edition Make:

Arduino

Ein schneller Einstieg in die Microcontroller-Entwicklung

von
Maik Schmidt

2., akt. u. erw. Aufl.

dpunkt.verlag 2015

Verlag C.H. Beck im Internet:
www.beck.de
ISBN 978 3 86490 126 3

Zu Inhaltsverzeichnis

5 Die Außenwelt wahrnehmen

Anstatt wie bei normalen Computern mit der Maus oder der Tastatur zu kommunizieren, müssen Sie an Arduino besondere Sensoren anschließen, damit Änderungen in der Umgebung wahrgenommen werden können. Dabei können Sie Sensoren verwenden, die die aktuelle Temperatur, die Beschleunigung oder die Entfernung zum nächstgelegenen Objekt messen.

Sensoren spielen eine wichtige Rolle beim physischen Computereinsatz. Arduino macht die Verwendung unterschiedlicher Arten von Sensoren zum Kinderspiel. In diesem Kapitel verwenden wir sowohl digitale als auch analoge Sensoren, um einige Zustände der physischen Welt zu erfassen. Dazu benötigen wir nicht mehr als eine Hand voll Kabel und einige kleine Programme.

Dabei sehen wir uns zwei Arten von Sensoren genauer an: einen Ultraschallsensor, der Entfernungen misst, und einen Temperatursensor, der natürlich zur Temperaturbestimmung da ist.

Mit dem Ultraschallsensor konstruieren wir eine digitale Messlatte, um Abstände aus der Ferne messen zu können.

Die Messergebnisse von Ultraschallsensoren sind zwar ziemlich genau, aber Sie können diese Genauigkeit mit einigen einfachen Tricks sogar noch steigern. Interessanterweise hilft uns der Temperatursensor dabei. Am Ende dieses Kapitels werden Sie über eine ziemlich genaue digitale Messlatte verfügen. Außerdem erstellen wir eine hübsche grafische Anwendung, um von den Sensoren gemessenen Daten zu visualisieren.

Arduino erleichtert jedoch nicht nur die Verwendung von Sensoren, sondern fördert auch gutes Design für Ihre Schaltungen und Ihre Software. Beispielsweise verwenden wir letzten Endes zwar zwei Sensoren, doch sind sie vollständig unabhängig voneinander. Alle Programme, die wir in diesem Kapitel entwickeln, laufen ohne Änderungen in der endgültigen Schaltung.

5.1 Was Sie benötigen

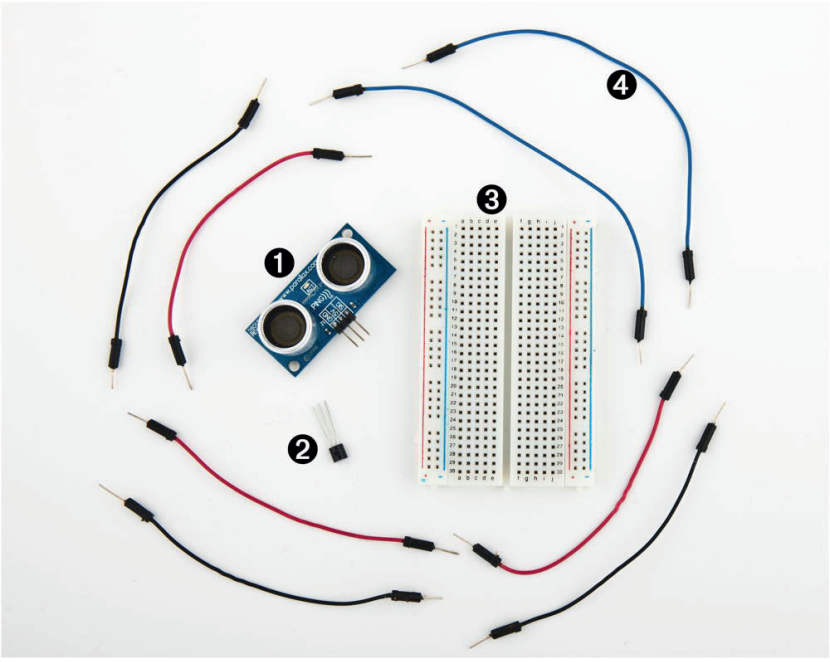

Abb. 5–1 *Die für dieses Kapitel benötigten Bauteile*

1. Parallax PING)))-Sensor
2. TMP36-Temperatursensor von Analog Devices[1]
3. ein Breadboard
4. einige Kabel
5. ein Arduino-Board wie das Uno, Duemilanove oder Diecimila
6. ein USB-Kabel, um Arduino mit Ihrem Computer zu verbinden

5.2 Entfernungen mit einem Ultraschallsensor messen

Eine automatische und fortlaufende Messung von Entfernungen ist in vielen Situationen praktisch. Denken Sie beispielsweise an einen Roboter, der automatisch versucht, sich auf seinem Weg zu orientieren, oder an einen automatischen Einbruchalarm, der eine Sirene auslöst oder die Polizei ruft, wenn jemand Ihrem Haus oder der Mona Lisa zu nahe kommt. All dies ist mit Arduino möglich. Bevor Sie

1 *In Europa ist der LM35 CZ-Sensor in der Regel einfacher zu bekommen, zum Beispiel bei Watterott. Er ist nicht vollkommen kompatibel zum TMP36, aber im Text wird auf die Unterschiede hingewiesen.*

den Einbruchalarm oder den Roboter konstruieren können, müssen Sie jedoch die Grundprinzipien kennen.

Es gibt viele verschiedene Arten von Sensoren für die Abstandsmessung, wobei Arduino mit den meisten gut umgehen kann. Manche Sensoren verwenden Ultraschall, andere wiederum Infrarot oder sogar Laser. Prinzipiell aber funktionieren alle Sensoren auf die gleiche Weise: Sie senden ein Signal aus, warten auf das Echo und messen, wie lange dieser Vorgang gedauert hat. Da wir wissen, wie hoch die Geschwindigkeiten von Schall und Licht in der Luft sind, können wir die gemessene Zeit in eine Entfernung umrechnen.

In unserem ersten Projekt bauen wir ein Gerät, das die Entfernung zum nächsten Objekt misst und am seriellen Port ausgibt. In diesem Projekt verwenden wir den Parallax PING)))-Ultraschallsensor[2] , da er leicht zu nutzen ist, mit einer ausgezeichneten Dokumentation geliefert wird und einen hervorragenden Funktionsumfang aufweist. Er kann Objekte in einem Bereich zwischen 2 cm und 3 m erkennen. Da wir ihn direkt an einem Breadboard anschließen, müssen wir nicht löten. Dies ist auch ein exzellentes Beispiel für einen Sensor, der Informationen über Impulse variabler Breite liefert (mehr darüber erfahren Sie einige Absätze weiter hinten). Mit dem PING)))-Sensor können wir ganz leicht ein Echolot oder einen Roboter bauen, der selbstständig einen Weg durch ein Labyrinth findet, ohne die Wände zu berühren.

Wie bereits vorher erwähnt, liefern Ultraschallsensoren nicht den Abstand zum nächsten Objekt, sondern geben die Laufzeit des Signals vom und zum Sensor zurück an. Der Ping))) bildet dabei keine Ausnahme (siehe Abbildung 5–2). Sein Aufbau ist recht komplex. Zum Glück liegt sein Innenleben abgeschirmt hinter drei einfachen Pins: Stromversorgung, Masse und Signal.

