ARDUINO
Befehlsübersicht

Übersetzt in deutsche Sprache
Mit zusätzlichen Erklärungen, Ergänzungen und hilfreichen Tipps

https://arduinoforum.de
DAS deutsche ARDUINO-Forum, die beste Quelle für Fragen, Probleme, Hilfe, Hardwaretipps etc. rund um die ARDUINO-Programmierung!

Unter dem Link: **https://www.arduinoforum.de/code-referenz**
kann diese Referenz heruntergeladen werden. Hier gibt es immer die aktuelle Version. Zusätzlich auch als Download verfügbar die im Buch verwendeten Libraries sowie Beispielsketche, soweit ich Zeit und Muße gefunden habe, diese zu erstellen. *Diese Downloads sind also nicht vollständig / abschließend.*

*** Andreas Nagel *** Stand: 5. Mrz. 2020 ***

Copyright

Jegliche Benutzung und Weitergabe dieses Werkes ist nur zu privaten und nichtkommerziellen Zwecken unentgeltlich erlaubt. Eine anderweitige Nutzung, gleich welcher Art, bedarf der Erlaubnis des Autors. Dazu zählen insbesondere der Verkauf, Verlegung als Buch, Beigabe zu kommerziellen Produkten, Übersetzung, Mikroverfilmung, Einspeicherung und Verarbeitung in kommerziellen elektronischen Systemen. Etwaige weitere Nutzungshinweise ergeben sich aus den Quellenangaben (siehe oben). Die grafisch schönen Handskizzen habe ich selbst erstellt. ☺
Alle Waren- und Markennamen werden ohne Gewährleistung der freien Verwendbarkeit zur Information benutzt und sind möglicherweise eingetragene Warenzeichen und in Besitz der jeweiligen Inhaber.
Sollte Ihnen dieses Werk verkauft worden sein, bitte ich um Kontakt via Email DL1AKP@web.de.

Danksagung

Ich möchte mich für die vielen Zuarbeiten bedanken, die mich im Laufe der Jahre erreicht haben und geholfen haben, dieses Werk zu verbessern, von Fehlern zu befreien und zu ergänzen.

In diesem Zusammenhang Dank an verschiedene Mitglieder des deutschen ARDUINO-Forums und andere Bastler, die mir selbst auch schon bei Fragen geholfen haben. An geeigneter Stelle sind diese namentlich erwähnt, wenn ich ihr Wissen oder Hinweise verwendet habe.

Besonderer Dank gilt René (via Email) und Ulrich (via Email), die mit viel Aufwand diese Referenz gegengelesen haben und dadurch unzählige Rechtschreibfehler und Unklarheiten in der Formulierung gefunden haben.

Danke an Heinz Baumstark, dessen interessante und gut verständliche Abhandlung zum Thema „Von delay zur Methode“ ich in diese Referenz übernehmen durfte.

Danke auch an meine Lebensgefährtin Sandra für ihre Geduld und Nachsicht, wenn ich mal wieder am Rechner saß und an diesem Werk gearbeitet habe.

Vorwort

In diesem Buch sind die wichtigsten Befehle und Funktionen der ARDUINO-Sprache aufgeführt, und dies alles komplett in Deutsch. Weiterhin auch einige, meiner Meinung nach wichtige, Libraries und deren Funktionen / Befehle.

Zur Benutzung von Libraries gibt es stark unterschiedliche Ansichten. Diese reichen von totaler Abneigung bis hin zu Euphorie. Ich persönlich nutze gern und viel Libraries, beschleunigen sie doch das Erstellen des funktionierenden Projektes massiv. Auch ist der Code wesentlich leichter lesbar. Nachteil ist, dass der Hex-Code, welcher hochgeladen wird, deutlich größer ist und man eventuell an die Speichergrenze kommt. Dann kann man nachdenken, entweder weniger Libraries zu benutzen, oder einen ARDUINO mit mehr Speicherplatz im Flash. Die Entscheidung obliegt jedem selbst, ich hatte diesen Fall noch nicht.

Ich denke, das Ergebnis ist entscheidend: Das Projekt soll wie gewünscht funktionieren. Tut es das, OK! Warum sich Gedanken machen, ob man noch das eine oder andere Prozent an Speicher einsparen kann, oder das Programm ein paar Mikrosekunden schneller läuft? Die Zeit kann man doch lieber mit einem neuen, faszinierenden Projekt verbringen, oder nicht?

Auch über das „optische“ Ergebnis lässt sich streiten. Also: Ist mein Sketch auch schön? Schönheit liegt immer im Auge des Betrachters. Also macht es so, dass es für Euch am optimalsten ist. Viel Erläuterungen einfügen, Hinweise, Beschaltungen und ähnliches. Gefällt es einem anderen nicht – was soll’s! Er kann ja seine Projekte machen, wie er es möchte.

Die erste Version dieser Referenz habe ich im Juli 2013 erstellt, damals als Hilfe für meine interessierten Kollegen. Dann habe ich sie noch einigen anderen gezeigt und so entstand die Idee, dieses Werk zu veröffentlichen und so vielen Interessierten zu helfen. Erstmals online ging diese Referenz dann im März 2017. Ziel dieses Buches ist es, dem Leser den Einstieg so leicht wie möglich zu machen. Das geht am besten in der Muttersprache. Ich kann fließend englisch, aber einen Fachartikel in englisch zu lesen und zu verstehen ist nochmal etwas ganz Anderes. Diese Erkenntnis animierte mich zur Erstellung dieses Werkes.

Ich bemühe mich, dieses Buch mit weiteren, sinnvollen Kapiteln zu ergänzen. Also zum Beispiel die am häufigsten benutzten Sensoren, Libraries, Aktoren. Ein bisschen Schaltungstechnik, wie schließe ich ein Relais an, und so weiter. Dabei soll es so bleiben, dass der Laie nicht überlastet wird und die Lust verliert, das wäre schade. Daher wird es keine komplizierten, theoretischen Abhandlungen geben, keine Berechnungen und Verweise auf etwaige zu beschaffende, teure Messtechnik.

Bei der Programmierung von ARDUINO’s gibt es die berühmten vielen Wege, die nach Rom führen. Meine Ausführungen in den Kapiteln zur Benutzung der verschiedenen Sensoren, Aktoren etc. sind daher nur eine von vielen. Sicher gibt es andere, oder elegantere, oder schnellere, oder schönere, oder bessere, oder coolere, oder libraryfreie, oder… Also nagelt mich nicht fest, das meine Version die einzig wahre ist. Aber es ist eine von mir getestete und funktionierende Variante.

Also: Lasst uns anfangen, faszinierende Projekte umzusetzen, ob es eine Uhr für das Wohnzimmer ist, eine Steuerung für den Gartenspringbrunnen, oder gar ein Roboter…

Es ist wichtig zu wissen, **dass auch alle sonstigen C/C++ Befehle benutzt werden können**. Das kann hier nicht aufgeführt werden, da es extrem umfangreich ist, und einfach zu weit führen würde.
Wer sich dafür interessiert oder C/C++Befehle verwenden will, sei auf das Internet verwiesen, hier speziell auf Quelle [12].

Es stehen auch teilweise Beispieldateien sowie die verwendeten Libraries zum Download bereit. Im Buch wird darauf hingewiesen, wenn es so etwas gibt.

Falls im Buch Kapitel fehlen, unvollständig sind, oder Hinweise auf Libraries oder Files fehlen, denkt bitte daran, das dieses Werk zum privaten Gebrauch komplett kostenlos ist.
Da ich voll berufstätig bin, ist meine Zeit begrenzt und ich kann nicht regelmäßig daran arbeiten.

Weil es permanent mit dem Stichwortverzeichnis Probleme gab, und ich es mühsam händisch nachbearbeiten musste, habe ich es ganz entfernt. Das PDF ist einfach nach Stichworten durchsuchbar, also sollte das kein Problem darstellen.

Viel Spaß!

Andreas Nagel, DL1AKP@web.de 2017 bis 2020

Inhaltsverzeichnis

Copyright ... 2
Danksagung ... 2
Vorwort ... 3
Inhaltsverzeichnis ... 4
ARDUINO-Hardware ... 12
Allgemeines ... 12
Die Betriebsspannung +UB ... 12
Belastbarkeit ... 12
Begriffserklärungen ... 14
Programmstruktur ... 14
Variablen ... 14
Befehle ... 14
Funktionen ... 14
Methoden ... 15
Operatoren ... 15
Abfragen ... 16
Schleifen ... 16
Bibliotheken ... 16
Struktur ... 18
setup() ... 18
loop() ... 18
Weitere Syntax ... 19
Kommentare ... 19
; Semikolon ... 19
{} geschweifte Klammern ... 19
#define ... 20
#include ... 21
#ifdef #else #endif ... 21
Datentypen ... 23
void ... 23
byte (uint8_t) ... 23
int (int16_t) ... 23
short ... 24
word (uint16_t) ... 24
unsigned int (uint16_t) ... 24
long (int32_t) ... 24
unsigned long (uint32_t) ... 24
float ... 24

double 25
boolean (bool) 25
Array‘s 25
char (int8_t) 26
unsigned char (uint8_t) 26
string (Datentyp & char-Array) 27
Datentypen-Bezeichner z.B. UL, U, L etc. 29
Konstanten 31
Logische Pegel, true und false (Bool’sche Konstanten) 31
false 31
true 31
Pin-Zustände HIGH und LOW 31
Pegel in der Digitaltechnik 31
HIGH 32
LOW 32
Digitale Pins konfigurieren 32
Aliase für Pins benutzen 32
Pins konfiguriert als Eingänge (INPUT) 33
Pins konfiguriert als INPUT_PULLUP (internen Pullup-Widerstand einschalten) 33
Pins konfiguriert als Ausgänge (Outputs) 33
Integer Konstanten 34
Floating point Konstanten (Fließkomma) 34
Verzweigungen und Schleifen 35
for-Schleifen 35
if (Bedingung) und ==, !=, <, > (Vergleichsoperatoren) 35
Vergleichsoperatoren 36
if / else 36
switch / case Anweisungen 37
while - Schleifen 38
do – while 38
break 38
continue 39
return 39
goto 40
Rechenoperationen 41
= Zuweisungsoperator 41
Grundrechenarten (Addition, Subtraktion, Division, Multiplikation) 41
% Modulo 42
Verknüpfungsoperationen (Boolesche Operatoren) 43
&& (logisches und) 43
|| (logisches oder) 43

! (nicht) ... 43
Digital In / Out ... 44
pinMode() ... 44
digitalWrite() ... 44
digitalRead() ... 45
Grundlegendes Analog In / Out ... 46
analogReference() ... 46
analogRead() ... 46
analogWrite() ... 47
Gleichspannung analog ausgeben ... 48
Erweitertes Analoges In / Out ... 50
tone() ... 50
noTone() ... 50
shiftOut() ... 51
shiftIn() ... 52
pulseIn() ... 53
Datenkonvertierung ... 54
byte() ... 54
int() ... 54
word() ... 55
long() ... 55
float() ... 55
Zeit ... 57
delay() ... 57
micros() ... 57
millis() ... 58
delayMicroseconds() ... 58
Richtige Benutzung des millis() – Befehles ... 59
Zusammengesetzte Operatoren ... 61
Zusammengesetztes bitweises ODER (|=) ... 61
Zusammengesetztes bitweises UND (&=) ... 61
+= , -= , *= , /= Zusammengesetzte + - * / ... 61
++ (inkrement) / — (dekrement) ... 62
Bitoperatoren ... 63
Bitweises links- (<<) und bitweises rechtsschieben (>>) ... 63
Bitweises NICHT (~) ... 64
Bitweises UND (&) ... 64
Bitweises ODER (|) ... 64
Bitweises XOR (^) ... 65
Zufallszahlen ... 66
randomSeed(seed) ... 66

random() ... 66
Externe Interrupts ... 68
attachInterrupt() ... 68
detachInterrupt() ... 69
Interrupts (global) ... 70
interrupts() ... 70
noInterrupts() ... 70
Mathematische Funktionen ... 72
min(x, y) ... 72
max(x, y) ... 72
abs(x) ... 72
constrain(x, a, b) ... 73
map(value, fromLow, fromHigh, toLow, toHigh) ... 73
pow(base, exponent) ... 74
sqrt(x) ... 74
Geltungsbereich und Qualifikatoren ... 76
Geltungsbereich von Variablen ... 76
static ... 76
volatile ... 77
const ... 78
#define oder const? ... 78
Trigonometrie (Dreiecksberechnungen) ... 79
sin(rad) ... 79
cos(rad) ... 79
tan(rad) ... 79
Zeichen (Character-) Funktionen ... 80
isAlphaNumeric(thisChar) ... 80
isAlpha(thisChar) ... 80
isAscii(thisChar) ... 81
isWhiteSpace(thisChar) ... 81
isControl(thisChar) ... 82
isDigit(thisChar) ... 82
isGraph(thisChar) ... 83
isLowerCase(thisChar) ... 83
isPrintable(thisChar) ... 84
isPunct(thisChar) ... 84
isSpace(thisChar) ... 85
isUpperCase(thisChar) ... 85
isHexadecimalDigit(thisChar) ... 86
Hilfsfunktionen ... 87
sizeof ... 87

PROGMEM 87
F()-Makro 89
Bits und Bytes 90
lowByte() 90
highByte() 90
bitRead() 90
bitWrite() 91
bitSet() 91
bitClear() 91
bit() 92
Serielle Kommunikation (UART) 93
Allgemeines 93
Serial.begin(speed) 94
Serial.print(data) Serial.print(data, encoding) 94
Serial.println(data) Serial.println(data, encoding) 95
Serial.available() 95
Serial.read() 95
Serial.flush() 95
String Objects 96
Allgemeines 96
String() 96
charAt() 97
compareTo() 98
concat() 99
endsWith() 99
equals() 100
equalsIgnoreCase() 100
getBytes() 101
indexOf() 101
lastIndexOf() 102
length() 103
replace() 103
setCharAt() 104
startsWith() 104
substring() 105
toCharArray() 105
toInt() 106
toFloat() 106
toLowerCase() 107
toUpperCase() 107
trim() 108

String Operatoren 109
[] (Zugriff auf Zeichen) 109
+ (Operator „Anhängen“) 109
== (Operator „Vergleich“) 110
LCD-Textdisplays und deren Benutzung 111
Allgemeines 111
LiquidCrystal() 112
begin() 112
clear() 113
home() 113
setCursor() 114
write() 114
print() 115
cursor() 115
noCursor() 116
blink() 116
noBlink() 116
display() 117
noDisplay() 117
scrollDisplayLeft() 117
scrollDisplayRight() 118
autoscroll() 118
noAutoscroll() 119
leftToRight() 119
rightToLeft() 119
createChar() 120
Sonderzeichen, Symbole und Umlaute 121
LCD-Grafikdisplays und deren Benutzung 123
Allgemeines 123
Programmierung 125
EEPROM schreiben und lesen 127
Allgemeines 127
Die EEPROM-Library 127
EEPROM.read() 127
EEPROM.write() 128
EEPROM.update() 129
EEPROM.put() 129
EEPROM[] 130
Timer-Interrupts und deren Benutzung 131
Allgemeines 131
Verwendung des Timer1 131

Taster und deren Benutzung 133
Anschließen von Tastern 133
Taster nach GND 133
Taster nach +UB 133
Benutzung von Tastern mit Interrupts 133
Entprellen von Tastern 134
Allgemeines 134
Verwendung der Debounce-Library 134
Entprellen mit zusätzlichen Bauteilen 135
Mehrfachfunktionen mit Tastern (Doppelklick, langer Klick etc.) 136
Benutzung der Library OneButton 136
Einfacher kurzer Klick 136
Doppelklick 137
Langer Klick wiederholte Funktion 137
Langer Klick einfach 137
RTC-Uhr und deren Benutzung 138
Allgemeines 138
Die RTC DS3231 138
Beschaltung 139
Benutzung der Library 139
Sommer- / Winterzeit Berechnung 140
Temperatursensoren und deren Benutzung 141
Allgemeines 141
DS18B20 / DS18S20 (1wire-Bus-Sensor) 141
Anschließen des Sensors 142
Programmierung 142
MCP9700 (analoger Sensor) 143
Anschließen des Sensors 143
Programmierung 143
LM75 (I2C-Sensor) 144
Luftfeuchte-Sensoren und deren Benutzung 145
Anschließen des Sensors 145
Programmierung 145
Ultraschall-Sensoren und deren Benutzung 146
Allgemeines 146
Beschaltung 146
Funktion 146
Programmierung 146
Der I2C-Bus und seine Benutzung 149
Allgemeines 149
I2C-LCD-Displays 149

LM75 Temperatursensor 150
Portexpander PCF8574 / 8575 152
Portexpander MCP23008 / 23017 154
Der Watchdog-Timer und seine Benutzung 156
Allgemeines 156
Benutzung des Watchdog-Timers 156
Hinweis bei Benutzung von China-Klones 157
Von delay bis zur Methode 159
Allgemeines 159
Worum geht es? 159
Der Grundsketch 160
Den Sketch für die zweite LED erweitern 161
delay() durch millis() ersetzen 161
Eine eigene Funktion erstellen 162
Objekte und Methode 164
Array mit Objekten 164
Kleine hilfreiche Programm-Teile 166
Freien RAM anzeigen 166
Programm anhalten (für immer) 166
Software-Reset 166
RAM sparen bei Serial.print 167
Digital-Ausgang (z.B. LED) togglen 167
Binärausgabe 8bit, 16bit, 32bit auf seriellem Port 167
Binärausgabe 8bit 167
Binärausgabe 16bit 167
Binärausgabe 32bit 168
String nach char kopieren 168
RAM sparen bei der Verwendung von Ethernet 168
Quellenverzeichnis 171
Versionsverlauf / Historie der Änderungen & Ergänzungen 172

ARDUINO-Hardware

Allgemeines

Vorab ist es wichtig, einige Dinge über die ARDUINO-Board's zu wissen, damit der Bastelspaß nicht getrübt wird. Darum geht es in diesem Kapitel.

Man kann die unterschiedlichen ARDUINO-Board's aus verschiedenen Lieferquellen beziehen. Eine bevorzugte möchte ich hier nicht nennen, das muss jeder selbst wissen. Der Bezug aus Deutschland ist oft (unangemessen) teuer, so kommt man schließlich auf die Bezugsquelle China. Hier ist zu beachten, das die Board's direkt aus China oftmals als **USB-Chip einen CH340** verwenden. Das ist prinzipiell nicht schlimm, die Treibersuche gestaltet sich allerdings als abenteuerlich. Ich habe funktionierende Treiber für WINDOWS und MacOS gefunden. Sie befinden sich im Download zur Referenz.

Auch gibt es verschiedene Boards für die unterschiedlichsten Anwendungen. Sie unterscheiden sich durch die Anzahl der IO-Pins, der Betriebsspannung, der Größe und des Preises. Man findet also für jeden Anwendungsfall das Richtige.

Die Betriebsspannung +UB

Mit der Betriebsspannung ist es auch so eine Sache... Da ich schon seit Anfang der 80er Jahre mit digitalen Baugruppen gearbeitet habe, bin ich an die Spannung **5V DC** gewöhnt. Mit diesen 5V arbeiten auch die meisten ARDUINO-Board`s. Es gibt aber auch welche, die mit nur **3,3V DC** arbeiten. Diesen wichtigen Punkt muss man immer beachten! Ansonsten kann es zur Zerstörung des entsprechenden Board's kommen. Die ARDUINO's tolerieren an ihren Pins maximal die Betriebsspannung plus 0,5V.

Da die meisten benutzbaren Shield's oder Baugruppen mit der 5V Betriebsspannung arbeiten, empfehle ich die Verwendung der ARDUINO's mit 5V DC Betriebsspannung. Man erspart sich so Probleme, Fehlfunktionen und im schlimmsten Falle Zerstörungen.

Hat man aber doch ein 3,3V-Board, darf an den Eingängen auch maximal diese Spannung anliegen. NICHT 5V DC! Wenn es ein Ausgang ist, dann liegen bei HIGH-Pegel nur 3,3V DC an. Dies ist bei der weiteren Beschaltung, zum Beispiel mit TTL-Schaltkreisen zu beachten.

Belastbarkeit

Auf den gängigen ARDUINO's arbeitet als Mikrocontroller ein AVR der Firma ATMEL. Dieser hat natürlich auch elektrische Kennwerte, welche man aus dem Datenblatt entnehmen kann. Dieses Datenblatt ist in englisch und sehr umfangreich, dadurch etwas unübersichtlich.

Belastbarkeit einzelner Pins

Im Datenblatt stehen unter anderem die Belastbarkeit der entsprechenden Pins, wenn diese als Ausgang benutzt werden. Hier gibt es die gesamte erlaubte Belastung des Schaltkreises und die Belastung eines einzelnen Ausganges.

Ein einzelner Pin, als Ausgang verwendet, kann mit maximal 40mA (absolutes Maximum laut Datenblatt, man sollte es nicht ausreizen) belastet werden. Das ist ein kleines Relais oder eine starke LED. Will man mehr Strom, dann ist ein Transistor als Treiber notwendig. Um auf der sicheren Seite zu sein, sollte man 20mA nicht überschreiten. Für LED's reichen heutzutage bereits 2mA gut aus, um diese hell leuchten zu lassen.

Belastbarkeit des gesamten Chip's

Laut Datenblatt soll eine gesamte Belastbarkeit ALLER Pins gleichzeitig nicht mehr als 200mA betragen. Hier sieht man schon, das man also maximal nur fünf Pins mit je 20mA belasten kann, um diese Grenze zu erreichen. Der ARDUINO hat aber mehr IO-Pins, welche man als Ausgang verwenden kann. Dies ist beim Schaltungsentwurf zu beachten. Gerade wenn man viele Ausgänge braucht, die einen hohen Strom treiben sollen, empfiehlt sich die Verwendung von Treiber-IC. Diese Schaltkreise beinhalten mehrere „Kanäle" und treiben oftmals Strom bis zu 200mA je Kanal. Exemplarisch soll hier der Typ ULN2803 genannt werden, welcher 8 Kanäle hat und sehr gut funktioniert.

Belastbarkeit der Stromversorgung des ARDUINO

Auf den ARDUINO-Boards befindet sich bereits ein Spannungsregler, sodass man diese durch einspeisen an dem Pin „Vin" oder der verbauten Hohlsteckerbuchse mit Spannung mehr als 5V DC versorgen kann. Dieser Spannungsregler hat auch eine maximale Belastbarkeit, welche nicht überschritten werden sollte. Es empfiehlt sich, an dieser Buchse bzw. „Vin", zwischen 9V DC und 12V DC anzulegen. Der Spannungsregler auf dem Board wandelt, einfach ausgedrückt, die zu viele Spannung und den fließenden Strom in Wärme um. Diese kann das Board und den IC schnell überhitzen und zur Zerstörung oder Fehlfunktionen führen.

Um das zu vermeiden, am besten so vorgehen:

1. Bei Versorgung über die USB-Buchse nicht mehr als 500mA Strom ziehen.
2. Bei Verwendung von Vin nicht mehr als 100mA und Spannung von 9V DC bis 12V DC
3. Bei Verwendung der Hohlsteckerbuchse siehe 2.
4. Wird mehr als 100mA gebraucht und die USB-Buchse kann nicht verwendet werden, empfiehlt es sich, über den Pin „5V" eine stabilisierte Spannung von 5V DC mit der gewünschten Belastbarkeit einzuspeisen, zum Beispiel aus einem Steckernetzteil.

Dies sind persönliche Empfehlungen von mir! In den Unterlagen zu den verschiedenen ARDUINO-Board's steht möglicherweise etwas anderes. Bei Beachtung von Punkt eins bis vier ist man auf jeden Fall auf der sicheren Seite.

Begriffserklärungen

Programmstruktur

Ein Programm in ARDUINO-Sprache besteht aus mindestens zwei Anweisungsblöcken (auch Methoden oder Funktionen genannt). Das ist eine Gruppe von Befehlen, die durch den Aufruf des Namens ausgeführt werden. Der Anweisungsblock `void setup()` wird **einmal** nach dem Anlegen der Betriebsspannung ausgeführt, wobei `void loop()` ständig wiederholt wird (loop = Schleife).

```
void setup(){
}

void loop(){
}
```

Variablen

Eine Variable ist ein Container für Werte des Typs der Variable. Variablentypen sind:

Variablentyp	Bedeutung	Beschreibung
byte	Byte	ganze Zahl von 0 bis 255
int	Integer	ganze Zahlen (-32.768 bis 32.767)
long	ganze Zahlen	(-2 Milliarden bis 2 Milliarden)
float	Fließkommazahl	gebrochene Zahlen (3,14 etc.)
boolean	Boolescher Wert	Logischer Wert 0 (low) oder 1 (high)
char	engl: character	Alphanumerische Zeichen (Buchstaben, Zahlen, Sonderzeichen)
array	Variablenfeld	mehrere Werte eines Variablentyps können gespeichert werden

Variablen können **global** (überall verfügbar) oder **lokal** (nur in der jeweiligen Funktion verfügbar) deklariert werden. Das ist unbedingt zu beachten, sonst kann es zu Fehlermeldungen oder unlogischen Programmabläufen kommen. **Siehe hierzu Kapitel „Geltungsbereich von Variablen“.**

Befehle

Befehle sind Anweisungen, die Methoden in der ARDUINO-Software aufrufen.
z.B. pinMode(), digitalWrite etc.….

Funktionen

Funktionen sind Programmanweisungsblöcke. Wiederkehrende Abfolgen von Befehlen können in Funktionen sinnvoll strukturiert werden. Parameter können an Funktionen übergeben und Werte zurückgeliefert werden. In BASIC wird so etwas „Unterprogramm“ genannt, C ist aber im Umgang mit Funktionen wesentlich leistungsfähiger als BASIC.

Eine einfache Funktion könnte so aussehen:

```
void led(){
// Anweisungsblock Start
digitalWrite(ledPin, HIGH);
delay(500);
```

```
digitalWrite(ledPin, LOW);
delay(500);
// Anweisungsblock Ende
}
```

Nun kann man die Funktion, z.B. aus dem `void loop()`, aufrufen mit: `led();`
Parameter lassen sich auch an eine Funktion übergeben. Die Struktur sieht dann so aus:

```
void led (int thePin, int dauer){
digitalWrite(thePin, HIGH);
delay(dauer);
digitalWrite(thePin, LOW);
delay(dauer);
}
```

Hierbei wird der Parameter thePin und dauer übergeben. Der Aufruf kann dann so erfolgen:

```
led(3,1000);
```

Man kann auch einen Wert von einer Funktion zurückgeben lassen. Bisher wurde als erstes Wort `void` geschrieben, um anzuzeigen, dass nichts (=void) zurückgegeben wird. Daher verwendet man jetzt anstelle von `void` den Variablentyp, den das Ergebnis haben wird und liefert es am Ende des Anweisungsblockes mit dem Schlüsselwort `return` an die Funktion.

```
float quadrat(float x){
float ergebnis = x*x;
return ergebnis;
}
```

Der Aufruf wäre z.B.:

```
wert = quadrat(12.3);
```

Methoden

Methoden sind Funktionen, wenn sie Elemente einer Klasse sind.
Im Code kann man das leicht unterscheiden: Besteht der Name aus 2 Teilen:
Objektname.methodenname() handelt es sich um eine Methode. Z.B. `Serial.begin(9600);` oder `Servo.write(90);`

Bei einfachen Namen wie z.B. `digitalWrite();` ist es nur eine normale Funktion.

Danke für diese Erklärung an Mitglied „Microbahner“ aus dem deutschen ARDUINOforum.

Operatoren

Operatoren sind mathematische oder logische Funktionen. Hier die wichtigsten im Überblick.

Operator	Bedeutung	Anwendung	Funktion
Arithmetische Operatoren			
=	Zuweisung	a=2*b	Weist der linken Seite den Wert auf der rechten Seite zu.

+	Addition	a=b+c	
-	Subtraktion	a=b-c	
++	Inkrementieren	a++	Zählt zu der Variable 1 hinzu (+1).
--	Dekrementieren	a--	Zieht von der Variable 1 ab (-1)
*	Multiplikation	a=b*c	
/	Division	a=b/c	Dabei darf c nie gleich Null sein
%	Modulo	a=b%c a=7%5 ; a=2 a=10%5 ; a=0	Liefert den Rest bei der Division von b/c. Ist b durch c teilbar, so ist das Ergebnis = 0.
	Vergleichsoperatoren		
==	Gleichheit	a==b	Prüft auf Gleichheit.
!=	Ungleichheit	a!=b	Prüft auf Ungleichheit.
<	kleiner als	a<b	
>	größer als	a>b	
<=	kleiner gleich	a<=b	
>=	größer gleich	a>=b	
	Boolesche Operatoren (können wahr oder falsch sein)		
&&	UND	(a==2)&&(b==5)	Wenn beide Seiten wahr sind, ist das Ergebnis auch wahr.
||	ODER	(a==2)||(b==5)	Wenn eine oder beide Seiten wahr sind, ist das Ergebnis wahr.
!	NICHT	!(a==3)	Ist wahr, wenn a nicht 3 ist.

Abfragen

Eine Abfrage prüft, ob eine Bedingung (nicht) erfüllt ist. Abfragen können also den Programmablauf steuern.

Beispiele: if-Abfrage, switch-case-Abfrage

Schleifen

Schleifen können Anweisungen bis zum Erreichen einer Abbruchbedingung wiederholen. Schleifen können somit ebenfalls den Programmablauf steuern.

Beispiele: for-Schleife, do-while-Schleife

Bibliotheken

Bibliotheken (Libraries) erweitern den Funktionsumfang der ARDUINO-Software um weitere Anweisungen. Es gibt Bibliotheken für Servos, LCDs, und hunderte mehr. Will man sie verwenden, müssen sie in den Sketch eingebunden werden.

Die von mir in diesem Buch verwendeten Libraries sind diese, welche ich selbst benutze. Es gibt zu jedem Problem mit hoher Wahrscheinlichkeit viele verschiedene Libraries, mit unterschiedlichen Funktionen, mehr oder wenig einfach und so weiter. Ob die hier verwendete Library dem eigenen Geschmack oder Vorstellungen entspricht, muss jeder selbst entscheiden. Es ist kein Dogma und die

Verwendung ist jedem freigestellt. **Es ist nur EIN WEG VON VIELEN**. Mit fortschreitender Erfahrung mit den ARDUINO's findet man dann auch selbst die bevorzugten Libraries.

Verwendung von Bibliotheken in der IDE

Der Benutzung der Bibliotheken steht erstmal die korrekte Platzierung in dem jeweiligen Betriebssystem voraus. Das ist leider bei jedem System unterschiedlich und ändert sich auch teilweise. So hängt es auch von der Version der IDE ab, wo sich die Libraries befinden und wo man eigene Libraries speichert. Hier ist ein gewisses Grundverständnis für das verwendete Betriebssystem von Nöten.

Im Hauptmenü findet man unter ‚**Sketch**' den Befehl ‚**Bibliothek einfügen**'. Hier wählt man einfach die Bibliothek aus, die man verwenden will und im Sketch erscheint die Include-Zeile. Z.B.:

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

Man kann auch eine neue Library hinzufügen. Der entpackte Ordner der Library muss in den Libraries-Ordner kopiert werden. Existiert dieser Ordner nicht, muss man ihn anlegen. Nach dem Programm-Neustart steht die Bibliothek zum Einfügen in den Sketch bereit.

Es kann auch vorkommen, das eine Library mit der IDE-Version 1.x.x funktioniert, und nach einem Update auf die neuere IDE-Version dann nicht mehr. Das ist äußerst ärgerlich und nicht gut gelöst. Hier kann man nur entweder so lang wie möglich bei einer IDE-Version bleiben, keine Libraries verwenden, den Code immer wieder anpassen oder warten, bis die Library gegebenenfalls auch ein Update bekommt.

Hinweis

Bei der Verwendung von aus dem Internet geladenen Libraries ist zu beachten, dass es für ein und denselben Zweck oft mehrere, komplett unterschiedliche Libraries gibt. Findet man nun irgendeinen Beispielsketch, kann es sein, dass dieser nicht funktioniert. Also muss immer genau geschaut werden: Passt mein Beispielsketch zu der geladenen Library?

Strukturierte Programmierung ist eine Kunst. Leider beherrsche ich diese Kunst auch nicht, ich stamme aus der "echten" Elektronik. Diese hat den Vorteil, dass sie anhand des Schaltplanes lesbar ist. Die Lesbarkeit muss hier im Sketch liegen. Das obliegt jedem selbst, sein Programm zu gestalten. Letztlich muss es wie gewollt funktionieren.

Praktisch ist es auch, im eigenen Sketch hinter der verwendeten Library zu vermerken, wo diese her stammt. So kann man später Verwirrungen und Probleme vermeiden. (Danke Info hotsystems vom ARDUINOforum)

```
#include <LiquidCrystal_I2C.h>       // https://bitbucket.org/fmalpartida/new-liquidcrystal/downloads
```

Struktur

setup()

Die **setup()** Funktion wird zu Beginn des Programms aufgerufen. Sie wird benutzt, um unter anderem die Verwendung der Pins (Eingang / Ausgang) festzulegen, Variablen einen bestimmten Wert zuzuweisen und vorher eingebundene Bibliotheken über bestimmte Schlüsselbefehle zu initialisieren bzw. zu starten. Sie wird nur einmal beim Programmstart als erste Funktion aufgerufen z.B. bei einem Reset des ARDUINO oder das Anlegen der Betriebsspannung.

Beispiel

```
int buttonPin = 3;

void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

void loop()
{
  // ...
}
```

loop()

Die **loop()** Funktion wird nach der **setup()** Funktion aufgerufen und stellt eine Endlosschleife dar. Hier ist der Platz für das Hauptprogramm.

Beispiel

```
#define buttonPin 3

// setup initialisiert serial und den button pin
void setup()
{
  Serial.begin(9600);
  pinMode(buttonPin, INPUT);
}

// loop überprüft den button pin jedes Mal
// und sendet ein H wenn der button gedrückt wird
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
    Serial.write('L');

  delay(1000);
}
```

Weitere Syntax

Kommentare

Kommentare in einem Programm sind Zeilen im Programmcode, um sich selbst oder andere darüber zu informieren, wie das Programm funktioniert. Kommentare werden vom Compiler ignoriert und nehmen keinen Platz auf dem Chip in Anspruch.
Es gibt zwei Arten von Kommentaren: einzeilige und mehrzeilige Kommentare.

Beispiel

```
x = 5;  // Das ist ein einzeiliger Kommentar, alles hinter dem
         // ist ein Kommentar bis zum Ende der Zeile

/* Das ist ein mehrzeiliger Kommentar, er kann benutzt werden,
um ganze Passagen auszukommentieren

if (gwb == 0){    // ein einzeiliger Kommentar in einem mehrzeiligen ist ok
x = 3;            /* noch ein mehrzeiliger Kommentar ist nicht ok */
}
// Vergiss nicht das "schließende" Kommentarzeichen!
*/
```

Hinweis

Wenn man im Programmcode nach Fehlern sucht, kann es hilfreich sein, ganze Teile des Programms auszukommentieren. Dies behält die Zeilen des möglicherweise nicht funktionalen Codes bei und der Compiler ignoriert diese Zeilen.
Weiterhin empfiehlt es sich auf jeden Fall, nicht mit Kommentaren zu geizen. Sie kosten nichts und es kommt ganz sicher der Zeitpunkt, wo man sich später mal fragt: „Was habe ich mir hierbei nur gedacht…?“

; Semikolon

Das Semikolon wird benutzt, um eine Anweisung abzuschließen.

Beispiel

```
int a = 13;
```

Hinweis

Ein vergessenes Semikolon hat einen Compiler-Error zur Folge. Die Fehlermeldung des Compilers ist oftmals sehr kryptisch und man (jedenfalls ich) kann nichts damit anfangen. Wenn eine Fehlermeldung immer wieder auftritt, sollte zuerst nach einem fehlenden Semikolon in der Zeile gesucht werden, die in der Fehlermeldung genannt wird. Ein Semikolon kann auch weit über oder unter der in der Fehlermeldung genannten Zeile fehlen! Das hätte ich mir komfortabler gewünscht, aber es ist eben so.

{} geschweifte Klammern

Die geschweiften Klammern sind einer der wichtigsten Bestandteile der C-Syntax. Beginner werden oft durch die vielen geschweiften Klammern abgeschreckt oder verwirrt. Auf eine öffnende “{” Klammer muss immer eine schließende “}” Klammer folgen. Es können jedoch auch mehrere öffnende Klammern verwendet werden (verschachtelt), und erst später dann die gleiche Anzahl schließende Klammern. Eine vergessene schließende Klammer kann zu kryptischen Fehlermeldungen des Compilers führen.

Ein guter Weg, um dem Vergessen von Klammern vorzubeugen, ist es, direkt eine öffnende und schließende Klammer hintereinander zu setzen und einfach ein paar Zeilenumbrüche einzufügen und den Code dann dort hineinzuschreiben. (Mittlerweile macht die IDE die schließende Klammer oft automatisch, wenn man eine öffnende setzt.)

Verwendungszwecke der geschweiften Klammern:

Funktionen

```
void meineFunktion(datatyp Argument){

    // Anweisungen
}
```

Schleifen

```
while (boolescher Ausdruck)
  {
     // Anweisungen
  }
-----------------------------------------------------------
do
  {
    // Anweisungen
  } while (boolescher Ausdruck);
-----------------------------------------------------------
for (Initialisation; Abbruchbedingung; Inkrementierung)
  {
     // Anweisungen
  }
```

Bedingungsabfrage

```
if (boolescher Ausdruck)
  {
     // Anweisungen
  }
else if (boolescher Ausdruck)
  {
     // Anweisungen
  }
else
  {
     // Anweisungen
  }
```

#define

`#define` ist eine hilfreiche C-Komponente, die es erlaubt, eine bestimmte Konstante oder einen Alias zu definieren. Der Vorteil ist, dass dieser Alias keinen Speicherplatz auf dem Chip benötigt, da der Compiler eine Referenz zur definierten Konstante/Alias während des Kompilierens durch den definierten Wert ersetzt wird. Der Alias dient der besseren Lesbarkeit des Codes.
Dies kann allerdings auch zu einigen negativen Nebeneffekten führen, wenn z.B. der Name einer definierten Konstante in einem anderen Variablennamen verwendet wird.
Um Konstanten zu definieren sollte der Typqualifikator ***const*** eingesetzt werden.

Syntax

```
#define NamederKonstante Wert
```

Beispiel

```
#define ledPin 3
// Der Compiler wird während des Kompilierens jede Referenz zu ledPin
// mit dem Wert 3 ersetzen
```

Hinweis

Nach dem `#define` wird kein Semikolon und auch kein "=" benötigt:

```
#define ledPin 3;     // dies ist ein Fehler
#define ledPin = 3    // dies ist ebenfalls ein Fehler
```

#include

`#include` wird benutzt um externe Software-Bibliotheken in das Programm einzubinden (siehe auch „Bibliotheken"), welche bereits einige Standardfunktionen beinhalten. `#include` erlaubt den Zugriff auf Standard C Bibliotheken und auch auf ARDUINO spezifische Bibliotheken. Eine Auswahl vieler Bibliotheken ist auf arduino.cc zu finden, und auch in vielen Foren im Internet, bei Anbietern von Hardware etc.

`#include` MUSS vor dem setup() eingefügt werden!

Zu beachten ist ferner, dass `#include` genau wie `#define` kein Semikolon am Ende benötigt!

Beispiel

Dieses Beispiel bindet eine Bibliothek ein, die benutzt wird, um Daten in den *flash*-Speicher anstatt in den RAM-Speicher zu legen. Dies spart Platz im RAM-Speicher, wenn dynamischer Speicher benötigt wird und macht große Datentabellen praktischer.

```
#include <avr/pgmspace.h>

prog_uint16_t meineKonstanten[] PROGMEM = {0, 21140, 702   , 9128,  0, 25764,
8456,0,0,0,0,0,0,0,0,29810,8968,29762,29762,4500};
```

#ifdef #else #endif

Beispiel-Sketch verfügbar: _ifdef__else__endif.ino

Manchmal ist es sinnvoll, während der Testphase des Sketches oder in einem Probeaufbau auf dem Steckbrett, etwas mehr Code zu haben, oder auch etwas anderen. Das wären beispielsweise zusätzliche Ausgaben auf der seriellen Schnittstelle, um das Finden von Fehlern leichter zu machen. Oder auch Taster, die auf dem Steckbrett nach +UB gehen, in der fertigen Schaltung aber nach GND. Oder zeitliche Abläufe, die im Testbetrieb schneller laufen sollen als in der fertigen Schaltung. Oder...

Sicher, man kann den Code dann ja einfach umschreiben. Aber praktisch ist das nicht. Funktioniert der Sketch doch nicht so, muss man wieder zusätzliche Codezeilen einfügen und so weiter.

Hier bietet sich die sogenannte **bedingte Compilierung** an. Das heißt, man „sagt" dem Compiler: Jetzt alles MIT dem zusätzlichen Code kompilieren, und dann für die fertige Schaltung OHNE diesen. Das lässt sich an einem oder zwei Beispielen sicher besser verstehen.

Beispiel 1: HIGH-aktive oder LOW-aktive Taster

Ich habe für mein Steckbrett eine kleine Leiterplatte, auf welcher sich acht Taster befinden und ein Schaltkreis 74HC14. Dieser entprellt die Taster gleich, sodass man sich im Sketch nicht darum kümmern muss. Drückt man einen Taster, geht das Signal am Ausgang dieses Schaltkreises von LOW nach HIGH. Das ist also genau umgekehrt, wie man sonst die Taster DIREKT am ARDUINO anschließt. Um nun einfach den Code wahlweise auf dem Steckbrett laufen zu lassen, oder dann auch in der fertigen Schaltung, kann man es folgendermaßen machen:

```
#define Feature_Taster_low    // nicht auskommentiert

#ifdef  Feature_Taster_low    // Tasterfunktion bedingt compilieren
  #define ON  LOW             // Taster nach GND
#else
  #define ON  HIGH            // Taster nach +UB
#endif
```

Benutzt man den Codeschnipsel in dieser Art, dann wird die Konstante „ON“ mit dem Wert „LOW“ belegt. Kommentiert man die erste Zeile aus, dann wird die Konstante „ON“ mit dem Wert „HIGH“ belegt.

```
//#define Feature_Taster_low        // auskommentiert
```

Nun kann im weiteren Verlauf der Zustand des Tasters immer mit

```
if (digitalRead(taster) == ON)
```

abgefragt werden. In beiden Varianten kommt das richtige Ergebnis. Zauberei, oder?

Beispiel 2: serielle Ausgabe, um Fehler zu finden (debugging)

Schreibt man einen umfangreichen Sketch, sind fast immer Fehler im Code. Leider ist das so. Diese Fehler muss man auch finden. Das kann manchmal etwas schwierig werden. Hier ist die Ausgabe verschiedener Werte, Daten, Variablen, Sensorwerte über den seriellen Monitor eine super Hilfe. Läuft alles, was dann? Den zusätzlichen Code wieder aufwändig löschen? Ihn drin lassen? Oder gar gegebenenfalls neu einfügen, wenn es doch nicht richtig geht...Hier bietet sich auch wieder die bedingte Compilierung an. Man fügt einfach VOR dem setup() die folgende Zeile an:

```
#define debug           // bedingte Compilierung
```

Nun kann man im normalen Programm einfach Ausgaben auf die serielle Schnittstelle mit einfügen, wie man möchte:

```
#ifdef debug
   Serial.println(sensor1)    // Wert von Variable sensor1 ausgeben
#endif
```

Wenn man die Zeile `#define debug` auskommentiert, dann wird die Zeile mit dem `Serial.println` NICHT in den ARDUINO eingebrannt. Man spart so wertvollen Flash-Speicher und Rechenzeit ein.

