Projekte mit Arduino und ESP

Danny Schreiter

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Informationen sind im Internet über http://dnb.d-nb.de abrufbar.

www.bmu-verlag.de
info@bmu-verlag.de

Lektorat: Dr. phil. Annette Schönberg-Al Meklef
Einbandgestaltung: Pro ebookcovers Angie
Druck und Bindung: Wydawnictwo Poligraf sp. zo.o. (Polen)

Taschenbuch-ISBN: 978-3-96645-112-3
Hardcover-ISBN: 978-3-96645-113-0
E-Book-ISBN: 978-3-96645-111-6

Projekte mit Arduino und ESP

Inhaltsverzeichnis

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:
https://bmu-verlag.de/projekte-arduino-esp

Außerdem erhalten Sie die eBook-Ausgabe zum Buch im PDF-Format kostenlos auf unserer Website:

https://bmu-verlag.de/projekte-arduino-esp
Downloadcode: siehe Kapitel 28

Kapitel 1
Einleitung

Kreativität ist nicht an Leinwände gebunden. Auch nicht an Bücher oder Musikinstrumente. Kreativität bedeutet, etwas zu erschaffen – etwas Neues, Nützliches, Originelles. Das gilt auch im Bereich der Elektronik! Die rasante Entwicklung der Mikroelektronik während der letzten Jahrzehnte hat nicht nur ermöglicht, dass an Computern fantastische Welten erschaffen werden, sondern auch, dass normale Hausgeräte immer „smarter“ werden. Wie clever ein Küchenradio ist, hängt kaum noch von technischen Beschränkungen, sondern fast nur noch vom Einfallsreichtum des Erfinders ab.

Die Arduino-Welt bietet mit ihrer flexiblen Mikrocontroller-Plattform unzählige Möglichkeiten, Kreativität und Ideenreichtum in praktische, dekorative oder gar skurrile Objekte und Geräte münden zu lassen. Dieses Buch richtet sich an all jene, die mit Lötkolben und Tastatur selbst zum kreativen Schöpfer werden möchten. Vielleicht haben Sie bereits eine Idee im Kopf und wissen noch nicht so recht, wie man sie umsetzen könnte? Auf den folgenden Seiten erhalten Sie viele Impulse, die Ihnen zeigen, was möglich ist. Oder treibt Sie gar kein Schöpfergeist, aber Sie haben enormen Spaß am Basteln? Auch in diesem Fall sind Sie hier genau richtig.

In den folgenden Kapiteln werden 25 Projekte vorgestellt, die jeder Heimwerker nachbasteln kann. Kreative Anpassungen sind dabei durchaus willkommen! Niemand zwingt Sie, ein Projekt genau wie hier vorgestellt umzusetzen. Lassen Sie gern eigene Ideen einfließen und entdecken Sie neue Möglichkeiten.

Die Projekte steigen in ihrer Schwierigkeit und ihren Anforderungen an. Außerdem wird in jedem Kapitel ein bestimmter Aspekt, beispielsweise eine bestimmte Programmierweise oder eine neues Hardware-Modul, etwas ausführlicher behandelt. Sollten Sie direkt mit einem

hinteren Kapitel anfangen wollen und dabei auf Probleme stoßen, blättern Sie doch einfach nochmal ein paar Seiten zurück. Um die Anleitungen nachvollziehen zu können, sollten Sie bereits mit der Software *Arduino IDE* vertraut sein und wissen, wie man in dieser Entwicklungsumgebung einen Sketch auf den Arduino lädt. Die grundlegende Struktur eines Arduino-Programms (mit Befehlen wie `setup()` und `loop()`) sollte Ihnen genauso wenig Kopfzerbrechen bereiten wie die Bedeutung der Variablentypen `byte` und `long`. Falls Sie Ihr Wissen im Bereich der Programmier- oder Elektronik-Grundlagen wieder etwas auffrischen möchten, sei an dieser Stelle auf das ebenfalls im BMU-Verlag erschienene Arduino-Kompendium verwiesen.