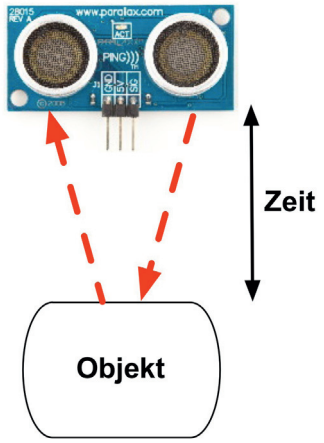

Abb. 5–2 *Grundlegendes Funktionsprinzip des PING)))-Sensors*

2 *http://www.parallax.com/StoreSearchResults/tabid/768/txtSearch/28015/List/0/SortField/4/ProductID/92/Default.aspx*

Dadurch wird der Anschluss des Sensors an Arduino einfach. Verbinden Sie als Erstes den Masse- und den 5-V-Stromanschluss von Arduino mit den entsprechenden PING)))-Pins. Schließen Sie dann den Sensor-Pin des PING))) an einen der digitalen I/O-Pins von Arduino an (wir verwenden hier ganz willkürlich Pin 7). Das Diagramm der Schaltung sehen Sie in Abbildung 5–3 und ein Foto in Abbildung 5–4.

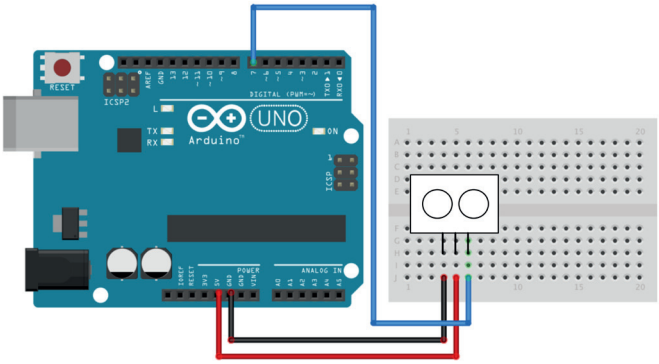

Abb. 5–3 *Grundlegende PING)))-Schaltung*

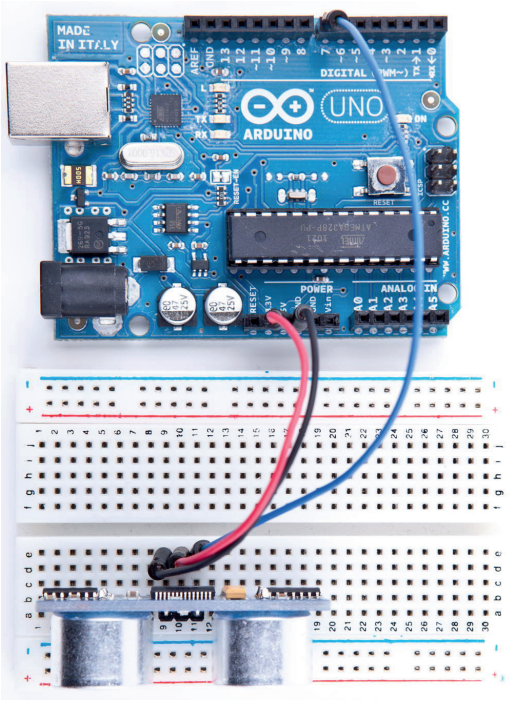

Abb. 5–4 *Foto der PING)))-Grundschaltung*

Um die Schaltung zum Leben zu erwecken, benötigen wir etwas Code, der mit dem PING)))-Sensor kommuniziert:

InputDevices/Ultrasonic/Simple/Simple.ino

```
Zeile 1   const unsigned int PING_SENSOR_IO_PIN = 7;
      -   const unsigned int BAUD_RATE = 9600;
      -
      -   void setup() {
      5     Serial.begin(BAUD_RATE);
      -   }
      -
      -   void loop() {
      -     pinMode(PING_SENSOR_IO_PIN, OUTPUT);
     10     digitalWrite(PING_SENSOR_IO_PIN, LOW);
      -     delayMicroseconds(2);
      -
      -     digitalWrite(PING_SENSOR_IO_PIN, HIGH);
      -     delayMicroseconds(5);
     15     digitalWrite(PING_SENSOR_IO_PIN, LOW);
      -
      -     pinMode(PING_SENSOR_IO_PIN, INPUT);
      -     const unsigned long duration = pulseIn(PING_SENSOR_IO_PIN, HIGH);
      -     if (duration == 0) {
     20       Serial.println("Warning: We did not get a pulse from sensor.");
      -     } else {
      -       Serial.print("Distance to nearest object: ");
      -       Serial.print(microseconds_to_cm(duration));
      -       Serial.println(" cm");
     25     }
      -
      -     delay(100);
      -   }
      -
     30   unsigned long microseconds_to_cm(const unsigned long microseconds) {
      -     return microseconds / 29 / 2;
      -   }
```

Als Erstes definieren wir eine Konstante für den I/O-Pin, an den der PING)))-Sensor angeschlossen ist.

Wenn Sie Ihren Sensor an einen anderen digitalen I/O-Pin anschließen möchten, müssen Sie die erste Zeile des Programms ändern. In der Methode *setup* legen wir die Baudrate des seriellen Ports auf 9600 fest, da wir einige Sensordaten auf dem seriellen Monitor sehen möchten.

Wirklich interessant wird es in *loop*, wo wir das PING)))-Protokoll dann tatsächlich implementieren. Laut Datenblatt[3] können wir die Sensoren mit Impulsen steuern. Auch die Ergebnisse werden als Impulse variabler Breite zurückgegeben.

In den Zeilen 9 bis 11 setzen wir den Signal-Pin des Sensors für 2 Mikrosekunden auf LOW, um ihn in einen sauberen Zustand zu bringen. Das sorgt für klare HIGH-Impulse, die wir in den nächsten Schritten brauchen (in der Elektronik sollten Sie immer auf leichte Schwankungen in der Stromversorgung vorbereitet sein).

3 *http://www.parallax.com/dl/docs/prod/acc/28015-PING-v1.5.pdf*

Schließlich müssen wir dem Sensor mitteilen, dass er etwas tun soll. In den Zeilen 13 bis 15 setzen wir den Signal-Pin des Sensors für 5 Mikrosekunden auf HIGH, um eine neue Messung zu beginnen.

Anschließend setzen wir den Pin wieder auf LOW, da der Sensor mit einem HIGH-Impulse variabler Länge auf demselben Pin antwortet.

Bei einem digitalen Pin haben Sie nur wenige Optionen, um Informationen zu übermitteln. Sie können den Pin auf HIGH oder LOW setzen und festlegen, wie lange er in einem bestimmten Zustand verbleibt. Für viele Zwecke ist das absolut ausreichend. Das gilt auch für diesen Fall. Der PING)))-Sensor sendet einen 40-kHz-Ton aus und setzt den Signal-Pin auf HIGH. Beim Empfang des Echos setzt er ihn auf LOW zurück. Der Signal-Pin bleibt also genau so lange im Status HIGH, wie der Schall benötigt, um zu einem Objekt und zurück zum Sensor zu gelangen. Grob gesagt, verwenden wir einen digitalen Pin zum Messen eines analogen Signals. Das Diagramm in Abbildung 5–5 zeigt die typische Aktivität auf einem digitalen Pin, der an einen PING)))-Sensor angeschlossen ist.

