatus(status);  
    switch (getStatus()) {  
        case eingeschaltet: analogWrite(_pin, _helligkeit);  
                             break;  
        case ausgeschaltet: analogWrite(_pin, 0);  
                             break;  
    }  
}
```


Instanzvariablen und Klassenvariablen

Das Schlüsselwort **static** haben wir ja schon bei den Konstanten kennengelernt. Damit kann auf den Wert zugegriffen werden, ohne dass eine Instanz der Klasse existiert.

Richtig interessant wird es aber erst, wenn wir Klassenvariablen erzeugen. Diese Variablen können unabhängig von allfälligen Instanzen manipuliert werden. Sie gehören aber der Klasse, das heisst auch, dass es pro Klasse nur einen einzigen Wert gibt.

Hier muss eine syntaktische Besonderheit beachtet werden. Die Deklaration der statischen Variablen erzeugt diese nicht, das muss noch ausserhalb der Klassendefinition durchgeführt werden. Dort kann dann auch eine Initialisierung auf einen bestimmten Wert vorgenommen werden.

```
class Irgendwas {
    public
        static int staticVar;
};

int Irgendwas::staticVar = 5;
```

Erst jetzt kann sie benutzt werden.

Ein bekanntes Beispiel für den Einsatz einer Klassenvariablen ist das Erstellen eines Instanzzählers. Das ist im Video im Detail erklärt. Schau dir auch die Datei **Klassen3.pdf** an. Diese findest du zusammen mit dem Quellcode in den Begleitunterlagen.

Die Idee ist einfach. Es wird eine statische Variable angelegt (**anzahl**). Diese wird im **Konstruktor** jeweils um 1 hochgezählt. Im **Destruktor** wird die Anzahl wieder dekrementiert.

```
class Irgendwas {
    public:
        Irgendwas() {
            anzahl++;
        }

        ~Irgendwas() {
            anzahl--;
        }

        static int anzahl;
};

int Irgendwas::anzahl = 0;
```

Was ist eigentlich ein **Destruktor**?

Das Gegenteil vom **Konstruktor**. Der **Destruktor** wird automatisch ausgeführt, wenn ein Objekt wieder entfernt wird.

Vererbung aus der Basisklasse

Jetzt beschäftigen wir uns wieder mit der Basisklasse **LedFunktionen** und simulieren damit eine einfache Blinkschaltung. Diese ist aber unschön, da wir mit **delay()** arbeiten. Methoden, wie wir das vermeiden können, kennen wir genügend. Anstatt das mit Hilfe einer Funktion und einer statischen Variablen zu realisieren, bauen wir es doch gleich in die Klasse ein.

Was müssen wir unserer Klasse mitteilen?

Da ist einmal die gewünschte Blinkzeit, die legen wir in einer privaten Variablen **_blinkzeit** ab. Ausserdem benötigen wir eine Funktion, die wir regelmässig aufrufen. Diese nennen wir **blink()**. Sie entscheidet bei jedem Aufruf, ob es notwendig ist, den Zustand der Led zu wechseln. Um das zu entscheiden, müssen wir die Zeit des letzten Wechsels speichern. Dazu führen wir die Variable **_oldTime** ein.

Für **_blinkzeit** benötigen wir noch einen **Setter**, ein **Getter** ist nicht unbedingt notwendig.

Dazu erweitern wir unsere Klasse um diesen Code:

```
class LedFunktionen {
public:
    void setBlinkZeit(int zeit) {
        _blinkZeit = zeit;
    }

    void blink() {
        if (millis() > _oldTime + _blinkZeit) {
            _oldTime = millis();
            if (_status == eingeschaltet) {
                ausschalten();
            } else {
                einschalten();
            }
        }
    }

private:
    int _blinkZeit;
    unsigned long _oldTime = 0;
}
```

Der Test ist einfach, wir setzen die gewünschte Blinkzeit in **setup()** und rufen in **loop()** jeweils **blink()** auf.