Datentypen

Datentypen geben an, welches Format oder Art die unter dem entsprechenden Namen gespeicherten Werte haben, bzw. haben dürfen. Hier gibt es bedingt aus dem C/C++ Sprachraum noch weitere Bezeichnungen, die ich hier als „alternative“ Bezeichnungen angebe. Sie sind nicht so leicht verständlich und nur der Vollständigkeit halber erwähnt. Eventuell hat man ja mal einen Code eines anderen Programmierers, der diese Bezeichner verwendet. So steht man nicht auf dem Schlauch, und weiß, was gemeint ist.

void

void wird als Keyword nur zur Deklaration von Funktionen eingesetzt. Es zeigt an, dass die Funktion keine Rückgabewerte an die aufrufende Funktion zurück liefert.

```
// die Aktionen werden in den Funktionen "setup" und "loop" ausgeführt,
// aber keine Werte werden an das aufrufende,
// übergeordnete Gesamtprogramm übergeben

void setup(){
// ...
}

void loop(){
// ...
}
```

byte (uint8_t)

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

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

int (int16_t)

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

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

Hinweis:

Integer Variablen werden bei Überschreiten der Limits 'überrollen'. Zum Beispiel wenn x = 32767 und eine Anweisung addiert 1 zu x, x = x + 1 oder x++, wird 'x' dabei 'überrollen' und den Wert -32768 annehmen.

short

Ein short ist ein 16Bit Datentyp und reicht von -32768 bis 32767. Bei den ATMEGA und ARM-basierten ARDUINO's ist ein short also mit dem int identisch. Es sollte besser ein int verwendet werden.

```
short someVariable = 1500; // deklariert 'someVariable'
// als einen 'short' Datentyp
```

word (uint16_t)

Ein word speichert eine vorzeichenlose 16-bit Zahl, von 0 bis 65535. Genauso wie eine unsigned int.

```
word w = 60000;   // Variable zu groß für int, aber word ist OK!
```

unsigned int (uint16_t)

Vorzeichenlose Integer-Variable. Wert kann von 0 bis 65535 sein.

```
unsigned int variable = 40000;
```

long (int32_t)

Datentyp für lange Integer mit erweiterter Größe, ohne Dezimalkomma, gespeichert in einem 32-bit Wert in einem Spektrum von -2.147.483.648 bis 2.147.483.647

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

unsigned long (uint32_t)

Vorzeichenlose long-Variable. Wert 0 bis 4.294.967.295

```
unsigned long variable = 3000000;
```

float

Ein Datentyp für Fließkommawerte oder Zahlen mit Nachkommastelle. Fließkommawerte haben einen größeren Wertebereich als Integer und werden als 32bit-Wert mit einem Spektrum von -3.4028235E+38 bis 3.4028235E+38 gespeichert.
Floats haben eine Genauigkeit von nur 6-7 Dezimalstellen. Das bedeutet die Gesamtzahl der Ziffern, nicht die Zahl rechts vom Dezimalpunkt.

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

Hinweis:

Fließkommazahlen sind nicht präzise und führen möglicherweise zu merkwürdigen Resultaten, wenn sie verglichen werden. Außerdem sind Fließkomma-Berechnungen viel langsamer als solche mit Integer-Datentypen. Berechnungen mit Fließkommawerten sollten nach Möglichkeit vermieden werden. Siehe auch hier.

double

Eine Fliesskommazahl mit doppelter Auflösung als float. Allerdings ist bei den UNO's, NANO's und andere ATMEGA-basierten Boards die Auflösung genau identisch mit der von float. Man hat also keinen Gewinn an Genauigkeit.

Beim ARDUINO DUE ist double eine 8 Byte Zahl (64bit) und hat somit die doppelte Auflösung als float.

Hinweis

Wenn man sich Code aus anderen Projekten „borgt" und in das eigene Projekt einfügt, oder ganze Sketche, sollte man vorher schauen, ob die höhere Präzision der double im eigenen Projekt gebraucht wird oder nicht. Ansonsten kann es zu ungenauen Ergebnissen führen. Da ist dann nur der Einsatz eines DUE möglich, oder aber anderes Coding.

boolean (bool)

Der Datentyp boolean kann zwei Zustände haben: true oder false.
Die Datentypen kann man angelehnt an die digitale Elektronik sehen, in denen man auch von 0 und 1 redet (oder HIGH und LOW). Siehe auch hier in der Referenz.

```
boolean buzzerflag = false;                 // Flag zur Erkennung
bool buzzerflag = false;                    // das gleiche…
```

Ich benutze sehr gern sogenannte Flags, um im Programm zu markieren, ob ein gewisser Zustand gerade aktiv ist oder nicht. Dazu verwende ich den Datentyp boolean.

false: bedeutet immer 0 (Null)
true: bedeutet 1, ABER: auch jeder andere Wert außer Null ist true.

boolean ist ein nicht standardmäßiger Typalias für bool, welcher von Arduino definiert wurde. Es wird empfohlen, stattdessen den Standardtyp bool zu verwenden, der identisch ist.

Obwohl mit 'bool' nur 1 Bit gespeichert werden kann, belegt es im Arduino aber doch ein ganzes Byte (8 Bit).

Array's

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

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

Genauso ist es möglich ein Array zuerst mit Datentyp und Größe zu deklarieren und später einer Indexposition einen Wert zu geben.

```
int myArray[5]; // deklariert Datentyp 'integer' als Array mit 5
Positionen (von 0 bis 4)
myArray[3] = 10; // gibt dem 4. Index den Wert 10
```

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

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

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

```
int ledPin = 10; // LED auf Pin 10
byte flicker[] = {180, 30, 255, 200, 10, 90, 150, 60};
// Array mit 8 verschiedenen Werten

void setup(){
pinMode(ledPin, OUTPUT); // Setzt den OUTPUT Pin
}//

void loop(){
for(int i=0; i<7; i++) // Schleife gleicht der Anzahl
  { // der Werte im Array
  analogWrite(ledPin, flicker[i]); // schreibt den Indexwert auf die LED
  delay(200); // 200ms Pause
  }
}
```

char (int8_t)

Ein Datentyp, der 1 Byte Speicher braucht um einen Buchstaben zu speichern. Buchstaben müssen in einfache Anführungszeichen gesetzt werden: 'A'. Für mehrere Buchstaben (Wörter) sind doppelte Anführungszeichen erforderlich: "ABC".
Wie auch immer – Buchstaben werden immer als Byte-Zahl gespeichert (im ASCII-Code).
Der char-Datentyp ist ein Vorzeichen-Typ, was bedeutet, dass Buchstaben immer von -128 bis 127 codiert werden.

```
  char myChar = 'A';
  char myChar = 65;      // beides bedeutet das gleiche
```

unsigned char (uint8_t)

Ein **unsiged char** speichert ein vorzeichenloses 8-bit Zeichen mit dem Wertebereich von 0 bis 255. Identisch mit **byte**. Um konsistent mit dem ARDUINO Programmierstil zu sein, sollte man den Datentyp **byte** verwenden.

```
unsigned char meinZeichen = 240;
```

string (Datentyp & char-Array)

Text-Strings können auf zwei Arten gebildet werden:

1. Man kann den Datentyp „String“ (String-Object) verwenden.

2. Man bildet ein Array von Typ „char“. Dieses muß mit einem Null-Terminator beendet werden. Das macht der Compiler entweder selbst (unteres Beispiel Str2), oder man hängt diese ‚\0‘ selbst an (Beispiel Str3). Hier geht es um das Array vom Typ char.

Für weiterführende Details über das ***String-Object***, (Punkt 2), welches mehr Möglichkeiten bietet, aber auch mehr Speicher braucht, siehe hierzu ***String Objects***. Diese Möglichkeiten sind sehr umfangreich, und bedürfen Einarbeitungszeit.

Die folgenden Beispiele sind alles gültige Definitionen von char-arrays:

```
char str1[15];  // nimmt 15 Objekte des Datentyps char auf

char str2[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o',};

char str3[8] = {'a', 'r', 'd', 'u', 'i', 'n', 'o', '\0'};

char str4[ ] = "arduino";

char str5[8] = "arduino";

char str6[15] = "arduino";
```

Möglichkeiten der oben genannten String-Deklaration:

1. Array von alphanummerischen Zeichen, ohne es zu initialisieren ➔ siehe str1

2. Deklariere ein Array von alphanummerischen Zeichen (mit einem zusätzlichen Zeichen) und der Compiler wird die notwendige Null selbst hinzufügen ➔ siehe str2

3. Geben sie die null-termination explizit mit an ➔ siehe str3

4. Initialisiere eine string-Konstante in Anführungszeichen. Der Compiler bildet die erforderliche String-Größe selbst, fügt die null-termination hinzu, sodass alles passt. ➔ siehe str4

5. Initialisiere das Array selbst mit einer genauen Größe und string-Konstanten ➔ siehe str5

6. Initialisiere das Array selbst und lass zusätzlichen Platz für größere, längere Wörter ➔ siehe str6

null termination / null character

Normalerweise werden char-Arrays mit einem “null Character” beendet (ASCII Code 0). Dies gibt Funktionen wie `Serial.print()` die Möglichkeit, das Ende des Strings zu erkennen. Ohne diesen würde die Funktion weitere folgende Bytes lesen, die nicht Teil des gewünschten Strings sind.

Das bedeutet, dass der String immer Platz haben muss für ein zusätzliches Zeichen als der Text enthält, der im String gespeichert werden soll. Deshalb hat Str2 und Str5 8 Zeichen, obwohl das Wort „ARDUINO“ nur 7 Buchstaben hat. Die letzte freie Position wird automatisch mit dem „null character“

gefüllt. str4 wird automatisch auf die Größe von 8 Zeichen gebracht, einen für die Extra-Null. In str3 haben wir selbst den „null character“ angefügt (wird '\0’ geschrieben).

Beachten Sie, dass es möglich ist, einen String zu haben OHNE den null character (z.B., wenn Sie str2 mit der Länge von 7 anstatt von 8 deklariert haben). Das wird das korrekte Arbeiten der meisten Funktionen zerstören, welche Strings nutzen. Das sollte man also unterlassen. Wenn Sie also seltsames Verhalten im Zusammenhang der Benutzung von Strings feststellen (z.B. arbeiten mit Buchstaben, welche nicht im String vorkommen), dann könnte das die Ursache sein.

Einfache oder doppelte Anführungszeichen?

Strings (ganze Wörter, Sätze) werden immer definiert in doppelten Anführungszeichen ("Abc"), einzelne Buchstaben dagegen in einfachen Anführungszeichen ('A').

Einbinden langer Strings (Sätze)

Lange Sätze oder Wörter können Sie so einbinden:

```
char myString[] = "This is the first line"

" this is the second line"

" etcetera"; // Semikolon am Ende !!
```

String-Arrays

Oftmals ist es sehr komfortabel, wenn man mit langen Zeichenketten, Wörtern oder Sätzen arbeitet (z.B. ein Projekt mit LCD-Anzeige), dann ein String-Array zu bilden. Da Strings ja selbst schon ein Array sind, handelt es sich hier also um ein zweidimensionales Array.

In dem unten angeführten Code-Beispiel bedeutet und bezeichnet der Stern hinter dem Datentyp char, dass es sich hierbei um einen String mit “Pointern” handelt. Da alle Arraynamen eigentlich Pointer sind, ist das notwendig, um ein Array von Arrays zu bilden (Zweidimensionales Array). Pointer sind ein Teil der eher „esoterischen“ Teile der C-Programmiersprache und schwer zu verstehen für Anfänger. Aber in diesem Codebeispiel ist es nicht notwendig, die Benutzung von Pointern zu verstehen, um sie effizient einzusetzen (ideal für Auswahl unterschiedlicher Anzeigetexte in LCD’s aufgrund von Entscheidungen im Programm [A.N.]).

```
char* myStrings[]={"This is string 1", "This is string 2", "This is string 3",
"This is string 4", "This is string 5","This is string 6"};

void setup(){            // Setup wird einmal durchlaufen
Serial.begin(9600); // serielle Schnittstelle 9600Bd initialisieren
}

void loop(){       // Hauptprogramm wird in Schleife durchlaufen
for (int i = 0; i < 6; i++){  // 6 mal ausführen
   Serial.println(myStrings[i]);
   delay(500);     // Pause 0,5 Sekunden
   }
}
```

String-Objects

Das ist ein sehr umfangreiches Kapitel, deshalb ist ihm auch ein extra Abschnitt gewidmet. Siehe „String Objects“

Datentypen-Bezeichner z.B. UL, U, L etc.

Danke hierzu an Mitglied ***Scheams*** im ARDUINO-Forum (Quelle 10) und Ulrich (Email) für die genauere Formulierung einer korrekten Erklärung.

Manchmal kann es sinnvoll sein, die verwendeten Datentypen im Code zu bezeichnen. Hier ist es nicht notwendig:

```
unsigned int var = 5U; // unnötig ->
unsigned int var = 5;
```

Wenn man aber sowas hat:

```
int var1 = (15000 * 20000) / 9375;
```

Das Ergebnis kann man per Taschenrechner leicht auf 300.000.000 / 9.375 = 32.000 berechnen.

Was man dabei wissen muss, ist, dass die Arduino IDE alle auftretenden Zahlen auf der rechten Seite des Gleichheitszeichens als Datentyp int (16 bit, Wertebereich -32.768 bis +32.767) betrachtet. Multipliziert man nun 15.000 * 20.000 ergibt sich 300.000.000. Das liegt außerhalb des int-Wertebereichs und es kommt zum Überlauf - die sich ergebenden Ziffern sind falsch. Diese werden dann noch durch 9.375 dividiert. Es ist eine gute, nicht ganz einfache Übung, diesen Vorgang nachzuvollziehen, indem man die Berechnung in Binärzahlen darstellt und den Überlauf durch Abschneiden der Bits vor der 16. Stelle nachvollzieht. Das (falsche) Ergebnis ist -2.

Wie kommt man zum richtigen Ergebnis? Man denkt an "Überlauf" und "Der Wertebereich von int ist zu klein" und könnte dies versuchen:

```
long var2 = (15000 * 20000) / 9375;
```

Aber auch dies wird ein falsches Ergebnis zur Folge haben, da der 16-bit-Überlauf nicht links, sondern rechts des Gleichheitszeichens bei der Multiplikation auftritt.
Die Lösung ist, dem Compiler mitzuteilen, dass mindestens eine Zahl rechts vom Gleichheitszeichen von einem größeren Datentyp als int ist. Das kann man tun, indem man an die erste Zahl das Suffix L (bedeutet long) anhängt.

```
int var3 = (15000L * 20000) / 9375;
```

Korrektes Ergebnis: 32.000

Dann wird alles, was auf der rechten Seite folgt, im Wertebereich long (-2.147.483.648 bis 2.147.483.647), also mit 32 Bit Breite berechnet. Da passt das Zwischenergebnis 300.000.000 locker rein. Darauf erfolgt die Division durch 9.375. Da aber auf der linken Seite `int var3` steht, werden die führenden 16 Bit des 32-bit-Ergebnisses bei der Zuweisung auf die Variable var3 abgeschnitten. Sofern das Ergebnis wie hier im Wertebereich von int liegt, stimmt es. Ist es grösser, natürlich nicht.
Dann müsste man links `long var3` schreiben.

Will man das Ganze auf vorzeichenlose Zahlen übertragen, gilt sinngemäß das Gleiche. Als Suffix ist dann aber UL (bedeutet Unsigned Long) statt L anzuhängen.

Zur Integer-Arithmetik noch eine Besonderheit.

```
int var4 = 15000 / 9375 * 20000;
```

Hier tritt auf der rechten Seite KEIN Überlauf auf. Aber was wird herauskommen? Zuerst wird 15000 / 9375 berechnet. Auf dem Taschenrechner kommt 1,6 heraus. Der Arduino rechnet hier natürlich mit Integer Zahlen, das Zwischenergebnis ist 1. var4 wird daher am Ende 20.000 enthalten.
Der beträchtliche Fehler entsteht bei der ganzzahligen Division.

Man sieht, dass man bei Integer-Arithmetik ganz schön aufpassen muss. Die bisweilen komplizierte Beachtung der Wertgrenzen und der Berechnungsreihenfolge treten in den Hintergrund, wenn Variablen des Typs float verwendet werden. Man erkauft sich diese Bequemlichkeit aber mit mehr Speicherverbrauch, wesentlich mehr Rechenzeit und in vielen Fällen mehr Ungenauigkeit. Wo immer möglich, sollte man Integer-Arithmetik verwenden. Man fühlt sich dann auch mehr als "echter" Programmierer.

Konstanten

Konstanten sind vordefinierte Variablen in der ARDUINO-Sprache. Sie machen die Programme einfacher zu lesen und zu verstehen. Wir klassifizieren die Konstanten in verschiedenen Gruppen.

Logische Pegel, true und false (Bool'sche Konstanten)

Eine Variable des Typs bool (boolean) enthält einen von zwei Werten, true oder false. Jede bool-Variable belegt ein ganzes Byte an Speicher.

false

false ist definiert als Logisch 0 (zero)

true

true wird oft mit Logisch 1 definiert, was auch richtig ist. ABER true hat noch eine weitergehende Definition. Jede Integerzahl (Datentyp int), die nicht 0 ist, ist true. (in der Booleschen Algebra)

Also sind -1, 2, 200, -200 ebenfalls definiert als true, in der Booleschen Algebra.

Beachten Sie, dass true- und false-Konstanten in Kleinbuchstaben geschrieben werden, während HIGH, LOW, INPUT, OUTPUT in Großbuchstaben geschrieben werden!

Pin-Zustände HIGH und LOW

Wenn auf einen Pin des ARDUINO geschrieben oder gelesen werden soll, dann gibt es nur zwei mögliche Werte: HIGH und LOW. (auch analogWrite schreibt nur HIGH und LOW, aber eben als pulsweitenmodulierte Spannung…)

Pegel in der Digitaltechnik

In der digitalen Elektronik ist oft von HIGH-Pegel oder LOW-Pegel die Rede. In der Tat sind das die beiden wichtigsten Zustände. So auch bei ARDUINO's.

Der Einfachheit halber schreibe ich hier in diesem Buch nur HIGH (oder +UB) und LOW (oder GND).

Die Pegeldefinitionen gemäß der Auflistung rechts haben jedoch einen gewissen Wertebereich. Bei ARDUINO's haben wir meistens den TTL-Pegel (die meisten Standard-Arduinos) oder LVTTL. Warum ist das so? Nun, es handelt sich um

gängige Logikpegel (alle Angaben in Volt)

	Eingang		Ausgang	
Technologie	Low (V_{IL})	High (V_{IH})	Low (V_{OL})	High (V_{OH})
TTL 5 V	≤ 0,8	≥ 2,0	≤ 0,4	≥ 2,4
CMOS 5 V	≤ 1,5	≥ 3,5	≤ 0,5	≥ 4,44
LVTTL 3,3 V	≤ 0,8	≥ 2,0	≤ 0,4	≥ 2,4
CMOS 2,5 V	≤ 0,7	≥ 1,7	≤ 0,2	≥ 2,3
CMOS 1,8 V	≤ 0,7	≥ 1,17	≤ 0,45	≥ 1,2
ECL	≤ −1,4	≥ −1,2	?	?
RS-232(*)	−15 bis −3	+3 bis +15	−15 bis −5[1]	+5 bis +15[1]
HTL 10...30 V	≤ 0,2 × U_B	≥ 0,6 × U_B		≈ U_B
(*) = negative Logik, d. h. low=1, high=0				

Logikpegel (Quelle: Wikipedia)

elektronische Bauelemente, die alle eine gewisse Toleranz, Temperaturabhängigkeit, Verluste, Innenwiderstände haben.

HIGH

Die Bedeutung von HIGH (in Bezug auf einen Schaltkreis-Pin) ist etwas unterschiedlich, je nachdem ob der Pin als INPUT oder OUTPUT gesetzt ist. Wenn ein Pin mit **pinMode** als INPUT konfiguriert ist, und mit **digitalRead** gelesen wird, dann wird der ARDUINO ein HIGH melden, wenn die anliegende Spannung am Pin 3V oder mehr beträgt.

Ein Pin kann auch als INPUT konfiguriert werden mit **pinMode** und danach auf HIGH gesetzt werden mit **digitalWrite**. Das schaltet den internen Pullup-Widerstand von ca. 20kOhm ein. Das setzt den Pin auf HIGH, wenn er nicht beschaltet ist. Um ein LOW zu erzeugen, muss der Pin extern auf LOW (Potential GND) gezogen werden, z.B. durch einen Taster, Sensor etc. Die zusammengehörigen Befehle

```
pinMode(4, INPUT);
digitalWrite (4, HIGH);
```

können ersetzt werden durch: `pinMode(4, INPUT_PULLUP);` was kürzer und eleganter ist.

Wenn ein Pin als OUTPUT konfiguriert ist mit dem Befehl `pinMode` und auf HIGH gesetzt wird mit `digitalWrite`, dann liegt eine Spannung von 5V (bzw. die ARDUINO-Betriebsspannung) am Pin an. Der Ausgang kann nun Strom treiben gegen Masse, z.B. um eine LED mit Vorwiderstand zum Leuchten zu bringen. Der Maximalstrom des Ausganges ist hier zu beachten und die gesamte maximale Verlustleistung (Datenblatt) des gesamten Schaltkreises. Auch die Verbindung mit einem anderen Pin ist möglich, wenn dieser als INPUT gesetzt ist.

LOW

Die Bedeutung von LOW ist auch unterschiedlich, je nachdem ob es sich um einen Pin als INPUT oder OUTPUT handelt. Ist ein Pin als INPUT konfiguriert mit pinMode und wird mit digitalRead gelesen, dann wird der ARDUINO ein LOW melden, wenn die Spannung 2V oder kleiner ist (Siehe „Pegel in der Digitaltechnik“).

Ist der Pin als OUTPUT konfiguriert, so kann er Strom treiben nach GND und auch nach +UB. Man kann also zum Beispiel eine LED mit Vorwiderstand sowohl vom Pin nach GND als auch nach +UB schalten. Der Unterschied ist folgender:

```
digitalWrite(LED, HIGH;
// LED nach +UB leuchtet nicht
// LED nach GND leuchtet

digitalWrite(LED, LOW);
// LED nach +UB leuchtet
// LED nach GND leuchtet nicht
```

Digitale Pins konfigurieren

Aliase für Pins benutzen

Um den Code lesbar zu gestalten, ist es sinnvoll für Pins (also Eingänge und Ausgänge) verständliche und eindeutige Namen zu verwenden. Das ist durch mehrere Varianten möglich.

```
#define LED 13
```

Ersetzt quasi den Text „LED“ mit dem Wert 13. Siehe Erklärung zu #define hier.

```
const int LED = 13;
```
Eine Konstante braucht ebenfalls keinen RAM.

```
int LED = 13;
```
Viele Beispiele, auch in der IDE, benutzen der Einfachheit halber int für pin-Definitionen. KANN man machen, aber speichersparend und besserer Stil ist const oder define.

Also, noch mal zum Vergleich:	`#define LED 13`	➔	kein RAM-Verbrauch
	`const int LED = 13;`	➔	kein RAM-Verbrauch
	`int LED = 13;`	➔	2 Byte RAM-Verbrauch

Die Pins eines ARDUINO können als digitaler Eingang oder Ausgang und auch als analoger Eingang benutzt werden. Einen „echten“ analogen Ausgang gibt es so nicht, man kann sich aber anders helfen, wenn man so etwas benötigt. Dazu später mehr.

Pins konfiguriert als Eingänge (INPUT)

ARDUINO (Atmega) Pins, die als INPUT konfiguriert sind mit pinMode, haben einen hohen Innenwiderstand. (high-impedance state). INPUT-Pins verbrauchen extrem wenig Strom aus der Schaltung, vergleichbar mit einem Widerstand von 100 Megaohm oder mehr.

Wenn ein Pin als INPUT konfiguriert ist, dann will man ihn oft auf einem definierten Potential haben. Das wird erreicht mit einem externen Pulldown-Widerstand vom Pin nach Masse. Dann liegt der Pin im unbeschalteten Zustand auf LOW. Das Gegenteil wird erreicht bei Benutzung eines externen Pullup-Widerstandes vom Pin nach +UB oder durch Aktivieren des internen Pullup mit INPUT_PULLUP.

Analoge Eingänge, welche mit dem Befehl `analogRead` abgefragt werden sollen, müssen nicht mit `PinMode` konfiguriert werden, denn sie sind immer als Input vordefiniert. (Es schadet aber auch nicht.)

```
pinMode(1, INPUT);      // Setzt den Digital-Pin 1 als Eingang
```

Pins konfiguriert als INPUT_PULLUP (internen Pullup-Widerstand einschalten)

Der ATMega-Chip, welcher sich auf den ARDUINO-Boards befindet, hat interne Pullup-Widerstände, die Sie zuschalten können, wenn nötig. Wenn sie diese den extern in die Schaltung zu bringenden Widerstände vorziehen, dann können sie mit INPUT_PULLUP als Argument im Befehl pinMode() eingeschaltet werden.

```
pinMode(1, INPUT_PULLUP); // setzt Pin1 als Eingang und aktiviert Pullup
```

Pins konfiguriert als Ausgänge (Outputs)

Pins, welche als Ausgänge konfiguriert sind, befinden sich in einem Zustand mit geringem Innenwiderstand (low-impedance state). Das bedeutet, dass sie einen bestimmten Strom in andere Teile der Schaltung treiben können. Bei den AVR’s im ARDUINO ist dieser Strom maximal 40mA, welcher

getrieben (gegen Masse) oder gezogen (nach +UB) werden kann. Das macht sie nützlich, um LED's zum Leuchten zu bringen, oder Relais zu schalten, oder auch andere Teile der Schaltung anzusteuern.

Siehe hierzu auch: Belastbarkeit

```
pinMode(13, OUTPUT);      // Setzt den Digital-Pin 13 als Ausgang
```

Integer Konstanten

Konstanten mit dem Datentyp int speichern Integer-Zahlen (-32768 bis 32768) und können nicht geändert werden. Sie erleichtern das Lesen des Codes.

```
const int Sonntag = 7;        // Sonntag ist der siebte Tag der Woche
```

In diesem Fall wird dem Compiler der ARDUINO-IDE gesagt, er soll jedes Mal, wenn das Wort „Sonntag" erscheint, dieses mit der Zahl 7 ersetzen. Wie man sieht, kann man hier viel tun, um den Code einfacher lesbar zu gestalten. Die Zahlen können auch im Binärcode, Oktal, hexadezimal etc. angeben. Wen das näher interessiert, dem möge die Original-Referenz in Englisch helfen.

Floating point Konstanten (Fließkomma)

Hier gilt das eben gesagte, halt für Fließkommazahlen.

```
const float pi = 3.14;        // Konstante „pi" erhält den Wert 3,14
```

Fließkomma-Konstanten können auch in wissenschaftlicher Darstellung eingegeben werden. Hierzu kann das „E" wie auch "e" als Indikator des Exponenten verwendet werden

```
const float test = 5.87E5           // entspricht 587000
const float test2 = 4.3e-3          // entspricht 0.0043
```

Verzweigungen und Schleifen

for-Schleifen

For-Schleifen werden benutzt um bestimmte Aktionen wiederholt auszuführen, z.B. um ein Array mit Daten eines Sensors zu füllen.
Eine for-Schleife benötigt drei Parameter:

```
for (Initialisierung; Bedingung; Inkrement bzw. Dekrement)
{
 // Anweisungen
}
```

Die Initialisierung geschieht nur einmal. Danach wird die Bedingung geprüft und falls diese Bedingung stimmt, wird das Inkrement (i++) ausgeführt d.h. der Zählerwert wird erhöht oder es wird das Dekrement (i--) ausgeführt, d.h. der Wert wird erniedrigt.

```
// Eine LED dimmen mithilfe von PWM

int PWMpin = 10; // LED ist an Pin 10 angeschlossen

void setup()
{
  // es wird kein setup benötigt
}

void loop()
{
   for (int i=0; i <= 255; i++){
      analogWrite(PWMpin, i);
      delay(10);
   }
}
```

Die `for`-Schleife in C ist viel flexibler als for-Schleifen in anderen Programmiersprachen. Das Inkrement/Dekrement kann aus jeder C konformen Rechenoperation bestehen, z.B. einer Multiplikation, Addition oder sogar Potenzierung:

```
for(int x = 2; x < 100; x = x * 1.5)
{
   Serial.println(x);
}
```

if (Bedingung) und ==, !=, <, > (Vergleichsoperatoren)

`if` in Verbindung mit einem Vergleichsoperator prüft, ob eine bestimmte Bedingung erfüllt ist, z.B. ob der Wert einer Variablen einen festgelegten Wert überschritten hat.
Eine `if`-Abfrage kann z.B. so aussehen:

```
if (Variable > 50) Serial.print(„Variable ist größer als 50”);
```

Diese Kurzversion funktioniert nur, wenn nur eine einzige Anweisung der Bedingung folgt.

Die normale Schreibweise ist im Folgenden aufgeführt:

```
if (Variable > 50)
 {
  // Führe eine Aktion aus. Wenn der Wert in Variable die 50 überschritten hat,
  // z.B. setze den Wert von Variable zurück
 }
```

Wenn die Bedingungen in den runden Klammern erfüllt sind, wird der Code in den geschweiften Klammern ausgeführt. Wenn die Bedingung nicht erfüllt ist, wird der Code in den geschweiften Klammern einfach übersprungen.

Vergleichsoperatoren

x == y (x entspricht y)
x != y (x ungleich y)
x < y (x ist kleiner als y)
x > y (x ist größer als y)
x <= y (x ist kleiner oder gleich y)
x >= y (x ist größer oder gleich y)

Zu beachten ist, dass == verwendet wird, um zu prüfen ob eine Variable einem Wert entspricht!

Verkürzte Schreibweise

if (wert != 0)	ist gleichbedeutend mit	if (wert)
if (wert == 0)	ist gleichbedeutend mit	if (!wert)

if / else

`if/else` erlaubt eine bessere Kontrolle über den Programmablauf als die simple `if`-Abfrage, da mehrere Bedingungen geprüft werden können.
Eine Anwendung könnte z.B. so aussehen: Die Spannung an einem analogen Eingang wird gemessen, in einen Wert umgewandelt und es soll eine Aktion ausgeführt werden, wenn der Wert kleiner als 500 ist.

```
if(analogRead(5) < 500)
 {
  // Aktion A
 }
else  // wenn nicht, dann tue folgendes
 {
  // Aktion B
 }
```

Wenn der Wert kleiner als 500 ist, wird Aktion A ausgeführt, andernfalls wird Aktion B ausgeführt.
`else` und `if` können auch kombiniert werden, dabei wird solange zur nächsten `else if`-Anweisung gesprungen bis eine Bedingung erfüllt ist:

```
if (analogRead(5) < 500)
 {
  // Aktion A
 }
else if (analogRead(5) >= 1000)
 {
```

```
  // Aktion B
 }
else
 {
  // Aktion C
 }
```

switch / case Anweisungen

Ähnlich wie `if`-Anweisungen erlauben die `switch/case`-Anweisungen die Ausführung bestimmter Abläufe zu kontrollieren. Eine `switch`-Anweisung vergleicht den Wert einer Variablen mit den Werten aus den `case`-Anweisungen. Wenn eine Übereinstimmung gefunden wird, wird das Programm ab der jeweiligen `case`-Anweisung ***bis zum nächsten*** `break` ausgeführt.

Syntax

```
switch (var) {
  case label1:
    // Anweisungen
    break;
  case label2:
    // Anweisungen
    break;
  default:
    // Anweisungen
}
```

Parameter

var: eine Variable deren Wert zwischen den einzelnen case-Anweisungen verglichen wird
label: ein Wert der mit var verglichen wird

Beispiel

```
switch (var) {
    case 1:
      //etwas tun wenn var == 1
      break;
    case 2:
      //etwas tun wenn var == 2
      break;
    default:
      // wenn nichts zutrifft, führe default (standard) code aus
      // default ist optional, sollte aber immer hinzugefügt werden
  }
```

Range switch case

Man kann in der case auch einen Bereich unterbringen, um sich so viele verschiedene case zu ersparen. Dies stammt aus Quelle [26] – danke an hotsystems. Dieser „Range“ (englisch für „Bereich“) kann ein numerischer oder auch ein ASCII-Bereich sein und wird folgendermaßen verwendet:

```
case 23 ... 26:
      // tu etwas wenn var zwischen 23 und 26

case 'A' ... 'Z':
      // tu etwas wenn ASCII-Zeichen zwischen A und Z
```

Zu beachten ist hier die korrekte Schreibweise! Vor und nach den drei Punkten muss ein Leerzeichen sein! Auch dürfen sich die Bereiche nicht überlappen, wenn mehrere Range benutzt werden.

while - Schleifen

Eine `while`-Schleife wiederholt den Code innerhalb der geschweiften Klammern bis die Bedingung in den runden Klammern nicht mehr wahr ist bzw. nicht mehr zutrifft. Der Wert muss geändert werden, sei es innerhalb der Schleife durch eine Variable die inkrementiert wird oder durch den Druck eines Tasters, um die Schleife zu verlassen.

Syntax

```
while(Bedingung)
 {
  // Anweisung(en)
 }
```

Parameter

eine Bedingung die entweder wahr oder falsch sein kann

Beispiel

```
var = 0;
while(var < 200)
 {
  // wiederhole etwas 200 mal
  var++;          // erhöhe var jedes Mal um 1
 }
```

do – while

Die `do`-Schleife verhält sich genau wie die `while`-Schleife, nur, dass die Bedingung immer am Ende der Schleife geprüft wird. Das bedeutet, dass die Schleife immer mindestens einmal durchlaufen wird.

Syntax

```
Do
 {
    // Anweisung(en)
 }
while (Bedingung);
```

Beispiel

```
do
 {
  delay(50);           // warte 50 Millisekunden
  x = readSensors();   // Lese die Sensoren aus

 }
while (x < 100);
```

break

`break` wird benutzt, um eine `do, for oder while`-Schleife zu verlassen, ohne dass die Bedingung zum Schleifenabbruch erfüllt ist.

```
for (x = 0; x < 255; x ++)
 {
  digitalWrite(PWMpin, x);
  sens = analogRead(sensorPin);
```

```
 if (sens > Grenzwert) {  // verlasse  Schleife, wenn der Sensor einen größeren Wert
                          // als den Grenzwert liefert
       x = 0;
       break;
    }
    delay(50);
 }
```

continue

Die Anweisung `continue` springt ans Ende der aktuellen Schleifenanweisungen (`do, for oder while` Schleife). Die Schleife setzt mit der Prüfung der Schleifenbedingung fort und führt entsprechend der Schleife die nächste Iteration aus.

```
for (x = 0; x < 255; x++)
 {
    if (x > 40 && x < 120)
     {
       // überspringe die Anweisungen weiter unten, wenn x zwischen 40 und 120 liegt
       continue;
     }
    digitalWrite(PWMpin, x);
    delay(50);
 }
```

return

Beendet eine Funktion und gibt der aufrufenden Funktion einen Wert zurück, wenn dies gewünscht ist.

Syntax

return;
return wert; // beide Formen sind korrekt

Parameter

wert: beliebige Variable oder Konstante

Beispiel

Eine Funktion, die den Sensorwert mit einem Grenzwert vergleicht

```
int checkSensor(){
    if (analogRead(0) > 400)
     {
       return 1;
     }
    else
     {
       return 0;
     }
}
```

Es gibt auch die Möglichkeit, bei nur einer Anweisung hinter if oder else, diese verkürzt so zu schreiben:

```
int checkSensor(){
    if (analogRead(0) > 400) return 1;
    else return 0;
}
```

goto

Leitet den Programmfluss zu einer bestimmten Sektion im Programmcode, welcher im Label definiert ist.

Syntax

```
goto label; // leitet den Programmfluss zur Sektion "label"

label:
// hier weitere Anweisungen
```

Hinweis

Die Benutzung des goto-Befehls ist unter C-Programmierern verpönt, da man einen undefinierten Programmfluss erzeugen kann, welcher sich nicht debuggen lässt.

Dennoch kann der goto-Befehl in manchen Situationen sehr nützlich sein, z.B. um aus stark verzweigten for-Schleifen auszusteigen.

Beispiel

```
for(byte r = 0; r < 255; r++){
    for(byte g = 255; g > -1; g--){
        for(byte b = 0; b < 255; b++){
            if (analogRead(0) > 250){ goto ausstieg;}
            // weitere Anweisungen
        }
    }
}
ausstieg:
```

Rechenoperationen

= Zuweisungsoperator

Speichert den Wert rechts des = in die Variable links des =.

Das =-Zeichen in der Programmiersprache C heißt **Zuweisungsoperator**. Das = funktioniert ähnlich wie man es aus dem Matheunterricht kennt. Die linke Seite ist gleich der rechten Seite. Allerdings funktioniert das ***nur*** in diese Richtung!

Beispiel

```
// Ein kleiner Tipp:

int a = 10; // funktioniert, schreibt den Wert 10 in die Variable a
5 = a; // funktioniert nicht, da 5 nicht der Wert aus a zugewiesen werden kann,
       //da 5 bereits selber ein Wert ist

// Deklaration und Wertzuweisung getrennt
int sensVal;
sensVal = analogRead(0);

// Deklaration mit sofortiger Wertezuweisung
int sensVal = analogRead(0);
```

Hinweis

Die Variable auf der linken Seite des Zuweisungsoperators muss groß genug sein, um den zugewiesenen Wert zu speichern, andernfalls ist der gespeicherte Wert inkorrekt. Der Wert 1023 (10-bit groß) kann also nicht in eine byte-Variable (8-bit groß) gespeichert werden.

= (einfaches =) und == (doppeltes =) dürfen nicht verwechselt werden, da = der Zuweisungsoperator, zum Zuweisen eines Wertes, und == der Vergleichsoperator ist, welcher prüft, ob zwei Werte gleich groß sind.

Grundrechenarten (Addition, Subtraktion, Division, Multiplikation)

Die Rechenoperatoren ermöglichen alle Grundrechenarten. Die Operatoren geben die Summe, die Differenz, das Produkt oder den Quotienten der Faktoren rechts und links der Operatoren zurück. Das Ergebnis hängt maßgeblich vom Datentypen der gewählten Variablen ab.
Ein kleines Beispiel:

`int ergebnis = 9 / 4` schreibt den Wert 2 in die Variable ergebnis, obwohl der Quotient beider Zahlen ja eigentlich 2,25 ist, da der Datentyp `int` nur ganzzahlige Werte aufnehmen kann. Der Rest 0.25 wird einfach “abgeschnitten”, weil er hinter dem Komma steht. Nur Variablen des Typs `float` oder `double` können Dezimalbrüche (Zahlen mit Nachkommastellen) aufnehmen.

Syntax

```
ergebnis = Faktor1 + Faktor2;
ergebnis = Faktor1 - Faktor2;
ergebnis = Faktor1 * Faktor2;
```

```
ergebnis = Faktor1 / Faktor2;
```

Parameter
Faktor1: Eine Variable beliebigen Datentyps
Faktor2: Eine Variable beliebigen Datentyps

Beispiel
```
w = w + 3;
x = x - 7;
y = y * 6;
z = z / 5;
```

% Modulo

Der Modulo Operator gibt den Rest einer Division zweier ganzzahligen Faktoren zurück.
Zu kompliziert? Also ein kleines Beispiel:

`9 / 2` ergibt 4,5. Wenn wir diesen Wert allerdings in eine int-Variable schreiben, nimmt die Variable den Wert 4 an, da der Nachkommawert einfach abgeschnitten wird.

Der Modulo Operator gibt nun den Rest der Division zurück:

`9 % 2` ergibt 1. Wieso?
Besser zu verstehen ist das, wenn wir uns einmal das Konzept des schriftlichen Dividierens in der 3. Klasse anschauen. Wir schauen, wie oft die 2 in die 9 hineinpasst. Die 2 passt 4-mal in die 9, denn 4*2 = 8 Rest 1. Das Modulo gibt uns diesen Rest zurück.

Der Vollständigkeit halber mache ich hier nun weiter: wir setzen ein Komma hinter die 4 und zu der 1 ziehen wir eine 0 hinunter. Jetzt fragen wir uns, wie oft passt die 2 in die 10 hinein? Klar, 5-mal. Hinter die 4, schreiben wir jetzt also eine 5 und wir bekommen 4,5.

Syntax
```
ergebnis = dividend % divisor
```

Parameter
dividend:	die zu teilende Zahl
divisor:	der Teiler

Beispiel
```
x = 7 % 5;   // x enthält 2
x = 9 % 5;   // x enthält 4
x = 5 % 5;   // x enthält 0
x = 4 % 5;   // x enthält 4
```

Hinweis
Der Modulo Operator funktioniert logischerweise nicht mit floats. (Fließkommazahlen)

Verknüpfungsoperationen (Boolesche Operatoren)

Diese Operatoren können innerhalb einer if-Schleife benutzt werden, um z.B. zwei Bedingungen logisch zu verknüpfen.

&& (logisches und)

ist nur wahr, wenn beide Bedingungen erfüllt sind, z.B.

```
if (digitalRead(2) == HIGH  && digitalRead(3) == HIGH) { // lese zwei Schalter aus
 // ...
}
```

Prüft, ob beide Eingänge logisch HIGH sind.

|| (logisches oder)

ist wahr, wenn eine der beiden Bedingungen wahr ist, z.B.

```
if(x > 0 || y > 0){
  // ...
}
```

Ist wahr, wenn x oder y größer 0 ist.

! (nicht)

wahr wenn die Bedingung nicht wahr ist, z.B.

```
if(!(x==3)){
  // ...
}
```

ist wahr, wenn x nicht gleich 3 ist.

Hinweis

Beachte, dass das logische UND ein „&&“ ist und das bitweise UND ein „&“ ist, denn die beiden sind komplett verschieden! Auch haben das logische ODER (||) und das bitweise ODER (|) unterschiedliche Funktionen. Das bitweise NICHT ~ sieht zwar anders aus als das logische NICHT !, aber man sollte sich doch im Klaren sein, welches wo verwendet wird.

```
if (a >= 10 && a <= 20){}
```
// ist wahr wenn a zwischen 10 und 20 liegt

Digital In / Out

pinMode()

Beschreibung
Bestimmt die Datenrichtung eines Pins, d.h. ob ein Pin als Ausgang oder Eingang genutzt wird.
Siehe auch hier.

Syntax
pinMode(pin, modus);

Parameter
pin: die Pinnummer des Pins, der konfiguriert werden soll (z.B. 13)
modus: entweder INPUT für Eingang oder OUTPUT für Ausgang

Beispiel
```
int ledPin = 13;                  // LED an digital pin 13
int eingang = 10;                 // Eingang an Pin 10

void setup()
{
  pinMode(ledPin, OUTPUT);       // Pin 13 als Ausgang konfigurieren
  pinMode (eingang, INPUT);       //
}
```

Analoge Eingänge, welche mit analogRead ausgelesen werden sollen, müssen nicht mit pinMode konfiguriert werden.

digitalWrite()

Beschreibung
Den Pin auf logisch HIGH oder LOW setzen.
Wenn der Pin als Ausgang konfiguriert ist, dann liegt der Pin bei HIGH auf der Betriebsspannung (5V oder 3.3V) des Boards und bei LOW auf 0V, sprich Masse (GND).
Wenn der Pin als Eingang konfiguriert ist, dann wird bei HIGH ein interner 20kOhm Pullup-Widerstand an den Pin geschaltet (kann verwendet werden, um die Spannung an dem Pin auf 5V bzw. 3.3V zu halten, sodass ein definierter Zustand herrscht), bei LOW wird dieser wieder von Pin getrennt.

Syntax
digitalWrite(pin, wert);

Parameter
pin: die Pinnummer des Pins, der geschrieben werden soll (z.B. 13)
wert: entweder HIGH oder LOW

Beispiel
```
int ledPin = 13;                    // LED an digital pin 13
int eingang = 10;                   // Eingang an Pin 10

void setup()
{
```

```
  pinMode(ledPin, OUTPUT);          // digital pin 13 als Ausgang konfigurieren
  pinMode (eingang, INPUT);         //
  digitalWrite(eingang, HIGH);      // schaltet den internen PullUp-Widerstand ein
}

digitalWrite(ledPin, HIGH);         // setzt ledPin auf logisch HIGH
```

Hinweis
Die Analog-Pins können auch als digitale Ausgänge genutzt werden, sie heißen A0 bis A5.
Die Pins A6 und A7 beim ARDUINO NANO können nicht als digitale Ausgänge verwendet werden!

digitalRead()

Beschreibung
Liest den Zustand des Pins entweder HIGH oder LOW ein und gibt diesen zurück.

