c't Make:

Make:

Keine Vorkenntnisse erforderlich!

ARDUINO SPECIAL

Grundlagen

- LEDs steuern
- Eingänge abfragen
- Spannungen messen
- Sensoren einlesen
- Motoren schalten
- Mikrofon anschließen
- Fernsehausgabe
- Töne erzeugen

ANFÄNGER-PROJEKTE

Batterietester • Lärmampel
Temperaturanzeige • Synthesizer

Hardware & Software einfach erklärt

1/2016

Heise

www.make-magazin.de

Übersicht wichtiger Befehle und Funktionen

Struktur & Programmfluss

Grundsätzliche Sketch-Struktur

```
void setup() {
  // läuft nur einmal beim Start
}
void loop() {
  // wiederholt sich kontinuierlich
}
```

Kontroll-Strukturen

```
if (x < 5) { ... } else { ... }
while (x < 5) { ... }
for (int i = 0; i < 10; i++) { ... }
break;      // Schleife sofort beenden
continue;   // weiter mit nächster Iteration
switch (var) {
  case 1:
    ...
    break;
  case 2:
    ...
    break;
  default:
    ...
}
return x;   // x ist der Rückgabewert
return;     // ohne Rückgabewert
```

Funktionen

```
<ret. type> <name>(<params>) { ... }
bspws. int double(int x) {return x*2;}
```

Operatoren

Allgemein

=	Zuweisung		
+	Addieren	-	Subtrahieren
*	Multiplizieren	/	Dividieren
%	Modulo		
==	gleich	!=	ungleich
<	kleiner als	>	größer als
<=	kleiner oder gleich		
>=	größer oder gleich		
&&	logisch UND	||	logisch ODER
!	NICHT		

Kombinierte Operatoren

++	Variable um 1 erhöhen
--	erniedrigen
+=	wie a = a + b
-=	wie a = a - b
*=	wie a = a * b
/=	wie a = a / b
&=	wie a = a & b
|=	wie a = a | b

Bitoperatoren

&	bitweises UND	|	bitweises ODER
^	bitweises ODER	~	bitweises NICHT
<<	um 1 Bit nach links schieben		
>>	um 1 Bit nach rechts schieben		

Pointer

&	Referenz: Adresse des Pointers
*	Dereferenzierer: Ziel des Pointers

Eingebaute Funktionen

Pin Input/Output

Digital I/O - Pins 0-13 A0-A5

```
pinMode(pin,
  [INPUT, OUTPUT, INPUT_PULLUP])
int digitalread(pin)
digitalWrite(pin, [HIGH, LOW])
```

Analog In - pins A0-A5

```
int analogRead(pin)
analogReference(
  [DEFAULT, INTERNAL, EXTERNAL])
```

PWM Out - pins 3 5 6 9 10 11

```
analogWrite(pin, value)
```

Advanced I/O

```
tone(pin, Freq_Hz)
tone(pin, Freq_Hz, Dauer_ms)
noTone(pin)
shiftOut(dataPin, clockPin,
  [MSBFIRST, LSBFIRST], WERT)
unsigned long pulseIn(pin,
  [HIGH, LOW])
```

Zeit

```
unsigned long millis()
  // läuft nach 50 Tagen über
unsigned long micros()
  // läuft nach 70 Minuten über
delay(msec)
delayMicroseconds(usec)
```

Mathe-Funktionen

```
min(x, y)   max(x, y)   abs(x)
sin(rad)    cos(rad)    tan(rad)
sqrt(x)     pow(Basis, Exponent)
constrain(x, Minl, Maxl)
map(val, fromL, fromH, toL, toH)
```

Zufallszahlen

```
randomSeed(seed)   // long oder int
long random(max)   // 0 bis max-1
long random(min, max)
```

Bits und Bytes

```
lowByte(x)   highByte(x)
bitRead(x, bitn)
bitWrite(x, bitn, bit)
bitSet(x, bitn)
bitClear(x, bitn)
bit(bitn)   // bitn: 0=LSB 7=MSB
```

Konvertierung von Datentypen

```
char(Wert)   byte(Wert)
int(Wert)    word(Wert)
long(Wert)   float(Wert)
//int(Fließkommazahl) wandelt bspws
//Fließkommazahl in Integer-Wert um
```

Externe Interrupts

```
attachInterrupt(interrupt, func,
  [LOW, CHANGE, RISING, FALLING])
detachInterrupt(interrupt)
interrupts()
noInterrupts()
```

Variablen, Arrays und Daten

Datentypen

boolean	true	|	false
char	-128	-	127, 'a' '$' etc.
unsigned char	0	-	255
byte	0	-	255
int	-32768	-	32767
unsigned int	0	-	65535
word	0	-	65535
long	-2147483648	-	2147483647
unsigned long	0	-	4294967295
float	-3.4028e+38	-	3.4028e+38
double	wie float		
void	kein Rückgabewert/ unbestimmt		

Strings

```
char str1[8] =
  {'A','r','d','u','i','n','o','\0'};
  // manuelle \0 Null-Terminierung
char str2[8] =
  {'A','r','d','u','i','n','o'};
  // Compiler fügt Null-Terminierung hinzu
char str3[] = "Arduino";
char str4[8] = "Arduino";
```

Numerische Konstanten

123	dezimal
0b01111011	binär
0173	oktal (Basis 8)
0x7B	hexadezimal (Basis 16)
123U	erzwinge unsigned
123L	erzwinge long
123UL	erzwinge unsigned long
123.0	erzwinge floating point
1.23e6	1.23*10^6 = 1230000

Eigenschaften von Variablen

static	Variable bleibt über Aufrufe erhalten
volatile	Variable im RAM statt Register speichern
const	Konstante, nur lesbar
PROGMEM	im Flash ablegen

Arrays

```
int myPins[] = {2, 4, 8, 3, 6};
int myInts[6];     // Array mit 6 ints
myInts[0] = 42;  // Zuweisung der
                    ersten Stelle
myInts[6] = 12;  // Fehler, da 6. Variable
                    // den 5. Platz belegt
```

Bibliotheken

Serial – serielle Kommunikation mit PC oder via RX/TX

```
begin(long Baudrate)   // bis 115200
end()
int available()  // Anzahl empfangener Bytes
int read()       // -1 wenn keine Daten
int peek()       // Lesen ohne den Puffer
                    zu löschen
flush()   // Puffer löschen
print(data)     println(data)
write(byte)     write(char * string)
write(byte * data, size)
SerialEvent()   // wenn Daten fertig
```

SoftwareSerial.h – serielle Kommunikation über beliebige Pins

```
SoftwareSerial(rxPin, txPin)
begin(long Baudrate)   // bis 115200
listen()         // immer nur eine serielle
isListening()    // kann empfangen
read, peek, print, println, write
```

EEPROM.h – Zugriff auf nicht flüchtigen Speicher

```
byte read(Addr)
write(addr, byte)
EEPROM[index]
```

Servo.h – Servo-Motoren steuern

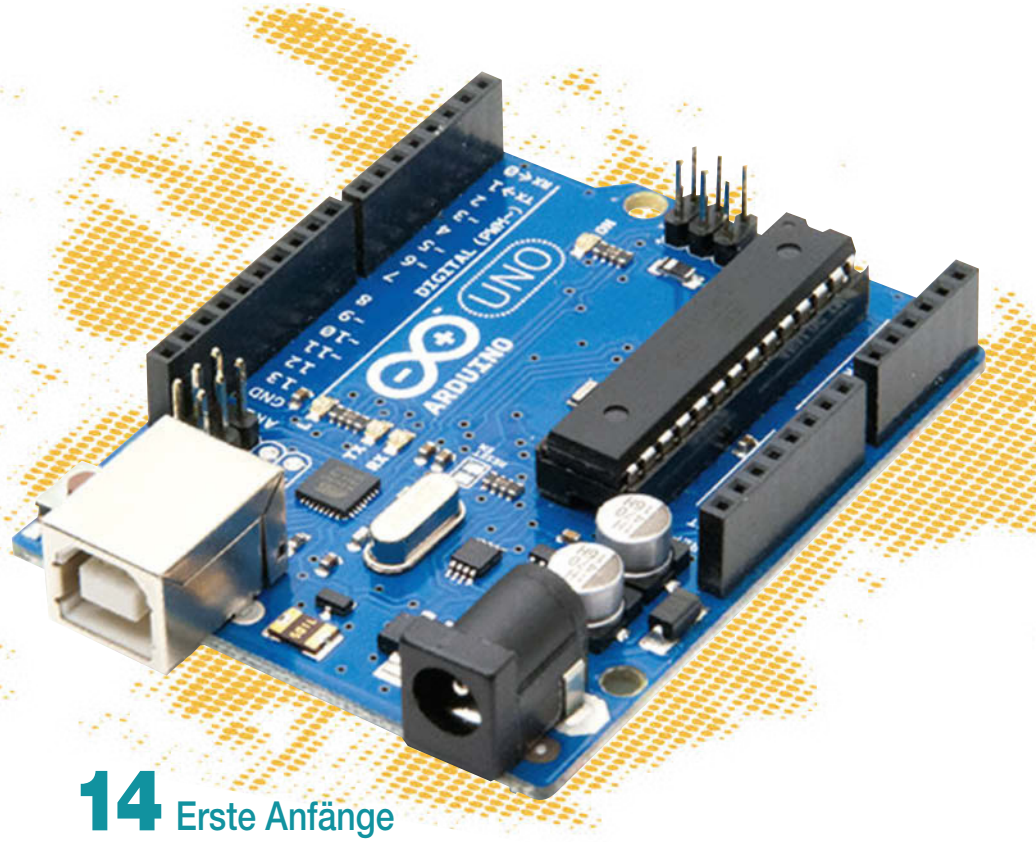
```
attach(pin, [min_uS, max_uS])
write(Winkel)    // 0 bis 180
writeMicroseconds(uS)
  // 1000-2000; 1500 ist Mitte
int read()       // 0 bis 180
bool attached()
detach()
```

Wire.h – I²C-Kommunikation

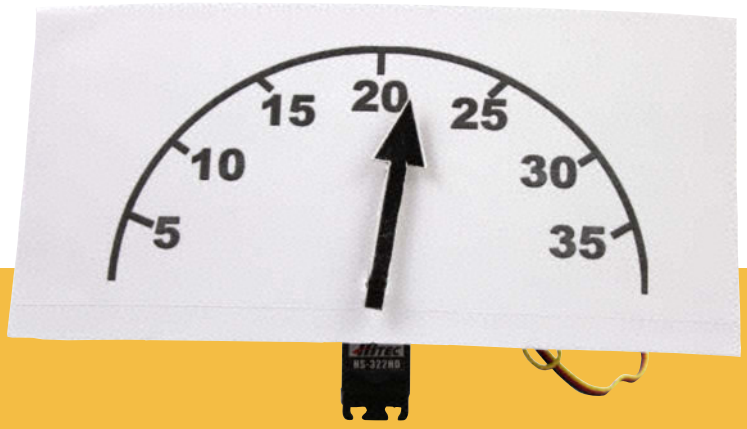
```
begin()
begin(Addr)   // Slave ansprechen
requestFrom(Addr, count)
beginTransmission(Addr)   // Schritt 1
send(byte)                // Schritt 2
send(char * string)
send(byte * data, size)
endTransmission()         // Schritt 3
int available()   // verfügbare Bytes
byte receive()    // nächstes Byte lesen
onReceive(handler)
onRequest(handler)
```

Weitere „Cheat Sheets" finden Sie unter ▶ overapi.com

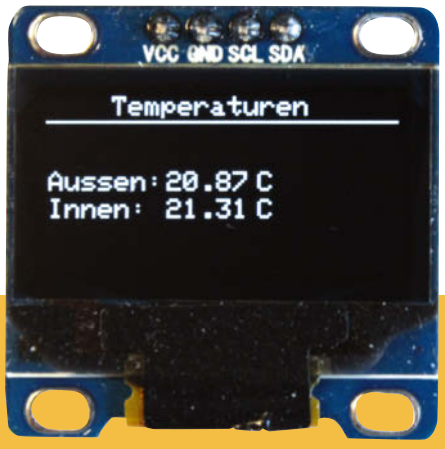
Inhalt



14 Erste Anfänge
Beispiele, die ohne Zusatzhardware sofort funktionieren.



29 Motoren steuern
Mit Motoren kommt Bewegung ins Spiel, um etwas anzutreiben oder einen Zeiger zu bewegen.



42 Mäusekino
Mit einem kleinen Display lässt sich der Arduino als mobiles Messgerät einsetzen.

Wozu eigentlich?

Mikrocontroller stecken hinter der Fassade vieler elektronischer Produkte. Mit dem Arduino kann man spielerisch Geräte nach eigenen Ideen bauen.

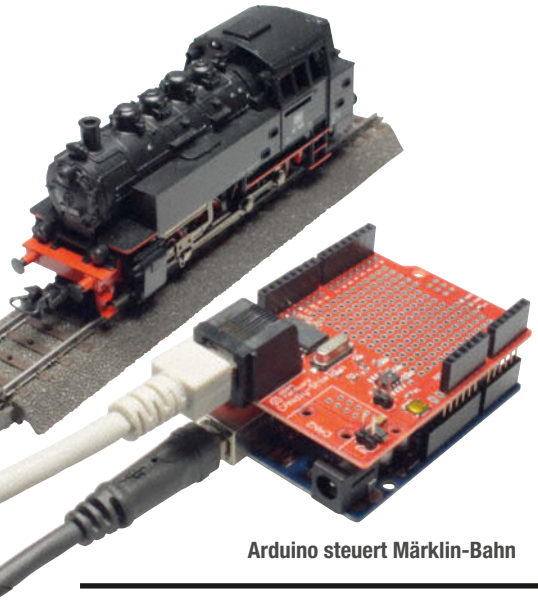
von Daniel Bachfeld

Arducopter

Jeder hat ihn, aber nur wenige wissen es: Mikrocontroller stecken überall drin. Ob im Auto, DSL-Router, Smartphone oder Tablet: Mikroprozessoren helfen Dinge automatisiert zu schalten, zu steuern, zu regeln und zu kontrollieren. Auch im nicht kommerziellen Bereich lassen sich Mikrocontroller kreativ und praktisch einsetzen, etwa um regelmäßig wiederkehrende Aufgaben erledigen zu lassen, statischen Dingen Leben einzuhauchen oder um etwa die Umwelt mit Sensoren zu erfassen und die Daten anzuzeigen – und das alles ohne einen vergleichsweise teuren und klobigen PC.

Dank des großen und günstigen Angebots an elektronischen Komponenten kann der Privatmann leicht Geräte entwickeln, die es so bislang gar nicht am Markt gibt oder sie passgenau auf seine Anforderungen zuschneidern. Aquarienfutterautomat, Gartenbewässerungssteuerung, Luftgütemessgerät, Füllstandsüberwachung, Abstandswarner, Smartwatch, Drohnen, Roboter, Lichtsteuerung, Heimautomation: Je nach Gusto übernimmt der Mikrocontroller verschiedene Aufgaben in Haus, Hof und Hobby.

Tetris



Arduino steuert Märklin-Bahn

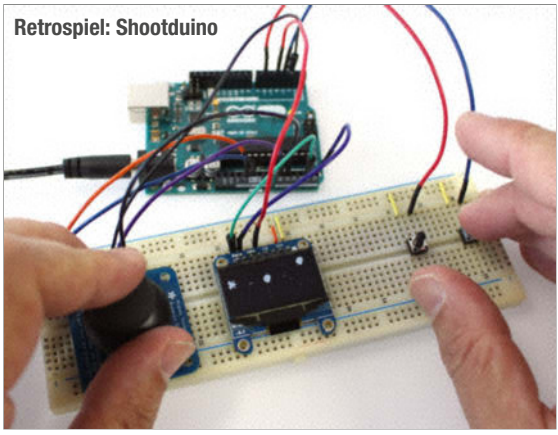
Mit dem Arduino und der auch für Nicht-Informatiker leicht verständlichen Software-entwicklungsumgebung findet der Anfänger einen schnellen Einstieg in die Programmierung von Mikrocontrollern. Und obwohl der Arduino und seine Software ursprünglich für eher IT-ferne Anwender wie Künstler, Designer und Medienwissenschaftler entwickelt wurde, nutzen ihn mittlerweile auch professionelle Hardware-Entwickler für den Bau von Prototypen. Durch sein Stecksystem lässt sich der Arduino nämlich ohne Löten spielend leicht und schnell mit weiterer Hardware – sogenannten Shields – erweitern.

Schnell, einfach und günstig statt langwierig, kompliziert und teuer: Durch dieses Grundprinzip hat sich eine große Internet-Community um den Arduino geschart, die sich gegenseitig inspiriert und unter die Arme greift. Arduino gilt derzeit als erste Wahl, wenn es um die Umsetzung eigener Projekte geht. Deshalb bieten zahlreiche Hersteller Zusatzprodukte an, die sowohl hardware- als auch softwarekompatibel sind, mitunter aber mehr Rechenleistung liefern. Mit Intels briefmarkengroßem Mikrocontroller Edison beispielsweise können Arduino-Programme die Rechenleistung von Atom-Prozessoren nutzen und sogar Funktionen wie WLAN und Bluetooth nutzen, die der Original-Arduino nur durch zusätzliche Shields bietet.

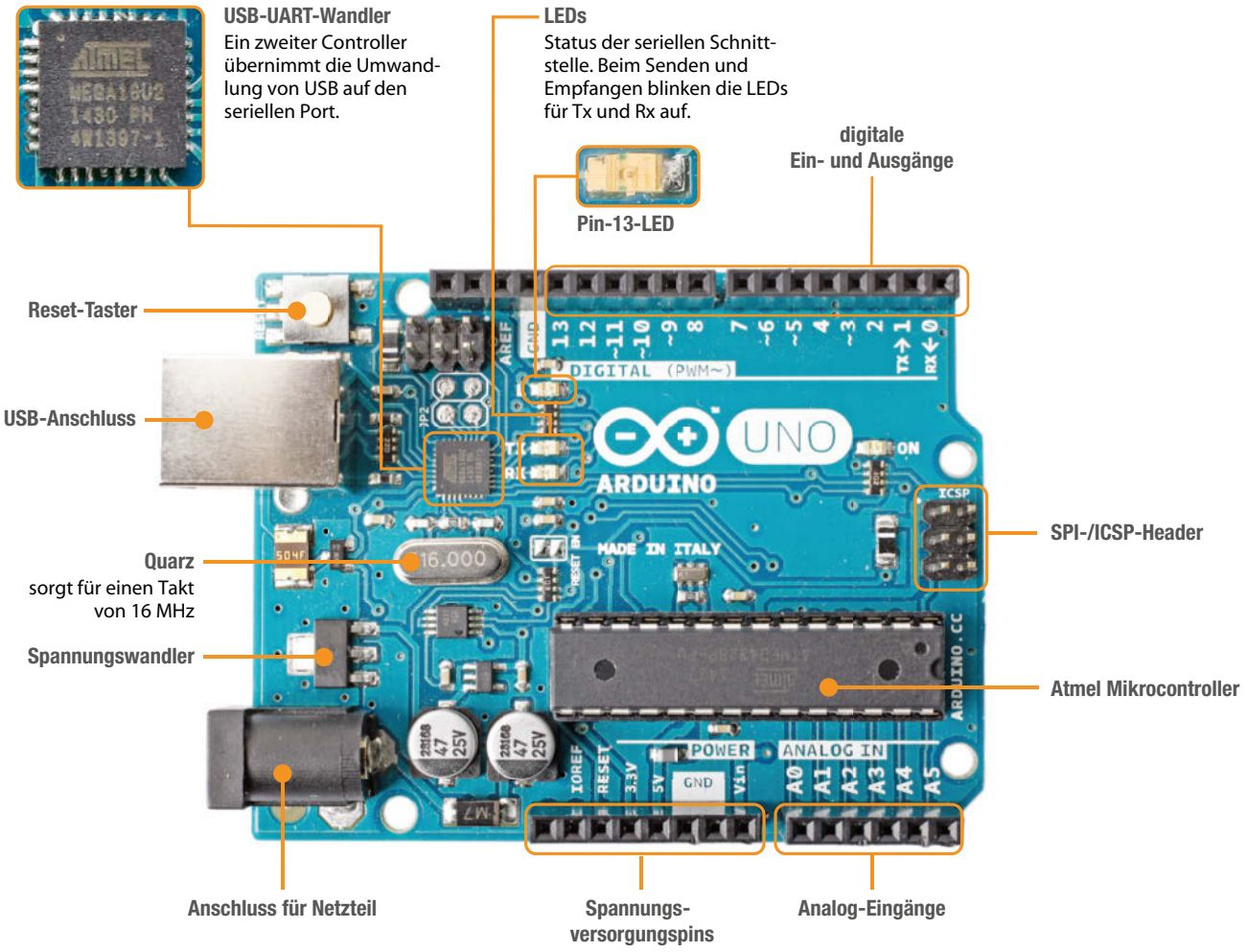
Mit diesem Make-Special „Arduino“ wollen wir einen leichten, kurzweiligen, aber langfristig fesselnden Einstieg in den Arduino und seine Möglichkeiten bieten. Alles, was Sie für die ersten Schritte benötigen, sind ein USB-Kabel und ein Stück Draht – zur Not tut es auch eine auseinandergebogene Büroklammer. Zur Programmierung steht die kostenlose Arduino-IDE unter arduino.cc im Internet bereit. Und nun viel Spaß!

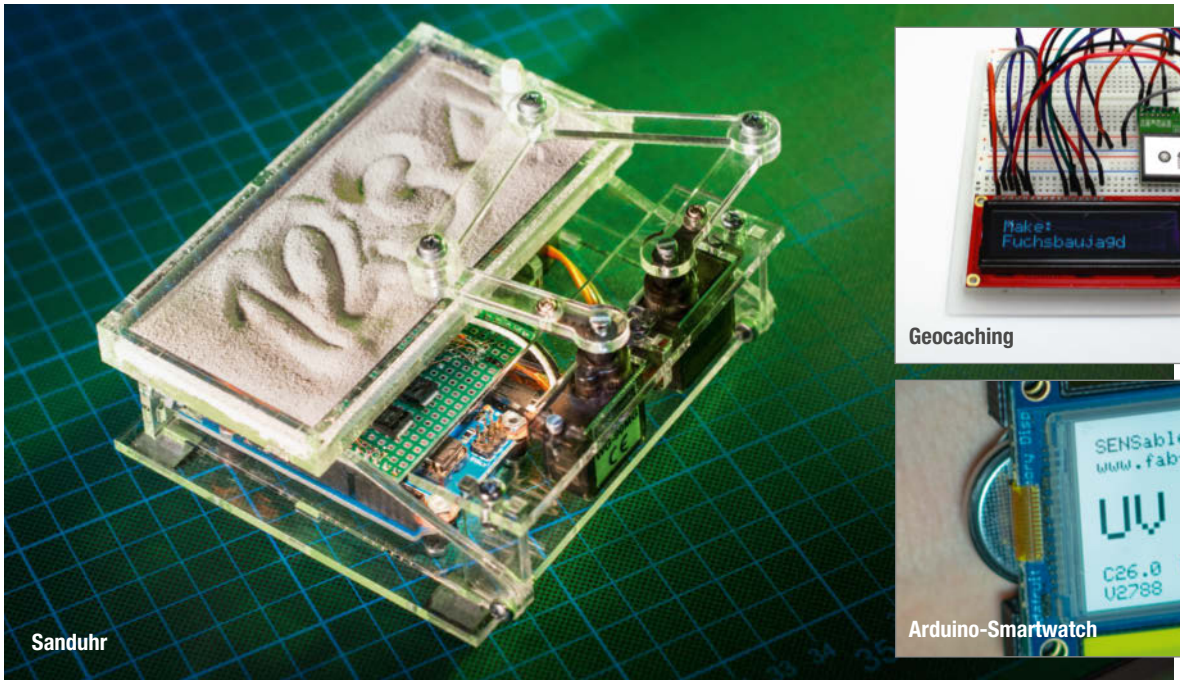
Links und Foren
make-magazin.de/x82g

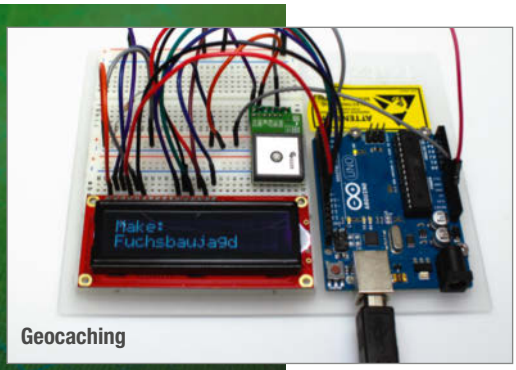
Retrospiel: Shootduino

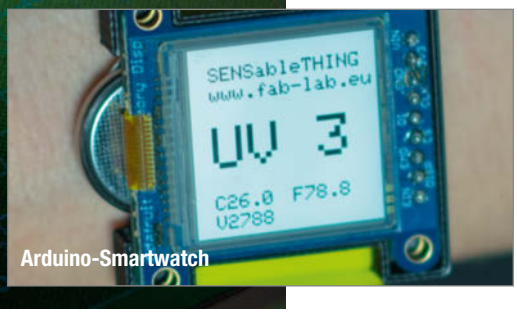


Bestandteile eines Arduino UNO (Revision 3)









Die Pins

Eine Übersicht über die Funktion der Arduino-Pins

Der Arduino hat intern mehr I/O-Funktionen zu bieten, als seine Beinchen (Pins) außen zur Verfügung stellen können. Wie schafft er es dennoch, die Funktionen extern anbieten zu können? Mit einer ausgefeilten Logik kann man festlegen, welche interne Funktion mit der Außenwelt in Verbindung steht. So kann ein Pin mal ein analoger Eingang oder Vergleicher, mal ein digitaler Ausgang, mal ein Generator für Rechtecksignale oder in Kombination mit weiteren Pins sogar ein sogenannter Bus zum Datenaustausch mit anderen elektronischen Systemen sein.

Durch die Flexiblität des verwendeten Atmel-Prozessors lässt sich der Arduino für verschiedene Aufgaben einsetzen. Es genügt, die Verdrahtung der alten Komponenten abzunehmen, neue Komponenten anzuschließen und einen neuen Sketch zu installeren. Hier erklären wir die Bedeutung der Pins des Arduino, zuerst die mit Sonderfunktionen:

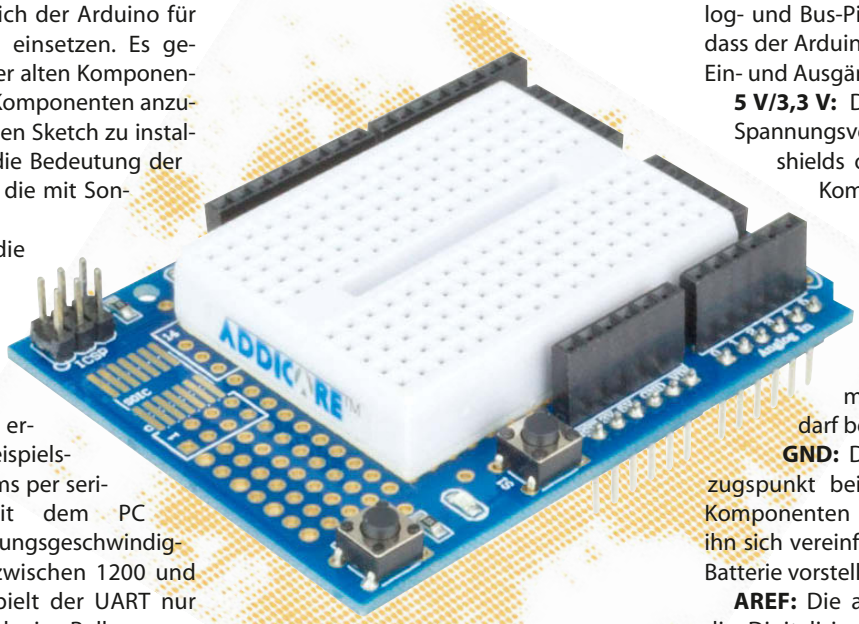
RX, TX: Das sind die Sende- und Empfangspins der seriellen Schnittstelle (UART, Universal Asynchronous Receiver Transmitter). Bevor der USB-Bus erfunden wurde, waren beispielsweise Mäuse und Modems per serieller Schnittstelle mit dem PC verbunden. Die Übertragungsgeschwindigkeit liegt in der Regel zwischen 1200 und 115 200 Baud. Heute spielt der UART nur noch im Entwicklerbereich eine Rolle.

SCL, SDA: Die zwei Pins gehören zum I²C-Bus (Inter IC Communication), über den der Arduino Daten mit Sensoren und anderer Elektronik austauschen kann. Im Unterschied zum UART gibt es nur eine Leitung für die Hin- und Rückrichtung (SDA, Serial Data), dank eines einheitlichen Protokolls und eines Taktsignals (SCL, Serial Clock) gibt es aber keine Konflikte auf dem Bus.

SPI: Das Serial Peripheral Interface ist ein synchroner serieller Bus, mit Hin-, Rück- und Taktleitung. Beim Arduino dient er dem Datenaustausch mit Sensoren und Steuermodulen, aber auch der Programmierung seines Flash-Speichers mit einem speziellen Gerät. Standardmäßig arbeitet der SPI auf dem Arduino mit 4 MHz.

Digital (PWM~): Die digitalen Pins lassen sich jeweils als Ein- oder Ausgang programmieren. Arbeiten sie als Eingang, kann man dem Arduino mit einer Spannung von 0 V eine logische 0 und mit 5 V eine logische 1 von außen signalisieren, auf die er reagieren kann. Ist ein Pin als Ausgang konfiguriert, kann er per Programm seiner Umwelt oder weiterer Elektronik eine logische 1 oder eine logische 0 signalisieren.

Daneben kann man die Ausgangsspannung von 5 V auch direkt benutzen, um eine Leuchtdiode (LED) zum Leuchten zu bringen oder ein Relais zu schalten, das wiederum ein anderes Gerät mit Batterieversorgung oder gar mit Netzspannung anschaltet. Zu beachten ist, dass ein Ausgang als Dauerlast nur einen Strom von 20 Milliampere liefert, kurzzeitig auch bis zu 40. Darüber droht ein dauerhafter Schaden des Pins. Um das zu verhindern, muss man den Strom mit einem Widerstand begrenzen. Wie man das macht, erfahren Sie im hinteren Teil des Hefts.



Auf Prototypingshields lassen sich alle Pins des Arduino für eigene Elektronik anzapfen.

Alternativ können einige der digitalen Pins (zu erkennen an der Tilde ~ vor der Nummer auf dem Board) ein sogenanntes PWM-Signal ausgeben. Neben der Leistungssteuerung von LEDs benötigt man dieses Signal in erster Linie zur Ansteuerung von Servo-Motoren, wie man sie im Modellbau einsetzt.

Analog In (A0–A5): Da die Welt eben nicht nur schwarz (5 V) und weiß (0 V) ist, benötigt man noch eine Möglichkeit, Spannungen irgendwo dazwischen zu messen. Ein Analog-Digital-Wander wandelt, wie beim Mikrofon-Eingang am PC, analoge Signale in digitale Werte zwischen 0 und 1023 (10 Bit). Der UNO hat eigentlich nur einen einzigen internen Wandler, mit einem eingebauten Umschalter kann er jedoch die einzelnen externen Eingänge auf den Eingang des A/D-Wandlers legen.

Grundsätzlich lassen sich auch alle Analog- und Bus-Pins als I/O-Pins festlegen, sodass der Arduino UNO insgesamt 20 digitale Ein- und Ausgänge befehligen kann.

5 V/3,3 V: Die beiden Pins dienen der Spannungsversorgung von Erweiterungsshields oder anderen elektronischen Komponenten.

VIn: Hier liegt die Spannung an, mit der der Arduino bei Verwendung eines externen Netzteils versorgt wird. Shields mit Motoren und anderen Verbrauchern mit höherem Spannungsbedarf benötigen diesen Pin.

GND: Das ist der gemeinsame Bezugspunkt beim Zusammenschalten von Komponenten mit dem Arduino. Man kann ihn sich vereinfacht wie den Minuspol einer Batterie vorstellen.

AREF: Die analogen Eingänge nehmen die Digitalisierung in Bezug auf eine bestimmte Spannung vor, standardmäßig 5 V. Man kann aber auch durch eine externe Spannungsreferenz einen anderen Wert vorgeben. Liefert beispielsweise ein Sensor nur eine maximale Spannung von 1 V, würde man bei AREF = 5 V nur ein Fünftel des Wertebereichs (0–205) nutzen. Sinnvoller wäre, AREF mit einer Referenzspannung von 1 V zu speisen.

IOREF: Damit signalisiert der Arduino einem Shield, mit welchem Spannungspegel er arbeitet. In der Regel spielt dieser Pin keine Rolle.

Reset: Neben dem manuellen Druck auf den Resetkopf kann auch eine externe Schaltung einen Neustart des Arduino auslösen.

—dab

Interna

Ein kurzer Überblick über den internen Aufbau hilft, die Arbeitsweise der Programme besser zu verstehen.

Der Kern des Arduino ist der Mikrocontroller ATmega328 des Herstellers Atmel: Das Blockschaltbild des ATmega328 ist recht umfangreich, denn in dem kleinen IC sind viele Funktionen untergebracht, die früher beispielsweise – abgesehen von der Videoausgabe – als einzelne Komponenten auf den Platinen der Heimcomputer oder auf den Mainboards der ersten PCs verbaut waren. Wenn man sich die einzelnen Blöcke in Ruhe anschaut, ergeben sich aber recht einfach die Zusammenhänge.

Zentrales System ist der 8-Bit-Prozessor (**CPU**), dem ein Arbeitsspeicher (**SRAM**) für das Ablegen von temporären Daten und ein Programmspeicher (**Flash**) zur Seite gestellt sind. Die 32 KByte Größe des Flash und die 2 KByte des RAM sind zwar im Vergleich zu einem modernen PC lachhaft klein, allerdings wollen wir ja auch keine grafische Benutzeroberfläche mit Browser, Mail und anderem Schnickschnack auf dem Arduino laufen lassen.

Aus dem Flash liest die CPU die einzelnen Befehle des Programms (Sketch) nacheinander ein und führt sie aus. Ein sogenannter Programm Counter hilft ihr, sich die Speicheradresse zu merken, ab der der nächste Befehl dran ist. Wenn die CPU sich zwischendurch mal bestimmte Werte merken muss, legt sie diese in ihren internen Registern oder im RAM ab – der Flash kann nämlich während des Betriebs nicht so ohne Weiteres Daten speichern. Während das RAM beim Abschalten alle Daten verliert, behält der Flash jedoch sein Programm fest gespeichert wie ein USB-Stick. Um trotzdem auch mal während eines Programmlaufs erzeugte Daten über das Abschalten hinwegzuretten, dient der **EEPROM**. Das laufende Programm übergibt ihm Daten für die feste Speicherung, auf die es später nach jedem Anschalten wieder zugreifen kann.

Dem aufmerksamen Leser wird an dieser Stelle der fundamentale Unterschied zum PC aufgefallen sein: Der PC kopiert seine Programme vor der Ausführung immer in seinen Arbeitsspeicher, um anschließend von dort die Befehle einzulesen. Dieses Konzept nennt sich Von-Neumann-Architektur, benannt nach dem Mathematiker John von Neumann. Das Konzept der getrennten Speicher wie beim ATmega nennt sich hingegen Harvard-Architektur.

Durch die Zuschaltung von Port-Bausteinen (**PORT A, B, C**) wird der Mikroprozessor zum Mikrocontroller – weil er über die Ports nun seine Umwelt kontrollieren kann. Neben den bereits erwähnten Schnittstellen **USART**, **SPI** und **I²C** (der heißt aus lizenzrechtlichen Gründen bei Atmel allerdings Two Wire Interface, kurz **TWI**) gibt es noch mehrere Timer und Counter (**T/C 0, 1, 2**) im ATmega. Sie helfen der CPU, Zeitspannen zu messen, etwa um Ausgänge mikrosekundengenau ein- beziehungsweise auszuschalten oder die Dauer extern anliegender Signale zu messen.

Um die CPU gruppieren sich noch weitere Bausteine, die die zuverlässige Arbeit gewährleisten sollen. Bleibt ein Programm bei seinem Ablauf aus irgendeinem Grund hängen – viele kennen das bereits vom PC – kann der **Watchdog** den Controller neu starten und somit wieder auf die Spur bringen. Einen ähnlichen Zweck hat die eingebaute Überwachung der Spannungsversorgung **Power Supervision**: Sinkt die Spannung unter 4 V, so kann dies die Logik des Controllers durcheinanderbringen, die anschließend unter Umständen mit falschen Daten weiterrechnet. Sicherheitshalber resetted die Überwachung den Controller – und zwar so lange, bis die Versorgungsspannung wieder auf einen für den zuverlässigen Betrieb geeigneten Wert gestiegen ist. *—dab*

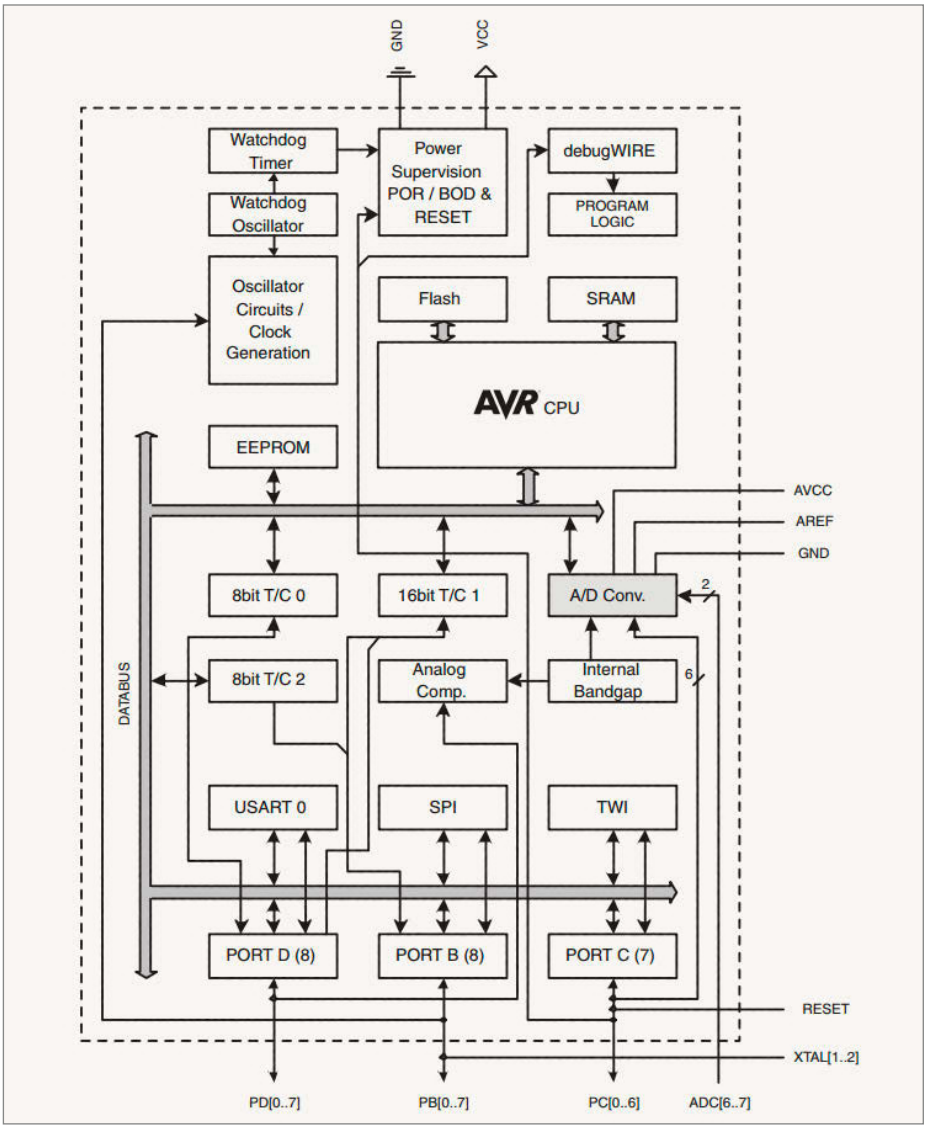


Bild: Atmel

Software

Ohne Software läuft keine Hardware. Die Arduino-IDE nimmt dem Anwender aber alle komplizierten Arbeiten ab.

Links und Foren
make-magazin.de/xbrc

von Daniel Bachfeld

Zum Konzept von Arduino gehört eine leicht zu bedienende Entwicklungsumgebung (Integrated Development Environment, IDE), die viele fertige Softwaremodule (Bibliotheken, Libraries) zur Lösung diverser Probleme und komplette Programmbeispiele für spannende Projekte mitbringt, die sich leicht für eigene Ideen einsetzen beziehungsweise anpassen lassen. Die IDE selbst ist an das Schwesterprojekt Processing zur PC-Programmierung für Einsteiger angelehnt und in der Sprache Java programmiert. Man muss aber kein Java beherrschen oder programmieren können – man kommt mit Java gar nicht in Berührung.

Bei der Programmiersprache für den Arduino handelt es sich genau genommen um C++, was man aber jedoch nur bei den grundlegenden Befehlen wie Schleifen zur Wiederholung von Befehlen, bedingten Anweisungen und Verzweigungen sowie der Verarbeitung von Variablen merkt. Die meisten anderen Befehle sind eigentlich sogenannte Methoden, die dank der selbsterklärenden Benennung wie Befehle aussehen. Die Anweisung zur Konfiguration eines Portpins lautet beispielsweise `pinMode(2, OUTPUT);` Pin 2 wird als Ausgang eingestellt – war doch leicht zu verstehen, oder? Da viele wichtige Funktionen bereits verfügbar sind und zudem verständlich formuliert sind, muss man kein Informatikstudium mehr absolviert haben, um Programme – die beim Arduino Sketches heißen – zu entwickeln. Zugegeben: Ein paar Englischkenntnisse muss man schon noch mitbringen, aber die halten sich in Grenzen. Eine Übersicht der wichtigsten Befehle und Funktionen finden Sie auf der vorderen inneren Umschlagseite dieses Heftes.