```
LedFunktionen led1;

void setup() {
    led1.setBlinkZeit(500); // Pause in ms
}

void printStatus() {
    // ...
}

void loop() {
    led1.blink();
    printStatus();
}
```

Da es sich um eine Simulation handelt, muss der aktuelle LED - Status jeweils auf den seriellen Monitor herausgeschrieben werden.

Schön, die Simulation funktioniert. Aber jetzt sollen auch unsere digitalen und analogen LEDs diese Fähigkeit bekommen. Müssen wir das ganze Spiel mit den beiden anderen Klassen wiederholen?

Nein, das ist nicht notwendig. Wir haben der Basisklasse die neue Fähigkeit beigebracht und die notwendigen Funktionen (**setBlinkZeit()** und **blink()**) public gemacht. Darum stehen sie auch in den abgeleiteten Klassen zur Verfügung.

Dem Blinken unserer realen LEDs steht also nichts mehr im Weg!

Dieses einfache Beispiel zeigt anschaulich, was Vererbung bewirken kann. Ich empfehle dir, damit zu experimentieren. Vererbung ist ein wichtiges Prinzip, dass dir viel Arbeit ersparen kann.

Klassen werden oft in Bibliotheken verwendet. Also werden wir in der nächsten Lektion eine kleine Library aus einer eigenen Klasse erstellen.

Von der Klasse zur Bibliothek (Video 35 - 36)

Um einmal von unseren blinkenden LEDs wegzukommen, beschäftigen wir uns jetzt mit Servos. Die Grundlagen dazu haben wir schon in Lektion 14 betrachtet. Servos sind dazu ausgelegt, mit möglichst hoher Geschwindigkeit einen bestimmten Winkel anzufahren. Dazu existiert in der Arduino - Umgebung bereits eine Servo - Bibliothek.

Wir möchten aber die Möglichkeit schaffen, die Position auch langsam anzufahren. Dazu erstellen wir eine eigene Klasse, übernehmen aber dafür die Funktionalität der vorhandenen Servo - Klasse.

Im **ersten Video** der dreiteiligen Serie geht es darum eine einfache Klasse für einen **langsamen Servo** zu erstellen. Wir teilen diese bereits in Header- und Implementationsfile auf und schreiben auch gleich ein Testprogramm dazu.

Im **zweiten Video** machen wir daraus eine Arduino-Bibliothek, die von jedem Projekt aus mit einem einfachen #include verwendet werden kann.

Im **dritten Video** werden wir die Klasse noch etwas verbessern und robuster machen. Dabei geht es auch darum, wie man Verbesserungen einbauen kann, ohne dass die bestehende Programme Probleme bekommen.

Erben aller Eigenschaften von Servo

Dazu erben wir einfach die ganze Klasse Servo und erstellen daraus die Klasse **LangsamerServo**.

Im Headerfile (**MySlowServo.h**) definieren wir die Klasse, implementieren sie aber nicht. Wir fügen auch noch keine zusätzliche Funktionalität hinzu.

```
class LangsamerServo : public Servo {
public:
    LangsamerServo(); // Constructor
};
```

Die Implementierung erfolgt im cpp - File (**MySlowServo.cpp**).

```
#include "myslowservo.h"

LangsamerServo::LangsamerServo() : Servo() {}
```

Da wir keine neue Funktionalität hinzufügen, benötigen wir hier keinen zusätzlichen Code. Im **Testprogramm** steuern wir die neue Klasse wie ein Standard-Servo an.

```
#include "MySlowServo.h"
LangsamerServo servo;

void setup() {
    servo.attach(7);
    servo.write(90);
}

void loop() {
    servo.write(20);
    delay(1000);
    servo.write(150);
    delay(1000);
}
```

Der Servo bekommt eine Bremse

Normalerweise steuern wir mit **`servo.write(winkel)`** einen bestimmten Winkel an. Der Servo erledigt den Befehl mit höchstmöglicher Geschwindigkeit. Wir möchten diese Geschwindigkeit aber etwas abbremsen können und erstellen dafür einen erweiterten `write()` - Befehl. Mit **`servo.write(winkel, bremse)`** verlangsamen wir die Bewegung. Zusätzlich soll noch abgefragt werden, ob wir die Position schon erreicht haben. Dazu erstellen wir die Funktion **`amZiel()`**.

Da die langsamen Bewegungen 'im Hintergrund' ablaufen, brauchen wir einen Zeittakt, der in kurzen Abständen immer wieder aufgerufen werden muss. Diese Funktion nennen wir **`tic()`**. Die Klasse benötigt noch einige private Variablen. Diese können von aussen nicht direkt angesprochen werden.