Syntax
digitalRead(pin);

Parameter
pin: die Pinnummer des Pins, der gelesen werden soll (z.B. 10)

Beispiel
```
int ledPin = 13; // LED an pin 13
int inPin = 7;   // Taster an Digital-Pin 7
int val = 0;     // variable um den Pinstatus zu speichern

void setup()
{
  pinMode(ledPin, OUTPUT);          // setzt pin 13 als Ausgang
  pinMode(inPin, INPUT);            // setzt pin 7 als Eingang
}
void loop()
{
  val = digitalRead(inPin);         // liest pin 7 ein
  digitalWrite(ledPin, val);        // setzt pin 13 auf den Wert von pin 7
}
```

Hinweis
Die beiden **Analog-Pins A6 und A7** können nicht als digitale Eingänge benutzt werden! (Danke an Member hotsystems und Bitklopfer).
Hier hilft nur, falls man mehr digitale Eingänge benötigt, ein Portexpander oder aber diese Eingänge mit `analogRead` auszulesen und dann den Analogwert auszuwerten als digitalen Wert.
Die Pins dürfen nur mit Spannungen bis maximal der Betriebsspannung des jeweiligen ARDUINO-Boards beaufschlagt werden! Das können max. 5V DC sein, oder max. 3,3V DC (z.B. ARDUINO PRO MINI).

Grundlegendes Analog In / Out

analogReference()

Beschreibung
Bestimmt die Referenzspannung für die analogen Eingänge (den ADC):

- DEFAULT: Die standardmäßige Referenzspannung für den ADC beträgt 5V bzw. 3.3V (je nach Betriebsspannung des Boards)
- INTERNAL: Interne Spannungsreferenz beim ATmega168 und ATmega328P beträgt die Referenzspannung 1.1V, beim ATmega8 2.56 (nicht beim ARDUINO Mega vorhanden)
- INTERNAL1V1: Interne Spannungsreferenz von 1.1V (nur ARDUINO Mega)
- INTERNAL2V56: Interne Spannungsreferenz von 2.56V (nur ARDUINO Mega)
- EXTERNAL: Die Spannung am AREF Pin wird als Spannungsreferenz verwendet. Achtung! Maximal die Betriebsspannung VCC des Boards (je nach Board 3,3V oder 5V)

Syntax
```
analogReference(type);
```

Parameter
Eines der oben genannten Makros z.B. EXTERNAL

Hinweis
Der AREF Pin darf nur mit Spannungen zwischen 0V und +5V DC beschaltet werden, ansonsten kann der Controller beschädigt werden! Wenn der AREF-Pin beschaltet ist muss zuerst die Referenzspannung mit analogReference(EXTERNAL) auf extern gestellt werden, bevor analogRead() aufgerufen wird, ansonsten werden die interne und externe Referenzspannung kurzgeschlossen und der Controller wird beschädigt.

analogRead()

Beschreibung
Liest den angegebenen Analogeingang aus. Der ARDUINO hat einen 6-Kanal (8 beim Mini und Nano, 16 beim Mega) 10-bit ADC, dass bedeutet der ARDUINO UNO z.B. wandelt Spannungen von 0V bis 5V in Zahlen von 0 bis 1023 um. Bei 5V Referenzspannung ist das eine Auflösung von 5V / 1024 = 0.0049V = 4,9mV. Es dauert ungefähr 100µs einen Analogwert zu lesen, somit ergibt sich eine theoretische, maximale Frequenz von 10.000 Abfragen pro Sekunde.

Syntax
```
analogRead(pin);
```

Parameter
pin: die Pinnummer des Pins, der konfiguriert werden soll (A0 bis A5 bei den meisten Boards, A0 bis A7 beim Mini und Nano, A0 bis A15 beim Mega)

Rückgabewert
int (Integer-Wert) zwischen 0 und 1023

Beispiel
```
int analogPin = 3;     // Potentiometer Schleifer am Pin A3
```

```
int val = 0;                          // Variable um Messergebnis zu
speichern

void setup()
{
  Serial.begin(9600);                 //  serielle Kommunikation einschalten
}

void loop()
{
  val = analogRead(analogPin);        // Eingang auslesen
  Serial.println(val);                // Messwert über serielle Verbindung senden
}
```

analogWrite()

Beschreibung

Gibt eine PWM (Pulsweitenmodulation, Rechteckspannung) am definierten Pin aus. Diese PWM kann z.B. benutzt werden um eine LED zu dimmen oder einen Motor in der Geschwindigkeit zu regeln (natürlich über einen Treiber). Die PWM wird solange ausgegeben bis die Pulsweite durch einen weiteren Aufruf von analogWrite() geändert wird oder digitalWrite() bzw. digitalRead() für diesen Pin aufgerufen wird.

Die PWM-Frequenz liegt beim ARDUINO UNO (und den ATMega328-basierten Boards) bei ca. 490Hz an den Pins 3,9,10,11 und ca. 980Hz an Pins 5 und 6. Diese kann aber auch geändert werden (siehe Internet). Die Spannung am Ausgang liegt jedoch immer entweder bei 0V oder 5V. Siehe hierzu nächstes Kapitel!

analogWrite() funktioniert nur bei den Pins, die für PWM vorgesehen sind. Das sind im Allgemeinen 3, 5, 6, 9, 10, 11. Es gibt aber noch mehr Möglichkeiten per Software (bei Bedarf im Netz nachlesen)

Syntax

```
analogWrite(pin, pulsweite);
```

Parameter

pin: die Pinnummer des Pins, an dem die PWM ausgegeben werden soll
pulsweite: zwischen 0 und 255

Beispiel

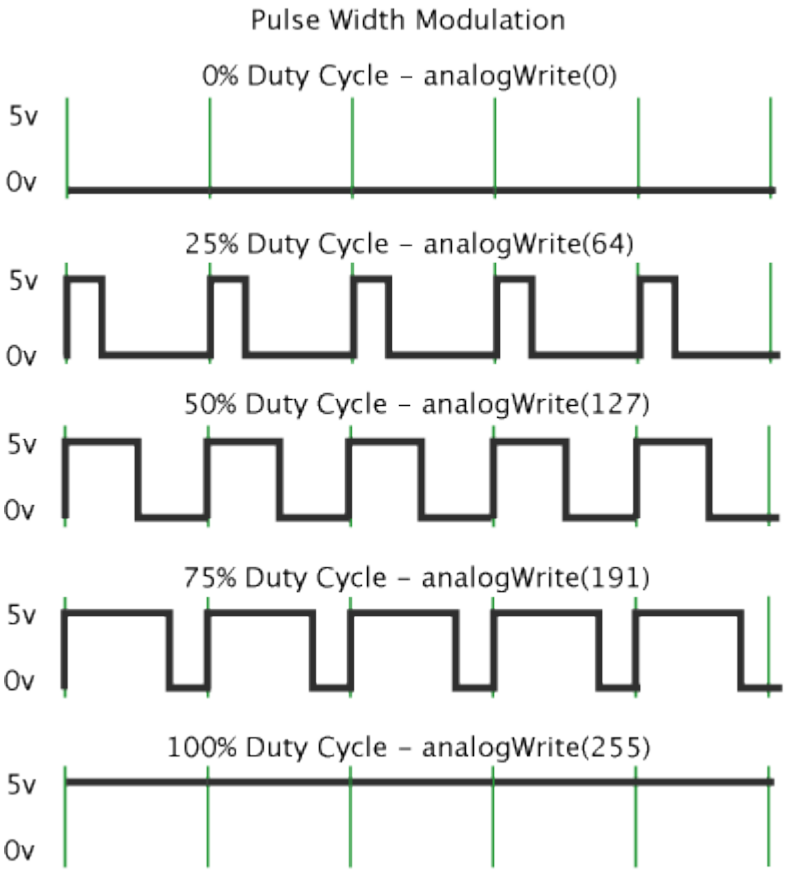
```
int ledPin = 9;       // LED an Pin 9
int analogPin = 3;    // Potentiometer an A3
int val = 0;          // Variable um den Analogwert zu speichern

void setup()
{
  pinMode(ledPin, OUTPUT);   // den Pin als Ausgang konfigurieren
}

void loop()
{
  val = analogRead(analogPin);    // Pin A3 lesen
  analogWrite(ledPin, val / 4);   // analogRead Werte reichen von 0 bis 1023,
                                  // analogWrite Werte von 0 bis 255
}
```

Gleichspannung analog ausgeben

Der Befehl `analogWrite()` ist etwas verwirrend, wird doch hier keine „analoge“ Spannung ausgegeben, sondern eine Rechteckfrequenz mit unterschiedlicher Pulsbreite. Es ist also weiterhin eine „digitale“ Spannung mit den Werten 0V (LOW) und +UB (HIGH).

Es gibt jedoch Anwendungsfälle, da braucht man eine echte Gleichspannung zwischen 0V und +UB. Um diesen Fall geht es hier

`analogWrite()` **macht folgendes:**

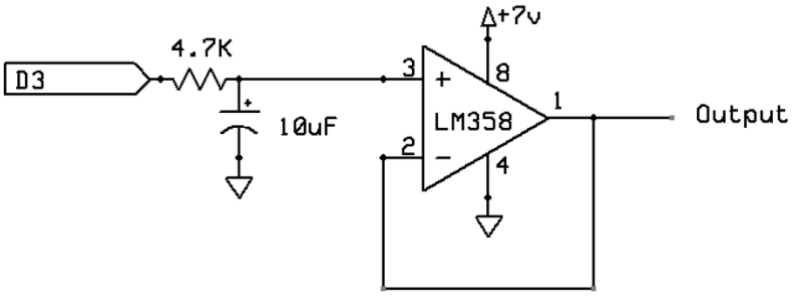


Funktion des Befehls analogWrite() mit unterschiedlichen Werten. (Quelle [15])

Das ist ja nun keine Gleichspannung (außer 0% und 100%)…. Was also tun?
Hier hilft uns althergebrachte Analog-Elektronik. Schön, dass es sie noch gibt. Ich arbeite sehr gern damit, habe ich doch so meine Elektroniker-Tätigkeit vor mehr als 35 Jahren begonnen.

Als erstes braucht man ein sogenanntes RC-Glied als Tiefpass geschaltet. Dieses „glättet“ in gewisser Weise die Impulse (vereinfacht gesagt). Die mathematische Funktion, welche dahintersteckt, nennt sich Integration. Ich will das nicht unnötig verkomplizieren, daher unterlasse ich hier absichtlich weitere Erklärungen. Wer das näher wissen will, kann im Netz Informationen finden.

Im nächsten Bild sieht man eine geeignete Schaltung, welche nur als Vorschlag unter vielen Möglichkeiten dienen soll. Der Widerstand mit 4,7 kOhm und der Kondensator 10 µF bilden das RC-Glied. Diese beiden Bauteile sind zwingend erforderlich. Das nachfolgende Symbol (Bauteil) ist ein sogenannter Operationsverstärker (OPV, OV, opamp). Er dient hier als „Treiber“, d.h. er liefert den gegebenenfalls höheren Strom für die nachfolgende Schaltung. Wird er weggelassen, dann kann es sein, dass die Ausgangsspannung zusammenbricht und nicht mehr dem durch den ARDUINO programmierten Ausgangswert entspricht. Dies verhindert der OPV weitgehend unter normalen Bedingungen. Besonders geeignet sind hier Operationsverstärker vom Typ „rail-to-rail“.



Wichtig: Der Kondensator C speichert Energie. Dadurch gibt es eine zeitliche Verzögerung vom Anlegen der Ausgangsspannung mit dem Befehl `analogWrite()` bis dieser Wert exakt am Ausgang des OPV anliegt. Das ist zu beachten! Gegebenenfalls die Funktion der Schaltung auf einem Steckbrett zusammenbauen und alles testen (idealerweise mit einem Oszilloskop) und bei Bedarf die Werte von R und C ändern. Man kann das alles auch kompliziert berechnen, aber dies soll ja ein Praxisbuch sein…

So ein Steckbrett (Breadboard) ist eine super Sache! Ich baue fast alles erst mal auf einem Steckbrett zusammen. Wenn etwas nicht geht, oder nicht richtig, kann man schnell Änderungen machen. Funktioniert alles wie gewünscht, kann man es auf einer Platine oder Lochrasterplatte aufbauen. So spart man Zeit und eventuell auch Geld, falls man eine Leiterplatte herstellen möchte.

Man kann die gewünschte Funktion von R und C auch mit entsprechender Software simulieren. Leider ist die „Lernkurve“ solcher Software nicht sehr praxistauglich für einmalige Verwendung. Aber das muss jeder selbst entscheiden.

Erweitertes Analoges In / Out

tone()

Beschreibung

Gibt eine PWM mit der angegebenen Frequenz und 50% duty cycle (Tastverhältnis) an einem Pin aus. Der Pin kann an einen Piezo-Lautsprecher angeschlossen werden um einen Ton auszugeben. Es kann eine bestimmte Dauer für den Ton definiert werden, wenn keine Dauer angegeben wird, wird der Ton gespielt bis zum nächsten Aufruf von noTone().
Es kann nur ein Ton zu einer Zeit gespielt werden. Wenn an einem Pin bereits ein Ton ausgegeben wird hat der Aufruf von tone() an einem anderen Pin keinen Effekt. Wenn derselbe Pin aufgerufen wird, wird die Frequenz geändert.

Um einen Ton an einem anderen Pin ausgeben zu können, muss zuerst noTone() aufgerufen werden, bevor der nächste Aufruf von tone() erfolgt.

Achtung! Ein normaler Lautsprecher kann nicht direkt an einen ARDUINO-Pin angeschlossen werden! Lautsprecher haben einen Innenwiderstand zwischen 4 Ohm und 16 Ohm. Das würde den Ausgang des ARDUINO durch zu hohen Stromfluss beschädigen. In diesem Fall ist ein Vorwiderstand, ein nachgeschalteter NF-Verstärker-IC oder eine Transistorstufe entsprechender Dimensionierung nötig.

Syntax

```
tone(pin, frequenz);
tone(pin, frequenz, dauer);
```

Parameter

pin:	die Pinnummer des Pins an dem der Ton ausgegeben werden soll
frequenz:	die Frequenz des Tons (Tonhöhe)
dauer:	die Tondauer in Millisekunden (optional)

noTone()

Beschreibung

Stoppt die Ausgabe einer PWM welche durch tone() verursacht wird.
Diese Funktion muss nach tone() aufgerufen werden, wenn mehrere Töne an verschiedenen Pins ausgegeben werden sollen.

Syntax

```
noTone(pin);
```

Parameter

pin: die Pinnummer des Pins an dem der Ton gestoppt werden soll

shiftOut()

Beschreibung

Gibt ein Bit einer Bytevariablen an einem Port aus. Beginnt entweder von links (Most Significant Bit) oder von rechts (Least Significant Bit). Jedes Bit wird auf einen Data-Pin geschrieben, nachdem ein Clock-Pin getaktet wurde (auf high setzen, dann wieder auf low), welches anzeigt, dass das Bit verfügbar ist. Es kann immer nur ein Bit auf einmal ausgegeben werden. Sollen also alle acht Bits eines Bytes geändert werden, ist dieser Befehl acht Mal auszuführen.
Wenn es um ein Gerät geht, das getaktet wird von steigenden Flanken, dann muss sichergestellt sein, dass der Clock-Pin zuerst wieder auf Low sein muss, bevor shiftOut() aufgerufen werden kann.
Das ist eine Software-Funktion; siehe hier auch SPI library. Die SPI-Library ist eine Hardware-Funktion, diese ist schneller, funktioniert aber nur an bestimmten Pins.

Syntax

shiftOut(dataPin, clockPin, bitOrder, value)

Parameter

dataPin: Der Pin, an dem das Byte ausgegeben werden soll (***int***)
clockPin: Der Pin, der einmal getoggelt werden soll, nachdem **dataPin** auf den korrekten Wert gesetzt wurde *(int)*

bitOrder: welche Richtung das Bit geschoben werden soll, **MSBFIRST** oder **LSBFIRST**. (Most Significant Bit zuerst, oder, Least Significant Bit zuerst)
value: Der Wert, welcher geschoben werden soll. (***byte***)

Rückgabewert

keiner

Hinweis

Der **dataPin** und **clockPin** müssen zuerst als Outputs konfiguriert sein, durch Aufruf des Befehls pinMode(). **shiftOut** ist aktuell beschränkt darauf nur ein Byte (8 Bits) auszugeben. Es sind also zwei Operationen notwendig.

```
// Do this for MSBFIRST serial
int data = 500;
// shift out highbyte
shiftOut(dataPin, clock, MSBFIRST, (data >> 8));
// shift out lowbyte
shiftOut(data, clock, MSBFIRST, data);

// Or do this for LSBFIRST serial
data = 500;
// shift out lowbyte
shiftOut(dataPin, clock, LSBFIRST, data);
// shift out highbyte
shiftOut(dataPin, clock, LSBFIRST, (data >> 8));
```

Beispiel

Zwecks Schaltplan: siehe tutorial on controlling a 74HC595 shift register.

```
//**************************************************************//
//  Name    : shiftOutCode, Hello World                         //
//  Author  : Carlyn Maw,Tom Igoe                               //
//  Date    : 25 Oct, 2006                                      //
```

```
//  Version : 1.0                                                 //
//  Notes   : Code for using a 74HC595 Shift Register             //
//          : to count from 0 to 255                              //
//****************************************************************

//Pin connected to ST_CP of 74HC595
int latchPin = 8;
//Pin connected to SH_CP of 74HC595
int clockPin = 12;
////Pin connected to DS of 74HC595
int dataPin = 11;

void setup() {
  //set pins to output because they are addressed in the main loop
  pinMode(latchPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(dataPin, OUTPUT);
}

void loop() {
  //count up routine
  for (int j = 0; j < 256; j++) {
    //ground latchPin and hold low for as long as you are transmitting
    digitalWrite(latchPin, LOW);
    shiftOut(dataPin, clockPin, LSBFIRST, j);
    //return the latch pin high to signal chip that it
    //no longer needs to listen for information
    digitalWrite(latchPin, HIGH);
    delay(1000);
  }
}
```

shiftIn()

Beschreibung

Liest ein Byte von einem DataPin ein, immer nur ein Bit auf einmal. Beginnt entweder von links (Most Significant Bit) oder von rechts (Least Significant Bit). Bei jedem Bit wird der ClockPin auf high gezogen, das nächste Bit wird gelesen vom DataPin, und der ClockPin wird wieder auf Low gesetzt.
Das ist eine Software-Funktion; siehe hier auch SPI library. Die SPI-Library ist eine Hardware-Funktion, diese ist schneller, funktioniert aber nur an bestimmten Pins.

Syntax

byte incoming = shiftIn(dataPin, clockPin, bitOrder)

Parameter

dataPin:	Der Pin, an welchem das Bit eingelesen werden soll. (***int***)
clockPin:	Der Pin, welcher getoggelt wird, um zu signalisieren, dass ein Bit am dataPin eingelesen werden kann.
bitOrder:	welcher Richtung das Bit gelesen werden soll, **MSBFIRST** oder **LSBFIRST**. (Most Significant Bit zuerst oder Least Significant Bit zuerst)

Rückgabe

den eingelesenen Wert (***byte***)

pulseIn()

Beschreibung

Misst die Dauer eines Pulses an einem Pin. Es kann entweder der Wert LOW oder HIGH übergeben werden. Wenn der übergebene Wert HIGH ist, wird gewartet bis der Pin auf HIGH gezogen wird, startet einen Timer und wenn der Pin wieder auf LOW fällt, stoppt der Timer und die Pulslänge in Mikrosekunden (µs) wird zurückgegeben. Wenn ein Timeout spezifiziert wird, gibt die Funktion nach dieser Dauer den Wert 0 zurück. Die Funktion wurde nur berechnet und kann bei längeren Pulsen zu Fehlmessungen führen. Die Funktion kann Pulslängen von 10µs bis 3 Minuten messen.

Syntax

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

Parameter

pin:	die Pinnummer des Pins an dem der Puls gemessen werden soll
wert:	der Zustand bei dem der Timer gestartet werden soll, HIGH oder LOW
timeout:	die Zeit in Mikrosekunden die die Funktion auf einen Puls wartet, bis sie 0 zurückgibt und beendet wird

Rückgabewert

Die Pulslänge in Mikrosekunden oder 0 wenn kein Puls innerhalb des timeouts gestartet wurde

Beispiel

```
int pin = 7;
unsigned long dauer;

void setup()
{
  pinMode(pin, INPUT);
}

void loop()
{
  dauer = pulseIn(pin, HIGH);
}
```

Datenkonvertierung

byte()

Beispiel-Sketch verfügbar: byte_Datenkonvertierung.ino

Beschreibung
Konvertiert einen Wert in den Datentyp *byte*

Syntax
byte(x);

Parameter
x : beliebiger Wert

Rückgabewert
Variable im byte-Datentyp

Beispiel
```
byte meinByte;
char meinChar = 'A';
meinByte = byte(meinChar + 1);      // meinByte=66
```

int()

Beispiel-Sketch verfügbar: int_Datenkonvertierung.ino

Beschreibung
Konvertiert einen Wert in den Datentyp *int.*

Syntax
int(x)

Parameter
x: beliebiger Wert

Rückgabewert
Variable im int-Datentyp

Beispiel
```
byte meinByte = 200;
int meinInt;
meinInt = int(meinByte);
```

word()

Beschreibung
Wandelt einen Wert in den word-Datentyp oder bildet ein word aus zwei Bytes.

Syntax
word(x)
word(h, l)

Parameter
x: beliebiger Wert
h : höherwertiges Byte
l : niederwertiges Byte

Rückgabewert
Variable im word-Datentyp

long()

Beispiel-Sketch verfügbar: long_Datenkonvertierung.ino

Beschreibung
Wandelt einen Wert in den long Datentyp (Werte: –2.147.483.648 bis 2.147.483.647)

Syntax
long(x)

Parameter
x: beliebiger Wert

Rückgabewert
Variable im long-Datentyp

float()

Beschreibung
Wandelt einen Wert in den float-Datentyp (Fließkommazahl. Vorsichtig benutzen! Siehe hierzu Infos bei Datentypen).

Syntax
float(x)

Parameter
x: beliebiger Wert

Rückgabewert

Hinweis

Der Datentyp „float“ ist eine Zahl mit einem Dezimalpunkt, d.h. mit Nachkommastellen. Sie werden oft benutzt, um analoge und fortlaufende Werte mit hinreichender Genauigkeit darzustellen, weil sie einen wesentlich größeren Wertebereich haben als Integers. Float’s können Werte annehmen von -3.4028235E+38 bis 3.4028235E+38. Sie werden als 32bit-Information gespeichert (4 Bytes).

Float’s haben eine maximale Genauigkeit von 6 bis 7 Stellen. Das bedeutet insgesamt bis zu 7 Stellen, und nicht die Anzahl der Dezimalstellen rechts vom Dezimalpunkt! Auf den ATMega-basierten Boards, also auch UNO, NANO etc. haben die Fließkommazahlen eine Genauigkeit von 7 Stellen. Verwendet man einen ARDUINO DUE kann man als Datentyp double verwenden. Dieser hat die doppelte Genauigkeit und wird als 64bit-Wert behandelt.

Fließkommazahlen sind nicht absolut genau und können seltsame Ergebnisse bei Vergleichen hervorbringen. Zum Beispiel kann die Berechnung 6,00 geteilt durch 3,00 ein Ergebnis hervorbringen, das nicht genau 2,00 ist. Wenn man float’s vergleicht sollte man stattdessen abprüfen, ob die beiden Werte nur eine geringe Abweichung haben, anstatt zu prüfen, ob sie absolut identisch sind.

Weiterhin sind mathematische Operationen mit Fließkommazahlen viel langsamer als Integer und sollten vermieden werden, wenn es zum Beispiel nötig ist, die loop() mit maximaler Geschwindigkeit laufen zu lassen wegen etwaiger zeitkritischer Funktionen. Daher gehen Programmierer auch manchmal den Umweg über eine Konversion zu Integers, um bei mathematischen Operationen die Geschwindigkeit zu erhöhen.

Wenn man mathematische Funktionen mit float’s ausführt, muss man einen Dezimalpunkt verwenden. Ansonsten wird der Wert als Integer behandelt.

Zeit

Bei Benutzung von millis() und micros(): Siehe ergänzende Infos am Ende des Kapitels!

delay()

Beschreibung
Pausiert die Ausführung des Programms um den übergebenen Parameter in Millisekunden.

Syntax
```
delay(ms);
```

Parameter
ms: Die Dauer in Millisekunden um die das Programm pausiert werden soll.

Beispiel

```
int ledPin = 13;                // LED an Pin 13

void setup()
{
  pinMode(ledPin, OUTPUT);      // setzt den ledPin auf Ausgang
}

void loop()
{
  digitalWrite(ledPin, HIGH); // schaltet die LED ein
  delay(1000);                  // wartet eine Sekunde
  digitalWrite(ledPin, LOW);  // schaltet die LED aus
  delay(1000);                  // wartet eine Sekunde
}
```

Hinweis
Es ist zwar einfach eine LED mittels der **delay()**-Funktion zum blinken zu bringen, allerdings blockiert diese Funktion die meisten anderen Aktivitäten des Chips, da keine Berechnungen oder Sensorabfragen stattfinden können. Am besten ist es, die **delay()**-Funktion zu vermeiden wenn es sich um Pausen von mehr als ein paar 10ms handelt, um die Rechenzeit effektiver nutzen zu können.
Fortgeschrittene Programmierer versuchen diese Funktion zu umgehen und verwenden stattdessen lieber **millis()**.
Ein paar Dinge geschehen jedoch während der delay() aktiv ist, da delay() keine Interrupts abschalten. Somit können Daten weiterhin über die serielle Schnittstelle empfangen werden und auch alle anderen Interrupts funktionieren so wie sie sollen.

micros()

Beschreibung
Die Funktion **micros()** gibt die Dauer in Mikrosekunden (µs) seit dem Beginn der Ausführung des Programms zurück. Diese Zahl wird nach ungefähr 70 Minuten überlaufen (auf 0 zurückgesetzt).
Auf 16MHz ARDUINO Boards (UNO, Duemilanove etc.) hat die Funktion eine Auflösung von 4µs, d.h. der Rückgabewert ist ein Vielfaches von 4. Auf 8MHz Boards (z.B. LilyPad) hat der Rückgabewert eine Auflösung von 8µs.

Parameter
Keine

Beispiel

```
unsigned long zeit;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Zeit: ");
  zeit = micros();
  Serial.println(zeit);       // gibt die Zeit seit dem Start des Programms aus
  delay(1000);                // eine Sekunde warten
}
```

millis()

Beschreibung
Die Funktion **millis()** gibt die Dauer in Millisekunden seit dem Beginn der Ausführung des Programms zurück. Diese Zahl wird nach ungefähr 50 Tagen überlaufen (auf 0 zurückgesetzt).

Parameter
Keine

Beispiel
```
unsigned long zeit;

void setup(){
  Serial.begin(9600);
}
void loop(){
  Serial.print("Zeit: ");
  zeit = millis();
  Serial.println(zeit);       // gibt die Zeit seit dem Start des Programms aus
  delay(1000);                // eine Sekunde warten
}
```

delayMicroseconds()

Beschreibung
Pausiert die Ausführung des Programms um den übergebenen Parameter in Mikrosekunden.

Aktuell ist der größte verwendbare Wert 16383, um akkurate Zeiten zu erzeugen. Das könnte sich in späteren ARDUINO-Versionen ändern. Für Pausen größer als ein paar tausend Mikrosekunden sollte man besser delay() benutzen.

Syntax
delayMicroseconds(us)

Parameter
us: Anzahl Mikrosekunden, welche pausiert werden soll (Datentyp: ***unsigned int***)

Rückgabewert
keiner

Beispiel

```
int outPin = 8;                 // digitaler Pin 8 am ARDUINO-Board

void setup()
{
  pinMode(outPin, OUTPUT);      // als Output setzen
}

void loop()
{
  digitalWrite(outPin, HIGH);   // Pin einschalten
  delayMicroseconds(50);        // Pause 50 Mikrosekunden
  digitalWrite(outPin, LOW);    // Pin ausschalten
  delayMicroseconds(50);        // Pause 50 Mikrosekunden
}
```

Konfiguriert Pin 8 als Ausgang. Sendet Impulse mit einer Periodendauer von 100µsek.

Hinweis
Diese Funktion arbeitet sehr akkurat im Bereich von 3 Mikrosekunden und höher. Darunter kann ein korrekter Wert nicht garantiert werden.

Richtige Benutzung des millis() – Befehles

Wichtig ist die richtige Benutzung des millis()-Befehls, da sonst beim Überlauf der Variable Falschberechnungen entstehen! Bei langen Laufzeiten des ARDUINO unbedingt beachten!

```
// Deklarationen am Anfang des Sketches

unsigned long Alte_Zeit;          //Variable zum Speichern des alten Wertes
unsigned long Zeitinterval;       //Variable für gewünschte Zeitdauer (1sek=1000)

// im laufenden Programm…
if (millis() -Alte_Zeit > Zeitinterval)
{
Alte_Zeit = millis();
    .... weiterer Code hier
}
```

Beispiel 1: Alte_Zeit sei 900.000
millis() sei 900.900
Zeitinterval sei 1000 (entspricht einer Sekunde)
900.900 – 900.000 = 900 ➔ 900 ist NICHT größer als 1000, also if-Code wird NICHT ausgeführt.

Beispiel 2: Alte_Zeit sei 900.000
millis() sei 901.100
Zeitinterval sei 1000 (entspricht einer Sekunde)

901.100 – 900.000 = 1100 ➔1100 ist größer als 1000, also if-Code wird ausgeführt.

Aufgrund der Besonderheiten der Binär-Arithmetik funktioniert das auch bei einem Überlauf. Da wir unsigned long Variablen benutzen, kann der maximale Wert dieses Variablentypes 4.294.967.295 sein.

Angenommen, Alte_Zeit sei 4.294.967.000 und Zeitinterval ist 1000. Irgendwann einmal gibt es einen Überlauf, und millis() gibt als Wert die 0 zurück. Man könnte nun meinen, der Vergleich ist folgender:

0 – 4.294.967.000 > 1000? Also nach mathematischer Logik: -4.294.967.000 > 1000 ?
Ergebnis des Vergleiches: False, d.h. if-Code wird nicht ausgeführt!

Aber bei der Benutzung von unsigned long Variablen kann es keinen negativen Wert geben und der zurückgegebene Wert der Subtraktion wird 295 sein. Das hängt damit zusammen, dass der Maximalwert nur 4.294.967.295 sein kann. Dieser wird anstatt der 0 verwendet. Also:
4.294.967.295 - 4.294.967.000 > 1000?
Ergebnis des Vergleiches: False, d.h. if-Code wird nicht ausgeführt!

Also: Auch bei einem Überlauf funktioniert der Vergleich korrekt, es gibt keine Fehlberechnungen.
Die angegebene Verfahrensweise gilt analog auch bei micros().

Quelle: [9]

Die Funktionen millis() und micros() können auch mehrfach im Programm verwendet werden, mit unterschiedlichen Zeiten. Damit kann man viele unterschiedliche Zeitabschnitte verwirklichen, wenn man es braucht.

```
// Deklarationen am Anfang des Sketches

unsigned long Alte_Zeit1;
unsigned long Zeitinterval1;
unsigned long Alte_Zeit2;
unsigned long Zeitinterval2;
```

Zusammengesetzte Operatoren

Zusammengesetztes bitweises ODER (|=)

Beschreibung
Das zusammengesetzte bitweise ODER (|=) wird häufig verwendet, um gezielt bestimmte Bits innerhalb eines Integers auf 1 zu setzen.

Syntax
```
x |= y;   // entspricht x = x | y;
```

Parameter
x: eine char, int oder long Variable
y: eine ganzzahlige Konstante oder eine char, int oder long Variable

Zusammengesetztes bitweises UND (&=)

Beschreibung
Das zusammengesetzte bitweise UND (&=) wird oft verwendet, um gezielt bestimmte Bits innerhalb eines Integers mit Hilfe einer Bitmaske auf 0 zu setzen. Oftmals wird dies “clearing” oder “resetting” von Bits genannt.

Syntax
```
x &= y;   // entspricht x = x & y;
```

Parameter
x: eine char, int oder long Variable
y: eine ganzzahlige Konstante oder eine char, int oder long Variable

+= , -= , *= , /= Zusammengesetzte + - * /

Beschreibung
Führt eine mathematische Operation an einer Variable mit einer anderen Variable oder Konstante durch, += usw. sind nur praktische Abkürzungen der Standard Syntax.

Syntax
```
x += y;   // entspricht x = x + y;
x -= y;   // entspricht x = x - y;
x *= y;   // entspricht x = x * y;
x /= y;   // entspricht x = x / y;
```

Parameter
x: jeder beliebige Datentyp
y: jeder beliebiger Datentyp oder Konstante

Beispiele

```
x = 2;
x += 4;       // x = 6
x -= 3;       // x = 3
x *= 10;      // x = 30
x /= 2;       // x = 15
```

++ (inkrement) / — (dekrement)

Beschreibung

Eine Variable inkrementieren (erhöhen / addieren) / dekrementieren (erniedrigen / subtrahieren)

Syntax

```
x++;  // inkrementiere x um eins und gebe den alten Wert von x zurück
++x;  // inkrementiere x um eins und gebe den neuen Wert von x zurück

x--;  // dekrementiere x um eins und gebe den alten Wert von x zurück
--x;  // dekrementiere x um eins und gebe den neuen Wert von x zurück
```

Parameter

x: (unsigned) int oder long

Rückgabewert

Der originale oder der inkrementierte / dekrementierte Wert von x

Beispiel

```
x = 2;
y = ++x;      // x enthält 3, y enthält 3
y = x--;      // x enthält wieder 2, y enthält immer noch 3
```

Bitoperatoren

Bitweises links- (<<) und bitweises rechtsschieben (>>)

Beschreibung

Es gibt zwei Bitschiebeoperatoren in C/C++: den Linksschiebe- (<<) und den Rechtsschiebeoperator (>>). Diese Operatoren bewirken, dass die Bits im linken Parameter um die Anzahl der Bits im rechten Parameter nach rechts oder links geschoben werden.

Syntax

```
Variable << Anzahl_der_Bits
Variable >> Anzahl_der_Bits
```

Parameter

Variable := (byte, int, long)
Anzahl_der_Bits := int <= 32

Beispiel

```
int a = 5;         // binary: 0000000000000101
int b = a << 3;    // binary: 0000000000101000, 40 in dezimal
int c = b >> 3;    // binary: 0000000000000101, oder zurück zu 5 wie am Anfang
```

Wenn man den Wert x um y Bits nach links schiebt (x<<="" p="">

```
int a = 5;         // binär: 0000000000000101
int b = a << 14;   // binär: 0100000000000000 - die Erste 1 in 101 wurde "aus dem Byte geschoben"
```

Wenn man davon ausgeht, dass keine der Bits links des Wertes aus dem Byte herausgeschoben werden, dann kann man sich den << Operator etwa so vorstellen:
Er multipliziert den linken Wert mit 2 hoch dem rechten Wert also so:

```
1 <<  0  ==    1
1 <<  1  ==    2
1 <<  2  ==    4
1 <<  3  ==    8
...
1 <<  8  ==  256
1 <<  9  ==  512
1 << 10  == 1024
```

Wenn man x um y nach rechts schiebt (x>>y), dann hängt das genaue Ergebnis vom Datentyp des Wertes ab. Wenn es sich um einen "normalen" also signed integer handelt (int), dann wird das Signed-Bit immer mit nach rechts geschoben (signed extension):

```
int x = -16;     // binary: 1111111111110000
int y = x >> 3;  // binary: 1111111111111110
```

Meistens möchte man aber, dass das Signed-Bit nicht mitgeschoben wird, d.h. von links her werden 0en eingeschoben, indem man per typecast zuerst einen unsigned int daraus macht:

```
int x = -16;                  // binary: 1111111111110000
int y = (unsigned int)x >> 3;  // binary: 0001111111111110
```

Wenn man weiß, dass die signed extension keine Probleme bereiten wird, dann kann man den >> Operator als divisor verwenden. x wird geteilt durch 2 hoch y:

```
int x = 1000;
int y = x >> 3;   // integer division von 1000 geteilt durch 8, Ergebnis y = 125
```

Bitweises NICHT (~)

Das bitweise NICHT ist dieses ~ Symbol. Das bitweise NICHT dient anders als das bitweise UND bzw. ODER nicht zur Verknüpfung zweier Operanden, sondern der Invertierung eines Operanden:

```
0  1    operand1
-----------------
1  0   ~ operand1

int a = 103;     // binär:  0000000001100111
int b = ~a;      // binär:  1111111110011000 = -104
```

Es mag vielleicht überraschend sein, dass eine negative Zahl wie -104 bei der Invertierung als Ergebnis auftaucht.
Dies liegt daran, dass in einer int (Integer) Variable das oberste Bit das sog. Signed-Bit ist, wenn es also eine 1 ist, dann gilt die Zahl als negativ. Dieses Verfahren nennt man **Zweierkomplement** (engl. two's complement).
Mehr Information darüber kann auf Wikipedia nachgelesen werden.
Möchte man das Signed-Bit umgehen, muss man die Variable als unsigned int, also als Integer ohne Vorzeichen deklarieren.

Bitweises UND (&)

Das bitweise UND ist ein einfaches & und wird zwischen zwei Integer Variablen eingefügt, um diese zu verknüpfen. Das bitweise UND wirkt sich auf jedes Bit der Operanden aus, unabhängig von den anderen Bits nach dem folgendem Schema: wenn beide Bits 1 sind, dann ist dieses Bit im Ergebnis auch 1, andernfalls ist es 0.

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  0  0  1    (operand1 & operand) = ergebnis
```

In der ARDUINO IDE ist der Datentyp int 16 Bit breit also werden im folgenden Codeschnipsel 16 einzelne UND Operationen durchgeführt:

```
int a =  92;     // binär:    0000000001011100
int b = 101;     // binär:    0000000001100101
int c = a & b;   // Ergebnis: 0000000001000100, oder 68 in dezimal
```

Das bitweise UND wird oft verwendet, um gezielt Bits aus einem Integer auszuwählen, auch Maskieren genannt.

Bitweises ODER (|)

Das bitweise ODER ist das | Symbol und wirkt sich wie das bitweise UND auf jedes Bit eines Integers einzeln aus. Das bitweise ODER gibt eine 1 zurück, wenn eines oder beide Bits 1 sind, andernfalls gibt es 0 zurück.

```
0  0  1  1    operand1
```

```
0  1  0  1    operand2
----------
0  1  1  1    (operand1 | operand2)
```

Angewendet auf unser vorheriges Beispiel bedeutet das:

```
int a =  92;    // binär:    0000000001011100
int b = 101;    // binär:    0000000001100101
int c = a | b;  // Ergebnis: 0000000001111101, oder 125 in dezimal
```

Bitweises XOR (^)

Das bitweise EXKLUSIV ODER, auch XOR (ausgesprochen: “eks-or”), ist das ^ Symbol. Der Operator ist ähnlich zum bitweisen ODER, allerdings ist das Ergebnis nur dann 1, wenn nur eines der Bits 1 ist:

```
0  0  1  1    operand1
0  1  0  1    operand2
----------
0  1  1  0    (operand1 ^ operand2)
```

Man kann sich das bitweise XOR auch so vorstellen, dass es eine 1 zurückgibt, wenn die Bits einer Stelle in den Operatoren verschieden sind und eine 0 zurückgibt, wenn beide Bits in den Operatoren gleich sind.

Hier ein Codebeispiel:

```
int x = 12;     // binär: 1100
int y = 10;     // binär: 1010
int z = x ^ y;  // binär: 0110, oder dezimal 6
```

Wie man an der untereinander geschrieben Weise der Bits sieht, ist das Ergebnis bitweise umgesetzt. Nur, wenn die beiden verglichenen Bits verschieden sind, ist das Ergebnis 1.

Das bitweise XOR wird meist verwendet um bestimmte Bits zu “togglen” (engl. to toggle; umschalten), also von 0 auf 1 und umgekehrt zu schalten.

Hier ein Beispiel, welches eine LED an Pin 13 (interne LED) blinken lässt:

```
void setup(){
   DDRB = DDRB | B00100000; // Pin 13 auf Ausgang
}

void loop(){
   PORTB = PORTB ^ B00100000;  // invertiere Bit 5, ohne andere Bits zu invertieren
   delay(100);
}
```

Zufallszahlen

randomSeed(seed)

Beschreibung
Setzt einen Wert als Ausgangspunkt für die random() Funktion.

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

Umgekehrt kann es manchmal notwendig sein, pseudo-zufällige Sequenzen zu erzeugen, die sich exakt wiederholen. Das kann erreicht werden durch Aufruf der randomSeed() Funktion mit einem festen Wert.

Parameter
Long oder int-Zahl, um die Random-Funktion zu starten.

Beispiel
```
long randNumber;

void setup(){
  Serial.begin(9600);
  randomSeed(analogRead(0));
}

void loop(){
  randNumber = random(300);
  Serial.println(randNumber);

  delay(50);
}
```

random()

Beschreibung
Die random() Funktion erlaubt die Erzeugung der pseudo-zufälligen Werte innerhalb eines definierten Bereiches von Minimum und Maximum Werten.
Benutzt wird dieses nach der randomSeed() Funktion.

Syntax
random(max)
random(min, max)

Parameter
min – unterer Startwert (inclusive) ***(optional)***
max – oberer Endwert (exclusive)

Rückgabewert
Zufallszahl zwischen min und max-1 (***long-Datentyp***)

Hinweis

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

Umgekehrt kann es manchmal notwendig sein pseudo-zufällige Sequenzen zu erzeugen, die sich exakt wiederholen. Das kann erreicht werden durch Aufruf der randomSeed() Funktion mit einem festen Wert.

Beispiel

```
long randNumber;

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

  // wenn Analog-Pin 0 offen ist, dann wird zufälliges Rauschen
  // am Pin benutzt, um randomSeed aufzurufen.
  // Jedes Mal, wenn die Funktion aufgerufen wird, entstehen so unter-
  // schiedliche Werte.

randomSeed(analogRead(0));
}

void loop() {
  // Print Zufallszahl von 0 bis 299
  randNumber = random(300);
  Serial.println(randNumber);

  // Print Zufallszahl von 10 bis 19
  randNumber = random(10, 20);
  Serial.println(randNumber);
delay(50);
}
```

Externe Interrupts

attachInterrupt()

Beschreibung

Erzeugt den Aufruf einer Funktion, wenn ein externer Interrupt eintritt, also ein Ereignis an einem Pin des ARDUINO. Ersetzt etwaige vorher diesem Interrupt zugeordnete Funktionen. Die meisten ARDUINO Boards haben zwei externe Interrupts: Interrupt 0 an DigitalPin 2 und Interrupt1 an DigitalPin 3. Die folgende Tabelle zeigt die verfügbaren Interrupts an den unterschiedlichen Boards.

Board	int.0	int.1	int.2	int.3	int.4	int.5
Uno, Ethernet	2	3				
Mega2560	2	3	21	20	19	18
Leonardo	3	2	0	1		
Due	(siehe unten)					

Das ARDUINO Due Board hat leistungsstarke Möglichkeiten, um einen Interrupt an allen verfügbaren Pins zuzuordnen. Sie können den gewünschten Pin direkt im Befehl attachInterrupt() angeben.

Syntax

attachInterrupt(interrupt, function, mode)

attachInterrupt(pin, function, mode) (NUR ARDUINO Due)

Parameter

interrupt:	Die Nummer des Interrupts (int-Datentyp)	
pin:	Die Pin-Nummer	(NUR ARDUINO Due)
function:	Die Funktion, welche aufgerufen werden soll, wenn der Interrupt eintritt. Diese Funktion übernimmt keine Parameter und gibt auch keine zurück. Diese Funktion wird oft auch **Interrupt Service Routine** genannt (ISR).	
mode:	Definiert, wann der Interrupt ausgelöst werden soll. Vier Konstanten sind als gültige Werte vordefiniert: ***LOW*** löst den Interrupt aus, wenn der Pin low ist ***CHANGE*** löst den Interrupt aus, wenn der Zustand des Pin sich ändert (von high nach low, von low nach high) ***RISING*** löst den Interrupt aus, wenn der Pin von low nach high wechselt ***FALLING*** löst den Interrupt aus, wenn der Pin von high nach low wechselt. Der **ARDUINO Due** bietet zusätzlich noch folgendes: ***HIGH*** löst den Interrupt aus, wenn der Pin high ist	

Rückgabewerte:
keine

Hinweis
Innerhalb der durch den Interrupt aufgerufenen Funktion arbeitet der Befehl delay() nicht, der Wert der Variable millis() erhöht sich nicht, empfangene serielle Daten gehen möglicherweise verloren. Variablen, welche in der durch den Interrupt aufgerufenen Funktion verwendet werden, sollten als volatile (➔ siehe „Geltungsbereich von Variablen“ ➔ volatile) deklariert werden.

Benutzung von Interrupts
Interrupts sind sehr nützlich, wenn Dinge automatisch passieren sollen im ARDUINO, ohne Zutun des Nutzers und können helfen, einige Timing-Probleme zu lösen. Gute Anwendungen für Interrupts sind z.B. das Auslesen eines Dreh-Encoders, Überwachen von Nutzer-Eingaben, Multitasking von LED-Anzeigen.
Wenn Sie absolut sicher gehen wollen, dass ein Programm die Impulse eines Dreh-Encoders registriert, und auch nicht einen Impuls verlieren soll, dann wird es sehr schwierig sein, ein Programm zu schreiben, das auch noch etwas Anderes nebenbei tun kann. Es muss immer der Pin gepollt werden. Andere Sensoren arbeiten oft ähnlich, z.B., wenn Sie feststellen wollen, ob ein Sound-Sensor einen Ton registriert hat, oder ein PIR-Melder Bewegung. In all diesen Situationen wird die Benutzung eines Interrupts den Controller frei halten für den normalen Programmablauf, und trotzdem keinen einzigen Input eines so angeschlossenen Sensors verpassen.

Beispiel
```
int pin = 13;
volatile int state = LOW;

void setup(){
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);
}
void loop(){
  digitalWrite(pin, state);
}

void blink(){
  state = !state;
}
```

detachInterrupt()

Beschreibung
Schaltet das gewünschte Interrupt ab.

Syntax
detachInterrupt(interrupt)

detachInterrupt(pin) (NUR ARDUINO Due)

Parameter
interrupt: Die Nummer des zu deaktivierenden Interrupts (siehe attachInterrupt() für weiteres).
pin: die Pin-Nummer des zu deaktivierenden Interrupts (NUR ARDUINO Due)

Interrupts (global)

interrupts()

Beschreibung
Aktiviert Interrupts (sowohl interne wie externe) wieder, nachdem sie deaktiviert worden sind mit noInterrupts(). Interrupts erlauben es, bestimmte Funktionen im Hintergrund passieren zu lassen und sind standardmäßig aktiviert. Einige bestimmte Funktionen funktionieren nicht, während Interrupts deaktiviert sind, und eingehende Kommunikation könnte ignoriert werden. Interrupts können aber auch den Programmablauf kurz unterbrechen und so das Timing empfindlich stören. Für solche Fälle können Interrupts deaktiviert werden.

Parameter
keine

Rückgabewerte
keine

Beispiel

```
void setup() {}

void loop()
{
  noInterrupts();
  // zeit-kritischen Code hier einfügen
  interrupts();
  // anderen Code hier
}
```

noInterrupts()

Beschreibung
Deaktiviert Interrupts (sowohl interne wie externe). Sie können sie wieder reaktivieren mit interrupts(). Interrupts erlauben es, bestimmte Funktionen im Hintergrund passieren zu lassen und sind standardmäßig aktiviert. Einige bestimmte Funktionen funktionieren nicht, während Interrupts deaktiviert sind, und eingehende Kommunikation könnte ignoriert werden. Interrupts können aber auch den Programmablauf kurz unterbrechen und so das Timing empfindlich stören. Für solche Fälle können Interrupts deaktiviert werden.

Parameter
keine.

Rückgabewerte
Keine.

Beispiel

```
void setup() {}

void loop()
{
  noInterrupts();
```

```
  // zeit-kritischen Code hier einfügen
  interrupts();
  // anderen Code hier
}
```

Mathematische Funktionen

min(x, y)

Beschreibung
Ermittelt die kleinere von zwei Zahlen.

Parameter
x: die erste Zahl, beliebiger Datentyp
y: die zweite Zahl, beliebiger Datentyp

Rückgabewerte
Die kleinere der beiden Zahlen.

Beispiel
```
sensVal = min(sensVal, 100); // setzt sensVal auf den kleineren Wert,oder 100
// stellt sicher, das sensVal niemals über den Wert 100 geht!
```

Hinweis
Wahrscheinlich nicht wirklich intuitiv, aber max() wird oft benutzt, um das untere Ende eines Variablenbereiches zu begrenzen. Min() wird benutzt, um das obere Ende des Variablenbereiches zu ermitteln.

max(x, y)

Beschreibung
Ermittelt die größere von zwei Zahlen.

Parameter
x: die erste Zahl, beliebiger Datentyp
y: die zweite Zahl, beliebiger Datentyp

Rückgabewerte
Die größere der beiden Zahlen.

Beispiel
```
sensVal = max(sensVal, 20); // setzt sensVal auf den größeren Wert, oder 20
// (stellt sicher, das der Wert mindestens 20 beträgt)
```

Hinweis
Wahrscheinlich nicht wirklich intuitiv, aber max() wird oft benutzt, um das untere Ende eines Variablenbereiches zu begrenzen. Min() wird benutzt, um das obere Ende des Variablenbereiches zu ermitteln.

abs(x)

Beschreibung

Ermittelt den Absolutwert einer Zahl. Die Funktion macht also aus negativen Zahlen (unter 0) eine positive Zahl. Wenn X=5 ist, dann wird 5 zurückgegeben, ist X=-5, dann wird ebenso 5 zurückgegeben.

Parameter
x: die Zahl

Rückgabewerte
x: immer eine positive Zahl, egal ob x vorher negativ oder positiv war.

Hinweis
Aufgrund der Art und Weise, wie die abs() Funktion implementiert ist, vermeiden sie es andere Funktionen innerhalb der Klammern zu benutzen. Das könnte zu falschen Ergebnissen führen.