Wie wird nun aus dem Sketch (auch Quell- oder Sourcecode genannt) eine für den Arduino verständliche Liste von Befehlen, da er doch nur Maschinensprache versteht? Ein Übersetzungsprogramm, sprich Compiler, macht aus der für Menschen verständlichen Hochsprache zunächst sogenannten Assembler-Code und daraus dann Maschinencode. Das geschieht, wenn man in der IDE den Button „Verifizieren“ klickt. Allerdings ist der Compiler leider relativ streng, was die Einhaltung der Rechtschreibung und den Aufbau eines Sketches angeht. Ein einziger Vertipper, ein vergessenes Leerzeichen oder Semikolon am Ende einer Befehlszeile lassen die Übersetzung scheitern. Glücklicherweise verrät der Compiler, über welchen Fehler er gestolpert ist und gibt dazu im Statusfenster Hinweise auf die betroffene Zeile für die Korrektur.

Ist der Übersetzungsvorgang erfolgreich durchgelaufen, kann man das erzeugte „Kompilat“ (Binary) über die USB-Schnittstelle auf den Arduino laden und laufen lassen. Was sich so einfach anhört, ist im Hintergrund mit relativ viel Aufwand verbunden: Ein rudimentäres vorinstalliertes Bootloader-Programm auf dem Arduino nimmt die Daten vom PC über eine virtuelle serielle Schnittstelle entgegen und legt sie im Flash des ATmega ab. Vor der Einführung des Arduino waren noch spezielle Programmiergeräte in Kombination mit komplizierter Programmiersoftware zum Flashen nötig.

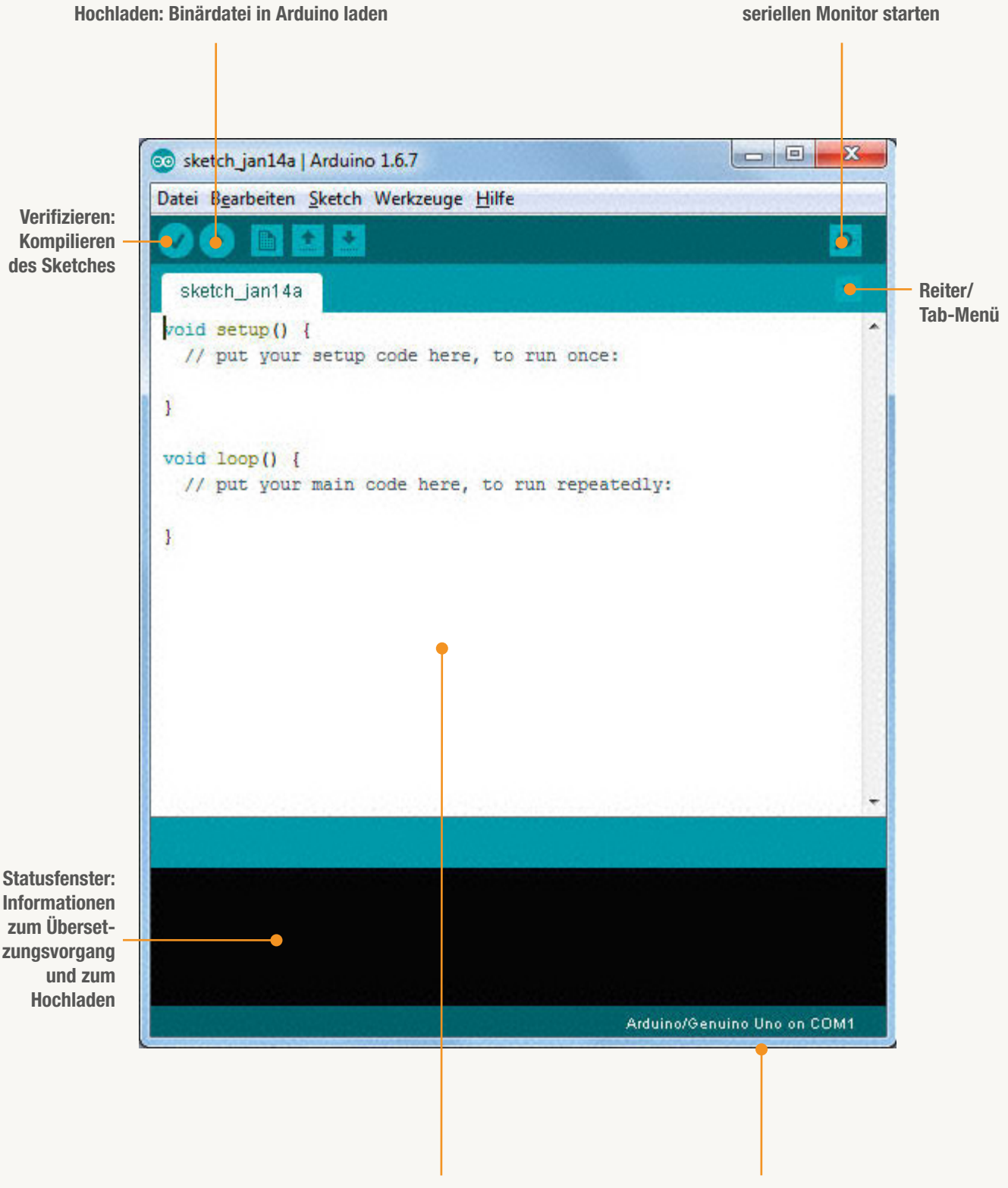
Die IDE meldet, wenn das Programm vollständig übertragen ist. Anschließend startet der Arduino das nun fest im Flash abgelegte Programm automatisch. Prinzipiell könnte man die USB-Verbindung zum PC nun trennen. Allerdings benötigt der Arduino weiterhin eine Spannungsversorgung. Möchte man den Arduino ohne PC betreiben, bieten sich Netzteile mit USB-Anschluss oder mobile Powerbanks mit USB-Ports an.

Zudem kann man die USB-Schnittstelle nicht nur zum Hochladen des Binary benutzen. Nach dem Start des Arduino-Sketches steht sie auch für die Kommunikation zwischen Arduino und PC zur Verfügung, beispielsweise um Daten auszutauschen oder Befehle zur Steuerung zu übermitteln. Entwickler nutzen das serielle Monitor-Programm der IDE auch gerne, um den richtigen Ablauf der Sketches zu kontrollieren.

ARDUINO GEGEN ARDUINO

Mitte 2015 haben sich die ursprünglichen Initiatoren des Arduino-Projekts in zwei Firmen aufgespalten, die sich auch vor Gericht beharken. Seitdem gibt es zwei offizielle Arduino-Webseiten: die schon länger bekannte arduino.cc von Arduino LLC, dem US-amerikanischen Ableger, sowie arduino.org, betrieben von Arduino S.R.L. aus Italien. Auf beiden Webseiten gibt es die Arduino-IDE zum Download, allerdings nicht in derselben Version: Während die Entwicklungsumgebung der Amerikaner bei Redaktionsschluss bei 1.6.7 stand, hat arduino.org die Versionsnummer in seinen eigenen Branch inzwischen auf Version 1.7.8 gehievt und entwickelt parallel eine neue Programmierumgebung namens Arduino Studio.

Während die Mikrocontroller-Boards von Arduino LLC in den USA weiterhin unter dem Namen Arduino vertrieben werden, heißen sie im Rest der Welt aus namensrechtlichen Gründen Genuino. Das mit diesem Heft verkaufte Board ist ein Arduino von Arduino S.R.L aus Italien. Trotz verschiedener Namen: Ein Arduino Uno und Genuino Uno sind gleich aufgebaut. Deshalb ist es auch grundsätzlich egal, ob man die Software aus Italien oder den USA benutzt. Allerdings steht hinter der US-Version eine größere Community, was dazu führt, dass verbesserte Funktionen und Bibliotheken früher zur Verfügung stehen. Wir setzen deshalb in diesem Heft auf die IDE von **arduino.cc**, weil sie beispielsweise eine grandios einfach zu bedienende Funktion zum Hinzufügen neuer Bibliotheken enthält.

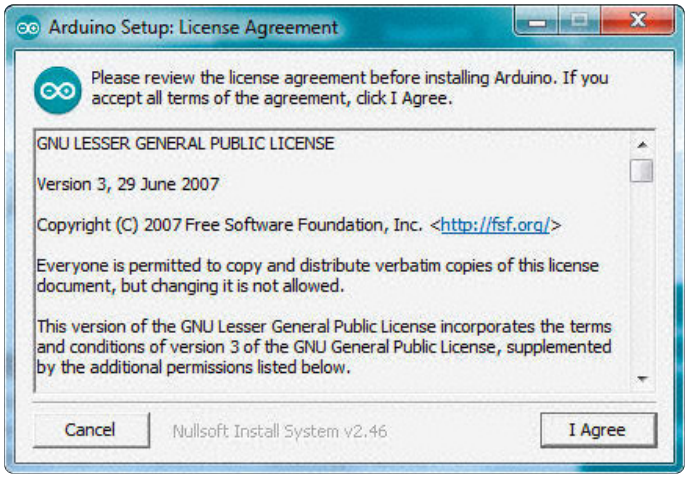

Hochladen: Binärdatei in Arduino laden
seriellen Monitor starten
Verifizieren: Kompilieren des Sketches
Reiter/ Tab-Menü
sketch_jan14a | Arduino 1.6.7
Datei Bearbeiten Sketch Werkzeuge Hilfe
sketch_jan14a
void setup() {
// put your setup code here, to run once:
}
void loop() {
// put your main code here, to run repeatedly:
}
Statusfenster: Informationen zum Übersetzungsvorgang und zum Hochladen
Arduino/Genuino Uno on COM1
Programmierfenster: Hier gibt man seine Sketches ein oder kopiert sie per Copy & Paste hinein
eingestelltes Board und serielle Schnittstelle

Installation

Die wichtigsten Schritte zur Installation und Konfiguration der Arduino IDE

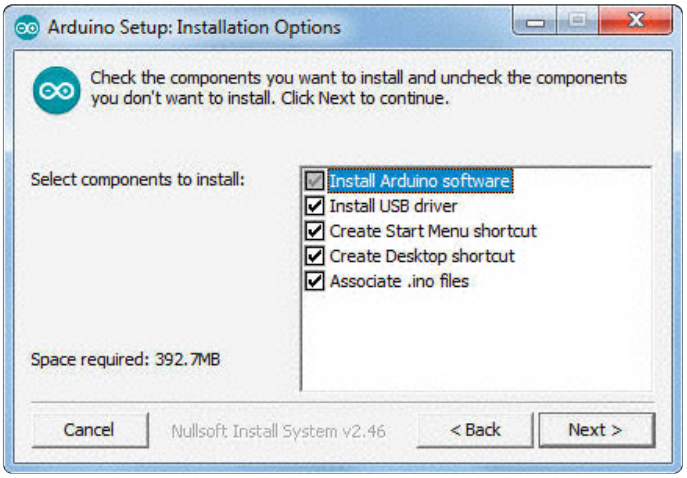
1 Installer herunterladen

Auf der Webseite **www.arduino.cc** finden sich unter dem Reiter „Download“ die aktuellen Versionen für Windows, Mac und Linux. Laden Sie bitte den „Windows Installer“ auf Ihren Windows-PC.



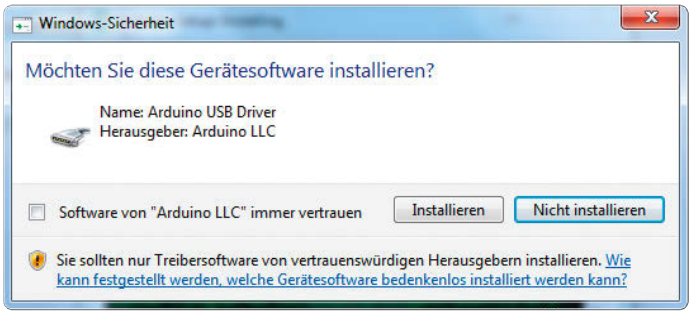
2 Installer starten

Die Arduino-IDE ist Open Source und steht unter der GLGPL. Sie dürfen die Software so oft kopieren und weitergeben wie Sie wollen, nur verändern dürfen Sie sie nicht. Mit „I Agree“ stimmen Sie dem zu.



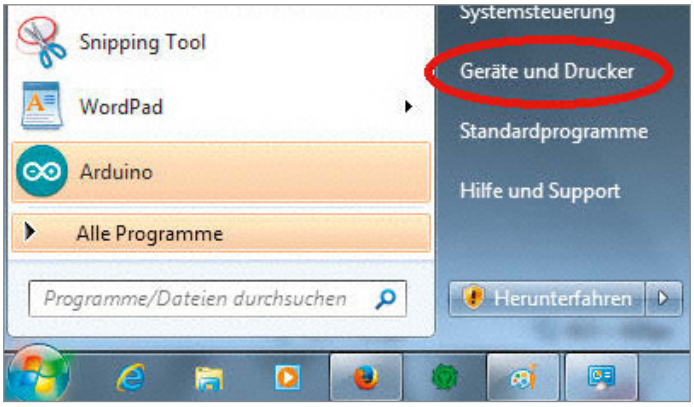
3 Installationsoptionen auswählen

Der Software-Installer nimmt einige Veränderungen auf dem PC vor: Er installiert die IDE sowie diverse USB-Treiber. Zudem richtet er die Verknüpfung von Arduino-Sketches mit der Endung .ino ein.



4 Treiberinstallation erlauben

Der Installer richtet die USB-Treiber für diverse Arduino-Modelle ein. Damit kann der PC mit dem Arduino wie über eine serielle Schnittstelle kommunizieren. Das vereinfacht die Programmierung und die Einbindung in andere Windows-Anwendungen erheblich. Trotz der Querelen zwischen Arduino LLC und Arduino S.R.L. wird auch der Treiber der Konkurrenz mitinstalliert.

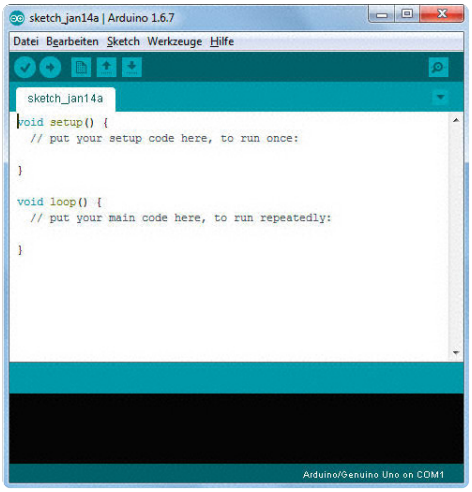


5 COM-Ports aufrufen

Um die Arduino-IDE später richtig zu konfigurieren, muss man die serielle Schnittstelle herausfinden, die Windows dem Arduino zuordnet. Dazu schließt man nach Abschluss der Installation den Arduino an den PC an und öffnet das Windows-Startfenster und klickt auf „Geräte und Drucker“.

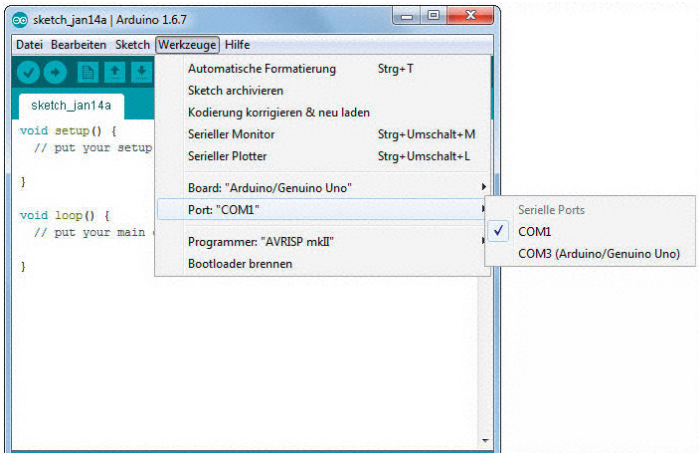
6 Arduino-COM-Port finden

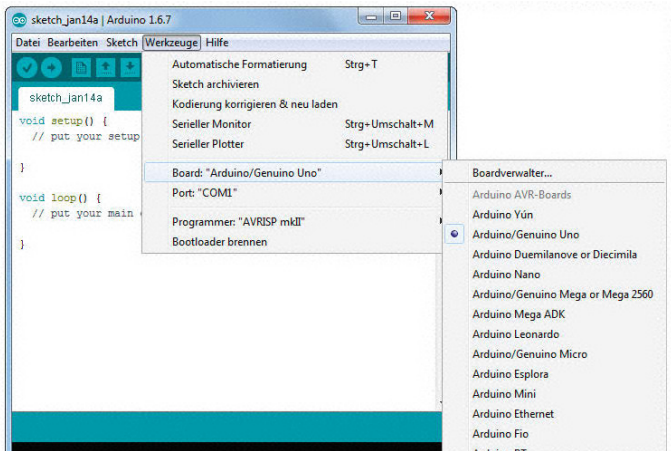
In der Übersicht findet sich ganz unten der Arduino Uno mit der Angabe der seriellen Schnittstelle COM3. Die Angaben können je nach System erheblich abweichen und bei vielen durch andere Geräte eingetragene COM-Ports auch mal zweistellig sein. Unter Umständen ändert sich auch die einmal ermittelte Nummer des Ports, wenn man andere Geräte entfernt beziehungsweise hinzufügt.



7 IDE starten

Starten Sie nun die IDE durch den Klick auf den Eintrag im Start-Menü.



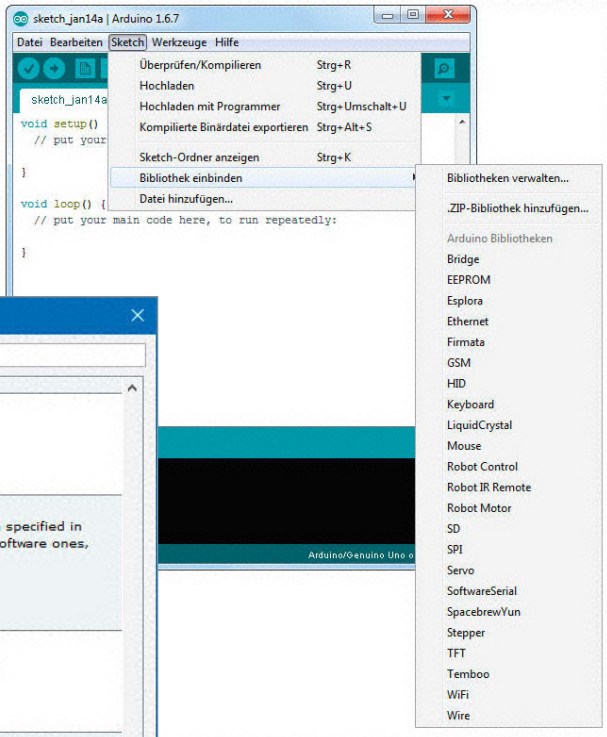


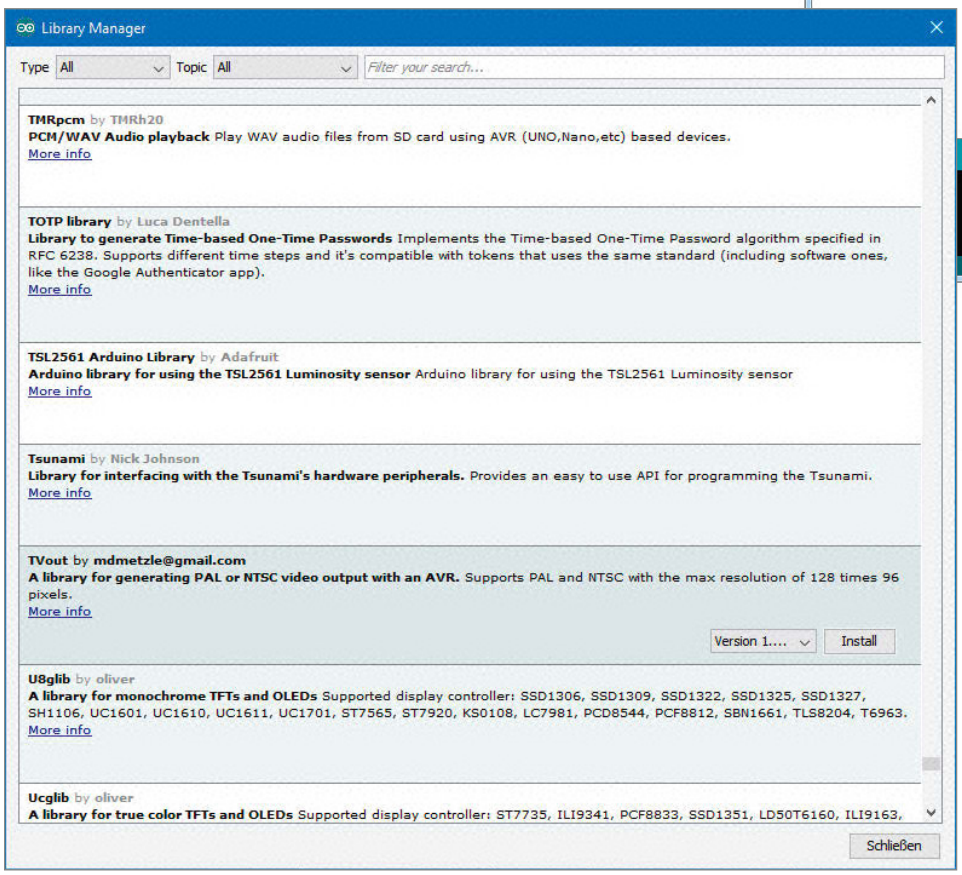
8 Schnittstelle und Arduino-Version konfigurieren

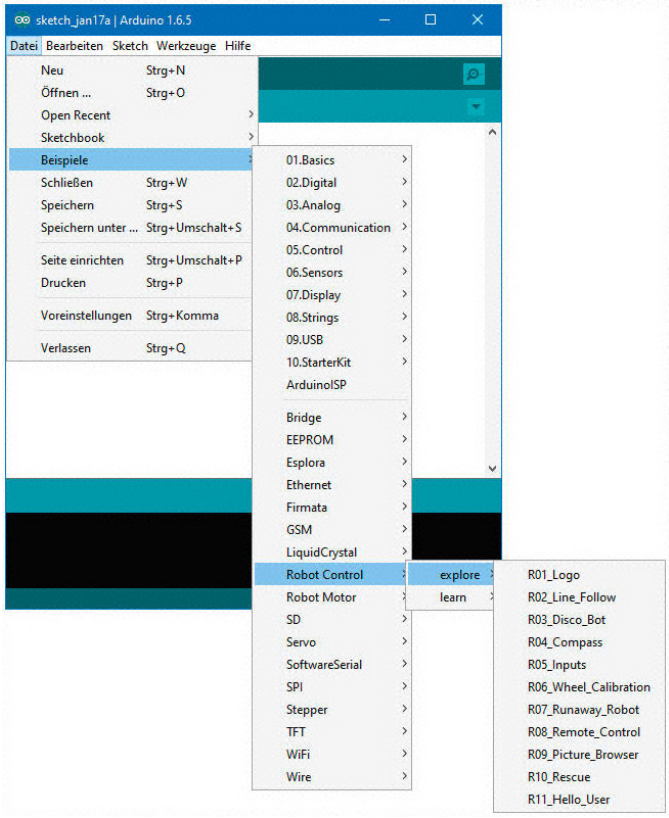
Unter dem Punkt „Werkzeuge/Port“ müssen Sie den oben ermittelten COM-Port auswählen. Unter „Werkzeuge/Board“ muss „Arduino/Genuino Uno“ ausgewählt sein, damit der Compiler aus dem Sketch-Quellcode den richtige Maschinencode baut.

9 Bibliotheken nachinstallieren

Für die ersten Experimente mit dem Arduino reichen die mitgelieferten Software-Bibliotheken aus. Später im Heft müssen Sie bei einigen Beispielen Bibliotheken nachinstallieren. Dies erledigen Sie unter dem Punkt „Sketch/Bibliotheken verwalten“. Die Übersicht zeigt zahlreiche online verfügbare Module, die die IDE per Mausklick automatisch aus dem Internet nachlädt und installiert.

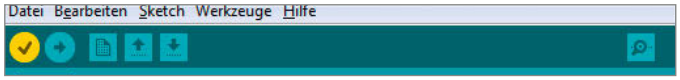






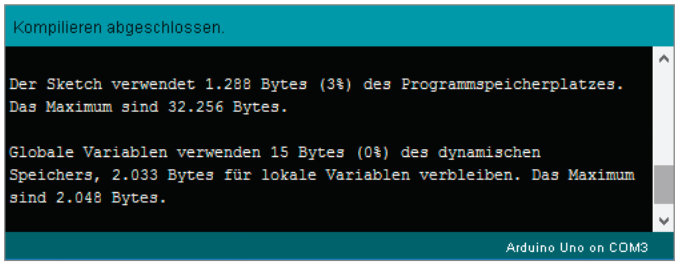
10 Beispiele laden

Weil man das Programmieren am besten durch das Studieren von funktionierenden Code-Schnipseln lernt, liefert die IDE zu jeder bereits mitgelieferten und nachinstallierten Bibliothek mehrere gut kommentierte Beispiele mit. Die lassen sich für viele eigene Projekte oft durch wenige Änderungen in den eigenen Sketch übernehmen.



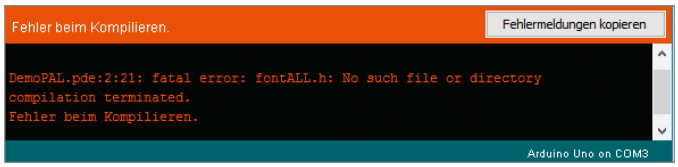
11 Sketch übersetzen

Um den Sketch zu übersetzen, klickt man auf den Button „Verifizieren“. Unten zeigt die IDE den Fortschritt des Vorgangs an. Das kann insbesondere beim erstmaligen Übersetzen eines Sketches etwas dauern, länger als eine Minute sollte das Kompilieren jedoch nicht dauern.



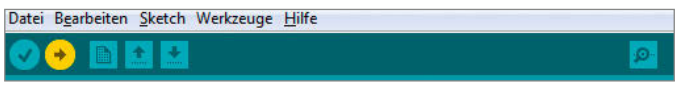
12 Erfolgsmeldung

War der Sketch ohne Fehler und hat der Compiler alle notwendigen Bibliotheken gefunden, läuft die Übersetzung ohne Fehler durch. Das Statusfenster zeigt an, wie viel Platz der Sketch im Flash belegt und wie viel Platz der Sketch beim Ablauf im RAM des Arduino verbraucht.



13 Fehlermeldung

Meldet der Compiler, dass irgendetwas fehlt („No such File or Directory“), ist meist die erforderliche Bibliothek noch nicht installiert.



14 Sketch auf Arduino laden

Um das erzeugte Binary auf den Arduino zu laden, klicken Sie auf den Button „Hochladen“. Der Fortschrittsbalken zeigt an, wie viel des Programms bereits übertragen wurden.

15 Falscher COM-Port

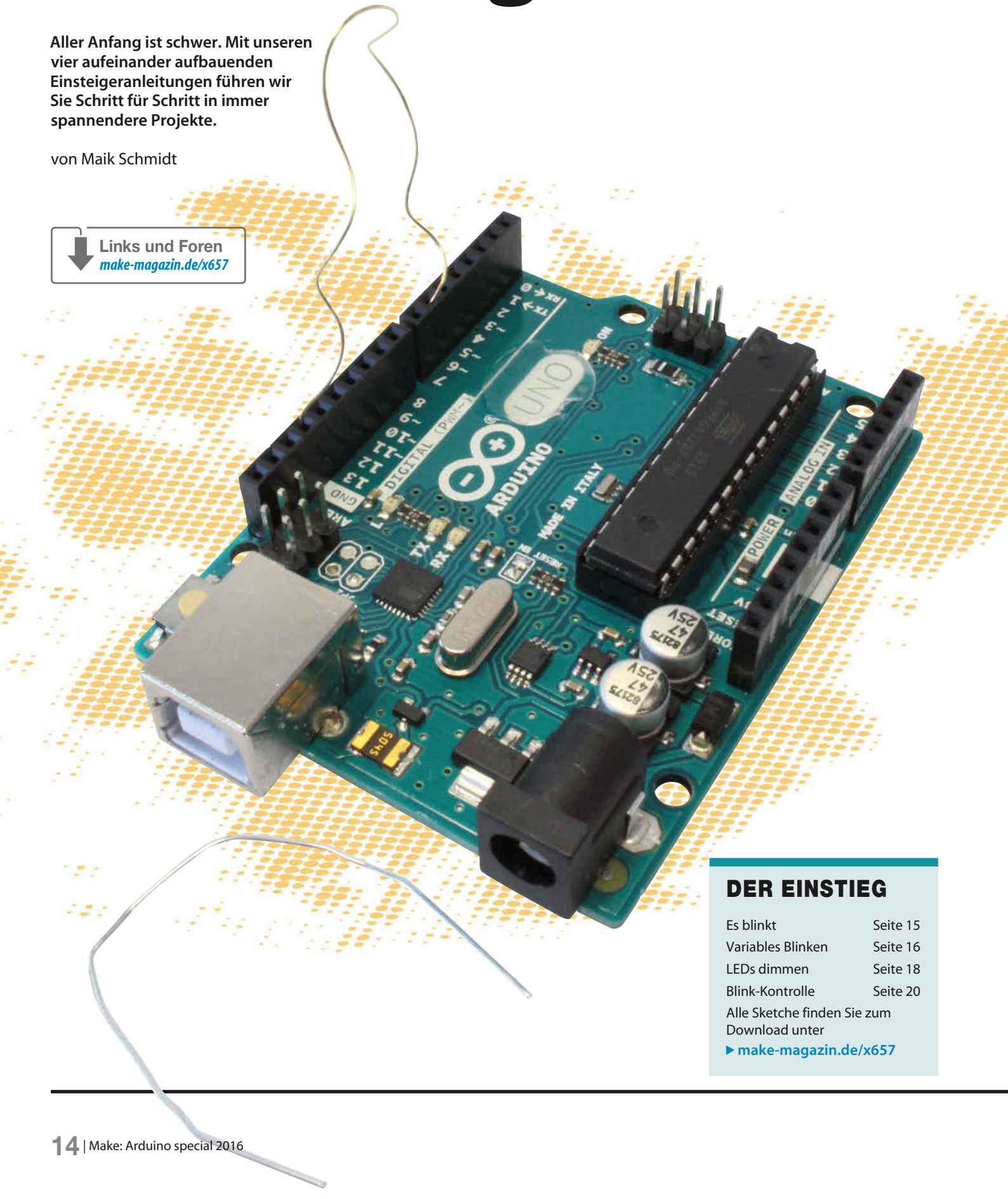
Spuckt das Statusfenster diese Fehlermeldungen beim Versuch des Hochladens aus, so ist entweder der Arduino nicht angeschlossen, der falsche COM-Port eingestellt oder der USB-Treiber nicht installiert.—*dab*

Der Einstieg

Aller Anfang ist schwer. Mit unseren vier aufeinander aufbauenden Einsteigeranleitungen führen wir Sie Schritt für Schritt in immer spannendere Projekte.

von Maik Schmidt

Links und Foren
make-magazin.de/x657



DER EINSTIEG

Alle Sketche finden Sie zum Download unter
▶ make-magazin.de/x657

Es blinkt

Im ersten Projekt soll der Arduino eine LED zum Blinken bringen.

Der Arduino hat insgesamt vier LEDs.

Die erste Aufgabe eines jeden Arduino-Neulings ist es, die Status-LED auf dem Arduino-Board zum Blinken zu bringen. Das mag nicht gerade spektakulär erscheinen, aber die Übung erfüllt in vielerlei Hinsicht einen wichtigen Zweck. Zu allererst stellt sie sicher, dass die Entwicklungsumgebung und die eingesetzte Hardware funktionieren. Wenn dieses Projekt nicht läuft, dann wird auch kein anderes laufen.

Darüber hinaus lassen sich an diesem überschaubaren Beispiel schon viele Konzepte erlernen, die später in so gut wie allen Arduino-Programmen benötigt werden.

Für die erste Übung benutzen wir nur die Bordmittel des Arduino, nämlich eine seiner LEDs. Die Abkürzung LED steht für Light-emitting diode (deutsch: Leuchtdiode). Eine LED ist also eine Diode, die auch leuchten kann. Eine Diode wiederum ist ein elektronisches Bauteil, das Strom nur in einer Richtung passieren lässt. Diese Eigenschaft macht Dioden zu wichtigen Komponenten in vielen Schaltkreisen, aber LEDs werden in der Regel aufgrund ihres Leuchtens eingesetzt. Sie dienen beispielsweise als Statusanzeigen oder auch nur als hübsche Effekte.

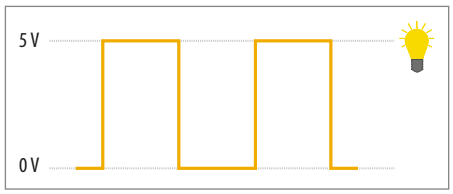
Auf dem Arduino-Board befinden sich vier LEDs in SMD-Bauweise, die alle unterschiedlichen Zwecken dienen. Ziel des ersten Projekts ist es, die Status-LED, die mit dem Buchstaben L gekennzeichnet ist, zum Blinken zu bringen.

Wann blinkt eine LED?

Damit eine LED blinkt, muss sie permanent zwischen den Zuständen Strom an und Strom aus wechseln. Die Dauer der jeweiligen Zustände bestimmt die Frequenz des Blinkens. Ist die LED beispielsweise für jeweils eine Sekunde im Zustand Strom an und dann für eine Sekunde im Zustand Strom aus, so blinkt sie verhältnismäßig langsam. Hektischer wird es, wenn die LED nur für 100 Millisekunden im jeweiligen Zustand bleibt.

Auf dem Arduino trägt der Zustand Strom an den Namen HIGH und Strom aus heißt LOW. HIGH entspricht auf dem Arduino Uno einer Spannung von 5 V und LOW entspricht 0 V. Jeder digitale Pin des Arduino befindet sich entweder im Zustand HIGH oder LOW.

Das Diagramm zeigt die Aktivität an einem digitalen Ausgang, der mit einer blinkenden LED verbunden ist. Der Ausgang wechselt permanent zwischen 5 V und 0 V und bleibt jeweils für die Zeit der definierten Pause in einem der beiden Zustände.



Um die Status-LED zum Blinken zu bringen, muss man auch noch wissen, dass sie mit dem digitalen Pin 13 des Arduino verbunden ist. Ist Pin 13 im Zustand HIGH, leuchtet die LED. Andernfalls ist sie aus.

Die Software

Das Programm, das die Status-LED zum Blinken bringt, ist neun Zeilen lang und besteht nur aus den Funktionen `setup` und `loop`, die für jedes Arduino-Programm definiert werden müssen.

Zuerst wird die Funktion `setup` definiert, die der Arduino automatisch beim Start aufruft. Sie dient der Initialisierung und in diesem Fall initialisiert sie den Pin, der mit der LED verbunden ist. Die digitalen Pins des Arduino können sowohl als Eingang als auch als Ausgang fungieren. In diesem Fall legt die Funktion `pinMode` fest, dass Pin 13 als Ausgang (`OUTPUT`) verwendet wird. Per Voreinstellung ist jeder digitale Pin nämlich erst einmal ein Eingang (`INPUT`).

Die Anweisung `pinMode(13, OUTPUT);` teilt dem Arduino also mit, dass Pin 13 ab sofort ein Ausgang ist und somit entweder `HIGH` oder `LOW` sein kann.

Die Funktion `pinMode` erwartet als Argumente die Nummer des zu initialisierenden Pins und den Modus, in den der Pin versetzt werden soll. Der Modus kann unter anderem `INPUT` oder `OUTPUT` sein. Ist ein Pin im Modus `OUTPUT`, kann er anschließend in die Zustände `HIGH` und `LOW` versetzt werden.

Den Wechsel zwischen `HIGH` und `LOW` regelt die Funktion `loop`, die ab Zeile 4 definiert wird. Sie ruft in Zeile 5 zuerst die Funktion `digitalWrite` auf, um Pin 13 in den Zustand `HIGH` zu versetzen. Damit wird die Status-LED eingeschaltet. Die Funktion `delay` sorgt dann dafür, dass dies für eine Sekunde (1000 Millisekunden) lang so bleibt, denn `delay` legt den Arduino für die angegebene Zeitspanne schlafen.

Sobald der Arduino wieder aufwacht, schaltet der nächste Aufruf von `digitalWrite` die LED wieder aus, denn diesmal wird der Pin 13 auf `LOW` gesetzt. Auch dieser Zustand soll wieder eine Sekunde lang so bleiben, wofür ein erneuter Aufruf von `delay` sorgt.

Würde die Funktion `loop` nur einmal aufgerufen, würde die Status-LED nur einmal ein- und nach einer Sekunde wieder ausgeschaltet. Der Arduino ruft `loop` aber kontinuierlich auf. Er führt also die folgenden Kommandos aus, solange er mit Strom versorgt wird:

```
digitalWrite(13, HIGH);   delay(1000)
digitalWrite(13, LOW);    delay(1000)
digitalWrite(13, HIGH);   delay(1000)
```

Blinken

```
1 void setup() {
2   pinMode(13, OUTPUT);
3 }
4 void loop() {
5   digitalWrite(13, HIGH);
6   delay(1000);
7   digitalWrite(13, LOW);
8   delay(1000);
9 }
```

```
digitalWrite(13, LOW);    delay(1000)
digitalWrite(13, HIGH);   delay(1000)
digitalWrite(13, LOW);    delay(1000)   ...
```

So entsteht letzten Endes das Blinken.

Action!

Das Programm muss jetzt nur noch übersetzt und auf das Arduino-Board übertragen werden. Dann sollte die Status-LED auf dem Board im Sekundentakt blinken und ambitionierteren Projekten steht nun nichts mehr im Wege.

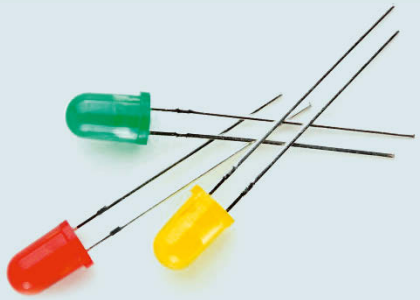
Um die Funktionsweise des Programms besser zu verstehen, bietet es sich an, mit verschiedenen Pausen-Werten zu experimentieren. Insbesondere mit sehr kurzen Pausen. Auch müssen die Pausen für die beiden Aufrufe von `delay` nicht gleich lang sein.

Etwas lästig ist bei solchen Experimenten, dass das Programm immer wieder übersetzt und auf den Arduino übertragen werden muss. Das nächste Projekt schafft hier ein wenig Abhilfe. *—Maik Schmidt/dab*

LEDS

LEDs gibt es – wie auch viele andere elektronische Bauteile – in zwei verschiedenen Bauformen. Im Hobby-Bereich sind LEDs zur Durchsteckmontage (englisch: Through-hole technology) beliebt, weil sie sich leicht handhaben lassen. Sie passen in Steckplatinen und lassen sich auch von Anfängern mühelos auf Platinen löten. Zu erkennen sind solche Bauteile an langen Drähten.

In der Industrie und in anspruchsvolleren Projekten kommen mittlerweile fast ausschließlich LEDs in der SMD-Bauform (surface-mount device, deutsch: oberflächenmontiertes Bauelement) vor. Sie sind oft effizienter und viel kleiner als ihre Pendants aus dem Bereich der Durchsteckmontage. Allerdings sind sie auch schwieriger zu handhaben, eben weil sie so klein sind und keine Anschlussdrähte mehr haben. Viele Teile sind mittlerweile so klein, dass sie nur noch maschinell und nicht mehr per Hand auf Platinen gelötet werden können.



LEDs gibt es in verschiedenen Farben und Bauformen.

Variables Blinken

Der Arduino reagiert auf seine Umwelt und lässt die LED schneller oder langsamer blinken.

Es ist schon ein erhebendes Gefühl, die kleine Status-LED auf dem Arduino-Board zu kontrollieren und blinken zu lassen. Plötzlich hat Software eine direkte Auswirkung auf die physische Welt.

Ähnlich spannend ist es, die physische Welt mittels Sensoren zu erfassen und auf Sensordaten zu reagieren. Dieser Artikel zeigt, wie sich die Blink-Frequenz der Status-LED in Abhängigkeit eines Eingangssignals verändern lässt.

Dazu muss dem Arduino von außen mitgeteilt werden, welche Blink-Frequenz verwendet werden soll. Beispielsweise könnten drei verschiedene Frequenzen voreingestellt sein, von denen der Nutzer jeweils eine wählen kann.

Dazu muss es ein Eingabegerät geben und die Versuchung liegt nah, einen Drucktaster, ein Potentiometer oder Ähnliches einzusetzen. Das hieße aber, mit Kanonen auf Spatzen zu schießen. Viel spannender ist es, mit simpelster Hardware auszukommen und ein wenig MacGyver-Flair zu spüren. Als Eingabegerät dient daher lediglich ein Stückchen Schaltdraht. Drahtreste findet man vielleicht noch in der Werkzeugliste, zur Not tut es auch eine auseinandergebogene Büroklammer, Bindedraht aus dem Garten oder eine Feder aus einem Kugelschreiber.