```
class LangsamerServo : public Servo {
public:
    LangsamerServo(); // Constructor

    // Positionieren auf einen bestimmten Winkel
    void write(int grad);           // mit maximaler Geschwindigkeit
    void write(int grad, int bremse); // langsames Positionieren (ms
    // Pause pro Grad)

    bool amZiel();

    void tic(); // muss regelmässig aufgerufen werden

private:
    const int _undefiniert = -999;
    int _bremse;
    int _ziel;
    unsigned long _start_zeit;
};
```

In der Implementierung werden die einzelnen Funktionen programmiert. Es gibt hier keinen richtigen Klassenzusammenhalt. Es wird einfach jede Funktion mit dem vorangestellten Klassennamen versehen.

```
LangsamerServo::LangsamerServo() : Servo() {
    _ziel = _undefiniert;
    _start_zeit = 0;
}

void LangsamerServo::write(int grad, int bremse) {
    _ziel = grad;
    _bremse = bremse;
    _start_zeit = 0;
    tic();
}

void LangsamerServo::write(int grad) {
    _ziel = _undefiniert;
    Servo::write(grad);
}

...
```

Um Funktionen der Basisklasse aufzurufen, wird dem Funktionsnamen einfach der Name der Basisklasse vorangestellt.

delay() - Vermeidung im Testprogramm

Da wir jetzt regelmässig **tic()** aufrufen müssen, dürfen wir in **loop()** **delay()** nicht mehr verwenden. Da **delay()** aber sehr bequem ist, können wir mit wenig Aufwand eine ähnliche Funktion erstellen.

Zuerst erstellen wir eine zweite loop() - Funktion (**_loop()**). In diese Funktion packen wir alle Befehle, die ohne Verzögerung aufgerufen werden müssen. **_loop()** wird vom normalen **loop()** aufgerufen. Ausserdem wird **_loop()** überall im Programm, wo wir auf etwas warten, dauernd aufgerufen.

delay() ersetzen wir durch die eigene Funktion **pause()**. Diese verharrt in einer Schleife, bis die Zeit abgelaufen ist. Dabei ruft sie aber immer **_loop()** auf.

```
// Das soll anstelle von delay() verwendet werden
void pause(int ms) {
    unsigned long start = millis();
    while (millis() < start + ms) {
        _loop();
    }
}

// in diesen Loop kommt alles, was regelmässig aufgerufen werden muss
void _loop() {
    servo.tic();
}

void loop() {
    _loop();
    servo.write(20,10);
    while (!servo.amZiel()) _loop();
    pause(500);
    servo.write(150,100);
    while (!servo.amZiel()) _loop();
    pause(500);
}
```

Die Dateistruktur

Um die Programmteile für eine Bibliothek vorzubereiten, müssen sie in eine bestimmte Dateistruktur kopiert werden.

Eine offizielle Anleitung dazu findest du unter <https://arduino.github.io/arduino-cli/library-specification/>

Hobbyelektroniker_Langsamer_Servo	(Name der Library)
keywords.txt	(für Syntax - Hervorhebung)
library.properties	(Eigenschaften der Bibliothek)
docs	(optional, für Dokumentation)
examples	(Beispielprojekte)
SlowServoTest	(unser einziges Beispielprojekt)
SlowServoTest.ino	
extras	(optional)
src	(alle Quelltexte der Library)
MySlowServo.cpp	
MySlowServo.h	

Die *Dateien* müssen erstellt oder angepasst werden.

keywords.txt

Mit dieser Datei werden die Klassen und Methoden der Bibliothek bekannt gemacht, so dass der Editor sie passend einfärben kann. Ich verwende hier eine vereinfachte Version, die vollständige Definition kann in der [Arduino - Dokumentation](#) nachgeschlagen werden.