Abb. 5–5 *PING)))-Impulsdiagramm*

Wir könnten jetzt manuell messen, wie lange der Pin im Zustand HIGH bleibt, doch erledigt die Methode *pulseIn* diese Arbeit für uns. Daher verwenden wir sie in Zeile 18, nachdem wir den Signal-Pin wieder in den Eingabemodus versetzt haben. *pulseIn* verwendet drei Parameter:

- *pin*: die Nummer des Pins, von dem der Impuls gelesen wird
- *type*: der Typ des Impulses, der gelesen werden soll, also entweder HIGH oder LOW
- *timeout*: das Timeout in Mikrosekunden. Wenn innerhalb der Timeout-Periode kein Impuls erkannt werden kann, gibt *pulseIn* 0 zurück. Dieser Parameter ist optional und auf eine Sekunde voreingestellt.

Beachten Sie, dass bei diesem ganzen Vorgang nur ein Pin zur Kommunikation mit dem PING))) verwendet wird. Früher oder später werden Sie feststellen, dass I/O-Pins eine kostbare Ressource von Arduino sind. Daher ist es wirklich ein schöner Zug, dass der PING))) nur einen einzigen digitalen Pin belegt. Wenn Sie zwischen verschiedenen Teilen auswählen können, die dieselbe Aufgabe erledigen, versuchen Sie, so wenige Pins wie möglich dafür einzusetzen.

Wir müssen nur noch eines tun, nämlich die gemessene Zeitdauer in eine Länge umrechnen. Schall bewegt sich mit 343 m/s fort. Für die Strecke von einem Zentimeter benötigt er also 29,155 Mikrosekunden. Daher müssen wir die Dauer durch 29 und dann durch 2 teilen, da der Schall die Strecke ja zweimal zurücklegt, nämlich einmal zum Objekt und dann wieder zurück zum PING)))-Sensor. Die Methode *microseconds_to_cm* führt die Berechnung durch.

Laut Spezifikation des PING)))-Sensors müssen Sie zwischen zwei Messungen mindestens 200 Mikrosekunden warten. Für Hochgeschwindigkeitsmessungen könnten wir die Länge einer Pause genauer berechnen, indem wir die Zeit messen, die zum Ausführen des Codes benötigt wird. In unserem Fall ist das sinnlos, da alle zwischen zwei Messungen ausgeführten Anweisungen in der Methode *loop* viel mehr Zeit als 200 Mikrosekunden benötigen. Schon die Ausgabe der Daten an die serielle Verbindung ist mit einem hohen Zeitaufwand verbunden. Trotzdem haben wir eine kleine Verzögerung von 100 Mikrosekunden eingebaut, um die Ausgabe ein wenig zu verlangsamen.

Vielleicht fragen Sie sich, warum wir das Schlüsselwort *const* so häufig verwenden. Die Arduino-Sprache beruht auf C/C++, und in diesen Sprachen gilt es als guter Stil, konstante Werte mit *const* zu deklarieren (siehe *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* [Mey97]). Die Verwendung von *const* macht Ihr Programm nicht nur präziser und verhindert das Auftreten logischer Fehler möglichst frühzeitig, sie hilft dem Compiler auch dabei, die Programmgröße zu verringern.

Auch wenn die meisten Arduino-Programm relativ klein sind, so ist die Softwareentwicklung für Arduino dennoch Softwareentwicklung und sollte daher nach bewährten Praktiken erfolgen. Wenn Sie in Ihrem Programm also einen konstanten Wert definieren, sollten Sie ihn auch als solchen deklarieren (mit *const* und nicht mit *#define*).

Das gilt auch für andere Programmiersprachen, weshalb wir in Java-Programmen auch *final* verwenden.

Als Nächstes werden wir ein wenig mit dem Sensor herumspielen, um uns mit seinen Stärken und Schwächen vertraut zu machen. Kompilieren Sie das Programm, laden Sie es zu Ihrem Arduino-Board hoch und öffnen Sie den seriellen Monitor (denken Sie daran, die Baudrate auf 9600 zu setzen). Jetzt sollten Sie in etwa folgende Anzeige sehen:

```
Abstand zum nächsten Objekt: 42 cm
Abstand zum nächsten Objekt: 33 cm
Abstand zum nächsten Objekt: 27 cm
Abstand zum nächsten Objekt: 27 cm
Abstand zum nächsten Objekt: 29 cm
Abstand zum nächsten Objekt: 36 cm
```

Neben der Ausgabe im Terminal können Sie auch erkennen, dass die LED des PING)))-Sensors eingeschaltet wird, wenn der Sensor mit einer neuen Messung beginnt.

Testen Sie die Fähigkeiten des Sensors, indem Sie versuchen, große oder sehr kleine Objekte zu erkennen. Versuchen Sie, Objekte aus verschiedenen Winkeln zu erkennen, sowie Objekte, die sich unterhalb oder oberhalb des Sensors befinden. Sie sollten auch einige Experimente mit Objekten vornehmen, die keine glatte Oberfläche aufweisen. Versuchen Sie zum Beispiel, Plüschtiere anzupeilen. Dabei werden Sie feststellen, dass sie nicht so gut erkannt werden wie feste Objekte. (Das ist möglicherweise der Grund, warum Fledermäuse keine Bären jagen: Sie können sie nicht sehen.)

Mit nur drei Kabeln und einigen wenigen Codezeilen haben wir die erste Version einer digitalen Messlatte erstellt. Zurzeit gibt sie nur Abstände in ganzzahligen Zentimeterwerten aus, aber wir werden die Genauigkeit in den nächsten Abschnitten gewaltig steigern, indem wir unsere Software umschreiben und mehr Hardware hinzufügen.

5.3 Die Genauigkeit mit Fließkommazahlen erhöhen

Laut Spezifikation kann der PING)))-Sensor den Abstand zu Objekten genau messen, die zwischen 2 cm und 3 m von ihm entfernt sind. (Der Grund dafür ist übrigens die Dauer des generierten Impulses. Die Mindestdauer beträgt 115 Mikrosekunden, die Maximallänge 18,5 Millisekunden.) Bei unserer jetzigen Vorgehensweise können wir diese Genauigkeit nicht voll ausschöpfen, da wir alle Berechnungen mit ganzzahligen Werten vornehmen. Daher können wir Entfernungen nur mit einer Genauigkeit von einem Zentimeter messen. Um in den Millimeterbereich vorzustoßen, müssen wir Fließkommazahlen verwenden.

Normalerweise ist es sinnvoll, Operationen mit Ganzzahlen vorzunehmen, da die Kapazität des Arbeitsspeichers und der CPU von Arduino im Vergleich mit regulären Computern stark eingeschränkt ist und Berechnungen mit Fließkommazahlen häufig viel Aufwand erfordern.

Manchmal ist es jedoch sinnvoll, sich den Luxus sehr genauer Fließkommazahlen zu gönnen. Mit Arduino ist das auch gut möglich. Wir setzen sie jetzt ein, um unser Projekt zu verbessern.