Erkennt die Signale

Zuerst wollen wir erklären, wie der Arduino überhaupt Eingangssignale erkennen und verarbeiten kann. Der Arduino kann zwei grundlegend verschiedene Arten von Signalen erfassen: digitale und analoge. Digitale Signale sind die einfachsten, denn sie können entweder an (HIGH, Spannung an) oder aus (LOW, Spannung aus) sein. Unterhalb eines bestimmten Werts gilt eine Spannung als LOW und darüber wird sie als HIGH interpretiert.

Für viele Anwendungen ist die Unterscheidung zwischen diesen beiden Zuständen völlig ausreichend. Beispielsweise liefern die meisten Bewegungsmelder, die in vielen Außenbeleuchtungen verbaut sind, ein digitales Signal. Solange sich in der Nähe des Bewegungsmelders niemand bewegt, liefert das Gerät den Wert LOW und die Lampe bleibt aus. Sobald eine Bewegung erkannt wird, sendet das Gerät den Wert HIGH und die Lampe wird eingeschaltet. Ein digitales Signal ist in diesem Fall nichts anderes als der Zustand eines Schalters.

Für das vorliegende Projekt, das die Status-LED in unterschiedlichen Geschwindigkeiten blinken lässt, reichen digitale Signale aus.

Was ist los auf der Welt?

Der Arduino verfügt über vierzehn digitale Pins und es ist einfach, sich über den aktuel-

PinStatus

```
1 void setup() {
2   pinMode(6, INPUT);
3   Serial.begin(9600);
4 }
5
6 void loop() {
7   Serial.println(digitalRead(6));
8 }
```

```
PinStatusPullup

1 void setup() {
2   pinMode(6, INPUT_PULLUP);
3   Serial.begin(9600);
4 }
5
6 void loop() {
7   Serial.println(digitalRead(6));
8 }
```

len Zustand eines jeden Pins zu informieren. Das Programm PinStatus gibt den aktuellen Wert, der an Pin 6 gemessen wird, kontinuierlich auf der seriellen Schnittstelle aus.

Dazu macht die setup-Funktion aus Pin 6 erst einmal einen Eingangspin. Anschließend ruft sie Serial.begin(9600) auf, so dass im weiteren Verlauf Nachrichten über die serielle Schnittstelle übertragen werden können.

Die loop-Funktion enthält nur eine einzige Anweisung und gibt unentwegt das Ergebnis der Funktion digitalRead auf der seriellen Schnittstelle aus. digitalRead ermittelt den aktuellen Zustand eines digitalen Pins, in diesem Fall des Pins 6. Wenn der Pin im Zustand HIGH ist, liefert die Funktion den Wert HIGH zurück, andernfalls LOW. Die Namen HIGH und LOW stehen intern allerdings für die Zahlen 1 und 0, das heißt, wenn am Pin das Signal HIGH anliegt, liefert digitalRead die Zahl 1 zurück, sonst die 0. In der Ausgabe des seriellen Monitors erscheinen später daher die Zahlen 1 und 0 statt der Wörter HIGH und LOW.

Bevor das Programm auf den Arduino übertragen wird, sollte Pin 6 noch mit einem Stück Schaltdraht verbunden werden. Es sieht dann so aus, als hätte der Arduino eine Antenne.

Wird das Programm gestartet, erscheint die Ausgabe im seriellen Monitor eher zufällig. Genau genommen ist sie das auch, denn die digitalen Pins des Arduino reagieren sehr empfindlich auf elektrostatische Ladungen, also auch auf elektrische Ladung der Luft oder auf der Haut. Sie können zu einem Wechsel von HIGH zu LOW oder umgekehrt führen. Es gibt aber einen einfachen Weg, dem Arduino dieses flatterhafte Verhalten auszutreiben.

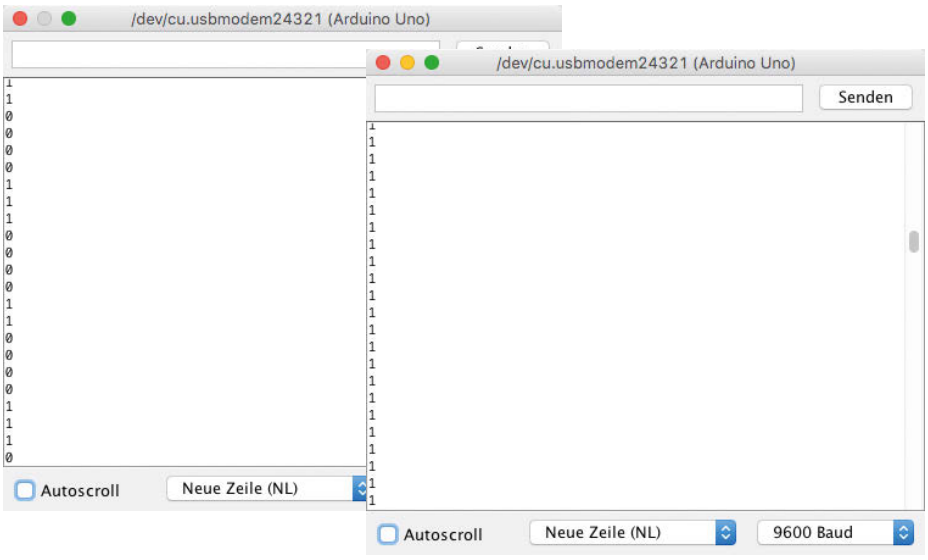
Ruhig bleiben!

Das Hauptproblem ist, dass „elektrostatischer Lärm“ am Pin ankommt, der unterdrückt werden muss. Dazu eignet sich ein Widerstand und der Arduino hat praktischerweise schon einen für jeden Pin eingebaut. Der muss nur aktiviert werden; das Programm PinStatusPullup zeigt, wie das geht.

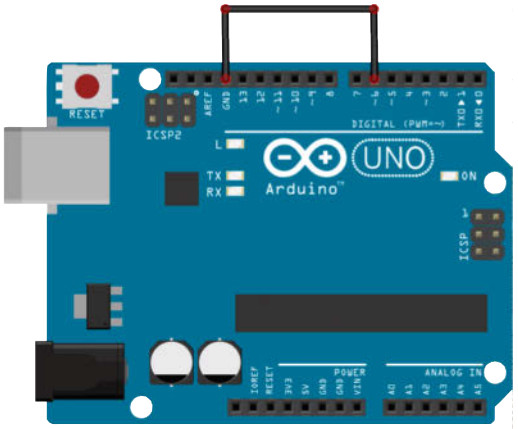
Vom vorhergehenden Programm unterscheidet sich das neue nur in der Initialisierung des Eingangspins in Zeile 2:

```
pinMode(6, INPUT_PULLUP);
```

Der Modus INPUT_PULLUP macht einen digitalen Pin zu einem Eingangspin und aktiviert gleichzeitig einen Pullup-Widerstand für diesen Pin. Der sorgt dafür, dass der Pin nicht mehr „flackert“, sondern immer ein sauberes HIGH- oder LOW-Signal liefert. Per Voreinstellung ist der Pin nun über einen Widerstand mit 5 Volt verbunden, also im Zustand HIGH respektive an. Somit lässt er sich prima als Schalter verwenden, denn sobald er mit einem LOW-Signal verbunden wird, schaltet er auf LOW um. (Dank des Widerstandes gibt es trotzdem keinen Kurzschluss). Das entsprechende LOW-Signal können die GND-Pins des Arduino liefern.



Die Eingangssignale eines offenen Pins: links ohne, rechts mit Pullup-Widerstand



An Pin 6 liegen über den Draht 0 V an.

Verbindet man mit einem Schaltdraht einen der GND-Pins mit Pin 6, so gibt das aktuelle Programm den Wert 0 aus – solange der GND-Pin und Pin 6 verbunden sind. Wird die Verbindung getrennt, gibt das Programm wieder 1 aus.

Der Code

Mit einem verlässlichen Eingangssignal lässt sich der Wechsel der Blink-Frequenz leicht als Programm realisieren. Im einfachsten Fall werden feste Frequenzwerte auf bestimmte Pins gelegt. Pin 6 kann zum Beispiel für Pausen von 600 Millisekunden stehen, Pin 4 für 400 Millisekunden und Pin 2 für 200 Millisekunden. Verbindet der Anwender dann den GND-Pin und Pin 6 mittels des Schaltdrahts, so blinkt die Status-LED mit einer Pause von 600 Millisekunden. Wird der GND-Pin mit Pin 4 verbunden, erhöht sich die Frequenz auf 400 Millisekunden.

Jetzt fehlt nur noch ein Programm, das den Status der Pins 2, 4 und 6 überwacht und die Blink-Frequenz entsprechend anpasst. Das Programm BlinkenVariabel beginnt mit der Anweisung

```
int aktuelle_pause = 1000;
```

Damit wird eine Variable mit dem Namen aktuelle_pause angelegt, die den Wert 1000 hat. Eine Variable steht für einen Platz im Speicher des Arduino. Dieser Platz hat einen Namen und einen bestimmten Wert. Der Wert kann sich im Verlauf eines Programms ändern. Ferner muss einer Variablen eine bestimmte Größe zugewiesen werden, damit die Arduino-Umgebung weiß, wie viel Platz sie für den Wert reservieren muss. In diesem Fall ist der Typ der Variablen int und das bedeutet, dass aktuelle_pause Werte zwischen –32768 und +32767 annehmen kann.

Die setup-Funktion stellt sicher, dass Pin 13 ein Ausgangspin ist und somit die Status-LED ein- und ausschalten kann. Anschließend macht sie die Pins 2, 4 und 6 zu Eingangspins und schaltet deren Pullup-Widerstände ein.

Die loop-Funktion übernimmt die eigentliche Steuerung des Programms und fragt den aktuellen Zustand der Pins 2, 4 und 6 ab. Dazu verwendet sie die if-Anweisung (wenn-dann-andernfalls). In Zeile 11 prüft sie wie folgt, ob der Pin 2 gerade mit dem GND-Pin verbunden ist:

```
if (digitalRead(2) == LOW) {
    aktuelle_pause = 200;   }
```

Wenn das der Fall ist, setzt das Programm die Variable aktuelle_pause auf den Wert 200. Falls Pin 2 nicht den Wert LOW hat, prüft das Programm den aktuellen Zustand von Pin 4 mit else if:

```
else if (digitalRead(4) == LOW) {
    aktuelle_pause = 400;
  }
```

Ist Pin 4 mit dem GND-Pin verbunden, erhält aktuelle_pause den Wert 400.

Analog verläuft das Ganze mit Pin 6 und falls keiner der Pins mit einem GND-Pin verbunden ist, wird aktuelle_pause der Wert 1000 zugewiesen. Dazu dient die else-Klausel am Ende:

```
else {
    aktuelle_pause = 1000;
  }
```

Am Ende der loop-Funktion sorgt die bekannte Kombination aus digitalWrite und delay dafür, dass die Status-LED wie gewünscht blinkt. Im Unterschied zu den vorherigen Beispielen bekommt delay aber keinen konstanten Wert übergeben, sondern die Variable aktuelle_pause. Je nachdem, mit welchem Eingangspin der GND-Pin verbunden ist, kann diese Variable die Werte 200, 400, 600 oder 1000 enthalten.

BlinkenVariabel

```
 1 int aktuelle_pause = 1000;
 2
 3 void setup() {
 4   pinMode(13, OUTPUT);
 5   pinMode(2, INPUT_PULLUP);
 6   pinMode(4, INPUT_PULLUP);
 7   pinMode(6, INPUT_PULLUP);
 8 }
 9
10 void loop() {
11   if (digitalRead(2) == LOW) {
12     aktuelle_pause = 200;
13   } else if (digitalRead(4) == LOW) {
14     aktuelle_pause = 400;
15   } else if (digitalRead(6) == LOW) {
16     aktuelle_pause = 600;
17   } else {
18     aktuelle_pause = 1000;
19   }
20
21   digitalWrite(13, HIGH);
22   delay(aktuelle_pause);
23   digitalWrite(13, LOW);
24   delay(aktuelle_pause);
25 }
```

ANALOGE SIGNALE

Analoge Signale können deutlich mehr Zustände abbilden. Auf dem Arduino sind es 1024 verschiedene, die durch die Zahlen 0 bis 1023 repräsentiert werden. Diese Signale eignen sich gut, um kontinuierliche Prozesse zu messen. So gibt es zum Beispiel viele analoge Temperatursensoren. Diese Sensoren liefern für jede mögliche Temperatur in einem festgelegten Temperaturbereich eine bestimmte Spannung. Je höher die Temperatur ist, umso höher ist in der Regel die dazugehörige Spannung. Diese Spannung wird dann vom Arduino auf einen der Werte zwischen 0 und 1023 abgebildet, wobei die 0 für 0 V und die 1023 für 5 V steht.

Nachdem das übersetzte Programm auf den Arduino geladen wurde, reicht ein abisoliertes Stück Schaltdraht zwischen einem GND-Pin und einem der Pins 2, 4 und 6, um die Status-LED auf dem Board in unterschiedlichen Geschwindigkeiten blinken zu lassen. *—dab*

LEDs dimmen

Obwohl der Arduino nur digitale Ausgänge hat, kann er trotzdem scheinbar analoge Ausgangssignale erzeugen.

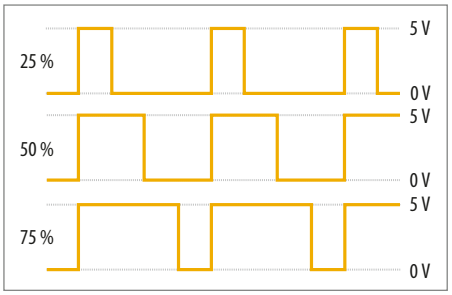
Wer ein wenig mit den bisherigen Blink-Beispielen experimentiert hat, konnte vielleicht feststellen, dass besonders kurze Blink-Pausen zu interessanten Effekten führen. Bei Blink-Pausen von 10 bis 15 Millisekunden lässt sich das Blinken der Status-LED nämlich kaum noch wahrnehmen und es scheint so, als würde die LED zwar konstant, aber etwas dunkler leuchten. Schuld daran ist das menschliche Auge, denn es ist verhältnismäßig träge und kann die einzelnen Blink-Phasen ab einer bestimmten Frequenz nicht mehr auseinanderhalten.

So richtig sauber ist der Effekt aber nicht und eine stufenlose und flackerfreie Regulierung der Helligkeit lässt sich so nicht erreichen. Eine perfekte Lösung wäre in so einem Fall ein analoger Ausgang, denn dieser könnte alle Spannungen zwischen 0 V und 5 V liefern und somit eine LED annähernd stufenlos dimmen.

Eine echte analoge Ausgabe stellt der Arduino zwar nicht zur Verfügung, aber er kann sie immerhin recht gut simulieren. Dazu dient die so genannte Pulsweitenmodulation (PWM).

Bei der Pulsweitenmodulation wechselt ein Ausgangssignal ständig und sehr schnell zwischen den Pegeln 0 V und 5 V. Je nachdem wie lang das Signal jeweils im Zustand 5 V verharrt, kommt am Ende eine höhere oder eine niedrigere Spannung an.

Das Diagramm zeigt, wie der Signal-Verlauf für die Werte 25 %, 50 % und 75 % der Maximalspannung von 5 V in etwa aussieht. Im Falle von 25 % ist das Signal zum Beispiel nur für 25 % der Zeit im Zustand 5 V. Im Mittel wird daher auch nur circa 25 % dieser Span-



Die Frequenz bleibt bei PWM gleich, aber die Dauer der Anschaltzeit wird länger.

nung (1,25 V) beim Empfänger ankommen. Statt also direkt 1,25 V zu senden, sendet der Arduino im schnellen Wechsel mal 0 V und mal 5 V. Dabei sorgt er dafür, dass in 25 % der Zeit 5 V gesendet werden. Weil das alles sehr schnell passiert, kommen effektiv die gewünschten 1,25 V an. Das funktioniert für alle anderen Prozentwerte ganz genauso und in den Extremfällen 0 % und 100 % kommen eben 0 V beziehungsweise die ganzen 5 V an.

Ans Werk

Von Hause aus unterstützen nur die digitalen Pins 3, 5, 6, 9, 10 und 11 des Arduino PWM direkt. Sie sind daher auf dem Board alle mit einem ~-Zeichen markiert. Leider gehört der Pin 13, mit dem die Status-LED verbunden ist, nicht dazu. Es gibt aber eine Vielzahl von Bibliotheken, die jedem digitalen Arduino-Pin das PWM-Verfahren beibringen können. Eine davon ist SoftPWM (https://github.com/bhagman/SoftPWM).

Vor der ersten Verwendung der Bibliothek muss sie installiert werden. SoftPWM taucht noch nicht in der Liste der offiziellen Arduino-Bibliotheken auf und kann daher nicht direkt über den Library-Manager installiert werden. Trotzdem ist die Installation keine große Hürde. Dazu muss zuerst das ZIP-Archiv unter https://github.com/bhagman/SoftPWM/archive/master.zip der Bibliothek heruntergeladen werden. Dieses Archiv kann dann mit der Arduino-IDE über den Menüpunkt „Sketch > Bibliothek einbinden > .ZIP-Bibliothek hinzufügen …“ importiert werden.

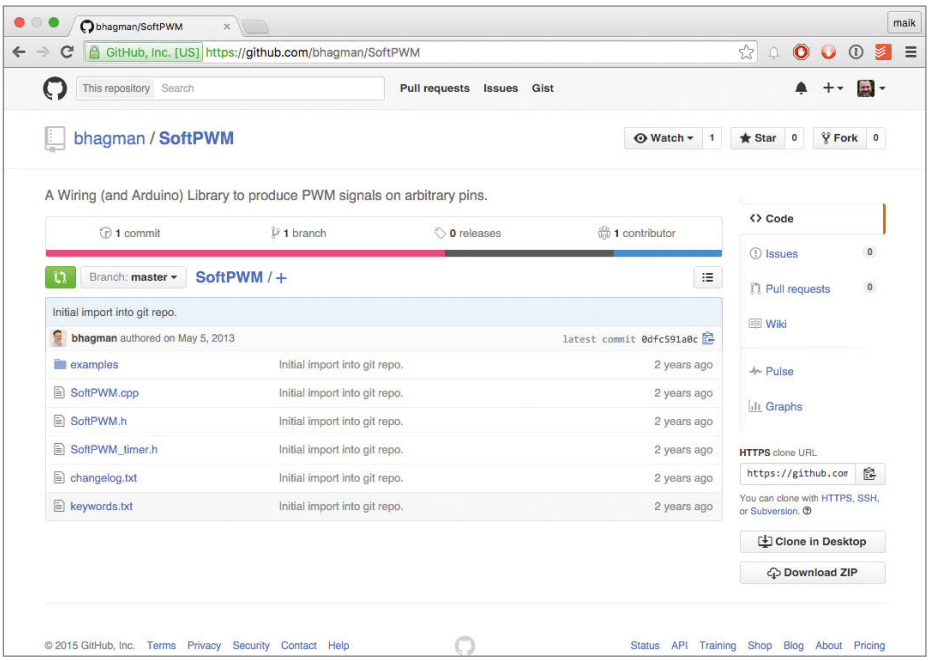
Ein erstes Beispiel

Sobald die SoftPWM-Bibliothek installiert ist, wird das Dimmen der Status-LED zum Kinderspiel, denn SoftPWM versteckt die ganze Komplexität der Signalerzeugung hinter einer denkbar einfachen Schnittstelle. Das kurze Programm zeigt, wie sich die Status-LED auf 10 % ihrer maximalen Helligkeit dimmen lässt.

Das Programm beginnt mit einer #include-Anweisung, um die SoftPWM-Bibliothek einzubinden. Ohne diese Anweisung würde die Arduino-Umgebung die Funktionen, die von der Bibliothek angeboten werden, nicht kennen.

Anschließend definiert das Programm die Variable LED_PIN und setzt sie auf den Pin 13, also die Status-LED des Arduino. Die Variable LED_PIN kann dann im weiteren Verlauf des Programms überall dort verwendet werden, wo sonst die Zahl 13 direkt eingetragen würde.

Diese Vorgehensweise hat eine Reihe von Vorteilen. So ist zum Beispiel an jeder Stelle klar, welche Bedeutung die Variable LED_PIN hat. Bei einer blanken Zahl wie der 13 könnte es auch schon mal zu Missverständnissen kommen. Darüber hinaus lässt sich das Programm auch leichter anpassen. Wenn zum Beispiel eine LED an einem anderen Pin verwendet werden soll, muss die Nummer dieses Pins nur an einer Stelle im Programm eingetragen werden, nämlich bei der Definition von LED_PIN.



Die SoftPWM-Libary lädt man mit dem „Download-ZIP“-Button rechts unten herunter.

Im Gegensatz zu vorhergehenden Beispielen ist der Typ der Variablen LED_PIN nicht mehr int, sondern unsigned int. Das bedeutet, dass die Variable LED_PIN nur positive Werte im Bereich 0 bis 65535 speichern kann. Für eine Variable, die die Nummer eines Pins speichert, ist das sinnvoll, denn Pins mit negativen Nummern gibt es nicht.

Die setup-Funktion ruft die SoftPWMBegin-Funktion auf, um die SoftPWM-Bibliothek zu initialisieren. Anschließend setzt sie die Helligkeit der Status-LED mit der SoftPWMSetPercent-Funktion auf 10 %. Die loop-Funktion tut anschließend nichts mehr und nachdem das Programm auf den Arduino geladen wurde, leuchtet die Status-LED, ohne zu flackern, mit 10 % ihrer maximalen Leuchtkraft.

Mehr Flexibilität

Prinzipiell funktioniert das Dimmen also, aber das erste Programm ist nicht sonderlich flexibel und müsste für neue Helligkeitswerte jedes Mal übersetzt und auf den Arduino übertragen werden. Um die Helligkeit der Status-LED etwas genauer zu kontrollieren, soll daher wieder ein Schaltdraht zum Einsatz kommen.

Zur Steuerung sollen die digitalen Pins 3, 4, 5 und 6 dienen. Wenn Pin 3 LOW ist, ist die Helligkeit der LED 0 %, das heißt, sie ist aus. Bei Pin 4 steigt die Helligkeit auf 25 %, Pin 5 steht für 50 % und Pin 6 für 75 %. Falls keiner der Pins auf LOW ist, leuchtet die LED mit vollen 100 %.

Das entsprechende Programm bindet wie gehabt die SoftPWM-Bibliothek mit der #include-Anweisung ein. Anschließend definiert sie wieder die Variable LED_PIN. Zusätzlich definiert sie die Variable STANDARD_HELLIGKEIT, die festlegt, mit welcher Helligkeit die Status-LED leuchtet, wenn an keinem der definierten Eingangspins das Signal LOW anliegt.

Schließlich gibt es noch eine dritte Variable mit dem Namen helligkeit. Sie enthält die aktuell zu setzende Helligkeit und steht erst einmal auf 100 %.

Die setup-Funktion initialisiert die SoftPWM-Bibliothek mit der Funktion SoftPWMBegin und versetzt dann die Pins 3 bis 6 in den Zustand INPUT_PULLUP. Sie werden also zu Eingangspins und ihre internen Pullup-Widerstände werden aktiviert, so dass sie immer ein sauberes HIGH- oder LOW-Signal senden.

LED dimmen

```
1 #include <SoftPWM.h>
2
3 unsigned int LED_PIN = 13;
4
5 void setup() {
6   SoftPWMBegin();
7   SoftPWMSetPercent
    (LED_PIN, 10);
8 }
9
10 void loop() { }
```

Die loop-Funktion ist im Kern dieselbe wie im vorhergehenden Artikel. Diesmal werden mit der if-Anweisung die Pins 3 bis 6 abgefragt und der Wert der Variablen helligkeit entsprechend gesetzt. Beispielsweise sorgt die Anweisung

```
else if (digitalRead(5) == LOW) {
    helligkeit = 50;
}
```

dafür, dass die Variable helligkeit auf den Wert 50 gesetzt wird, wenn Pin 5 im Zustand LOW ist.

Am Ende der loop-Funktion stellt die Funktion SoftPWMSetPercent für die Status-LED die zuvor ermittelte Helligkeit ein.

Fazit

Die Simulation analoger Signale mittels Pulsweitenmodulation ist eine gängige und wichtige Technik. Sie kommt nicht nur beim Dimmen von LEDs zum Einsatz, sondern unter anderem auch bei der Kontrolle von Motoren. Der Arduino unterstützt das Verfahren recht gut und durch den Einsatz von Bibliotheken wie SoftPWM lässt sich die Situation sogar noch verbessern. *—dab*

LED Helligkeit steuern

```
#include <SoftPWM.h>

unsigned int LED_PIN = 13;
unsigned int STANDARD_HELLIGKEIT = 100;

unsigned int helligkeit = STANDARD_HELLIGKEIT;

void setup() {
  SoftPWMBegin();
  pinMode(3, INPUT_PULLUP);
  pinMode(4, INPUT_PULLUP);
  pinMode(5, INPUT_PULLUP);
  pinMode(6, INPUT_PULLUP);
}

void loop() {
  if (digitalRead(3) == LOW) {
    helligkeit = 0;
  } else if (digitalRead(4) == LOW) {
    helligkeit = 25;
  } else if (digitalRead(5) == LOW) {
    helligkeit = 50;
  } else if (digitalRead(6) == LOW) {
    helligkeit = 75;
  } else {
    helligkeit = STANDARD_HELLIGKEIT;
  }

  SoftPWMSetPercent(LED_PIN, helligkeit);
}
```

Blink-Kontrolle

Über die serielle Schnittstelle erteilen wir dem Arduino Befehle, wie seine LED blinken soll.

Die bisherigen Projekte hatten die Status-LED des Arduino schon recht gut im Griff. Die LED blinkt im vorgegebenen Takt und lässt sich sogar stufenlos dimmen. Auch konnte das Verhalten der LED mittels eines Schaltdrahts in festgelegten Grenzen verändert werden.

Die Lösung mit dem Schaltdraht funktioniert aber nur für eine Handvoll voreingestellter Werte. Sie eignet sich nicht, um annähernd beliebige Blink-Pausen oder alle möglichen Helligkeitsabstufungen einer LED vorzugeben. Auch andere Eingabegeräte, wie zum Beispiel Drucktaster, kommen hier schnell an ihre Grenzen.

Das wohl universellste Eingabemedium ist und bleibt die gute alte Tastatur und über die serielle Schnittstelle lässt sie sich problemlos zur Kontrolle des Arduino einsetzen.

An die Tasten

Es liegt daher nahe, die gewünschte Blink-Pause der Status-LED über die serielle Schnittstelle abzufragen und einzustellen. Damit ließe sich die Pause dann in beliebiger Genauigkeit festlegen, denn sie könnte auf der PC-Tastatur eingegeben werden.

Das Programm LED_Seriell_Steuern erledigt diesen Job mit simplem Mitteln. Vor der Analyse des Quelltextes ist es sinnvoll, das Programm zu übersetzen und auf den Arduino zu übertragen. Anschließend kann die gewünschte Blink-Pause der Status-LED über den seriellen Monitor des Arduino eingestellt werden. Wenn der Arduino einen neuen Wert für die Blink-Pause empfangen hat, gibt er ihn auf der seriellen Schnittstelle aus.

Die Initialisierung

Wie gewohnt definiert das Programm zuerst eine Variable, die die Nummer des digitalen Pins enthält, mit dem die Status-LED des Arduino verbunden ist. In diesem Fall weicht die Definition der Variablen aber geringfügig von den bisherigen Beispielen ab:

```
unsigned char LED_PIN = 13;
```

Statt des Datentyps unsigned int wird hier der Datentyp unsigned char verwendet. Dieser Typ kann die Werte 0 bis 255 annehmen und eignet sich somit hervorragend zur Speicherung einer Pin-Nummer. Darüber hinaus spart er gegenüber unsigned int ein wenig Speicherplatz und ist somit eine sinnvolle Wahl.

Für den Datentypen unsigned char unterstützt der Arduino übrigens auch die knappere Formulierung byte. Somit ist

```
unsigned char LED_PIN = 13;
```

dasselbe wie

```
byte LED_PIN = 13;
```

Die zweite Zeile des Programms definiert die Variable BAUD_RATE, die die Geschwindigkeit der seriellen Kommunikation festlegt. Für dieses Programm reichen gemächliche 9600 Baud aus, aber potenziell kann die Geschwindigkeit durchaus Werte bis zu 250 000

LED seriell steuern

```
 1 unsigned char LED_PIN = 13;
 2 unsigned long BAUD_RATE = 9600;
 3 unsigned int aktuelle_pause = 500;
 4 void setup() {
 5   pinMode(LED_PIN, OUTPUT);
 6   Serial.begin(BAUD_RATE);
 7 }
 8 void loop() {
 9   if (Serial.available() > 0) {
10     aktuelle_pause = Serial.parseInt();
11     if (Serial.read() == ,\n') {
12       Serial.print("Neue Pause: ");
13       Serial.println(aktuelle_pause);
14     }
15   }
16   digitalWrite(LED_PIN, HIGH);
17   delay(aktuelle_pause);
18   digitalWrite(LED_PIN, LOW);
19   delay(aktuelle_pause);
20 }
```

In der Zeile oben gibt man die Pausendauer ein und überträgt sie mit dem „Senden“-Button.

annehmen. Dafür ist der Datentyp unsigned int nicht ausgelegt, denn er reicht nur bis 65 535. Der Datentyp unsigned long hingegen macht erst bei 4 294 967 295 schlapp. Daher kommt er hier und für zukünftige Projekte zum Einsatz.

Schließlich definiert die Variable aktuelle_pause die momentan eingestellte Blink-Pause. Ihr Wert kann durch neue Eingaben, die über die serielle Schnittstelle ankommen, verändert werden. Per Voreinstellung liegt er bei 500 Millisekunden.

Als Nächstes wird die Funktion setup definiert. Sie versetzt den Pin, der mit der Status-LED verbunden ist, in den Ausgabemodus und initialisiert die Kommunikation über die serielle Schnittstelle.

Serielle Kommunikation

Spannend wird es in der loop-Funktion. Sie prüft zuallererst, ob neue Daten an der seriellen Schnittstelle anliegen. Dazu nutzt sie die Funktion Serial.available. Obwohl der Name so klingt, prüft sie nicht, ob eine serielle Schnittstelle vorhanden ist. Vielmehr liefert sie die Anzahl neuer Bytes zurück, die über die serielle Schnittstelle empfangen wurden und darauf warten, abgeholt zu werden.

Wenn dort bereits mehr als 0 Bytes liegen, ruft das Programm die Funktion Serial.parseInt auf. Diese Funktion versucht, einen ganzzahligen Wert von der seriellen Schnittstelle zu lesen. Angenommen, über die serielle Schnittstelle wurden die Zeichen ‚1‘, ‚2‘ und ‚3‘ übermittelt, dann liefert Serial.parseInt die Zahl 123.

Bei der Übertragung von Daten über eine serielle Schnittstelle ist es sehr wichtig, dass sich Sender und Empfänger genau auf das Format der zu übertragenden Daten einigen. Die Regeln, die dazu aufgestellt werden, heißen Protokoll.

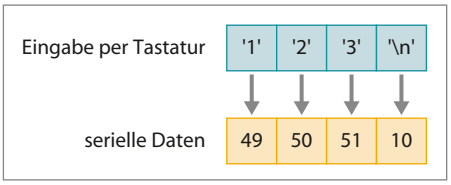
Bei Protokollen, die über eine serielle Schnittstelle ausgetauscht werden, ist es besonders wichtig, den Anfang und das Ende von Daten zu kennzeichnen. Um das Ende einer Übertragung anzuzeigen, ist das Newline-Zeichen \n sehr geläufig. Auch in diesem Beispiel kommt es zum Einsatz und es wird praktischerweise per Voreinstellung vom seriellen Monitor des Arduino gesendet, wenn der Benutzer die Return-Taste drückt.

Deshalb prüft das Programm nach dem Aufruf von Serial.parseInt, ob als Nächstes Zeichen ein \n-Zeichen (Newline) übertragen wurde, ob also der Benutzer die Return-Taste gedrückt hat. Zum Lesen eines einzelnen Zeichens dient die Funktion Serial.read und wenn ein Newline-Zeichen empfangen wurde, gibt das Programm den Wert der Variablen aktuelle_pause aus.

Am Ende der loop-Funktion läuft der altbekannte Code, der die Status-LED zum Blinken bringt. Er wird immer durchlaufen, also auch, wenn keine neue Blink-Pause über die serielle Schnittstelle übertragen wurde.

Hinter den Kulissen

Die Arduino-Umgebung versteckt komplexe Vorgänge wie die serielle Kommunikation hinter einer einfachen Fassade. Trotzdem hilft es zu verstehen, was sich hinter den Kulissen so tut. Wenn ein Anwender zum Beispiel die Zahl 123 eingibt und dann die Return-Taste drückt, passiert eine ganze Menge. Die Zahl wird nämlich nicht als ganze Zahl, sondern Zeichen für Zeichen nacheinander übertragen. Darüber hinaus werden die Zeichen auch noch kodiert, so dass die gesamte Kommunikation am Ende aus lauter Zahlen besteht.



Zahlen wurden als ASCII-Zeichen übertragen.

Das Bild unten zeigt, was im konkreten Fall der Zahl 123 passiert. Statt nämlich die Zahl 123 am Stück zu übertragen, wird sie in ihre einzelnen Ziffern (1, 2 und 3) zerlegt. Ferner wird noch das Newline-Zeichen gesendet, das beim Drücken der Return-Taste erzeugt wird.

Alle Zeichen – also Ziffern und das Newline-Zeichen – werden in Zahlen umgewandelt. Dazu wird die ASCII-Kodierung (American Standard Code for Information Interchange) verwendet. In dieser Kodierung steht die Zahl 48 für die 0, die Zahl 49 für die 1 und so weiter. Dem Newline-Zeichen ist die Zahl 10 zugeordnet und so werden am Ende die Zahlen 49, 50, 51 und 10 übertragen, um die Zahl 123 zu kodieren.

Grenzfälle

Im Programm ist die Anweisung

```
aktuelle_pause = Serial.parseInt();
```

einigermaßen brisant. Die Funktion Serial.parseInt liefert nämlich einen Wert von Typen long zurück. Der kann potenziell Werte von –2 147 483 648 bis 2 147 483 647 aufnehmen. Weil die Variable aktuelle_pause aber den Typen unsigned int hat, wird der Wert bei der Zuweisung entsprechend „zurechtgestutzt“. Wer mehr über dieses Verhalten lernen will, sollte negative und besonders große Werte eingeben und sehen, was im seriellen Monitor tatsächlich ausgegeben wird.

Während der Entwicklung neuer Programme und bei der Fehlersuche ist der derielle Monitor ein unverzichtbares Hilfsmittel, um sich die Werte von Variablen anzusehen. *—dab*

Fortschritt

Nachdem Sie sich in den vorhergehenden Kapiteln in die Grundlagen der Hardware-Programmierung eingearbeitet haben, steigern wir das Niveau nun ein wenig. Der Arduino interagiert nun mit seiner Umwelt, misst Temperaturen und steuert Motoren.

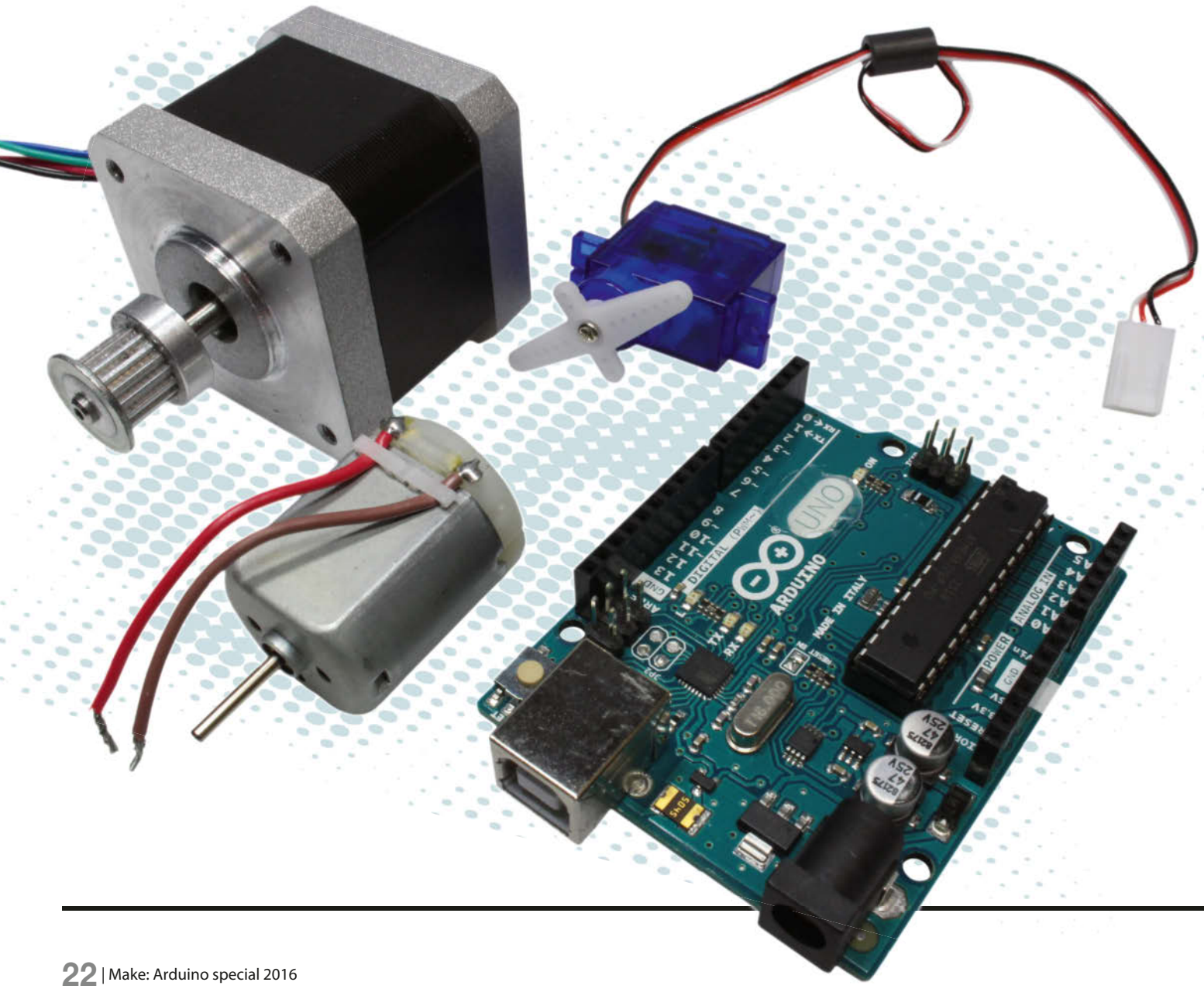
von Maik Schmidt

Links und Foren
make-magazin.de/xn5p

FORTSCHRITT

Ab hier benötigen Sie zusätzliche Materialien für die Projekte. Eine Einkaufsliste finden Sie unter dem Link.

Alle Sketche finden Sie zum Download unter
▶ www.make-magazin.de/xn5p



Analoge Eingänge

Mit seinen analogen Eingängen kann der Arduino seine Umwelt differenzierter betrachten.

Die bisherigen Projekte machten bereits regen Gebrauch von den digitalen Ports des Arduino. Sie wurden als Eingänge und als Ausgänge eingesetzt und simulierten sogar die Ausgabe analoger Werte, um die Status-LED zu dimmen.

Wichtig sind aber auch die analogen Eingänge, denn es gibt eine Vielzahl spannender und nützlicher analoger Sensoren. Aber auch ohne Sensoren lässt sich mit den analogen Eingängen schon einiges anstellen. Sie verwandeln den Arduino zum Beispiel im Handumdrehen in einen Batterie-Tester.

Analoge Eingänge lesen

Die analogen Eingänge A0 bis A5 lassen sich ganz ähnlich abfragen wie die digitalen. Die dazugehörige Funktion heißt `analogRead` und sie erwartet die Nummer des Eingangs, der ausgelesen werden soll. Sie liefert den am Eingang gemessenen Wert als eine Zahl zwischen 0 und 1023 zurück. Diese Zahlen entsprechen der gemessenen Spannung, und je größer die Zahl ist, um so höher war die gemessene Spannung.

Das Programm `Analogport_Lesen` gibt die aktuellen Werte aller analogen Eingänge permanent auf der seriellen Schnittstelle aus. Das Programm beginnt mit der Definition der Variablen `BAUD_RATE`, weil es die gemessene Batteriespannung auf der seriellen Schnittstelle ausgibt. Die Definition von `BAUD_RATE` unterscheidet sich geringfügig von den Definitionen in den früheren Beispielen, denn sie beginnt mit dem Schlüsselwort `const`. Dieses Schlüsselwort teilt dem Arduino mit, dass `BAUD_RATE` eine Konstante ist und dass sich der Wert von `BAUD_RATE` bis zum Programmende nicht ändern kann und wird. Das ist eine äußerst nützliche Eigenschaft, denn sie erleichtert das Studium des Quelltextes und verhindert unbeabsichtigte Änderungen.

Die setup-Funktion nutzt die Konstante `BAUD_RATE` zur Initialisierung der seriellen Schnittstelle. Danach gibt die `loop`-Funktion kontinuierlich die aktuellen Werte aller analogen Eingänge aus. Die Funktion wirkt erst einmal wüst, aber im Grunde tut sie sechsmal dasselbe. Sie gibt den Namen des Eingangs und ein Gleichheitszeichen auf der seriellen Schnittstelle aus. Dann liest sie den Wert des Eingangs mittels `analogRead` und gibt auch diesen Wert aus. Schließlich gibt sie noch ein Komma aus, um die Ausgabe etwas übersichtlicher zu gestalten.