Und bitte lassen Sie sich nicht von den teils umfangreichen Sketchen irritieren: Alle aufgeführten Programmbeispiele können Sie bequem online von der Verlagsseite herunterladen – Sie müssen also nichts abtippen.

Nun steht dem Bastelspaß nichts mehr im Wege. Ich wünsche Ihnen viel Vergnügen und gutes Gelingen!

Kapitel 2

LED-Würfel

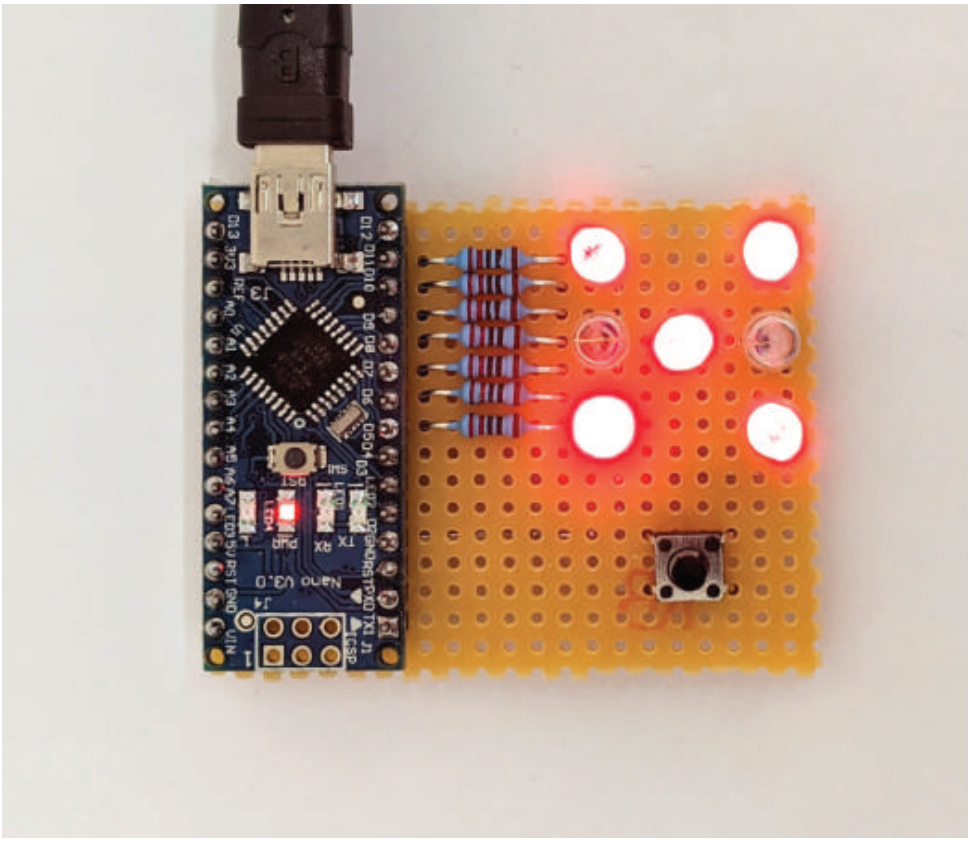

Abb. 2.1 Das fertige Modul findet auf einer kleinen Platine Platz.

Unser erstes Projekt soll ein elektronischer Würfel sein, der mit Hilfe von LEDs die „erwürfelte“ Zahl darstellt. Dabei genügen 7 LEDs, um jedes mögliche Ergebnis anzeigen zu können:

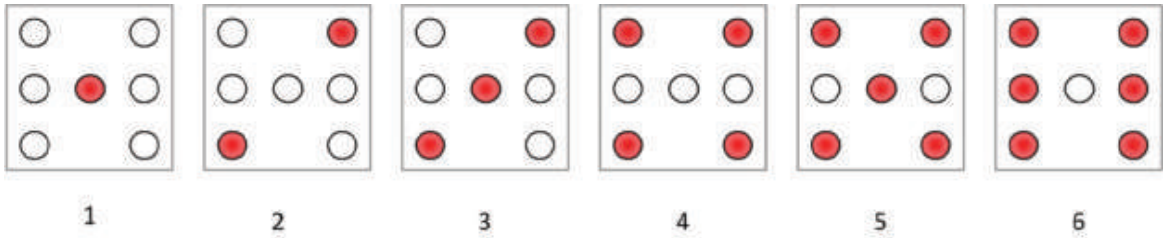

Abb. 2.2 Alle Würfelbilder können mit sieben LEDs dargestellt werden.

2.1 Materialien

Folgende wesentliche Bauteile[1] werden benötigt:

- 7 LEDs, zum Beispiel 5 mm, rot
- 7 Vorwiderstände, hier: 220 Ω
- 1 Arduino Nano bzw. gleichwertiger Nachbau
- 1 Taster