InputDevices/Ultrasonic/Float/Float.ino

```
Zeile 1   const unsigned int PING_SENSOR_IO_PIN = 7;
      -   const unsigned int BAUD_RATE = 9600;
      -   const float MICROSECONDS_PER_CM = 29.155;
      -   const float MOUNTING_GAP = 0.2;
      5   const float SENSOR_OFFSET = MOUNTING_GAP * MICROSECONDS_PER_CM * 2;
      -
      -   void setup() {
      -     Serial.begin(BAUD_RATE);
      -   }
     10   void loop() {
      -     const unsigned long duration = measure_distance();
      -     if (duration == 0)
      -       Serial.println("Warning: We did not get a pulse from sensor.");
      -     else
     15       output_distance(duration);
      -   }
```

```
-
-   const float microseconds_to_cm(const unsigned long microseconds) {
-     const float net_distance = max(0, microseconds - SENSOR_OFFSET);
20    return net_distance / MICROSECONDS_PER_CM / 2;
-   }
-
-   const unsigned long measure_distance() {
-     pinMode(PING_SENSOR_IO_PIN, OUTPUT);
25    digitalWrite(PING_SENSOR_IO_PIN, LOW);
-     delayMicroseconds(2);
-     digitalWrite(PING_SENSOR_IO_PIN, HIGH);
-     delayMicroseconds(5);
-     digitalWrite(PING_SENSOR_IO_PIN, LOW);
30    pinMode(PING_SENSOR_IO_PIN, INPUT);
-     return pulseIn(PING_SENSOR_IO_PIN, HIGH);
-   }
-
-   void output_distance(const unsigned long duration) {
35    Serial.print("Distance to nearest object: ");
-     Serial.print(microseconds_to_cm(duration));
-     Serial.println(" cm");
-   }
```

Dieses Programm unterscheidet sich nicht sehr stark von der ersten Version. Zunächst haben wir für die Zeit in Mikrosekunden, die der Schall für die Strecke von 1 cm benötigt, den genaueren Wert 29,155 genommen. Außerdem wird bei der Entfernungsberechnung jetzt eine mögliche Lücke zwischen Sensor und Gehäuse berücksichtigt. Wenn Sie den Sensor am Breadboard anschließen, bleibt gewöhnlich eine kleine Lücke zwischen ihm und der Kante des Breadboards. Diese Lücke wird in Zeile 5 angegeben und später bei der Distanzberechnung verwendet. Die Lücke wird in Zentimetern gemessen und mit 2 multipliziert, da der Schall hin- und zurückläuft.

Die Methode *loop* sieht jetzt sauberer aus, da die Hauptfunktionalität des Programms in getrennte Funktionen verschoben wurde. Die gesamte Logik zur Sensorsteuerung befindet sich jetzt in der Methode *measure_distance*, während sich *output_distance* um die Ausgabe der Werte an den seriellen Port kümmert. Die größten Änderungen finden sich in der Funktion *microseconds_to_cm*. Sie gibt jetzt einen Fließkommawert zurück und zieht die Lücke zwischen Sensor und Breadboardkante von dem ermittelten Abstand ab. Damit wir keine negativen Werte erhalten, verwenden wir die Funktion *max*.

Kompilieren Sie das Programm und laden Sie es hoch. Jetzt sollten Sie eine Anzeige wie die folgende im Fenster des seriellen Monitors sehen:

```
Abstand zum nächsten Objekt: 17,26 cm
Abstand zum nächsten Objekt: 17,93 cm
Abstand zum nächsten Objekt: 17,79 cm
Abstand zum nächsten Objekt: 18,17 cm
Abstand zum nächsten Objekt: 18,65 cm
Abstand zum nächsten Objekt: 18,85 cm
```

Das sieht nicht nur genauer aus als unsere vorherige Version, sondern ist es auch.

Wenn Sie schon in anderen Programmiersprachen mit Fließkommazahlen gearbeitet haben, fragen Sie sich vielleicht, warum Arduino sie automatisch auf zwei Nachkommastellen rundet. Das Geheimnis liegt in der Methode *print* der Klasse *Serial*. In aktuellen Versionen der Arduino-Plattform funktioniert sie bei allen möglichen Datentypen, und wenn sie eine *float*-Variable empfängt, rundet sie sie vor der Ausgabe auf zwei Stellen. Die Anzahl der Nachkommastellen können Sie festlegen.

Beispielsweise führt *Serial.println(3.141592, 4);* zur Ausgabe *3.1416*.

Nur die Ausgabe ist davon betroffen, intern handelt es sich immer noch um eine Fließkommavariable. Übrigens sind zurzeit bei den meisten Arduinos die *float*- und *double*-Variablen gleich. Nur auf dem Arduino Due ist *double* genauer als *float*.

Was kostet es uns nun, *float*-Variablen zu verwenden? Der Speicherverbrauch dieser Variablen beträgt 4 Byte. Das heißt, dass sie genauso viel Arbeitsspeicher verbrauchen wie *long*-Variablen. Andererseits sind Fließkommaberechnungen ziemlich teuer, weshalb Sie in zeitkritischen Teilen Ihrer Software darauf verzichten sollten. Die größten Kosten verursachen die zusätzlichen Bibliotheksfunktionen, die zur Unterstützung von *float*-Zahlen mit Ihrem Programm verlinkt werden müssen. *Serial.print(3.14)* mag harmlos aussehen, erhöht die Programmgröße aber erheblich.

Kommentieren Sie Zeile 36 aus und kompilieren Sie das Programm erneut, um sich diese Auswirkung anzusehen. Es funktioniert jetzt nicht mehr richtig, aber wir erkennen den Einfluss auf die Programmgröße. Bei meiner derzeitigen Installation umfasst das Programm 3.002 Byte ohne *float*-Unterstützung für *Serial.print* und anderenfalls 5.070 Byte. Das macht einen Unterschied von 2.068 Byte aus!

In einigen Fällen bekommen Sie das beste aus beiden Welten: *Float*-Unterstützung ohne dafür mit Speicher bezahlen zu müssen. Sie können eine Menge Platz sparen, indem Sie die *float*-Werte in Ganzzahlen umwandeln, bevor Sie sie über eine serielle Verbindung senden. Um die Daten mit der Genauigkeit von zwei Nachkommastellen zu übertragen, multiplizieren Sie sie mit 100. Denken Sie daran, sie auf der Empfängerseite wieder durch 100 zu dividieren. Diesen Trick (einschließlich der Rundung) werden wir später anwenden.

5.4 Die Genauigkeit mithilfe eines Temperatursensors erhöhen

Die Unterstützung für Fließkommazahlen ist sicherlich eine Verbesserung. Hauptsächlich aber erhöht sich dadurch die Genauigkeit der Programmausgabe. Einen ähnlichen Effekt könnten wir mit einem mathematischen Trick für ganze Zahlen erzielen. Jetzt aber fügen wir eine noch wirkungsvollere Verbesserung hinzu, die sich nicht mit Software nachstellen lässt, nämlich einen Temperatursensor.

Als ich schrieb, dass sich Schall in Luft mit 343 m/s fortbewegt, war ich nicht ganz genau, denn die Schallgeschwindigkeit ist nicht konstant, sondern hängt unter anderem von der Lufttemperatur ab. Wenn Sie die Temperatur nicht berücksichtigen, kann der Fehler bis auf signifikante 12 Prozent ansteigen. Die tatsächliche Schallgeschwindigkeit C berechnen wir mit folgender einfacher Formel:

```
C = 331,5 + (0,6 * T)
```

Um diese Formel verwenden zu können, müssen wir die aktuelle Temperatur T in Celsius bestimmen. Dazu verwenden wir die Spannung am Ausgang des Temperatursensors TMP36 von Analog Devices.[4] Er ist günstig und einfach in der Anwendung.

Um den TMP36 an Arduino anzuschließen, verbinden Sie Masse und Stromquelle von Arduino mit den entsprechenden Pins des Sensors. Verbinden Sie den Signal-Pin des Sensors dann mit A0, also dem analogen Pin Nr. 0 (siehe Abbildung 5–6).