```
abs(a++);          // vermeiden - ergibt falsches Ergebnis
a++;               // stattdessen so - inkrementiert a als erstes
abs(a);            // ermittelt dann den Absolutwert.
```

constrain(x, a, b)

Beschreibung
Begrenzt eine Zahl innerhalb eines Bereiches (von minimal bis maximal.

Parameter
x: die zu begrenzende Zahl, alle Datentypen
a: das niedrigere Ende des Bereiches, alle Datentypen
b: das obere Ende des Bereiches, alle Datentypen

Rückgabewerte
x: wenn x innerhalb des erlaubten Bereiches von a und b ist
a: wenn x kleiner als a ist
b: wenn x größer als b ist

Beispiel
```
sensVal = constrain(sensVal, 10, 150);
// begrenzt den Bereich des Sensors auf Werte von 10 bis 150
```

map(value, fromLow, fromHigh, toLow, toHigh)

Beschreibung
Wandelt eine Zahl von einem Wertebereich in einen anderen um. Das heißt, ein Wert wird vom Bereich fromLow bis fromHigh in den Bereich toLow bis toHigh umgewandelt.
Die Werte werden nicht begrenzt, wie es die Funktion constrain macht, weil Werte außerhalb des Bereiches manchmal nützlich sein können. Die constrain Funktion kann vorher oder nachher benutzt werden, wenn Wertebegrenzung gewünscht ist.
Zu beachten ist hier, dass die Untergrenze jedes Bereiches größer oder kleiner sein kann als die Obergrenze. Die map() Funktion kann also benutzt werden, um einen Wertebereich umzukehren, z.B.

y = map(x, 1, 50, 50, 1);

Die Funktion kann auch mit negative Zahlen umgehen, also

y = map(x, 1, 50, 50, -100);

ist ebenfalls gültig und funktioniert auch gut.

Die map() Funktion benutzt Integer-Operationen, es gibt also keinen Rest, selbst wenn es einen mathematischen Rest gäbe. Entstehende Reste (Nachkommastellen) werden einfach fallen gelassen, sie werden nicht gerundet.

Parameter

value:	die Zahl, die umgewandelt werden soll
fromLow:	die Untergrenze des aktuellen Bereiches
fromHigh:	die Obergrenze des aktuellen Bereiches
toLow:	die Untergrenze des neuen Bereiches
toHigh:	die Obergrenze des neuen Bereiches

Rückgabewerte

Der in den neuen Bereich umgerechnete Wert.

Beispiel

```
// umwandeln eines analogen Wertes in einen 8bit-Wert (0 bis 255)

void setup() {}

void loop()
{
  int val = analogRead(0); // val als Analogwert von A0 (0 bis 1023)
  val = map(val, 0, 1023, 0, 255); // umwandeln
  analogWrite(9, val); // auf Analogpin A9 ausgeben
}
```

pow(base, exponent)

Beschreibung

Berechnet die Potenz einer Zahl.

Parameter

base: die Zahl (Basis) (float-Datentyp)
exponent: die Potenz, in welche die Basis erhoben werden soll (float-Datentyp)

Rückgabewert

Das Ergebnis der Potenzierung (double-Datentyp)

Beispiel

Siehe fscale.

sqrt(x)

Beschreibung

Berechnet die Quadratwurzel einer Zahl.

Parameter

x: die Zahl, beliebiger Datentyp

Rückgabewert

Die Quadratwurzel der Zahl (double-Datentyp).

Geltungsbereich und Qualifikatoren

Geltungsbereich von Variablen

Variablen in der C Programmiersprache, welche der ARDUINO benutzt, haben eine Eigenschaft, die "Scope" (Geltungsbereich) genannt wird. Das ist einer der Unterschiede zu anderen Sprachen, z.B. BASIC, wo Variablen immer globale (also überall, im gesamten Programm geltende) Variablen sind.

Eine **globale Variable** ist eine Variable, die von allen Funktionen im gesamten Programm gesehen werden kann. **Lokale Variablen** sind Variablen, die NUR in den Funktionen gesehen werden können, in denen sie deklariert wurden. In ARDUINO, JEDE Variable, die außerhalb einer Funktion deklariert wurde (z.B. in setup(), loop(), etc.), ist eine globale Variable.

Wenn ein Programm beginnt umfangreicher und komplexer zu werden, sind lokale Variablen nützlich, um sicherzugehen, dass nur die genutzte Funktion Zugriff auf ihre Variablen hat. Das verhindert Programmierfehler, wenn eine Funktion Variablen ändert, die gleichzeitig auch von einer anderen Funktion genutzt werden.

Es ist auch manchmal nützlich, eine Variable innerhalb einer Schleife zu deklarieren und zu initialisieren. Diese Variable kann NUR innerhalb der Schleife benutzt werden.

Beispiele

```
int gPWMval;  // das gesamte Programm sieht diese Variable

void setup()
{
  // ...
}

void loop()
{
  int i;    // "i" ist nur sichtbar innerhalb von "loop"
  float f;  // "f" ist nur sichtbar innerhalb von "loop"
  // ...

  for (int j = 0; j <100; j++){
  // Variable j kann nur innerhalb der for-loop-Klammern benutzt werden
  }

}
```

static

Static wird benutzt, um Variablen zu erzeugen, die nur von einer Funktion gesehen und geändert werden können. Anders als lokale Variablen, die jedes Mal neu generiert und gelöscht werden, wenn eine Funktion aufgerufen wird – static-Variablen bleiben bestehen und behalten ihre Werte auch zwischen Funktionsaufrufen. Static-Variablen werden nur einmal beim ersten Aufruf einer Funktion deklariert.

Beispiel

```
/* RandomWalk
* Paul Badger 2007
* RandomWalk bewegt sich zufällig zwischen zwei Endpunkten
* Der maximale Schritt in einem Durchlauf ist begrenzt durch
* den Parameter "stepsize".
* Eine static-Variable bewegt sich hoch und runter um einen zufälligen Betrag.
```

```
* Diese Technik ist auch bekannt als "pink noise" und "drunken walk".
*/

#define randomWalkLowRange -20
#define randomWalkHighRange 20
int stepsize;

int thisTime;
int total;

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

void loop()
{        //  test randomWalk function
  stepsize = 5;
  thisTime = randomWalk(stepsize);
  Serial.println(thisTime);
   delay(10);
}

int randomWalk(int moveSize){
   static int  place;     // Variable um Wert zu speichern innerhalb random walk
   //deklariert, um die Werte zwischen den Aufrufen beizubehalten und
//keine andere Funktion kann darauf zugreifen

  place = place + (random(-moveSize, moveSize + 1));

  if (place < randomWalkLowRange){  // check untere + obere Limits
    place = place + (randomWalkLowRange - place);     // reflect number back in
positive direction
  }
  else if(place > randomWalkHighRange){
    place = place - (place - randomWalkHighRange);     // reflect number back in
negative direction
  }

  return place;
}
```

volatile

Volatile ist ein Schlüsselwort, das normalerweise vor dem Datentyp einer Variablen benutzt wird, um die Art zu bestimmen, in der das folgende Programm und der Compiler die Variable behandelt. Eine Variable als „***volatile***“ zu deklarieren ist eine Anweisung an den Compiler. Der Compiler ist eine Software, welche den C-Code des ARDUINO in Maschinencode übersetzt.

Um genau zu sein, es veranlasst den Compiler die Variable aus dem RAM zu laden, und nicht aus einem Speicher-Register. Ein Register ist ein Speicherbereich, wo Variablen normalerweise zwischengespeichert und bearbeitet werden. Unter bestimmten Bedingungen kann der Wert einer im Register abgelegten Variable ungenau sein.
Eine Variable sollte immer dann als ***„volatile“*** deklariert werden, wenn ihr Wert durch ein Ereignis beeinflusst werden kann, welches außerhalb des Codes stattfindet, in dem sie sich befindet. Das kann z.B. eine gleichzeitig aufgerufene Funktion sein. Im ARDUINO – die einzige Funktion, die so etwas kann, ist eine ***Interrupt-Funktion***, genannt ***Interrupt Service Routine.***

Beispiel

```
// toggelt LED, wenn ein Interrupt-Pin seinen Zustand ändert
```

```
int pin = 13;
volatile int state = LOW;

void setup()
{
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE); //springt in Funktion „blink“
}
void loop()
{
  digitalWrite(pin, state);
}
void blink()      //Funktion (= Unterprogramm) blink
{
  state = !state; // state wird negiert
}
```

const

Das Schlüsselwort ***const*** steht für Konstante. Es ist ein Variablem-Kriterium und ändert das Verhalten der Variable in „read-only“, d. h. die Variable kann ganz normal benutzt werden, aber NICHT geändert! Sie bekommen einen Compiler-Fehler, wenn Sie versuchen eine const-Variable im Programm zu ändern.

Konstanten, die mit dem ***const***-Schlüsselwort erstellt werden, folgen den Regeln von anderen Variablen (siehe oben ➔ Geltungsbereich von Variablen, etc.)
Diese Tatsache, und die Fallstricke beim Benutzen von ***#define***, machen das const-Schlüsselwort zu einer vorzüglichen Möglichkeit um Konstanten zu definieren und sollte auf jeden Fall dem ***#define*** bevorzugt werden.

Beispiel

```
const float pi = 3.14;
float x;          // Variable x mit dem Datentyp float

// ....

x = pi * 2;     // eine schöne Sache, und gut zu lesen mit Konstanten

pi = 7;         // NICHT erlaubt! Sie können keine Konstante ändern!
```

#define oder const?

Sie können entweder ***const*** oder ***#define*** benutzen, um numerische oder String-Konstanten zu definieren. Für Arrays ist die Benutzung von const erforderlich. Im Allgemeinen ist die Definition mit "const" gegenüber der mit #define vorzuziehen. #define ist eine Präprozessor-Anweisung, d.h. die hinter #define stehende Zahl wird durch den Compiler eingesetzt, ohne das irgendeine Prüfung durch den Compiler stattfinden kann (z.B. richtiger Datentyp) Danke an Gregor und Quelle [11].

Trigonometrie (Dreiecksberechnungen)

sin(rad)

Beschreibung
Berechnet den Sinus eines Winkels (in Radiant / Bogenmaß). Das Ergebnis bewegt sich zwischen -1 und +1.

Parameter
rad: der Winkel im Bogenmaß (float-Datentyp)

Rückgabewerte
Der Sinus des Winkels (double-Datentyp)

cos(rad)

Beschreibung
Berechnet den Cosinus eines Winkels (in Radiant / Bogenmaß). Das Ergebnis bewegt sich zwischen -1 und +1.

Parameter
rad: der Winkel im Bogenmaß (float-Datentyp)

Rückgabewerte
Der Cosinus des Winkels (double-Datentyp)

tan(rad)

Beschreibung
Berechnet den Tangens eines Winkels (in Radiant / Bogenmaß). Das Ergebnis bewegt sich zwischen –unendlich und +unendlich.

Parameter
rad: der Winkel im Bogenmaß (float-Datentyp)

Rückgabewerte
Der Tangens des Winkels (double-Datentyp)

Zeichen (Character-) Funktionen

Im folgenden Kapitel geht es um Zeichen-Funktionen. Als „Zeichen“ werden hier alle eingehenden Zeichen genannt, welche z.B. über die serielle Schnittstelle empfangen werden können. Zeichen sind dabei nicht nur Buchstaben, sondern auch Ziffern, Sonderzeichen und Steuerzeichen. In gewissen Situationen kann es erforderlich sein, zu erkennen, ob eingegangene Zeichen zum Beispiel ein Sonderzeichen enthalten, oder ein ganz bestimmtes Zeichen. Oder auch, ob das Zeichen eine Ziffer oder ein Buchstabe ist oder ein Steuerzeichen. Genau das können die folgenden Funktionen für uns erledigen.

isAlphaNumeric(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen entweder ein Buchstabe ist oder eine Ziffer.

Syntax
ergebnis = isAlphaNumeric(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isAlphaNumeric(thisChar);
Serial.print("Zeichen ist Buchstabe/Ziffer (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isAlpha(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein Buchstabe ist.

Syntax
ergebnis = isAlpha(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isAlpha(thisChar);
Serial.print("Zeichen ist Buchstabe (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isAscii(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein ASCII-Zeichen ist

Syntax
ergebnis = isAscii(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isAscii(thisChar);
Serial.print("Zeichen ist ASCII-Zeichen (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isWhiteSpace(thisChar)

Beispiel-Sketch verfügbar: ASCII_Zeichen_Funktionen.ino

Beschreibung
Stellt fest, ob ein Zeichen eine **Leerstelle** ist.
Eine Leerstelle kann immer auch ein Leerzeichen sein. Als **Leerstelle** werden die Steuerzeichen LINEFEED (10, DEC), Ver.TAB (11, DEC), FORMFEED (12, DEC), CARRIGERETURN (13, DEC) und als **Leerzeichen** Hor.TAB (9, DEC) und SPACE (32, DEC) gekennzeichnet.
HINWEIS: Aktuell stimmt die originale ARDUINO-Referenz nicht mit der Software (1.8.5) überein. Die Funktionen isSpace() und isWhitespace() geben die Werte vertauscht wieder.
Zur Prüfung dieses Verhalten ist der Sketch ASCII_Zeichen_Funktionen.ino im Download verfügbar!

Syntax
ergebnis = isWhiteSpace(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isWhiteSpace(thisChar);
Serial.print("Zeichen ist Leerstelle (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isControl(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein Steuerzeichen ist. Steuerzeichen sind nichtdruckbare Zeichen, wie z.B. Linefeed (Zeilenvorschub), Neue Zeile (CarriageReturn), Formfeed (Seitenvorschub) etc. Siehe hierzu z.B. Wikipedia.

Syntax
ergebnis = isControl(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isControl(thisChar);
Serial.print("Zeichen ist Steuerzeichen (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isDigit(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen eine Ziffer ist

Syntax
ergebnis = isControl(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isDigit(thisChar);
Serial.print("Zeichen ist eine Ziffer (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isGraph(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein **sichtbares** Zeichen ist. Ein Leerzeichen (SPACE, Leertaste) ist ein nicht sichtbares Zeichen, eine Prüfung mit isGraph(thisChar) würde als Ergebnis ein „false“ ergeben.

Syntax
ergebnis = isGraph(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isGraph(thisChar);
Serial.print("Zeichen ist ein druckbares Zeichen (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isLowerCase(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein klein geschriebenes Zeichen (Buchstabe) ist.

Syntax
ergebnis = isLowerCase(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isLowerCase(thisChar);
Serial.print("Zeichen ist ein klein geschriebenes Zeichen (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isPrintable(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen ein **druckbares** Zeichen ist. Das Leerzeichen (Space-Taste) ist ein druckbares Zeichen, welches aber nicht sichtbar ist. Eine Prüfung mit isPrintable(thisChar) führt als Ergebnis zu einem „true“.
Im Unterschied dazu würde eine Prüfung mit isGraph(thisChar) zu einem „false“ führen, da das Leerzeichen nicht sichtbar ist.

Syntax
ergebnis = isPrintable(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isPrintable(thisChar);
Serial.print("Zeichen ist druckbar (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isPunct(thisChar)

Beschreibung
Stellt fest, ob ein Zeichen eine Interpunktion ist. Interpunktionen sind Satzzeichen wie z.B. Punkt, Doppelpunkt, Ausrufezeichen, Fragezeichen, Komma etc.

Syntax
ergebnis = isPunct(thisChar)

Parameter
thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert
true (wahr; 1) oder false (nicht wahr; 0)

Beispiel
```
ergebnis = isPunct(thisChar);
Serial.print("Zeichen ist Interpunktion (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isSpace(thisChar)

Beispiel-Sketch verfügbar: ASCII_Zeichen_Funktionen.ino

Beschreibung

Stellt fest, ob ein Zeichen ein **Leerzeichen** ist.
Als **Leerzeichen** werden Hor.TAB (9, DEC) und SPACE (32, DEC) bezeichnet.
Als **Leerstelle** dagegen die Steuerzeichen LINEFEED (10, DEC), Ver.TAB (11, DEC), FORMFEED (12, DEC), CARRIGERETURN (13, DEC).
Eine **Leerstelle** kann immer auch ein **Leerzeichen** sein. HINWEIS: Aktuell stimmt die originale ARDUINO-Referenz nicht mit der Software (1.8.5) überein. Die Funktionen isSpace() und isWhitespace() geben die Werte vertauscht wieder.
Zur Prüfung dieses Verhalten ist der Sketch ASCII_Zeichen_Funktionen.ino im Download verfügbar!

Syntax

ergebnis = isSpace(thisChar)

Parameter

thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert

true (wahr; 1) oder false (nicht wahr; 0)

Beispiel

```
ergebnis = isSpace(thisChar);
Serial.print("Zeichen ist Leerzeichen (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isUpperCase(thisChar)

Beschreibung

Stellt fest, ob ein Zeichen ein groß geschriebenes Zeichen (Buchstabe) ist.

Syntax

ergebnis = isUpperCase(thisChar)

Parameter

thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert

true (wahr; 1) oder false (nicht wahr; 0)

Beispiel

```
ergebnis = isUpperCase(thisChar);
Serial.print("Zeichen ist groß geschrieben (1) oder nicht (0): ");
Serial.println(ergebnis);
```

isHexadecimalDigit(thisChar)

Beschreibung

Stellt fest, ob ein Zeichen eine gültige, hexadezimale Zahl ist.

Syntax

ergebnis = isHexadecimalDigit(thisChar)

Parameter

thisChar	Das Zeichen, welches analysiert werden soll
ergebnis	Wert der Prüfung, boolean (true / false)

Rückgabewert

true (wahr; 1) oder false (nicht wahr; 0)

Beispiel

```
ergebnis = isHexadecimalDigit(thisChar);
Serial.print("Zeichen ist hexadezimal (1) oder nicht (0): ");
Serial.println(ergebnis);
```

Hilfsfunktionen

sizeof

Beschreibung
Der sizeof Operator ermittelt die Anzahl Bytes in einem Variablentyp, oder die Anzahl von Bytes, welche durch ein Array belegt werden.

Syntax
sizeof(variable)

Parameter
Variable: jeder Variablentype oder Array (z.B. int, float, byte)

Beispiel
Der sizeof-Operator ist nützlich, wenn man mit Arrays arbeitet (wie z.B. Strings) wo es komfortabel ist, die Größe des Arrays zu ändern, ohne andere Teile des Programms zu zerstören oder zu unterbrechen. Das folgende Programm druckt einen Textstring, immer einen Buchstaben einzeln. Versuchen sie, die Textphrase zu verändern…

```
char myStr[] = "this is a test";
int i;

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

void loop() {
  for (i = 0; i < sizeof(myStr) - 1; i++){
    Serial.print(i, DEC);
    Serial.print(" = ");
    Serial.write(myStr[i]);
    Serial.println();
  }
  delay(5000); // slow down the program
}
```

PROGMEM

Beschreibung
Dieses Schlüsselwort sorgt dafür, dass Daten im Flash des ARDUINO anstatt im SRAM abgespeichert werden. Wie wir wissen, ist der SRAM begrenzt: bei ATMega328-basierten Boards, wie dem Uno, Nano, Micro sind das 2kB. Der Flash (dort wo auch das Programm gespeichert wird) hat bei diesen ARDUINOs 32kB.
Durch die Benutzung von PROGMEM kann man also SRAM sparen, wenn dieser im Programm knapp wird. Die PROGMEM-Funktion ist mittlerweile in der IDE implementiert.

PROGMEM modifiziert eine Variable und funktioniert mit den meisten Datentypen, die beim ARDUINO verwendet werden. Nutzer haben aber herausgefunden, das aufgrund der Version des GCC-Compilers manche Anwendungen des PROGMEM-Keywords funktioniert haben, andere wiederum nicht.

Syntax

```
const dataType variableName[] PROGMEM = {data0, data1, data3...};
```

Parameter

datatype: Wert beliebiger Datentyp
variableName: der Name des Datenarrays

Hinweis

Aufgrund der o.g. möglichen Unterschiede, je nach Compiler-Version, können unterschiedliche Schreibweisen funktionieren. Die folgenden funktionieren:

```
const dataType variableName[] PROGMEM = {};   // so geht's
const PROGMEM  dataType  variableName[] = {}; // oder so
const dataType PROGMEM variableName[] = {};   // aber NICHT so
```

Auch, wenn es theoretisch möglich ist, diese Schreibweise für nur eine einzige Variable zu benutzen, macht es wirklich nur Sinn bei großen Datenmengen, welche in Arrays gespeichert werden müssen. Ich persönlich habe es noch nie benutzt, nur einmal, um es zu testen. Ich denke, für Anfänger ist es einfach zu kompliziert…
Die Benutzung von PROGMEM ist etwas umständlich und auch eine zweistufige Arbeitsweise. Nachdem die Daten mit der Funktion in den Flash gelangt sind, müssen sie zur Benutzung auch dort wieder heraus in den SRAM geholt werden.

Beispiel

```
// save some unsigned ints
const PROGMEM  uint16_t charSet[]  = { 65000, 32796, 16843, 10, 11234};

// save some chars
const char signMessage[] PROGMEM  = {"I AM PREDATOR,  UNSEEN COMBATANT. CREATED BY
THE UNITED STATES DEPART"};

unsigned int displayInt;
int k;    // counter variable
char myChar;


void setup() {
  Serial.begin(9600);
  while (!Serial);

  // put your setup code here, to run once:
  // read back a 2-byte int
  for (k = 0; k < 5; k++)
  {
    displayInt = pgm_read_word_near(charSet + k);
    Serial.println(displayInt);
  }
  Serial.println();

  // read back a char
  int len = strlen_P(signMessage);
  for (k = 0; k < len; k++)
  {
    myChar =  pgm_read_byte_near(signMessage + k);
    Serial.print(myChar);
  }

  Serial.println();
}
```

```
void loop() {
  // put your main code here, to run repeatedly:

}
```

Es ist zu beachten, dass die Variablen, welche mit PROGMEM benutzt werden sollen, global definiert werden müssen. Oder aber mit dem Keyword static, damit sie mit PROGMEM arbeiten.

Der folgende Ausdruck wird innerhalb einer Funktion nicht gehen:

```
const char long_str[] PROGMEM = "Hi, I would like to tell you a bit about myself.\n";
```

Dieser hier geht, auch wenn man ihn lokal in einer Funktion definiert:

```
const static char long_str[] PROGMEM = "Hi, I would like to tell you a bit about
myself.\n"
```

F()-Makro

In diesem Zusammenhang sei auch noch auf das F()-Makro verwiesen. Man kann es benutzen, wenn viele Text-Ausgaben auf den seriellen Monitor erfolgen. Damit kann man sehr schnell den SRAM füllen und hat nicht mehr genügend für den Rest des Programms. Vermeiden lässt sich das mit der folgenden Schreibweise:

```
Serial.print(F("Etwas auf dem Serial Monitor anzeigen…"));
```

Siehe hierzu auch „RAM sparen bei Serial.print“ im Abschnitt „Kleine hilfreiche Programm-Teile“

lowByte()

Beschreibung
Ermittelt das niederwertige Byte einer Variablen (z.B. Word-Datentyp)

Syntax
lowByte(x)

Parameter
x: Wert beliebiger Datentyp

Rückgabewerte
byte

highByte()

Beschreibung
Ermittelt den höherwertigen Byte eines Word, oder den zweitniedrigsten eines noch größeren Datentypes.

Syntax
highByte(x)

Parameter
x: Wert beliebiger Datentyp

Rückgabewerte
byte

bitRead()

Beschreibung
Liest ein bestimmtes Bit einer Zahl.

Syntax
bitRead(x, n)

Parameter
x: die Zahl, von der ein Bit ermittelt werden soll
n: welches Bit gelesen werden soll, beginnend mit 0 für das niedrigstwertige Bit (das am meisten rechts befindliche Bit)

Rückgabewerte

Der Wert des Bit’s, also 0 oder 1

bitWrite()

Beschreibung
Schreibt ein bestimmtes Bit einer numerischen Variable (also keine Strings)

Syntax
bitWrite(x, n, b)

Parameter
x: die numerische Variable, in der das Bit geschrieben werden soll
n: welches Bit geschrieben werden soll, beginnend mit 0 für das niedrigstwertige Bit (das am meisten rechts befindliche Bit)
b: der Wert des Bit’s, der geschrieben werden soll (0 oder 1)

Rückgabewerte
keine

bitSet()

Beschreibung
Setzt (schreibt eine 1) eines bestimmten Bit’s einer numerischen Variablen.

Syntax
bitSet(x, n)

Parameter
x: die numerische Variable, in der ein Bit gesetzt werden soll
n: welches Bit gesetzt werden soll, beginnend mit 0 für das niedrigstwertige Bit (am meisten rechts befindlich)

Rückgabewerte
keine

bitClear()

Beschreibung
Löscht (schreibt eine 0) ein bestimmtes Bit einer numerischen Variablen.

Syntax
bitClear(x, n)

Parameter
x: die numerische Variable, in der ein Bit gelöscht werden soll

n: welches Bit gelöscht werden soll, beginnend mit 0 für das niedrigstwertige Bit (am meisten rechts befindlich)

Rückgabewerte
Keine

bit()

Beschreibung
Ermittelt den Wert eines bestimmten Bits (Bit 0 ist 1, Bit 1 ist 2, Bit 2 ist 4, etc.).
Hier bedarf es einer kurzen Erklärung: Ein Byte besteht aus acht Bits, also z.B. die Zahl 9 entspricht in Binärschreibweise &B00001001:

Binärzahl (Bits)	&B	0	0	0	0	1	0	0	1
Wert der Bits		128	64	32	16	8	4	2	1

Wie man sieht, für die Zahl 9 ist das Bit für die „1“ und die „8“ gesetzt, 1+8=9! Einfach, oder?
Die Bits werden folgendermaßen gezählt:

Nummer des Bits	7	6	5	4	3	2	1	0

Syntax
bit(n)

Parameter
n: das Bit, dessen Wert berechnet werden soll

Rückgabewerte
Der Wert des Bits

Serielle Kommunikation (UART)

Allgemeines

Serielle Schnittstellen werden benutzt, damit der ARDUINO mit einem Computer oder anderer Hardware kommunizieren kann. Alle ARDUINO-Boards haben mindestens eine serielle Schnittstelle, auch bekannt unter der Bezeichnung UART oder USART. Die Datenübertragung geschieht an bestimmten Pins: RX und TX, wie auch über USB mit dem Computer oder auch z.B. Tastatur, Maus etc.
Wenn Sie die serielle Schnittstelle benutzen, dann können diese o.g. Pins nicht für andere Funktionen, wie Eingang oder Ausgang genutzt werden.

In der **ARDUINO-IDE** (Das Programm, um ARDUINO-Sketche zu schreiben), ist ein serielles Terminal eingebaut. Dieses kann benutzt werden, um mit dem ARDUINO-Board über USB zu kommunizieren. Klicken Sie auf den Button für den seriellen Monitor in der Toolbar und wählen sie dieselbe Baudrate wie in der Definierung mit dem Befehl Serial.begin().

Bei allen ARDUINO's ist die serielle Schnittstelle (welche an die USB-Buchse geht) an den Digital-Pins Pin0 (RX) und Pin1 (TX) angeschlossen.

Der **ARDUINO MEGA** hat drei zusätzliche serielle Schnittstellen: **Serial1** an Pin19 (RX) und Pin18 (TX), **Serial2** an Pin17 (RX) und Pin16 (TX), **Serial3** an Pin15 (RX) und Pin14 (TX).
Um diese Pins zu nutzen zur Kommunikation mit einem Computer wird ein weiterer USB-zu-Seriell-Konverter benötigt. Die zusätzlichen seriellen Schnittstellen des MEGA haben keinen USB-Adapter eingebaut.
Soll mit einem externen Gerät über TTL-Pegel kommuniziert werden, muss der RX-Pin des ARDUINO mit dem TX-Pin des externen Gerätes verbunden werden sowie der TX-Pin des ARDUINO mit dem RX-Pin des externen Gerätes. Weiterhin ist natürlich die Verbindung der Masse (GND) notwendig.
Die Pins können NICHT direkt an RS232-Schnittstellen angeschlossen werden, dass würde das ARDUINO-Board zerstören (wegen der -12V und +12V Pegel der RS232).

Der **ARDUINO DUE** hat drei zusätzliche 3,3V-TTL serielle Schnittstellen: **Serial1** an Pin19 (RX) und Pin18 (TX), **Serial2** an Pin17 (RX) und Pin16 (TX), **Serial3** an Pin15 (RX) und Pin14 (TX).
Zusätzlich ist hier noch ein nativer USB-Seriell-Port mit dem SAM3X-Chip vorhanden: SerialUSB.

Der **ARDUINO LEONARDO** benutzt **Serial1** um mit TTL-Pegel an Pins Pin0 (RX) und Pin1 (TX) zu kommunizieren. **Serial** ist reserviert für USB CDC Kommunikation.

Der **ARDUINO YUN** hat ebenfalls Serial1 onBoard, nutzt diesen aber für WiFi, USB-Host, Ethernet und SD-Karte.

Falls gewünscht, müssen weitere Infos im Netz nachgelesen werden.

Hier folgen nun kurze Beschreibungen der sechs wichtigsten Befehle für die serielle Kommunikation.
Aktuell ist keine detaillierte Beschreibung im Netz zu finden, also immer mal nachschauen: Vielleicht gibt es mittlerweile etwas. Das kann dann auch noch in dieses Buch aufgenommen werden.
Alle Funktionen (Befehle) für die serielle Kommunikation beginnen mit dem ‚Serial', gefolgt von der entsprechenden Funktion. Es sind die folgenden Funktionen verfügbar:

- **if (Serial)**
- **available ()** Beschreibung verfügbar, siehe nächste Seiten
- **begin ()** Beschreibung verfügbar, siehe nächste Seiten

- **end ()**
- **find ()**
- **findUntil ()**
- **flush ()** Beschreibung verfügbar, siehe nächste Seiten
- **parseFloat ()**
- **parseInt ()**
- **peek ()**
- **print ()** Beschreibung verfügbar, siehe nächste Seiten
- **println ()** Beschreibung verfügbar, siehe nächste Seiten
- **read ()** Beschreibung verfügbar, siehe nächste Seiten
- **readBytes ()**
- **readBytesUntil ()**
- **readString()**
- **setTimeout ()**
- **write ()**
- **serialEvent()**

Serial.begin(speed)

Beschreibung
Initialisiert die serielle Kommunikation mit der Geschwindigkeit speed. Für gewöhnlich wird man hier 9600 Baud benutzen, es sind aber auch andere Werte möglich, maximal aber 115200 Baud.

Syntax
Serial.begin(speed)

Parameter
speed: die gewünschte Geschwindigkeit in Baud (9600, etc.)

Rückgabewerte
Keine

Beispiel:
```
Serial.begin(9600);  // initialisiert mit 9600 Baud
```

Serial.print(data) Serial.print(data, encoding)

Beschreibung
Sendet Daten an den seriellen Port. Encoding ist optional, wenn es weggelassen wird, dann wird der ARDUINO versuchen so viel wie möglich einfachen Text zu verwenden. Dieser wird von den meisten externen Komponenten richtig wiedergegeben. Die Ausgabe endet in der selben Zeile OHNE Zeilenumbruch. Neue Ausgaben werden fortlaufend angefügt.

Beispiel:
```
Serial.print (var); // druckt den Wert der Variable var
Serial.print(75); // druckt "75"
Serial.print(75, DEC); // wie oben.
Serial.print(75, HEX); // "4B" (75 in hexadezimal)
```

```
Serial.print(75, OCT); // "113" (75 in oktal)
Serial.print(75, BIN); // "1001011" (75 in Binärcode)
Serial.print(75, byte()); // "K" (Rowbyte entspricht 75 in ASCII Code)
```

Serial.println(data) Serial.println(data, encoding)

Beschreibung

Identisch mit Serial.print, AUSSER das ein ***Carriage Return*** und ***Linefeed (\r \n)*** angefügt wird. Das ist etwa so, als wenn man in einer Textverarbeitungssoftware eine Zeile Text schreibt, und dann RETURN oder ENTER drückt.

Beispiel:

```
Wie oben, aber es wird nach dem Drucken in eine neue Zeile gesprungen
```

Serial.available()

Beschreibung

Ermittelt, wie viele ungelesene Bytes am seriellen Port verfügbar sind, um mit der read()-Funktion gelesen zu werden. Nachdem alles mit Serial.read() gelesen wurde, ergibt Serial.available() den Wert 0, solange, bis wieder Daten über den seriellen Port empfangen wurden.

Rückgabewert

Anzahl Variable Datentyp int, welche der Anzahl verfügbarer Datenbytes entspricht

Beispiel

```
int Anzahl = Serial.available();
```

Serial.read()

Beschreibung

Holt ein hereingekommenes Byte aus der seriellen Schnittstelle ab

Rückgabewert

data Variable Datentyp byte, die dem empfangenen Zeichen entspricht (ASCII-Code)

Beispiel

```
int data = Serial.read();
```

Serial.flush()

Über den seriellen Port können Daten schneller reinkommen, als sie vom ARDUINO verarbeitet werden. Diese werden in einem Zwischenspeicher gespeichert. Um diesen Speicher zu löschen, damit er mit neuen Daten gefüllt wird, können wir die flush() Funktion benutzen.

Beispiel

```
Serial.flush();
```

String Objects

Allgemeines

Die String objects erlauben die Auswertung und Manipulation von Strings in einer wesentlich komplexeren Art, als das mit char-Array‘s möglich ist. Sie können Strings verbinden, andere Strings anhängen, suchen und ersetzen von Unter-Strings und vieles mehr. Es braucht mehr Speicher, ist aber auch wesentlich leistungsfähiger.
Zur Unterscheidung: Strings in Arrays werden mit einem kleinen s geschrieben, wenn es um String objects geht wird String mit einem großen S geschrieben. Beachten Sie, dass String-Konstanten, die in doppelten Anführungszeichen geschrieben werden, immer ein String-Array sind.

Den Datentyp String zu verwenden, bringt Bequemlichkeit, "kostet" aber 1212 Bytes Programmspeicher (Flash) und 48 Bytes RAM. Oft ist es das wert, wenn man mit dem Flash-Speicher nicht schon an der Grenze ist. Quelle: [30]

Bei den String Objects gibt es die folgenden Funktionen (Befehle):

- **String()**
- **charAt()**
- **compareTo()**
- **concat()**
- **endsWith()**
- **equals()**
- **equalsIgnoreCase()**
- **getBytes()**
- **indexOf()**
- **lastIndexOf()**
- **length()**
- **replace()**
- **setCharAt()**
- **startsWith()**
- **substring()**
- **toCharArray()**
- **toLowerCase()**
- **toUpperCase()**
- **trim()**

Diese Funktionen werden auf den nächsten Seiten näher beschrieben.

String()

Beschreibung

Erstellt ein String-Objekt. Es gibt mehrere Versionen, ein String zu erstellen, aus verschiedenen Datentypen. Hier z.B., wenn man sie formatiert hat als einzelne Sequenzen von Buchstaben. Das beinhaltet:

- Eine String-Konstante, bestehend aus Buchstaben, in doppelten Anführungszeichen (z.B. ein Char-Array)
- Eine String-Konstante, bestehend aus EINEM Buchstaben, in einfachen Anführungszeichen

- Ein weiteres String-Objekt
- Eine Integer- oder long-integer-Konstante
- Eine Integer- oder long-integer-Konstante, die eine bestimmte Basis benutzt
- Eine Integer- oder long-integer-Variable
- Eine Integer- oder long-integer-Variable, die eine bestimmte Basis benutzt

Wenn ein String aus einer Zahl erstellt wird, dann enthält dieser das ASCII-Zeichen, das dieser Zahl entspricht, die Standard-Basis ist 10, also

```
String thisString = String(13)
```

Erstellt den String mit den Zeichen „13“ (Die 13 ist hier eine Zeichenkette, keine Zahl!) Man kann auch eine andere Basis benutzen, z.B. HEX

```
String thisString = String(13, HEX)
```

Das erstellt den String “D“, was der hexadezimalen Darstellung der Zahl „13“ entspricht. Oder wenn Sie Binärcode bevorzugen:

```
String thisString = String(13, BIN)
```

Erstellt den String “1101”, was der binären Darstellung der Zahl 13 entspricht.

Syntax
String(val)
String(val, base)

Parameter
val: eine Variable, die als String formatiert (umgewandelt in String) werden soll. Datentypen: string, char, byte, int, long, unsigned int, unsigned long
base: (optional) Die Basis der Variable val (also DEZ, HEX, BIN, OCT)

Beispiel:
Alle folgenden Ausdrücke sind gültige String-Deklarationen