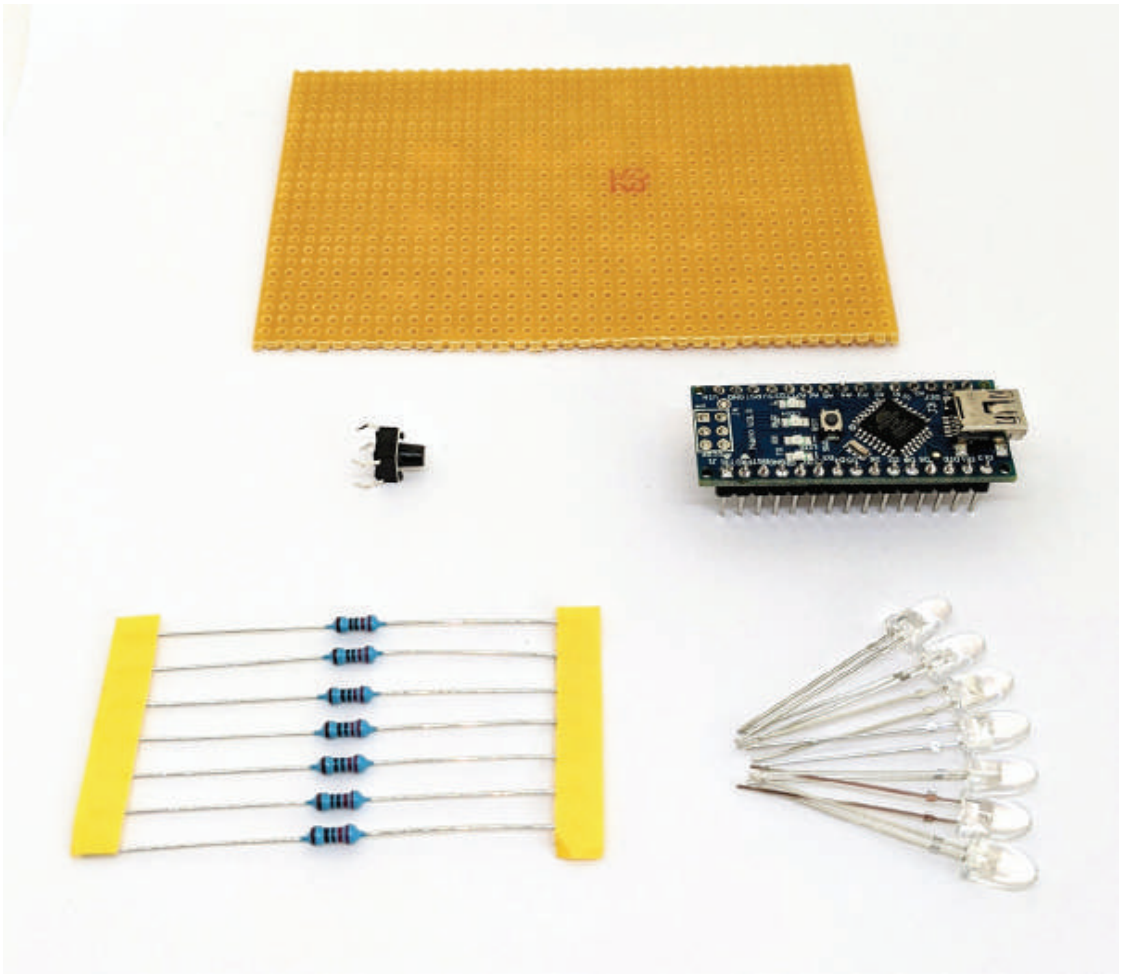

Abb. 2.3 Jede LED benötigt ihren eigenen Vorwiderstand.

[1] Die Bezugsquellen der Materialien finden Sie im Anhang des Buches. Grundlegende Ausstattung (Lötzinn, Lochrasterplatinen, Litze, Heißkleber, ...) wird nicht gesondert aufgeführt.

Im Beispiel nutzen wir eine Platine, bei der die Verbindungen fest verlötet werden. Entsprechend sind dafür eine Lochrasterplatine, Litze und Löt-Equipment notwendig. Alternativ kann die Schaltung aber auch problemlos auf einem Breadboard (Steckbrett) aufgebaut werden. Die Verdrahtung ist relativ simpel: Da der Arduino Nano 13 digitale Ein- und Ausgangspins besitzt, können wir für jede der 7 LEDs einen eigenen Pin verwenden. Ein weiterer Pin wird als digitaler Eingang für den Taster genutzt.

Auf keinen Fall sollten die Vorwiderstände der LEDs vergessen werden, denn sie dienen der Strombegrenzung. Beim direkten Anschluss an den HIGH-Pegel der Ausgangspins (5 V) ergäbe sich ein zu hoher Strom, der sowohl die LEDs als auch die Mikrocontroller-Ausgänge beschädigen könnte. Die hier genutzten roten LEDs haben eine Flussspannung von etwa $U_F = 1{,}8\ \mathrm{V}$. Der zulässige Strom ist vom Hersteller mit $I_F = 15$ mA. angegeben[2]. Mit der in der Arduino-Welt üblichen Betriebsspannung $U_0 = 5\ \mathrm{V}$. lässt sich aus diesen Werten der benötigte Vorwiderstand berechnen:

$$R_V = \frac{U_0 - U_F}{I_F} = \frac{5\ \mathrm{V} - 1{,}8\ \mathrm{V}}{0{,}015\ \mathrm{A}} = 213\ \Omega$$

Widerstände sind allerdings nicht mit beliebigen Werten, sondern in einer genormten Folge, der sogenannten E-Reihe, erhältlich. Im Normalfall wählt man dabei stets die zum berechneten Wert nächsthöhere Stufe, da ein Abrunden zu einem zu hohen Strom führen könnte. In unserem Beispiel wählen wir daher $R_V = 220\ \Omega$.

Wenn Ihnen für Ihre LEDs keine Kenndaten vorliegen, können Sie sich für übliche Modelle an der folgenden Tabelle orientieren, sofern die Bauform den LEDs in diesem Beispiel ähnelt. Viele Modelle halten auch Ströme bis 20 mA aus. In dieser Tabelle wurden als Strom nur 10 mA

[2] Der Index *F* steht für „forward“. Gemeint ist damit, dass die (Leucht-)Diode in Durchlassrichtung, also „vorwärts“ betrieben wird. Einige Arten von Dioden lassen sich auch in ihre Sperrichtung („reverse“) nutzen, bei LEDs ist dies jedoch nicht der Fall.

angesetzt, da bei diesem Wert fast immer ein Leuchten zu sehen ist und selbst empfindliche LEDs dadurch nicht zerstört werden.