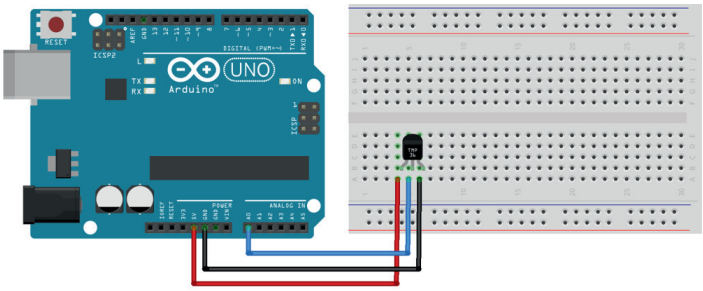

Abb. 5–6 *Anschließen des Temperatursensors an Arduino*

Wie Sie vielleicht schon aufgrund des Herstellernamens erraten haben, handelt es sich beim TMP36 um ein analoges Gerät: Es ändert die Spannung an seinem Signal-Pin entsprechend der aktuellen Temperatur. Je höher die Temperatur, umso höher die Spannung. Für uns ist das eine hervorragende Gelegenheit, den Umgang mit den analogen I/O-Pins von Arduino zu erlernen.

4 *http://tinyurl.com/msard-analog*

Dazu schreiben wir etwas Code zur Benutzung des Sensors:

InputDevices/Temperature/SensorTest/SensorTest.ino

```
Line 1  const unsigned int TEMP_SENSOR_PIN = A0;
    -   const float SUPPLY_VOLTAGE = 5.0;
    -   const unsigned int BAUD_RATE = 9600;
    -
    5   void setup() {
    -     Serial.begin(BAUD_RATE);
    -   }
    -
    -   void loop() {
   10     const float tempC = get_temperature();
    -     const float tempF = (tempC * 9.0 / 5.0) + 32.0;
    -     Serial.print(tempC);
    -     Serial.print(" C, ");
    -     Serial.print(tempF);
   15     Serial.println(" F");
    -     delay(1000);
    -   }
    -
    -   const float get_temperature() {
   20     const int sensor_voltage = analogRead(TEMP_SENSOR_PIN);
    -     const float voltage = sensor_voltage * SUPPLY_VOLTAGE / 1024;
    -     return (voltage * 1000 - 500) / 10;
    -   }
```

In den ersten beiden Zeilen definieren wir Konstanten für den analogen Pin, an den der Sensor angeschlossen ist, und für die Spannung der Stromversorgung von Arduino. Dann folgen eine ganz normale *setup*-Methode und eine *loop*-Methode, die jede Sekunde die aktuelle Temperatur ausgibt. Die gesamte Sensorlogik ist in der Methode *get_temperature* gekapselt. Sie gibt die Temperatur in Grad Celsius zurück, die wir dann zusätzlich in Fahrenheit konvertieren.

Für den PING)))-Sensor brauchten wir nur einen digitalen Pin, der entweder HIGH oder LOW sein konnte. Analoge Pins sind anders: Sie stellen einen Spannungsbereich von 0 V bis zur Spannung der jeweiligen Stromquelle dar (gewöhnlich 5 V). Die analogen Pins von Arduino können wir mit der Methode *analogRead* auslesen, die einen Wert zwischen 0 und 1023 zurückgibt, da analoge Pins eine Auflösung von zehn Bits haben ($1024 = 2^{10}$). Diese Methode haben wir in Zeile 20 verwendet, um die Spannung abzulesen, die zurzeit vom TMP36 bereitgestellt wird. Wenn Sie statt des TMP36 den LM35 CZ-Sensor verwenden, müssen Sie Zeile 22 durch »return voltage * 100;« ersetzen. Allerdings können Sie in diesem Fall keine negativen Temperaturen messen.

Sensordaten kodieren

Das Kodieren von Sensordaten ist ein Problem, das in Arduino-Projekten häufig gelöst werden muss, da all die schönen Daten, die wir erheben, meistens von Anwendungen auf regulären Computern interpretiert werden müssen.

Bei der Definition eines Datenformats müssen Sie mehrere Dinge berücksichtigen. Beispielsweise sollte das Format nicht verschwenderisch mit dem kostbaren Arbeitsspeicher von Arduino umgehen. In unserem Fall hätten wir zur Kodierung der Sensordaten auch XML verwenden können:

```
<sensor-data>
   <temperature>30.05</temperature>
   <distance>51.19</distance>
</sensor-data>
```

Das ist jedoch offensichtlich keine gute Wahl, da wir hierbei für die Struktur des Dateiformats ein Vielfaches des Arbeitsspeichers verschwenden, der für die eigentlichen Daten erforderlich ist. Außerdem muss die Empfängeranwendung einen XML-Parser einsetzen, um die Daten zu verstehen.

Das andere Extrem sollten Sie aber auch vermeiden. Das heißt, verwenden Sie Binärformate nur dann, wenn es absolut notwendig ist oder wenn die Empfängeranwendung ohnehin Binärdaten erwartet.

Alles in allem sind die einfachsten Datenformate wie zeichengetrennte Werte (CSV) häufig auch die beste Wahl.

Es gibt jedoch noch ein Problem: Wir müssen den von *analogRead* zurückgegebenen Wert in einen echten Spannungswert zurückwandeln. Dazu müssen wir die Spannung der Arduino-Stromversorgung kennen. Gewöhnlich beträgt sie 5 V, aber es gibt Arduino-Modelle (z.B. Arduino Pro) mit lediglich 3,3 V. Die Konstante *SUPPLY_VOLTAGE* müssen Sie daher entsprechend anpassen.

Wenn wir die Spannung der Stromquelle kennen, können wir die Ausgabe des analogen Pins in einen Spannungswert umwandeln, indem wir sie durch 1024 teilen und anschließend mit der Quellspannung multiplizieren. Das geschieht in Zeile 21.

Jetzt müssen wir den Spannungswert, den der Sensor liefert, in Celsiusgrade umrechnen.

Im Datenblatt des Sensors finden wir folgende Formel:

```
T = ((Sensorausgabe in mV) - 500) / 10
```

500 mV müssen abgezogen werden, da der Sensor stets eine positive Spannung abgibt. Dadurch sind wir auch in der Lage, negative Temperaturen wiederzugeben. Die Sensorauflösung beträgt 10 mV, weshalb wir eine Division durch 10 durchfüh-

ren müssen. Ein Spannungswert von 750 mV entspricht beispielsweise einer Temperatur von (750 - 500) / 10 = 25 °C. Die Implementierung erfolgt in Zeile 22.

Kompilieren Sie das Programm und laden Sie es auf Arduino hoch. Anschließend sollten Sie im seriellen Monitor etwa Folgendes sehen:

```
20.80 C, 69.44 F
20.80 C, 69.44 F
20,31 C, 68,56 F
20.80 C, 69.44 F
20.80 C, 69.44 F
```

Wie Sie erkennen können, benötigt der Sensor ein bisschen Zeit zur Kalibrierung, aber die Ergebnisse werden ziemlich schnell stabil. Übrigens müssen Sie stets eine kurze Verzögerung zwischen zwei Aufrufen von *analogRead* einbauen, da das interne analoge System von Arduino zwischen zwei Lesevorgängen etwas Zeit benötigt (0,0001 s auf dem Uno). Wir haben hier eine Verzögerung von einer ganzen Sekunde verwendet, um die Ausgabe leichter lesbar zu machen. Außerdem erwarten wir keine schnellen Temperaturänderungen. Anderenfalls würde eine Verzögerung von einer Millisekunde ausreichen.