So gut wie alle Ausgaben erledigt die Funktion `Serial.print`, denn sie gibt nur die übergebenen Daten und keinen Zeilenumbruch aus. So stehen alle Werte in der Ausgabe nebeneinander. Nur die letzte Anweisung nutzt `Serial.println` und sorgt so dafür, dass die nächste Ausgabe in einer neuen Zeile beginnt.

Alle Werte, die ausgegeben werden, wenn das Programm auf den Arduino übertragen wurde, sind mehr oder weniger zufällig. Sie hängen nicht zuletzt vom aktuellen Wetter und von atmosphärischen Störungen ab. Die vorhergehenden Projekte haben bereits gezeigt, wie sich der Zustand eines digitalen Pins auf dem Arduino gezielt kontrollieren lässt. Aber auch die Werte für die analogen Pins A0 bis A5 lassen sich zu Testzwecken direkt über den Arduino halbwegs akkurat setzen.

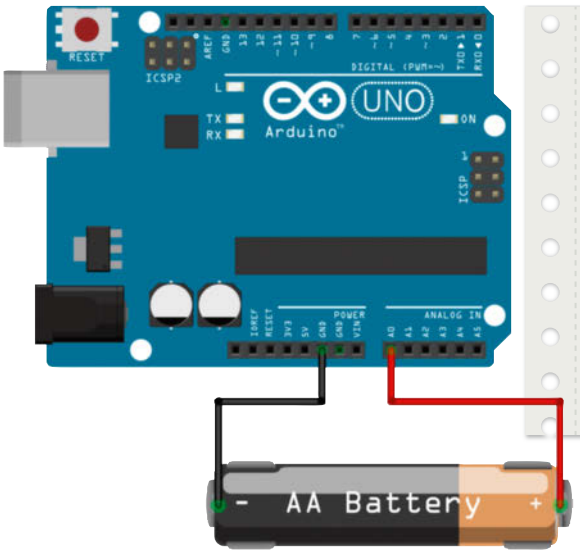
Verbindet man beispielsweise den 3V3-Pin und den Port A0 mit einem Schaltdraht, so wird der Wert von Port A0 in der Ausgabe des Programms ungefähr bei 663 landen. Das bedeutet, dass die Spannung von 3,3 V, die der Pin 3V3 liefert, in den analogen Wert 663 umgewandelt wird. Die tatsächlichen Ergebnisse können dabei leicht variieren.

Der gemessene Wert lässt sich wiederum leicht in einen Spannungswert umrechnen. Dazu muss man nur wissen, dass der Arduino

Analogports_lesen

```
1  const unsigned long BAUD_RATE = 9600;
2
3  void setup() {
4    Serial.begin(BAUD_RATE);
5  }
6
7  void loop() {
8    Serial.print("A0=");
9    Serial.print(analogRead(0));
10   Serial.print(",");
11   Serial.print("A1=");
12   Serial.print(analogRead(1));
13   Serial.print(",");
14   Serial.print("A2=");
15   Serial.print(analogRead(2));
16   Serial.print(",");
17   Serial.print("A3=");
18   Serial.print(analogRead(3));
19   Serial.print(",");
20   Serial.print("A4=");
21   Serial.print(analogRead(4));
22   Serial.print(",");
23   Serial.print("A5=");
24   Serial.println(analogRead(5));
25 }
```

Über den seriellen Monitor lassen sich die Werte der analogen Ports anzeigen.



Batterie-Tester

```
 1 const unsigned long BAUD_RATE = 9600;
 2 const unsigned int AUFLOESUNG = 1024;
 3 const float ARDUINO_SPANNUNG = 5.0;
 4
 5 void setup() {
 6   Serial.begin(BAUD_RATE);
 7 }
 8
 9 void loop() {
10   unsigned int batterie_spannung = analogRead(A0);
11   Serial.print(ARDUINO_SPANNUNG * batterie_spannung / AUFLOESUNG);
12   Serial.println("V");
13 }
```

Der Arduino als Batterietester

Uno mit einer Versorgungsspannung von 5 V läuft (es gibt auch andere Arduino-Modelle, die mit 3,3 V funktionieren) und dass ein analoger Port eine gemessene Spannung in einen Wert zwischen 0 und 1023 umwandelt. Das sind 1024 mögliche Werte für die Spannungen 0 V bis 5 V.

Für die 663 ergibt sich daher eine Spannung von 663 × 5 V / 1024 = 3,24 V. Das sind zwar nicht exakt 3,3 V, aber mit geringen Schwankungen muss man in der Elektronik leben. In der Praxis sind sie selten relevant.

Aufbauarbeit

Wenn es so einfach ist, Spannungen zu lesen, die der Arduino selbst erzeugt, kann es doch auch nicht schwierig sein, andere Spannungen zu lesen. Batterien drängen sich hier geradezu auf und tatsächlich ist es ein Leichtes, den Zustand handelsüblicher Batterien mit dem Arduino zu prüfen.

Dazu reichen zwei Drähte, von denen einer mit einem GND-Pin des Arduino und der andere mit dem A0-Pin verbunden wird. Dann fehlt nur noch das Programm, das die Spannung einer Batterie ausgibt, die mit dem GND-Pin und dem A0-Pin verbunden ist. Sobald das Programm auf den Arduino übertragen wurde, kann der GND-Draht an den Minus-Pol der Batterie und der A0-Draht an den Plus-Pol der Batterie gehalten werden.

Wie das Programm zum Auslesen aller analogen Eingänge beginnt auch dieses Programm mit der Definition der Variablen `BAUD_RATE`, weil es die gemessene Batteriespannung auf der seriellen Schnittstelle ausgibt. Es folgt die Definition der Konstanten `AUFLOESUNG`. Sie definiert die maximale Anzahl der Werte, die von einem der analogen Pins des Arduino zurückgeliefert werden können (0 bis 1023), also mit wie vielen Werten der Arduino Spannungen von 0 V bis 5 V in Zahlen „auflöst".

Schließlich wird die Konstante `ARDUINO_SPANNUNG` definiert. Sie enthält die Betriebsspannung des Arduino und muss daher auch Nachkommastellen unterstützen, weil es auch Arduinos gibt, die mit 3,3 V und nicht mit 5 V arbeiten.

Um Fließkommazahlen zu speichern, bietet der Arduino den Datentyp `float` an. Er kann Werte zwischen –3.4E+38 und +3.4E+38 abbilden. Diese Werte sind allerdings für maximal sechs bis sieben Nachkommastellen genau. Für viele Zwecke reicht das aber völlig aus.

Die angeschlossene Batterie ist noch frisch.

KEINE 9-V-BLOCK-BATTERIEN!

Es dürfen keine Batterien angeschlossen werden, die mehr als 5 V liefern! Damit würde der Arduino im Extremfall zerstört. 6-V- und 9-V-Blockbatterien sind damit tabu.

Alle_Ports_lesen

```
const unsigned long BAUD_RATE = 9600;
const unsigned char MAX_ANALOG_PINS = 6;
const unsigned char MAX_DIGITAL_PINS = 14;

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

void loop() {
  // Werte aller analogen Eingänge ausgeben.
  for (unsigned char analog_pin = 0; analog_pin < MAX_ANALOG_PINS; analog_pin++) {
    Serial.print("A");
    Serial.print(analog_pin);
    Serial.print("=");
    Serial.print(analogRead(analog_pin));
    Serial.print(",");
  }

  // Werte aller digitalen Eingänge ausgeben.
  for (unsigned char digital_pin = 0; digital_pin < MAX_DIGITAL_PINS; digital_pin++) {
    Serial.print("D");
    Serial.print(digital_pin);
    Serial.print("=");
    Serial.print(digitalRead(digital_pin));
    if (digital_pin < MAX_DIGITAL_PINS -1) {
      Serial.print(",");
    }
  }
  Serial.println();
}
```

Die setup-Funktion initialisiert nur die serielle Schnittstelle. Die loop-Funktion liest zunächst den Wert, der an Pin A0 anliegt. Dann wandelt sie den gemessenen Wert in eine Spannung um und gibt sie mit Serial.print auf der seriellen Schnittstelle aus. Anschließend nutzt sie Serial.println, um noch den Buchstaben V und einen Zeilenumbruch auszugeben. Der Screenshot zeigt die Ausgabe des Batterie-Testers für eine frische AA-Batterie.

Alle Ports lesen

Abschließend soll noch ein Programm entwickelt werden, das die aktuellen Werte aller Arduino-Pins ausgibt. Der Code zur Ausgabe der analogen Eingangswerte war ein wenig umständlich, weil er eine ganze Menge ähnlicher Anweisungen wiederholt hat. Bei sechs Eingängen ist das vielleicht noch vertretbar, aber bei insgesamt 20 digitalen und analogen Werten wird das Ganze schnell unübersichtlich.

Das Programm Alle_Ports_Lesen ist in etwa so umfangreich wie das Programm zur Ausgabe der analogen Eingänge, aber es gibt zusätzlich noch den Zustand der vierzehn digitalen Eingänge aus.

Dazu definiert es wie gewohnt die Konstante BAUD_RATE und darüber hinaus die Konstanten MAX_ANALOG_PINS und MAX_DIGITAL_PINS, die jeweils die maximale Anzahl an analogen und digitalen Pins speichern. Die setup-Funktion initialisiert die Kommunikation auf der seriellen Schnittstelle.

Bahnbrechende Neuerungen gibt es dann aber in der Funktion loop. Statt dieselben Anweisungen immer und immer wieder zu wiederholen, um die aktuellen Werte der analogen Eingänge auszugeben, nutzt sie eine for-Schleife. Schleifen sind ein wichtiges Hilfsmittel in Programmen, denn sie helfen bei der Automatisierung von sequenziellen Abläufen. Immer dann, wenn ein und dieselbe Tätigkeit immer und immer wieder ausgeführt werden muss, sind Schleifen das Mittel der Wahl.

Die Struktur einer for-Schleife sieht wie folgt aus:

```
for (<Initialisierung>; <Bedingung>;
<Aktualisierung>) {
      // Anweisungen, die in der Schleife
ausgeführt werden sollen
  }
```

In der Initialisierung können beliebig viele Variablen definiert werden, die in der Schleife verwendet werden sollen. Im vorliegenden Fall ist es die Variable analog_pin. Sie hat den Datentypen unsigned char, kann also Werte von 0 bis 255 aufnehmen und wird mit dem Wert 0 initialisiert. Die Va-

Der serielle Monitor zeigt die Werte aller analogen und digitalen Eingänge.

riable dient dazu, die analogen Pins abzuzählen.

In der Bedingung der Schleife wird daher geprüft, ob `analog_pin` kleiner ist als `MAX_ANALOG_PINS`. Die Schleife läuft so lange, wie diese Bedingung wahr ist, das heißt, die Anweisungen in der Schleife werden ausgeführt, so lange `analog_pin` den Wert 0, 1, 2, 3, 4 oder 5 hat.

Damit sich aber der Wert von `analog_pin` überhaupt verändert, muss er in jedem Schleifendurchlauf angepasst werden. Dazu dient die Aktualisierung in der for-Schleife. Die Anweisung `analog_pin++` erhöht den Wert von `analog_pin` um 1.

Die Anweisungen in der Schleife verwenden dann jeweils den aktuellen Wert von `analog_pin`. Im ersten Schleifendurchlauf hat `analog_pin` beispielsweise den Wert 0 und so gibt der Code innerhalb der Schleife den Wert des analogen Eingangs A0 auf der seriellen Schnittstelle aus.

All das wird völlig analog in der zweiten for-Schleife für die digitalen Pins ausgeführt. Nur am Ende der zweiten Schleife findet sich eine Besonderheit. Damit die auszugebende Zeile nicht mit einem überflüssigen Komma endet, stellt eine if-Anweisung sicher, dass kein Komma für den letzten digitalen Eingang ausgegeben wird.

Die Ausgabe des Programms auf dem seriellen Monitor sieht recht wüst aus. Sie kann aber durchaus dabei helfen, sich mit neuen Sensoren vertraut zu machen und beispielsweise die passende Spannung für AREF (siehe Seite 6) zu finden. *—dab*

Temperaturen messen

Aus dem Arduino wird dank eines simplen ICs nun ein Thermometer mit hoher Genauigkeit und Verlaufsaufzeichnung.

Temperatursensoren sind nützliche Bauelemente, die sich einfach verwenden lassen und nicht viel kosten. Es gibt eine große Vielfalt an Sensoren und sie unterscheiden sich in verschiedenen Eigenschaften. Wichtig ist zum Beispiel der Messbereich, also die minimale und die maximale Temperatur, die ein Sensor messen kann. Manche können nur Temperaturen oberhalb des Gefrierpunktes messen und auch bei der maximal messbaren Temperatur gibt es große Unterschiede.

Ferner unterscheiden sich die Sensoren bezüglich ihrer Verwendung. Besonders einfach sind analoge Temperatursensoren, die je nach gemessener Temperatur eine andere Spannung zurückliefern. Je höher die Spannung ist, um so höher ist in der Regel die gemessene Temperatur. Solche Sensoren lassen sich direkt mit einem analogen Eingang des Arduino verbinden. Die gemessene Spannung muss dann nur noch mit einer Formel, die man dem Datenblatt des Sensors entnehmen kann, in eine Temperatur umgerechnet werden.

Es gibt aber auch weitaus komplexere Sensoren, die über ein serielles Protokoll mit dem Arduino kommunizieren. Besonders beliebt sind dabei die Protokolle I2C (Inter-Integrated Circuit) und 1-Wire (oder auch „One-Wire“).

Warum nicht einfach?

Der große Vorteil der komplexeren Sensoren ist, dass eine große Anzahl davon gleichzeitig mit dem Arduino verbunden werden kann, denn sie können alle über denselben Pin kommunizieren. Auch funktionieren sie mit langen Kabeln, sodass sie leicht ein ganzes Temperatursensornetz bilden können. Mit analogen Sensoren funktioniert so etwas nicht ohne weiteres.

In diesem Artikel kommt der 1-Wire-Sensor DS18B20 zum Einsatz. Er ist sehr populär, günstig zu haben und es gibt ihn sogar in einer wasserdichten Bauform. Somit eignet er sich zum Beispiel auch als elektronisches Aquarium-Thermometer.

Mogelpackung

1-Wire ist eine serielle Schnittstelle, die mit minimaler Hardware auskommt. Im Gegensatz zum etwas irreführenden Namen benötigt sie aber mindestens zwei Leitungen, nämlich einen Masse-Anschluss (Minus) und eine Datenleitung. In vielen Fällen reicht das aus, aber manchmal wird noch eine dritte Leitung für eine zusätzliche Spannungsversorgung benötigt.

Der DS18B20 ist bei der Stromversorgung nicht wählerisch und kommt mit Spannungen zwischen 3 V und 5,5 V zurecht. Wie die meisten 1-Wire-Geräte kann er auf mehr als eine Art angeschlossen werden. Im Normalmodus (englisch: normal mode) wird das Gerät an eine eigene Spannungsversorgung angeschlossen und es werden drei Anschlussdrähte benötigt. Dies ist dann sinnvoll, wenn mehr als ein Sensor an den Bus angeschlossen wird oder wenn die Verbindung zwischen Rechner und Sensor besonders lang ist. An einen 1-Wire-Bus lassen sich bis zu 100 Geräte anschließen und auch Kabellängen von bis zu 100 Metern sind machbar.

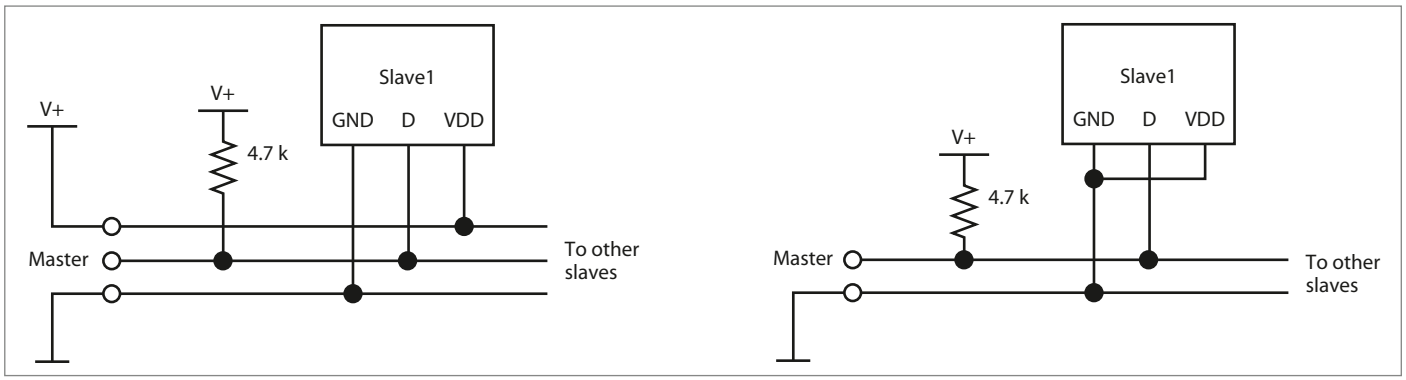
Im sogenannten parasitären Modus (englisch: parasite mode) reichen hingegen zwei Leitungen, nämlich eine für die Masse und eine für das Signal. In diesem Fall bezieht der Sensor die benötigte Energie aus der Datenleitung. Dazu enthält das Gerät einen eigens dafür vorgesehenen Kondensator.

Für das Beispiel-Projekt wird der Sensor im „normal mode“ angeschlossen. Für den reibungslosen Betrieb ist ein Pullup-Widerstand von 4,7 kOhm zwischen der Spannungsversorgung und der Signal-Leitung notwendig. Aufmerksamkeit ist bei der Verkabelung eines wasserdichten Sensors gefragt, denn stellenweise haben deren Drähte recht abenteuerliche Farben. Beim Testgerät war zum Beispiel die Spannungsversorgung rot, das Signalkabel gelb und die Masse blau.

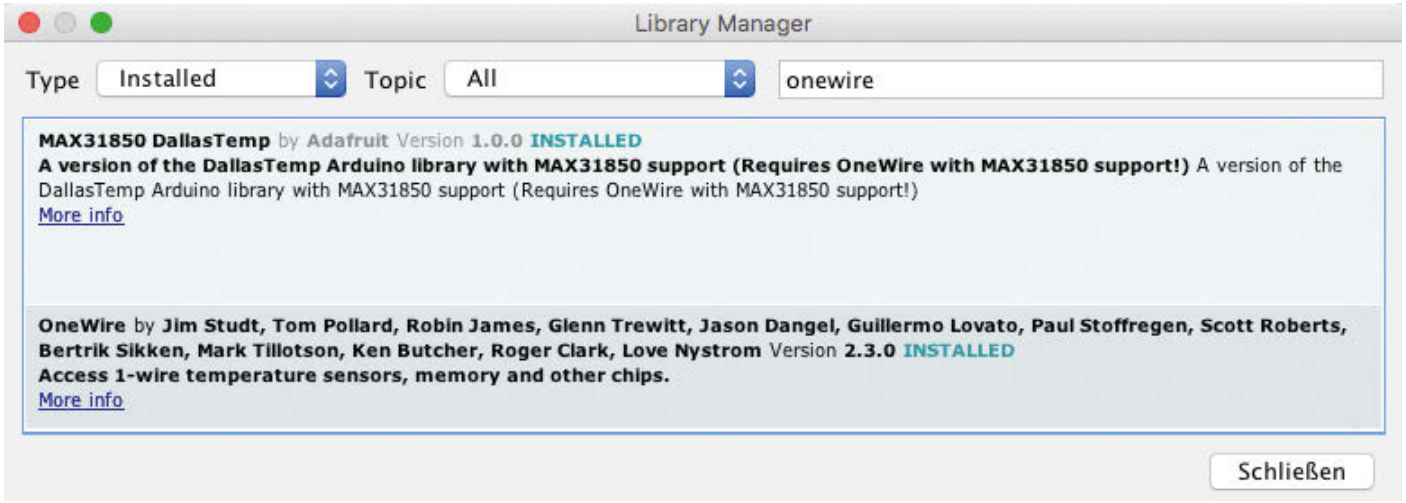
Bibliotheken installieren

Die Implementierung des eigentlichen 1-Wire-Protokolls ist eine knifflige Angelegenheit und setzt eine Menge Erfahrung im Bereich der hardwarenahen Programmierung voraus. Dankenswerter Weise haben andere diese Arbeit bereits erledigt und stellen die Früchte ihrer Anstrengungen als freie Bibliothek namens OneWire zur Verfügung.

Analoges gilt für die Kontrolle des DS18B20-Sensors, denn auch dazu ist die Implementierung eines speziellen Protokolls notwendig. Das steckt in der Bibliothek MAX31850 DallasTemp.



Der 1-Wire-Bus: links im Normal Mode, rechts im Parasite Mode



Diese Bibliotheken muss man installieren.

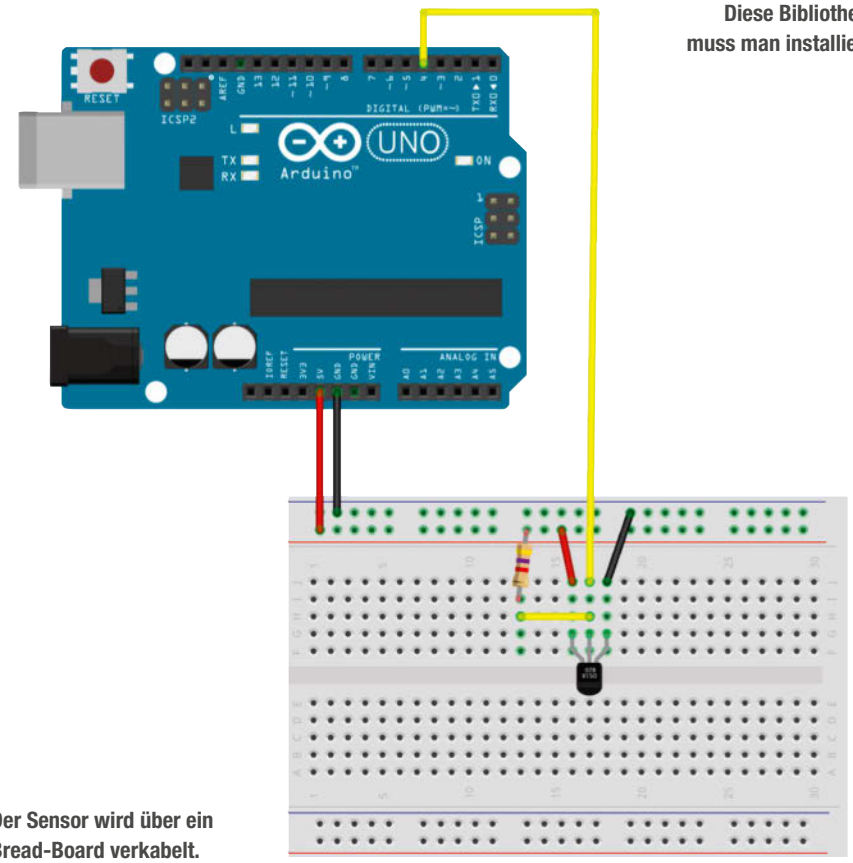
Die Installation kann komfortabel über den Library-Manager der Arduino-Umgebung erfolgen. Allerdings ist hier Aufmerksamkeit gefragt, denn es gibt mehrere Bibliotheken, die bei der Suche nach OneWire beziehungsweise MAX31850 auftauchen. Der Screenshot zeigt, welches die richtigen sind.

Los geht's

Nachdem alle Vorkehrungen getroffen wurden, kann der Sensor endlich abgefragt werden. Das Programm `Temperatur_Messen` zeigt, wie das geht.

Zuerst bindet es die zuvor installierten Bibliotheken OneWire und DallasTemperature ein. Anschließend wird die Konstante `BAUD_RATE` definiert, weil im weiteren Verlauf Informationen über die serielle Schnittstelle ausgegeben werden. Die Konstante `ONE_WIRE_BUS` legt fest, über welchen digitalen Pin der Temperatursensor DS18B20 angesprochen werden soll. In diesem Fall ist es Pin 4. Es kann aber jeder digitale Pin sein.

Das Wort Bus steht hier übrigens nicht für ein öffentliches Verkehrsmittel, sondern für einen Daten-Bus, also eine Leitung, die sich mehrere Geräte teilen können. Prinzipiell können so 100 DS18B20-Sensoren an einen



Der Sensor wird über ein Bread-Board verkabelt.

Temperatur_messen

```
 1 #include <OneWire.h>
 2 #include <DallasTemperature.h>
 3
 4 const unsigned long BAUD_RATE = 9600;
 5 const unsigned char ONE_WIRE_BUS = 4;
 6
 7 OneWire oneWire(ONE_WIRE_BUS);
 8 DallasTemperature sensoren(&oneWire);
 9
10 void setup() {
11   sensoren.begin();
12   Serial.begin(BAUD_RATE);
13   Serial.print("Anzahl Sensoren: ");
14   Serial.println(sensoren.getDeviceCount());
15   Serial.print("Modus ist: ");
16   if (sensoren.isParasitePowerMode()) {
17     Serial.println("Parasitär");
18   } else {
19     Serial.println("Normal");
20   }
21 }
22
23 void loop() {
24   sensoren.requestTemperatures();
25   float temperatur = sensoren.getTempCByIndex(0);
26   Serial.println(temperatur);
27   delay(1000);
28 }
```

Die Ausgabe auf den seriellen Monitor zeigt den Status und die Temperatur.

einzigen Arduino angeschlossen werden und sie könnten sich alle den Pin 4 teilen. Der Trick bei der ganzen Sache ist, dass jeder Sensor eine eindeutige Adresse hat. Diese Adresse kann der Arduino dann gezielt ansprechen, wenn er etwas von einem bestimmten Sensor wissen möchte.

Nach den Konstanten werden zwei globale Variablen definiert. Die erste heißt oneWire und ist vom Datentyp OneWire. Dieser Typ wird von der Bibliothek OneWire definiert und hilft dem Arduino, mit 1-Wire-Geräten zu kommunizieren. Die Variable oneWire versteckt also die komplexe serielle Kommunikation mit beliebigen 1-Wire-Geräten hinter einer einfach zu nutzenden Fassade.

Die nächste Variable heißt sensoren und ist vom Typ DallasTemperature. Sie repräsentiert alle Temperatur-Sensoren, die gerade mit dem Arduino verbunden sind. Dazu muss der Variablen bei der Definition aber mitgeteilt werden, über welchen 1-Wire-Bus sie die Sensoren finden kann. Daher wird bei der Definition das zuvor angelegte Objekt oneWire übergeben. Genauer gesagt wird die Adresse des Objekts im Speicher des Arduino übergeben. Dafür steht das &-Zeichen vor dem Namen. Das ist der sogenannte Address-of-Operator. Damit ist es dem sensoren-Objekt möglich, Änderungen am oneWire-Objekt vorzunehmen, wenn es nötig sein sollte.

Diese Art der Initialisierung ist nicht unüblich, denn die beiden Variablen oneWire und sensoren repräsentieren einen sogenannten Protokoll-Stapel. oneWire kümmert sich um die ganz niedrigen Funktionen, die gebraucht werden, um die physikalische Kommunikation zwischen Arduino und 1-Wire-Geräten zu ermöglichen. Die Variable sensoren baut darauf auf und nutzt die Funktionen von oneWire, um höhere Operationen zu realisieren.

Die setup-Funktion hat diesmal mehr zu tun als sonst und startet mit dem Aufruf von sensoren.begin. Das sorgt dafür, dass das Objekt sensoren nach allen Sensoren sucht, die mit Pin 4 verbunden sind. Anschließend wird die serielle Kommunikation per Serial.begin initialisiert.

Zu diesem Zeitpunkt sind in der Variablen sensoren bereits alle Informationen über alle angeschlossenen Sensoren gespeichert. Insbesondere ist bekannt, wie viele Sensoren verfügbar sind und diese Information wird jetzt auf der seriellen Schnittstelle ausgegeben. Dazu dient die Funktion getDeviceCount.

Ganz analog gibt das Programm aus, ob der 1-Wire-Bus im „normal mode“ oder im „parasite mode“ betrieben wird. Dies stellt die Funktion isParasiteMode fest.

Die Funktion loop ist etwas knapper gehalten und weist erst einmal das sensoren-Objekt an, die aktuellen Temperaturen an allen angeschlossenen Sensoren zu messen. Das erfolgt mit der Funktion requestTemperatures. Danach ermittelt getTempCByIndex die Temperatur am ersten Sensor in der Reihe, was in diesem Fall der einzige Sensor ist. Die Nummerierung der Sensoren beginnt bei 0, sodass 0 für den ersten Sensor steht.

Die Funktion liefert die Temperatur als float-Wert und in der Einheit Celsius zurück. Ersetzt man das C im Funktionsnamen durch ein F, wird die Temperatur in Fahrenheit ermittelt.

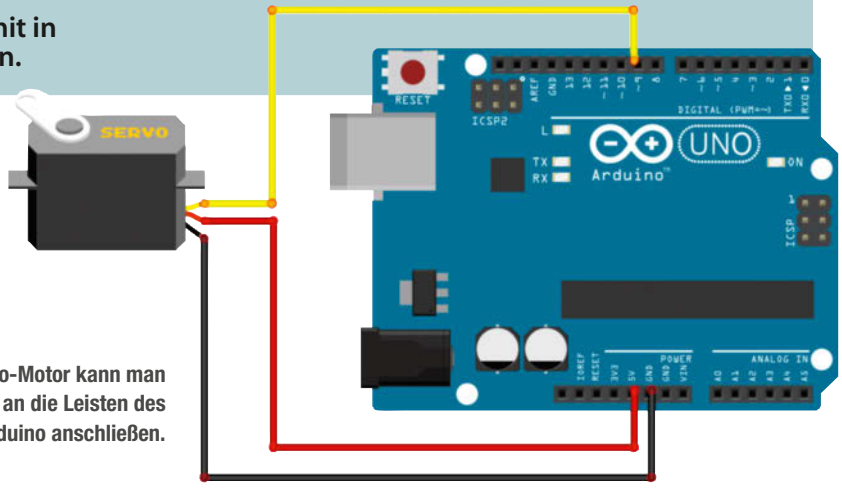
Am Ende wird die Temperatur auf der seriellen Schnittstelle ausgegeben und mit Hilfe der delay-Funktion eine Pause von einer Sekunde eingelegt, um die Ausgabe des Programms etwas zu verlangsamen. *—dab*

Motoren steuern

Der Arduino kann Motoren ansteuern und damit in der physischen Welt Dinge in Bewegung setzen.

LEDs zum Leuchten zu bringen und die aktuelle Temperatur zu messen, sind zumeist auch für erfahrene Programmierer ein erhebendes Gefühl. Die Interaktion von Software und Hardware birgt eine ganz eigene Faszination und macht die Arbeit mit dem Arduino so interessant.

So richtig hoch schlägt das Maker-Herz aber erst, wenn sich etwas bewegt. Spätestens wenn Motoren ins Spiel kommen, kann sich kaum einer dem Zauber des „physical computing" entziehen und der Arduino macht die Integration von Motoren zum Kinderspiel.



Den Servo-Motor kann man direkt an die Leisten des Arduino anschließen.

Die Qual der Wahl

Bastlern steht eine große Auswahl an Motoren zur Verfügung. Die Abbildung zeigt von rechts nach links erst einmal einen Gleichstrommotor. Diese Motoren sind besonders einfach zu kontrollieren und sie haben nur zwei Anschlüsse. Je mehr Strom am Motor ankommt, um so schneller dreht er. Auch die Laufrichtung des Motors lässt sich umkehren und solche Motoren eignen sich zum Beispiel für Bohrmaschinen und ferngesteuerte Autos.

Als Nächstes gibt es Schritt-Motoren, die sich in genau festgelegten Schritten kontrollieren lassen. Sie kommen unter anderem in Diskettenlaufwerken und Druckern zum Einsatz und ihr charakteristisches Klacken ist unverkennbar.

Servo-Motoren sind im Hobby-Bereich und insbesondere unter Modellbauern sehr beliebt. Sie lassen sich leicht steuern und sind in zwei Ausprägungen erhältlich. Die eine lässt Drehungen im Bereich von 0 bis 180 Grad zu und die andere erlaubt das volle Spektrum von 360 Grad.

Verschiedene Arten von Motoren: Servo-, Schritt- und Gleichstrommotor

Schritt für Schritt

Für erste Experimente am Arduino eignen sich kleine Servo-Motoren sehr gut, weil sie sich direkt mit dem Arduino steuern lassen. Gute Kandidaten sind zum Beispiel der Hitec HS-311, Hitec HS-322HD oder der TowerPro SG92R. Zur Not tun es auch die Billig-Servos einschlägiger Elektronikketten. Sie können mit nur drei Kabeln mit dem Board verbunden werden.

Das Masse-Kabel (schwarz) des Motors gehört an einen der GND-Pins des Arduino. Der Stromanschluss wird mit dem 5-V-Pin verbunden und das Signal-Kabel mit einem der digitalen Pins. In diesem Fall ist es Pin 9.

Das Beispiel-Programm dieses Artikels erlaubt die Steuerung eines Servos über die serielle Schnittstelle. Nutzer können einen Winkel zwischen 0 und 180 Grad eingeben und der Motor fährt dann die angegebene Position an.

Der Code

Um Servo-Motoren zu kontrollieren, wird erneut eine Bibliothek benötigt. Die Servo-Bibliothek gehört zu den Standard-Bibliotheken des Arduino und muss nicht separat installiert werden. Trotzdem ist es notwendig, sie mit einer #include-Anweisung einzubinden, damit die Servo-Funktionen zur Verfügung stehen.

Das Programm definiert im Anschluss eine Reihe von Konstanten. Weil sowohl Eingaben als auch Ausgaben über die serielle Schnittstelle erfolgen, legt `BAUD_RATE` die Übertragungsgeschwindigkeit fest. `MOTOR_PIN` enthält die Nummer des Pins, über den der Motor kontrolliert wird. Übrigens kann dies jeder der digitalen Pins 2 bis 13 sein (die Pins 0 und 1 sind für die serielle Kommunikation reserviert). Die Servo-Bibliothek nutzt nämlich nicht die PWM-Funktionen der Arduino-Hardware, sondern simuliert das entsprechende Verhalten in Software. Daher funktionieren Servos nicht nur an den PWM-Pins.

Die Konstante `MOTOR_PAUSE` enthält einen Wert, der in Millisekunden gemessen wird. Er legt die Pause fest, die der Arduino einlegen muss, nachdem er ein Kommando an den Motor gesendet hat. Software zur Steuerung eines Servo-Motors muss ein paar Gesetzmäßigkeiten der Physik berücksichtigen. Beispielsweise reagiert ein Servo-Motor im Vergleich zur Geschwindigkeit elektrischer Signale verhältnismäßig träge. Daher muss die Software, nachdem sie ein Kommando an den Motor gesendet hat, ein wenig warten. Der Motor braucht halt eine Weile, bis er eine neue Position angefahren hat. 15 Millisekunden reichen hier aus.

Darüber hinaus haben Servo-Motoren in den Grenzbereichen, also nahe bei 0 Grad und nahe bei 180 Grad, schon mal ein paar kleine Ungenauigkeiten. Die können dann zu einem leichten Brummen oder Klacken des Motors führen, weil der Servo versucht, eine Position anzufahren, die er nicht erreichen kann. Mit den Konstanten `MIN_WINKEL` und `MAX_WINKEL` kann der tatsächliche maximale Aktionsradius definiert werden.

Nach den Konstanten führt das Programm drei globale Variablen ein. Die Va-

riable servo hat den Datentyp Servo und dient im weiteren Verlauf zur Kontrolle des Motors. neuer_winkel hat den Typen bool, der nur die beiden Werte true (wahr) oder false (falsch) annehmen kann. Er eignet sich prima zur Steuerung von Ja-/Nein-Entscheidungen. In diesem Fall zeigt der Wert von neuer_winkel an, ob der Benutzer einen neuen Winkel über die serielle Schnittstelle übertragen hat und der Motor entsprechend neu positioniert werden muss. Schließlich enthält aktueller_winkel den Winkel, der zuletzt über die serielle Schnittstelle übertragen wurde.

Auf all die Konstanten und Variablen folgt die setup-Funktion und sie enthält keine großen Überraschungen. Sie initialisiert die serielle Schnittstelle und ruft die attach-Funktion auf der servo-Variablen auf, um festzulegen, über welchen Pin der Servo-Motor kontrolliert werden soll. Anschließend verwendet sie die write-Funktion, um den Motor so weit wie möglich nach links zu fahren. Um dem Motor ausreichend Zeit zu geben die Position anzufahren, wartet das Programm mittels der delay-Funktion.

Serielle Daten einmal anders

Spannend wird es in der Funktion serialEvent. Sie liest den einzustellenden Winkel mittels Serial.parseInt und setzt die Variable neuer_winkel auf true, wenn tatsächlich ein neuer Winkel über die serielle Schnittstelle empfangen wurde. Aber was genau ist die Funktion serialEvent und wer ruft sie auf?

Weil die Verarbeitung von Daten, die über die serielle Schnittstelle übertragen werden, eine häufige Anwendung ist, hat das Arduino-Team dafür eine eigene Funktion spendiert. Diese Funktion heißt serialEvent und sie agiert auf demselben Niveau wie setup und loop. Etwas vereinfacht sieht die Haupt-Logik des Arduino wie folgt aus:

```
setup();
  for (;;) {
    loop();
    if (serialAvailable > 0) {
      serialEvent();
    }
  }
```

Servo_seriell_steuern

```
#include <Servo.h>

const unsigned long BAUD_RATE = 9600;
const unsigned char MOTOR_PIN = 9;
const unsigned int MOTOR_PAUSE = 15;
const unsigned int MIN_WINKEL = 2;
const unsigned int MAX_WINKEL = 180;

Servo servo;
bool neuer_winkel = false;
unsigned int aktueller_winkel = 0;

void setup() {
  Serial.begin(BAUD_RATE);
  servo.attach(MOTOR_PIN);
  delay(MOTOR_PAUSE);
  servo.write(1);
  delay(MOTOR_PAUSE);
}

void serialEvent() {
  aktueller_winkel = Serial.parseInt();
  if (Serial.read() == '\n') {
    neuer_winkel = true;
  }
}

void loop() {
  if (neuer_winkel) {
    neuer_winkel = false;
    if (aktueller_winkel < MIN_WINKEL) {
      aktueller_winkel = MIN_WINKEL;
    } else if (aktueller_winkel > MAX_WINKEL) {
      aktueller_winkel = MAX_WINKEL;
    }
    Serial.print(aktueller_winkel);
    Serial.println(" Grad.");
    servo.write(aktueller_winkel);
    delay(MOTOR_PAUSE);
  }
}
```

Sobald der Arduino startet, ruft er erst einmal die setup-Funktion des Programms auf. Anschließend beginnt er mit der Anweisung for(;;) eine Endlosschleife, das heißt, alle nachfolgenden Anweisungen werden immer und immer wieder ausgeführt. In dieser Endlosschleife wird die Funktion loop aufgerufen. Darüber hinaus kommt aber auch die Funktion serialEvent zum Zug, wenn gerade neue Daten an der seriellen Schnittstelle anliegen.

Die Funktion serialEvent kann also eingesetzt werden, um die Verarbeitung der seriellen Kommunikation und die eigentliche Logik des Programms sauber zu trennen. Im vorliegenden Fall liest loop zum Beispiel keine Daten von der seriellen Schnittstelle und prüft nicht einmal, ob überhaupt Daten auf der Schnittstelle auf ihre Abholung warten.

loop prüft mittels der Variablen neuer_winkel, die von der Funktion serialEvent bei Bedarf gesetzt wird, ob überhaupt etwas zu tun ist. Ist dies der Fall, steht in der Variablen aktueller_winkel der neu einzustellende Winkel für den Servo-Motor.

Zuerst wird neuer_winkel auf false gesetzt, damit nicht derselbe Winkel immer wieder angefahren wird. Danach stellt die Funktion sicher, dass sich aktueller_winkel im festgelegten Bereich befindet. Schließlich wird der neue Winkel auf der seriellen Schnittstelle ausgegeben und dann mit der Funktion write an den Motor übermittelt. Nach einer kurzen Pause geht es dann weiter im Programm.

Der Screenshot links zeigt eine typische Ausgabe des Programms. Für erste Tests ist es hilfreich, einen kleinen Pfeil aus Papier, Schaltdraht oder aus einer Büroklammer am Motor zu befestigen. So lässt sich der Aktionsradius am besten beobachten und den meisten Motoren liegt ein wenig Befestigungsmaterial bei.

Allerdings darf die Last am Motor keinesfalls zu groß werden, weil der Motor sonst zu viel Strom zieht und den Arduino überlastet. Wie dieses Problem gelöst werden kann, zeigt der nächste Artikel. *—dab*

Großverbraucher

Durch einen externen Transistor versetzen wir den Arduino in die Lage, auch große Stromverbraucher zu steuern.

Die Anschlüsse eines MOSFET heißen Gate, Drain und Source.

Prinzipiell ist die Kontrolle von Motoren, insbesondere von Servo-Motoren, mit den Arduino-Bibliotheken eine einfach Angelegenheit. Leider ist die Welt der Physik und der Elektronik nicht ganz so einfach wie die Software-Welt.

Die Pins des Arduino liefern Strom mit einer maximalen Stärke von 20 mA (Milliampere) bei einer Spannung von 5 V, wenn der Arduino über einen USB-Anschluss gespeist wird. Das reicht, um einen kleinen Servo-Motor, der keine nennenswerte Last trägt, anzutreiben. Sobald es etwas mehr sein soll, kommt der Arduino an seine Grenzen. Er kann die Last nicht mehr bewegen und im Extremfall sogar Schaden erleiden.