LED-Farbe	U_F	I_F	R_V (berechnet → aufgerundet)
rot	1,8 V	10 mA	320 Ω → 330 Ω
gelb	2,1 V	10 mA	290 Ω → 330 Ω
grün	2,2 V	10 mA	280 Ω → 330 Ω
blau	3,1 V	10 mA	190 Ω → 220 Ω
weiß	3,1 V	10 mA	190 Ω → 220 Ω

Tab. 2.1 Vorwiderstände handelsüblicher LEDs (bei 5 V Betriebsspannung).

Die Zuordnung der LEDs zu den Ausgangspins kann willkürlich erfolgen, bei nachfolgendem Schema wurde eine möglichst einfache Verdrahtung auf der späteren Platine zum Ziel gesetzt.

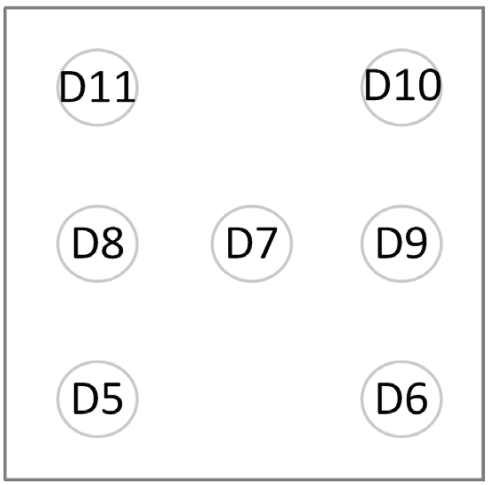

Abb. 2.4 Zuordnung von Ausgangspins zu den LEDs des Würfelbildes.

Zusammen mit den Vorwiderständen ergibt sich eine überschaubare Verdrahtung. Im Stromlaufplan wurden zur besseren Übersicht alle Masseverbindungen blau dargestellt, die anderen Verbindungen schwarz.