```
LangsamerServo  KEYWORD1
amZiel          KEYWORD2
tic             KEYWORD2
```

Zwischen dem Schlüssel (LangsamerServo) und dem Typ (KEYWORD1) befindet sich ein TAB, kein Leerzeichen!

Für Klassen und Datentyp wird KEYWORD1 benutzt, für Funktionen geben wir KEYWORD2 an.

library.properties

```
name=Langsamer Servo
version=1.0.0
author=Der Hobbyelektroniker, <der.hobbyelektroniker@gmail.com>
maintainer=Der Hobbyelektroniker, <der.hobbyelektroniker@gmail.com>
sentence=Langsame Bewegung eines Servos.
paragraph=Die Bibliothek ist eine Erweiterung der Standard Servo -
Library.
category=Device Control
url=https://community.hobbyelektroniker.ch
architectures=*
depends=Arduino Servo - Library
```

Mit Ausnahme von depends sind alle Angaben obligatorisch!

Zusammenfassung Programmierung

Name	Name der Bibliothek
Version	Version im Format 1.0 oder 1.0.0
author	Name und EMail des Entwicklers. Format: Vorname Name, <EMail>
maintainer	Betreuer der Software Das Format ist identisch mit author
sentence	Erster Satz der Beschreibung. Wird als Titel verwendet.
paragraph	Rest der Beschreibung
category	Eine dieser Kategorien: <ul style="list-style-type: none">• Display• Communication• Signal Input/Output• Sensors• Device Control• Timing• Data Storage• Data Processing• Other• Uncategorized
url	Webseite des Projekts, oft wird hier eine Github - Seite angegeben
architectures	Liste der Boards, die die Library benutzen können (mit , separiert). Wenn keine Einschränkungen bestehen, kann * angegeben werden.
depends	Abhängigkeit von anderen Libraries.

SlowServoTest.ino

Das ist unser Testprojekt. Wir müssen nur den Import der Library ändern. Dieser erfolgt jetzt mit spitzen Klammern, da wir aus dem Bibliotheksverzeichnis importieren möchten.

```
#include <MySlowServo.h>
```

Ausserdem werden die Bibliotheksdateien MySlowServo.h und .cpp aus dem Projektverzeichnis gelöscht.

MySlowServo.cpp

Auch hier werden für den Import die spitzen Klammern verwendet.

```
#include <MySlowServo.h>
```

MySlowServo.h

Diese Datei kann unverändert übernommen werden.

Das ZIP - File

Jetzt kann aus der Dateistruktur eine ZIP - Datei erstellt werden.

In meinem Fall erhalte ich **Hobbyelektroniker_LangsamerServo.zip**. Obwohl der Name der ZIP - Datei nicht zwingend dem Verzeichnisnamen entsprechen muss, ist es natürlich sinnvoll.

Diese Bibliothek kann jetzt in der Arduino - IDE mit Sketch / Bibliothek einbinden / .ZIP Bibliothek hinzufügen... installiert werden. Sie steht nachher allen Projekten zur Verfügung und das Beispielprojekt ist unter Datei / Beispiele zu finden.

Nicht besprochen wird hier die Einbindung der eigenen Bibliothek in die Bibliotheksverwaltung. Das würde hier zu weit führen. Wenn dich das interessiert, findest du weitere Informationen unter

<https://github.com/arduino/Arduino/wiki/Library-Manager-FAQ>

Wo sind den nun die Dateien gespeichert?

In deinem **Sketchverzeichnis** findest du ein Unterverzeichnis **libraries**. Dort findest du deine vollständige Dateistruktur aus dem ZIP - File. Es ist auch möglich, diese Dateien manuell in den libraries - Ordner hinein zu kopieren. Beim nächsten Start der IDE wird die Bibliothek dann automatisch erkannt.