Mehrere und größere Motoren können und sollten mit einem „nackten" Arduino und der USB-Stromversorgung daher nicht angetrieben werden. Es nutzt aber auch nicht viel, den Arduino einfach mit einer stärkeren Stromversorgung zu verbinden, weil es auch hier eine Obergrenze gibt.

Wer packt mit an?

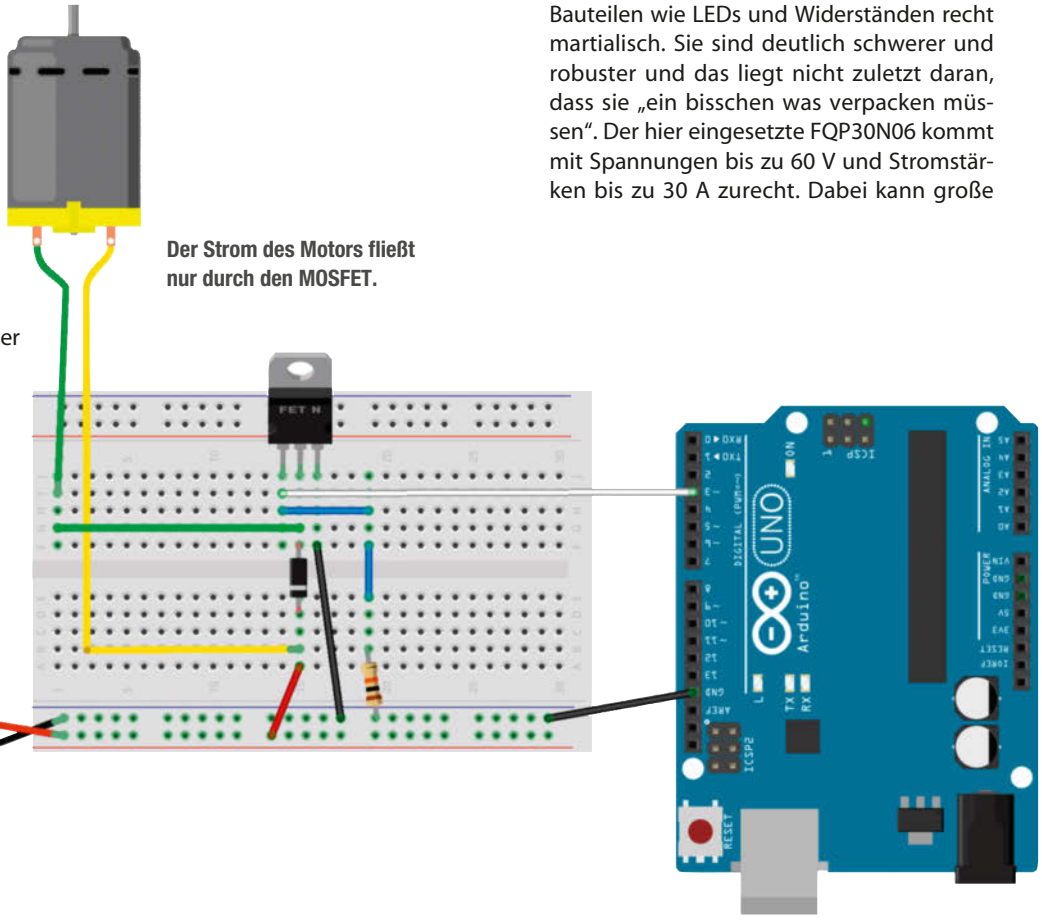
Im Folgenden wird die Geschwindigkeit eines Gleichstrommotors (Johnson 20703) stufenlos mit dem Arduino geregelt. Gleichstrommotoren werden unter anderem gerne im Modellbau eingesetzt, wo sie ferngesteuerte Autos oder Boote antreiben.

Gleichstrommotoren haben nur zwei Anschlüsse, an die zuerst zwei ausreichend lange Drähte gelötet werden müssen. Häufig – aber nicht immer – ist einer der beiden Anschlüsse als Plus-Pol gekennzeichnet. Einen Standard gibt es hier nicht. Manchmal ist es ein Pluszeichen, manchmal ein roter oder weißer Punkt.

Wird der Plus-Pol mit der Spannungsquelle und der andere Anschluss mit der Masse verbunden, dreht der Motor rechts herum. So lässt sich die Polung auch leicht mit einer 9-V-Batterie prüfen. Einfach die beiden Anschlüsse der Batterie mit den Anschlüssen des Motors verbinden und sehen, in welche Richtung er dreht. Werden die beiden Anschlüsse vertauscht, dreht der Motor anders herum. Es empfiehlt sich daher, die Kabel, die an den Motor gelötet werden, zu kennzeichnen oder gleich zwei unterschiedliche Farben zu verwenden.

Motoren direkt an den Arduino anzuschließen ist keine gute Idee, denn die fordern schon bei wenig Beanspruchung mehr Strom, als der Arduino über einen USB-Anschluss liefern kann. Eine externe Stromquelle ist daher unabdingbar und für erste Experimente eignet sich eine handelsübliche 9-V-Batterie mit einem Batterie-Clip.

Die Batterie speist den Motor, aber es verbleibt noch ein Problem: Wie wird der Motor ein- und ausgeschaltet und wie lässt sich seine Geschwindigkeit regulieren? Ab einer gewissen Stärke dürfen die Ströme, die den Motor antreiben, nicht mehr durch den Arduino fließen. Es wird also ein Bauteil benötigt, das der Arduino mit seinen beschränkten Mitteln kontrollieren kann und das gleichzeitig mit großen Strömen zurechtkommt.



Der Strom des Motors fließt nur durch den MOSFET.

So etwas gibt es tatsächlich und es hat den blumigen Namen MOSFET (Metall-Oxid-Halbleiter-Feldeffekttransistor). Ein MOSFET ist also in erster Linie ein Transistor und hat in der Regel drei Anschlüsse: G (Gate, deutsch: Steuerelektrode), D (Drain, deutsch: Abfluss) und S (Source, deutsch: Quelle).

Im Kern fungiert ein MOSFET als ein digitaler Schalter, der große Lasten zwischen D- und S-Pin schalten kann, ohne dass der Arduino mit diesen Lasten in Berührung kommt. Der Arduino nutzt lediglich kleine Ströme, um den MOSFET über den G-Pin ein- und auszuschalten. Dazu ist es wichtig, dass der eingesetzte MOSFET zur Familie der Logic-Level MOSFETS (auch: Low-Gate-Voltage MOSFET) gehört. Nur diese lassen sich mit den geringen Spannungen des Arduino auch tatsächlich schalten.

MOSFETs wirken im Vergleich zu anderen Bauteilen wie LEDs und Widerständen recht martialisch. Sie sind deutlich schwerer und robuster und das liegt nicht zuletzt daran, dass sie „ein bisschen was verpacken müssen". Der hier eingesetzte FQP30N06 kommt mit Spannungen bis zu 60 V und Stromstärken bis zu 30 A zurecht. Dabei kann große

Motor_drehen

```
const unsigned long BAUD_RATE = 9600;
const unsigned int MOTOR_PIN = 3;
const unsigned int MAX_GESCHWINDIGKEIT = 255;

bool neue_geschwindigkeit = false;
int geschwindigkeit = 0;

void setup() {
  Serial.begin(BAUD_RATE);
  pinMode(MOTOR_PIN, OUTPUT);
}

void serialEvent() {
  geschwindigkeit = Serial.parseInt();
  if (Serial.read() == '\n') {
    neue_geschwindigkeit = true;
  }
}
void loop() {
  if (neue_geschwindigkeit) {
    neue_geschwindigkeit = false;
    geschwindigkeit = min(geschwindigkeit, MAX_GESCHWINDIGKEIT);
    Serial.print("Geschwindigkeit: ");
    Serial.println(geschwindigkeit);
    analogWrite(MOTOR_PIN, geschwindigkeit);
    delay(1000);
  }
}
```

Hitze entstehen und unter Umständen muss ein solcher Baustein mit einem Wärmeableitblech (english: heatsink) versehen werden.

Die Schaltung

Die Schaltung ist recht komplex, lässt sich aber stückweise leicht entschlüsseln. Auffällig ist zunächst die 9-V-Batterie, die über einen Batterie-Clip mit dem Breadboard verbunden wird. Diese Batterie speist den Motor und ist von der Stromversorgung des Arduino getrennt. Der wird weiterhin über den USB-Anschluss versorgt und teilt sich lediglich die Masse-Leitung mit der Batterie.

Der Masse-Anschluss der Batterie ist ebenfalls direkt mit dem Motor verbunden. Die eigentliche Spannung nimmt hingegen einen kleinen Umweg über eine Diode. Diese dient dem Schutz des MOSFET, denn Motoren können beim Abschalten große Induktionsspannungen zurückliefern. Die Geschwindigkeit des Motors wird später mittels der Pulsweiten-Modulation geregelt werden und bei diesem Verfahren erfolgt das Abschalten der Spannung naturgemäß relativ häufig. Daher müssen etwaige Induktionsspannungen abgefangen werden. Dafür ist die Diode (1N-4001) zuständig, denn sie lässt Strom nur in eine Richtung, die sogenannte Durchlaufrichtung, passieren. Strom, der die Diode entgegen der Durchlaufrichtung passieren möchte, wird in Wärme umgewandelt.

Schließlich wird noch Pin 3 mit dem Gate-Pin des MOSFET verbunden. Der Pulldown-Widerstand (10 kOhm) zwischen dem Signal-Pin 3 und dem Gate-Pin des Transistors verhindert, dass kleine Schwankungen den Transistor ungewollt aktivieren und somit den Motor einschalten.

Der Code

Die Geschwindigkeit des Motors zu kontrollieren, ist vergleichsweise einfach und das entsprechende Programm Motor_drehen keine 30 Zeilen lang. Es ähnelt sehr dem Programm zur Kontrolle eines Servo-Motors und erlaubt es, die Geschwindigkeit des Motors über die serielle Schnittstelle einzustellen.

Es beginnt mit der Definition der Konstanten `BAUD_RATE` und setzt danach die Konstante `MOTOR_PIN` auf den Wert 3. Das heißt, dass die Motor-Geschwindigkeit über den Pin 3 gesteuert wird. Für dieses Projekt ist es wichtig, dass der Kontroll-Pin einer der PWM-Pins des Arduino ist.

`MAX_GESCHWINDIGKEIT` legt die maximale Geschwindigkeit fest, die über die serielle Schnittstelle übermittelt werden kann. Erlaubt sind Werte von 0 bis 255.

Die Variable `neue_geschwindigkeit` zeigt an, ob ein neuer Geschwindigkeitswert auf der seriellen Schnittstelle empfangen wurde. In diesem Fall hat sie den Wert `true`, andernfalls den Wert `false`. Sie wird später in der Funktion `serialEvent` gesetzt.

Die Funktion `setup` ist kurz und initialisiert die serielle Schnittstelle und versetzt den Pin zur Kontrolle des Motors in den Ausgabe-Modus. Auch die `serialEvent`-Funktion, die immer dann aufgerufen wird, wenn neue Daten auf der seriellen Schnittstelle empfangen wurden, hat nicht viel zu tun. Sie liest die neu einzustellende Geschwindigkeit mittels `parseInt` in die Variable `geschwindigkeit` und setzt `neue_geschwindigkeit` auf `true`, wenn ein Zeilenumbruch-Zeichen empfangen wurde.

Genau diese Variable wird in der `loop`-Funktion geprüft. Wenn sie den Wert `true` hat, wird sie erst einmal auf den Wert `false` zurückgesetzt, damit neue Kommandos, die auf der seriellen Schnittstelle übertragen werden, korrekt erkannt werden. Anschließend sorgt `loop` mit der `max`-Funktion dafür, dass die Variable `geschwindigkeit` immer Werte zwischen 0 und 255 enthält. Dann wird der Wert der Variablen auf der seriellen Schnittstelle ausgegeben.

Den Kern des ganzen Programms bildet der Aufruf der Funktion `analogWrite`. Sie simuliert ein analoges Ausgabesignal mithilfe der Pulsweitenmodulation. `analogWrite` erwartet Eingabewerte zwischen 0 und 255 und wandelt sie in entsprechende Ausgangsspannungen zwischen 0 V und 5 V um. Bei einem Wert von 0 ruht der Motor und bei einem Wert von 255 dreht er sich mit maximaler Geschwindigkeit.

Die `loop`-Funktion endet mit einem Aufruf von `delay`, um sicherzustellen, dass jede neu eingestellte Geschwindigkeit für mindestens eine Sekunde aktiv ist.

Nachdem das Programm auf den Arduino geladen wurde, kann die 9-V-Batterie angeschlossen werden. Dann kann man den Motor über den seriellen Monitor auf Touren bringen.

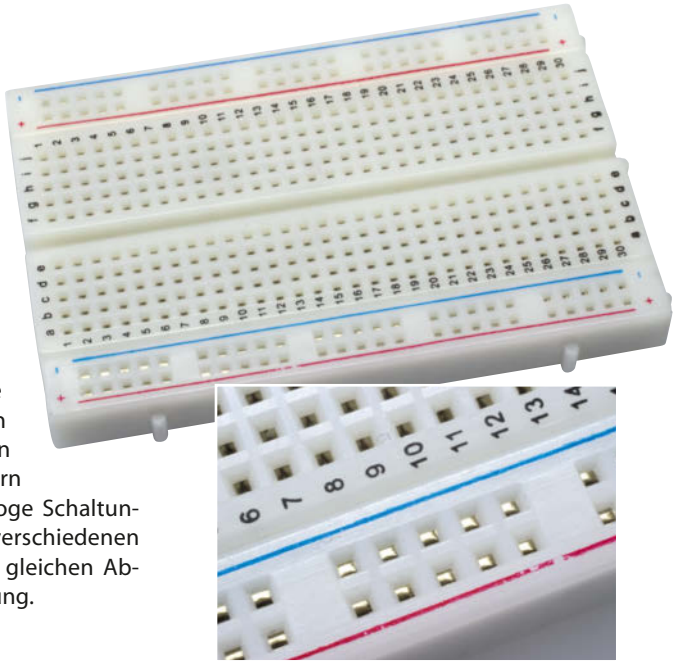
Fazit

Größere Lasten mit dem Arduino zu schalten ist prinzipiell kein Problem. Das beschränkt sich nicht auf Motoren, sondern gilt auch für energiehungrige LED-Anwendungen oder die Steuerung von Relais. Allerdings stellt die Kontrolle solcher Geräte ganz andere Ansprüche als die bisherigen Projekte. Reines Probieren ohne ein gesundes theoretisches Fundament führt hier selten zum Ziel, sondern schnell zu durchgebrannten Bauteilen.

Wer dennoch vorhat, Motoren oder andere Lasten mit einem Arduino zu kontrollieren, sollte in jedem Fall entsprechende Shields einsetzen und von den Erfahrungen anderer profitieren. *—dab*

Steckbrett & Stolpersteine

Nützliche Hinweise zum Steckbrett und der Software



DAS BREADBOARD

Mit dem Steckbrett erspart man sich zeitraubende Lötarbeiten auf Platinen. Man steckt einfach die Anschlussdrähte der elektronischen Bauteile in die Löcher des Breadboards. Mit sogenannten Jumper-Kabeln (Male-Male) verdrahtet man die Bauelemente untereinander mit einer Spannungsversorgung sowie mit weiterer Elektronik – in unserem Fall dem Arduino.

Für Elektronik-Anfänger ist jedoch oft nicht ersichtlich, welche Löcher des Breadboard denn bereits miteinander verbunden sind. Auf dem Bild rechts sind die Verbindungen auf einem typischen Breadboard in gelb eingezeichnet. Die horizontalen Verbindungen sind üblicherweise der Spannungsversorgung vorbehalten, die vertikalen Verbindungen nutzt man, um die Anschlüsse der Bauelemente miteinander zu verbinden.

Breadboards lassen sich aber nur begrenzt für den Aufbau von Prototypen einsetzen. Durch ihre Bauform haben die Kontakte eine hohe Kapazität, die etwa bei digitalen Schaltungen mit einem Takt von mehreren MHz zu Störungen und Fehlern führen. Gleiches gilt für analoge Schaltungen. Breadboards gibt es in verschiedenen Breiten, aber immer mit den gleichen Abmessungen in vertikaler Richtung.

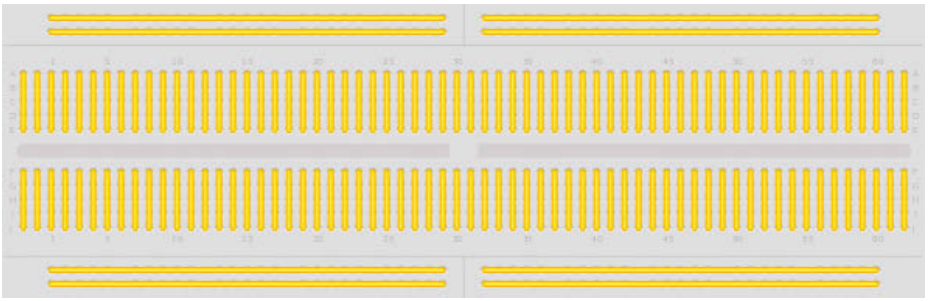


Bild: Wikipedia, CC BY-SA 3.0

SOFTWARE-STOLPERSTEINE

Die Beispiele der Arduino-IDE und unsere zum Download bereitgestellten Sketches sind alle funktional und syntaktisch korrekt. Sobald Sie beginnen, eigene Sketches zu programmieren, wird der Compiler vermutlich Fehler melden. Ein klassischer Anfängerfehler sind vergessene Semikola am Ende von Zeilen. In C muss **jede Anweisung** mit einem **Semikolon (;)** abgeschlossen werden. Davon ausgenommen sind Compiler-Anweisungen wie #include zum Einbau von Bibliotheken, Schleifenstrukturen (`for`, `while`), Auswahlstrukturen (`if`, `then`, `else`, `switch`) sowie Funktionsköpfe, also die erste Zeile von Funktionen wie `void setup` und `void loop`. Die Arduino-IDE zeigt die Zeile des ersten aufgetretenen Fehlers im Editor rot hinterlegt an. Alle weiteren gemeldeten Fehler sind meist Folgefehler eines vergessenen Semikolons, die mit der Korrektur des ersten Fehlers verschwinden.

Apropos `void setup`: manche Anfänger und viele Umsteiger von anderen Programmiersprachen tun sich mit dem **void** vor C-Funktionen schwer. Bevor Sie sich den Kopf zerbrechen, warum man dem Compiler extra sagen muss, dass eine Funktion keinen Wert zurückliefert: finden Sie sich einfach damit ab, dass da `void` steht. Es ist eine Erfindung von „Unix-Freaks“, die bei der Entwicklung der Sprache C vor Jahrzehnten bestimmte Regeln eingeführt haben, um den Compiler die Arbeit zu erleichtern. Bestimmte Dinge würde man heute so nicht mehr machen, dennoch gelten die Standards weiterhin.

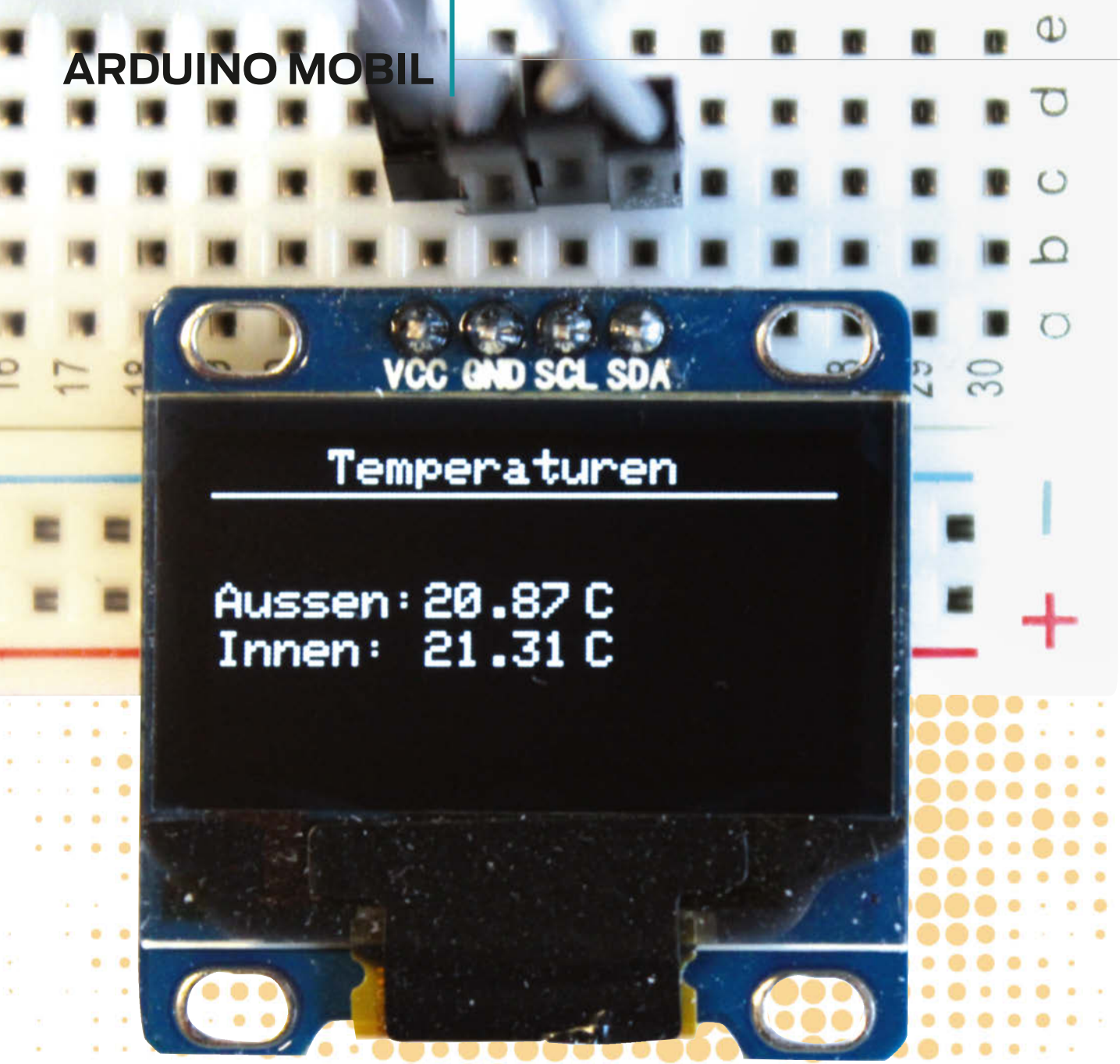
Vergessene Bibliotheken

Ein weiterer häufiger Fehler ist, benötigte Bibliotheken nicht in den eigenen Sketch mit der Anweisung **`#include`** einzubinden. Der Compiler spuckt dann im Statusfenster Hinweise wie **„xyz not declared in this scope“** und **„does not name a type“** aus, was so viel heißt, dass er im Sketch verwendete Datentypen und Funktionen nicht kennt. Sobald man die Bibliothek einbindet, verschwinden die Fehler bei den einfachen Beispielen.

Leider ist es manchmal gar nicht so leicht herauszufinden, wie die Bibliothek genau heißt. Bei den komplizierteren Beispielen hängen die Bibliotheken zudem teilweise voneinander ab oder sind in verschiedene Unterbibliotheken aufgeteilt. Alle richtigen Namen dann selbst herauszufinden, ist eine Strafarbeit. Die Arduino-IDE gibt hier aber Hilfestellung. Unter dem Punkt „Sketch/-Bibliothek einbinden“ findet man alle bereits installierten Bibliotheken. Ein Klick auf eine der Bibliotheken fügt die gewünschte Bibliothek als #include-Zeile am Anfang des Sketches hinzu – und alle weiteren irgendwie damit zusammenhängenden Bibliotheken ebenfalls.

Nicht benötigte Bibliotheken sollten Sie im Sketch löschen, da das Kompilat sonst unnötig groß wird und unter Umständen nicht mehr in den Flash des UNO hineinpasst.

—dab



Alleinstehend

Bei den bisherigen Projekten gab der Arduino seine Informationen an einen PC weiter oder erhielt Befehle von ihm. Nun stellen wir Projekte vor, bei denen man den PC nicht mehr unbedingt benötigt.

von Maik Schmidt

Links und Foren
make-magazin.de/xatq

ALLEINSTEHEND

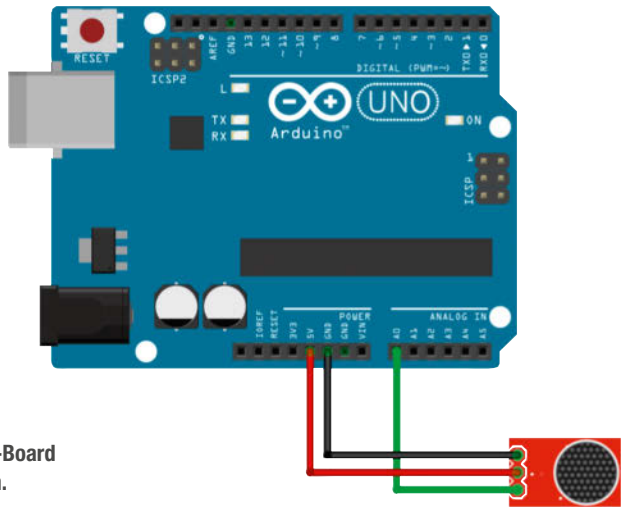
Alle Sketche finden Sie zum Download unter:
▶ www.make-magazin.de/xatq

Lärmampel

Der Arduino nimmt Geräusche mit einem Mikrofon auf und zeigt mit farbigen LEDs an, ob die gewünschte Lautstärke überschritten wurde.

Besonders faszinierend bei der Arbeit mit Mikrocontroller-Boards ist das Zusammenspiel zwischen Sensoren und Aktoren. Zwar ist das auch bei regulären Computern der Fall, wenn Tastatur-Eingaben (Sensor) zu Ausgaben auf dem Bildschirm (Aktor) führen, aber trotzdem unterliegt die Programmierung von Boards wie dem Arduino einer ganz eigenen Magie.

In einem neuen Projekt wird der Arduino daher zu einer Lärmampel, die in Abhängigkeit des aktuellen Umgebungslärms eine rote, gelbe oder grüne LED zum Leuchten bringen. Wenn alles still ist, leuchtet die grüne LED, bei einem mittleren Geräuschpegel ist es die gelbe und wenn es allzu laut wird, hilft nur noch die rote LED.



Das Mikrofon – Breakout-Board ist schnell angeschlossen.

Was wird gebraucht?

Zumindest die Status-LED wurde in vorherigen Projekten schon verwendet und so ist die Arbeit mit LEDs kein Hexenwerk mehr. Für das aktuelle Projekt werden aber drei LEDs benötigt und sie sollten auch die Farben Rot, Gelb und Grün haben, um den gewünschten Effekt zu unterstreichen.

Zur Messung des Geräuschpegels eignet sich ein Kondensator-Mikrofon. Diese gibt es (inklusive eines eingebauten Verstärkers) auf handlichen Breakout-Boards. Sie lassen sich direkt mit dem Arduino verbinden und liefern ein analoges Signal, das sich problemlos weiterverarbeiten lässt. Ein günstiges und passendes Modell gibt es zum Beispiel von Watterott (www.watterott.com/de/Microphone-Breakout). Um minimale Lötarbeiten kommt man hier aber nicht rum, denn das Breakout-Board muss noch mit einem Pin-Header (drei Anschlüsse) verbunden werden.

PegelMessen

```
 1 const unsigned long BAUD_RATE = 9600;
 2 const unsigned char MIKROFON = A0;
 3
 4 unsigned int minPegel = 1023;
 5 unsigned int maxPegel = 0;
 6
 7 void setup() {
 8   Serial.begin(BAUD_RATE);
 9 }
10
11 void loop() {
12   unsigned int pegel = analogRead(MIKROFON);
13   minPegel = min(pegel, minPegel);
14   maxPegel = max(pegel, maxPegel);
15
16   Serial.print("Pegel: ");
17   Serial.print(pegel);
18   Serial.print(", Min: ");
19   Serial.print(minPegel);
20   Serial.print(", Max: ");
21   Serial.println(maxPegel);
22 }
```

Klein anfangen

Für erste Tests reicht es aus, das Mikrofon direkt mit dem Arduino zu verbinden. Dazu eignen sich am besten drei Jumper-Wire mit jeweils einem männlichen und und einem weiblichen Ende. Der Masse-Anschluss (GND) des Mikrofons kommt an einen GND-Pin des Arduino, der VCC-Anschluss an den 5V-Pin und der Ausgang an den Analog-Port A0. Von Hersteller zu Hersteller kann der Name des Ausgangs variieren. Beim Sparkfun-Gerät heißt er AUD und bei Watterott ist es OUT.

Um sich einen Überblick über die Ausgabe des Mikrofons zu verschaffen, reicht ein simples Programm PegelMessen, das permanent die Werte ausgibt, die am analogen Eingang A0 anliegen. Ferner berechnet das Programm das Minimum und das Maximum der gemessenen Werte.

Der Quelltext beginnt mit der Definition der `BAUD_RATE`, die bei der Kommunikation mit der seriellen Schnittstelle verwendet wird. Er legt dann mit der Konstanten `MIKROFON` fest, mit welchem analogen Eingang das Mikrofon verbunden ist.

Danach definiert es die Variablen `minPegel` und `maxPegel`, in denen jeweils das Minimum und das Maximum der gemessenen Pegel-Werte gespeichert werden. Um sicherzugehen, dass die beiden Variablen am Ende auch die echten gemessenen Werte enthalten, wird `minPegel` mit dem maximal möglichen Messwert 1023 und `maxPegel` mit dem minimal möglichen Messwert 0 initialisiert. So

werden die beiden Variablen zumindest einmal überschrieben werden.

Für die setup-Funktion gibt es nicht viel zu tun und so initialisiert sie nur die serielle Schnittstelle. Auch loop ist übersichtlich und liest erst einmal den aktuellen Pegel-Wert vom zuvor festgelegten Mikrofon-Pin. Dann berechnet die Funktion das neue Minimum und das neue Maximum. Dazu verwendet sie die Funktionen min und max. Die Funktion min liefert das Minimum der beiden übergebenen Werte zurück. Ist also der aktuelle Pegel-Wert kleiner als der bisherige Wert der Variablen minPegel, dann wird minPegel auf diesen Wert gesetzt. Andernfalls bleibt der Wert so, wie er vorher war. Ganz analog verläuft das mit dem Maximum.

Alternativ hätten auch zwei if-Anweisungen zur Berechnung von Minimum und Maximum funktioniert, aber der Einsatz der min- und max-Funktionen ist kompakter und einfacher zu lesen.

Am Ende gibt loop den gemessenen Pegel, das aktuelle Minimum und das aktuelle Maximum auf der seriellen Schnittstelle aus.

Was bedeutet das alles?

Wurde das Programm auf den Arduino geladen, sieht die Ausgabe im seriellen Monitor relativ konstant aus und enthält nur Werte, die kaum von 511 abweichen. Das Mikrofon liefert also bei Stille einen Wert, der ziemlich genau bei der Hälfte des möglichen Spektrums von 0 bis 1023 liegt, zurück. Selbst ohne allzu großes Geschrei lassen sich Minimum und Maximum dann aber schnell auf ihre Extremwerte 0 und 1023 bringen. Offensichtlich stehen also nicht nur besonders große Werte für lautere Geräusche, denn bei Lärm geht der Ausschlag auch nach unten.

Das erscheint auch sinnvoll, denn stellt man sich Geräusche als eine Welle vor, dann gibt es halt Ausschläge nach oben und nach unten. Laute Geräusche erzeugen also hohe Werte oberhalb und unterhalb der Kennlinie für die absolute Stille.

Lautstärke ermitteln

Statt den Pegel direkt einzusetzen, ist es daher leichter, den gemessenen Wert in so etwas wie einen Lautstärke-Wert umzurechnen. Das ist ganz leicht und das Programm LautstaerkeMessen zeigt, wie es geht. Obendrein erzeugt es noch einen netten Effekt auf der seriellen Schnittstelle.

Abgesehen von der loop-Funktion unterscheidet sich dieses Programm kaum vom ersten Test-Programm und auch hier misst loop zunächst einmal den aktuellen Pegel. Dann wird die Differenz zwischen Pegel und 511 gebildet, um den „Abstand zur Stille“ zu berechnen. Die Variable differenz hat den Datentypen int, weil sie auch negativ werden kann, wenn der gemessene Pegel kleiner als 511 ist.

Anschließend wird ein Lautstärke-Wert errechnet und der Kern der Formel ist die Funktion abs. Sie berechnet den Absolutbetrag einer Zahl, also die Zahl ohne ein etwaiges Vorzeichen. Beispielsweise ist abs(-42) gleich 42 und abs(12) ergibt 12. Das Ergebnis von abs ist also immer eine positive Zahl.

Angewendet wird die Funktion auf differenz. Die Variable pegel kann Werte von 0 bis 1023 annehmen und wenn von denen der Wert 511 subtrahiert wird, reicht die Differenz von –511 bis 512. Wendet man auf diese Werte dann die Funktion abs an, so erhält man am Ende nur noch Werte zwischen 0 und 512. Die entsprechen genau den Ausschlägen nach oben und unten und repräsentieren die Lautstärke.

Der Lärmpegel über die Zeit dargestellt

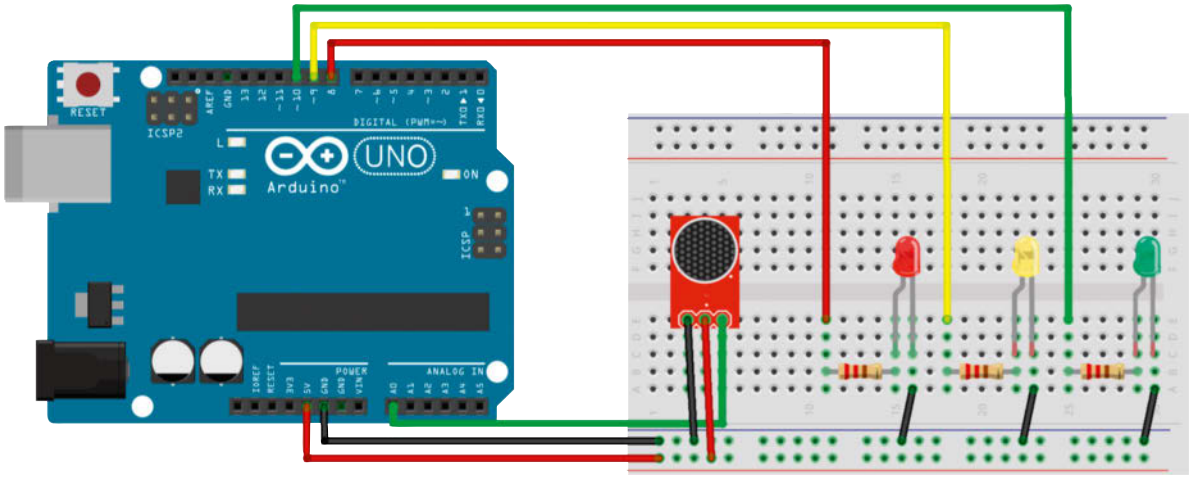
Weil für die Lärmampel nur die relative – und nicht die absolute – Lautstärke relevant ist, wird die berechnete Lautstärke noch durch 10 dividiert. Somit bleiben am Ende Lautstärke-Werte von 0 bis 51.

In der anschließenden for-Schleife wird die gemessene Lautstärke dann als eine Sequenz von #-Zeichen ausgegeben. Die Ausgabe enthält dabei immer ein #-Zeichen mehr als der berechnete Lautstärkewert. Hat lautstaerke zum Beispiel den Wert 5, so wird ###### ausgegeben. Dieses Verfahren sorgt dafür, dass immer mindestens ein #-Zeichen ausgeben wird. Nach der Schleife wird ein Zeilenumbruch ausgegeben, sodass die nächste Ausgabe in einer neuen Zeile beginnt. Außerdem wird mit der delay-Funktion die Ausgabe noch etwas verlangsamt.

Der Screenshot zeigt den Effekt des Programms. Spricht man in das Mikrofon, werden die Lautstärke-Ausschläge im seriellen Monitor als Balken angezeigt. Dreht man im Geiste die Ausgabe um 90 Grad gegen den Uhrzeigersinn, entspricht die Ausgabe in

LautstaerkeMessen

```
 1 const unsigned long BAUD_RATE = 9600;
 2 const unsigned char MIKROFON = A0;
 3
 4 void setup() {
 5   Serial.begin(BAUD_RATE);
 6 }
 7
 8 void loop() {
 9   unsigned int pegel = analogRead(MIKROFON);
10   int differenz = pegel - 511;
11   unsigned int lautstaerke = abs(differenz) / 10;
12
13   for (unsigned int i = 0; i < lautstaerke + 1 ; i++) {
14     Serial.print("#");
15   }
16   Serial.println();
17
18   delay(100);
19 }
```



Mit drei farbigen LEDs erhält man eine Lärmampel.

etwa den vu-Metern, wie sie in alten Kassetten-Spielern oder HiFi-Anlagen verbaut waren. Der Effekt würde noch deutlicher, wenn die Ausgabe des seriellen Monitors Farben unterstützte, sodass besonders große Werte rot und die übrigen gelb und grün eingefärbt werden könnten.

Lichtorgel

Die serielle Schnittstelle unterstützt zwar keine Farben, aber dafür lassen sich problemlos farbige LEDs mit dem Arduino verbinden. Die entsprechende Schaltung sieht komplex aus, ist aber im Grunde ganz einfach.

Die Verbindung zwischen dem Mikrofon und dem Arduino bleibt wie gehabt. Das Mikrofon sollte jetzt aber auf das Breadboard gesteckt werden, weil sich das Endprodukt so einfacher handhaben lässt. Es spricht aber sonst nichts dagegen, das Mikrofon weiterhin über drei Kabel mit dem Arduino direkt zu verbinden. Dann muss lediglich für die Versorgung des Breadboards mit der Masse ein weiterer GND-Pin des Arduino verwendet werden. Auf jeden Fall sollte ein GND-Pin und der 5V-Pin des Arduino das Breadboard mit Strom versorgen.

Der Rest der Schaltung wiederholt das gleiche Muster dreimal, nämlich für jede LED. Die werden in der Reihenfolge rot, gelb und grün in das Breadboard gesteckt und zwar so, dass die Anode nach links zeigt. Jede LED hat zwei Anschlüsse, nämlich einen positiven (Anode) und einen negativen (Kathode). Mit der Eselsbrücke „Die _Ka_thode ist ne_ka_tiv.“ lässt sich das leicht merken.

In der Regel ist der Anschlussdraht der Anode etwas länger als der der Kathode. Auch liefert die Bauform der LED einen Anhaltspunkt, denn auf der Seite der Kathode ist sie flach, während die Seite der Anode gewölbt ist. Auch das lässt sich leicht merken, denn die Anode ist positiv, also Plus. Daher ist der Draht auch länger und die Wölbung dicker. Kleiner Tipp: Wenn man die Anschlussdrähte einer LED kürzt, weil sie sich so leichter ins Breadboard stecken lässt, sollte man den Draht der Anode etwas länger lassen.

Die Kathoden der drei LEDs müssen mit der Masse des Arduino verbunden werden und dazu eignen sich drei kurze Drähte. Ferner müssen die Anoden mit jeweils einem Digitalpin verbunden werden, damit der Arduino sie ein- und ausschalten kann. In diesem Fall sind das die Pins 8 (rot), 9 (gelb) und 10 (grün).

LEDs benötigen einen Vorwiderstand und dürfen nicht direkt mit einem der digitalen Pins verbunden werden. Andernfalls könnten sie zerstört werden. Die Größe des Widerstandes hat direkten Einfluss auf die Helligkeit der LED. Ist der Widerstand zu hoch, leuchtet die LED zu dunkel. Ein Widerstand von 220 Ohm sollte in den meisten Fällen ein guter Kompromiss sein.

Widerstände haben keine Richtung. Es ist also egal, wie sie in das Breadboard gesteckt werden. Wichtig ist nur, dass ein Draht des

Laermampel

```
 1 const unsigned long BAUD_RATE = 9600;
 2 const unsigned char MIKROFON = A0;
 3 const unsigned char LED_ROT = 8;
 4 const unsigned char LED_GELB = 9;
 5 const unsigned char LED_GRUEN = 10;
 6 const unsigned int SCHWELLE_ROT = 35;
 7 const unsigned int SCHWELLE_GELB = 25;
 8 const unsigned int LEUCHT_DAUER = 500;
 9
10 void setup() {
11   pinMode(LED_ROT, OUTPUT);
12   pinMode(LED_GELB, OUTPUT);
13   pinMode(LED_GRUEN, OUTPUT);
14   Serial.begin(BAUD_RATE);
15 }
16
17 void loop() {
18   unsigned int pegel = analogRead(MIKROFON);
19   int differenz = pegel - 511;
20   unsigned int lautstaerke = abs(differenz) / 10;
21
22   if (lautstaerke > SCHWELLE_ROT) {
23     Serial.println("Zu laut!");
24     schalte_ampel(HIGH, LOW, LOW);
25     delay(LEUCHT_DAUER);
26   }  else if (lautstaerke > SCHWELLE_GELB){
27     Serial.println("Ziemlich laut.");
28     schalte_ampel(LOW, HIGH, LOW);
29     delay(LEUCHT_DAUER);
30   } else {
31     Serial.println("Alles ruhig.");
32     schalte_ampel(LOW, LOW, HIGH);
33   }
34 }
35
36 void schalte_ampel(unsigned int rot, unsigned int gelb, ↗
   unsigned int gruen) {
37   digitalWrite(LED_ROT, rot);
38   digitalWrite(LED_GELB, gelb);
39   digitalWrite(LED_GRUEN, gruen);
40 }
```

Widerstandes mit der Masse des Arduino und das andere Ende mit einem digitalen Pin verbunden ist.