Jetzt haben wir zwei getrennte Schaltungen: eine für die Messung von Entfernungen und eine für die Messung von Temperaturen. In Abbildung 5–7 und im Foto aus Abbildung 5–8 sind sie zu einer einzigen Schaltung kombiniert.

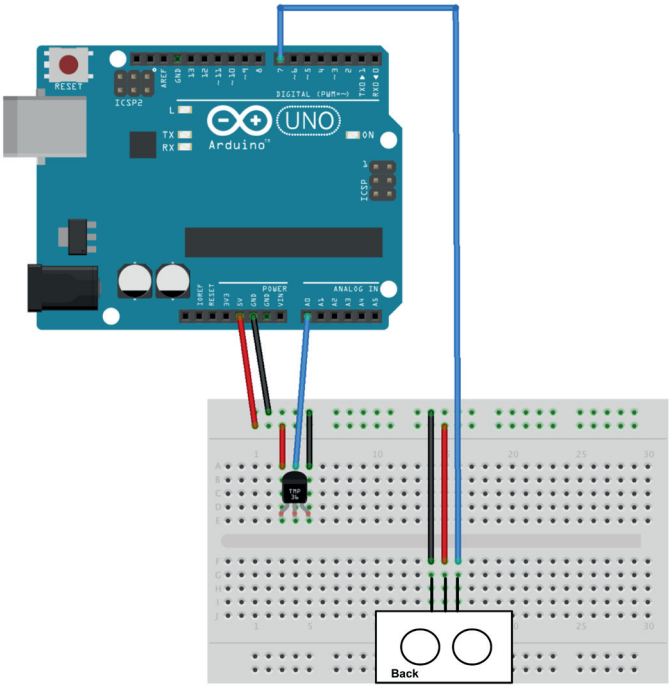

Abb. 5–7 *Zusammenspiel des TMP36- und des PING)))-Sensors*

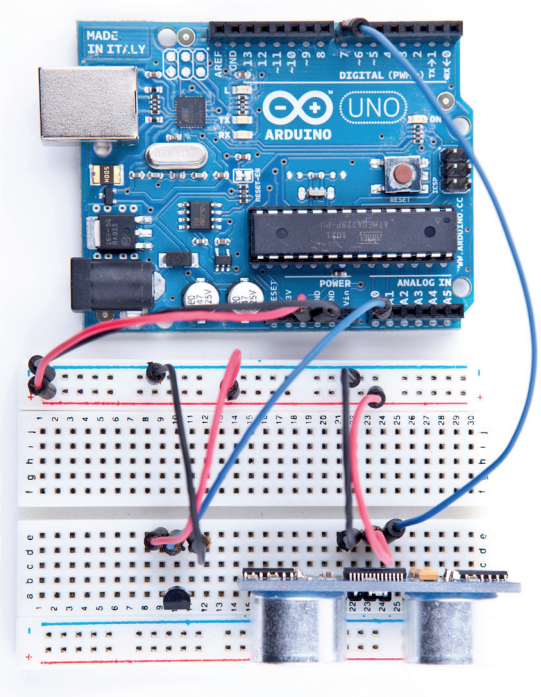

Abb. 5–8 *Foto der endgültigen Schaltung*

Erwecken Sie diese Schaltung mit dem folgenden Programm zum Leben:

Listing InputDevices/Ultrasonic/PreciseSensor/PreciseSensor.ino

```
Zeile 1   const unsigned int TEMP_SENSOR_PIN = A0;
      -   const float SUPPLY_VOLTAGE = 5.0;
      -   const unsigned int PING_SENSOR_IO_PIN = 7;
      -   const float SENSOR_GAP = 0.2;
      5   const unsigned int BAUD_RATE = 9600;
      -   float current_temperature = 0.0;
      -   unsigned long last_measurement = millis();
      -
      -   void setup() {
     10     Serial.begin(BAUD_RATE);
      -   }
      -
      -   void loop() {
      -     unsigned long current_millis = millis();
     15     if (abs(current_millis - last_measurement) >= 1000) {
      -       current_temperature = get_temperature();
      -       last_measurement = current_millis;
      -     }
      -     Serial.print(scaled_value(current_temperature));
     20     Serial.print(",");
      -     const unsigned long duration = measure_distance();
      -     Serial.println(scaled_value(microseconds_to_cm(duration)));
      -   }
      -
```

```
25  long scaled_value(const float value) {
    -   float round_offset = value < 0 ? -0.5 : 0.5;
    -   return (long)(value * 100 + round_offset);
    -  }
    -
30  const float get_temperature() {
    -   const int sensor_voltage = analogRead(TEMP_SENSOR_PIN);
    -   const float voltage = sensor_voltage * SUPPLY_VOLTAGE / 1024;
    -   return (voltage * 1000 - 500) / 10;
    -  }
35
    -  const float microseconds_per_cm() {
    -   return 1 / ((331.5 + (0.6 * current_temperature)) / 10000);
    -  }
    -
40  const float sensor_offset() {
    -   return SENSOR_GAP * microseconds_per_cm() * 2;
    -  }
    -
    -  const float microseconds_to_cm(const unsigned long microseconds) {
45    const float net_distance = max(0, microseconds - sensor_offset());
    -   return net_distance / microseconds_per_cm() / 2;
    -  }
    -
    -  const unsigned long measure_distance() {
50    pinMode(PING_SENSOR_IO_PIN, OUTPUT);
    -   digitalWrite(PING_SENSOR_IO_PIN, LOW);
    -   delayMicroseconds(2);
    -
    -   digitalWrite(PING_SENSOR_IO_PIN, HIGH);
55    delayMicroseconds(5);
    -   digitalWrite(PING_SENSOR_IO_PIN, LOW);
    -   pinMode(PING_SENSOR_IO_PIN, INPUT);
    -   return pulseIn(PING_SENSOR_IO_PIN, HIGH);
    -  }
```

Der Code ist beinahe eine perfekte Verschmelzung der Programme, die wir benutzt haben, um den PING)))- und den TMP36-Sensor zum Laufen zu bringen. Es haben sich nur einige wenige Dinge geändert:

- Die Konstante *MICROSECONDS_PER_CM* wurde durch die Funktion *microseconds_per_cm* ersetzt, die dynamisch auf der Grundlage der aktuellen Temperatur bestimmt, wie viele Mikrosekunden der Schall für die Strecke von 1 cm benötigt.
- Da die aktuelle Temperatur sich erwartungsgemäß nicht häufig oder schnell ändert, messen wir sie nicht mehr ständig, sondern nur einmal pro Sekunde. In Zeile 7 verwenden wir *millis*, um zu bestimmen, wie viele Millisekunden seit dem Start von Arduino vergangen sind. In den Zeilen 14 bis 18 prüfen wir, ob seit der letzten Messung mehr als eine Sekunde verstrichen ist. Wenn ja, messen wir die aktuelle Temperatur erneut.

- Wir übertragen die Sensordaten nicht mehr als Fließkommazahlen an den seriellen Port, sondern verwenden stattdessen skalierte Integerwerte. Das geschieht mithilfe der Funktion *scaled_value*, die einen *float*-Wert auf zwei Stellen rundet und durch die Multiplikation mit 100 in einen *long*-Wert umwandelt. Auf der Empfängerseite müssen wir die Zahl wieder durch 100 teilen.