```
String stringOne = "Hello String";   // using a constant String
String stringOne =  String('a');     // converting a constant char into a String
String stringTwo =  String("This is a string"); // converting a constant string into
  a String object
String stringOne =  String(stringTwo + " with more");  // concatenating two strings
String stringOne =  String(13);                        // using a constant integer
String stringOne =  String(analogRead(0), DEC);        // using an int and a base
String stringOne =  String(45, HEX);   // using an int and a base (hexadecimal)
String stringOne =  String(255, BIN);  // using an int and a base (binary)
String stringOne =  String(millis(), DEC);        // using a long and a base
```

charAt()

Beschreibung
Greift auf einen bestimmten Buchstaben eines Strings zu.

Syntax
string.charAt(n)

Parameter
string: eine Variable vom Datentyp String
n: der n'te Buchstabe, welcher ermittelt werden

Rückgabewert
Das n'te Zeichen des Strings

Beispiel
```
char x = string.charAt(5);       // gibt das 5. Zeichen des Strings an die Variable x
```

compareTo()

Beispiel-Sketch verfügbar: compareTo.ino

Beschreibung
Vergleicht zwei Strings. Testet, ob einer vor oder hinter dem anderen kommt, oder ob sie identisch sind. Die Strings werden Zeichen für Zeichen verglichen, indem die ASCII-Werte der Zeichen benutzt werden. Das bedeutet, z.B. das ‚a' vor ‚b' kommt, aber nach ‚A'. Ziffern kommen vor Buchstaben.

Syntax
string.compareTo(string2)

Parameter
string: eine Variable vom Datentyp String
string2: eine weitere Variable vom Typ String

Rückgabewert
eine negative Zahl:	wenn string VOR string2 kommt
0	wenn string GLEICH string2 ist
eine positive Zahl:	wenn string NACH string2 kommt

Beispiel
```
String string1 = "Hans";
String string2 = "Peter";
String string3 = "Hans";
int x = 0;

x= string1.compareTo(string2);
Serial.println(x); // negative Zahl, wenn string1 VOR string2 ist

x= string2.compareTo(string1);
Serial.println(x); // positive Zahl, wenn string1 NACH string2 ist

x= string3.compareTo(string1);
Serial.println(x); // Null, wenn beide Strings identisch sind
```

concat()

Beispiel-Sketch verfügbar: concat.ino

Beschreibung
Hängt einen String an einen anderen hinten dran.

Syntax
String1.concat(string2);

Parameter
String1: eine Variable vom Datentyp String
string2: eine weitere Variable vom Typ String

Rückgabewert
Ein neuer String, der aus den beiden Einzelstrings besteht.

Beispiel
```
String string1 = "Hans";
String string2 = "Peter";
string1.concat(string2); // string2 an string1 anhängen
                                  // Ergebnis: HansPeter
```

endsWith()

Beispiel-Sketch verfügbar: endsWith.ino

Beschreibung
Testet, ob oder ob nicht ein String mit dem Zeichen eines anderen Strings endet.

Syntax
string.endsWith(string2)

Parameter
string: eine Variable vom Datentyp String
string2: eine weitere Variable vom Typ String

Rückgabewert
true: wenn string mit dem Zeichen von string2 endet
false: wenn nicht

Beispiel
```
String string1 = "Hans";
String string2 = "s";
String string3 = "r";
boolean x;

x= string1.endsWith(string3);
Serial.println(x); // false, wenn string1 NICHT mit dem Zeichen von string2 endet
x= string1.endsWith(string2);
Serial.println(x); // true, wenn string1 mit dem Zeichen von string2 endet
```

equals()

Beispiel-Sketch verfügbar: equals.ino

Beschreibung
Vergleicht zwei Strings, ob diese gleich sind. Der Vergleich ist „case-sensitive“. Das bedeutet, der String „hello“ ist nicht identisch mit dem String „HELLO“ oder „Hello“. Es wird die exakte Schreibweise verglichen!

Syntax
string.equals(string2)

Parameter
string: eine Variable vom Datentyp String
string2: eine weitere Variable vom Typ String

Rückgabewert
true: string ist identisch mit string2
false: wenn nicht

Beispiel
```
String string1 = "Hans";
String string2 = "Hans";
String string3 = "hans";
boolean x;

x= string1.equals(string3);
Serial.println(x); // false, wenn string1 NICHT identisch mit string3
x= string1.equals(string2);
Serial.println(x); // true, wenn string1 identisch mit string2
```

equalsIgnoreCase()

Beispiel-Sketch verfügbar: equalsIgnoreCase.ino

Beschreibung
Vergleicht zwei Strings, ob diese gleich sind. Der Vergleich ist NICHT „case-sensitive“. Das bedeutet, der String „hello“ ist identisch mit dem String „HELLO“ oder „Hello“.

Syntax
string.equalsIgnoreCase(string2)

Parameter
string: eine Variable vom Datentyp String
string2: eine weitere Variable vom Typ String

Rückgabewert
true: string ist identisch mit string2
false: wenn nicht

Beispiel
```
String string1 = "Hans";
String string2 = "HANS";
```

```
String string3 = "hans";
boolean x;

x= string1.equalsIgnoreCase(string3);
Serial.println(x); // true, weil 'hans' = 'Hans' ist
x= string1.equals(string2);
Serial.println(x); // auch true, NICHT case-sensitive
```

getBytes()

Beispiel-Sketch verfügbar: _getBytes_length.ino

Beschreibung
Kopiert die Zeichen des Strings als ASCII-Wert in den bereitgestellten Pufferspeicher

Syntax
string.getBytes(buf, len)

Parameter
string:	eine Variable vom Datentyp String
buf:	der Pufferspeicher, in den der ASCII-Wert kopiert werden sollen (byte[]-Array)
len:	die Größe des Puffers (unsigned int Datentyp)

Rückgabewert
keine

Beispiel
```
String nachricht = "Dies ist ein Test";

byte bytes[nachricht.length() + 1]; // Größe byte-Array ermitteln (mit 0-Term.)
nachricht.getBytes(bytes, nachricht.length() + 1);      // Bytes extrahieren

for (int i = 0; i < nachricht.length(); i++){
    if (i > 0){Serial.print(" ");}       // Leerzeichen einfügen
    Serial.print(bytes[i],DEC);  // anzeigen
// Ausgabe:  68 105 101 115 32 105 115 116 32 101 105 110 32 32 84 101 115 116
```

Das heißt, es ist im Byte-Array so hinterlegt:
Bytes[0] = 68, Bytes[1] = 105, Bytes[2] = 101 … usw.

indexOf()

Beschreibung
Lokalisiert ein Zeichen / String innerhalb eines anderen Strings. Standardmäßig wird vom Anfang des Strings begonnen, es kann aber auch vorwärts von einer bestimmten Stelle aus gesucht werden. Das erlaubt das Feststellen von mehrfach vorhandenen Zeichen / Strings innerhalb des vorhandenen Strings.

Syntax
string.indexOf(val)
string.indexOf(val, from)

Parameter
string: eine Variable vom Datentyp String
val: nach was soll gesucht werden (Zeichen oder String)
from: der Punkt, an dem die Suche vorwärts gestartet werden soll

Rückgabewert
Eine Zahl, welche der Stelle entspricht, an dem die gesuchten Zeichen oder Strings gefunden worden sind. Oder -1, wenn das Zeichen oder String NICHT gefunden wurde.

Beispiel
```
String string1 = "- Hallo! Hallo! Hier ist DL1AKP.";

int erstesAusrufezeichen = string1.indexOf('!');
Serial.println(erstesAusrufezeichen);
int zweitesAusrufezeichen = string1.indexOf('!', erstesAusrufezeichen + 1 );
Serial.println(zweitesAusrufezeichen);
int fragezeichen = string1.indexOf('?');  // Wo ist Fragezeichen?
Serial.println(fragezeichen); // Wert ausgeben (Negativ, weil nicht vorhanden)
```

Als Ergebnis werden die Zahlen 7, 14 und -1 ausgegeben. Da die Zählung bei Null beginnt, ist das erste Zeichen die 0. Stelle. Wird das Zeichen nicht gefunden ist der Wert negativ. Man kann also damit feststellen, ob ein Zeichen oder String in einem anderen enthalten ist und wo, vom Anfang beginnend.

lastIndexOf()

Beschreibung
Lokalisiert ein Zeichen oder String innerhalb eines anderen Strings. Standardmäßig wird vom Ende des Strings begonnen, es kann aber auch rückwärts von einer bestimmten Stelle aus gesucht werden. Das erlaubt das Feststellen von mehrfach vorhandenen Zeichen oder Strings innerhalb des vorhandenen Strings.

Syntax
string.lastIndexOf(val)
string.lastIndexOf(val, from)

Parameter
string: eine Variable vom Datentyp String
val: nach was soll gesucht werden (Zeichen oder String)
from: der Punkt, an dem die Suche rückwärts gestartet werden soll

Rückgabewert
Eine Zahl, welche der Stelle entspricht, an dem die gesuchten Zeichen oder Strings gefunden worden sind. Oder -1, wenn das Zeichen oder String NICHT gefunden wurde.

Beispiel
```
String string1 = "- Hallo! Hallo! Hier ist DL1AKP.";

int erstesAusrufezeichen = string1.lastIndexOf('!'); // Wo ist 1. Ausr.Z. ?
Serial.println(erstesAusrufezeichen);   // Wert ausgeben
// Wo ist zweites Ausrufezeichen? Beginne 1 Zeichen hinter dem ersten zu suchen!
int zweitesAusrufezeichen = string1.lastIndexOf('!', erstesAusrufezeichen - 1 );
Serial.println(zweitesAusrufezeichen); // Wert ausgeben
int fragezeichen = string1.lastIndexOf('?');  // Wo ist Fragezeichen?
Serial.println(fragezeichen); // Wert ausgeben (Negativ, weil nicht vorhanden)
```

Hier werden als Ergebnis die Zahlen 14, 7 und -1 ausgegeben. Von hinten beginnend ist an Position 14 das erste Ausrufezeichen. (Achtung! Das Ergebnis gibt die Position von vorn aus gezählt wieder und NICHT wie gedacht von hinten aus!)
Das zweite Ausrufezeichen ist an Stelle 7 von vorn aus gesehen und das Fragezeichen nicht vorhanden (negativer Wert)

length()

Beispiel-Sketch verfügbar: _getBytes_length.ino

Beschreibung
Ermittelt die Länge eines Strings in Anzahl der Zeichen. Hierbei wird ein angehängter Null-Charakter (siehe Erklärung bei String-Arrays) nicht mitgezählt.

Syntax
string.length()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert
Die Länge des Strings in Anzahl der Zeichen

Beispiel
```
String string1 = "Hans";
String string2 = "Hans-Dieter";

x= string1.length();
Serial.println(x); // Ergebnis: 4

x= string2.length();
Serial.println(x);        // Ergebnis: 11
```

replace()

Beschreibung
Die Stringfunktion replace() erlaubt es, ALLE in einem String vorhandenen Zeichen mit einem gewünschten Zeichen zu ersetzen. Dies funktioniert auch mit Teilstrings, welche durch einen anderen Teilstring ersetzt werden sollen.

Syntax
string.replace(substring1, substring2)

Parameter
string: eine Variable vom Datentyp String
substring1: eine Variable vom Datentyp String
substring2: eine Variable vom Datentyp String

Rückgabewert

Ein neuer String, welcher den alten String mit den ersetzten Zeichen oder Teilstrings enthält

Beispiel
```
String string1 = "- Hallo! Hallo! Hier ist DL1AKP.";

string1.replace("Hallo", "Hoh");
Serial.println(string1); // Ausgabe Hallo ist durch Hoh ersetzt
```

setCharAt()

Beschreibung
Setzt ein bestimmtes Zeichen eines Strings. Ist wirkungslos, wenn der String kürzer ist, als die gewünschte Stelle, an der ein Zeichen gesetzt werden soll.

Syntax
string.setCharAt(index, c)

Parameter
string:	eine Variable vom Datentyp String
index:	die Stelle (Index), an der ein Zeichen gesetzt werden soll (beginnend bei 0)
c:	das Zeichen, welches an der Stelle gesetzt werden soll

Rückgabewert
keine

Beispiel
```
String string1 = "Hallo";

string1.setCharAt(1, 'e');
Serial.println(string1); // aus Hallo wird Hello
```

startsWith()

Beschreibung
Ermittelt, ob oder ob nicht ein String mit dem Zeichen eines anderen Strings beginnt.

Syntax
string.startsWith(string2)

Parameter
string:	eine Variable vom Datentyp String
string2:	eine Variable vom Datentyp String

Rückgabewert
true:	wenn string mit dem Zeichen von string2 beginnt
false:	wenn nicht

Beispiel
```
String string1 = "Hallo";
```

```
String string2 = "Ha";
String string3 = "Ho";
boolean x;

x = string1.startsWith(string2);
Serial.println(x);  // Ausgabe "1", weil string1 mit string2 beginnt

x = string1.startsWith(string3);
Serial.println(x);  // Ausgabe "0", weil string1 nicht mit string3 beginnt
```

substring()

Beschreibung

Findet einen Teilstring innerhalb eines Strings. Die Startzahl (beginnend bei 0) ist inklusive (d.h. das entsprechende Zeichen gehört schon zum Teilstring), aber die optionale Endzahl ist exclusive (d.h. das entsprechende Zeichen gehört nicht mehr zum Teilstring). Wenn die Endzahl weggelassen wird, dann geht der Teilstring bis zum Ende des Strings.

Syntax

string.substring(from)
string.substring(from, to)

Parameter

string:	eine Variable vom Datentyp String
from:	die Startposition, an welcher der Teilstring beginnt
to:	(optional) die Endposition, bevor dieser der Teilstring endet

Rückgabewert

Den Teilstring

Beispiel

```
String string1 = "Heute ist kein schönes Wetter";
String string2 = "";

string2 = string1.substring(10);   // beginne an Stelle 10
Serial.println(string2);  // Ausgabe: "kein schönes Wetter"

string2 = string1.substring(10,23);   // von Stelle 10 bis Stelle 22
Serial.println(string2);  // Ausgabe: "kein schönes"
```

toCharArray()

Beschreibung

Kopiert die Zeichen des Strings in einen bereitgestellten Pufferspeicher.
Achtung: Es kann zu unvorhersehbarem Verhalten des Programms bis hin zum Absturz kommen, wenn der Puffer nicht groß genug ist. Also am besten, den Wert len an die Bedürfnisse anpassen.

Syntax

string.toCharArray(buf, len)

Parameter

string:	eine Variable vom Datentyp String
buf:	der Puffer, in den die Zeichen gespeichert werden sollen (*char []*)

len: die Größe des Puffers (*unsigned int*)

Rückgabewert
keine

Beispiel
```
String string1 = "Heute ist schönes Wetter";
char buf[50]; // 49 Zeichen passen in "buf" (+ null-Terminator)

string1.toCharArray(buf, 50);
Serial.print(buf);
```

toInt()

Beschreibung
Konvertiert einen gültigen String in einen long (Integer). Der Quell-String muss mit einer gültigen Integer-Zahl beginnen. Ist das nicht der Fall, stoppt die Umwandlung.

Syntax
string.toInt()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert
long
Wenn der String NICHT mit einer Integerzahl beginnt, dann wird als Ergebnis 0 zurückgegeben.

Beispiel
```
String string1 = "495 Hunde warten im Hof";
long zahl = 0;

zahl = string1.toInt();
Serial.println(zahl);            // ausgegeben wird: 495
```

toFloat()

Beispiel-Sketch verfügbar: toFloat.ino

Beschreibung
Konvertiert einen gültigen String in einen float-Wert (Fließkommazahl). Der String muss dafür mit einer Ziffer beginnen. Wenn die Funktion keine Ziffern mehr findet, endet die Umwandlung. Die Strings „ 45.87“ und „650“ und „650klaus“ ergeben die Fließkommazahlen 45,87 und 650,00 und 650,00.

Syntax
string.toFloat()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert

float
Wenn der String NICHT mit einer Ziffer beginnt, dann wird als Ergebnis 0 zurückgegeben.

Beispiel
```
String string1 = "169.51 Grad Celsius";
float wert;

wert = string1.toFloat();
Serial.println(wert);                    // Ausgabe: 169.51
```

toLowerCase()

Beschreibung
Modifiziert einen String dahingehend, dass er in Kleinbuchstaben umgewandelt wird.

Syntax
string.toLowerCase()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert
Der Originalstring, nur aus Kleinbuchstaben bestehend

Beispiel
```
String string1 = "HALLO LEUTE";

string1.toLowerCase();
Serial.print(string1);  // Ausgabe "hallo leute"
```

toUpperCase()

Beschreibung
Modifiziert einen String dahingehend, dass er in Großbuchstaben umgewandelt wird.

Syntax
string.toUpperCase()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert
Der Originalstring, nur aus Großbuchstaben bestehend

Beispiel
```
String string1 = "hallo LEute";

string1.toUpperCase();
Serial.print(string1);  // Ausgabe "HALLO LEUTE"
```

trim()

Beispiel-Sketch verfügbar: temp_MCP9700.ino

Beschreibung
Entfernt alle bestehenden Leerräume (Leerzeichen) vor und nach dem String.

Syntax
string.trim()

Parameter
string: eine Variable vom Datentyp String

Rückgabewert
Der bearbeitete Originalstring

Beispiel
```
String string1 = "    speicher   ";

Serial.print("Programm");
string1.trim();
Serial.print(string1);
Serial.println("platz"); // Ausgabe: Programmspeicherplatz
```

String Operatoren

[] (Zugriff auf Zeichen)

Beschreibung
Erlaubt den Zugriff auf die einzelnen Zeichen eines Strings

Syntax
char meinZeichen = string1[n]

Parameter
char meinZeichen	Variable (Datentyp char)
string1	ein String
int n	die Stelle, auf die zugegriffen werden soll (beginnend mit 0)

Rückgabewert
Das n-te Zeichen eines Strings. Ergibt das gleiche Ergebnis wie der Befehl ‚**charAt()**‘

Beispiel
```
String string1 = "Programmspeicher";

char mychar = string1[5];
Serial.print(mychar);     // Ausgabe: a  (5. Zeichen bei 0 zählend)
```

+ (Operator „Anhängen“)

Beschreibung
Verbindet, bzw. („hängt an“) zwei Strings zu einem neuen String. Der zweite String wird an den ersten angehängt, das Ergebnis wird in einen neuen String ausgegeben. Funktioniert auf die gleiche Art wie der Befehl **string.concat().**

Syntax
string3 = string1 + string 2;
string3 += string2;

Parameter
string1, string2, string3: String-Variablen

Rückgabewert
Ein neuer String, der eine Kombination der beiden Original-Strings ist.

Beispiel
```
String string1 = "Programm";
String string2 = "speicher";
String string3 = "";

string3 = string1 + string2;
Serial.println(string3);
```

== (Operator „Vergleich“)

Beschreibung

Vergleicht zwei Strings auf Gleichheit. Der Vergleich ist „case-sensitive“, d.h. „hello“ ist nicht identisch mit „Hello“. Funktioniert auf die gleiche Art wie der Befehl **string.equals()**.

Syntax

string1 == string2 Achtung! Zwei = hintereinander!

Parameter

string1, string2: String-Variablen

Rückgabewert

true wenn beide Strings identisch sind
false wenn nicht

Beispiel

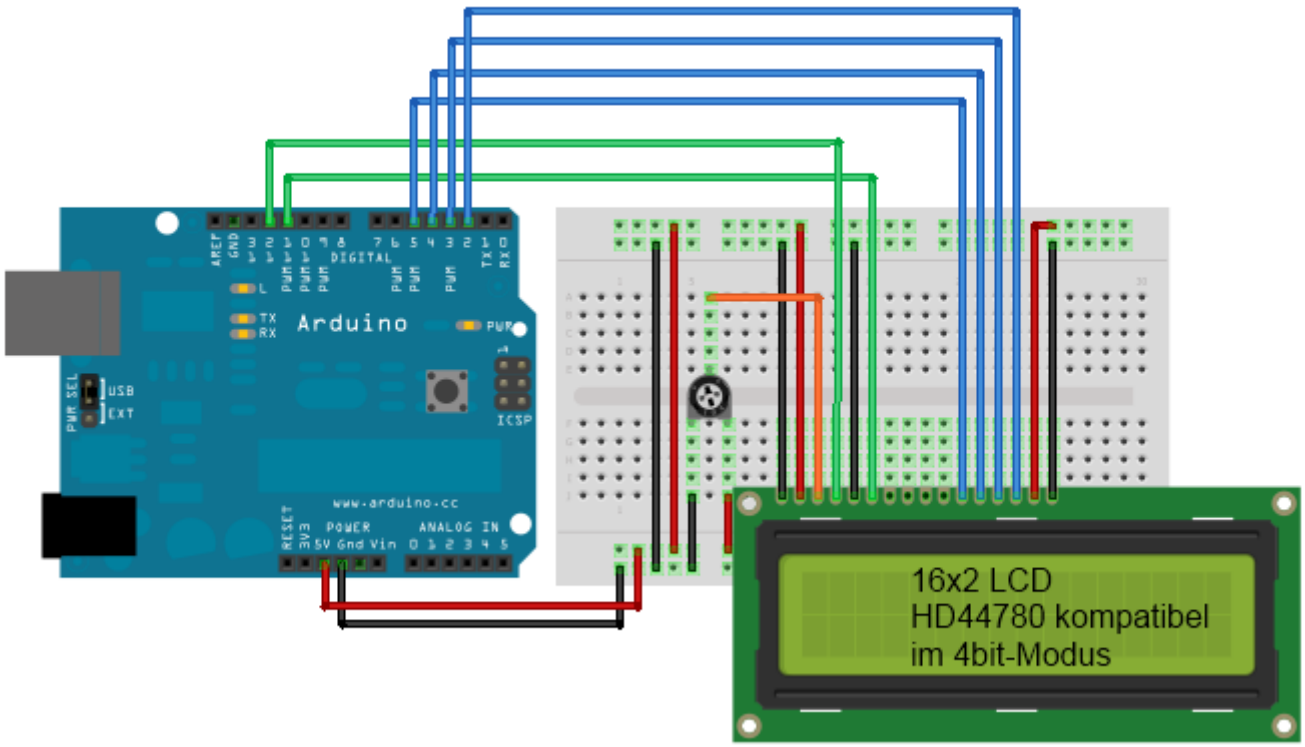
```
String string1 = "Programm";
String string2 = "Klaus";
String string3 = "Programm";
boolean x;

x= string1 == string2;
Serial.println(x);  // Ausgabe 0, weil string1 nicht gleich string2 ist

x= string1 == string3;
Serial.println(x);  // Ausgabe 1, weil string1 gleich string3 ist
```

LCD-Textdisplays und deren Benutzung

Allgemeines

LCD-Anzeigen zählen mit zu den faszinierendsten Bauteilen bei einem Mikrocontroller-Projekt. Sie vermitteln Informationen und sehen einfach super aus, speziell die blauen. Deshalb hier eine Einführung zu den textbasierten LCDs mit dem Chip Hitachi HD44780 und deren Benutzung, Anschlussbelegung und die dazu erforderlichen Befehle (Funktionen).
Die Benutzung eines Grafik-Displays wird in einem gesonderten Kapitel behandelt.
Die folgende Passage inklusive des Bildes wurde aus [4] übernommen:

Dank der Library „LiquidCrystal“ , welche in der ARDUINO-IDE bereits inkludiert ist, ist es sehr einfach ein Text-LCD mit dem ARDUINO zu verwenden: Die blauen Leitungen entsprechen den Datenleitungen (4 im 4bit-Modus, sonst 8), rot/schwarz auf der linken Seite sind die Versorgungsspannung des Chips und rot/schwarz auf der rechten Seite dienen der LED-Hintergrundbeleuchtung. Das orangene Kabel dient zur Einstellung des Kontrastes, wobei das Potenzial am mittleren Pin eines 10 kΩ-Poti‘s abgegriffen wird. Die beiden grünen Kabel (register select und enable) dienen der Kommunikation zwischen ARDUINO und LCD, das mittlere schwarze legt fest, dass wir das LCD nur zur Ausgabe ("Schreiben") benutzen wollen.



Verdrahtung des LCD. Es ist ein zweizeiliges Display. Natürlich können nicht drei Zeilen dargestellt werden.

Tipp: Die Kontrasteinstellung soll eigentlich mit einem Potenzial zwischen 0 und 1,5 V erfolgen. Wenn man lediglich ein Trimmwiderstand verwendet, spielt sich die Kontrastveränderung in einem sehr kleinen Drehbereich ab. Zur Abhilfe kann ein großer Widerstand (z.B. 20 kΩ) zum Trimmwiderstand in Reihe (oberhalb von +UB ausgehend) geschaltet werden. Damit ist die Potenzialdifferenz am Trimmwiderstand selbst deutlich geringer.

Auf den folgenden Seiten werden die Befehle zur Arbeit mit LCD's beschrieben, und auch an Beispielen erklärt. Wichtig ist es, dem ARDUINO die Verwendung der verschiedenen Leitungen (Verbindungen) zum Display mitzuteilen. Die Benutzung des 4bit-Bus-Modus ist wohl zu 99% ausreichend.

Der **8bit-Bus-Modus** verbraucht im Unterschied zum 4bit-Bus-Modus 4 Pins des ARDUINO mehr. Der Unterschied ist, dass dadurch die Übertragungsgeschwindigkeit zum LCD steigt und hiermit das Schreiben eigener Routinen einfacher ist (wenn man keine Library benutzt). So können hier 8 Bit eines Bytes gleichzeitig übertragen werden. Beim 4bit-Bus-Modus werden je vier Bit nacheinander übertragen (diese 4 Bit nennt man Nibble). Es wird zuerst das höherwertige Nibble, also Bit 4 bis Bit 7 übertragen, danach das niederwertige Nibble, Bit 0 bis Bit 3. Quelle [22].
Das dauert natürlich länger… Wobei „lang" hier sicherlich zu vernachlässigen ist. Ich persönlich hatte nie ein Problem mit dem 4bit-Bus-Modus.
Eine Überlegung wert ist auch das LCD mit **I2C-Bus**, spart man hier doch noch mehr ARDUINO-Pins ein. Dieser Übertragungsart widmet sich ein eigenes Kapitel.

LiquidCrystal()

Beispiel-Sketch verfügbar: LCD.ino

Beschreibung
Erzeugt eine Variable vom Typ LiquidCrystal. Das dient dazu, dem ARDUINO die Pinbelegung zur Verbindung mit dem Display mitzuteilen. Die nichtbenutzten Anschlüsse D0 bis D3 des Displays können einfach unbeschaltet bleiben. RW kann direkt mit GND verbunden werden.

Syntax
```
LiquidCrystal(rs, enable, d4, d5, d6, d7)                    // 4bit-Mode (bevorzugt!!)
LiquidCrystal(rs, rw, enable, d4, d5, d6, d7)                // 4bit-Mode mit RW
LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)    // 8bit-Mode
LiquidCrystal(rs, rw, enable, d0, d1, d2, d3, d4, d5, d6, d7) // 8bit-Mode mit RW
```

Parameter
rs:	der ARDUINO-Pin, welcher mit RS des Displays verbunden ist
rw:	der ARDUINO-Pin, welcher mit RW des Displays verbunden ist (optional, A.N.)
enable:	der ARDUINO-Pin, welcher mit enable (E) des Displays verbunden ist
d0…d7:	Die Pins des ARDUINO, welche mit den entsprechenden Datenleitungen D0 bis D7 des Displays verbunden sind

Rückgabewert
keine

Beispiel
```
#include <LiquidCrystal.h>   // binden Bibliothek LiquidCrystal ein
LiquidCrystal lcd(12, 11, 5, 4, 3, 2); // initialisieren Bibliothek LCD-Pins

void setup()
{
  lcd.print("Hello, world!");
}
```

begin()

Beispiel-Sketch verfügbar: LCD.ino

Beschreibung
Beschreibt die Dimensionen des Displays, also Breite (Anzahl der Zeichen) und Höhe (Zeilen)

Syntax
lcd.begin(cols, rows)

Parameter
lcd: eine Variable vom Typ LiquidCrystal
cols: Anzahl der Zeichen (16, 20, 40…)
rows: Anzahl der Zeilen (1, 2, 4…)

Rückgabewert
keine

Beispiel
```
void setup()
{
  lcd.begin(16, 2); // Display hat 16 Zeichen x 2 Zeilen
}
```

clear()

Beispiel-Sketch verfügbar: LCD.ino

Beschreibung
Löscht das Display und setzt den Cursor in die obere linke Ecke.

Syntax
lcd.clear()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.clear();   // Display löschen
```

home()

Beschreibung
Setzt den Cursor in die obere linke Ecke. Wird benutzt, um folgenden Text oben links beginnend anzuzeigen. Soll das Display auch gelöscht werden, dann benutzen Sie den Befehl clear() stattdessen.

Syntax
lcd.home()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.home();     // Cursor links oben setzen
```

setCursor()

Beispiel-Sketch verfügbar: LCD.ino

Beschreibung
Setzt den Cursor an eine gewünschte Position. Also nicht links oben, wie beim Befehl home(), sondern an eine beliebige Position. Dort wird der folgende Text beginnend angezeigt.

Syntax
lcd.setCursor(col, row)

Parameter
lcd: eine Variable vom Typ LiquidCrystal
col: Die Zeichen-Stelle, an welcher der Cursor gesetzt werden soll
row: die Zeile, in welcher der Cursor gesetzt werden soll

Rückgabewert
keine

Beispiel
```
lcd.setCursor(0, 1); // Cursor auf Position 1, Zeile 2 (0, 1) setzen
```

write()

Beschreibung
Schreibt ein Zeichen auf das Display.

Syntax
lcd.write(data)

Parameter
lcd: eine Variable vom Typ LiquidCrystal
data: das Zeichen, welches geschrieben werden soll

Rückgabewert
Eine Byte-Zahl: Wird NUR der Befehl write() ohne data benutzt, dann gibt er die Anzahl geschriebener Bytes zurück. Aber das Lesen dieser Anzahl Bytes ist optional. Der Befehl dient primär dazu, Zeichen an das Display zu senden.

Beispiel

```
void loop()    //Hauptprogramm
{
  if (Serial.available()) {    // wenn Daten über die serielle Schnittstelle da sind
    lcd.write(Serial.read());  // dann zeige sie an!
  }
}
```

print()

Beschreibung

Druckt Text auf das Display

Syntax

lcd.print(data)
lcd.print(data, BASE)

Parameter

lcd:	eine Variable vom Typ LiquidCrystal
data:	die Daten, welche gedruckt werden sollen (Datentypen: char, byte, int, long, string)
BASE:	optional, Die Basis, in welcher die Daten gedruckt werden sollen (BIN=binär, DEC=dezimal, OCT=oktal, HEX=hexadezimal)

Rückgabewert

Eine Byte-Zahl: Wird NUR der Befehl print() ohne data benutzt, dann gibt er die Anzahl geschriebener Bytes zurück. Aber das Lesen dieser Anzahl Bytes ist optional. Der Befehl dient primär dazu, Zeichen an das Display zu senden.

Beispiel

```
lcd.print("hello, world!");  //…
```

cursor()

Beschreibung

Zeigt den Cursor an. Ein Unterstrich erscheint an der Stelle, an welcher dann der Text erscheint.
Das Gegenteil ist noCursor().

Syntax

lcd.cursor()

Parameter

lcd:	eine Variable vom Typ LiquidCrystal

Rückgabewert

keine

Beispiel

```
lcd.cursor();          //Cursor anzeigen
```

noCursor()

Beschreibung
Versteckt den Cursor. Das Gegenteil ist cursor().

Syntax
lcd.noCursor()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.noCursor();         //Cursor verstecken
```

blink()

Beschreibung
Zeigt den blinkenden Cursor an. Wenn es benutzt wird in Verbindung mit cursor(), dann hängt das Ergebnis von dem jeweils benutzten Displaytyp ab.

Syntax
lcd.blink()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.blink();    //Cursor anzeigen und blinken lassen
```

noBlink()

Beschreibung
Schaltet den blinkenden Cursor ab.

Syntax
lcd.noBlink()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert

keine

Beispiel
```
lcd.noBlink();          //Cursor blinken abschalten
```

display()

Beschreibung
Schaltet das Display ein, nachdem es durch noDisplay() abgeschaltet wurde. Es wird der Text und Cursor wiederhergestellt, wie er vorher war.

Syntax
lcd.display()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.display();          //LCD einschalten
```

noDisplay()

Beschreibung
Schaltet das Display ab, ohne das der angezeigte Text verloren geht. Mit display() kann es wieder eingeschaltet werden, und der vorher dargestellte Inhalt wird wieder angezeigt.

Syntax
lcd.noDisplay()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.noDisplay();              //LCD  abschalten
```

scrollDisplayLeft()

Beschreibung
Schiebt den Inhalt des Displays (Text UND Cursor) um eine Stelle nach links

Syntax
lcd.scrollDisplayLeft()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.scrollDisplayLeft();              // eine Stelle nach links
```

scrollDisplayRight()

Beschreibung
Schiebt den Inhalt des Displays (Text UND Cursor) um eine Stelle nach rechts

Syntax
lcd.scrollDisplayRight()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.scrollDisplayRight();             // eine Stelle nach rechts
```

autoscroll()

Beschreibung
Schaltet das automatische Scrollen des Displays an. Diese Funktion erzwingt das Weiterschieben der vorhergehenden Zeichen auf dem Display um eine Stelle, wenn ein neues Zeichen ausgegeben wird. Wenn die aktuelle Textrichtung links-nach-rechts ist (Standardeinstellung), dann wird der Text um eine Stelle nach links geschoben (gescrollt). Ist die Textrichtung rechts-nach-links, dann wird der Text um eine Stelle nach rechts geschoben. Dadurch wird jedes neue Zeichen, welches an das Display gesendet wird, an genau derselben Stelle angezeigt.

Syntax
lcd.autoscroll()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.autoscroll();      // Autoscroll einschalten
```

noAutoscroll()

Beschreibung
Autoscroll abschalten (siehe oben)

Syntax
lcd.noAutoscroll()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.noAutoscroll();    // Autoscroll aus
```

leftToRight()

Beschreibung
Setzt die Schreibrichtung des Displays auf links-nach-rechts, das ist die Standardeinstellung. Das bedeutet, dass alle nachfolgend auf das Display gesendeten Zeichen von links nach rechts erscheinen. Es hat keinen Einfluss auf bereits an das Display gesendete Zeichen.

Syntax
lcd.leftToRight()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.leftToRight();     //
```

rightToLeft()

Beschreibung
Setzt die Schreibrichtung des Displays auf rechts-nach-links. Das bedeutet, dass alle nachfolgend auf das Display gesendeten Zeichen von rechts nach links erscheinen. Es hat keinen Einfluss auf bereits an das Display gesendete Zeichen.

Syntax
lcd.rightToLeft()

Parameter
lcd: eine Variable vom Typ LiquidCrystal

Rückgabewert
keine

Beispiel
```
lcd.rightToLeft();      //
```

createChar()

Beschreibung
Erzeugt ein frei selbst zu erstellendes Zeichen. Bis zu maximal 8 verschiedene Zeichen (0 bis 7) von 5x8 Pixeln jeweils sind möglich. Das Aussehen jedes einzelnen Zeichens wird festgelegt durch ein Datenarray von 8 Bytes, eines für jede Reihe von Pixeln.

Syntax
lcd.createChar(num, data)

Parameter
lcd: eine Variable vom Typ LiquidCrystal
num: welches Zeichen soll erstellt werden (0 bis 7)
data: die Pixeldaten des neuen Zeichens

Rückgabewert
keine

Beispiel
```
#include <LiquidCrystal.h>     //LCD-Library einbinden

LiquidCrystal lcd(12, 11, 5, 4, 3, 2);  // Pinbelegung LCD und ARDUINO

byte smiley[8] = {             //erstellt Zeichen Smiley
  B00000,
  B10001,
  B00000,
  B00000,
  B10001,
  B01110,
  B00000,
};

void setup() {                 //Setup, wird einmal durchlaufen.
  lcd.createChar(0, smiley);  //erstelltes Zeichen benutzen
  lcd.begin(16, 2);            //LCD definieren mit 16zeichen, 2 Zeilen
  lcd.write(byte(0));          //Zeichen anzeigen
}

void loop() {}                 //Hauptprogramm
```

Sonderzeichen, Symbole und Umlaute

Beispiel-Sketch verfügbar: LCD_Char.ino

Allgemeines

Die LCD-Anzeigen können mehr, als nur einfachen Text anzuzeigen. Im Speicher des LCD-Moduls ist ein ganzes Arsenal an zusätzlichen Zeichen abgespeichert. Das sind z.B. die deutschen Umlaute, Sonderzeichen wie { } ° µ Ohm-Zeichen, und weitere Symbole etc. Um die Benutzung dieser Zeichen, wie man sie darstellt und anzeigt, geht es in diesem Abschnitt. Danke hierfür an Gregor Hebecker und auch René (via E-Mail), der mir diese Infos und auch ein .ino-File zur Verwendung in dieser Referenz zur Verfügung stellte.

So ohne weiteres kann man keine Sonderzeichen oder Symbole mit lcd.print auf das Display bringen. Man muss sich hierfür die integrierten Zeichen zu Nutze machen. Sie sind in einer Art Tabelle im LCD-Display abgelegt und können über ihren hexadezimalen Code angezeigt werden.
Die nachfolgende Kurzübersicht zeigt die wichtigsten Sonderzeichen und ihren HEX-Code:

ä : \xE1
ö : \xEF
ü : \xF5
ß : \xE2

° :	\xDF	z.B. in °C (Grad Celsius)
µ :	\xE4	z. B. in µF (Mikrofarad) oder µC (Mikrocontroller)
Ω :	\xF4	das große Omega steht für die Einheit des Widerstandes (Ohm)
π :	\xF7	Zahl PI (3,14…)
√ :	\xE8	Quadratwurzel
∞ :	\xF3	unendlich
∑ :	\xF6	Summensymbol

Zu beachten ist auch, das im ROM des Displaymoduls die hinterlegten Zeichen oberhalb 127(DEZ) nicht fest definiert sind. Hier muss gegebenenfalls das Datenblatt zu Rate gezogen werden, will man spezielle Zeichen korrekt darstellen.

Benutzung im Sketch

Nun wollen wir die Zeichen bei Bedarf auch korrekt auf dem Display anzeigen lassen.
Zur Anzeige des Grad-Zeichens bei einer Temperatur kann man es wie folgt machen:

```
lcd.print("Temp: " myTemp "\xDF" "C");        // °C mit Variable myTemp
```

Eine weitere Möglichkeit ist es, das gewünschte Zeichen vorher zu definieren. Das macht den Code auch gleich einfacher lesbar. Hier am Beispiel des Ohm-Zeichens:

```
#define OHM   "\xF4"                  // vor setup() einfügen
#define char tgrad[] = "\337C";       // oktal 337 entspricht dem °Zeichen
lcd.print( "480" OHM );               // Anzeige:  480 Ω
lcd.print("37"); lcd.print(tgrad)     // Anzeige:  37°
```

Das Wort "Datenübertragung" enthält ein ü.
ein Umlaut, der nicht ohne weiteres auf das LCD hingeschrieben werden kann.
Die Lösung wäre z. B. eine sogenannte Escape-Sequenz mit der Hexadezimalzahl F5:

```
lcd.print("Daten");
lcd.print("\xF5");              // das ist ein ü
lcd.print("bertragung");
```

oder kürzer und eleganter so:

```
lcd.print("Daten" "\xF5" "bertragung");       // Datenübertragung
```

Dieses seltsame "\xF5" ist wie folgt aufgebaut:

"":	Anführungszeichen, da es sich um einen String handelt
\ :	Backslash, da es sich um eine Escapesequenz handelt
x :	wegen Hexadezimalzahl
F5:	diejenige Hexadezimalzahl, die ein ü repräsentiert

LCD-Grafikdisplays und deren Benutzung

Allgemeines

Dieser Artikel entstand dank der Zuarbeit von Renè (via Email). Die Beschaltung und der Sketch konnte von mir noch nicht überprüft werden, da ich selbst ein solches Display nicht besitze.
Grafikdisplays gestatten die Darstellung pixelgenau, d.h. man ist nicht auf einen bestimmten Zeichensatz festgelegt. Mit einem Textdisplay ist man auf den dort implementierten Zeichensatz beschränkt. Ein Grafikdisplay jedoch kann frei beschrieben werden, bei Bedarf jeden Pixel einzeln.

Die verbreitetsten Displays dieser Art haben in der Horizontale 128 Pixel und in der Vertikale 64 Pixel und sind hintergrundbeleuchtet. Sie sind sehr günstig in verschiedenen Farben bei einschlägigen Händlern zu bekommen.
Der dort verwendete Controller folgt ähnlich wie bei den Textdisplays (dort ist es der HD44780 oder kompatible) einer gewissen Normung, sodass es nur bestimmte Controller gibt.

Wir verwenden hier wieder eine Library, die nahezu alle (wenn nicht gar wirklich alle) Grafikdisplays unterstützt. Es ist die Library „U8G2“, welche über die Bibliotheksverwaltung der IDE geladen werden kann.

Wir befassen uns hier mit den universellen und mit reichhaltigen Funktionen ausgestatteten QC12864, die eine Auflösung von 128x64 Pixeln bieten.
Die Displays gibt es in verschiedenen Ausführungen verschiedenster Lieferer. Das Display gibt es für wenige Euro auf verschiedenen Marktplätzen. Die Grundkonfiguration ist immer die Gleiche. Die Controller der Displays sind ST7290. Diese können sowohl im Parallelmodus mit 4 oder 8 Datenleitungen, als auch im seriellen Modus mit 2 Datenleitungen betrieben werden.
Um dem chronischen PIN-Mangel am ARDUINO gerecht zu werden, wird hier nur die serielle Verarbeitung betrachtet.

Anschluss an den ARDUINO

Den Displays dieser Gattung liegt die folgende PIN-Konfiguration zugrunde:

Pin-Nr.	Funktion
1	VSS (0V)
2	VDD (+5V)
3	V0 (Kontrast)
4	RS (RegisterSelect)
5	R/W (Register read/write)
6	E (Enable)
7-14	DB0-DB7
15	PSB (Parallel-Seriell-Select)
16	NC
17	RESET
18	VEE/NC
19	Anode (+5V Hintergrundbeleuchtung)
20	Kathode

Für die folgende Beschreibung sind neben der Spannungsversorgung nur die PINs 4,5,6 und 15 von Bedeutung.

Für alle anderen Anwendungsfälle wird auf Datenblätter verwiesen, siehe hierzu Quelle [24].

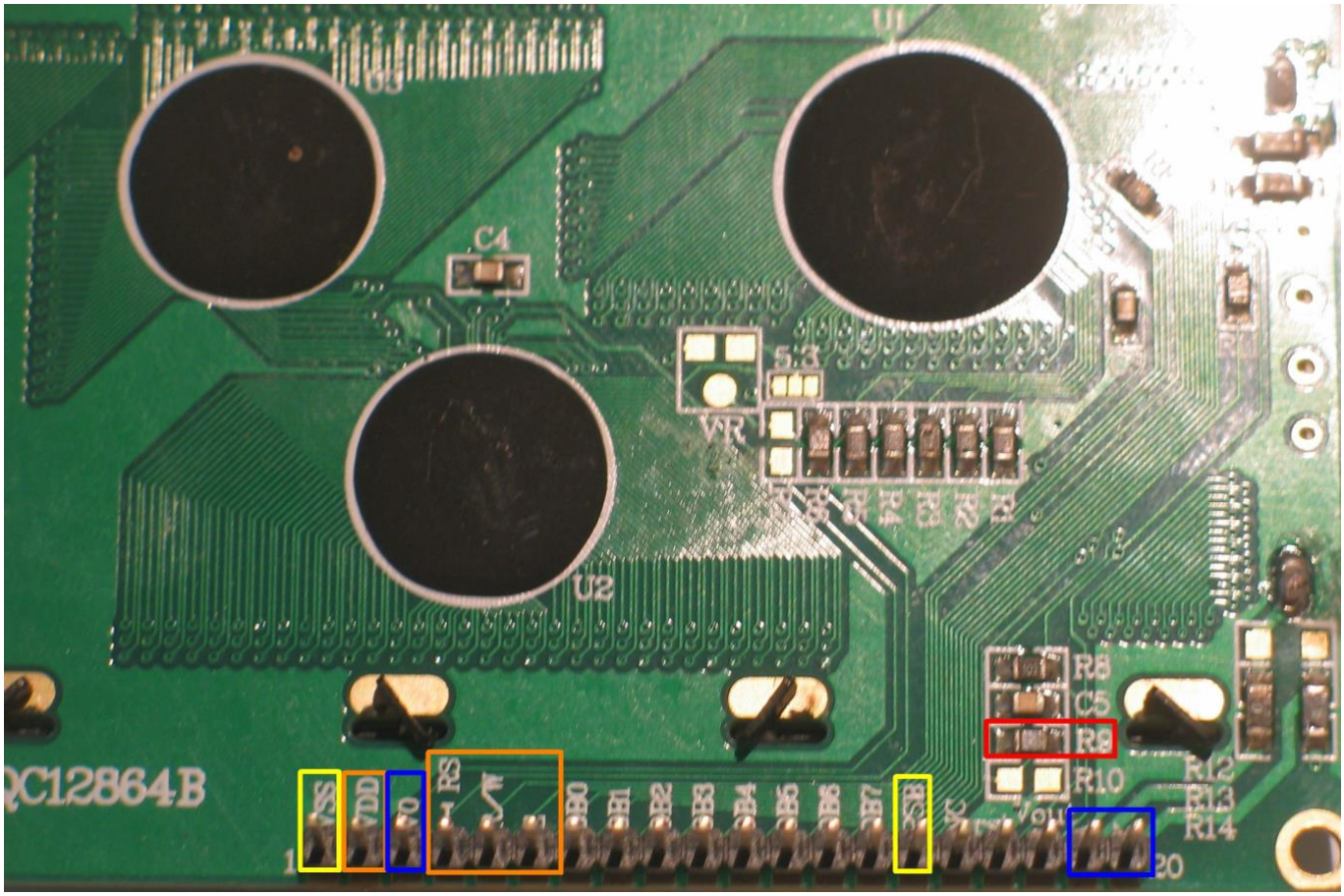
Oft kommt das Display ohne angelötete Stiftleiste. Darüber hinaus gibt es Displays mit anderer Beschaltung von PIN 3 und PIN 18. Vereinzelt gibt es auch Modelle mit 3,3V Versorgungsspannung, die hier nicht nutzbar sind.