Ein wenig mehr Code

Das Programm Laermampel ist geringfügig umfangreicher als das Programm zur Ausgabe der Lautstärke, aber es ähnelt seinem Vorgänger strukturell recht stark. Beispielsweise beginnt es mit der Definition derselben Konstanten `BAUD_RATE` und `MIKROFON`.

Es folgen die Definitionen der Konstanten `LED_ROT`, `LED_GELB` und `LED_GRUEN` für die Pins, mit denen die LEDs verbunden sind. `SCHWELLE_ROT` und `SCHWELLE_GELB` steuern, ab welcher Lautstärke die rote beziehungsweise die gelbe LED eingeschaltet werden. Bei Lautstärken unterhalb von `SCHWELLE_GELB` leuchtet die grüne LED.

Schließlich legt `LEUCHT_DAUER` noch fest, wie lang eine LED mindestens leuchten soll, nachdem sie eingeschaltet wurde. Damit wird ein allzu hektisches Flackern bei schnell wechselnden Geräuschpegeln vermieden.

Die `setup`-Funktion versetzt die LED-Pins in den Ausgabemodus und initialisiert die serielle Schnittstelle. Anschließend berechnet `loop` wie gehabt die aktuell gemessene Lautstärke. In Abhängigkeit von der Lautstärke werden dann die LEDs ein- beziehungsweise ausgeschaltet. Dazu dient die Funktion `schalte_ampel`. Sie erwartet den für die rote, gelbe und grüne LED gewünschten Zustand (HIGH oder LOW) und setzt die damit verbundenen Pins entsprechend.

Das ist schon ein recht anspruchsvolles Projekt, aber es zeigt auch, dass sich alle Teile schnell zusammenfügen, wenn man die Daten, die ein Sensor wie das Mikrofon liefert, korrekt interpretiert und in ein passendes Format umwandelt. *—dab*

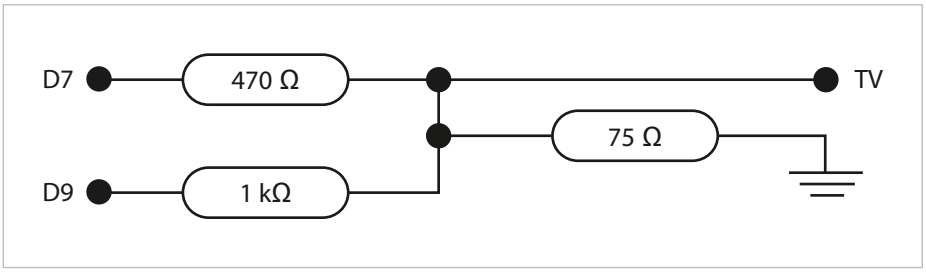
Kontrollstation

Mit wenigen zusätzlichen Bauteilen kann der Arduino einen alten PAL-Fernseher als Ausgabegerät nutzen.

Um mit der Außenwelt zu kommunizieren, verwendeten die bisherigen Arduino-Projekte die serielle Schnittstelle und LEDs. Für viele Anwendungen reicht das aus, aber hin und wieder darf es auch ein bisschen mehr sein.

Eine günstige und technisch einfache Alternative ist es um Beispiel, den Arduino mit einem Fernseher zu verbinden. Dazu reichen ein AV-Kabel, zwei Widerstände und ein bisschen Software.

Dieser Artikel zeigt, wie das geht, und vermittelt nicht nur die Grundlagen analoger Videosignale, sondern verwandelt den Arduino auch in den vielleicht überdimensioniertesten Batterie-Tester der Welt.



Mit dieser Minimalschaltung kann der Arduino ein Fernsehbild erzeugen.

Was versteht so ein Fernseher?

In Zeiten von HDTV ist jeder Fernseher im Grunde nichts anderes als ein PC. Aktuelle Geräte verbinden sich mit dem Internet und lassen sich via Bluetooth mit Maus und Tastatur bedienen. Video- und Audiosignale werden digital per HDMI eingespeist und von leistungsstarken Prozessoren fürs LED-Panel aufbereitet.

So gut wie alle TV-Geräte kommen heute aber immer noch mit analogen Signalen zurecht und in vielen Haushalten dürfte sich auch noch das eine oder andere analoge Schätzchen finden. Ein Arduino Uno reicht aus, um ein stabiles monochromes Videosignal zu erzeugen. Die Auflösung ist zwar recht niedrig, für viele Anwendungen aber mehr als genug.

Prinzipiell ist die Erzeugung eines monochromen Videosignals kein Hexenwerk, denn ein Fernseher erwartet minimal nur drei verschiedene Signale, die mittels verschiedener Spannungen kodiert werden. Das Sync-Signal wird durch eine Spannung von 0 Volt repräsentiert und dient der Synchronisation von Sender und Empfänger. Das Schwarz-Signal liegt in der Regel bei 0,3 Volt und dient zur Darstellung schwarzer Punkte. Analog wird das Weiß-Signal zur Darstellung weißer Punkte verwendet und mit einer Spannung von 1 Volt kodiert.

Wer mit einem Arduino ein Videosignal erzeugen möchte, muss mindestens diese drei analogen Signale abbilden können. Allerdings hat der Arduino keine echten analogen Ausgänge und kann analoge Signale nur mittels Pulsweitenmodulation simulieren. Das ist ausreichend für das Dimmen von LEDs oder die Ansteuerung von Motoren, aber für ein stabiles Videosignal ist das Verfahren ungeeignet.

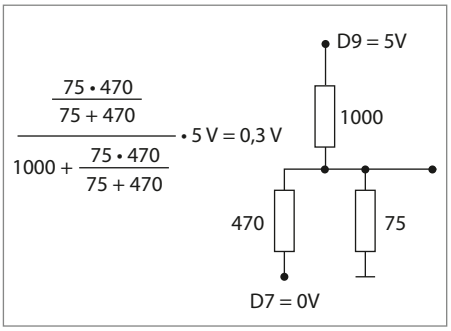
Die einfachste Lösung besteht darin, zwei digitale Pins mit einem Digital-/Analog-Wandler (DA) zu verbinden. Dieser DA hat

Cinch-Kabel in verschiedenen Farben

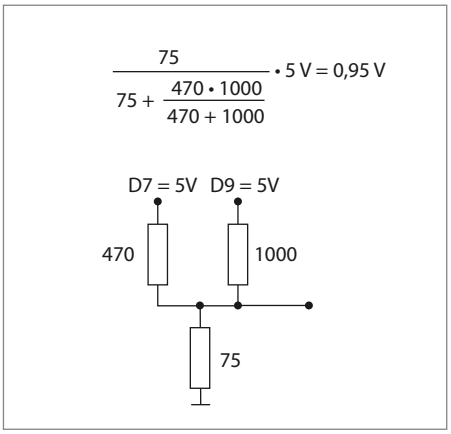
eine Auflösung von zwei Bits und kann daher vier unterschiedliche Eingangswerte jeweils in eine Spannung zwischen 0V und 1V umwandeln. Ein solcher DA lässt sich mit wenigen Widerständen realisieren, wobei für jedes Bit ein Widerstand benötigt wird. Je höherwertiger das Bit ist, umso größer muss der Widerstand sein. Ferner muss die Größe eines Widerstands immer circa doppelt so groß sein, wie der Widerstand des benachbarten niederwertigen Bits.

In einer Beispiel-Schaltung repräsentiert Pin 7 das niederwertige Bit (470 Ohm) und Pin 9 das höherwertige (1 kOhm). Das reicht schon aus, um alle benötigten Signale zu erzeugen. Für das Sync-Signal (0V) sind die Pins 7 und 9 lediglich auf LOW zu setzen. Die Spannungen der übrigen Signale lassen sich mit dem Ohm'schen Gesetz und den Rechenregeln für Spannungsteiler berechnen. Dazu muss man wissen, dass ein Fernseher in der Regel einen Eingangswiderstand von 75 Ohm hat. Unsere Schaltung enthält daher insgesamt drei Widerstände.

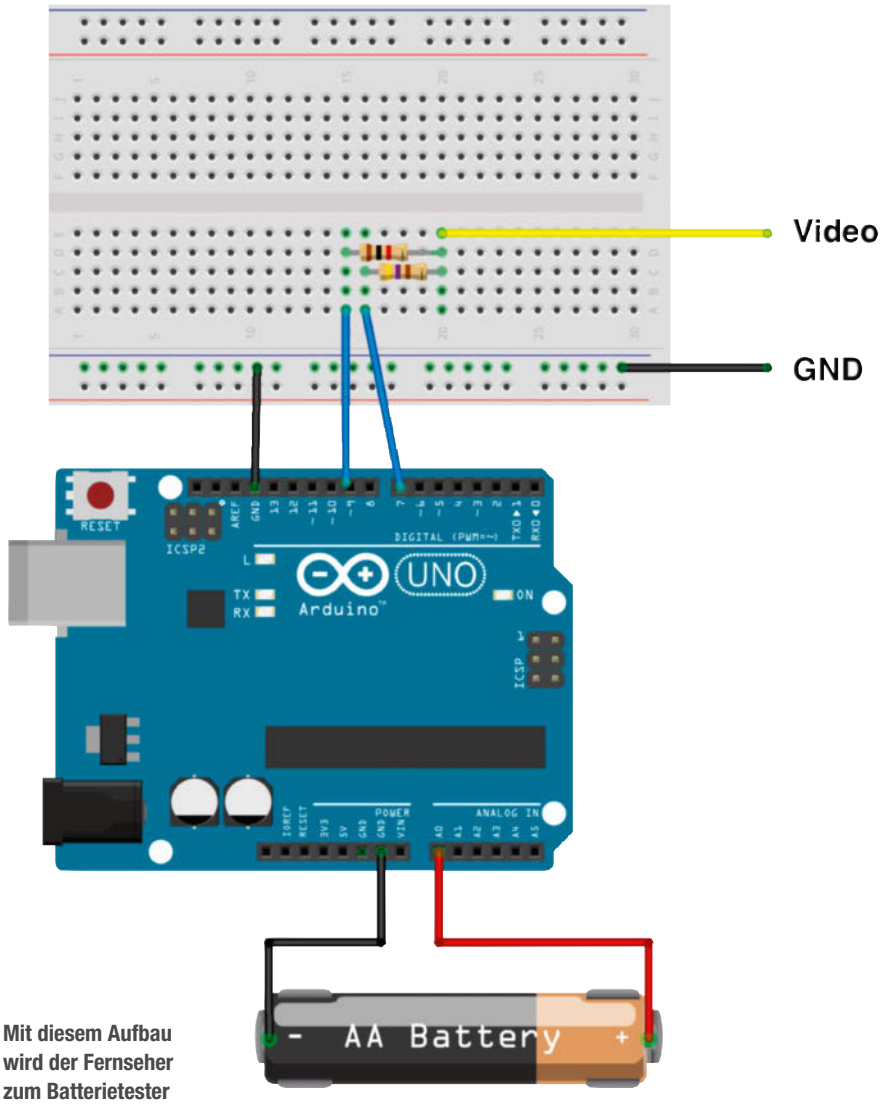
In der gewählten Konfiguration berechnet sich das Schwarz-Signal dann wie folgt:



Für Weiß sieht es ganz ähnlich aus:



Zwar liegt das Signal nicht exakt bei 1 V, aber in der Praxis ist das unerheblich. Für grau ergibt sich ein Wert von ungefähr 0,65 V.



Mit diesem Aufbau wird der Fernseher zum Batterietester

Kabel basteln

Mit der Schaltung lassen sich die benötigten Signale erzeugen, aber irgendwie müssen sie noch ins Fernsehgerät gelangen. Dazu eignet sich ein AV-Kabel mit Cinch-Stecker. Zwar verfügt der Arduino über keine Cinch-Buchse, aber mit ein paar Handgriffen lässt sich ein AV-Kabel so modifizieren, dass man es mit dem Arduino verbinden kann.

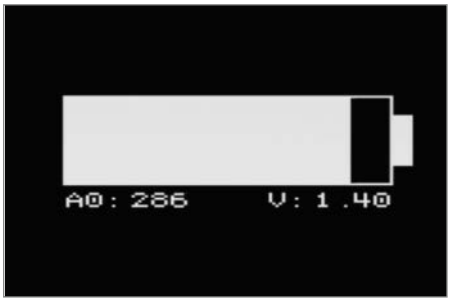
Zuerst schneidet man einen Stecker des Kabels ab und entfernt dann circa zwei Zentimeter der äußeren Isolierung vorsichtig mit einem scharfen Messer. Zum Vorschein kommt ein dünnes Drahtgeflecht, das man zwischen Daumen und Zeigefinger zu einem Draht zwirbeln kann. Anschließend muss die innere Isolierung vorsichtig mit einem schar-

An einer Seite muss das Cinch-Kabel abisoliert werden.

Mit zwei Widerständen kann man die Pegel für TV erzeugen.

fen Messer entfernt werden, um die eigentliche Signal-Leitung freizulegen.

Im letzten Schritt wird die Signal-Leitung mit den beiden Widerständen verlötet und das verzwirbelte Drahtgeflecht mit einem Stück Schaltdraht verlängert. Jetzt kann das Massekabel mit einer der GND-Buchsen des Arduino verbunden werden und die freien Enden der Widerstände gehören an die digitalen Pins 7 (470 Ohm) und 9 (1 kOhm).



Der Batteriestand auf dem Fernseher

Die Software

Die Hardware ist schon vollständig und es fehlt nur noch ein wenig Software, um sie zum Leben zu erwecken. Dazu muss man aber genau wissen, wie ein Fernsehbild entsteht.

Ein analoges Fernsehbild wird von oben nach unten zeilenweise aufgebaut. Am Ende einer jeden Zeile gibt es eine so genannte horizontale Austastlücke, in der bei Kathodenstrahlgeräten der Elektronenstrahl an den Anfang der nächsten Zeile bewegt werden kann. Am unteren Ende des Bilds gibt es eine vertikale Austastlücke, in der der Elektronenstrahl zurück in die erste Zeile wandert. In den Zeilen selbst dienen die Impulse von 0,3 V und 1 V zur Erzeugung der Bildinformationen.

Auch wenn in modernen Fernsehgeräten schon längst kein Elektronenstrahl mehr werkelt, hat sich am analogen Protokoll nichts geändert. Mittels Software lässt es sich nicht so einfach implementieren, denn die benötigten Signale müssen in exakten Zeitintervallen generiert werden. Ohne maschinennahe Programmierung ist das kaum zu bewerkstelligen.

Glücklicher Weise gibt es für so gut wie alle Arduino-Modelle die TVout-Bibliothek. Sie erzeugt nicht nur ein stabiles Videosignal, sondern bietet auch noch eine Menge Funktionen zum Zeichnen geometrischer Figuren und zur Darstellung von Texten.

Die Installation erfolgt über den Library-Manager. Leider reicht die Installation in diesem Fall nicht aus, denn die aktuelle Version von TVout benötigt noch eine manuelle Korrektur im zentralen Bibliotheksverzeichnis des Arduino. Es kann über das Einstellungen-Menü ermittelt werden. Unter Windows liegt das Verzeichnis in der Regel im Ordner Eigene Dokumente\Arduino\libraries, unter Mac OS X ist es Documents\Arduino\libraries und unter Linux ist es in der Regel ein Verzeichnis libraries im lokalen Ordner für die Arduino-Sketches.

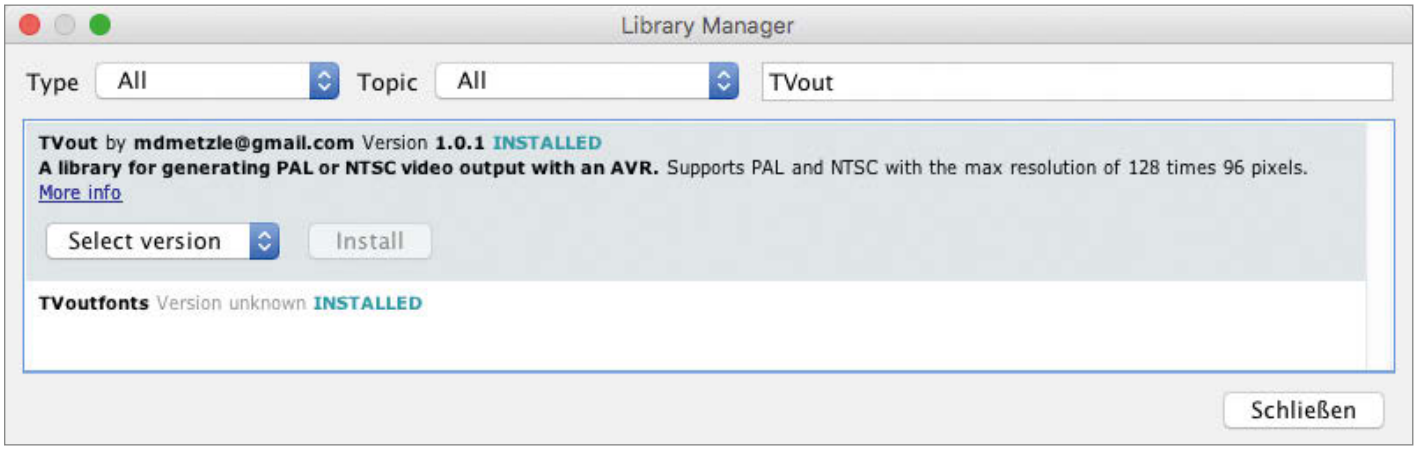
Nach der Installation enthält dieses Verzeichnis ein neues Unterverzeichnis namens TVout. Dieses wiederum enthält das Unterverzeichnis TVoutfonts. Das TVoutfonts-Verzeichnis muss auf dieselbe Ebene verschoben werden, in der auch das TVout-Verzeichnis liegt. Am Ende enthält also das zentrale Bibliotheksverzeichnis die Unterverzeichnisse TVout und TVoutfonts. Anschließend muss die Arduino-IDE neu gestartet werden.

Der Batterie-Tester „Enterprise 3000“

Als Beispiel-Projekt dient ein Batterie-Tester für handelsübliche AA- und AAA-Zellen mit einer Maximalspannung von 1,6 Volt. Im Gegensatz zum Vorgänger-Projekt wird die gemessene Spannung aber nicht auf der seriellen Schnittstelle ausgegeben. Diesmal erscheint sie in Form einer dynamischen Batterie-Grafik auf einem beliebig großen Fernseher. Die Grafik ändert sich in Abhängigkeit der Batteriespannung und zusätzlich gibt das Programm den am analogen Eingang A0 gemessenen Wert und die korrespondierende Spannung aus.

Dazu wird im Programm BatterieTester zuerst die TVout-Bibliothek mittels einer `#include`-Anweisung eingebunden. Weil nicht nur grafische Elemente, sondern auch Texte ausgegeben werden sollen, muss auch die Datei `fontALL.h` eingebunden werden.

Die Konstante `MAX_SPANNUNG` legt fest, welcher Maximalwert am analogen Eingang für die Batteriespannung erwartet wird. Die Zahl 328 entspricht einer Spannung von 328 / 1024 × 5,0 = 1,6V. `TV_BREITE` und `TV_HOEHE` definieren die Bildschirm-Abmessungen in Pixeln.



TVout enthält alle Funktionen zur Erzeugung des TV-Bilds.

Das globale Objekt TV vom Typ TVout dient zur Ansteuerung des Fernsehers und wird in der setup-Funktion mit der Methode begin initialisiert. TVout unterstützt sowohl den TV-Standard PAL als auch das hauptsächlich in den USA verbreitete NTSC. Mit begin kann man eines der beiden Verfahren auswählen und in Deutschland ist PAL zumeist die sinnvollere Wahl.

Darüber hinaus kontrolliert die Funktion die Breite und Höhe des dargestellten Bilds. 120 mal 96 Bildpunkte sind so ziemlich das Ende der Fahnenstange, wenn noch etwas Speicher für die Anwendung übrig bleiben soll. Bei dieser Auflösung beansprucht der Video-Speicher 120 / 8 × 96 = 1440 Bytes. Auf einem Arduino Uno mit 2048 Bytes RAM wird es dann schnell knapp.

Die Funktion select_font wählt einen der drei verfügbaren Zeichensätze aus. Möglich sind die Zeichensatzgrößen 4 × 6, 6 × 8 und 8 × 8. Zwischen den Größen kann im Verlaufe des Programms beliebig gewechselt werden, indem die Funktion select_font erneut aufgerufen wird.

In loop sorgt der Aufruf von clear_screen dafür, dass der Bildschirm gelöscht wird. Danach wird die Spannung gemessen, die am analogen Eingang A0 anliegt. Mit der max-Funktion wird sie dann auf den Maximalwert gestutzt und anschließend an die Funktion zeichne_batterie übergeben.

Diese Funktion verwendet die draw_rect-Funktion, um eine Batterie mit Hilfe von zwei Rechtecken auf den Bildschirm zu zeichnen. draw_rect erwartet die X- und Y-Koordinate der linken oberen Ecke des zu zeichnenden Rechtecks. Ferner müssen die Breite, die Höhe und die Farbe übergeben werden. Für TVout liegt der Ursprung aller Koordinaten in der linken oberen Ecke des Bildschirms. Links oben liegt der Punkt (0, 0) und rechts unten der Punkt (119, 95). Als Farben sind BLACK, WHITE und INVERT erlaubt. INVERT invertiert die Farben der Punkte im Zielbereich, das heißt, wenn sie zuvor schwarz waren, werden sie weiß und umgekehrt. Einen Grauton unterstützt TVout nicht.

Die Anweisung

```
TV.draw_rect(7, 40, 100, 36, WHITE);
```

zeichnet ein weiß umrandetes Rechteck, dessen linke obere Ecke die Koordinaten (7, 40) hat. Das Rechteck ist 100 Pixel breit und 36 Pixel hoch. Es bildet den Körper der Batterie und der anschließende Aufruf von draw_rect zeichnet den Pluspol. Das Rechteck für den Pluspol wird mit der Farbe Weiß gefüllt. Daher bekommt draw_rect noch ein weiteres Argument für die Füllfarbe (WHITE) übergeben.

Um den Zustand der Batterie zu visualisieren, muss noch der Körper der Batterie in Abhängigkeit der gemessenen Spannung gefüllt werden. Dazu wird ein weiteres Rechteck gezeichnet, dessen Breite mit der map-Funktion berechnet wird. Diese Funktion bildet einen Wert vom einem Wertebereich in einen anderen ab. In diesem Fall bildet sie einen Spannungswert von 0 bis 328 (MAX_SPANNUNG) auf den Bereich 0 bis 100 (die Breite des zu füllenden Batterie-Rechtecks) ab.

Die loop-Funktion gibt am Ende unterhalb der Batterie den am analogen Eingang A0 gemessenen Wert und die korrespondierende Spannung aus. Dabei hilft die Funktion print, die einen Text an einer beliebigen Position auf dem Bildschirm ausgibt.

Der Aufruf von delay_frame am Ende der Funktion verhindert hässliches Flackern und wartet auf das Ende der vertikalen Austastlücke.

Fazit

Es ist faszinierend, den Arduino mit Haushaltsgeräten wie einem Fernseher zu kombinieren. Zum einen kann man dabei eine Menge über Alltagstechnologien lernen und zum anderen eröffnen sich damit ganz neue Möglichkeiten für zukünftige Projekte.

Die TVout-Bibliothek enthält übrigens ein aufwendiges Beispiel-Projekt, das alle Funktionen der Bibliothek demonstriert. Es ist beeindruckend, was sich mit einem kleinen Mikrocontroller so anstellen lässt, und es reicht sogar für die Umsetzung unterhaltsamer Video-Spiele (siehe c't Hardware-Hacks 2/2013 ab Seite 82). *—dab*

BatterieTester

```
 1 #include <TVout.h>
 2 #include <fontALL.h>
 3
 4 const unsigned int MAX_SPANNUNG = 328; // ~1,6V
 5 const unsigned char TV_BREITE = 120;
 6 const unsigned char TV_HOEHE = 96;
 7
 8 TVout TV;
 9
10 void setup() {
11   TV.begin(PAL, TV_BREITE, TV_HOEHE);
12   TV.select_font(font6x8);
13 }
14
15 void loop() {
16   TV.clear_screen();
17   unsigned int spannung = analogRead(A0);
18   spannung = min(spannung, MAX_SPANNUNG);
19   zeichne_batterie(spannung);
20   TV.print(8, 80, "A0:");
21   TV.print(28, 80, spannung);
22   TV.print(70, 80, "V:");
23   TV.print(84, 80, spannung * 5.0 / 1024);
24   TV.delay_frame(1);
25 }
26
27 void zeichne_batterie(unsigned int spannung) {
28   TV.draw_rect(7, 40, 100, 36, WHITE);
29   TV.draw_rect(107, 48, 6, 20, WHITE, WHITE);
30   unsigned int breite = map(spannung, 0, MAX_SPANNUNG, 0, 100);
31   TV.draw_rect(7, 40, breite, 36, WHITE, WHITE);
32 }
```

Mäusekino

Der Arduino kann kleinere digitale Displays ansteuern, um auf ihnen Informationen und Grafiken anzuzeigen.

Die bisherigen Projekte haben meist über die serielle Schnittstelle mit der Außenwelt kommuniziert. Das ist bequem, schränkt aber die Mobilität von Projekten ein. Wenn ein Arduino-Projekt Informationen ausschließlich über die serielle Schnittstelle ausgibt, kann es nur zusammen mit einem anderen Gerät, das diese Informationen aufbereitet, benutzt werden. Viel praktischer wäre es in vielen Fällen, wenn der Arduino einen eigenen Bildschirm hätte.

Dieser Wunsch lässt sich überraschend einfach und günstig erfüllen, denn durch den massenhaften Bedarf an MP3-Spielern und Handys gibt es eine Fülle von preiswerten und leistungsfähigen Mini-Bildschirmen.

Mit so einem Bildschirm und zwei Temperatur-Sensoren lässt sich aus dem Arduino ein Thermometer basteln, das die Innen- und Außen-Temperatur anzeigt.

Der Bus füllt sich

Vor der Ausgabe der Temperaturen auf einem Display wollen diese aber erst einmal gemessen werden und so ist zu klären, wie der Arduino mehr als einen Sensor abfragen kann. Mit dem DS18B20 ist das ein Kinderspiel, denn davon können wegen des 1-Wire-Protokolls viele über einen gemeinsamen Pin mit dem Arduino kommunizieren.

Wenn mehrere Sensoren über denselben 1-Wire-Bus kommunizieren, ist ein wenig Vorbereitung vonnöten, denn für jeden Sensor muss klar sein, welchem Zweck er dient und wo er sich gerade befindet. Beispielsweise kann ein Sensor die Außentemperatur und ein anderer die Innentemperatur messen. Dazu muss jeder Sensor identifizierbar sein und tatsächlich verfügen alle DS18B20-Sensoren über eine eindeutige Kennung. Diese Kennung lässt sich mit der auf Seite 27 gezeigten Schaltung und dem Programm Sensor_Adresse_Ermitteln ermitteln.

Bis zur Definition der loop-Funktion enthält das Programm nur eine einzige Neuerung: In Zeile 10 wird die Variable sensorKennung definiert, die den Typen DeviceAddress hat. Den definiert die Datei DallasTemperature.h und er dient zur Speicherung einer Sensor-Kennung. Das ist eine Zahl, die 64 Bit, also acht Bytes beziehungsweise acht Zahlen von 0 bis 255, umfasst.

Die folgende Anweisung liest mittels der Funktion getAddress die Kennung des ersten am Bus gefundenen Sensors (mit dem Index 0) in die Variable sensorKennung:

```
sensoren.getAddress(sensorKennung, 0)
```

Wenn die Kennung nicht ermittelt werden konnte, liefert die Funktion den Wert 0 zurück. Zeile 18 des Programms ruft die Funktion daher innerhalb einer if-Anweisung auf und prüft sogleich den Rückgabewert mit dem ==-Operator.

Wenn ein DS18B20-Sensor angeschlossen und erkannt wurde, gibt das Programm permanent die Kennung des Sensors, also acht Zahlen, aus. Dazu setzt es eine for-Schleife ein, deren Zählvariable i die Werte von 0 bis 7 annimmt. Mittels der Variablen i wird innerhalb der Schleife die Variable sensorKennung indiziert. Damit werden nach und nach die einzelnen Zahlen der Sensorkennung ausgegeben. Die nachfolgende if-Anweisung stellt sicher, dass nach jeder Zahl (außer nach der letzten) ein Komma und ein Leerzeichen ausgegeben werden.

Der hier eingesetzte Sensor hat die Kennung

```
40, 255, 1, 17, 101, 21, 2, 13
```

Damit kann er problemlos am 1-Wire-Bus identifiziert werden. Das Programm muss für die beiden Sensoren, die zum Einsatz kommen sollen, durchlaufen werden. Es ist sinnvoll, die jeweiligen Kennungen zu notieren, denn sie werden im Folgenden benötigt.

OLED-Display anschließen

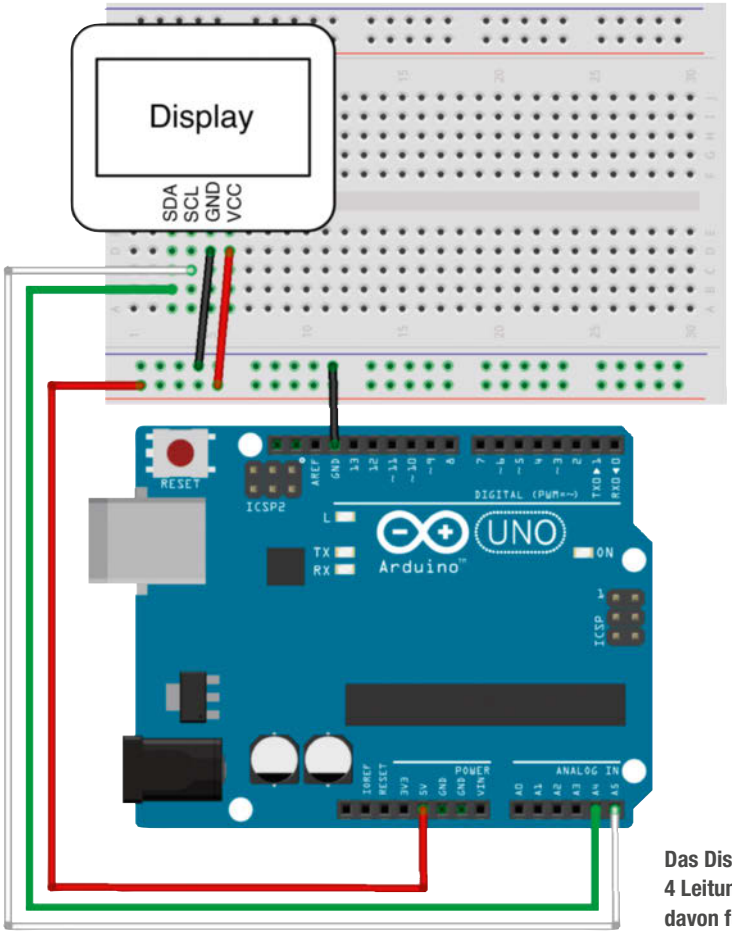
Für das Thermometer-Projekt kommt ein OLED-Display mit einer Bildschirmdiagonale von 0,96 Zoll und einer Auflösung von 128 × 64 Pixeln zum Einsatz. Das entspricht in etwa dem, was in den Nullerjahren des 21. Jahrhunderts in Handys und MP3-Spielern

Sensor_Adresse_Ermitteln

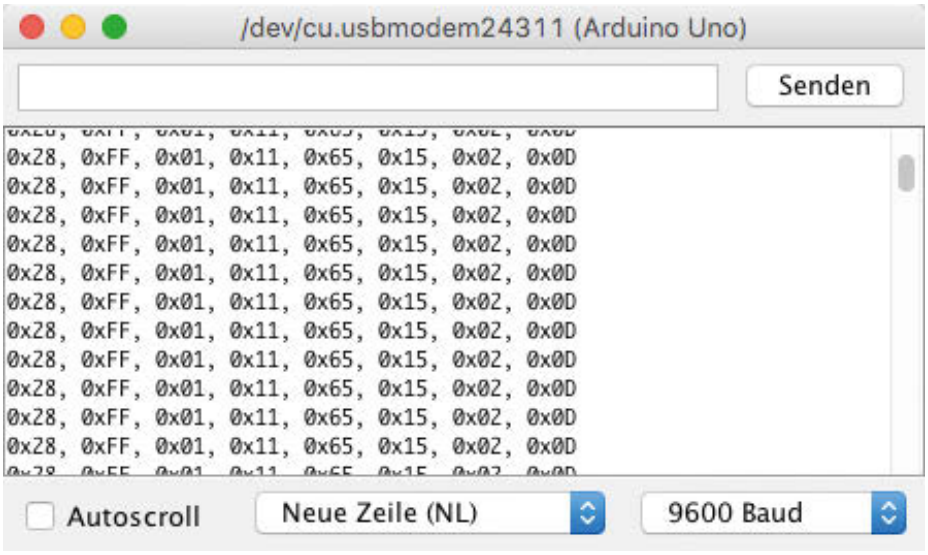
```
 1 #include <OneWire.h>
 2 #include <DallasTemperature.h>
 3
 4 const unsigned long BAUD_RATE = 9600;
 5 const unsigned char ONE_WIRE_BUS = 4;
 6
 7 OneWire oneWire(ONE_WIRE_BUS);
 8 DallasTemperature sensoren(&oneWire);
 9
10 DeviceAddress sensorKennung;
11
12 void setup() {
13   Serial.begin(BAUD_RATE);
14   sensoren.begin();
15 }
16
17 void loop() {
18   if (sensoren.getAddress(sensorKennung, 0) == 0) {
19     Serial.println("Die Kennung des Sensors konnte ↵
       nicht ermittelt werden.");
20   } else {
21     for (unsigned char i = 0; i < 8; i++) {
22       Serial.print(sensorKennung[i]);
23       if (i < 7) {
24         Serial.print(", ");
25       }
26     }
27     Serial.println();
28   }
29   delay(1000);
30 }
```

verbaut wurde. Diese Displays sind preiswert, verbrauchen nur wenig Energie und sind einfach zu programmieren.

Für die einfache Ansteuerung sorgt ein SSD1306-Controller. Der SSD1306 unterstützt unter anderem die seriellen Protokolle I2C und SPI und belegt daher nur zwei beziehungsweise vier Pins auf dem Arduino. Allerdings reichen nicht alle Displays die Pins für alle Protokolle nach außen. Das vorliegende Projekt setzt auf I2C und das Display hat auch tatsächlich nur vier Pins, die mit VCC, GND, SCL (Clock Line) und SDA (Data Line) gekennzeichnet sind. Wie gewohnt wird VCC mit dem 5V-Pin des Arduino und GND mit einem GND-Pin verbunden. SDA kommt an den analogen Eingang A4 und SCL gehört an den A5-Pin.



Das Display benötigt 4 Leitungen, zwei davon für I²C.



Die Kennung des angeschlossenen Sensors wird hexadezimal ausgegeben.

OLED-Display ansteuern

Das Protokoll zwischen Arduino und dem Display ist nicht allzu kompliziert. Dennoch wäre es lästig, zeitaufwendig und fehlerträchtig, das ganze Byte-Gefriemel selbst zu implementieren.

Dankenswerter Weise haben andere diese Arbeit bereits erledigt und die Bibliothek Adafruit-SSD1306 (https://github.com/adafruit/Adafruit_SSD1306) veröffentlicht. Diese Bibliothek kapselt alle Interna der SPI- und I2C-Kommunikation und stellt die Funktionen des Displays über eine schlanke Schnittstelle zur Verfügung.

Die Funktionen der SSD1306-Bibliothek beschränken sich auf die reine Hardware-Ebene, das heißt, sie repräsentieren nur die Operationen, die das Display von Hause aus mitbringt. Dazu gehören zum Beispiel die Invertierung der Ausgabe, das Dimmen des Displays oder das Scrollen von Inhalten. Alles andere, wie das Zeichnen von Punkten, Linien, Kreisen, Texten oder Bitmap-Grafiken, muss der Arduino übernehmen.

Erleichterung schafft hier eine weitere Bibliothek, nämlich Adafruit-GFX (https://github.com/adafruit/Adafruit-GFX-Library). Sie bietet eine Vielzahl an Funktionen zur Erstellung von Grafiken und zur Ausgabe von Texten.

Die Installation der beiden Bibliotheken erfolgt am einfachsten über den Library-Manager

Erste Schritte

Ein kleines Programm Hello_World dient dazu, die Funktionstüchtigkeit des Displays zu testen. Dazu bindet es erst einmal alle benötigten Bibliotheken ein. SPI.h und Wire.h werden von den Adafruit-Bibliotheken benötigt und daher zuerst geladen.

Viele Displays verfügen über eine Reset-Leitung, über die der Controller zurückgesetzt werden kann. Im aktuellen Projekt wird sie nicht verwendet, muss bei der Initialisierung aber angegeben werden. Daher wird sie ohne besonderen Grund auf den Pin 5 gesetzt. Der wird anschließend bei der Definition des Adafruit_SSD1306-Objekts mit dem Namen display übergeben.

Die setup-Funktion beginnt mit der folgenden Anweisung:

```
display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
```

Hier wird das Display initialisiert und die hexadezimale Zahl 0x3C (dezimal 60) als ID des anzusprechenden I2C-Geräts verwendet. Wie beim 1-Wire-Protokoll werden auch bei I2C alle angeschlossenen Geräte über eine Zahl identifiziert. Die überwiegende Mehrheit der Displays haben die ID 0x3C. Oft ist die ID auf der Platine des Displays vermerkt. Aber Achtung: Auf den Displays stehen in der Regel 8-Bit-Adressen, während der Arduino eine 7-Bit-Adresse verwendet. Auf dem eingesetzten Display steht beispielsweise die Adresse 0x78. Schiebt man die um ein Bit nach rechts (dividiert sie also durch 2), ergibt das 0x3c.

Danach sorgt die Funktion `clearDisplay` dafür, dass der aktuelle Inhalt des Bildschirms gelöscht wird. Die folgenden beiden Anweisungen setzen die Textfarbe auf Weiß (`setTextColor`) und die Textgröße auf 1 (`setTextSize`). Bei dieser Textgröße haben die Zeichen eine Breite von sechs Pixeln und sind acht Pixel hoch. Die weiteren Textgrößen sind jeweils Vielfache von Sechs, also haben Zeichen mit der Textgröße 4 eine Breite von 24 Pixeln.

Der Aufruf von `setCursor` positioniert den Cursor pixelgenau auf eine XY-Koordinate und `print` gibt schließlich den Text „Hello, world!“ aus. Der Text umfasst 13 Zeichen, das Display ist 128 Pixel breit und ein einzelnes Zeichen beansprucht 6 Pixel. Um den Text horizontal zu zentrieren, muss er an der X-Koordinate $(128 - 13 \times 6) / 2 = 25$ ausgegeben werden. Analoges ergibt sich für die Y-Koordinate: $(64 - 8) / 2 = 28$.

Mit der `display`-Funktion wird schließlich der Inhalt des Displays ausgegeben. In der `loop`-Funktion ist nichts zu tun, weil das Display den einmal übertragenen Inhalt so lange anzeigt, bis er verändert wird.

Das Thermometer-Projekt

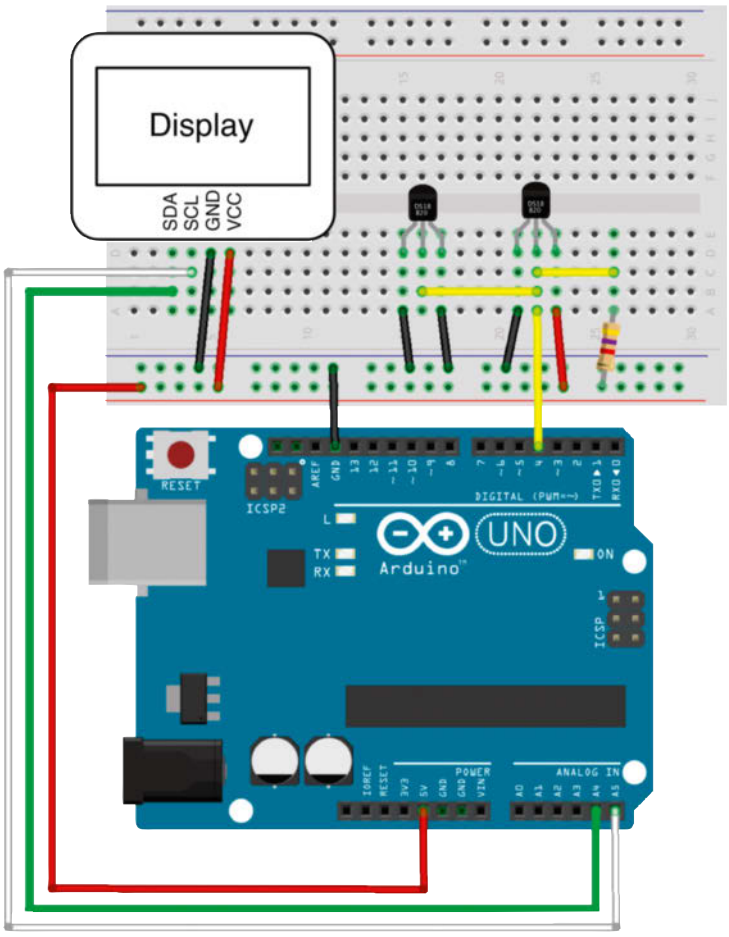
Auch die Ausgabe von Zahlen ist mit den Display-Bibliotheken kein Problem und das ist perfekt für die Umsetzung des Thermometers mit zwei Temperatursensoren. Dazu müssen die zunächst mit dem Arduino verbunden werden.