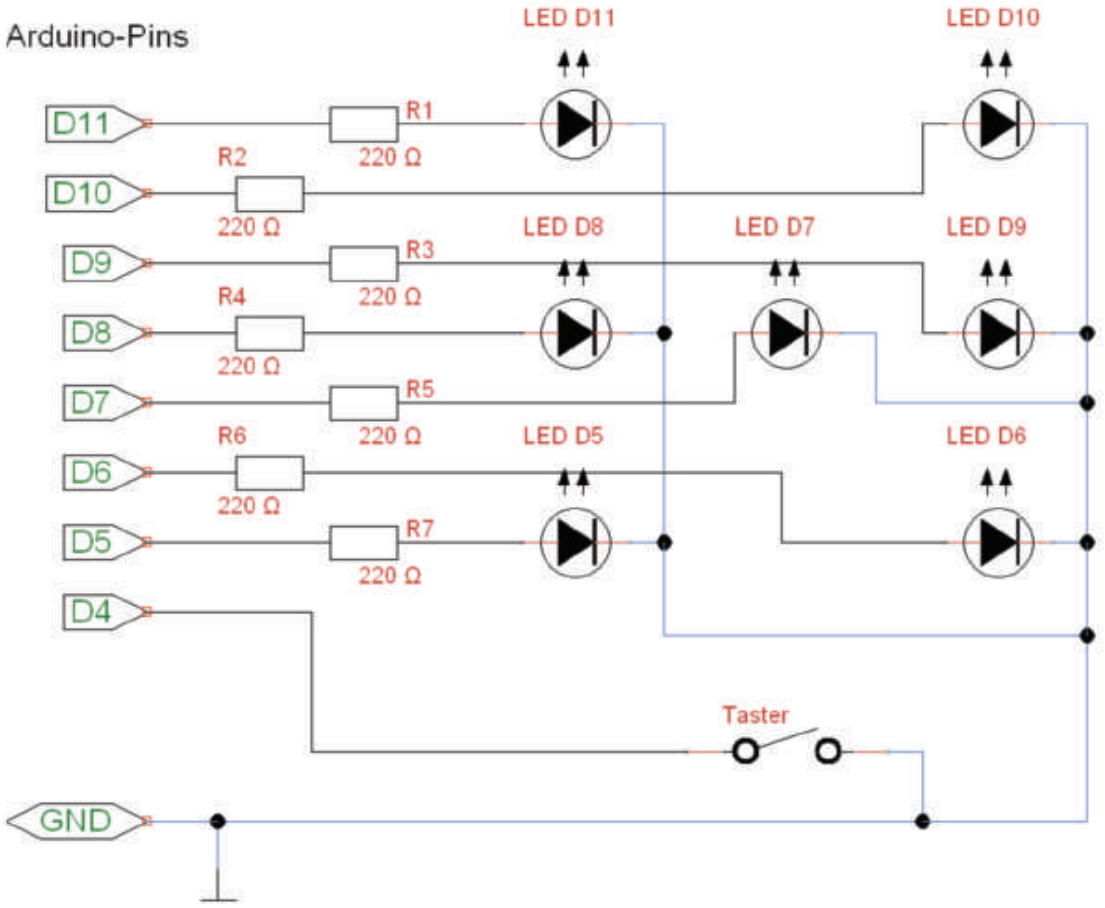

Abb. 2.5 Der Stromlaufplan wird dominiert von den LEDs und ihren Vorwiderständen.

Ein Sketch zu diesem Projekt lässt sich mit grundlegenden Arduino-Kenntnissen problemlos erstellen. Probieren Sie es gern einmal! Wenn Sie sich an die wesentlichen Funktionen `digitalWrite()` und `random()` erinnern, wird es Ihnen nicht schwerfallen.

Dieses Buch hat jedoch den Anspruch, dass in jedem Kapitel zusätzliche Kenntnisse vermittelt werden, die über die Grundlagen aus dem Arduino-Kompendium hinausgehen. Daher wollen wir dieses Beispiel nutzen, um die sogenannten Portregister kennenzulernen.

2.2 Portregister

Ein Register ist eine Byte-Variable, welche sich im Mikrocontroller an einem speziell festgelegten Speicherort befindet. Die Speicherzellen der einzelnen Bits solcher Variablen sind fest mit weiteren Schaltkreisen verdrahtet, um auf Hardware-Ebene direkten Einfluss auf bestimmte Funktionen zu nehmen. Bei den Port-Registern handelt es sich entsprechend um Register, welche den Ein- und Ausgangspins zugeordnet sind. Aufgrund der Anzahl der Ein- und Ausgangspins beim Arduino Nano gibt es insgesamt drei Port-Registersätze: B, C und D. Beim Arduino UNO ist die Belegung identisch, nur die Pins A7 und A6 entfallen. In der folgenden Tabelle ist dargestellt, welche Pins welchen Bits der 8-Bit-Registersätze zugeordnet sind.

	MSB[3]							LSB[4]
Registersatz D	D7	D6	D5	D4	D3	D2	D1	D0
Registersatz B	--	--	D13	D12	D11	D10	D9	D8
Registersatz C	A7	A6	A5	A4	A3	A2	A1	A0

Tab. 2.2 Zuordnung der Pins zu den Portregistern.

Jeder Registersatz besteht wiederum aus drei Registern: DDRx, PORTx und PINx, wobei das x für den entsprechenden Registersatz steht. Die digital-Pins D0 bis D7 werden also durch die Register DDRD, PORTD und PIND gesteuert.

- DDRx (*Data Direction Register*): Dieses Register gibt an, ob ein Pin als Ein- oder Ausgang fungieren soll. Bei Ausgängen ist das entsprechende Bit 1, bei Eingängen 0. Dieses Register darf gelesen und ge-

[3] MSB – ***most significant bit – höchstwertigstes Bit***, dieses Bit hat in einer 8-Bit-Binärzahl den Dezimalwert „128“

[4] LSB – ***least significant bit – niedrigwertigstes Bit***, dieses Bit hat in der Binärzahl den Dezimalwert „1“

schrieben werden. Die bekannte Funktion `pinMode()` verändert die Bits in diesem Register.