Die Hintergrundbeleuchtung liegt direkt an 5V, allerdings ist die Leistungsaufnahme sehr hoch, weshalb es sich empfiehlt, diese extern zu beschalten.
Der PIN 3 dient der Kontrasteinstellung, eine Beschaltung muss ggf. mit Poti analog HD44780 erfolgen.

Eine Besonderheit nach der Beschaffung ist, dass oftmals der PIN 15(PSB) der für die Auswahl Parallel- oder Serialmode verantwortlich ist, fest verdrahtet wurde.
Dies lässt sich aber leicht verändern.
Im folgenden Bild ist die Rückseite des Displays zu erkennen. Dort ist ein Widerstand (R9 - 0Ohm) rot markiert. Wenn dieser bestückt ist, wird PIN PSB auf +5V gezogen. Würde jetzt am PIN PSB 0V angelegt werden, kommt es zum Kurzschluss.

Dieser Widerstand muss entfernt werden, um mit dem PIN PSB selbst den Betriebsmodus auswählen zu können. Wer ein Ohmmeter hat, kann nach dem entlöten zwischen PIN PSB und VDD messen. Der Widerstand sollte größer 5Kohm sein.
Für den seriellen Modus wird jetzt der PIN mit VSS(0V) verbunden.

Die PIN haben dann folgende Bedeutung: PIN 1+2 Stromversorgung, 3 – Kontrast (bleibt im Beispiel unbeschaltet), 4 - ChipSelect (CS), 5 - SerialData (SDA), 6 - SerialClock (SCL), 17 - Reset (optional), 19+20 Backlight (optional)



Die Verdrahtung mit dem ARDUINO UNO erfolgt in diesem Beispiel wie folgt:

PIN an LCD-Display	PIN ARDUINO UNO
1	GND

2	+5V
4	10
5	11
6	13
17	8 (OPTIONAL - siehe Beispielcode)
15	Brücke nach GND

Programmierung

Softwareseitig wird die U8g2-LIB benötigt. Es gibt eine speziell auf den ARDUINO zugeschnittene "U8g2_Arduino-master"-LIB, die in die ARDUINO IDE eingebunden werden muss. Wie oben schon gesagt, geht das entweder über die Bibliotheksverwaltung oder als zip-File direkt an die richtige Stelle, je nach Betriebssystem.
Diese LIB ist so umfangreich, dass es von Vorteil ist, die Dokumentation dazu zu lesen.

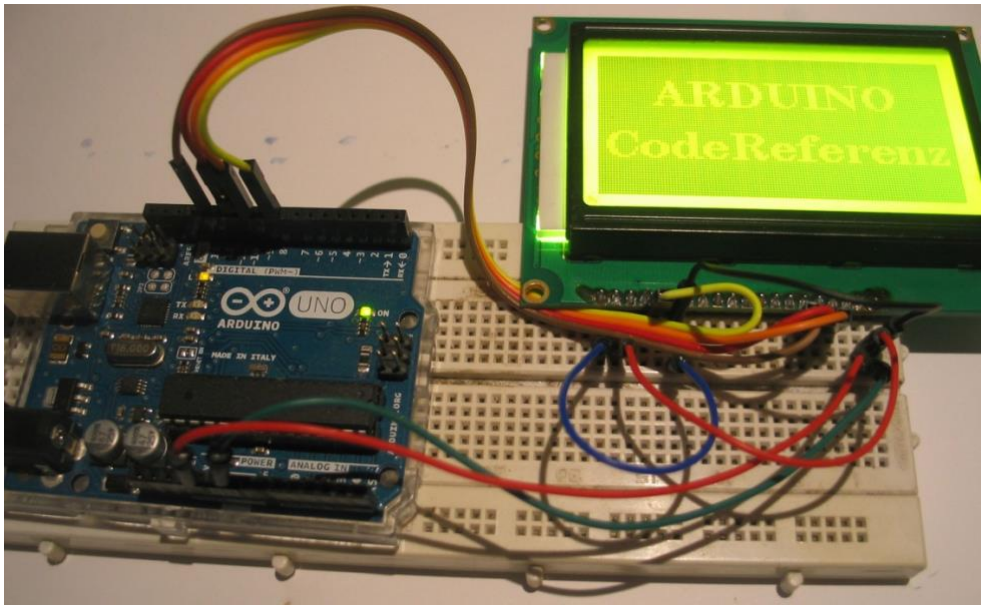
Im Folgenden wird anhand eines Beispielsketches die Grundfunktion erläutert.

```
#include <U8g2lib.h>  // Lib für Grafikdisplay
```

Die folgende Zeile ist der Constructor zur Initialisierung des Displays
Funktion: U8G2
Controller: ST7920
Display: 128x64
Puffer: 1 = 128byte (2 = 256byte F = 1024byte)
Ansteuerung: SW_SPI
Bezeichner: u8g2 (analog lcd=LiquidCrystal())
Werte in Klammer: (Rotation: keine, CLK, DATA, CS[, RESET])
Die gesamte Referenz zu den möglichen Befehlszeilen findet sich im Wiki der Library in Quelle[25]

```
U8G2_ST7920_128X64_1_SW_SPI u8g2(U8G2_R0, 13, 11, 10, 8); // hier mit optionalem
Resetpin für vollständige Darstellung

void setup() {
  u8g2.begin();               // Display initialisieren
}

void loop() {
  u8g2.firstPage();           // erste Seite
  do {
    u8g2.setFont(u8g2_font_ncenB12_tr); // stellt Schriftgröße ein - Größe 12 kannn
    auch z.B. 10 oder 14 sein
    u8g2.drawStr(20,24,"ARDUINO"); // 1.Zeile ab hor. Pixel 20, vert. Pixel 24
    u8g2.drawStr(2,48,"CodeReferenz"); // 2.Zeile ab hor. Pixel 2, vert. Pixel 48
   } while ( u8g2.nextPage() );
}
```

Im Ergebnis sieht der Aufbau und die Ausgabe dann wie folgt aus:



Dort als auch im Sketch ist RESET beschaltet, ist aber für die Funktion nicht notwendig.
Mit diesem Display kann man aber nicht nur grafischen Text mit verschiedenen Schriftarten darstellen.
NEIN ! **Auch Grafikelemente wie Kreise, Ellipsen, Rechtecke, Quadrate, Linien, Dreiecke... Hier ist wirklich probieren gefragt, um diese schönen Teile voll auszureizen.**

In diesem Sinne: Viel Spaß beim Probieren!

EEPROM schreiben und lesen

Allgemeines

Im Programmfluss hat sicher schon jeder berechnete Werte, ausgelesene Sensoren, Zustand von Eingangspins, Zeichen über den RS232-Port oder anderes im SRAM des ARDUINO zwischengespeichert, um ihn später wieder zu verwenden. Oder etwas damit zu berechnen. Das geht prima, der SRAM ist auch mit 2048 Byte bei einem Uno recht umfangreich. Das Speichern geht einfach von der Hand, man merkt davon nichts, der Compiler kümmert sich darum. Es gibt nur einen Haken: Mit dem Wegfall der Versorgungsspannung verschwinden diese gespeicherten Daten.
Was aber, wenn etwas dauerhaft gespeichert werden soll: Soll z.B. ein bestimmter Zustand, der nach einschalten eingenommen werden soll bei einer Steuerung, eine ganz bestimmte Schaltstellung verschiedener Relais, oder ein beim letzten Mal ausgelesener Wert soll nach Einschalten wieder weiterverarbeitet werden?
Eine solche dauerhafte Speicherung, auch ohne Betriebsspannung, ist unter Nutzung des sogenannten EEPROM möglich. EEPROM heißt „elektrisch löschbarer und programmierbarer Nur-Lese-Speicher“ Seine besondere Eigenschaft ist, das die gespeicherten Informationen auch ohne Spannungsversorgung erhalten bleiben.
Sein Nachteil ist eine endliche Lebensdauer. Es kann jede Speicherzelle etwa 100.000-mal beschrieben werden. Das garantiert der Hersteller. Danach geht es immer noch, aber nicht garantiert. Wen das näher interessiert, der findet sicher Massen Infos im Netz. Aber man muss sich nicht mit unnützem Wissen belasten, bei einigermaßen sinnvoller Programmierung werden wir die Lebensdauer des EEPROM nicht ausreizen.
Das klingt erst mal extrem viel… Ist es aber nicht. Wenn man eine solche Speicherung des EEPROM in die loop() schreibt, sind die 100.000 Schreibzyklen nach wenigen Minuten verbraucht. Es ist also eine kluge Programmierung erforderlich, sodass die Werte dort wirklich nur gespeichert werden, wenn es nötig ist. Und nicht einfach immer wieder. Auch ist es sinnvoll, nicht immer alle Werte auf eine bestimmte Speicherstelle zu schreiben, sondern eventuell diese Speicherstelle zu variieren, wenn sehr oft Werte geschrieben werden. So kann man die Lebensdauer auch optimieren, falls erforderlich. Bei wenigen Werten und geringer Speicherrate (selbst einmal pro Stunde wäre kein Problem, eine garantierte Lebensdauer von mindestens 11 Jahren ist so drin)
Weiterhin dauert das Speichern der Werte im EEPROM relativ „lange“, etwa 3 Millisekunden. Das ist bei der Programmierung zeitkritischer Anwendungen zu beachten.

Die EEPROM-Library

Es gibt natürlich eine Library zum Schreiben und Lesen des EEPROM, was die Benutzung relativ einfach macht. Weitere Erklärungen zur Library gibt es bei [2] in englischer Sprache. Besonders interessant finde ich den Operator EEPROM[]. Kann doch dieser genauso behandelt werden wie ein Daten-Array. Das macht es meiner Meinung nach besonders einfach. Zum Datentyp Array siehe hier.

EEPROM.read()

Beschreibung

Liest ein Byte aus dem EEPROM. Speicherstellen, die niemals vorher beschrieben worden sind, haben den Wert 255.

Syntax

EEPROM.read(addresse);

Parameter
addresse: die Speicherstelle, von der gelesen werden soll, beginnend bei 0. (Datentyp int)

Rückgabewert
Der in der Speicherstelle “adresse” hinterlegte Wert (Datentyp byte) 0…255

Beispiel
```
#include <EEPROM.h>
int x = 0;  // Speicherzelle
int wert;   // ausgelesener Wert

void setup()
{
  Serial.begin(9600);   // ser. Schnittstelle initialisieren
}

void loop(){
  wert = EEPROM.read(x);  // Wert aus Speicherzelle x an Variable wert geben

  Serial.print("Speicherzelle Nr.: "); Serial.print(x); // alles anzeigen
  Serial.print("\t");  Serial.print("Wert: "); Serial.print(wert);  Serial.println();

  x = x + 1;  // nächste Speicherzelle
  delay(50);
  if (x == 255) while(1); // bei Speicherzelle 255 Programm stoppen
}
```

EEPROM.write()

Beschreibung
Schreibt ein Byte in eine Speicherzelle des EEPROM

Syntax
EEPROM.write(addresse, wert);

Parameter
adresse	die Speicherzelle, in die geschrieben werden soll, beginnend mit 0 (Datentyp int)
wert	der Wert, welcher in die Speicherzelle geschrieben werden soll (Datentyp byte)

Rückgabewert
Keiner

Beispiel
```
#include <EEPROM.h>   // Lib einbinden

void setup(){
  for (int x = 0; x < 255; x++) // Speicherzelle 0...254 auswählen
    EEPROM.write(x, x); // Speicherzelle 0...254 mit Wert 0...254 schreiben
}

void loop(){
  // weiterer Code
}
```

EEPROM.update()

Beschreibung
Schreibt ein Byte in den EEPROM. Der Wert wird nur geschrieben, wenn er von dem bereits an dieser Speicherzelle abgelegten Wert abweicht.

Syntax
EEPROM.update(addresse, wert);

Parameter
adresse	die Speicherzelle, in die geschrieben werden soll, beginnend mit 0 (Datentyp int)
wert	der Wert, welcher in die Speicherzelle geschrieben werden soll (Datentyp byte)

Rückgabewert
Keiner

Beispiel
```
EEPROM.update(5, 128); // schreibt den Wert 128 in Zelle 5 nur, wenn er
                       // nicht schon 5 ist.
```

EEPROM.put()

Beschreibung
Schreibt jeden beliebigen Datentyp oder Object (struct) in den EEPROM. Man muss sich nicht darum kümmern, wie viele Speicherzellen belegt werden. Diese Methode funktioniert ähnlich der update()-Methode. Das heißt, der Wert wird nur geschrieben, wenn er vom vorher geschriebenen abweicht.

Syntax
EEPROM.put(addresse, wert);

Parameter
adresse	die Speicherzelle, in die geschrieben werden soll, beginnend mit 0 (Datentyp int)
wert	der Wert, welcher geschrieben werden soll (jeder Datentyp, oder auch Object

Rückgabewert
Keiner

Beispiel
```
#include <EEPROM.h>   // Lib einbinden

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

  float x = 56.8453;  // Variable zum Speichern
  int addresse = 0;   // Speicherstelle, wo gespeichert werden soll
  EEPROM.put(addresse, x);  // schreibt (updated) den Wert in Speicherstelle
  addresse += sizeof(float); // eine Speicherstelle hinter das Ende von float x
springen
  EEPROM.put(addresse, "Hallo Hans");
}

void loop(){
```

```
  // weiterer Code
}
```

EEPROM[]

Beschreibung
Mit dieser Methode können EEPROM-Speicherzellen beschrieben werden, als wären sie ein Daten-Array.

Syntax
EEPROM[addresse]

Parameter
adresse die Speicherzelle, in die geschrieben werden soll, beginnend mit 0 (Datentyp int)

Rückgabewert
Der Wert der Speicherzelle

Beispiel
```
wert = EEPROM[ 0 ];    // Liest den Wert aus Speicherzelle 0
EEPROM[ 0 ] = wert;    // Schreibt wert in Speicherzelle 0
```

Timer-Interrupts und deren Benutzung

Allgemeines

Quelle: [6]
Die hier beschriebene Verfahrensweise basiert auf der Library „Timer1“. Diese wie auch die Beschreibung in Englisch, kann von [6] heruntergeladen werden. Diese Anleitung entstand daher, weil ich von BASCOM her diese Herangehensweise kenne. Durch Verwendung des Timers1 können einfach Programmanweisungen nach einer genau definierten Zeit ausgeführt werden.
Beispiele hier sind das Blinken von Anzeige-LEDs, Sekunden-Timer etc.

Die Library ist eine Sammlung von Routinen zur Einrichtung des 16Bit-Hardware-Timers, genannt Timer1 beim ATMega 168/328. Es gibt 3 verschiedene Hardware-Timer bei diesem Chip. Diese können auf verschiedene Art konfiguriert werden, um eine ganze Reihe von Funktionen zu erreichen. Die Entwicklung dieser Library begann ursprünglich, um die PWM-Periodendauer oder PWM-Frequenz einfach und schnell zu verändern. Sie ist aber immer mehr gewachsen und beinhaltet nun auch Timer-overflow-Interrupts und andere Features.

Die Genauigkeit der Zeiten des Timers hängen vom Prozessortakt (beim ARDUINO 16MHz) und des Vorteilers ab. Dieser Vorteiler, genannt Prescaler, kann auf die Werte 1, 8, 64, 256 oder 1024 eingestellt werden.

Für 16MHz:

Prescaler	**Time per Counter-Tick**	**Max. Dauer**
1	0.0625 uS	8.192 mS
8	0.5 uS	65.536 mS
64	4 uS	524.288 mS
256	16 uS	2097.152 mS
1024	64uS	8388.608mS

Berechnung:

- Max. Dauer = (Prescaler)*(1/Frequenz)*(2^17)
- Time per Tick = (Prescaler)*(1/Frequenz)

Verwendung des Timer1

Als erstes muss die folgende Zeile an den Programmanfang eingefügt werden:

```
#include <TimerOne.h>          //Library einbinden
```

Dann muss im `Setup()` der Timer1 konfiguriert werden:

```
Timer1.initialize(100000);            // Timerlänge 100.000µsek (0,1 sek)
Timer1.attachInterrupt( timerIsr );   // ISR aufrufen
```

Hinter `Timer1.initialize` muss die Zeit in Mikrosekunden angegeben werden. Das ist das Intervall, indem die Interrupt-Service-Routine (ISR) aufgerufen wird. Diese wird dann als Funktion aufgerufen und muss den unter `Timer1.attachInterrupt` angegebenen Namen haben.

Beispiel:

```
#include <TimerOne.h>

void setup()
{
 pinMode(13, OUTPUT);                    // LED auf dem Board
 Timer1.initialize(100000);              // Timerlänge 0,1sek
 Timer1.attachInterrupt( timerIsr );     // ISR aufrufen
}

void loop()
{
  // Hauptprogrammschleife
  // TODO: Eigenes Programm hier rein
}

/// --------------------------------
///         Eigene ISR Timer Routine
/// --------------------------------
void timerIsr()
{
    digitalWrite( 13, digitalRead( 13 ) ^ 1 );      // Toggle LED
}
```

Hinweise:

Die Benutzung dieser Library kann die Benutzung der Servo-Library unmöglich machen. (gemäß Quelle [6], von mir nicht getestet). Diese Library hat noch weitere Funktionen, bei Interesse bitte selbst nachschauen.

Auch erzeugt die Benutzung der Library zusätzlichen Code (genannt „Overhead“). Dies gilt übrigens für jegliche Libraries. Die Funktionen einer Library, welche man nicht benutzt, belegen dennoch Speicherplatz im ARDUINO. Wenn der Speicherplatz knapp ist, kann das eventuell ein Problem werden. Also nur einsetzen, wenn es auch gebraucht wird. Profis entfernen in so einem Fall die zusätzlichen, nicht benötigten Funktionen in einer Library, oder benutzen erst gar keine.

Die erzielbare Genauigkeit ist für die meisten Funktionen ausreichend. Eine genaue Uhr kann man aber hiermit nicht programmieren. Dafür sollte dann eine temperaturkompensierte RTC (DS3231), DCF77, GPS oder NTP verwendet werden. Man kann aber die Genauigkeit durch anpassen des Wertes bei `Timer1.initialize` noch erhöhen. Über die Notwendigkeit muss jeder selbst entscheiden, ich empfehle eine temperaturkompensierte RTC mit dem Chip DS3231. Diese arbeitet sehr zuverlässig, hochgenau und ohne Empfangsprobleme wie bei einer DCF77-Uhr

Taster und deren Benutzung

Anschließen von Tastern

Beim Anschließen von Tastern gibt es zwei verschiedene Möglichkeiten. Möglichkeit eins wäre vom ARDUINO-Pin nach Masse (GND) und Variante zwei vom ARDUINO-Pin nach +UB. Beide Varianten führen zum Ziel. Beim Programmieren muss man lediglich wissen, welche der beiden Varianten man in seiner Schaltung verwendet. Sonst kann die Funktion genau umgekehrt sein, als man erwartet…

Taster nach GND

Ich persönlich (und die meisten Elektroniker) bevorzugen diese Variante. Dies hat unter Umständen Vorteile, man kann den einen Anschluss des Tasters direkt an ein (metallisches) Gehäuse klemmen und auch gegenüber Störungen ist diese Variante besser. Vom ARDUINO-Pin sollte auch noch ein sogenannter Pullup-Widerstand nach +UB geschaltet werden. Der Wert ist unkritisch und kann zwischen 5kOhm und 20kOhm liegen.
Es können auch die internen Pullup-Widerstände des ARDUINO verwendet werden. Jedoch haben nicht alle Pins diese Möglichkeit. Man umgeht durch den geringen Mehraufwand eines externen Widerstandes etwaige Probleme oder unerwartete Schaltfunktionen, damit Ärger und Frust. Die Verwendung des internen Pullup wird hier, hier und hier in der Referenz behandelt.
Drückt man nun diesen Taster, dann geht der Pegel am ARDUINO-Pin von HIGH nach LOW. Das nennt der Elektroniker „low-aktiv“. Im Programm muss also gehandelt werden, wenn der Pin auf LOW geht.
Das ist absolut wichtig und muss im Programm bedacht werden.

Taster nach +UB

Bei dieser Variante ist der Taster vom ARDUINO-Pin nach +UB geschaltet. Der Widerstand ist demzufolge vom ARDUINO-Pin nach Masse und nennt sich hier logischerweise Pulldown-Widerstand. Eine Benutzung des internen Pullup-Widerstandes ist bei dieser Variante nicht möglich.
Der Vorteil ist für den Elektronik-Laien der, das die Schaltlogik positiv ist. Das heißt, beim Drücken des Tasters geht der Pegel von LOW nach HIGH am ARDUINO-Pin, man nennt das „high-aktiv“. Vielen Anfängern erscheint das einfacher. Hier muss im Programm gehandelt werden, wenn der ARDUINO-Pin auf HIGH geht.
Das ist absolut wichtig und muss im Programm bedacht werden.
Ich empfehle die Benutzung der ersten Variante.

Benutzung von Tastern mit Interrupts

In einigen Fällen kann es erforderlich sein, SOFORT auf das Drücken eines Tasters zu reagieren. Bei sehr einfachem Programmablauf reicht dafür das Abfragen des Tasters im loop(). Oft sind aber Programme länger, haben zeitintensive Vorgänge (z.B. das Auslesen eines DS18B20-Sensors), oder springen in Unterprogramme.
Dann kann das Abfragen des Tasters schwierig werden. Das äußert sich darin, das auf einen Tasterdruck keine, oder nur eine verspätete Reaktion erfolgt.
Um diese Probleme zu umgehen, kann man sich der Interrupt Funktion bedienen. Klingt erst mal kompliziert, ist es aber gar nicht.
Der ARDUINO UNO und auch der NANO (auch andere haben sowas, gegebenenfalls nachlesen) haben digitale Pins, welche fähig sind, als externe Interruptquelle zu dienen. Das sind ganz bestimmte Pins, hier

die digitalen Pin 2 und Pin 3. Andere Pins funktionieren dafür nicht! Es ist also ratsam, beim Entwurf eines Projektes mit ARDUINO, wenn Taster verwendet werden sollen, von vorn herein diese schon an die Pins 2 und 3 anzuschließen. Wenn es mehr als zwei Taster sind, dann auf jeden Fall die beiden Pins mit nutzen. Wenn man ein Interrupt braucht, muss man nicht erst alles umändern, sondern braucht einfach nur das Programm danach anpassen.

Aber wie gehen wir nun vor? Die Taster werden gemäß der vorhergehenden beiden Kapiteln „Taster nach GND“ ***oder*** „Taster nach +UB“ angeschlossen. Entsprechend der beiden, komplett verschiedenen, Varianten muss nun die entsprechende Variante für das Interrupt gewählt werden. Diese Anweisung kommt in das setup():

```
attachInterrupt(0, interruptRoutine, HIGH);     // INT0 für Taster nach +UB an Pin 2
```
oder
```
attachInterrupt(0, interruptRoutine, LOW);      // INT0 für Taster nach GND an Pin 2
```

Für den anderen Interruptpin Pin 3 sind die Anweisungen so:

```
attachInterrupt(1, interruptRoutine, HIGH);     // INT1 für Taster nach +UB an Pin 2
```
oder
```
attachInterrupt(1, interruptRoutine, LOW);      // INT1 für Taster nach GND an Pin 2
```

Entprellen von Tastern

Allgemeines

Quelle: [8]
Tasten sind wichtige Eingabeorgane an Mikrocontrollern. So auch an ARDUINO’s. Allerdings haben mechanische Tasten ein gravierendes Problem, bekannt als „Prellen“. Dieses Prellen äußert sich darin, das beim einmaligen Drücken eines Tasters dieser in Wahrheit mehrfach, unvorhersehbar oft, geöffnet und geschlossen wird. Dieses passiert sehr schnell und ist nach wenigen Millisekunden vorbei. Der Bediener bemerkt davon nichts. Aber der Mikrocontroller ist viel schneller und registriert jede „Pseudo-Betätigung“ des Tasters. So kommt es zu unvorhersehbaren Funktionen, die nicht gewollt sind.
Daher muss der Taster entprellt werden. Dies kann mittels zusätzlichen Bauteilen oder auch per Software geschehen.

Um das Entprellen per Software in diesem Fall durchzuführen, ist die Library „Bounce2“ erforderlich, welche auch bei [8] heruntergeladen werden kann. Sie ist in den Library-Ordner zu installieren, oder per ARDUINO-IDE einzubinden.

Die Library hat mehrere Funktionen. Ich beziehe mich hier nur auf die einfache Entprellung eines Tasters. Gegebenenfalls die Beispiele anschauen, um die anderen Funktionen kennen zu lernen. Aber die Benutzung eines Tasters stellt eigentlich die Standard-Anwendung dar.

Verwendung der Debounce-Library

Als erstes ist es nötig, die Bounce-Library zu initialisieren. Dies geschieht VOR der setup()-Funktion:

```
Bounce debouncer = Bounce();   // Lib Init
```

Die weiteren wichtigen Dinge geschehen in der setup()-Funktion, entnommen aus Beispiel:

```
void setup() {
    pinMode(BUTTON_PIN,INPUT);         // Taster einbinden
```

```
    digitalWrite(BUTTON_PIN,HIGH);     // internen PullUp-Widerstand aktivieren

    debouncer.attach(BUTTON_PIN);      // Debouncer dem Taster zuordnen
    debouncer.interval(5);             // Entprellzeit in msec (5 Milli-Sek.)

    pinMode(LED_PIN,OUTPUT);           // LED einbinden
}
```

Um den Taster verwenden zu können, muss er natürlich regelmäßig (oft genug) abgefragt werden. Ansonsten entstehen unschöne Verzögerungen und Verwirrung, wenn der Taster nicht reagiert beim Drücken. Also die folgenden Befehle am besten im `loop()` unterbringen, damit er immer wieder durchlaufen wird.

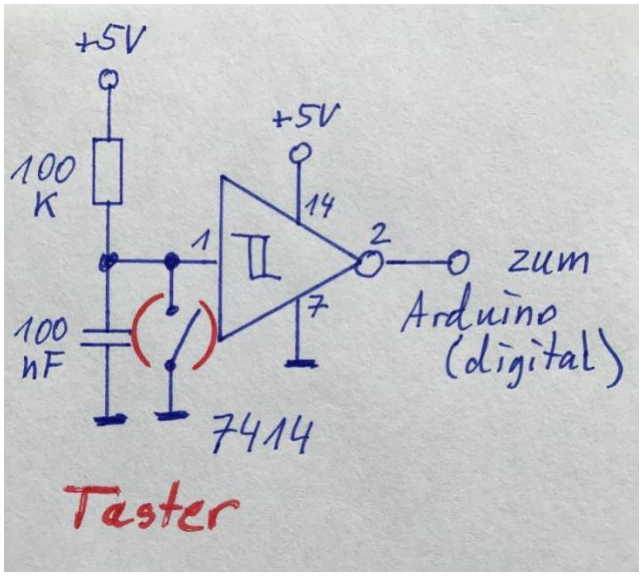
```
  debouncer.update();                  // Debouncer updaten

  int value = debouncer.read();        // Wert abfragen
```

Nun ist der aktuelle Wert des Tasters (logisch HIGH oder LOW) bekannt. Jetzt kann mit dem Programm darauf reagiert werden, was gemacht werden soll.

```
if ( value == HIGH) {
   digitalWrite(LED_PIN, HIGH );
 }
else {
    digitalWrite(LED_PIN, LOW );
 }
```

Entprellen mit zusätzlichen Bauteilen

Ich komme ursprünglich aus der analogen Elektronik. Warum nicht auch einen Taster mittels herkömmlicher Elektronik entprellen?
So kam es zu dieser kleinen Zusatzschaltung, die mit drei weiteren Bauteilen auskommt, plus des natürlich notwendigen Tasters. Es wird ein sogenannter Schmitt-Trigger eingesetzt, einfach ausgedrückt ein Schwellwertschalter. Ich möchte komplizierte Erklärungen weglassen, wen es interessiert kann im Netz alles finden. Nur soviel: Durch Drücken des Tasters wird ein Zeitglied aus Kondensator und Widerstand entladen oder geladen. Bei einem bestimmten Ladezustand des Kondensators schaltet der Schwellwertschalter schlagartig von LOW auf HIGH oder umgekehrt. Diese zeitabhängige Funktion nutzen wir zum Entprellen.



Der verwendete Schaltkreis ist ein 7414, welcher für ein paar Cent zu erwerben ist. Er enthält sechs Schmitt-Trigger, kann also auch maximal sechs Taster bedienen! Er ist so billig, dass sein Einsatz auch bei nur einem Taster gut ist. Die anderen fünf Schmitt-Trigger nutzt man eben nicht.
Ein angenehmer Nebeneffekt ist seine invertierende Funktion, d.h. man hat am Ausgang wieder eine „positive Logik“. Das Drücken des Tasters erzeugt also einen HIGH-Pegel und ist irgendwie schöner...

Ich benutze immer diese Variante, meine Projekte haben genug Platz für drei zusätzliche Bauteile (oder eben 13, wenn man alle sechs Schmitt-Trigger benutzt). Die Schaltung ist absolut betriebssicher, und man braucht keinen Speicherplatz für die Debounce-Library.
Eine einfache Abfrage des entsprechenden Tasters mittels analogRead oder per Interrupt reicht völlig aus, auch die Benutzung mit Mehrfachfunktionen (nächstes Kapitel) geht einwandfrei.

Mehrfachfunktionen mit Tastern (Doppelklick, langer Klick etc.)

Ein Taster ist sicher das am meisten benutzte Eingabe-Organ am ARDUINO. Es gibt noch andere, gute Eingabe-Organe, z.B. den Dreh-Encoder. Aber Taster sind einfach, billig und vielseitig anwendbar. Noch besser werden sie, wenn man Mehrfachfunktionen benutzen kann. Hier kommen im Speziellen der einfache Klick, der Doppelklick und der lange Klick zur Anwendung. Beim langen Klick gibt es die Möglichkeit, wiederholte Aktionen auszuführen, so lang der Taster gedrückt ist, oder nur eine am Ende des Klickes.
Am einfachsten geht das wieder mit einer Library…
Ein Entprellen ist hier nicht extra notwendig.

Benutzung der Library OneButton

Quelle: [13]
Die oben beschriebenen Tasterfunktionen lassen sich sehr einfach mit der Library „OneButton“ realisieren.
Ich habe diese Library schon mehrfach in eigenen Projekten benutzt, es macht richtig Spaß mit so wenig Programmieraufwand tolle Funktionen zu realisieren.
Als erstes muss die Library natürlich in die IDE eingebunden werden, dies geschieht am besten mit dem Bibliotheksverwalter der IDE. Danach kann die Library mit den folgenden Befehlen in den Sketch eingefügt werden und alle Funktionen stehen dann zur Verfügung.

```
#include "OneButton.h"         // Lib einbinden
OneButton button1(A1, true);   // Taster an Pin A1 anschließen
```

In diesem Fall wird der Taster vom Pin A1 nach GND (Masse) angeschlossen. Zusätzlich sollte noch ein Widerstand etwa 5k bis 10k nach +UB geschalten werden. Das wird hier genau erklärt.

Im setup(); ist noch zu deklarieren, welche Tasterfunktionen wir nutzen wollen und welches Unterprogramm dann angesprungen werden soll. Dies geschieht so:

```
button1.attachClick(click);
button1.attachDoubleClick(doubleclick);
button1.attachDuringLongPress(longPress);
button1.attachLongPressStop(longPressStop);
```

Damit der (oder auch mehrere) Taster regelmäßig abgefragt wird, muss im loop(); der Befehl

```
button1.tick();                    // Abfrage Taster
```

eingefügt werden. Erkennt diese Funktion ein Drücken des Tasters, dann wird durch die Library analysiert, welche Funktion der Taster ausführt, also einfacher Klick, Doppelklick etc. Dies geschieht ohne unser Zutun automatisch.
Die von uns gewünschten Programmfunktionen bei dem entsprechenden Klick müssen dann in die von der Library vorgegebenen Methoden eingefügt werden:

Einfacher kurzer Klick

In die folgende Methode wird unser gewünschter Code eingefügt, was bei einem kurzen Klicken des Tasters ausgeführt werden soll:

```
void click() {
  Serial.println("Button 1 einfacher Klick.");
  digitalWrite(led, HIGH);     // LED an
```

```
  // weiterer Code
} // end click
```

Doppelklick

Ein Doppelklick sind zwei kurz hintereinander durchgeführte Klicks, ähnlich wie beim Computer ein Doppelklick mit der Maus. Der auszuführende Code kommt in diese Methode:

```
void doubleclick() {
  Serial.println("Button 1 Doppelklick.");
  digitalWrite(led_gruen, HIGH);      // grüne LED an
  // weiterer Code
} // end doubleclick
```

Langer Klick wiederholte Funktion

Wird der Taster gedrückt und gehalten, so wird diese Funktion immer wieder aufgerufen, solange man den Taster gedrückt hält. Das kann sehr vorteilhaft genutzt werden, um einen Zähler hoch- oder runter zu zählen.

```
void longPress() {
  Serial.println("Button 1 langer Klick mehrfach...");
  zaehler ++;          // Zähler um eins erhöhen
  if zaehler == 100 then zaehler = 0;
} // end longPress
```

Langer Klick einfach

Wird der Taster lang gedrückt und gehalten, dann wird NACH dessen Loslassen die folgende Funktion ausgeführt. Es gibt auch eine Funktion, die arbeitet ähnlich, nur wird hier die Funktion schon ausgeführt, wenn die Zeit für einen langen Klick abgelaufen ist. Bei ersterer wird lediglich noch gewartet, bis der Taster wieder losgelassen wird. Welche der Funktionen man nutzt, ist den eigenen Vorlieben überlassen, ich benutzte immer diese hier:

```
void longPressStop() {
  Serial.println("Button 1 langer Klick einfach");
  digitalWrite(led2; HIGH);    // LED2 an
} // end longPressStop
```

RTC-Uhr und deren Benutzung

Allgemeines

Quelle: [14]
Vielfach besteht im fertigen Gerät oder Projekt das Erfordernis, eine Uhrzeit zu haben, anzuzeigen oder zu bestimmten Uhrzeiten eine Funktion auszuführen. Beispiele wären automatische Alarmanlagen, Türverriegelungen, Hühnerstall-Steuerungen usw. …
Nun gibt es mehrere Möglichkeiten, die Uhrzeit zu gewinnen, jeweils mit Vor- und Nachteilen. Eine von mir bevorzugte Variante ist das Benutzen einer sogenannten temperaturkompensierten „Real Time Clock“, kurz RTC genannt. Sie besteht aus einer kleinen Platine, auf dieser befinden sich der RTC-Schaltkreis, ein paar Bauteile sowie eine Batterie. Diese Batterie versorgt die Platine über viele Jahre mit Strom, die typische Lebensdauer einer solchen Batterie ist zwischen 5 Jahren und mehr als 10 Jahren. Dies dürfte für die meisten Fälle ausreichen. Und wenn nicht, kann man die Batterie immer noch wechseln. Manche RTC-Platinen besitzen auch einen Akku, sodass mit jedem Benutzen des Gerätes dieser aufgeladen wird. Das verlängert die Lebensdauer noch weiter.

Weitere Möglichkeiten für die Gewinnung der Uhrzeit:

1. Prozessortakt ➔ extrem ungenau, Uhrzeit bei Abschalten der Stromversorgung weg
2. DCF77 ➔ oft Empfangsprobleme, Code umfangreich, hochgenau
3. NTP ➔ Netzwerkanschluss erforderlich, hochgenau
4. GPS ➔ Empfangsprobleme in Gebäuden, Stromverbrauch hoch, hochgenau
5. Billige RTC ➔ keine Temperaturkompensation, daher extrem ungenau (DS1307)
6. Gute RTC ➔ Temperaturkompensation, sehr genau, wenige Sekunden / Jahr

Ich habe schon alle Varianten ausprobiert, geblieben bin ich bei der temperaturkompensierten RTC mit dem Schaltkreis DS3231. Sie ist sehr genau, spottbillig aus Fernost zu beziehen und einfach zu programmieren. Um die Benutzung dieser RTC geht es hier.

Die RTC DS3231

Die Genauigkeit ist sicher als erstes von Interesse. Wir neigen ja immer dazu, alles so genau wie möglich zu machen, obwohl das meistens mit Problemen behaftet ist. Laut Datenblatt hat dieser Chip eine Gangabweichung von +/- 2 Minuten pro Jahr im Temperaturbereich von -40°C bis +85°C. Vier Minuten Abweichung im Jahr klingt erst mal „unakzeptabel viel“… Damit kann man ja GAR NICHT leben!

ABER: Dieser Wert bezieht sich auf den gesamten Temperaturbereich und ist der schlechteste garantierte Wert! Die realen Werte sind viel besser. Und in welchem Gerät herrschen schon solche Temperaturunterschiede. Meistens bewegen sich die Temperaturen so zwischen 20°C und 35°C, ein Bruchteil des Datenblattes. Dadurch wird die Genauigkeit wesentlich besser, deutlich unter einer Minute pro Jahr.
Ich habe es mal bei einem Gerät über 5 Monate beobachtet und konnte KEINE Abweichung feststellen, jedenfalls keine messbare. Und weil der normale ARDUINO-Bastler vermutlich kein Atom-Physiker ist, wird er wohl mit der Abweichung von einigen Sekunden im Jahr leben können, oder? Wenn nicht, dann siehe „weitere Möglichkeiten“.

Das RTC-Modul DS3231 besitzt wie schon erwähnt eine Batterie zum Betrieb der Schaltung auch dann, wenn das Gerät abgeschaltet ist. Es werden das Datum, Uhrzeit in 12h/24H-Format, der Wochentag, Schaltjahr-Automatik und auch zwei Alarmzeiten bereitgestellt. Das erfüllt alle Wünsche.
Die Sommerzeit und Winterzeit wird im Programm durch ein paar Codezeilen korrekt dargestellt.

Wie schon fast vermutet, arbeite ich wieder mit einer Library. Der Mensch ist bequem, und bequemer geht es kaum. Ich verwende die Library aus Quelle [14]. Mit dieser Library können sowohl die DS1307, als auch DS3231 / DS3232 benutzt werden. Des Weiteren wird die **Time-Library** benötigt und die **Wire-Library** zum abfragen des I2C-Busses, welche die RTC benutzt. Diese können mit dem Bibliotheksverwalter der IDE eingepflegt werden, wenn dies nicht schon geschehen ist. Ursprünglich hatte ich eine andere Library im Einsatz, aber damit gab es Probleme beim kompilieren mit neueren IDE-Versionen.

Dieser Artikel nutzt als Grundlage das Beispiel, welches in der ARDUINO-IDE unter Datei ➔ Beispiele ➔ Time ➔ TimeRTC zu finden ist.

Beschaltung

Das wichtigste hierbei ist die korrekte Beschaltung der kleinen RTC-Platine, welche durchaus einen Fallstrick beinhalten kann. Also aufpassen. Im schlimmsten Fall geht es nicht, zerstören kann man die Platine nur bei groben Fehlern, etwa Falschpolung der Betriebsspannung.

Auf der RTC-Platine gibt es meistens einen vierpoligen Steckverbinder. Dieser hat die folgenden Anschlüsse:

+UB DC	Versorgungsspannung, wird mit wenigen mA belastet
GND	Masse
SDA	Datenleitung
SCL	Clock-Leitung

SDA und SCL sind die wichtigen Leitungen, mit denen die Kommunikation mit dem Chip erfolgt. Die Schnittstelle heißt „I2C“. Bei Interesse kann man weitere Infos im Netz finden. Die richtige Kommunikation wird durch die Library „wire“ sichergestellt. Diese beiden Leitungen brauchen einen sogenannten Pullup-Widerstand jeweils nach +UB. Auf den meisten RTC-Platinen ist dieser Widerstand schon drauf, also mal schauen oder lesen ob das der Fall ist. Kann man diese Info nicht herausfinden, einfach das Modul anschließen und schauen, ob es geht. Wenn nicht, dann je einen Widerstand 4,7kOhm von SDA nach +UB und einen von SCL nach +UB schalten.

Die beiden Leitungen SDA und SCL müssen auch an einen festgelegten Pin des ARDUINO, sonst geht nichts!
SDA an A4 und SCL an A5. Natürlich GND mit GND des ARDUINO verbinden, genauso wie auch die +UB.
Dann steht der Uhrzeit im ARDUINO nichts mehr im Wege…

Benutzung der Library

Als erstes müssen diese drei Zeilen in den Sketch oberhalb setup(); eingefügt werden:

```
#include <TimeLib.h>
#include <Wire.h>
#include <DS1307RTC.h>
```

Damit werden die benötigten Libraries eingebunden. Im setup(); ist dann die folgende Zeile notwendig. Damit wird durch die Library alle 5 Minuten die RTC ausgelesen und die time-Variable aktualisiert. So ist eine sehr genaue Zeit immer gewährleistet.

```
setSyncProvider(RTC.get);      // time alle 5 Minuten mit RTC syncen
```

Nun steht jederzeit die aktuelle Uhrzeit und das Datum in den Variablen hour(), minute(), second(), day(), month(), year(), weekday() zur Verfügung und kann im eigenen Programm verwendet werden. Je nach Anzeigevariante (seriell, LCD, 7-Segment, Grafikdisplay) mit den entsprechenden Codezeilen. Auch eine Weiterverarbeitung zwecks Funktionen zu einer bestimmten Zeit auszuführen ist einfach möglich. Hier nur mal exemplarisch ein paar Beispiele:

```
lcd.print(hour());      // Anzeige auf LCD
lcd.print(":");
lcd.print(minute());
lcd.print(":");
lcd.print(second());    // Achtung! Noch Formatierung (Vornull) erforderlich

if (stunde == 10 && minute() == 15)    // tue etwas um 10:15 Uhr
```

Weitere Infos zu den verfügbaren Variablen der Time-Funktion des ARDUINO sind unter http://playground.ARDUINO.cc/Code/time zu finden.

Sommer- / Winterzeit Berechnung

Die RTC speichert nur eine Uhrzeit, sinnvollerweise benutzt man hierzu die Winterzeit.
Doch was ist bei der Sommerzeit? Kein Problem, dafür gibt es eine Formel, mit der man berechnen kann, wann Sommerzeit aktiv ist. Im Internet findet man unzählige Beispiele dazu, auch in ARDUINO oder C/C++.
(Genaue Quelle unbekannt, weil an vielen Stellen im Netz zu finden)

```
boolean sommerzeit() // gibt TRUE (Sommerzeit) oder FALSE (Winterzeit) zurück
{
if (month()<3 || month()>10) return false; // keine Sommerzeit in Jan, Feb, Nov, Dez
if (month()>3 && month()<10) return true; // Sommerz. in Apr, Mai, Jun, Jul, Aug, Sep
if (month()==3 && day()<25) return false; // keine Sommerzeit bis 25.Maerz
if (month()==10 && day()<25) return true; // Sommerzeit bis 25.Okt.
if (month()==3 && (hour() + 24 * day())>=(1 + 24*(31 - (5 * year() /4 + 4) % 7))
    || month()==10 && (hour() + 24 * day())<(1 + 24*(31 - (5 * year() /4 + 1) % 7)))
    return true;
    else
    return false;
}
```

Diese Funktion einfach im Programm außerhalb loop(); eingefügt, kann benutzt werden, um abzufragen, ob Sommerzeit oder Winterzeit ist. Mit dem folgenden Code im Programm, wenn die Zeit ermittelt oder angezeigt werden soll, kann einfach darauf reagiert werden, ob die Stunde korrigiert werden muss oder nicht.

```
if (sommerzeit() == true)     {
        stunde = hour() + 1;
        if(stunde == 24) stunde = 0;     // damit korrekt 0 Uhr angezeigt wird
    }
else{ Stunde = hour(); }
```

Temperatursensoren und deren Benutzung

Allgemeines

Die Messung und Anzeige von Temperaturen gehört sicher zu den interessantesten Anwendungen, die man in seine Programme einfügen möchte. Ebenso kann man dann, wenn man erst mal die Temperatur kennt, im Programm in geeigneter Weise darauf reagieren. Zum Beispiel einen Lüfter einschalten, eine Warn-LED oder Pieper ein-, oder auch das zu warme Teil abschalten. Ich selbst habe schon mehrere Uhren mit großen LED-7-Segmentanzeigen gebaut, die auch die Temperatur alle 40 Sekunden für 10 Sekunden anzeigen.