Wie das geht, zeigt die Thermometer-Schaltung. Die enthält zwei alte Bekannte, denn das Display wird wie gehabt angeschlossen und auch die Anbindung des rechten Thermometers unterscheidet sich nicht vom bisherigen Vorgehen im Projekt „Temperatur Messen“.

Bleibt nur zu klären, ob es beim linken der beiden DS18B20-Sensoren irgendwelche Besonderheiten gibt. Sein Signal-Pin ist mit dem des rechten Sensors verbunden und so kommunizieren beide Sensoren über denselben 1-Wire-Bus. Auch beim Masse-Anschluss gibt

Hello_World

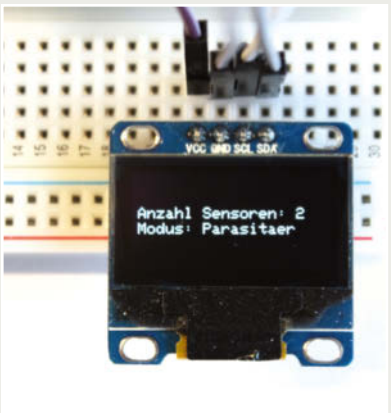
```
#include <SPI.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

const unsigned char OLED_RESET = 5;

Adafruit_SSD1306 display(OLED_RESET);

void setup() {
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
  display.clearDisplay();
  display.setTextColor(WHITE);
  display.setTextSize(1);
  display.setCursor(25, 28);
  display.print("Hello, world!");
  display.display();
}

void loop() { }
```



Ein Thermometer mit zwei Temperatursensoren

Thermometer

```
 1 #include <OneWire.h>
 2 #include <DallasTemperature.h>
 3 #include <SPI.h>
 4 #include <Wire.h>
 5 #include <Adafruit_GFX.h>
 6 #include <Adafruit_SSD1306.h>
 7
 8 const unsigned char ONE_WIRE_BUS = 4;
 9 const unsigned char OLED_RESET = 5;
10
11 Adafruit_SSD1306 display(OLED_RESET);
12 OneWire oneWire(ONE_WIRE_BUS);
13 DallasTemperature sensoren(&oneWire);
14
15 // Die folgenden beiden Werte müssen an die eigenen Sensoren
16 // angepasst werden.
17 DeviceAddress innenThermometer = { 40, 255, 221, 30, 101, 21, 2, 232 };
18 DeviceAddress aussenThermometer = { 40, 255, 1, 17, 101, 21, 2, 13 };
19
20 void setup() {
21   sensoren.begin();
22   display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
23   display.clearDisplay();
24   display.setTextSize(1);
25   display.setTextColor(WHITE);
26   display.setCursor(4, 23);
27   display.print("Anzahl Sensoren: ");
28   display.setCursor(106, 23);
29   display.println(sensoren.getDeviceCount());
30   display.setCursor(4, 33);
31   display.print("Modus: ");
32   display.setCursor(46, 33);
33   if (sensoren.isParasitePowerMode()) {
34     display.print("Parasitaer");
35   } else {
36     display.print("Normal");
37   }
38   display.display();
39   delay(3000);
40 }
41
42 void loop() {
43   sensoren.requestTemperatures();
44   float innenTemperatur = sensoren.getTempC(innenThermometer);
45   float aussenTemperatur = sensoren.getTempC(aussenThermometer);
46
47   display.clearDisplay();
48   display.setCursor(24, 0);
49   display.println("Temperaturen");
50   display.drawLine(0, 9, display.width(), 9, WHITE);
51
52   display.setCursor(1, 27);
53   display.print("Aussen: ");
54   display.setCursor(44, 27);
55   display.print(aussenTemperatur);
56   display.setCursor(77, 27);
57   display.print("C");
58
59   display.setCursor(1, 37);
60   display.print("Innen: ");
61   display.setCursor(44, 37);
62   display.print(innenTemperatur);
63   display.setCursor(77, 37);
64   display.print("C");
65
66   display.display();
67   delay(1000);
68 }
```



Beim Start zeigt der Sketch Informationen über die gefundenen Sensoren an.

es keine Überraschungen, denn der ist mit der GND-Leitung des Arduino verbunden.

Interessanterweise ist aber auch der VCC-Anschluss des Sensors mit dem GND-Pin verbunden und das ist durchaus beabsichtigt. Damit wird der Sensor nämlich in den parasitären Modus versetzt und kann mit nur zwei Leitungen betrieben werden. Für das Außen-Thermometer ist das optimal, insbesondere, weil auch die Länge der Kabel ausreichend lang gewählt werden kann.

Die Software Thermometer birgt ebenfalls keine großen Überraschungen und baut weitestgehend auf den bisherigen Beispielen auf. Zusätzlich zu den zuvor schon definierten globalen Variablen definiert sie in den Zeilen 17 und 18 zwei Objekte vom Typ `DeviceAddress`. Die enthalten jeweils die vorher ermittelten Sensor-Kennungen für die beiden DS18B20-Sensoren (bitte im eigenen Sketch anpassen).

Die `setup`-Funktion initialisiert mit den jeweiligen `begin`-Funktionen sowohl die Temperatur-Sensoren als auch das Display. Dann gibt sie die Informationen, die zuvor auf der seriellen Schnittstelle ausgegeben wurden, auf dem Display aus. Beim Starten des Thermometers wird ausgegeben, wie viele Sensoren am 1-Wire-Bus angeschlossen wurden und in welchem Modus der Bus betrieben wird.

In `loop` werden mittels `sensor.requestTemperatures` die von den beiden Sensoren gemessenen Temperaturen ermittelt. Anschließend werden sie mit der Funktion `getTemp` den Variablen `innenTemperatur` und `aussenTemperatur` zugewiesen. Zur Identifikation der Sensoren dienen die anfangs ermittelten Sensor-Kennungen.

Der Rest des Programms bereitet lediglich die Ausgabe hübsch auf. Beispielsweise wird auf dem Display eine Überschrift ausgegeben, die mittels einer horizontalen Linie von den eigentlichen Werten abgetrennt wird. Das Zeichnen der Linie übernimmt die Funktion `drawLine` in Zeile 50.

Fazit

Das Projekt vereinigt schon ein gerüttelt Maß an Komplexität. Die beiden Temperatur-Sensoren hängen gemeinsam an einem 1-Wire-Bus während das Display via I2C mit dem Arduino kommuniziert. Trotzdem wird das alles diskret hinter den Kulissen versteckt und mit nur wenig Code schick in Szene gesetzt.

Ein paar Mini-Displays sollte jeder Bastler immer in der Grabbelkiste haben, denn sie können mit wenig Aufwand die Attraktivität und Flexibilität von Projekten erhöhen. Ausgabe 5/15 des Make-Magazins zeigt ab Seite 126, wie sich mit dem kleinen Display auch grafisch anspruchsvollere Projekte wie Video-Spiele umsetzen lassen. *—dab*

Ohrenschmaus

Der Arduino erzeugt dank eines Lautsprechers Töne und spielt Sounds.

Die bisherigen Projekte setzten vornehmlich auf visuelle Ausgaben, also auf leuchtende LEDs und rotierende Motoren. Genauso wichtig sind aber auch Audio-Signale, die von einfachen Pieptönen bis hin zur Wiedergabe aufgezeichneter menschlicher Sprache reichen können.

Der geringe Speicher des Arduino setzt der Verarbeitung aufgezeichneter Klänge enge Grenzen. Trotzdem lassen sich dem Board ohne großen Aufwand interessante und nützliche akustische Effekte entlocken.

Make Some Noise!

Schall ist eine Schwingung, die durch die Veränderung des Luftdrucks entsteht. Eine solche Schwingung lässt sich mit der Membran auch eines sehr kleinen Lautsprechers oder eines Piezo-Summers erzeugen. Auf natürliche Weise erzeugte Schallwellen, wie zum Beispiel menschliche Sprache oder Musik, sind recht komplex und folgen keinen einfachen Mustern. Schall lässt sich aber auch leicht synthetisch erzeugen.

Für erste Experimente reicht ein Mini-Lautsprecher, der direkt mit dem Arduino verbunden wird. Bei vielen Lautsprechern müssen zuvor noch zwei Drähte angelötet werden. Es gibt sie aber auch schon fertig verkabelt, bei vielen Anbietern ist leider nicht ersichtlich, wie das Bauteil geliefert wird.

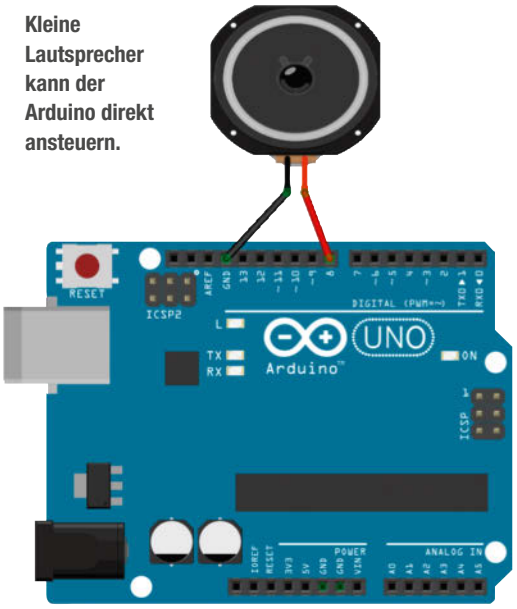
Mit nur zwölf Zeilen Code im Listing Saegezahn lässt sich eine so genannte Sägezahn-Kurve über den Lautsprecher ausgeben. Woher der Name Sägezahn stammt, wird beim Betrachten der Kurve schnell klar. Das Audio-Signal steigt von 0 steil bis zum Maximum an und fällt dann schlagartig wieder auf 0 zurück. Danach beginnt das Spiel von vorn, so dass die Kurve an ein Sägeblatt erinnert.

Der Verlauf der Kurve spiegelt sich weitestgehend auch im Code wider. Nachdem die Konstante `LAUTSPRECHER_PIN` initialisiert und in den Ausgabe-Modus versetzt wurde, kümmert sich die `loop`-Funktion um die Erzeugung der Töne. Dazu dient eine `for`-Schleife, die den Ausgabewert des Lautsprechers alle zehn Millisekunden um eins erhöht. Wenn die Schleife beendet ist, wird sie durch die `loop`-Funktion erneut aufgerufen.

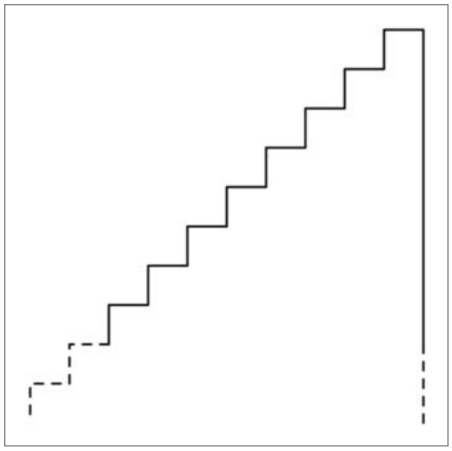
Der erzeugte Ton ist recht markant und die zugrunde liegende Schwingung wieder-

Saegezahn

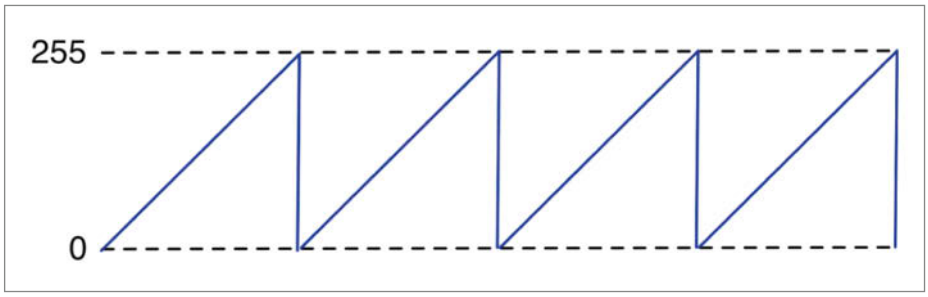
```
1  const unsigned char LAUTSPRECHER_PIN = 9;
2
3  void setup() {
4    pinMode(LAUTSPRECHER_PIN, OUTPUT);
5  }
6
7  void loop() {
8    for (unsigned char i = 0; i < 255; i++) {
9      analogWrite(LAUTSPRECHER_PIN, i);
10     delay(10);
11   }
12 }
```



Kleine Lautsprecher kann der Arduino direkt ansteuern.



Das Sägezahnsignal ist im Detail eigentlich trepppenartig.



Ein Sägezahnsignal am Ausgang des Arduino

holt sich alle 256 * 10 = 2560 Millisekunden. Das Endergebnis ist allerdings keine saubere Sägezahnkurve. Vielmehr wird der ansteigende Teil der Kurve über kleine Treppenstufen angenähert.

Smoooooth

Der erzeugte Ton klingt schon ganz gut, ist aber noch ein wenig eckig. Das ist ein häufiges Problem bei synthetischen Klängen. Insbesondere, wenn sie per Pulsweitenmodulation erzeugt wurden. Bei runden Kurven, wie zum Beispiel Sinusschwingungen, fällt das nicht so sehr ins Gewicht, aber Sägezahn- und Rechteckkurven klingen schon sehr künstlich. Der Grund dafür sind ungewollte Obertöne, die in Kombination mit dem eigentlichen Grundton etwas blechern klingen.

Ein passender Filter kann die unerwünschten Obertöne eliminieren. Einen Filter, der hohe Frequenzbereiche abschneidet und nur die niedrigeren passieren lässt, heißt Tiefpassfilter (englisch: low-pass filter). Der kann sowohl als Software als auch als Hardware umgesetzt werden. Bei der Hardware-Lösung reichen bereits ein Keramik-Kondensator (0,1uF) und ein Widerstand (1kOhm), um den Ton zu glätten. Die Schaltung implementiert einen Tiefpassfilter und entfernt die Störenfriede. Wie Widerstände haben auch Keramik-Kondensatoren keine Polung und können in beliebiger Richtung in die Schaltung eingesetzt werden. Bei Elektrolyt-Kondensatoren ist das nicht der Fall.

Mit dem Tiefpassfilter klingt der Ton runder, er ist aber auch deutlich leiser. Hier hilft bei Bedarf nur ein externer Verstärker.

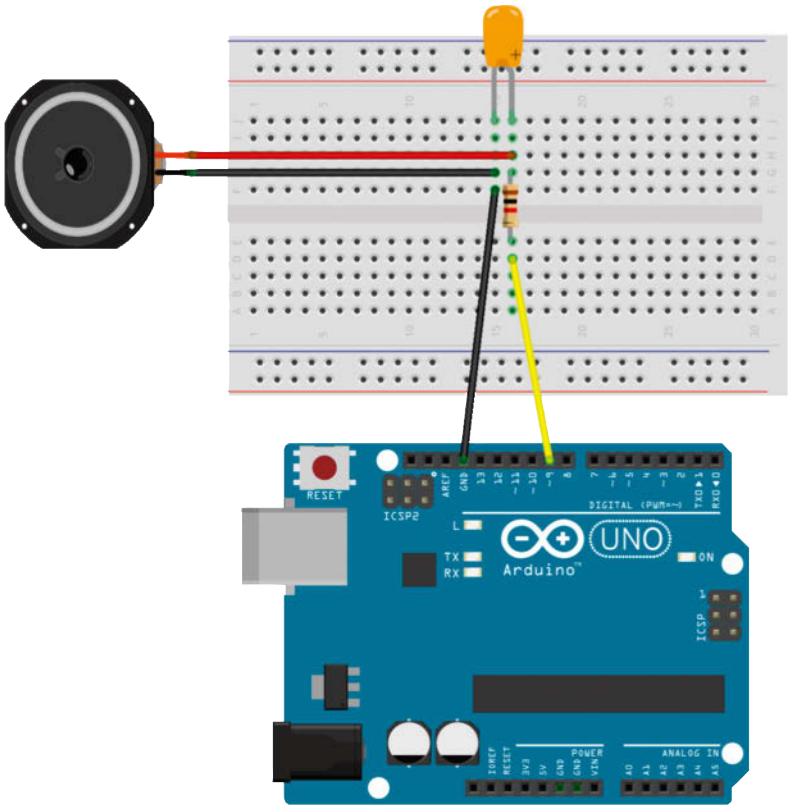
Mehr Struktur

Das erste Beispiel-Programm ist mehr oder weniger chaotisch und dient lediglich als Machbarkeitsstudie. Prinzipiell lassen sich also Töne mit einem Arduino und einem Lautsprecher erzeugen. Im Folgenden geht es darum, den Prozess in geordnete Bahnen zu lenken und nicht nur irgendwelche Töne zu erzeugen, sondern vorgegebene Noten mit festen Frequenzen wiederzugeben.

Das Programm Melodie ist eine abgewandelte Version des Codes, der sich in der Arduino-IDE unter Beispiele > 02.Digital > toneMelody findet. Es spielt eine kurze Melodie, die aus acht Noten besteht, und nutzt dazu die `tone`-Funktion. Zuerst bindet es die Datei `Tonhoehe.h` ein, die im selben Verzeichnis liegt, wie das Programm. Diese Datei definiert eine Enumeration (Aufzählung) namens `Tonhoehe` mit dem Schlüsselwort `enum`. Sie sieht wie folgt aus:

Melodie

```
#include "Tonhoehe.h"

const unsigned char LAUTSPRECHER_PIN = 9;
const unsigned int GANZE_NOTE = 1000;

struct Note {
  Tonhoehe frequenz;
  unsigned char laenge; // 4 = Viertelnote, 8 = Achtelnote etc.
};

Note melodie[] = {
  { NOTE_C4, 4 }, { NOTE_G3, 8 },
  { NOTE_G3, 8 }, { NOTE_A3, 4 },
  { NOTE_G3, 4 }, { PAUSE,   4 },
  { NOTE_B3, 4 }, { NOTE_C4, 4 }
};

void setup() { }

void loop() {
  unsigned int anzahl_noten = sizeof(melodie) / sizeof(melodie[0]);
  for (unsigned int note = 0; note < anzahl_noten; note++) {
    unsigned int notenLaenge = GANZE_NOTE / melodie[note].laenge;
    tone(LAUTSPRECHER_PIN, melodie[note].frequenz, notenLaenge);
    unsigned int pauseZwischenNoten = notenLaenge * 1.30;
    delay(pauseZwischenNoten);
  }
  delay(3000);
}
```



Der Widerstand und der Kondensator bilden einen Tiefpassfilter.

```
enum Tonhoehe {
      PAUSE = 0,
      NOTE_B0  = 31,
      NOTE_C1  = 33,
      NOTE_CS1 = 35,
      NOTE_D1  = 37,
      NOTE_DS1 = 39,
      ...
      NOTE_CS8 = 4435,
      NOTE_D8  = 4699,
      NOTE_DS8 = 4978,
   };
```

Enumerationen sind ein probates Mittel, um lange Listen von Konstanten zusammenzufassen und mit einem Typen zu versehen. Sie eignen sich zum Beispiel, um einen eigenen Datentypen für alle möglichen Wochentage zu definieren. Aber auch für die Definition von Noten in verschiedenen Oktaven sind sie nützlich. `Tonhoehe` weist den einzelnen Mitgliedern der Enumeration die zum jeweiligen Ton gehörende Frequenz zu. Der Wert `PAUSE` bekommt die Frequenz 0 und `NOTE_A4` repräsentiert den Standard-Kammerton mit der Frequenz 440 Hertz.

Im Programm geht es mit der Definition von zwei Konstanten weiter. `LAUTSPRECHER_PIN` legt fest, mit welchem Pin der Lautsprecher verbunden ist. `GANZE_NOTE` definiert die Länge einer ganzen Note gemessen in Millisekunden.

Es folgt die Deklaration einer Struktur (`struct`) namens `Note`, die alle wichtigen Eigenschaften einer Note zu einem Ganzen zusammenfasst. Das sind im Wesentlichen die Frequenz und die Länge. Die Länge wird in diesem Fall als Teiler einer ganzen Note angegeben, also steht 4 für eine Viertelnote, 8 für eine Achtelnote und so weiter.

Als Nächstes wird die eigentliche Melodie beschrieben und zwar in Form eines Feldes von Note-Strukturen. Das Feld heißt `melodie` und die beiden eckigen Klammern ([]) kennzeichnen `melodie` als Feld. Die Länge des Feldes berechnet die Arduino-Umgebung in diesem Fall automatisch. Jeder Eintrag des Felds ist eine Note-Struktur und die Werte ihrer Eigenschaften (`frequenz` und `laenge`) werden in geschweifte Klammern geschrieben und mit einem Komma separiert. Insgesamt besteht die Melodie aus acht Noten, wobei die erste die Viertelnote C4 und die zweite die Achtelnote G3 ist.

In der `setup`-Funktion ist gar nichts zu tun, weil die im Folgenden verwendete `tone`-Funktion sich um die Initialisierung des Lautsprecher-Pins kümmert.

Die `loop`-Funktion berechnet zuerst die Anzahl der Noten im Feld `melodie`. Dazu verwendet sie einen gängigen Trick und den `sizeof`-Operator. Der liefert die Anzahl der Bytes, die eine Variable im Speicher belegt, zurück. Die genaue Anzahl der Bytes spielt in diesem Fall keine wichtige Rolle, denn es geht nur darum, die Anzahl der Elemente in `melodie` zu ermitteln. Diese ergibt sich, wenn die Größe des gesamten Felds durch die Größe eines einzelnen Elements dividiert wird, weil alle Elemente des Felds dieselbe Größe haben.

Anschließend schnappt sich eine `for`-Schleife jede einzelne Note und berechnet deren Dauer, indem die Dauer einer ganzen Note durch die Dauer der aktuellen Note dividiert wird. Für die erste Note ergibt sich zum Beispiel ein Wert von 1000 / 4 = 250 Millisekunden.

Mit der `tone`-Funktion wird die aktuelle Note dann über den Lautsprecher ausgegeben. Sie erwartet die Nummer des Lautsprecher-Pins, die Frequenz des Tons und die Dauer des Tons. Wird die Dauer nicht angegeben, gibt `tone` den Ton so lange aus, bis die Funktion `noTone` aufgerufen wurde.

Die restlichen Anweisungen in der `for`-Schleife sorgen für eine kurze Pause, die zwischen den einzelnen Noten eingelegt wird.

Wird das Programm auf den Arduino geladen, spielt er alle drei Sekunden eine kurze Melodie, die ein wenig an die Spielautomaten in Imbiss-Buden der Achtziger Jahre erinnert. Es lohnt sich, ein wenig mit dem Wert der Konstanten `GANZE_NOTE` zu spielen, denn damit lässt sich die Abspielgeschwindigkeit der Melodie steuern.

Kern des letzten Beispiels ist die `tone`-Funktion. Sie erzeugt per Pulsweitenmodulation ein Rechteck-Signal mit einer vorgegebenen Frequenz und Dauer. Das unterscheidet sich nicht von den Signalen, die zuvor genutzt wurden, um Motoren zu steuern. Die Signale werden jetzt nur nicht mehr von einem Motor, sondern von einem Lautsprecher interpretiert.

Fortgeschrittene Projekte

Die Erzeugung von Schallwellen ist mit dem Arduino erst einmal kein Problem. Komplizierter ist es, gezielt wirklich gute Klänge zu erzeugen. Dazu bedarf es recht komplizierter Software, bei der es auf das richtige Timing bei der Erzeugung von Frequenzen ankommt. Doch auch dazu gibt es jede Menge hilfreicher und nützlicher Bibliotheken.

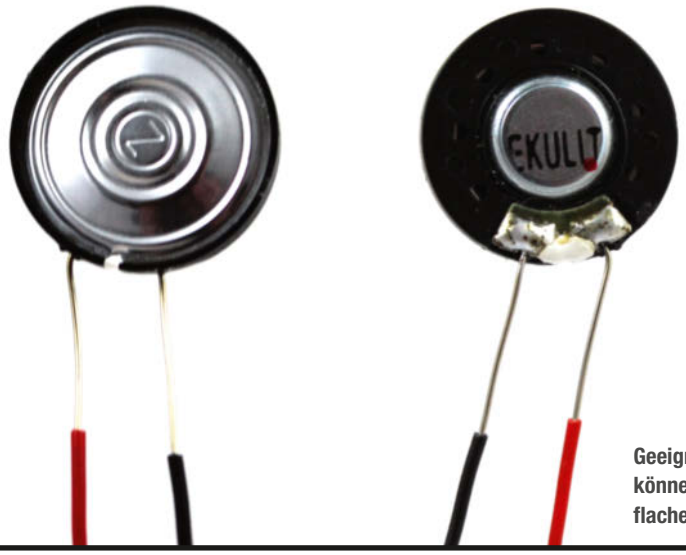
Eines der umfangreichsten Projekte ist Mozzi (http://sensorium.github.io/Mozzi/). Es bietet technikbegeisterten Musikern alles, was das Herz begehrt. Beispielsweise gibt es Funktionen zur Erzeugung synthetischer Klänge aller Art und auch die Wiedergabe von Samples ist problemlos möglich.

Darüber hinaus bietet die Bibliothek eine große Anzahl von Sound-Filtern und verwandelt den Arduino bei Bedarf sogar in ein MIDI-Gerät. Der Preis für die Funktionsvielfalt ist eine recht hohe Komplexität und eine teilweise steile Lernkurve.

Dafür ist das Projekt aber üppig dokumentiert und beinhaltet eine Vielzahl an Beispiel-Projekten. Diese finden sich nach der Installation der Zip-Datei über den Library-Manager im Menü Datei > Beispiele. Um die meisten der Beispiele auszuprobieren, reicht es völlig aus, den Arduino direkt mit einem Mini-Lautsprecher zu verbinden. In der Regel verwenden die Beispiele den digitalen Pin 9.

Fazit

Wie in vielen anderen Bereichen, weiß der Arduino auch bei der Erzeugung von Klängen zu überraschen. Es ist erstaunlich, was ein wenig kreative Software an Sounds aus dem Winzling rausholt. Darüber hinaus katapultiert der Einsatz spezieller Hardware, wie etwa diverse Sound-Shields, den Arduino schnell in eine ganz andere Liga. —*dab*



Geeignete Lautsprecher können auch eine sehr flache Bauform haben.

Uno, due, tres

Der Arduino Uno ist zwar das bekannteste, aber nicht das einzige Mitglied einer ganzen Familie. Wem die Ausstattung mit Pins zu wenig, der Speicher zu klein, die Platine zu groß oder die Bestückung mit Sonderfunktionen zu sparsam ist, der kann aus einem großen Angebot weiterer Modelle wählen, passend zum jeweiligen Projekt.

von Daniel Bachfeld

In verkleinerter Bauform gibt es den Arduino in den Ausführungen Micro, Pro Mini und Nano. Sie unterscheiden sich im Vergleich zum Uno durch das Fehlen der Anschlussleisten, einem kleineren (oder fehlendem) USB-Port und der knapp viermal kleineren Platine. Die eingesetzten Mikrocontroller sind die gleichen oder haben zumindest den gleichen Funktionsumfang.

In gleicher Baugröße wie den Uno hat Arduino kürzlich den 101 vorgestellt. Er enthält statt eines ATMega-Mikrocontrollers das Curie-Modul. Der System-on-a-Chip enthält einen 32-Bit-Prozessor aus Intels Quark-Schiene, dem 80 KByte SRAM und 384 KByte Flash-Speicher zur Seite stehen. Daneben sind bereits ein 6-Achsen-Sensor sowie eine Bluetooth-LE-Einheit verbaut. Damit frischt er die funktionsmäßig in die Jahre gekommene Uno-Klasse wieder auf.

Wer mehr digitale Ein- und Ausgänge oder Speicher benötigt, der greift zum Arduino Mega 2560. Er hält 54 I/O-Pins (16 davon analog und 15 PWM) bereit, mit denen sich auch größere Steuerungen umsetzen lassen. Sein 256 KByte großer Speicher ist auch größeren Sketches gewachsen.

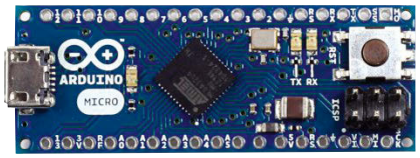
Wer statt mehr Pins mehr Rechenleistung benötigt, der greift zum Arduino Zero. Statt eines 8-Bit-Mikrocontrollers zieht ein 32-Bit-ARM-Cortex-M0+ mit 48 MHz die Strippen. Er hat sogar einen echten Digital-Analog-Wandler (D/A) an Board, um Spannungen in 1024 Schritten ausgeben zu können.

Eine Kombination aus Zero und Mega stellt der Arduino Due dar. Er hat einen ARM-Cortex-M3 mit 84 MHz an Bord und stellt 54 I/O-Pins für PWM, A/D und D/A bereit. Seine 512 KByte Flash nehmen auch sehr große Sketches auf.

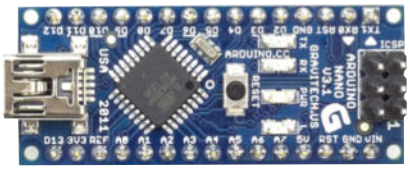
In der Brust des Yún schlagen zwei Herzen: ein MIPS-Prozessor mit 400 MHz und WLAN-Anbindung und Ethernet-Anbindung sowie ein ATmega32U4. Auf dem MIPS-Prozessor läuft ein Linux, das mit dem ATmega Daten austauschen kann.

Geht es um Elektronik in Kleidung, ist oft Minimalismus gefragt: Das Lillypad liefert einen ersten Ansatz, um diverse Elektronik in Textilien zu kontrollieren. Der Arduino Gemma ist dann nochmals geschrumpft, bietet aber im Kern ähnliche Funktionen wie ein Arduino Uno, nur alles reduziert: 3 × I/O, 2 × PWM und 1 × ADC bei einem Takt von 8 MHz.

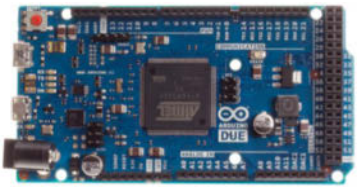
Arduino hat seit längerem den Tres angekündigt, der wie der Yún aus zwei Prozessoren besteht. Der Hauptprozessor soll der gleiche sein, der auch im Beaglebone Black eingesetzt wird. *—dab*



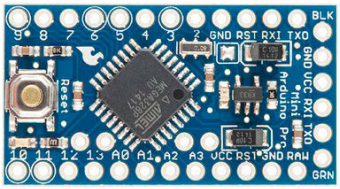
Arduino Micro



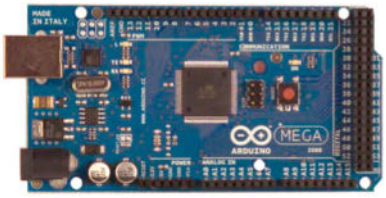
Arduino Nano



Arduino Due

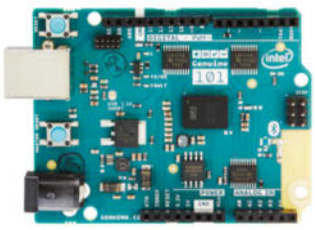


Arduino Pro Mini



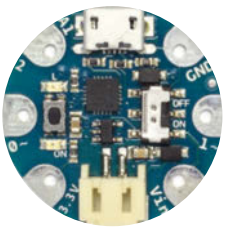
Arduino Mega 2560

Arduino Yún

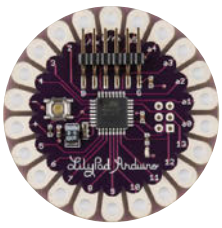


Arduino 101

Arduino Zero



Arduino Gemma



Arduino Lillypad

Protokolle von der Stange

Dank der Firmata-Software muss man den Arduino nur einmal programmieren und kann fortan seine Ports über ein standardisiertes Protokoll vom PC oder anderen Minicomputern steuern und lesen, die selbst keine I/O-Ports haben.

von Maik Schmidt

Links und Foren
make-magazin.de/x5eu

Schon nach wenigen Projekten wird klar, dass die Software vieler Arduino-Projekte im Kern gleich aussieht: Der Arduino kommuniziert über die serielle Schnittstelle mit einem PC, der über diese Verbindung Daten von Sensoren abfragen kann. Zusätzlich kann der PC Aktoren, die mit dem Arduino verbunden sind, Kommandos senden.

Bei ersten Gehversuchen erfolgt die Kommunikation über den seriellen Monitor des Arduino. Der reicht für viele Anwendungen aber nicht aus und so bleibt nur die Programmierung eigener Software auf dem PC, die den Arduino über die serielle Schnittstelle einbindet.

Das ist gar nicht so schwierig, wie es klingt, weil jede moderne Programmiersprache über mindestens eine Bibliothek zur seriellen Kommunikation verfügt. Allerdings sind diese Bibliotheken sehr allgemein gehalten und sie unterstützen alle möglichen Geräte und nicht nur den Arduino.

So richtig praktisch wäre eine Bibliothek, die dem PC den vollen Zugriff auf den Arduino „vorgaukelt“ und die gesamte serielle Kommunikation versteckt. Genau diesen Zweck erfüllt das Firmata-Protokoll.

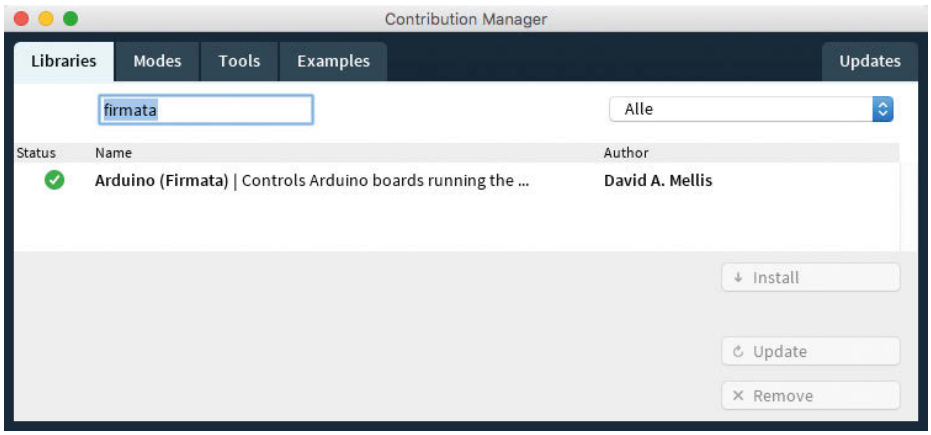
Was ist Firmata?

Firmata ist ein Protokoll, mit dem sich Zustände von Mikrocontroller-Boards wie dem Arduino besonders leicht übertragen und beeinflussen lassen. Etwas Ähnliches gab es in einem der vorherigen Beispiel-Projekte. Dort sendete ein Arduino-Programm unentwegt den aktuellen Zustand aller digitalen und analogen Eingänge. Die könnte ein PC-Programm dann leicht verarbeiten.

Diese Lösung hat allerdings zwei Nachteile: Die Daten wurden als Texte übertragen und benötigen unverhältnismäßig viel Platz und müssen auf der Empfängerseite mühsam zerlegt werden. Darüber hinaus lässt sich der Zustand des Arduino auf diese Weise nicht verändern, denn es fehlt noch die Gegenrichtung, in der der PC dem Arduino Kommandos senden kann.

Firmata behebt beide Nachteile, denn es verwendet ein kompaktes Kommunikationsprotokoll und bietet die Kommunikation in beide Richtungen an.

Im Folgenden wird Schritt für Schritt ein analoges Thermometer entwickelt. Es zeigt die aktuelle Temperatur sowohl auf dem PC-Bildschirm als auch mittels eines Servo-Motors, der mit dem Arduino verbunden ist, an. Während der gesamten Entwicklung wird dabei keine Zeile Code für den Arduino geschrieben, weil das Firmata-Protokoll schon alles bietet, was zur Umsetzung eines solchen Projekts benötigt wird.



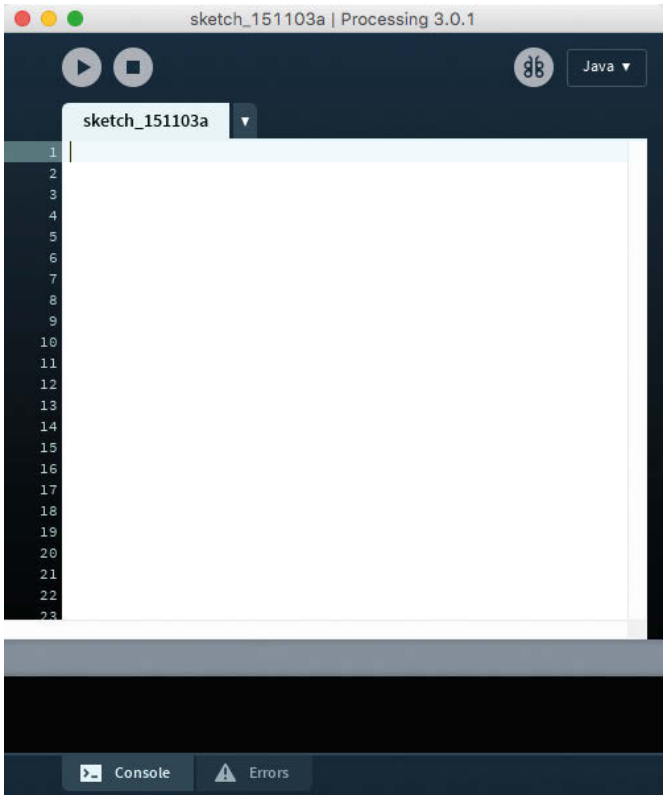
Auch Processing hat ein Tool zum Nachinstallieren von Bibliotheken.

Vorbereitungen

Für ein erstes Beispiel muss erneut das Blinken der Status-LED herhalten. Diesmal wird es aber komplett anders umgesetzt als zuvor. Dazu muss zuerst das Programm in der Arduino-IDE unter Datei > Beispiele > Firmata > StandardFirmata auf den Arduino geladen werden.

Mit dem StandardFirmata-Programm werden alle Projekte in diesem Kapitel umgesetzt, das heißt, es muss kein anderes Programm mehr auf den Arduino übertragen werden. StandardFirmata ermöglicht den lesenden und schreibenden Zugriff auf alle Ein- und Ausgänge des Arduino über das Firmata-Protokoll. Wenn dieses Programm auf dem Arduino läuft, können entsprechende Firmata-Programme auf dem PC die gesamte Bandbreite der Arduino-Funktionen nutzen.

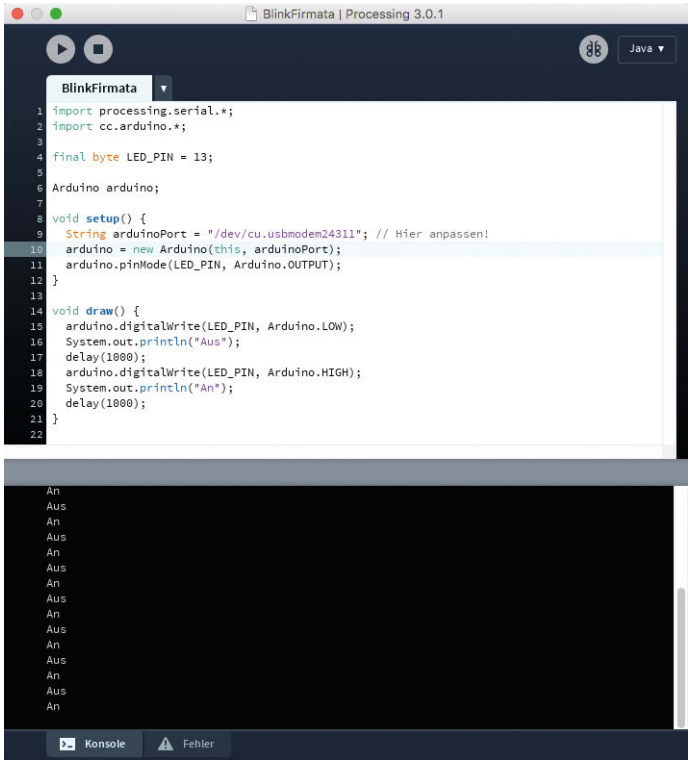
Solche Programme können in beinahe jeder Programmiersprache geschrieben werden, denn es gibt Firmata-Bibliotheken für die meisten. Die Beispiel-Programme dieses Artikels verwenden die Programmiersprache Processing (http://processing.org), weil sie



Die Oberfläche von Processing ähnelt stark der Arduino-IDE, kein Wunder, denn letztere stammt von Procesing ab.

BlinkFirmata

```
 1 import processing.serial.*;
 2 import cc.arduino.*;
 3
 4 final byte LED_PIN = 13;
 5
 6 Arduino arduino;
 7
 8 void setup() {
 9   String arduinoPort = "COM3"; // Hier anpassen!
10   arduino = new Arduino(this, arduinoPort);
11   arduino.pinMode(LED_PIN, Arduino.OUTPUT);
12 }
13
14 void draw() {
15   arduino.digitalWrite(LED_PIN, Arduino.HIGH);
16   System.out.println("An");
17   delay(1000);
18   arduino.digitalWrite(LED_PIN, Arduino.LOW);
19   System.out.println("Aus");
20   delay(1000);
21 }
```



Processing öffnet ein kleines Fenster, das man aber in den ersten Beispielen ignorieren kann.

eine hohe Affinität zum Arduino-Projekt hat und weil sie sich hervorragend für die Umsetzung multimedialer Anwendungen eignet.

Processing basiert auf der Programmiersprache Java und läuft auf allen populären Betriebssystemen. Nach dem Download (https://processing.org/download/?processing) muss es nur entpackt werden und kann sogleich gestartet werden. Die Processing-Oberfläche dürfte jedem Arduino-Freund vertraut erscheinen. Die Ähnlichkeiten zwischen den beiden IDEs sind in der Tat kein Zufall, denn die Arduino-IDE basiert auf dem Code der Processing-IDE.