Wenn Sie das Programm auf Ihren Arduino hochladen und mit den Händen vor dem Sensor herumspielen, sehen Sie eine Ausgabe wie die folgende:

```
2129,1016
2129,1027
2129,1071
2129,1063
2129,1063
2129,1063
```

Diese Ausgabe ist eine kommagetrennte Liste von Werten, wobei der erste Wert für die aktuelle Temperatur in Grad Celsius und der zweite für die Entfernung zum nächsten Objekt in Zentimetern steht. Beide Werte müssen noch durch 100 dividiert werden, um die tatsächlichen Sensordaten zu erhalten.

Unser kleines Projekt verfügt jetzt über zwei Sensoren. Einer ist mit einem digitalen Pin verbunden, der andere verwendet einen analogen Pin. Im nächsten Abschnitt erfahren Sie, wie Sie die Sensordaten an einen PC übertragen und dazu nutzen, Anwendungen zu erstellen, die auf aktuellen Eigenschaften der Umgebung beruhen.

Klimaschutz durch Sonarsensoren

Forscher von Northwestern und der University of Michigan haben ein Sonarsystem entwickelt, das nur anhand des Mikrofons und der Lautsprecher eines Computers herauszufinden, ob der Rechner gerade verwendet wird oder nicht.[a] Wenn nicht, schaltet der Computer automatisch den Bildschirm ab, was zum Umweltschutz beiträgt.

Anstelle von Mikrophon und Lautsprechern können wir auch einen PING)))-Sensor verwenden. Mit dem Wissen, das Sie sich in diesem Kapitel angeeignet haben, können Sie ein solches System mit Leichtigkeit selbst bauen. Versuchen Sie es!

a *http://blog.makezine.com/2009/10/15/using-sonar-to-save-power/*

5.5 Bauen Sie Ihr eigenes Armaturenbrett

Statt die Werte der digitalen und analogen Sensoren einfach nur über den seriellen Port auszugeben, simulieren wir in diesem Abschnitt einen kleinen Ausschnitt des Armaturenbretts eines modernen Autos. Viele Autos zeigen die aktuelle Temperatur an und besitzen ein Abstandssystem, das Sie beim Einparken warnt, wenn Sie einem Objekt zu nahe kommen.

In meinem Auto besteht das System z.B. aus einer Reihe oranger und roter LEDs. Befindet sich nichts neben dem Auto, sind alle LEDs ausgeschaltet. Sobald der Abstand zwischen Auto und einem potenziellen Hindernis jedoch zu klein wird, leuchtet die erste orange LED auf. Je kürzer der Abstand, desto mehr LEDs leuchten auf. Erreicht der Abstand einen kritischen Wert, leuchten alle LEDs und das Auto gibt einen nervtötenden Piepton aus.

In Abbildung 5–9 sehen Sie die Anwendung, die wir erstellen werden. Sie zeigt die aktuelle Temperatur an und Sie können an der ersten rot leuchtenden LED erkennen, dass sich bereits etwas in direkter Nähe des Abstandssensors befindet.

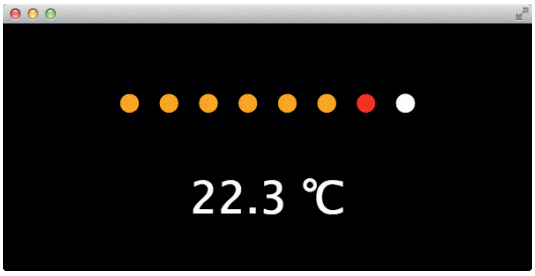

Abb. 5–9 *Unser Anzeigeinstrument in Aktion*

Wir implementieren die Anwendung als Chrome-App (jetzt wäre es sinnvoll, Anhang 4, »Arduino mit einem Browser steuern« auf Seite 277 zu lesen, wenn Sie das nicht bereits getan haben). Die Anwendungsdatei *manifest.json* enthält keine Überraschungen:

InputDevices/Dashboard/manifest.json

```
{
  "manifest_version": 2,
  "name": "Dashboard Demo",
  "version": "1",
  "permissions": [ "serial" ],
  "app": {
    "background": {
      "scripts": ["background.js"]
    }
  },
  "minimum_chrome_version": "33"
}
```

Sie definiert alle benötigten Metainformationen und deklariert, dass die Chrome-App auf den seriellen Port zugreifen muss. Die Datei *background.js* ist ebenfalls nicht besonders spannend:

InputDevices/Dashboard/background.js

```
chrome.app.runtime.onLaunched.addListener(function() {
  chrome.app.window.create('main.html', {
    id: 'main',
    bounds: { width: 600, height: 300 }
  });
});
```

Sie öffnet ein neues Fenster und zeigt die Datei *main.html* an:

InputDevices/Dashboard/main.html

```
Zeile 1  <!DOCTYPE html>
       - <html lang="en">
       -   <head>
       -     <meta charset="utf-8"/>
     5 -     <link rel="stylesheet" type="text/css" href="css/dashboard.css"/>
       -     <title>Dashboard Demo</title>
       -   </head>
       -   <body>
       -     <div id="dashboard">
    10 -       <div id="distance-display">
       -         <p>
       -           <span id="d1">&#x25cf;</span>
       -           <span id="d2">&#x25cf;</span>
       -           <span id="d3">&#x25cf;</span>
    15 -           <span id="d4">&#x25cf;</span>
       -           <span id="d5">&#x25cf;</span>
       -           <span id="d6">&#x25cf;</span>
       -           <span id="d7">&#x25cf;</span>
       -           <span id="d8">&#x25cf;</span>
    20 -         </p>
       -       </div>
       -       <div id="temperature-display">
       -         <p><span id="temperature"></span> &#x2103;</p>
       -       </div>
    25 -     </div>
       -     <script src="js/serial_device.js"></script>
       -     <script src="js/dashboard.js"></script>
       -   </body>
       - </html>
```

Um die Benutzerschnittstelle des Cockpits zu erstellen, benötigen wir ein wenig einfaches HTML. Zum Beispiel definieren wir die Anzeige der Einparkhilfe in den Zeilen 12 bis 19.

Die einzelnen LEDs stellen wir über ein <*span*>-Element mit dem Unicode-Zeichen (&#x25cf) dar (ein ausgefüllter Kreis). Jedes <*span*>-Element erhält eine eigene ID, so dass wir die einzelnen LEDs später ansprechen können.

Die Temperaturanzeige ist sogar noch einfacher. Sie besteht aus einem einzelnen <*span*>-Element.

Wir haben außerdem das Unicode-Zeichen für Grad Celsius hinzugefügt (&#x2103), damit die Sache professioneller wirkt. Nun verwenden wir ein wenig CSS, damit das Cockpit noch besser aussieht:

InputDevices/Dashboard/css/dashboard.css

```
body {
  font-size: 50px;
  background: black;
  color: white;
}

#distance-display {
  text-align: center;
}

#temperature-display {
  text-align: center;
}
```

Das Stylesheet vergrößert die Schrift, stellt die Hintergrundfarbe auf Schwarz und die Textfarbe auf Weiß. Außerdem zentriert sie die LED- und Temperaturanzeige.