Nun kommen wir zum „WIE"… Es gibt etliche, teilweise grundverschiedene Sensoren, die man verwenden kann. Sie sind von ganz billig bis teuer zu haben, von weniger genau bis hochpräzise.
Ich beschränke mich hier auf drei Typen:
Einmal einen Sensor (DS18B20 / DS18S20 / DS1820), der mit einem Eindraht-Bus arbeitet, wodurch mehrere Sensoren an einem einzigen ARDUINO-Pin angeschlossen werden können. DS1820 ist ausgelaufen, Erwähnung nur der Vollständigkeit halber.
Zum anderen auf einen Typ (MCP9700), welcher eine der Temperatur proportionale, lineare Gleichspannung ausgibt.
Der dritte ist der LM75, er arbeitet mit dem I2C-Bus.
Alle sind sehr preiswert, die letzten beiden im Centbereich, und einfach anzuwenden. Wir werden die beiden ersten hier in diesem Kapitel behandeln.
Den letzten, den Sensor LM75, welcher mit dem I2C-Bus arbeitet, werde ich im Kapitel „Der I2C-Bus und seine Benutzung" behandeln.

DS18B20 / DS18S20 (1wire-Bus-Sensor)

Beispiel-Sketch verfügbar: temp_DS18B20.ino

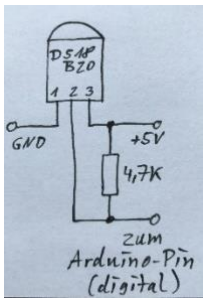
Bei diesem Sensor handelt es sich um einen komfortabel anzuwendenden Temperatursensor der Fa. Dallas, dessen Datenblatt unter Quelle [16] anzusehen ist, oder einfach selbst im Netz suchen.

Die großen Vorteile dieser Sensoren sind:

- Großer Temperaturbereich -55°C bis +125°C
- Kein Abgleich erforderlich
- Einfach zu verdrahten dank Eindrahtbus
- Viele Sensoren gleichzeitig an einem einzigen ARDUINO-Pin benutzbar
- digitale Ausgabe, Auflösung 9bit (DS18S20) oder 9-12bit programmierbar (DS18B20)

Der Sensor arbeitet mit dem sogenannten 1wire-Bus. Zur Benutzung dessen gibt es eine komfortable Library, oder besser zwei. Wer hätte das gedacht…? Die beiden Libraries, auf die ich mich hier beziehe, sind **OneWire** und **DallasTemperature**. Beide sind komfortabel über den Bibliotheksverwalter der IDE einzubinden.

Anschließen des Sensors



Ist das geschehen, muss nur noch der Sensor (oder mehrere) korrekt an den ARDUINO angeschlossen werden. Hier ist etwas zu beachten, sonst geht es nicht: Es muß ein sogenannter „PullUp-Widerstand“ vom Data-Pin des Sensors nach +UB geschaltet werden. Das ist schon alles. Das sieht dann so aus, siehe meine künstlerische Skizze.
Möchte man mehrere dieser Sensoren gleichzeitig nutzen, so sind diese einfach parallel zu schalten. Einen weiteren Widerstand 4,7k braucht man dann nicht! Einfach die weiteren Sensoren parallel zu dem ersten anschließen.
Es gibt noch eine andere Art, diesen Sensor anzuschließen, sie nennt sich „parasite power“. Hier wird die Betriebsspannung aus der Datenleitung entnommen. Der PIN 3 des Sensors bleibt unbeschaltet. Daher stammt auch die Bezeichnung 1Wire, da neben Ground, nur eine Leitung für Spannungsversorgung und Daten benutzt wird. Alles weitere dazu kann man im Datenblatt nachlesen. Die Programmierung ist dabei aufwendiger und problematischer, weil die Datenleitung bis zu 750ms mit Spannung versorgt werden muss.
Bessere Betriebssicherheit gibt es, wenn die Betriebsspannung mitgeführt wird.
HINWEIS: Es gibt neben den Sensoren DS18x20 noch welche mit der Bezeichnung DS18x20-PAR. Diese werden ausschließlich mit parasitePower betrieben, der Anschluß VDD ist nicht vorhanden! Zudem Temperaturfestigkeit nur bis +100°C (Quelle: https://datasheets.maximintegrated.com/en/ds/DS18S20-PAR.pdf und https://datasheets.maximintegrated.com/en/ds/DS18B20-PAR.pdf)

Hier wird zum Anschluss nur die erstere Variante betrachtet - die Programmierung ist dabei auch einfacher.

Programmierung

Hierzu sei der Beispiel-Sketch „Simple“ aus der Library DallasTemperature zum ersten Testen empfohlen. Auf diesen beziehe ich mich hier größtenteils.
Im eigenen Sketch müssen natürlich die beiden Libraries eingebunden werden.

```
#include <OneWire.h>
#include <DallasTemperature.h>
```

Als nächstes wird dem Compiler mitgeteilt, an welchem digitalen Pin der Sensor angeschlossen ist:

```
#define ONE_WIRE_BUS 2         // Sensor DS18B20 an digitalem Pin 2
```

Nun werden die Libraries initialisiert mit den folgenden beiden Anweisungen:

```
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
```

Im setup() muss die Library gestartet werden.

```
sensors.begin();
```

So, das war eigentlich alles, damit das Programm vorbereitet ist zum erfolgreichen Abfragen der Temperaturen. Wann dieses geschieht, liegt in den Vorlieben des Programmierers. Mir persönlich reicht es, wenn maximal 1mal pro Minute die Temperatur abgefragt wird.
Weiterhin ist zu bedenken, das die Sensoren relativ lange brauchen, um die Temperatur abzufragen und zu verarbeiten: bis zu 750ms. Das ist in ARDUINO-Programmen eine halbe Ewigkeit! Wenn man zeitkritische Dinge im Programm laufen hat, z.B. Tasterabfragen, sind diese dann per Interrupt zu lösen (siehe hier). Sonst erfolgt unter Umständen keine Reaktion auf den Tastendruck. Auch bei anderen Programmfunktionen ist diese lange Zeit eventuell ein Problem und man muss sich genau überlegen, wie man das Problem löst.
An geeigneter Stelle (oder auch in einer separaten Funktion) wird nun die Abfrage mit dem folgenden Befehl vorbereitet:

```
sensors.requestTemperatures();         // Temperatur holen
```

Der folgende Befehl übergibt den Wert des Sensors an eine float-Variable:

```
float temp = sensors.getTempCByIndex(0);
```

Die Zahl (0) bedeutet, dass der erste Sensor ausgelesen wird, wenn es mehrere sind. Will man den zweiten auslesen, kann man das mit (1) machen usw. Auch eine for-Schleife ist denkbar, bei vielen Sensoren.
Meine maximale Anzahl von Sensoren an einem Projekt waren bisher 2 Stück, da ist es am einfachsten, man macht es so:

```
float temp0 = sensors.getTempCByIndex(0);
float temp1 = sensors.getTempCByIndex(1);
```

Dann stehen die beiden Werte als Fliesskommazahl in den Variablen temp0 und temp1 zur weiteren Verarbeitung zur Verfügung. Man kann sie anzeigen, auf bestimmte Wert reagieren, zu hoch, zu tief…

MCP9700 (analoger Sensor)

Beispiel-Sketch verfügbar: temp_MCP9700.ino

Der MCP9700 ist ein analoger Sensor der Firma Microchip. Dieser liefert am Ausgang eine der Temperatur direkt proportionale Gleichspannung mit einer Auflösung von 10mV/°C. Bei 0°C beträgt die Ausgangsspannung 500mV, bei 20°C demzufolge 700mV, 30°C entsprechen 800mV, -20°C entsprechen 300mV usw. Diese Spannung muss also mit einem analogen Eingang des ARDUINO verarbeitet werden.

Die Eigenschaften des Sensors in Kurzform:

- Hoher Temperaturbereich -40°C bis +150°C
- Geringer Stromverbrauch
- Sehr günstig
- Simple Programmierung ohne Library

Nachteile:

- Jeder Sensor belegt einen analogen Eingang
- Genauigkeit nicht so gut mit +-4°C
- Anfällig gegen Schwankungen der Referenzspannung des ARDUINO (meist die +UB Betriebsspannung)

Anschließen des Sensors

Das Anschließen dieses Sensors ist besonders einfach. Einfach die Betriebsspannung 5V DC und GND anschließen. Der Ausgang wird nun an einen beliebigen Analogeingang gelegt. Das war es auch schon.

Programmierung

Zum Messen der Temperatur muss der Wert des AD-Wandlers (z.B. A0) in eine für den Menschen nützliche Form gebracht werden. Da der AD-Wandler lediglich einen Wert von 0…1024 entsprechend der anliegenden Spannung von 0…5V zurück gibt, ist etwas Mathematik nötig, um daraus eine Temperatur zu bekommen.
Diese Berechnung ist für den ARDUINO aber keine Schwierigkeit.

Wir definieren uns erst mal eine float-Variable, um den errechneten Wert der Temperatur dort zu speichern.

```
float temp = 0.0;              // errechnete Temperatur
```

Im Programm bringen wir nun an geeigneter Stelle unseren Code unter, um den 10bit-AD-Wert des AD-Wandlers in eine Temperatur umzuwandeln. Hier sollte auch wieder bedacht werden, wie oft man die Temperatur neu berechnet, es reicht sicher einmal pro Minute aus.

```
temp = (analogRead(A0) * 5.0 / 1024.0) - 0.5;
temp = temp / 0.01;
```

Allerdings geht diese Berechnung deutlich schneller als das Auslesen der 1wire-Sensoren.

Hinweis

Ein ähnlicher Sensor ist der LM35. Er hat auch eine Auflösung von 10mV/°C, liefert aber bei 25°C eine Ausgangsspannung von 250mV. Durch Anpassung der obigen Formel kann er genauso angewendet werden.

LM75 (I2C-Sensor)

Siehe hier im Kapitel für I2C

Luftfeuchte-Sensoren und deren Benutzung

Allgemeines

Die Sensoren der DHT-Reihe sind Kombisensoren für Temperatur und Luftfeuchtigkeit und damit interessant für den Einsatz z. B. zur Überwachung der Raumluft / Vorbeugung von Feuchteschäden (DHT11) bzw. Wetterstationen (DHT22). Da diese Sensoren auch recht einfach anzuwenden sind, werden sie hier mit aufgeführt. Gemäß Hinweisen im Internet sollen diese Sensoren altern. Das konnte ich persönlich bisher nicht überprüfen.

Anschließen des Sensors

Die elektrische Beschaltung dieser Sensoren ist denkbar einfach gemäß der nachfolgenden Hinweise. Es gibt die Sensoren sowohl als einzelnes Bauteil, als auch auf einer kleinen Platine. Sowohl mit drei als auch mit vier Anschlüssen.
Es gibt jeweils einen Anschluss +5V, GND und Signal. +5V muss an die +5V des ARDUINO angeklemmt werden, sinngemäß der GND auf GND des ARDUINO.
Ein etwaiger vierter Anschluss ist nicht beschaltet. Wenn der Anschluss „Signal“ nicht bereits (etwa auf der Platine, auf der sich der Sensor befindet) mit einem Widerstand nach +5V beschaltet ist, so ist von Signal nach +5V noch ein Widerstand von 4,7k zu beschalten.
Am besten hier vorher genau lesen, was bei dem entsprechenden Sensor dabei steht.

Programmierung

Zur Benutzung dieser Sensoren machen wir uns wieder eine Library zu Nutze. Es gibt hier viele verschiedene Libraries, die Auswahl fällt schwer. Ich habe die unter Quelle [28] angegebene Library installiert, da sie für alle DHT-Sensoren gleichermaßen Anwendung findet. Dort ist auch ein Anwendungsbeispiel gegeben.

Ultraschall-Sensoren und deren Benutzung

Allgemeines

Danke an Renè für diesen Beitrag. Der beigefügte Sketch konnte durch mich nicht getestet werden.
Ein sehr verbreiteter Kandidat der Ultraschall-Sensoren ist der HC-SR04.
Er besteht aus getrennter Sende- und Empfangseinheit auf einer Platine.
Mit diesem Sensor lassen sich wegen des teils sehr engen Erfassungsbereich von 15 Grad und einer Reichweite zwischen 2cm und 4M (typ) bzw. 6M (max) recht effektiv Abstandsmessungen mit hoher Genauigkeit durchführen.
Aufgrund des Aufbaus ist er nicht für den Außenbereich geeignet, hilft aber in der Garage beim Einparken, wenn das eigene Fahrzeug keine Sensoren hat oder im Modellbau als „Anstoßvermeidungssensor“. Will man dagegen Outdoor z.B. Türschließanlagen oder Fahrzeuge mit wasserdichten Sensoren ausstatten, kommt man um den SR04-T nicht vorbei. Die Sensoren sehen aus wie Einparksensoren. Kommen mit ca. 2,5M Kabel und einer abgesetzten Platine.
Bei diesen Sensoren ist der Erfassungswinkel größer; bis zu 75 Grad.

Beschaltung

Die Platinen kommen mit 4 Pins; VDD (+5V) und GND zur Versorgung sowie TX → zum auslösen des Sendeimpuls (Trigger) und RX → zur Übertragung des Empfangssignal (Echo).

Funktion

Die Funktion ist recht einfach: Der Triggerport liegt LOW. Mit einem HIGH-Triggerimpuls (nicht Flanke!) von in der Regel (dazu unten mehr) 10 Mikrosekunden wird die Sendung von acht 40Khz Impulsen gestartet. Nach Ende der letzten Amplitude wird auf dem ECHO-PIN solange ein HIGH-Pegel ausgegeben, bis der Empfänger das Signal wieder aufnimmt.
Die Länge des HIGH-Pegel ist also proportional des zurückgelegten Weges.
Hinweis für die Umrechnung: Der zurückgelegte Weg bemisst sich aus Hinweg nach Ende der letzten und Rückweg der ersten Schallwelle!
Um nun aus der Zeit des HIGH-Pegel auf die Entfernung zu kommen, benötigt es einiger Umrechnungen.
Im Regelfall wendet man eine Ausbreitungsgeschwindigkeit von 340m/s an. (Luft bei ca. 15°C)
Da erst ab der letzten gesendeten Amplitude bis zur ersten empfangenen Amplitude gemessen wird, muss die Sendezeit herausgerechnet werden.
Schließlich kommt man dann auf einen Divisor von 58 mit dem die Zeit geteilt wird um die Entfernung in cm zu erhalten.

Programmierung

Wie dem dortigen Beispiel zu entnehmen ist, geht mit wenig Zeilen Code die Messung sehr gut.
Gerade wenn viele Sensoren ins Spiel kommen, ist die Nutzung der Lib von Vorteil, da Sie eine Menge Arbeit abnimmt.

Das man sich damit auch selbst ein Bein stellen kann, wird im Folgenden gezeigt:
Für Projekte mit einem Sensor wurden SR04-T erworben. Diese tragen die Bezeichnung „V2.0“.
(Vollständig: JSN-SR04T-2.0)

Die vorher mit NewPing funktionierenden Sketche brachten eine Fehlerquote von mehr als 60%, da die Distanz immer mit 0 ausgegeben wurde.

Nach einem Vergleich der Platine mit einer älteren – leider defekten – fiel auf, das sich die Platine maßgeblich verändert hat. Endgültig Aufschluss gab das Datenblatt.
Quelle: https://www.jahankitshop.com/getattach.aspx?id=4635&Type=Product
Daraufhin wurde die Verwendung von NewPing verworfen, weil die Vermutung aufkam, das der Triggerimpuls trotz der Angaben im Datenblatt nicht ausreicht um den Sensor zur Arbeit zu bewegen.

Daraus entstand der folgend angefügte Sketch.
Und schon Trefferquote von 100% und der Nachweis, das 15µs notwendig sind um den Trigger auszulösen.
Gewissermaßen lässt der Sketch sich für jeden Ultraschallsensor anwenden.
Wer also weder viele Sensoren noch hochpräzise aufwendige Messungen braucht, ist mit dem Codeschnipsel vielleicht besser bedient als mit einer ganzen Lib.

```
/*********************************************
   Testsketch - Triggertime für JSN-SR04-T 2.0
   10 Mikrosekunden sind zum triggern zu kurz
   Dieser Sketch erhöht die Triggertime um 1µs
      bis keine Ausfälle (distanz 0) mehr
   Im seriellen Monitor sollte nach 15 Minuten
      keine Erhöhung von msek mehr erfolgen
*                                             *
  Mit 5 Sensoren erfolgreich auf 15µs getestet
*********************************************/
#define debugmsek true  // Debug Triggerlänge
#define debugdist true  // Debug errechnete Entfernung

#define EPIN 11         // Echo
#define TPIN 12         // Trigger
int msek = 0;

void setup() {
  Serial.begin(9600);
  pinMode(EPIN, INPUT);
  pinMode(TPIN, OUTPUT);
}

void loop() {
  digitalWrite(TPIN, LOW);  // Setzte definierten Ausgangszustand
  delayMicroseconds(5);     // Behalte Ausgangszustand für 5µs
  digitalWrite(TPIN, HIGH); // Setze Triggerpin auf HIGH
  delayMicroseconds(msek);  // Halte den HIGH-Zustand für msek µS
  digitalWrite(TPIN, LOW);  // Beende Triggerimpuls
  word distanz = pulseIn(EPIN, HIGH, 26000); // Lese Echopin, bis Level LOW
  if (debugmsek) {
    Serial.print(F("Laufzeit: "));  // gibt Zeit in µS aus
    Serial.println(distanz);
    Serial.print(F("msek: "));      // gibt Länge des Triggerimpuls in µS aus
    Serial.println(msek);
  }

  if (distanz < 1) {             // nicht auf = 0 prüfen! Variable ist Typ long!
    msek++;                      // erhöhe msek wenn distance nicht ermittelbar
```

```
  }

  if (debugdist) {
    // Ab hier Ausgabe der errechneten Entfernung
    distanz = distanz / 0.58;    // geht auch z.B. 0.582
    Serial.print(distanz, DEC);
    Serial.print(F("  10/mm   /  "));
    Serial.print(distanz / 100, DEC);
    Serial.print(F(","));
    int rest = distanz % 100;
    Serial.print(rest);
    Serial.println(F(" cm"));
  }
  delay(250); // Pause um Schallechos auszuschliessen und um den Monitor abzulesen
}
```

Der I2C-Bus und seine Benutzung

Allgemeines

Allgemeines ist z.B. unter Quelle: [17] zu finden.
Der I2C-Bus – wozu brauchen wir ihn…? Gute Frage. Nehmen wir als Beispiel ein LCD-Display, eines der beliebtesten Ausgabegeräte. Schließen wir es dem Standard entsprechend an, brauchen wir außer GND und +UB noch vier Datenleitungen, sowie E, und RW. Das macht insgesamt sechs (!) ARDUINO-Pins, die nur zur Ansteuerung des Displays verwendet sind. Ganz schön viel, speziell, wenn man einen kleinen ARDUINO benutzt, oder noch viele weitere Dinge, wie Taster, Relais oder LED's anschließen will.

Wäre es da nicht schön, wenn man die verbrauchten Pins in so einem Fall einschränken könnte, sagen wir, auf nur zwei Pins anstatt sechs? Genau das macht der I2C-Bus. Eine tolle Sache! Der I2C-Bus hat zwei Leitungen: SDA und SCL.

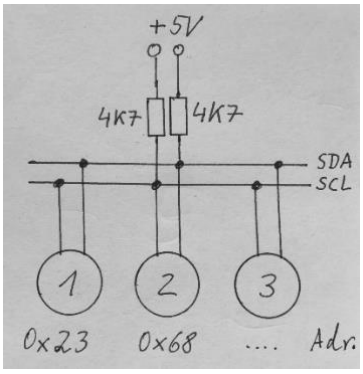
Hinweis

Wenn man mehrere I2C-Geräte benutzt, können dieses alle an diesen dafür verwendeten ARDUINO-Pins angeschlossen werden. Es werden also keine weiteren Pins verbraucht, nur diese beiden.
Unterschieden werden die „Teilnehmer" am Bus durch ihre Adresse. Diese ist ein HEX-Wert im Format 0x68. Im Sketch muss also immer „gesagt" werden, welcher Teilnehmer bei dem entsprechenden Befehl gemeint ist.

Es können nicht beliebige Pins benutzt werden! Bei den ATMega328-basierten ARDUINO's, also z.B. UNO, NANO sind die Pins die folgenden: **SDA an A4 SCL an A5**

Beschaltung

Wie in dem Bild zu sehen, ist von SDA und SCL je ein sogenannter „PullUp-Widerstand" nach + Versorgungsspannung nötig. Dieses aber nur einmal, egal wie viele Teilnehmer am Bus sind. Die I2C-Geräte benötigen selbst natürlich auch noch jeweils GND und +UB als Betriebsspannung. Mehr gibt es nicht zu beachten, dann kann es schon losgehen.



I2C-LCD-Displays

Beispiel-Sketch verfügbar: LCD_I2C.ino

Bei einem LCD-Display kann man durch Verwendung des I2C-Busses ganze fünf von sieben ARDUINO-Pins einsparen. Anzuschließen ist es gemäß obiger Grafik. Die Adresse hängt vom verwendeten I2C-Modul ab, welches verbaut wurde. Im Idealfall steht die Adresse auf dem Modul. Bei einigen kann man auch die Adresse wählen durch Einbringen einer Lötbrücke. Hier hilft gegebenenfalls das Datenblatt oder Informationen des Anbieters.
Ist beides nicht verfügbar, gibt es einen kleinen Sketch, welcher den Bus nach I2C-Teilnehmern absucht und die Adresse auf dem seriellen Port ausgibt. Dieser Sketch kann unter [18] heruntergeladen werden.

Programmierung

Zum Betreiben des Displays verwenden wir wieder eine Library, welche unter [19] heruntergeladen werden kann.
Weiterhin ist noch die prinzipielle Library für die Benutzung des I2C-Busses notwendig. Wir binden beide Libraries in unseren Sketch ein:

```
#include <Wire.h>                 // erforderlich
#include <LiquidCrystal_I2C.h>    // erforderlich
```

Nun kann die Programmierung ganz normal mit den bekannten Befehlen des Kapitels „LCD-Anzeigen und deren Benutzung“ fortgesetzt werden. Es gibt aber noch ein paar extra Befehle, welche man bei Bedarf in den Beispielen sieht und anwenden kann.
Damit es losgehen kann, muss das Display noch initialisiert werden:

```
LiquidCrystal_I2C lcd(0x3F, 16, 2);
```

Hier handelt es sich um ein Display mit der Adresse 0x3F und 16 Zeichen auf zwei Zeilen.
Im Setup muss das Display genauso behandelt werden wie ein normales Display mit dem folgenden Befehl:

```
lcd.begin();        // LCD starten
```

Ist es ein beleuchtetes Display, kann nun die Beleuchtung wie gewünscht an- oder ausgeschaltet werden. Das geschieht mit den folgenden Befehlen:

```
lcd.backlight();              // Backlight an
lcd.noBacklight();            // Backlight aus
```

Im laufenden Programm ist es auch möglich, abzufragen, ob die Beleuchtung gerade an ist oder nicht. Das ist ganz einfach möglich mit der folgenden Codezeile:

```
boolean state =  lcd.getBacklight();
```

Das Ergebnis der Variable „state“ ist entweder „0“ (Beleuchtung aus) oder „1“ (Beleuchtung an).

Hinweis

Bei der Verwendung von aus dem Internet geladenen Libraries ist zu beachten, dass es für ein und denselben Zweck oft mehrere, komplett unterschiedliche Libraries gibt. Findet man nun irgendeinen Beispielsketch, kann es sein, dass dieser nicht funktioniert. Also muss immer genau geschaut werden: Passt mein Beispielsketch zu der geladenen Library.

LM75 Temperatursensor

Beispiel-Sketch verfügbar: lm75_i2c.ino

Der LM75 ist ein Temperatursensor für den I2C-Bus mit einem Temperaturbereich von -55°C bis +125°C und wartet mit einer interessanten zusätzlichen Funktion auf. Zusätzlich zur Möglichkeit, die Temperatur über I2C abzufragen, hat der LM75 noch einen Übertemperatur-Alarm-Ausgang. Hier handelt es sich um einen Open-Drain-Ausgang (bitte mal nachlesen, was das bedeutet, falls unbekannt), welcher aktiviert wird, wenn die Temperatur einen bestimmten Wert überschreitet.
Dieser Ausgang hat zwei verschiedene Betriebsarten, die am meisten genutzte ist als Standard (Default) aktiviert. Der Schaltpunkt beträgt 80°C, der Schwellwert, also der Rückschaltpunkt 75°C. Beide Werte

können über I2C programmiert werden. Die Standardfunktion arbeitet folgendermaßen: Übersteigt die Temperatur den Schaltwert (TOS) von 80°C, wird der Ausgang OS aktiviert (niedrige Impedanz, also nahe null Ohm – ähnlich wie „Schalter geschlossen“). Unterschreitet die Temperatur den Wert der Schaltschwelle (THYST) von 75°C, dann wird der Ausgang OS wieder deaktiviert (hohe Impedanz – ähnlich wie „Schalter geöffnet“).
Die andere Betriebsart erwähne ich hier nicht, wen das näher interessiert, sei auf das Datenblatt verwiesen.

Da der Schaltkreis selbst ein SMD-Bauteil ist, besorgt man sich am besten eines dieser kleinen Leiterplatten, auf denen alles bereits bestückt und beschriftet ist. Sie sind für minimales Geld übers Internet zu beziehen.

Programmierung

Man kann zur Programmierung wieder auf Libraries zugreifen, oder es mit meist kryptischen Befehlen selbst programmieren. Ich benutze die Library der Quelle [20] und mache es mir so leicht wie möglich. Bei der Library ist auch ein Beispiel dabei, welches selbsterklärend ist. Ich beziehe mich hier auf genau jenes Beispiel der Quelle [20].
Zuerst müssen mal wieder zwei Libraries eingebunden werden:

```
#include <Wire.h>              // I2C-Lib
#include <LM75.h>              // LM75-Lib
```

Auf dem Board des LM75 befinden sich drei Lötbrücken oder Jumper zur Festlegung der I2C-Adresse. Belässt man das Board im Lieferzustand und will nur eines einsetzen, kann man die Standardadresse von 0x48 beibehalten. Dann wird der Sensor mit dem folgenden Befehl initialisiert:

```
LM75 sensor;                   // Sensor initialisieren
```

Hat man mit den Jumpern eine andere I2C-Adresse eingestellt, muss diese der Library mitgeteilt werden. Das geschieht in der folgenden Art und Weise:

```
LM75 sensor(LM75_ADDRESS | 0b001);     // Adresse 0x4C zugewiesen
```

In dieser Zeile wurde auf dem Modul der Jumper A0 auf GND, A1 auf GND und A2 auf +UB gelegt. Auf diese Art kann man insgesamt 8 verschiedene Adressen vergeben, somit 8 verschiedene Sensoren anschließen und auch abfragen.
Im setup() ist nun die wire-Library zu starten mit:

```
Wire.begin();          // I2C-Library starten
```

Nun kann an geeigneter Stelle im Programmcode die Temperatur abgefragt werden.
Entweder wie im Beispiel zur direkten Ausgabe auf den seriellen Monitor:

```
Serial.print("Aktuelle Temperatur: ");
Serial.print(sensor.temp());
Serial.println(" C");
```

Oder aber (das ist sicher eher der Fall) an eine Fließkommavariable übergeben werden:

```
float temperatur = sensor.temp();
```

Will man den float-Wert mit einer Kommastelle auf einem LCD ausgeben, kann man das so machen:

```
lcd.print(temperatur,1);       // anzeigen, mit einer Kommastelle
```

Portexpander PCF8574 / 8575

Beispiel-Sketche verfügbar: siehe Unterpunkte

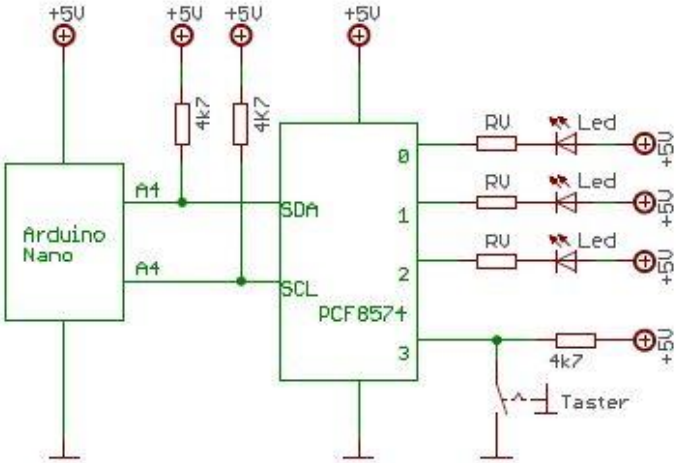
Der ARDUINO UNO beispielsweise hat 14 digitale I/O-Pins. Der ARDUINO Mega 2560 hat gar 54 digitale I/O-Pins. Das ist schon eine ganze Menge. Dummerweise kostet der Mega auch richtig Geld, im Vergleich zum UNO oder NANO.

Was aber, wenn die 14 I/O-Pins nicht ausreichen, weil man ein LCD, viele Taster, etliche Relais und LED's angeschlossen hat? Und es noch etwas mehr sein soll – angenommen wir brauchen 20 I/O-Pins. Einen MEGA zu nehmen ist eine Lösung. Aber die ist eben nicht günstig, oder aus Platzgründen nicht möglich. Hier kommen Portexpander ins Spiel.

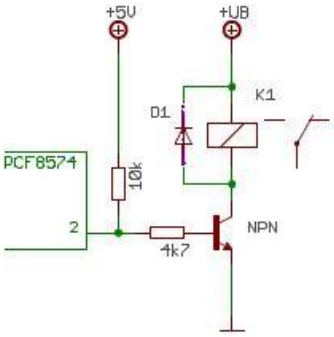
Was macht ein Portexpander? Das ist ein integrierter Schaltkreis, der von mehreren Herstellern gebaut wird. Der PCF8574 hat als „Eingang" eine I2C-Schnittstelle und als „Ausgang" acht digitale I/O-Pins. Der PCF8575 ist sehr ähnlich, nur hat dieser 16 digitale I/O-Pins. Diese Ausgänge können ähnlich derer der ARDUINO's beschaltet werden, also mit Taster und PullUp-Widerstand als Eingang, oder auch mit LED, Kleinrelais etc. als Ausgang.
Wir bekommen durch Nutzung von nur 2 Eingangsleitungen, nämlich SDA und SCL (A4 und A5 am ARDUINO), mit einem Chip insgesamt acht I/O-Pins. Perfekt! Durch Adressierung der Anschlüsse A0 bis A2 ist es möglich insgesamt 8 dieser Expander gleichzeitig zu betreiben. Und wem das nicht genügt, kann mit dem PCF8574A/75A ohne Veränderungen weitere 8 typgleiche Expander betreiben.
Um diese Pins nun nutzen zu können machen wir es uns wieder einfach und laden erst mal aus Quelle [23] oder über die Bibliotheksverwaltung des ARDUINO die entsprechende Library runter und installieren diese. Alle weiteren Erklärungen in diesem Artikel beziehen sich auf diese Library. Wobei ich nur die aus meiner Sicht wichtigsten Funktionen beschreibe. Bei weitergehendem Interesse einfach mal das dazugehörige Beispiel anschauen.

Beschaltung

Wie schon bei den anderen I2C-Bausteinen, ist natürlich SDA und SCL korrekt mit dem ARDUINO zu beschalten, am Beispiel des UNO oder NANO mit den Pins A4 und A5. Dort muss je ein PullUp-Widerstand von ca. 4,7k nach +UB geschaltet werden.
Aufgrund der Innenschaltung des PCF8574/8575 ist die Beschaltung als Ausgang, z.B. mit LED's etwas anders als sonst beim ARDUINO üblich. Hier muss die LED vom Ausgang nach +UB verdrahtet werden. Das bedeutet auch, wenn der Ausgang auf HIGH geschaltet wird, ist die LED aus. Das ist unbedingt zu beachten. Die verfügbaren Beispiel-Sketche sind folgendermaßen verdrahtet:



beispielhafte Beschaltung mit LED und Taster



Beispiel für Beschaltung mit größerer Last, hier einem Relais

Programmierung

Man kann zur Programmierung auf die Library zugreifen, oder es mit meist kryptischen Befehlen selbst programmieren. Ich benutze die Library der Quelle [23] und mache es mir so leicht wie möglich.
Zuerst müssen mal wieder vor dem setup() zwei Libraries eingebunden werden:

```
#include <Wire.h>               // I2C-Lib
#include "PCF8574.h"           // PCF8574-Lib
```

Nun muss eine Instanz dieser Library gestartet werden, das machen wir mit der folgenden Programmzeile, ebenfalls vor dem setup():

```
PCF8574 expander;
```

Im setup() muss nun noch die I2C-Adresse mitgeteilt werden. Ist diese nicht bekannt, kann einfach der Sketch I2C-Scanner.ino (im Download) genutzt werden, um selbige zu bestimmen.

```
expander.begin(0x38);  // PCF mit Adresse starten A0=0, A1=0, A2=0
```

Weiterhin muss im setup(), genau wie bei normalen Ein- und Ausgängen festgelegt werden, ob es ein Eingang oder ein Ausgang ist. Dieses geschieht mit den folgenden Zeilen:

```
expander.pinMode(0, OUTPUT);         // Pin0 am PCF8574 ist ein Ausgang
expander.pinMode(1, INPUT);          // Pin1 am PCF8574 ist ein Eingang
```

Das sind die Vorbereitungen, die zur Benutzung des I2C-Portexpanders notwendig sind. Wenden wir uns nun einigen möglichen (nicht allen) Verwendungsszenarien zu. Ich habe hier die einfachsten Varianten ausgewählt, um eine Einblick zu gewähren. Eventuell mal die Beispiele anschauen, welche bei der Library mitgeliefert werden. Die eben genannten Vorbereitungen sind dabei immer im Sketch einzufügen.

Pins ein-und ausschalten mit digitalWrite

Beispiel-Sketch verfügbar: digitalWrite.ino

Das schöne an dieser Library ist die der normalen ARDUINO-IDE nachempfundenen Anweisungen.
So gibt es zum Setzen eines Ausganges ebenfalls die Anweisung digitalWrite. Sie wird folgendermaßen benutzt:

```
expander.digitalWrite(0, HIGH);      // Pin0 auf HIGH setzen
expander.digitalWrite(0, LOW);       // Pin0 auf LOW setzen
```

Ich glaube, es bedarf keiner weiteren Erklärung. Diese Anweisung setzt Pin0 (der PCF8574 hat Pin0 bis Pin7) auf HIGH oder LOW. Ist gemäß der Skizze eine LED angeschlossen, dann leuchtet sie bei LOW und ist aus bei HIGH (negative Logik, ist zu beachten!)

Zustand von Pins lesen mit digitalRead

Beispiel-Sketch verfügbar: digitalRead.ino

Zum Auslesen eines Pinzustandes gibt es die Anweisung digitalRead. Ist das nicht wunderbar einfach? So wird sie angewendet: Wollen wir den Zustand eines Pins im Sketch weiter verarbeiten (was sicher am meisten der Fall ist), dann erstellen wir uns erst mal eine Variable, in der dieser Zustand gespeichert werden kann. Dazu eignet sich der Datentyp boolean:

```
boolean pinstate;
```

Nun können wir mit der Anweisung digitalRead(pin) dieser Variable den eingelesenen Zustand des Pins übergeben:

```
pinstate = expander.digitalRead(3);       // Zustand Pin 3 einlesen
```

Die Variable pinstate hat nun entweder den Wert 1 (true), wenn der Taster nicht gedrückt wurde, oder aber 0 (false), wenn der Taster gedrückt wurde.

Eine LED blinken lassen (oder Buzzer)

Beispiel-Sketch verfügbar: blink.ino

Mit dieser Library ist es auch super einfach eine LED blinken zu lassen. Oder einen Buzzer hupen zu lassen. Das macht echt Spaß, ist eigentlich zu einfach... Die Anweisung lautet blink und wird folgendermaßen verwendet:

```
expander.blink(1, 10, 10000); // Blink led 2 zehn mal innerhalb von 10sek.
```

Wie es schon im Kommentar steht, blinkt die angeschlossene LED an Pin 1 des PCF8574 (Die Zählweise beginnt immer bei null bei diesem Chip) nun zehn mal innerhalb von 10 Sekunden.

Den Zustand eines Pins umkehren (togglen)

Beispiel-Sketch verfügbar: toggle.ino

Ein Ausgangspin kann den Zustand LOW (aus, 0V) oder HIGH (an, 5V oder 3,3V) haben. Um diesen Zustand einfach zu ändern, gibt es die Anweisung toggle. Diese macht genau das.
Man könnte nun einfach digitalWrite benutzen, um den Pin nach LOW oder HIGH umzuschalten. Das setzt jedoch voraus, man weiß, in welchem Zustand der Pin gerade ist. Das kann aber manchmal nicht der Fall sein.

```
expander.toggle(1);  // Toggle led 2
```

Es bewirkt das umschalten des Zustandes der LED an Pin 1 des PCF8574 (Die Zählweise beginnt immer bei null bei diesem Chip).

Portexpander MCP23008 / 23017

Bei diesen beiden ICs handelt es sich um Portexpander der Firma Microchip. Ob nun der PCF oder MCP eingesetzt wird, muss jeder selbst entscheiden. Es gibt sicher noch weitere. Ich habe ihn hier erwähnt, da ich mit diesen Typ ebenfalls schon gearbeitet habe.
Ähnlich wie den ersteren, ist hier der MCP23008 ein Chip mit 8 IO-Pins, der 23017 einer mit 16 IO-Ports.

Beschaltung

Siehe hierzu gleichnamigen Abschnitt zum PCF8574 zwei Seiten vorher. Das dort geschriebene gilt hier entsprechend.

Zum Einstellen einer bestimmten I2C.-Adresse stehen drei Pins zur Verfügung: A0, A1, A2. Diese können jeweils mit 0 (GND und 1 (+UB) beschaltet werden und ergeben so die I2C-Adresse gemäß der folgenden Tabelle:

A2	A1	A0	I2C-Adresse
0	0	0	0x20
0	0	1	0x21
0	1	0	0x22
0	1	1	0x23
1	0	0	0x24
1	0	1	0x25
1	1	0	0x26
1	1	1	0x27

Programmierung

Zur komfortablen Programmierung benutze ich wieder eine Library. Diese kann bei Quelle [22] runter geladen werden oder über den Librarymanager der IDE.

Zur Benutzung müssen wieder als erstes die beiden folgenden Libraries vor setup() in den Sketch eingefügt werden:

```
#include <Wire.h>
#include "Adafruit_MCP23017.h"
```

Nun wird die Library initialisiert:

```
Adafruit_MCP23017 mcp;
```

Im setup() wird nun die Verwendung der einzelnen Pins festgelegt und die I2C-Adresse:

```
mcp.begin(0x20);      // I2C-Adresse 0x20 verwenden

mcp.pinMode(0, OUTPUT);       // Pin 0 als Ausgang verwenden
mcp.pinMode(1, INPUT);        // Pin 1 als Eingang verwenden
mcp.pullUp(0, HIGH);          // 100kOhm internen Pullup aktivieren
```

Durch Verwendung unterschiedlicher I2C-Adressen und unterschiedlicher Namen in der Initialisierung können auch mehrere MCP's verwendet werden:

```
Adafruit_MCP23017 mcp0;        // erster MCP initialisieren
Adafruit_MCP23017 mcp1;        // zweiter MCP
```

Im setup() vergibt man dann die I2C-Adressen:

```
mcp0.begin(0x20);      // I2C-Adresse 0x20 verwenden
mcp1.begin(0x21);      // I2C-Adresse 0x21 verwenden
```

Und so können mit den entsprechenden Befehlen mit mcp0 und mcp1 die einzelnen MCP's konfiguriert werden.

```
mcp0.pinMode(0, OUTPUT);       // Pin 0 an mcp0 als Ausgang verwenden
mcp1.pinMode(1, INPUT);        // Pin 1 an mcp1 als Eingang verwenden
```

Nun wollen wir natürlich auch die Eingänge und Ausgänge verwenden. Dies geschieht im Sketch an der gewünschten Stelle mit den folgenden Befehlen:

```
mcp.digitalWrite(0, HIGH);            //  Pin 0 auf HIGH setzen
mcp1.digitalWrite(0, HIGH);           // oder so, bei mehreren MCP's

taster = mcp.digitalRead(1);          // Wert von Pin 1 nach taster einlesen
taster = mcp1.digitalRead(1);         // oder so bei mehreren MCP's…
```

Wie bei jeder Library sind auch hier Beispiele mitgeliefert, welche man als Anregung benutzen kann.

Der Watchdog-Timer und seine Benutzung

Allgemeines

Quelle: [7]
Haben Sie schon mal eine Situation gehabt, wo der ARDUINO hängt, abgestürzt ist? Ja? Dafür kann es viele Gründe geben. Zum Beispiel eine defekte Leitung zu einem Sensor im Freien, falsche Programmabläufe etc. Aber was, wenn der ARDUINO weit abgesetzt, schlecht zugänglich, oder ohne Überwachung durch den Menschen arbeitet? Diese Situationen sollten also vermieden werden. Oder wollen Sie Ihre geliebten Pflanzen, welche automatisch bewässert werden, nach ein paar Tagen vertrocknet oder verfault wiederfinden? Sicher nicht…

Die meisten Controller, so auch die AVR's im ARDUINO haben eine spezielle Funktion integriert, den sogenannten Watchdog Timer (deutsch: Wachhund). Ein schöner Name, genau das ist die Aufgabe dieses Hundes. Er überwacht das laufende Programm und bei Fehlfunktionen „beißt" er!
Wenn er einmal konfiguriert ist, dann wartet er auf sogenannte „Heartbeat-Signals". Sollten diese nicht in einer vorher festgelegten Zeit erscheinen, wird der Prozessor einen RESET ausführen.
Das ist sehr komfortabel und stellt sicher, dass der ARDUINO niemals länger als diese festgelegte Zeitschwelle im Programm hängen bleibt.

Über diese Dinge werden meines Wissens nach keine Worte verloren in den bekannten ARDUINO Dokumentationen. Es ist aber wichtig und spart wirklich eine Menge Zeit und Ärger.

Aber Vorsicht: Es gibt auch Fallstricke! Also unbedingt lesen!

Benutzung des Watchdog-Timers

Als erstes muss die folgende Zeile an den Programmanfang eingefügt werden:

```
#include <avr/wdt.h>
```

Im nächsten Schritt wird der Watchdog aktiviert und konfiguriert:

```
wdt_enable(WDTO_4S);   // Watchdog-Zeit 4 Sekunden
```

Diese Zeile sollte in den setup() des Sketches eingefügt werden. Der Watchdog kann aber prinzipiell zu jeder Zeit aktiviert und auch wieder deaktiviert werden.

WDTO_4S ist eine Konstante und bedeutet eine Zeitschwelle von 4 Sekunden. Diese Zeitschwellen sind vorprogrammiert, Sie können nicht jeden beliebigen Wert verwenden. Wenn der Wachhund innerhalb dieser Zeit nicht „gestreichelt" (zurückgesetzt) wird, dann „beißt" er: Der Controller führt einen RESET aus! Der Controller darf also für seine normale, bestimmungsgemäße Arbeit im Programm nie länger als diese Zeit (4Sek. In diesem Fall) brauchen! Gegebenenfalls diese Zeit erhöhen, oder den Programmlauf optimieren. Aber welches Programm braucht schon 4 Sekunden für einen Durchlauf – das ist eine Ewigkeit für einen Controller.

Die folgenden Zeitschwellen sind möglich, siehe Tabelle:

Schwellwert	Konstanten-Name	Verfügbar bei
15 ms	WDTO_15MS	ATMega 8, 168, 328, 1280, 2560
30 ms	WDTO_30MS	ATMega 8, 168, 328, 1280, 2560
60 ms	WDTO_60MS	ATMega 8, 168, 328, 1280, 2560
120 ms	WDTO_120MS	ATMega 8, 168, 328, 1280, 2560
250 ms	WDTO_250MS	ATMega 8, 168, 328, 1280, 2560
500 ms	WDTO_500MS	ATMega 8, 168, 328, 1280, 2560
1 s	WDTO_1S	ATMega 8, 168, 328, 1280, 2560
2 s	WDTO_2S	ATMega 8, 168, 328, 1280, 2560
4 s	WDTO_4S	ATMega 168, 328, 1280, 2560
8 s	WDTO_8S	ATMega 168, 328, 1280, 2560

Zum Schluss muss dem Watchdog noch gesagt werden, dass alles OK ist („Streicheln“):

```
wdt_reset();          // Watchdog Zeit zurücksetzen
```

Natürlich muss dieser Reset-Befehl öfter als die in der Konfiguration angegebene Zeit (hier 4 Sekunden) aufgerufen werden. Am besten gleich am Anfang der `void loop()`!
Aber es hängt auch vom Programm ab. Wenn das Programm sehr lang in einer Funktion (Unterprogramm) hängt, dann kann es auch nötig sein, den `wdt_reset()` dort an geeigneter Stelle unterzubringen. Hier ist also Aufmerksamkeit erforderlich, um immer den Watchdog Timer rechtzeitig zurück zu setzen.
In allen zeitraubenden Funktionen muss dieser Reset eingefügt werden! Also auch z.B. bei Datenübertragungen, Warteschleifen, Auslesen von Sensoren etc.