Bevor die Programmierung des ersten Beispiels beginnen kann, muss noch die Firmata-Bibliothek für Processing installiert werden. Genau wie die Arduino-IDE hat auch Processing einen komfortablen Library-Manager, mit dem sich Bibliotheken per Mausklick installieren lassen. Er wird über das Menü Sketch > Library importieren > Library hinzufügen aufgerufen. Gibt man im Suchfeld den Begriff Firmata ein, wird die gewünschte Bibliothek angezeigt und kann durch einen Klick auf den Install-Knopf installiert werden. Danach stehen die Firmata-Funktionen allen Processing-Programmen zur Verfügung.

Blinken aus der Ferne

Das Processing-Programm nutzt die Firmata-Bibliothek, um die Status-LED des Arduino zum Blinken zu bringen. Aus der Ferne betrachtet sieht das Ganze einem Arduino-Programm recht ähnlich, unterscheidet sich aber doch in einigen Aspekten.

Der Code BlinkFirmata beginnt mit zwei import-Anweisungen, die im Kern dasselbe tun, wie die #include-Anweisung in Arduino-Programmen. Sie sorgen dafür, dass die Funktionen bestimmter Bibliotheken im Programm bekannt gemacht werden und verwendet werden können. Hier sind es die Funktionen zur seriellen Kommunikation (processing.serial.*) und zur Verwendung des Firmata-Protokolls (cc.arduino.*).

Die nachfolgende Anweisung definiert eine Konstante namens LED_PIN, die die Pin-Nummer der Status-LED enthält. Das Äquivalent zu const heißt in Processing final und wie beim Arduino gibt es auch in Processing einen Datentypen namens byte. Allerdings unterscheiden sich deren Wertebereiche! In Processing kann eine Variable vom Typ byte Zahlen von –128 bis +127 aufnehmen. Beim Arduino sind es die Zahlen von 0 bis 255.

Anschließend legt das Programm die Variable arduino vom Typ Arduino an. Dieser Typ wird in der Firmata-Bibliothek definiert und kümmert sich um die Kommunikation zwischen der Processing-Anwendung auf dem PC und dem Arduino.

Die nachfolgende setup-Funktion hat in Processing dieselbe Bedeutung wie in der Arduino-Umgebung. Sie wird einmalig beim Start des Programms aufgerufen und dient zur Initialisierung. In diesem Fall initialisiert sie die Verbindung zum Arduino und legt mit der Variablen arduinoPort den Namen der seriellen Schnittstelle fest, über die der Arduino mit dem PC verbunden ist. Dieser Name stimmt mit dem überein, den die Arduino-Umgebung unter dem Menü Werkzeuge > Port angibt. Er muss in jedem Fall an den aktuellen Zustand respektive Namen angepasst werden.

Danach wird der Variablen arduino ein neues Arduino-Objekt zugewiesen, das mit der zuvor definierten seriellen Schnittstelle verbunden ist. Der Bezeichner this bezieht sich auf das aktuelle Processing-Programm und schafft somit eine Verbindung zwischen dem Programm und dem Arduino.

Ab diesem Zeitpunkt verhält sich die Variable arduino wie ein physikalischer Arduino

und man kann zum Beispiel die Funktion pinMode aufrufen, um den digitalen Pin 13 zu einem Ausgang zu machen. Das sieht ganz ähnlich aus, wie in einem regulären Arduino-Programm. Allerdings können Funktionen wie pinMode oder digitalRead nicht einfach so aufgerufen werden. Sie müssen an ein Arduino-Objekt gebunden werden. In diesem Fall ist es das Objekt, das die Variable arduino referenziert. Die Bindung erfolgt, indem der Funktionsname mit einem Punkt an den Variablennamen gehängt wird. Die Anweisung

```
arduino.pinMode(LED_PIN, Arduino.OUTPUT);
```

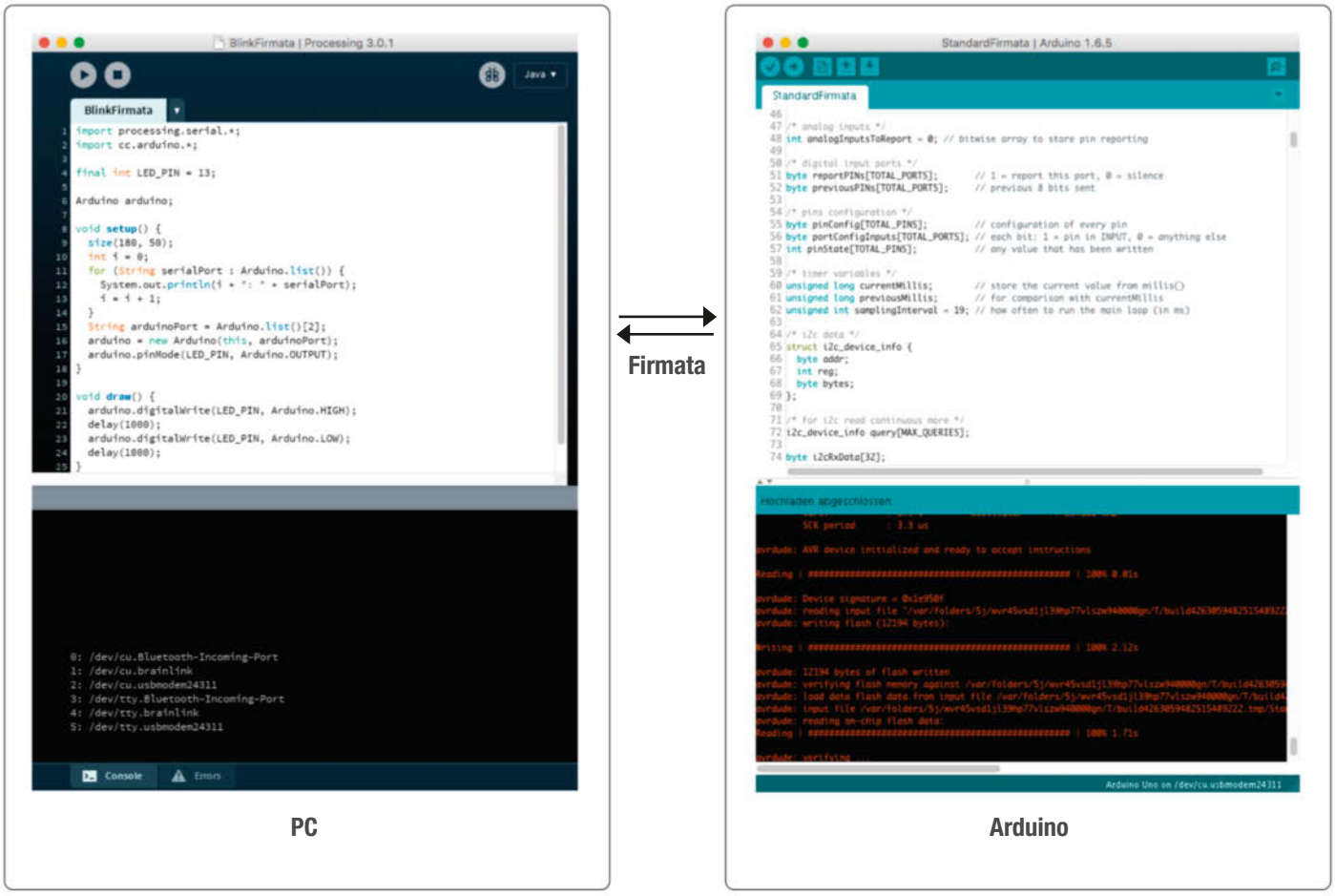
führt die Funktion pinMode also für das Objekt arduino aus. Das wirkt auf den ersten Blick umständlich, hat aber den großen Vorteil, dass sich so ganz leicht mehrere Arduinos vom selben Processing-Programm aus steuern lassen.

Ein kleine Abweichung gibt es noch bei der Benennung der Arduino-Konstanten wie OUTPUT, INPUT, HIGH und LOW. Ihnen muss in Processing-Programmen ein Arduino. vorangestellt werden.

Eine loop-Funktion wie beim Arduino gibt es in Processing nicht. Die Funktion draw erfüllt hier denselben Zweck und wird automa-

ServoSteuernFirmata

```
 1 import processing.serial.*;
 2 import cc.arduino.*;
 3
 4 final byte SERVO_PIN = 9;
 5
 6 Arduino arduino;
 7
 8 void setup() {
 9   String arduinoPort = "COM3"; // Hier anpassen!
10   arduino = new Arduino(this, arduinoPort);
11   arduino.pinMode(SERVO_PIN, Arduino.SERVO);
12 }
13
14 void draw() {
15   arduino.servoWrite(SERVO_PIN, 0);
16   delay(1000);
17   arduino.servoWrite(SERVO_PIN, 45);
18   delay(1000);
19   arduino.servoWrite(SERVO_PIN, 90);
20   delay(1000);
21   arduino.servoWrite(SERVO_PIN, 135);
22   delay(1000);
23   arduino.servoWrite(SERVO_PIN, 180);
24   delay(1000);
25 }
```



Der Processing-Sketch auf dem PC und der Firmata-Sketch auf dem Arduino tauschen Daten zur Kontrolle der Ports aus.

tisch in einer Endlosschleife aufgerufen. Um die Status-LED zum Blinken zu bringen, setzt sie den LED-Pin mit der Funktion digitalWrite auf HIGH. Genau wie zuvor bei der pinMode-Funktion erfolgt der Aufruf auf dem arduino-Objekt.

Anschließend gibt das Programm den Text auf der Processing-Konsole aus. Die Konsole ist der Fensterabschnitt unterhalb des Editor-Fensters und sie eignet sich hervorragend für Debug-Ausgaben und ähnliche Informationen. Die Ausgabe erfolgt mit der Funktion System.out.println, die nicht nur den Text, sondern auch einen Zeilenumbruch ausgibt. Ist der Zeilenumbruch nicht gewünscht, kann alternativ System.out.print eingesetzt werden.

Die Funktion delay verhält sich exakt wie ihr Arduino-Pendant und wartet für eine Sekunde, bis der LED-Pin wieder in den Zustand LOW versetzt wird. Nach einer weiteren Pause beginnt das Spiel dann wieder von vorn.

Wenn der Arduino an den PC angeschlossen ist und das StandardFirmata-Beispiel ausführt, steht einem Testlauf nichts im Wege (wichtig ist nur, den Namen der Schnittstelle in Zeile 9 korrekt zu setzen). Ein Klick auf den Play-Button oben links in der Processing-IDE startet das Programm.

Processing öffnet automatisch ein Fenster für jede Anwendung. In diesem Fall wird das Fenster nicht benötigt und so bleibt es leer und recht klein.

Während das Programm läuft, blinkt nicht nur die Status-LED. Auch die RX- und -TX-LEDs blinken unentwegt, weil permanent Daten zwischen dem PC und dem Arduino ausgetauscht werden. Das ist Teil des Firmata-Protokolls und die Kommunikation mit dem Arduino endet erst, wenn die Processing-Anwendung geschlossen wird.

Das erste Firmata-Programm läuft und es macht schon einen durchaus praktischen Eindruck. Die Software, die auf dem Arduino läuft, kommt sozusagen von der Stange und musste überhaupt nicht angepasst werden. Das eigentliche Programm konnte bequem mit der Programmiersprache Processing umgesetzt werden, die für viele Zwecke deutlich besser ausgestattet ist als die Arduino-Umgebung.

Processing und die Arduino-Umgebung passen hervorragend zusammen und können dank Firmata beide ihrer Stärken ausspielen.

Alles dreht sich

Das nächste Beispiel zeigt, wie sich ein Servo-Motor, der mit dem Arduino verbunden ist, über ein Processing-Programm kontrollieren lässt. Dazu wird der Arduino wie schon zuvor mit dem Servo-Motor verbunden, aber auf dem Arduino läuft weiterhin die StandardFirmata-Anwendung. Es wird keine spezielle Software zur Motorsteuerung für den Arduino geschrieben und alles Wesentliche läuft in der Processing-Umgebung.

Das Programm ServoSteuernFirmata beginnt beinahe genau so, wie das Blink-Programm. Statt der Konstanten LED_PIN definiert es aber die Konstante SERVO_PIN, die den Pin festlegt, über den der Arduino mit dem Servo-Motor verbunden ist. Diese Konstante wird in der setup-Funktion verwendet, um den entsprechenden Pin in den Zustand Arduino.SERVO zu versetzen. Das teilt der Firmata-Software auf dem Arduino mit, das an Pin 9 ein Servo hängt.

Die draw-Funktion ist denkbar einfach und fährt den Servo in die Position 0, 45, 90, 135 und 180 Grad. Nach jeder Aktion wartet sie für eine Sekunde. Nachdem das Processing-Programm gestartet wurde, setzt sich der Motor in Bewegung.

Etwas peppiger

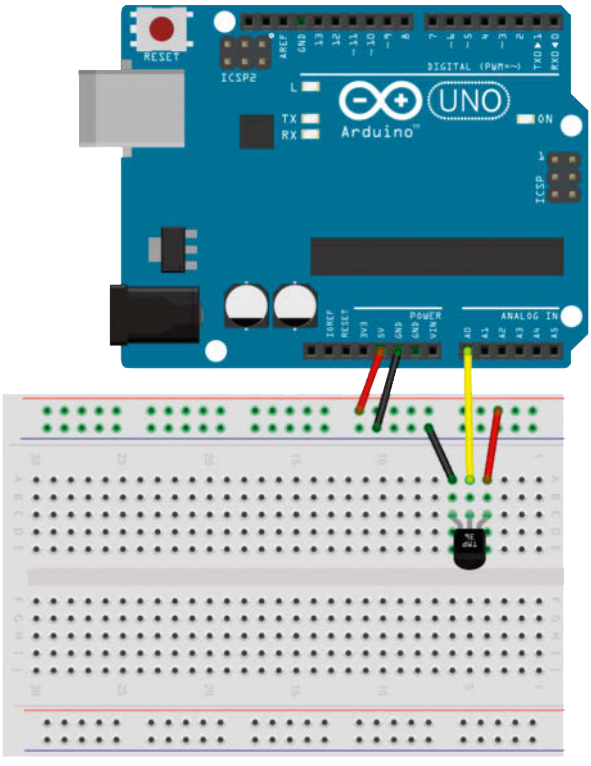
Aktoren wie LEDs und Servo-Motoren lassen sich mittels Processing und Firmata problemlos kontrollieren. Bei Sensoren ist das ähnlich, aber mit dem StandardFirmata-Programm auf dem Arduino ist es am sinnvollsten, rein digitale und analoge Sensoren einzubinden. Ein Nachteil von Firmata ist nämlich, dass die Software auf dem Arduino auf die Fähigkeiten beschränkt ist, die das aktuelle Firmata-Programm vorgibt. Das StandardFirmata-Programm ermöglicht den lesenden und schreibenden Zugriff auf die Ein- und Ausgänge des Arduino. Darüber hinaus unterstützt es die Kontrolle von Servo-Motoren. Für alles andere, wie zum Beispiel für die Kommunikation mit 1-Wire-Geräten wie dem DS18B20-Sensor, bedarf es einer anderen Firmata-Firmware. Die gibt es durchaus, aber Ziel dieses Artikels ist es, so weit wie möglich mit dem StandardFirmata-Beispiel zu kommen.

Für das nächste Beispiel kommt daher ein analoger Temperatursensor wie der TMP36GT9Z (kurz: TMP36) oder der LM35 CZ (kurz: LM35) zum Einsatz. Beide Sensoren sind günstig zu haben und leicht abzufragen. Sie liefern eine Spannung in Abhängigkeit der gemessenen Temperatur. Beim TMP36 lässt sich die gemessene Spannung gemäß der folgenden Formel in einen Celsius-Wert umrechnen:

```
Temperatur = (Spannung (mV) - 500) / 10
```

Von der in Millivolt gemessenen Spannung wird die Zahl 500 subtrahiert und das Ergebnis durch 10 geteilt. Die Subtraktion mag zunächst seltsam erscheinen, sie sorgt aber dafür, dass so auch leicht negative Temperaturen kodiert werden können. Liefert der Sensor beispielsweise 250mV, dann beträgt die korrespondierende Temperatur (250–500) / 10 = –25 Grad. Liefert der Sensor

Processing liest die Temperatur aus und zeigt sie in einem Fenster auf dem PC an.



Die Beschaltung zum Betrieb des Temperatursensors.

TemperaturLesenFirmata

```
 1 import processing.serial.*;
 2 import cc.arduino.*;
 3
 4 final byte THERMOMETER_PIN = 0;
 5 final int ZEICHEN_GROESSE = 32;
 6
 7 Arduino arduino;
 8
 9 void setup() {
10   size(320, 240);
11   String arduinoPort = "COM3"; // Hier anpassen!
12   arduino = new Arduino(this, arduinoPort);
13   PFont zeichensatz = createFont("Arial", ZEICHEN_GROESSE, true);
14   textFont(zeichensatz, ZEICHEN_GROESSE);
15 }
16
17 void draw() {
18   background(255);
19   fill(0);
20   textAlign(CENTER, CENTER);
21   float spannung = arduino.analogRead(THERMOMETER_PIN) * 5000 / 1024.0;
22   float temperatur = (spannung - 500) / 10; // TMP36
23   // float temperatur = spannung / 10;  // LM35 CZ
24   String temperaturText = nf(temperatur, 0, 1);
25   System.out.println(temperaturText);
26   text(temperaturText + " ?", width / 2, height / 2);
27   delay(1000);
28 }
```

800mV, so ist die Temperatur (800–500) / 10 = 30 Grad.

Beim LM35 sieht das ganz ähnlich aus, aber den netten Trick mit der Subtraktion bei Temperaturen unter null beherrscht er nicht. In diesem Fall liefert er tatsächlich negative Spannungen, die mit dem Arduino nicht ohne Weiteres messbar sind. Daher beschränken sich die Beispiele auf positive Temperaturen und die lassen sich für den LM35 wie folgt berechnen:

```
Temperatur = Spannung (mV) / 10
```

Liefert der LM35 also eine Spannung von 321mv, dann liegt die Temperatur bei 32,1 Grad.

Die Schaltung sieht für beide Sensoren gleich aus, denn sie verfügen jeweils über eine Signal-Leitung, einen Masse-Anschluss (GND) und eine Spannungsversorgung (VCC). Die letzten beiden gehören an die GND- und 5V-Pins des Arduinos. Die Signal-Leitung wird mit dem analogen Eingang A0 verbunden.

Ist es heiß hier?

Das Programm TemperaturLesenFirmata gibt die aktuelle Temperatur, die vom Arduino gemessen wurde, in ansprechender Form auf dem Bildschirm aus. Zum ersten Mal nutzt das Beispiel die grafischen Fähigkeiten von Processing.

Trotzdem beginnt der Code wie gehabt mit zwei import-Anweisungen und der Definition von zwei Konstanten. THERMOMETER_PIN legt fest, mit welchem analogen Eingang des Arduino der Temperatursensor verbunden ist. Über ZEICHEN_GROESSE lässt sich die Größe (Breite) des verwendeten Zeichensatzes einstellen. Processing bietet viele Funktionen zur Ausgabe von Texten und unterstützt alle modernen Zeichensatz-Formate.

Das Arduino-Objekt namens arduino dient zur Steuerung des Arduinos.

Die setup-Funktion beginnt diesmal mit dem Aufruf der Funktion size. Sie legt die Größe des Processing-Fensters fest und in diesem Fall ist das Fenster 320 Pixel breit und 200 Pixel hoch. Anschließend wird die Variable arduino initialisiert.

Die folgenden Anweisungen definieren den Zeichensatz, der vom Programm verwendet wird. Der Datentyp PFont repräsentiert Zeichensätze in Processing und createFont erzeugt einen bestimmten Zeichensatz. Hier ist es der Zeichensatz in der Schriftart _Arial_ mit einer Größe von 32 Punkten. Das Argument true legt fest, dass der Zeichensatz mittels Anti-Aliasing geglättet wird.

Mit textFont wird der zuvor initialisierte Zeichensatz als Standard-Zeichensatz für alle weiteren Text-Ausgaben festgelegt.

Die draw-Funktion nutzt die Funktion background, um den Hintergrund des Anwendungsfensters mit der Farbe Weiß zu löschen. Die background-Funktion gibt es in vielen unterschiedlichen Ausprägungen und sie kann zum Beispiel mit drei RGB-Werten (Rot, Grün, Blau) aufgerufen werden, um dem Hintergrund eine beliebige Farbe zu geben. Bekommt sie nur ein Argument, setzt sie die Werte für Rot, Grün und Blau auf den übergebenen Wert. Analog setzt die Funktion fill die aktuelle Zeichenfarbe auf Schwarz.

Der Aufruf von textAlign sorgt dafür, dass die nachfolgenden Text-Ausgaben sowohl horizontal als auch vertikal zentriert erfolgen. So wird die aktuelle Temperatur exakt in der Mitte des Fensters ausgegeben.

Nach der Vorbereitung des Layouts kümmert sich das Programm um die Kommunikation mit dem Arduino. Es nutzt die Funktion analogRead, um die aktuell am analogen Eingang A0 anliegende Spannung zu messen. Darüber hinaus wird der gemessene Wert in Millivolt umgewandelt, indem er mit der Betriebsspannung des Arduino (5000mv) multipliziert und anschließend durch die Auflösung des analogen Eingangs (1024 Werte) dividiert wird.

Die aktuelle Temperatur wird dann gemäß der zuvor beschriebenen Formel ermittelt. Dieser Wert hat in der Regel eine ganze Menge Nachkommastellen und eignet sich nicht zur Ausgabe auf dem Bildschirm. Daher wird er mit der Funktion nf (number format) gerundet und angepasst. Diese Funktion erwartet den zu rundenden Wert, die Anzahl der Stellen links vom Komma und die Anzahl der Stellen rechts vom Komma. Hier ist die Anzahl der Stellen links vom Komma 0 und damit werden alle Stellen links vom Komma ausgegeben. Die Anzahl der Nachkommastellen ist 1.

Die aktuelle Temperatur wird auf der Konsole und im Anwendungsfenster ausgegeben. Für die Ausgabe im Anwendungsfenster kommt die text-Funktion zum Einsatz. Sie erwartet den auszugebenden Text und dessen X-/Y-Koordinaten. Das Zeichen „\u2103“ ist das Unicode-Zeichen für „Grad Celsius“. Das macht ein bisschen mehr her als ein schnödes „C“.

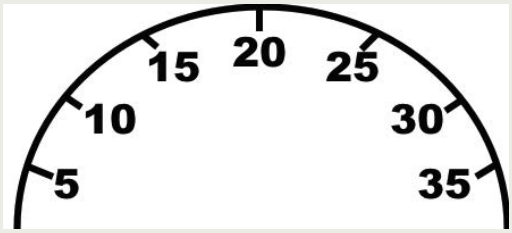
Krönender Abschluss

Mit diesem Rüstzeug steht einem analogen Thermometer, das auf dem Bildschirm angezeigt wird und darüber hinaus in der physikalischen Welt mit einem Motor realisiert wird, nichts mehr im Wege.

Zuerst müssen der Temperatursensor und der Servo-Motor in einer einzigen Schaltung vereint werden. Dabei gibt es keine Überraschungen und die Schaltung ist lediglich eine Verschmelzung der vorangegangenen Schaltungen.

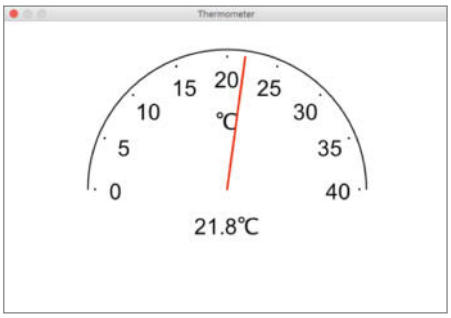
Die Software Thermometer sieht kompliziert aus, ist im Kern aber ganz einfach. Die ungewohnten Abschnitte kümmern sich hauptsächlich um die Darstellung des Thermometers auf dem Bildschirm.

Thermometer

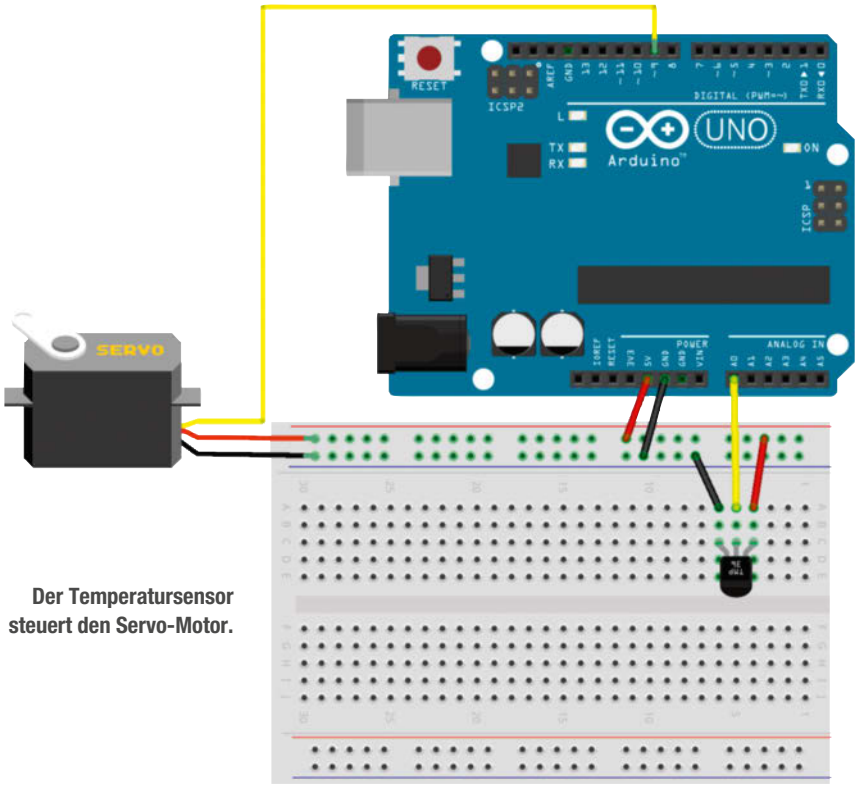


Eine Schablone für den Servo-Motor

```
 1 import processing.serial.*;
 2 import cc.arduino.*;
 3
 4 final byte SERVO_PIN = 9;
 5 final byte TEMP_PIN = 0;
 6 final float MIN_TEMP = 0;
 7 final float MAX_TEMP = 40;
 8 final int ZEICHEN_GROESSE = 32;
 9 final int BREITE = 640;
10 final int HOEHE = 480;
11 final int CX = BREITE / 2;
12 final int CY = HOEHE / 2;
13 final int RADIUS = 200;
14
15 Arduino arduino;
16
17 void settings() {
18   size(BREITE, HOEHE);
19 }
20
21 void setup() {
22   PFont zeichensatz = createFont("Arial", ZEICHEN_GROESSE, true);
23   textFont(zeichensatz, ZEICHEN_GROESSE);
24   String arduinoPort = "/dev/cu.usbmodem24311"; // Hier anpassen!
25   arduino = new Arduino(this, arduinoPort);
26   arduino.pinMode(SERVO_PIN, Arduino.SERVO);
27 }
28
29 void draw() {
30   background(255);
31   fill(0);
32   zeichneThermometer();
33   float temperatur = ermittleTemperatur();
34   zeichneTemperatur(temperatur);
35   steuereMotor(temperatur);
36   delay(1000);
37 }
38
39 void steuereMotor(float temperatur) {
40   float position = map(min(temperatur, MAX_TEMP), MIN_TEMP, MAX_TEMP, 0, 180);
41   arduino.servoWrite(SERVO_PIN, (int)position);
42 }
43
44 float ermittleTemperatur() {
45   float spannung = arduino.analogRead(TEMP_PIN) * 5000 / 1024.0;
46   float temperatur = (spannung - 500) / 10; // TMP36
47   // float temperatur = spannung / 10; // LM35 CZ
48   return temperatur;
49 }
50
51 void zeichneTemperatur(float temperatur) {
52   float t = map(min(temperatur, MAX_TEMP), MIN_TEMP, MAX_TEMP, -PI, 0);
53   stroke(255, 0, 0);
54   line(CX, CY, CX + cos(t) * RADIUS * 0.95, CY + sin(t) * RADIUS * 0.95);
55   textAlign(CENTER, CENTER);
56   text(nf(temperatur, 0, 1) + "?", BREITE / 2, HOEHE / 2 + 50);
57 }
58
59 void zeichneThermometer() {
60   noFill();
61   stroke(0);
62   strokeWeight(2);
63   arc(CX, CY, RADIUS * 2, RADIUS * 2, -PI, 0);
64   beginShape(POINTS);
65   int i = 0;
66   for (float a = -180; a <= 0; a += 22.5, i += 5) {
67     float winkel = radians(a);
68     float x = CX + cos(winkel) * RADIUS * 0.95;
69     float y = CY + sin(winkel) * RADIUS * 0.95;
70     strokeWeight(3);
71     vertex(x, y);
72     text(i, CX + cos(winkel) * RADIUS * 0.80, CY + sin(winkel) * RADIUS * 0.80);
73   }
74   endShape();
75   text("?", CX, CY - RADIUS / 2);
76 }
```



Processing zeigt die Temperatur per Zeiger an.



Der Temperatursensor steuert den Servo-Motor.

Erst einmal beginnt das Programm aber mit der Einbindung der benötigten Bibliotheken und der Definition einiger Konstanten und Variablen. Bei den Konstanten gibt es einige alte Bekannte, wie zum Beispiel SERVO_PIN, TEMP_PIN und ZEICHEN_GROESSE.

Es gibt aber auch ein paar Neuzugänge und so definieren MIN_TEMP und MAX_TEMP die minimale und maximale Temperatur, die vom Programm verarbeitet wird. BREITE und HOEHE legen die Abmessungen des Anwendungsfensters in Pixel fest und CX und CY bestimmen den Mittelpunkt des (Halb-)Kreises, der als Thermometer dient. Dieser Punkt liegt genau im Zentrum des Anwendungsfensters. RADIUS definiert den Radius des Thermometer-Halbkreises.

Fast schon obligatorisch ist das globale Objekt arduino, das zur Kommunikation mit dem Arduino dient.

Die settings-Funktion ist neu und sie ist auch eine Neuerung in der Programmiersprache Processing. Sie wird immer dann gebraucht, wenn die Größe des Anwendungsfensters in Form von Konstanten (hier: BREITE und HOEHE) festgelegt werden soll. Für den weiteren Programmverlauf ist das durchaus vorteilhaft und so kommt die Funktion settings zum Einsatz. Sie ruft lediglich die size-Funktion auf, um die Größe des Anwendungsfensters festzulegen.

In der setup-Funktion gibt es nichts Neues. Sie initialisiert den verwendeten Zeichensatz und den Arduino nebst Servo-Pin.

draw ist übersichtlich, weil sie viele Aufgaben an weitere Funktionen delegiert. Mit der background-Funktion löscht sie den Bildschirm und füllt ihn mit der Farbe Weiß (255, 255, 255). Mit der Funktion fill wird dann Schwarz (0, 0, 0) als Zeichenfarbe festgelegt.

Die Funktion zeichneThermometer zeichnet ein analoges Thermometer als einen Halbkreis, in dem die einzelnen Segmente eine Temperatur von 0 bis 40 Grad Celsius repräsentieren. Die tatsächliche Temperatur wird später als rote Linie angezeigt werden.

zeichneThermometer ruft zuerst noFill auf, damit der Halbkreis des Thermometers nicht mit der aktuellen Zeichenfarbe ausgefüllt wird. Die stroke-Funktion legt anschließend fest, dass alle folgenden Operationen die Zeichenfarbe Schwarz (0, 0, 0) verwenden. strokeWeight setzt die Breite des Pinsels auf zwei Pixel, das heißt, alle Linien werden im folgenden mit einer Strichdicke von zwei Pixeln gezeichnet.

Die Funktion arc (https://processing.org/reference/arc_.html) zeichnet einen beliebigen Bogen um einen vorgegebenen Mittelpunkt. Der Mittelpunkt ist in diesem Fall der Mittelpunkt des Anwendungsfensters (CX, CY) und der Radius des Bogens wird durch die Konstante RADIUS festgelegt. Der Rest der Funktion zeichnet die Punkte für die Temperaturwerte 5, 10, 15, 20, 25, 30 und 35. Dazu dient die Funktion beginShape und mit dem Argument POINTS legt sie fest, dass im Folgenden eine Reihe von Punkten gezeichnet werden soll.

Das Zeichnen der Punkte übernimmt die folgende Schleife. Sie zeichnet Punkte für die Winkel von –180 bis 0 Grad und wandelt diese erst einmal mit der radians-Funktion ins Bogenmaß um. Anschließend berechnet sie die Koordinaten der zu zeichnenden Punkte mithilfe der bekannten trigonometrischen Funktionen und übergibt sie an die Funktion vertex. Die text-Funktion gibt die zum jeweiligen Punkt gehörende Grad-Zahl aus.

Nachdem das Thermometer gezeichnet wurde, ermittelt draw die aktuelle Temperatur mit der Funktion ermittleTemperatur. Die verwendet die Funktion analogRead des Arduino, um die aktuellen Sensordaten zu messen, und wandelt sie dann gemäß der oben beschriebenen Formel in einen Temperaturwert um.

Dieser Wert wird als Text und in Form eines roten Zeigers mit der Funktion zeichneTemperatur in das Thermometer-Bild gezeichnet. Dazu berechnet die Funktion zuerst die Position der Temperatur innerhalb des Halbkreises. Dann setzt sie mit der stroke-Anweisung die Zeichenfarbe auf Rot. Die anschließende line-Anweisung zeichnet eine Linie vom Mittelpunkt des Thermometers zur aktuellen Temperatur. Der Faktor 0,95 sorgt dafür, dass die Linie nicht direkt auf dem Kreisbogen, sondern kurz davor endet.

Ganz am Ende wird die Temperatur mit der text-Funktion auch noch als Zahl ausgegeben.

Analog zum Zeiger auf dem Bildschirm wird mit der Funktion steuereMotor auch der angeschlossene Servo-Motor in die korrekte Position gebracht. Dazu wird die Temperatur mit der map-Funktion auf den Wertebereich des Servos (0 bis 180) abgebildet und anschließend der Servo-Motor mittels servoWrite positioniert.

Der Motor in der analogen Welt zeigt dieselbe Temperatur an, wie sein digitales Pendant auf dem Bildschirm. Besonders effektvoll wird das Beispiel, wenn ein Pfeil und eine Thermometer-Schablone am Motor befestigt werden.

Fazit

Sowohl Processing als auch Firmata sind nützliche Helfer bei der Umsetzung vieler Arduino-Projekte. Firmata automatisiert die lästige Implementierung eigener Protokolle und Processing macht dank ausgefeilter Multimedia-Eigenschaften jede Arduino-Anwendung schnell zu einem Augen- und Ohrenschmaus. *—dab*

IMPRESSUM

Redaktion

Make: Magazin
Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-300
Telefax: 05 11/53 52-417
Internet: www.make-magazin.de

Leserbriefe und Fragen zum Heft: info@make-magazin.de

Die E-Mail-Adressen der Redakteure haben die Form xx@make-magazin.de oder xxx@make-magazin.de. Setzen Sie statt „xx" oder „xxx" bitte das Redakteurs-Kürzel ein. Die Kürzel finden Sie am Ende der Artikel und hier im Impressum.

Chefredakteur: Daniel Bachfeld (dab)
(verantwortlich für den Textteil)

Stellv. Chefredakteur: Peter König (pek)

Redaktion: Helga Hansen (hch), Carsten Meyer (cm), Elke Schick (esk), Philip Steffan (phs)

Assistenz: Hans-Jürgen Berndt (hjb), Susanne Cölle (suc), Tim Rittmeier (tir), Sebastian Seck (sbs), Christopher Tränkmann (cht), Martin Triadan (mat)

DTP-Produktion: Wolfgang Otto (Ltg.), Martina Bruns, Martina Fredrich, Ines Gehre, Jürgen Gonnermann, Jörg Gottschalk, Birgit Graff, Angela Hilberg, Anja Kreft, Martin Kreft, Astrid Seifert, Edith Tötsches, Dieter Wahner, Brigitta Zurheiden

Art Direction: Martina Bruns (Art Director)

Layout-Konzept: Martina Bruns

Layout: Stefanie Rosemeyer, www.rosemeyer-design.de

Verlag

Maker Media GmbH
Postfach 61 04 07, 30604 Hannover
Karl-Wiechert-Allee 10, 30625 Hannover
Telefon: 05 11/53 52-0
Telefax: 05 11/53 52-129
Internet: www.make-magazin.de

Herausgeber: Christian Heise, Ansgar Heise

Geschäftsführer: Ansgar Heise, Dr. Alfons Schräder

Verlagsleiter: Dr. Alfons Schräder

Stellv. Verlagsleiter: Daniel Bachfeld

Anzeigenleitung: Michael Hanke (-167)
(verantwortlich für den Anzeigenteil),
www.heise.de/mediadaten/make

Leiter Vertrieb und Marketing: André Lux (-299)

Service Sonderdrucke: Julia Conrades (-156)

Druck: Goldschmidt GmbH, Josefstraße 35, 49809 Lingen

Vertrieb Einzelverkauf:
VU Verlagsunion KG
Am Klingenweg 10
65396 Walluf
Tel.: 0 61 23/62 01 32, Fax: 0 61 23/62 01 332
E-Mail: info@verlagsunion.de

Einzelpreis: 9,90 €; Österreich 10,90 €; Schweiz 17,50 CHF; Benelux, Italien, Spanien 10,90 €

Abonnement-Preise: Das Jahresabo (6 Ausgaben) kostet inkl. Versandkosten: Inland 52,80 €; Österreich 54,00 €; Schweiz 78,00 CHF; Europa 58,80 €; restl. Ausland 70,20 €. Schüler-/Studentenabo: Inland 32,40 €; Österreich 33,60 €; Schweiz CHF 56,40; Europa 38,40 €; restl. Ausland 49,80 €

Abo-Service:

Bestellungen, Adressänderungen, Lieferprobleme usw.:
Maker Media GmbH
Leserservice
Postfach 24 69
49014 Osnabrück
E-Mail: leserservice@heise.de
Telefon: 0541/80009-120
Telefax: 0541/80009-122

DIE AUTOREN DIESES HEFTS

Maik Schmidt arbeitet seit über 20 Jahren als Softwareentwickler für mittelständische und Großunternehmen. Außerdem schreibt er Buchkritiken und Artikel für internationale Zeitschriften und hat selbst einige Bücher verfasst. Unter anderem ist er Autor des Buchs „Arduino: A Quick-Start Guide, 2nd ed." (978-1-94122-224-9), das bei The Pragmatic Bookshelf erschienen ist und unter dem Titel „Arduino – Ein schneller Einstieg in die Microcontroller-Entwicklung" (ISBN 978-3-86490-126-3) vom dpunkt-Verlag ins Deutsche übersetzt wurde.

Daniel Bachfeld ist Chefredakteur beim Make-Magazin und Mit-Organisator der Maker Faire Hannover und Berlin. Mit ATmega-ICs bastelt er seit 1999. Er ist froh, dass ihm das Arduino-Konzept nun viel Arbeit beim Aufbau der Hardware und der Entwicklung der Software spart. Er freut sich über Projekteinreichungen und Artikelideen von Lesern, die auf Grundlage dieses Hefts entstanden sind: **dab@make-magazin.de**

Maker Faire®

DEUTSCHLAND • ÖSTERREICH • SCHWEIZ

FAMILIEN-FESTIVAL FÜR INSPIRATION, KREATIVITÄT & INNOVATION

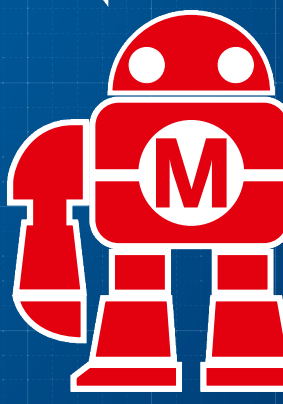
DER GROSSE MAKER-TREFFPUNKT.

3D-DRUCK
Wearables
Upcycling/Recycling
Arduino/Raspberry Pi & Co.
Hardware Hacking
ROBOTIK
Steampunk
Crafting/Handarbeit
Handwerk
Wissenschaft & Forschung
Elektronik
Musik
Internet of Things
QUADROCOPTER

FOR MAKERS ONLY:
Die Call for Makers sind eröffnet!

Unterstützen Sie als Sponsor die Maker-Szene, präsentieren Sie sich kreativen, technik- und wissenschaftlich begeisterten Menschen und zeigen Sie auf, welche Innovationen und welche Kreativität Sie mit Ihren Produkten verbindet. Um die Nachwuchsförderung zu unterstützen, wird auf den Maker Faires Hannover und Berlin 2016, erstmals ein zusätzlicher Bildungstag für Schüler und Lehrer stattfinden!

Weitere Informationen zu Sponsoring- und Austellungs-Möglichkeiten unter der Rubrik **Partner werden** auf **www.maker-faire.de.**



Dortmund	Wien	Hannover	Friedrichshafen	Berlin
12+13 März 2016 Maker Faire Ruhr	16 & 17 APRIL 2016 Maker Faire VIENNA	HANNOVER Maker Faire 27. – 29. MAI 2016	25. – 26. Juni 2016 Messe Friedrichshafen Maker Faire Bodensee	30. SEPTEMBER – 2. OKTOBER 2016 BERLIN Maker Faire
MÄRZ	APRIL	MAI	JUNI	OKTOBER

WWW.MAKER-FAIRE.DE