- PORTx: Dieses Register legt den Pegel eines Ausgangspins fest. Beim Binärwert 1 führt der zugehörige Ausgang einen HIGH-Pegel, sonst LOW. Wenn der zugehörige Pin im DDRx-Register als Eingang (Binärwert 0) konfiguriert ist, bestimmt das PORT-Register, ob die Pull-Up-Widerstände aktiv sind (binär 1) oder nicht (binär 0). Dieses Register darf ebenfalls sowohl geschrieben als auch gelesen werden.
- PINx: Dieses Register darf nur gelesen werden. Es enthält die Zustände der als Eingang definierten Pins. Ist das entsprechend Bit 1, so liegt am Eingang ein HIGH-Pegel an; bei einer 0 ist der Eingangspegel entsprechend LOW.

Der Registersatz C enthält die analogen Eingangspins, die auch wie normale Digitalspins genutzt werden können. Die Möglichkeit, analoge Eingangssignale zu verarbeiten, ist also lediglich eine Zusatzfunktion dieser Anschlüsse. Auch die Analogwerte lassen sich (alternativ zur Funktion `analogRead()`) direkt aus bestimmten Registern auslesen, diese Ausführung wäre hier jedoch zu umfangreich.

Die bereits bekannten Funktionen `pinMode()`, `digitalWrite()` und `digitalRead()` greifen also auf genau diese Register zu, und zwar in einer nutzerfreundlichen Form. Welchen Sinn macht es dann, einen so genauen Blick auf die Portregister zu werfen? Zugegeben, bei unserem einfachen LED-Würfel ist es tatsächlich egal, wie man auf die Pins zugreift. Dennoch bietet der direkte Portzugriff in der fortgeschrittenen Programmierung einige Vorteile:

- Der direkte Zugriff erfolgt mehr als 10-mal schneller als die Pinmanipulation über die genannten Funktionen. Die Ursache liegt darin, dass der Compiler Funktionen wie `digitalWrite()` in mehrere kleine Schritte (Microcode) zerlegen muss und dabei mögliche Optimierungen/Zusammenfassungen nicht immer erkannt werden können. Auch das direkte Auslesen der Register ist um ein Vielfaches schneller als das Lesen per `digitalRead()`. Für Signale, die sich

sehr schnell ändern (beispielsweise die codierten Signale von Infrarot-Fernbedienungen) kann dies entscheidend sein.

- Der Zugriff auf ein PORTx-Register erlaubt es, mehrere Ausgangspins dieses Registers zur exakt gleichen Zeit umzuschalten. Per `digitalWrite()` wäre dies nur nacheinander, mit wenigen Mikrosekunden Unterschied, möglich. Bei externen Modulen, die empfindlich auf derartige Zeitunterschiede reagieren, kann dies wichtig sein.
- Ein weiterer Vorteil ist es, dass die Befehle für den direkten Portzugriff im späteren Maschinencode deutlich weniger Speicherplatz benötigen als die herkömmlichen Funktionen. Umfangreiche Programme mit zahlreichen Portzugriffen können dadurch verschlankt werden.

In einem Arduino-Sketch können diese Register wie einfache Präprozessor-Konstanten angesprochen werden. Der Präprozessor erkennt die Namen der Register und ersetzt sie durch die korrekte Speicheradresse.