Falls nötig, kann der Watchdog auch wieder deaktiviert werden. Dies geschieht mit dem Befehl

```
wdt_disable();        // Watchdog deaktivieren
```

und kann auch an jeder beliebigen Stelle im Sketch erfolgen. Dann aber nicht vergessen, ihn auch wieder zu aktivieren, wenn das gewünscht ist.

Bei der Auswahl der Reset-Zeitschwelle müssen alle zeitbrauchenden Funktionen mit beachtet werden, also delay(), auszuführende Funktionen, Bus-Timeouts (wenn vorhanden) und Arbeitsgeschwindigkeit. Weiterhin sollte man im Auge behalten, das nur der ARDUINO zurückgesetzt wird. Alle andere eventuell angeschlossene Hardware nicht, wie GSM-Shield, Network-Shield usw. Sollen diese auch zurückgesetzt werden, ist dieses eventuell zu verdrahten oder anders vorzusehen.

Die Zeitschwelle sollte nicht zu klein gewählt werden! Ansonsten kann es vorkommen, das der ARDUINO nicht aus dem Bootloader rauskommt, weil schon vorher ein Reset erzeugt wird! Dann geht nichts mehr, auch nicht das neuprogrammieren!! Nur ein erneutes Aufbrennen des Bootloaders mit einem geeigneten AVR-Brenner oder einem anderen ARDUINO als Programmieradapter ist dann noch möglich!
Also lieber eine längere Zeit wählen, nicht unter 2 Sekunden. Ich benutze 4 Sekunden oder 8 Sekunden, da ist man auf der sicheren Seite.

Hinweis bei Benutzung von China-Klones

Nach einem Hinweis eines Forumsnutzers (-Christoph-) kann es Probleme mit China-Klonen geben. Dies äußert sich im „Einfrieren“ des ARDUINO und kann mit dem folgenden Code umgangen werden.
Diese Codeschnipsel stammen aus Quelle [29]

```
  wdt_reset();          //watchdog Zeit zurueck setzen

  WDTCSR = (1 << WDCE) | (1 << WDE);  //WDT interrupt einstellen

  //Start watchdog timer mit 4s Vorteiler
  WDTCSR = (1 << WDIE) | (1 << WDE) | (1 << WDP3) | (1 << WDP0);
```

Die oben angefügten Codeschnipsel wurden nicht von mir getestet. Meine WDT-Anwendung funktionierte mit dem Standard-Code.

Von delay bis zur Methode

Allgemeines

Dieses Kapitel wurde mir dankensweise von Heinz Baumstark zur Veröffentlichung in dieser Referenz zur Verfügung gestellt.
Er stellt eine fortgeschrittene Art der Programmierung dar, wie man Zeitverzögerungen, Fading etc. elegant programmieren kann und so nur einmal im Sketch einfügt und dann universell anwenden kann. Für Anfänger ist dieses Kapitel möglicherweise (noch) nicht geeignet, aber sehr lesenswert, wenn man weiter voran kommen möchte. Viel Spaß damit!

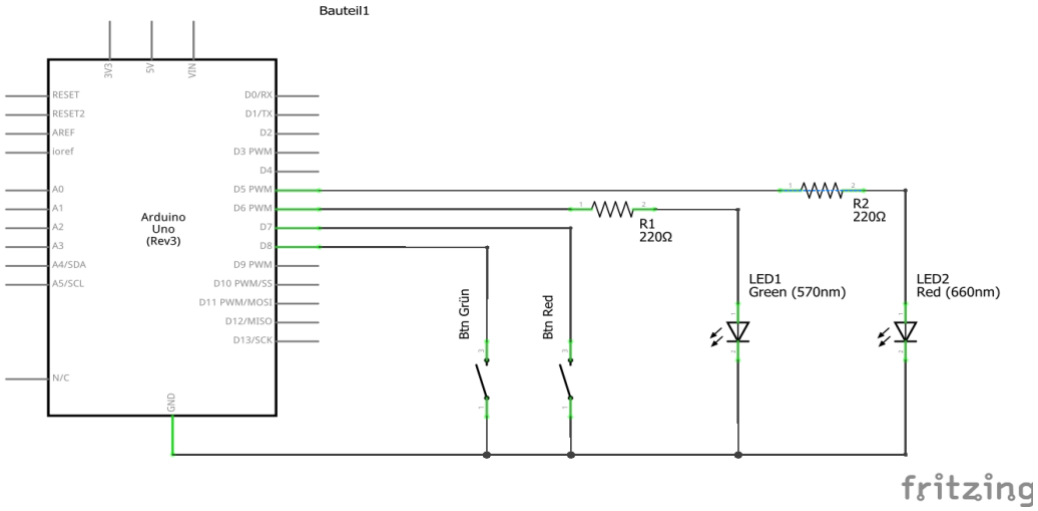
Worum geht es?

Dies soll ein kleines Tutorial sein, das dem Anfänger anhand eines einfachen Beispiels zeigen soll wie man *delay()* durch *millis()* ersetzt und was der Nachtwächter damit zu tun hat. Wir werden uns eine eigene Funktionen schreiben, eine eigene Klasse erstellen und daraus ein Objekt verwenden. Ich werde hier nicht darauf eingehen was Funktionen und Klassen sind, dazu gibt es Bücher oder z.B. https://de.wikibooks.org/wiki/C%2B%2B-Programmierung/_Inhaltsverzeichnis oder http://www.cplusplus.com/doc/tutorial/

Es geht mir darum zu zeigen, wie man eine bestimmte Aufgabe mit verscheiden Möglichkeiten mehr oder weniger gut realisieren kann. Es wird im Grunde immer der gleiche Sketch mit den gleichen Variablennamen verwendet, was es dem Anfänger leicht macht die Unterscheide zu erkennen. Wir werden zunächst einen Grundsketch erstellen und dann Schritt für Schritt vorgehen. Zu diesem Tutorial gehören mehrere Sketche, die mittels IDE auf den Arduino geladen und getestet werden können. (Zum Download verfügbar)

Als Beispiel habe ich mir eine einfache Aufgabe ausgedacht die vor einiger Zeit hier im Forum behandelt wurde. Dabei soll mit einem Tasters eine LED eingeschaltet werden. Die LED soll dabei langsam heller werden. Wird der Taster losgelassen soll die LED langsam wieder dunkler werden. Und weil es so schön ist, wollen das zweimal haben, je für eine rote LED und für eine grüne LED.

Wenn wir nun unter den mitgelieferten Beispielen in der IDE suchen, werden wir das Beispiel *Fading* finden. Wenn wir den Sketch auf unseren Arduino laden stellen wir fest das die LED schön hell und dunkel wird, genau das was wir wollen. Leider finden wir aber keinen Taster mit dem wir das ein oder ausschalten könnten. Und das ist bei der verwendeten for-Schleife auch ziemlich unsinnig. Wenn Anfänger darüber nachdenken kommt es häufig zu der Frage: „Wie kann ich eine for-Schleife vorzeitig abbrechen? “. Da ist also schon der Ansatz falsch. Was liegt also näher als das neu zu machen. For Schleifen brauchen wir hier nicht wir haben ja schon loop als Dauerschleife. Natürlich gibt es sinnvolle Verwendungen von for-Schleifen, schließlich gibt es die in jeder Programmiersprache. for-Schleifen machen aber, wie alle anderen Schleifen auch, bei einem Controller nur dann Sinn wenn sie schnell abgearbeitet werden. Das ist aber sicher nicht der Fall wenn ein *delay()* mit in der Schleife steht und die Schleife womöglich 1000 mal durchlaufen werden soll.



Schaltplan zum Tutorial

Der Grundsketch

Gerade für Anfänger ist es gut zunächst einen Programmablauf zu erstellen, hilft er doch die Gedanken logisch zu sortieren. Oft stellt man dabei auch schon mal fest das irgend etwas nicht logisch ist, in dem geplanten Vorgehen. Für unseren Grundsketch könnte das so aussehen.

wenn der Taster gedrückt ist dann
und wenn der max Wert noch nicht erreicht ist dann
wert um Schritt erhöhen
Wert an LED ausgeben
etwas warten
Wenn Taster nicht gedrückt (else) dann
und wenn min Wert noch nicht erreicht dann
wert um Schritt verringern
Wert an LED ausgeben
etwas warten
zum Anfang

Mit etwas Übung kann man nun aus fast jeder einzelnen Textzeile eine Codezeile machen und schon ist unser Grundsketch im Wesentlichen fertig. In den Allgemeinteil und das Setup kommen noch ein paar Definitionen für Variable und die Festlegung der E/A Pin´s. Um die Geschwindigkeit des „Fading" leichter ändern zu können, werden die Schrittgröße und die Wartezeit als Variable im Allgemeinteil vorgegeben. Die Pins und Variable haben aussagekräftige Namen erhalten aus denen sich ablesen lässt für was sie gut sind.

Im Setup finden wir noch die Zeile *Serial.begin(9600)*. Eigentlich brauchen wir das nicht, ich schreibe es aber immer schon mal mit dazu, dann kann man sich zum Finden eines Fehlers schnell mal ein *Serial.print()* an geeigneter Stelle mit einbauen.

Im Loop finden wir ein *else*, das zu einem *if* etwas weiter oben gehört. Wer nicht weiß was das ist und wie das funktioniert sollte hier jetzt erst mal eine Pause einlegen und im folgenden Link weiter lesen und üben. reference/de/language/structure/control-structure/else/

Übrigens sprechen Anfänger schon mal von if-Schleifen. If ist aber keine Schleife sondern eine Verzweigung. So jetzt geht's aber los.

Zu dem obigen Abschnitt gehört der Sketch fade1.ino .

Den Sketch für die zweite LED erweitern

Nun, das ist eigentlich ganz einfach. Wir erweitern unseren Grund-Sketch um die Festlegung der beiden neuen Pins für den zweiten neuen Taster und die grüne LED. Dann benötigen wir noch eine neue Variable für die Helligkeit der grünen LED.

Das was bisher im loop steht können wir mit copy&paste nochmal hinzufügen. Anschließend müssen wir noch die Namenszusätze der Variablen in dem neuen, zweiten Teil von Rot auf Gruen ändern. Wichtig dabei: verwendet keine Umlaute der Compiler mag das nicht.

Der ein oder andere Leser wird sich nach dem Test des Sketches fade2.ino am Ziel seiner Wünsche sehen. Schließlich haben wir jetzt einen Sketch der die Aufgabe augenscheinlich gut erfüllt. Mit je einem Taster wird eine LED langsam heller und dunkler und das unabhängig voneinander. Wenn wir allerdings genau hinschauen werden wir feststellen, das die Zeit des Fade Vorganges nicht konstant ist. Laufen gerade beide Vorgänge gleichzeitig, kann es bis zu doppelt so lange dauern bis der Vorgang abgeschlossen ist. Nun wie kommt das denn ? Wenn wir uns den Programmablauf genau ansehen, stellen wir fest, das *delay()* in einem Umlauf des Loop zweimal bearbeitet wird wenn beide fade Vorgänge gleichzeitig laufen. Damit ist die Umlaufzeit des Loop nicht konstant und damit die Geschwindigkeit des Fade Vorganges auch nicht.

Nun kann man auf die Idee kommen, die vier einzelnen *delay()* raus zu nehmen und durch ein einziges am Ende zu ersetzen. Probiert es aus, das funktioniert und macht genau was es soll, die fade-Zeit ist jetzt konstant, da die Loop-Zeit immer konstant ist. Allerdings hängt der Arduino bei jedem Umlauf in dem *delay()* in einer Pause fest, und macht da nichts als warten. Er wird eigentlich blockiert durch das *delay()*. Für unser Beispiel spielt das keine Rolle. Wenn man sich allerdings vorstellt das der Controller quasi gleichzeitig auch noch etwas Anderes, eventuell sogar Zeitkritisches, machen soll dann wird das so nichts werden.

Zu dem obigen Abschnitt gehört der Sketch fade2.ino

delay() durch millis() ersetzen

Schaut Euch mal das Beispiel „Blink“ aus der IDE an. Denkt mal darüber nach wie Ihr zu der dauernd blinkenden LED mit einem zusätzlichen Taster, eine zweite LED einschaltet solange der Taster gedrückt wird. Wer möchte kann das gerne mal versuchen. Bestenfalls bekommt Ihr es so hin, das die zweite LED an oder aus geht, wenn auch die blinkende LED ein oder aus geschaltet wird. Wenn man nur kurz auf den Taster drückt passiert meist gar nichts. Warum ist das so ? Nun es liegt an dem verwendeten *delay()*. Damit wird der Programmablauf angehalten, und es geht erst dann weiter wenn die Zeit für *delay()* abgelaufen ist. Der Controller kann nicht feststellen ob inzwischen der Taster gedrückt wurde weil der Programmablauf durch das *delay()* blockiert wurde und während dessen keine Abfrage des Tasters erfolgt.

Wenn man das blockierende *delay()* ersetzen will, und mal die Suchmaschine anwirft, wird man relativ schnell bei dem Beispiel „BlinkWithoutDelay“ und der Nachtwächter-Erklärung, im Forum forum.arduino.cc , landen. Jeder der sich ein bisschen ernsthaft mit der Arduino Welt beschäftigen will, sollte das verstanden haben, und morgens im Halbschlaf anwenden können. Was *millis()* nun eigentlich ist findet man in der Referenz z.B hier Arduino Reference (oder auch hier) oder auch über die Hilfe in der IDE.

Es gibt sicher auch tausende Erklärungen im Netz dazu wie man *millis()* einsetzt. Zudem gibt es etliche Bibliotheken die Timer Objekte bereitstellen und damit die Verwendung einfacher machen.

Was hat nun unser Aufgabe mit dem Nachtwächter zu tun? Lasst es mich mal so erklären. Wenn es dunkel wird (Taster gedrückt) dreht der Nachtwächter seine Runden. Eine Runde dauert 5 Minuten. Alle 15 Minuten soll er die Lampe kontrollieren. Immer wenn er an einer Lampe vorbeikommt schaut er auf seinem Zettel nach, wann er das letzte Mal hier etwas getan hat. Wenn er feststellt das es Zeit ist wieder was zu tun, schreibt er die aktuelle Uhrzeit auf seinen Zettel und schaut nach ob die Lampe schon ganz hell ist. Wenn nein, berechnet er wie hell er die Lampe jetzt einstellen muss. Wenn er feststellt sie ist schon ganz hell, macht er nichts. Ansonsten stellt er den neuen Wert an der Lampe ein und geht weiter seine Runde. Das macht er so die ganze Nacht. Wenn es nun hell wird (Taster nicht gedrückt) geht er ähnlich vor, nur das er die Lampe jedes Mal etwas dunkler stellt, wenn es an der Zeit ist. Jetzt stellen wir also fest, der arme Kerl ist eine arme Socke, er rennt da Tag und Nacht herum und hat fast nichts zu tun. Unserem Controller geht es ähnlich, der langweilt sich nämlich eigentlich auch. Ich hoffe ich hab Euch jetzt nicht komplett verwirrt , das war nicht meine Absicht.

Also, das oben Geschriebene kann man in Grunde in wenige Codezeile packen. Damit es zu keinen Überlauf oder Fehler kommt wenn *millis()* irgendwann, nach etwa 40 Tagen, mal wieder von vorne anfängt, hat sich die Abfrage, ob etwas zu tun ist, in folgenden Form durchgesetzt.

```
if(millis() - altzeit >= delaytime){      // ist die Zeit schon um
  altzeit = millis();                     // aktuelle Zeit merken
  // mach hier was immer zu tun ist
}
```

Zudem sollten alle Werte, die irgendwie mit *millis()* in Verbindung kommen, vom Datentyp unsigned long oder uint32_t sein. Wie Ihr die Variable dabei benennt bleibt Euch überlassen. Für unsere beiden LED´s müssen wir das natürlich getrennt machen. Für jede LED müssen wir uns merken wann wir das letzte Mal da waren. Klar das muss der Nachtwächter ja auch so machen wenn er seine Runde dreht. Wir deklarieren unser Variablen also so:

```
uint32_t delaytime = 50;
uint32_t altzeitRot, altzeitGruen;
```

Wenn Ihr nicht genau wisst was es mit den Datentypen auf sich hat dann solltet Ihr das nochmals nachlesen.

Zusammengefasst: Wenn unser Code nicht blockieren soll, oder wenn wir mehrere Dinge gleichzeitig machen wollen, können wir *delay()* nicht verwenden.

Zu dem obigen Abschnitt gehört der Sketch fade3.ino

Eine eigene Funktion erstellen

Unser Sketch fade3.ino verwendet jetzt zwei Teile die im Wesentlichen gleich sind. Sie unterscheiden sich lediglich dadurch, das bei den Variablennamen einmal Rot und einmal Gruen als Zusatz verwendet wird. Um Code mehrfach zu verwenden sind Funktionen bestens geeignet. Oft werden Funktionen auch verwendet um lediglich das Programm zu strukturieren. Häufig werden dann darin globale Variable verwendet oder auch verändert. Das sieht dann oft so aus:

```
void funktionsname()
```

Es gibt weder Rückgabewert noch einen Übergabewert. Das widerspricht dem eigentlichen Grundgedanken, schließlich sollen Funktionen universell und mehrfach einsetzbar sein. Dennoch macht

es viel Sinn zur Strukturierung eines Programmes so vorzugehen. Programme lassen sich so besser lesen, insbesondere dann, wenn der Funktionsname schon beschreibt was darin passiert.

Wenn man eine Funktion schreiben will, sollte man sich gut überlegen was mit in die Funktion rein soll und wie die Schnittstelle aussieht. In unserem Fall habe ich mich dazu entschieden drei Variable zu übergeben. Die beiden letzten davon kennen wir bereits.

```
void fadeLed(bool ein, uint32_t &altzeit, byte &hell)
```

das & Zeichen vor den beiden Variablen legt hier fest das die betreffende Variable als Referenz übergeben wird, und damit von der Funktion geändert werden kann. Die Variable ist dann sowohl Übergabe als auch Rückgabewert. Zum Nachlesen *„Übergabe als Wert oder Referenz“* sind hier das Stichwort. Der Name der internen Variablen und der Name von Variablen im Funktionsaufruf, haben nicht miteinander zu tun, sie können völlig unterschiedlich sein, können aber auch gleich sein. Ich nenne die interne Variable hier z.B altzeit, schließlich wird sie für beide LED benutzt. Beim Funktionsaufruf wir dann einmal *altzeitRo*t und einmal *altzeitGruen* übergeben. Der aufrufende Code muss nach der Rückgabe sicherstellen das die Variablen bis zum nächsten Mal gespeichert bleiben.

Die bool Variable „ein“ stellt den Status des Tasters dar, er muss nicht zurück gegeben werden, und sollte auch nicht von der Funktion verändert werden. Er wird also als Wert übergeben und ohne & verwendet. Dazu gibt es noch etwas Neues. Wenn Wir bisher den Taster Status benötigt haben, dann haben wir Ihn mit *digitalRead(pin)* eingelesen. Ab jetzt lesen wir den Taster nur einmal je Umlauf ein und speicher das Ergebnis in einen bool Variablen, die wir dann an die Funktion übergeben.

```
statrot = !digitalRead(btnRot);       // taste einlesen
```

Das macht bei Statuswerten die nur 0/1 bzw. true / false sein können eigentlich immer Sinn, da sie sich einfacher abfragen und logisch verknüpfen lassen. Mit der Ausrufezeichen vor *digitalRead* wird der Wert noch invertiert, damit ist jetzt bei gerücktem Taster der Wert von *statrot = true ,* was eigentlich unsere menschlichen Logik entspricht.

So macht z.B. die Zeile

```
if( stat1 & stat2 & !stat3)
```

durchaus Sinn und lässt sich auch einfach lesen. Aber das ist ein ganz anderes Thema. Hier nun der fertige Sketch mit unserer Funktion.

Zu dem obigen Abschnitt gehört der Sketch fade4.ino

Im Sketch fade4.ino habe wir die Funktionalität des Faden´s in einer Funktion mit dem Namen fadeLed zusammen gefasst. Diese Funktion können wir mehrfach verwenden. Wir übergeben beim Aufruf der Funktion die aktuellen Werte an die Funktion, und bekommen von Ihr die neuen geänderten Werte zurück geliefert.

Jetzt erarbeiten wir noch eine etwas geänderte Funktion. Im obigen Beispiel ist es ja so, das der aufrufende Code die Werte der Variablen *„altzeitRot, altzeitGruen, hellRot“* und *„hellGruen“* speichern muss. Schön wäre es, wenn die Funktion das selber machen könnte, und zudem einen Rückgabewert auf „normalem“ Weg über *return* liefert würde. Dazu bietet sich die Möglichkeit an die Variablen innerhalb der Funktion zu deklarieren und, damit sie bei Verlassen der Funktion nicht verloren gehen, zusätzlich als „static“. Der aufrufende Code müsste dann mitteilen, ob es sich um die Rote oder Grüne LED handelt. Damit das möglichst einfach geht, werden wir die beiden Variablen in der Funktion in je einem Array speichern. Über den Index sprechen wir das richtige Element des Array an. Der aufrufende Code benötigt

die Variablen nun nicht mehr. Da unsere Funktion jetzt einen richtigen Rückgabewert liefert, können wir den Funktionsaufruf auch gleich in unserer Ausgabe verwenden.

```
analogWrite(LedGruen, fadeLed(statgruen,1));
```

Zu dem obigen Abschnitt gehört der Sketch fade 4a.ino

Zusammengefasst: Funktionen geben uns die Möglichkeit unseren Code zu strukturieren und Code mehrfach zu verwenden. Die Möglichkeiten zur Speicherung von Daten für einen mehrfachen Aufruf sind jedoch begrenzt.

Wer hier noch etwas üben will, kann sich ja mal überlegen wie man die Funktion so ändern kann, das das Einlesen des Tasters und die Ausgabe an die LED mit in der Funktion ist. Als Parameter müsste dann der Pin für Taster und LED an die Funktion übergeben werden.

Objekte und Methode

Tatsächlich wurden Konstrukte die Daten innerhalb der Funktion speichern, früher häufig verwendet. Schlaue Leute kamen dann auf die Idee Klassen zu erfinden. Mit Klassen ist es möglich den obigen Nachteil von Funktionen zu umgehen. Datenspeicher und Funktionen werden hier zusammengeführt. In das Thema Klassen, Objekte, Methoden müsst Ihr Euch wieder selber einlesen. Ich zeige Euch wie das in unserem Beispiel verwendet werden kann.

Wir benötigen also eine Klasse die unsere Daten aufnehmen und speichern kann und eine Methode die die gewünschte Funktionalität zur Verfügung stellt. Etwas vereinfacht könnte man sagen Funktionen heißen jetzt Methoden.

Zur Erzeugung einer Klasse wird am Anfang das Schlüsselwort *class* verwendet und am Ende eine geschweifte Klammer mit einem Semikolon. Innerhalb wird festgelegt was privat und was public ist. Eigentlich ist das Schlüsselwort „*private*“ nicht erforderlich, dennoch wird es häufig verwendet. In unserem Beispiel soll der Datenspeicher für die Variablen privat sein. Die Methode jedoch muss public sein damit sie von außen zugänglich ist.

Zusätzlich benötigen wir noch einen Konstruktor, dieser wird bei der Erstellung einer Instanz automatisch bearbeitet. Er kümmert sich zur Laufzeit um das Anlegen der Daten im Speicher und in unserem Fall um die Initialisierung der Werte fadstep und delaytime innerhalb Instanz.

Wir erzeugen uns also zwei Objekte der Klasse Fade mit unterschiedlichem Namen und übergeben unsere Werte zur Initialisierung der internen Daten an den Konstruktor. Jedes Objekt stellt uns dann seine Eigenschaften und Methoden zur Verfügung.

```
Fade LEDrot(fadstep,delaytime), LEDgruen(fadstep,delaytime);// zwei Objekte erzeugen
```

Auf die Methode eines Objektes wird über deren Namen und einen Punkt zugegriffen. Ähnlich wie bei einer Struktur (struct).

```
byte wert = LEDrot.fade(statrot);
```

eine Methode kann wie eine Funktion einen Wert zurück liefern. Für die Übergabeparameter gelten die gleichen Bedingungen wie bei Funktionen.

Zu dem obigen Abschnitt gehört der Sketch fade5.ino

Array mit Objekten

Jetzt möchte ich noch zeigen wie man den letzten Sketch unter Verwendung von Arrays noch so umbauen kann, das der eigentliche Code sehr klein wird. Für unser Beispiel mit nur 2 LED trifft das allerdings nicht zu, aber es geht ja auch nur um´s Prinzip hier. Leider geht dabei die Information, ob es sich gerade um die rote, oder die grüne LED handelt, verloren. Anstelle dessen müssen wir uns mit dem Index als Unterscheidung anfreunden. Oft ist es aber tatsächlich auch egal ob die LED rot oder grün ist und die Unterscheidung durch den Index völlig ausreicht.

Natürlich macht es Sinn Eingänge und Ausgänge ebenfalls als Array´s anzulegen.

```
const byte led[] {5, 6};// Pin´s für LED zwei Elemente
const byte btn[] {7, 8};// Pin´s für Button
```

An der Klasse selbst wurden keine Änderungen gemacht. Etwas Tricki war noch die Erstellung der beiden Objekte, die jetzt ebenfalls als Array vorliegen sollen. Wegen der Übergabe der Werte *delaytime* und *fadstep* an den Konstruktor ist eine einfache Erstellung der beiden Objekte in der Form: Fade LED[2] nicht möglich. Es sind hier jedoch zwei Varianten möglich.

```
// Variante 1 zwei Objekte erstellen und Parameter einzeln übergeben
Fade LED[2] {{fadstep, delaytime}, {fadstep, delaytime}};
```

oder

```
// Variante 2 zwei Objekte erstellen und parameter an alle übergeben
Fade* LED = new Fade[2](fadstep, delaytime);// Besser für viele Elemente
```

damit wird ein Array der Objekte erzeugt. An jedes Objekt werden wieder die Werte zur Initialisierung der internen Variablen an den Konstruktor übergeben. Angesprochen wird unsere Methode z.B dann mit:

```
byte hell= LED[Index].fade(statusBtn);
```

Die analoge Ausgabe auf den Pin können wir dann auch in einer Zeile zusammen fassen.

```
analogWrite(led[Index], LED[Index].fadeLed(!digitalRead(btn[Index])));
```

Um die Elemente der Array´s zu verarbeiten und anzusprechen bietet sich, wie sollte es anders sein, diesmal eine for Schleife an.

Zu dem obigen Abschnitt gehört der Sketch Fade5a.ino

Zusammengefasst: Wenn wir einen Code mehrfach nutzen wollen, und gleichzeitig für jeden Aufruf des Code´s einen Datenspeicher benötigen, ist der Einsatz von Objekten und Methoden genau richtig.

An dieser Stelle bedanke ich (Heinz Baumstark) mich bei Tommy56 der mich als Co-Moderator unterstützt, und mir besonders beim Thema Klassen auf die Sprünge geholfen hat.

Jetzt hoffe ich das ich mich verständlich ausgedrückt habe, das es Spaß gemacht hat, und Ihr etwas gelernt habt.

*++++ **Ende des Artikels von Hans Baumstark** ++++*

Kleine hilfreiche Programm-Teile

Freien RAM anzeigen

Quelle: unbekannt, Internet
Mit dieser Funktion kann der noch im ARDUINO verfügbare freie RAM-Speicher angezeigt werden.
Die Funktion muss vorerst in den eigenen Code, wie ein „Unterprogramm“ eingefügt werden, also NICHT in setup() oder loop(). Am besten dahinter.

```
int freeRam () {
   extern int __heap_start, *__brkval;
   int v;
   return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int) __brkval);
}
```

Die Abfrage des freien RAM erfolgt dann wo es gewünscht ist im Programm zum Beispiel durch den Befehl:

```
Serial.print(freeRam());              // Ausgabe seriell USB
LCD.print(freeRam());                 // Ausgabe LCD
```

Programm anhalten (für immer)

Manchmal möchte man die komplette Programmausführung für immer (bis die Spannung abgeklemmt wird) anhalten, sodass der Prozessor absolut nichts mehr machen kann. Fragt vielleicht der ein oder andere: Warum? Nun, ich habe eine Überwachung der Spannung eines Bleiakkus programmiert. Damit dieser nicht tiefentladen wird, soll der Verbraucher per Relais getrennt werden. Und auch nie wieder angehen, bis ich mich um den Akku kümmere. Nach Abschalten durch das Relais soll also der Controller „anhalten“. Dies kann man mit diesem einfachen kurzen Befehl machen:

```
while(1);
```

Software-Reset

Manchmal möchte man definiert einen Reset des ARDUINO durchführen, also so, als hätte man gerade eben die Spannung angelegt und das Programm beginnt.
Das kann man mit Hardware realisieren, also z.B. einen Transistor an einen digitalen Ausgang und den Kollektor des Transistors an den Reset-Pin anschließen. Den Reset erreicht man dann durch einen HIGH-Pegel am Ansteuerpin mit digitalWrite.

Es gibt aber auch eine einfache Software-Möglichkeit. Hier braucht man keine zusätzlichen Bauteile.
Der Befehl ist ein simpler Assembler-Befehl, welcher ins Programm als Funktion eingefügt werden kann (siehe „Freien RAM anzeigen“):

```
void software_Reset()
 {
  asm volatile ("  jmp 0");
 }
```

Der Aufruf der Funktion geschieht dann an gewünschter Stelle mit dem Befehl:

```
software_Reset();              // Software-Reset ausführen
```

RAM sparen bei Serial.print

Wenn man viele String-Ausgaben macht mit dem Befehl Serial.print, kann man RAM-Speicher sparen, indem diese Strings im Flash abgespeichert werden. Das ist ganz einfach zu erreichen durch folgende Vorgehensweise:

```
Serial.println(F("Hallo Welt!"));             // speichert den String im Flash
```

Siehe auch hier in der Referenz

Digital-Ausgang (z.B. LED) togglen

Will man den Zustand eines Digital-Ausgangs umkehren, also von HIGH nach LOW oder umgekehrt, so kann man dazu den folgenden Ausdruck verwenden:

```
digitalWrite(LED, !(digitalRead(LED)));              // Pin togglen
```

Binärausgabe 8bit, 16bit, 32bit auf seriellem Port

Die folgenden drei Programmteile erzeugen die Ausgabe eines byte, int, long-Wertes als Binärwert auf den seriellen Port. Dies kann nützlich sein, um den Code bei Fehlfunktionen mit Bitoperationen o.ä. zu debuggen. So muss man nicht erst die dezimale Zahl z.B. **17** umständlich in das Format **B00010001** umrechnen. Danke hierfür **ardu_arne** vom deutschen ARDUINO-Forum.

Binärausgabe 8bit

```
void binout8 (byte wert) {     // Binaerausgabe 8 Bit
  Serial.print("B ");
  for (int i = 7; i >= 0; i--) {
    if ( i == 3 )            Serial.print(" ");
    if (bitRead(wert, i))    Serial.print("1");
    else                     Serial.print("0");
  }
  Serial.println();
}
```

Um nun einen Wert als Binärzahl auf den seriellen Port auszugeben, ist folgende Programmzeile nötig:

```
binout8(170);          // liefert "B 1010 1010" der dez. Zahl 170
```

Binärausgabe 16bit

```
void binout16 (unsigned int wert) {    // Binaerausgabe 16 Bit
  Serial.print("B ");
  for (int i = 15; i >= 0; i--) {
    if ( i == 3 || i == 11)    Serial.print(" ");
```

```
    if ( i == 7)               Serial.print("  ");
    if (bitRead(wert, i))      Serial.print("1");
    else                       Serial.print("0");
  }
  Serial.println();
}
```

Um nun einen Wert als Binärzahl auf den seriellen Port auszugeben, ist folgende Programmzeile nötig:

```
binout16(43690);      // liefert "B 1010 1010  1010 1010" der dez. Zahl
```

Binärausgabe 32bit

```
void binout32 (unsigned long wert) {   // Binaerausgabe 32 Bit
  Serial.print("B ");
  for (int i = 31; i >= 0; i--) {
    if ( i == 3 || i == 11 || i == 19 || i == 27)  Serial.print(" ");
    if ( i == 7 || i == 15 || i == 23)             Serial.print("  ");
    if (bitRead(wert, i))                          Serial.print("1");
    else                                           Serial.print("0");
  }
  Serial.println();
}
```

Um nun einen Wert als Binärzahl auf den seriellen Port auszugeben, ist folgende Programmzeile nötig:

```
binout32(2863311530); // liefert "B 1010 1010  1010 1010  1010 1010  1010 1010"
```

String nach char kopieren

Quelle: deutsches ARDUINO-Forum, Mitglied „Tommy56“
Manchmal besteht die Notwendigkeit, eine Variable vom Typ String in eine Variable vom Typ char-Array zu kopieren. Das geht so ohne weiteres nicht. Mit diesem kleinen Codeschnipsel ist es schnell erledigt:

```
String Text = "Hallo Welt";    // der String, welcher kopiert werden soll
char mychar[20];               // das char-Array, in welches kopiert werden soll

strcpy(mychar,Text.c_str());
```

RAM sparen bei der Verwendung von Ethernet

Quelle: Rene, per Email
Viele Anwendungen nutzen Verbindungen mithilfe eines Netzwerk. Sei es zur Synchronisation der Zeit mittels NTP oder Übergabe von Daten an (MySQL-)Datenbanken.
ARDUINO verfügt über eine eigene Bibliothek mit der die Konnektivität hergestellt werden kann.
Und hier gibt es großes Sparpotenzial.

Ob in einem privaten Heimnetz sich zwingend jedes Gerät selbst konfigurieren muss, ist eine Frage die eindeutig mit NEIN beantwortet werden kann. Geräte die nicht über die Grundstücksgrenze hinweg

kommunizieren, benötigen weder einen Gateway noch eine Subnetzmaske. Beim DNS kann man geteilter Meinung sein. Wenn im Haus z.B. ein RASPI als DNS-Server existiert, kann man den nutzen.

Das Einzige was zwingend konfiguriert werden muss, ist die MAC-Adresse.
Die minimalste Initialisierung lautet dann:

```
Ethernet.begin(mac);
```

Alle weiteren Einstellungen würde sich der ARDUINO über das Netzwerk holen.
Vorausgesetzt es gibt dafür einen funktionsfähigen DHCP-Server (z.B. Fritz.Box:-)). Fehlt dieser, ggfls. auch durch einen kurzfristigen Ausfall, würde keine IP-Konfiguration erfolgen. Zudem muss regelmäßig mit

```
Ethernet.maintain();
```

dem DHCP-Server mitgeteilt werden, das der ARDUINO noch "lebt".

Die Nachteile aus dem obigen Absatz können aber schon allein durch die Angabe einer festen IP-adresse beseitigt werden. Und wer jetzt meint, im Code ist doch dafür ein zusätzliches 4-Byte-Array und damit zusätzlicher Speicherplatz notwendig, irrt. Die Variablen werden sonst durch die Lib angelegt wenn die IP-Adresse via DHCP bezogen wird.
Um das anschaulich zu machen, kann folgender Sketch kompiliert werden (geht trocken ohne angeschlossenen ARDUINO - Hochladen nicht nötig):

```
/*
   Beispiel RAM-Verbrauch vs. Bequemlichkeit

 minimal: Ethernet.begin(mac);
 maximal: Ethernet.begin(mac, ip, dns, gateway, subnet);
 */

const byte option = 1; // 1 nur Mac, 2 ip, 3 gateway, 4 dns, 5 subnet

#include <SPI.h>
#include <Ethernet.h>


byte mac[] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
byte  ip[] = {192, 168, 0, 177};
byte  gw[] = {192, 168, 0, 1};
byte  ns[] = {192, 168, 0, 5};
byte sub[] = {255, 255, 255, 0};


void setup() {
  switch (option){
  case 1:
    Ethernet.begin(mac);
    break;
  case 2:
    Ethernet.begin(mac, ip);
    break;
  case 3:
    Ethernet.begin(mac, ip, gw);
    break;
  case 4:
    Ethernet.begin(mac, ip, gw, ns);
    break;
  case 5:
```

```
    Ethernet.begin(mac, ip, gw, ns, sub);
    break;
  }
}

void loop() {
  //
}
```

Während mit der Option=1 der Sketch noch 10632 (11338) Bytes groß ist, sind es bei Option=2 nur 3458 (3580) Bytes und bei Option=5 sogar nur 3062 (3416) Bytes insgesamt in ARDUINO 1.0.6 (Werte in Klammern in ARDUINO 1.8.6) was einer Freigabe von fast 2/3 des benutzen Speicherplatz bzw. 20%(!) des Gesamtspeichers entspricht.

Quellenverzeichnis

[0] **https://arduinoforum.de**
[1] http://torrentula.to.funpic.de/dokumentation/code-referenz (Link inaktiv)
[2] https://www.arduino.cc/reference/en/ sowie auch https://www.arduino.cc/reference/de/
[3] http://www.arduino-tutorial.de/
[4] http://popovic.info/html/arduino/arduinoUno_1.html#LCD CC BY-SA
[5] http://www.macherzin.net/article38-LC-Displays-am-Arduino (Link inaktiv)
[6] http://playground.arduino.cc/ CC BY-SA
[7] http://tushev.org/articles/arduino/item/46-arduino-and-watchdog-timer
[8] https://github.com/thomasfredericks/Bounce-Arduino-Wiring/wiki
[9] http://www.leonardomiliani.com/2012/come-gestire-loverflow-di-millis/?lang=e CC-BY-NC-ND
[10] https://www.arduinoforum.de/arduino-Thread-was-heist-15UL
[11] http://www.cpp-entwicklung.de/cpplinux/cpp_main/cpp_main.html CC-BY-NC-ND
[12] http://www.cprogramming.com/reference/
[13] http://www.mathertel.de/Arduino/OneButtonLibrary.aspx
[14] https://github.com/PaulStoffregen/DS1307RTC CC BY-SA
[15] https://www.arduino.cc/en/Tutorial/PWM CC BY-SA
[16] https://datasheets.maximintegrated.com/en/ds/DS18B20.pdf
[17] https://de.wikipedia.org/wiki/I%C2%B2C
[18] https://playground.arduino.cc/Main/I2cScanner
[19] https://github.com/fdebrabander/Arduino-LiquidCrystal-I2C-library
[20] https://github.com/thefekete/LM75
[21] https://github.com/skywodd/pcf8574_arduino_library
[22] Buch „AVR-Mikrocontroller Programmierung in C“ von Heimo Gleicher, S.342
[23] https://github.com/skywodd/pcf8574_arduino_library
[24] https://www.exploreembedded.com/wiki/images/7/77/QC12864B.pdf
[25] https://github.com/olikraus/u8g2/wiki
[26] https://www.geeksforgeeks.org/using-range-switch-case-cc/
[27] https://github.com/adafruit/Adafruit-MCP23008-library
[28] http://arduino.cc/playground/Main/DHTLib
[29] https://forum.arduino.cc/index.php?topic=513120.0
[30] https://stackoverflow.com/questions/7383606/converting-an-int-or-string-to-a-char-array-on-arduino

Diese Übersicht des ARDUINO-Befehlssatzes wurde aus verschiedenen Quellen, vieler eigener Übersetzung aus dem Englischen, sowie zusätzlichen eigenen Anmerkungen, Erweiterungen, Erfahrungen und Informationen erstellt. Sie ist nicht vollständig bzw. abschließend.

Aus den oben angegebenen Quellen wurden zum Zeitpunkt der Erstellung des entsprechenden Kapitels Informationen in diesem Buch verwendet. Inwieweit diese Links aktiv sind, obliegt dem Eigentümer, ich habe keinen Einfluss darauf. Die Nennung erfolgt hier nur im Rahmen eines Quellennachweises.

Jegliche Verantwortung, Haftung oder Schadensersatz aufgrund der Benutzung dieses Werkes wird durch den Verfasser ausgeschlossen. Es kann keine Garantie auf Richtigkeit und Vollständigkeit gegeben werden. Gegebenenfalls einfach mal jemand fragen, der sich mit so was auskennt!

Hinweise, konstruktive Kritik (kein Genörgel!) nehme ich gern unter meiner E-Mail DL1AKP@web.de entgegen.

Versionsverlauf / Historie der Änderungen & Ergänzungen

Version vom 20.07.2017
- Taster mit Interrupts
- Temperatursensoren und deren Benutzung
..1wire-Sensor DS18B20 / DS18S20
..analoger Sensor MCP9700

Version vom 11.08.2017
- DOWNLOAD: angepasste Beispiele zu bestimmten Kapiteln und Libraries
- PROGMEM aktualisiert, keine Lib mehr nötig
- float-Datenkonvertierung Hinweise ergänzt
- Datentyp double ergänzt
- Datentyp short ergänzt
- LCD-Anzeigen aktualisiert, neuer Unterprunkt "Sonderzeichen, Symbole, Umlaute"
- Entprellen von Tastern mittels zusätzlicher Bauteile
- String Object toInt() ergänzt
- String Object toFloat() ergänzt
- Neues Kapitel "Zeichen- (Character-) Funktionen" eingefügt
- alle Befehle gemäß ARDUINO.cc zugefügt
- I2C LCD-Displays
- I2C Temperatursensor LM75

Version vom 05.01 2018
- I2C-Portexpander hinzugefügt
- Berichtigung vieler Rechtschreibfehler, Grammatik, Satzbau
- kleine kosmetische Verbesserungen
- Ergänzungen und Berichtigungen Zeichenfunktionen (Space und WhiteSpace)
- #ifdef #else #endif hinzugefügt
- Ergänzungen zur maximalen Eingangsspannung an Pins
- DS18B20 parasite power ergänzt
- Kapitel über +UB, Belastbarkeit der Ausgänge und Allgemeines hinzugefügt
- Kapitel über Grafikdisplays 128x64 hinzugefügt

Version vom 26.02.2018
- Berichtigung kleinerer Rechtschreibfehler
- Berichtigung Fehler in Array S.25
- bessere Schaltbilder mittels EDA

Version vom 16.04.2018
- Range switch/case eingefügt
- kleine Formatierungsfehler / Rechtschreibfehler beseitigt
- Ergänzung Portexpander MCP23008 / 23017

Version vom März 2019
- Rechtschreibfehler beseitigt
- Umwandlung in odt-Format zur Bearbeitung mit LibreOffice (keine Word-Lizenz mehr vorhanden)
- Stichwortverzeichnis entfernt, da permanent Probleme beim Aktualisieren auftraten
- Neues Kapitel Ultraschall-Sensoren HC-SR04
- Neues Kapitel Luftfeuchte-Sensoren DHTxx
- RAM sparen bei Verwendung von Ethernet hinzu gefügt

Version vom 04.01.2020
- Rückumwandlung in MS-Office, da neue Lizenz vorhanden
- Korrektur Rechtschreibfehler
- Korrektur Fehler Pullup-Widerstände an analogen Pins
- Ergänzung Watchdog-Problem bei China-Klones

Version vom 08.02.2020
- kleine Ergänzung + Klarstellung bei Strings
- Rechtschreibfehler beseitigt
- Probleme mit fehlenden Bildern korrigiert
- Quellennachweis nach hinten gebracht

Version vom 23.02.2020
- kleine Ergänzungen und Rechtschreibfehler beseitigt
- Danksagung hinzugefügt
- Klarstellung zu Aliasen bei Pins
- Erweiterte Erklärung zum Thema „Datentypen-Bezeichner“
- neuer Unterpunkt „Pegel in der Digitaltechnik“
- Formatierungen optimiert

Version vom 05.03.2020
- Rechtschreibfehler beseitigt und Formatierung optimiert
- neues Kapitel „Von delay zur Methode“

Version vom xx.xx.2020
-