Nun wird es Zeit, das Cockpit mit JavaScript zum Leben zu erwecken:

InputDevices/Dashboard/js/dashboard.js

```
var arduino = new SerialDevice("/dev/tty.usbmodem24311", 9600);

arduino.onConnect.addListener(function() {
  console.log("Connected to: " + arduino.path);
});

arduino.onReadLine.addListener(function(line) {
  console.log("Read line: " + line);
  var attr = line.split(",");
  if (attr.length == 2) {
    var temperature = Math.round(parseInt(attr[0]) / 100.0 * 10) / 10;
    var distance = parseInt(attr[1]) / 100.0;
    updateUI(temperature, distance);
  }
});

var lights = {
  d1: [35.0, "orange"],
  d2: [30.0, "orange"],
  d3: [25.0, "orange"],
  d4: [20.0, "orange"],
```

```
  d5: [15.0, "orange"],
  d6: [10.0, "orange"],
  d7: [7.0, "red"],
  d8: [5.0, "red"]
};

function updateUI(temperature, distance) {
  document.getElementById("temperature").innerText = temperature;
  for (var i = 1; i < 9; i++) {
    var index = "d" + i;
    if (distance <= lights[index][0])
      document.getElementById(index).style.color = lights[index][1];
    else
      document.getElementById(index).style.color = "white";
  }
}

arduino.connect();
```

Um die Sensordaten vom Arduino zu lesen, verwenden wir die Klasse *SerialDevice*, die wir in Anhang D.5, *Eine Klasse für serielle Geräte schreiben*, definiert haben. Stellen Sie sicher, dass Sie den richtigen seriellen Port verwenden.

Dann definieren wir einen *onConnect*-Handler, der eine Nachricht auf der JavaScript-Konsole des Browsers ausgibt, wenn die Anwendung sich mit dem Arduino verbunden hat.

Im Prinzip benötigen Sie den *onConnect*-Handler nicht. In diesem Fall dient er nur zum leichteren Finden von Fehlern.

Mit dem *onRead*-Handler wird die Sache schon interessanter. In Zeile 9 teilen wir die vom Arduino erhaltenen Daten auf. Wir stellen sicher, dass wir zwei Werte empfangen haben. Dann wandeln wir beide Werte mittels *parseint* in Zahlen um und teilen sie durch 100, da der Arduino Werte sendet, die vorher mit 100 multipliziert wurden. In Zeile 11 nutzen wir einen beliebten Trick in JavaScript, um den Temperaturwert auf eine Dezimalstelle zu runden. Nachdem wir Abstand und Temperatur in korrekte Zahlen umgewandelt haben, übergeben wir sie an *updateUI*.

updateUI setzt in Zeile 29 erst einmal den neuen Temperaturwert. Dazu schlägt es das HTML-Element mit der ID *temperature* mittels der Funktion *getElementById* nach. Dann setzt es die Eigenschaft *innerText* auf die aktuelle Temperatur.

Das künstliche LED-Display zu aktualisieren ist ein bisschen komplizierter, aber nicht zu schwierig. Wir haben eine Datenstruktur namens *lights* definiert. Sie bildet die IDs der <*span*>-Elemente unserer Anzeige auf Arrays mit jeweils zwei Werten ab.

Zum Beispiel verweist die ID d1 auf ein Array, das die Werte 35.0 und »Orange« enthält. Dadurch wird die Farbe des Elements mit der ID d1 auf Orange gesetzt, wenn der Abstand zum nächsten Objekt 35 Zentimeter oder weniger beträgt.

Mit der Datenstruktur *lights* ist es einfach, das LED-Display zu implementieren. In Zeile 30 beginnen wir mit einer Schleife, die über alle LEDs läuft. Wir legen

die ID der aktuellen LED in Zeile 31 fest. Dann prüfen wir, ob der aktuelle Abstand kleiner oder gleich dem Schwellenwert ist, der zur aktuellen LED gehört. Wenn ja, ändern wir die Farbe der LED entsprechend. Wenn nicht, setzen wir die Farbe auf Weiß.

Spaß mit Sensoren

Mit einem Ultraschallsensor können Sie ganz leicht ermitteln, ob sich jemand in der Nähe befindet. Dazu fallen einem ganz unvermittelt eine Menge nützlicher Anwendungen ein. Beispielsweise können Sie eine Tür automatisch öffnen lassen, wenn jemand nahe genug an sie herantritt.

Außerdem können Sie moderne Technik auch für reine Spaßanwendungen nutzen. Wie wäre es mit einem Kürbis als Halloween-Gag, der immer dann Luftschlangenspray verschießt, wenn Sie eine unsichtbare Linie überschreiten?[a] Das wäre ein netter Gag für Ihre nächste Party, und Sie können ihn mit dem PING)))-Sensor umsetzen.

a *http://www.instructables.com/id/Arduino-controlled-Silly-String-shooter/*

Schließen Sie den Arduino an Ihren Computer an und laden Sie den im vorherigen Abschnitt entwickelten Sketch hoch. Starten Sie die Chrome-App und bewegen Sie Ihre Hand vor dem PING)))-Sensor vor und zurück. Das Display auf dem Bildschirm sieht genauso aus, wie in einem normalen Auto.

Sensoren sind eine spannende Sache und in diesem Kapitel haben Sie die Grundlagen gelernt, um analoge und digitale Sensoren zu verwenden. Im nächsten Kapitel legen wir nach und schließen Arduino an einen Beschleunigungsmesser an, um einen Game-Controller mit Bewegungserkennung zu bauen.

5.6 Wenn es nicht funktioniert

Schlagen Sie im Abschnitt »Wenn es nicht funktioniert« von Kapitel 3 nach und vergewissern Sie sich, dass Sie alle Teile ordnungsgemäß mit dem Breadboard verbunden haben. Seien Sie besonders sorgfältig beim Umgang mit dem PING)))- und dem TMP36-Sensor, da Sie mit diesen noch nicht gearbeitet haben. Vergewissern Sie sich, dass Sie die richtigen Pins mit den richtigen Anschlüssen der Sensoren verbunden haben.

Bei Fehlern in der Software – sei es der JavaScript- oder der Arduino-Code – können Sie den Code von der Website zum Buch herunterladen und ausprobieren, ob es damit geht.

Bei Problemen mit der seriellen Kommunikation überprüfen Sie sorgfältig, ob Sie die richtigen seriellen Ports und den richtigen Arduino-Typ verwendet haben. Denken Sie auch daran, den Pfad Ihres Arduinos in der ersten Zeile von *dashboard.js* anzupassen. Prüfen Sie auch, ob die Baudrate im JavaScript-Code mit der im Arduino-Code übereinstimmt.

Stellen Sie sicher, dass der serielle Port nicht von anderen Anwendungen blockiert wird, z.B. von einem Fenster des seriellen Monitors, das Sie zu schließen vergessen haben.

5.7 Übungen

- Bauen Sie einen automatischen Einbruchalarm, der auf dem Bildschirm ein Stoppschild[5] anzeigt, wenn jemand Ihrem Computer zu nahe kommt. Programmieren Sie die Anwendung so intelligent wie möglich. Beispielsweise sollte Sie eine kleine Aktivierungsverzögerung aufweisen, damit sie nicht sofort beim Starten ein Stoppschild anzeigt.
- Die Schallgeschwindigkeit hängt nicht nur von der Temperatur, sondern auch von der Luftfeuchtigkeit und dem Luftdruck ab. Recherchieren Sie die entsprechenden Formeln und die geeigneten Sensoren und nutzen Sie Ihre Ergebnisse, um die Abstandmessung Ihrer Schaltung noch präziser zu machen.
- Verwenden Sie eine alternative Technologie für die Entfernungsmessung, z.B. Infrarotsensoren. Versuchen Sie, einen geeigneten Sensor zu finden, lesen Sie das zugehörige Datenblatt und erstellen Sie eine einfache Schaltung, mit der Sie die Entfernung zum nächstgelegenen Objekt am seriellen Port ausgeben können.

5 Ein Stoppschild können Sie hier finden: *http://en.wikipedia.org/wiki/File:Stop_sign_MUTCD.svg*.