So würde der Befehl

```
1  DDRD = B00100000;          // das vorangestellte B markiert die
2                             // binäre Schreibweise
```

den Pin D5 als Ausgang definieren und gleichzeitig alle anderen Pins dieses Registers (D0 bis D7) als Eingänge. Alternativ könnte man dafür auch schreiben

```
1  DDRD = 32;
```

denn der obige Binärwert entspricht dezimal der Zahl 32. Üblicherweise möchte man aber nicht alle Pins eines Registers gleichzeitig manipulieren, sondern nur einige oder einen einzigen. In diesem Zusammenhang macht man sich binäre Operatoren zunutze, welche jedes einzelne Bit des Registers mit einem 8-Bit-Muster verknüpfen und so nur einzelne Bits ändern. Dies wird am besten an Beispielen deutlich:

Setzen eines Bits – bitweise ODER:

```
1  DDRD = DDRD | B00100000;
```

Der Operator | verknüpft jeweils alle gleichwertigen Bit zweier Bytes nach der ODER-Logik miteinander (also das höchstwertige Bit von DDRD mit dem höchstwertigen Bit von B00100000, das zweithöchstwertige Bit von DDRD mit dem zweithöchstwertigen von B00100000 und so weiter). Das Ergebnis-Bit ist 1, wenn mindestens eines der beiden Eingabe-Bits 1 ist. Obige Verknüpfung bewirkt somit, dass das für D5 zuständige Bit im DDRD in jedem Fall auf 1 gesetzt wird, denn „a ODER 1“ ergibt in jedem Fall 1 – egal, welchen Wert das Bit a hat. Alle anderen Bits im Register DDRD bleiben währenddessen unangetastet, denn „b ODER 0“ ergibt immer den Wert von b. Somit führt obige Anweisung dazu, dass D5 als Ausgang definiert wird, während alle anderen unverändert bleiben – die gleiche Wirkung wie `pinMode(5, OUTPUT)`.
Für obigen Ausdruck ist übrigens auch folgende Kurzschreibweise zulässig:

```
1  DDRD |= B00100000;
```

Löschen eines Bits – bitweise UND:

Möchte man ein bestimmtes Bit auf 0 setzten (also löschen), so eignet sich dafür die bitweise-UND-Verknüpfung (Operator &). Diese ergibt nur 1, wenn beide Eingabe-Bits auch 1 sind.

```
1  DDRD = DDRD & B11011111;    // ausführliche Schreibweise
2  DDRD &= B11011111;          // gleichwertige Kurzschreibweise
```

Im dargestellten Beispiel wird D5 auf 0 gesetzt (also als Eingang definiert), während alle anderen unverändert bleiben – denn „a UND 0“ ergibt immer 0, wohingegen „b UND 1“ stets den Wert von b ergibt.

Auslesen eines einzelnen Eingangs-Pins – bitweise UND:

Die eben vorgestellte Verknüpfung lässt sich auch nutzen, um Eingangspins auszulesen. So ist die Bedingung

```
1  if(PIND & B00100000) { ...
```

genau dann erfüllt, wenn am (zuvor als Eingang definierten) Pin D5 ein HIGH-Signal anliegt, denn nur das Bit an dieser Stelle beeinflusst das Ergebnis. Alle anderen Bits werden durch die „UND 0"-Verknüpfung ausgeblendet – man spricht in diesem Zusammenhang auch von Bit-Maskierung. Das rechte Byte (B00010000) ist die Bitmaske. Liegt am Pin D5 ein LOW-Pegel, so sind im Ergebnis alle Bits 0 und die `if`-Anweisung wertet dies als `false` – also als nicht erfüllte Bedingung. Jeder Wert größer als null wird hingegen als `true` gewertet. Daher ist der sich ergebende Zahlenwert (in diesem Beispiel würde ein HIGH-Pegel an D5 das Ergebnis B00100000, also dezimal 32, liefern) nicht von Belang. Obige Anweisung ist demnach gleichwertig mit der folgenden:

```
1  if(digitalRead(5)) { ...
```

Durch Variieren der Bitmasken lassen sich auch mehrere Pins gleichzeitig manipulieren oder auslesen. Es gibt zudem noch weitere binäre Operatoren; in diesem Zusammenhang sei auf entsprechende Nachschlagewerke verwiesen. So können mittels der bitweise-NICHT-Verknüpfung (Operator ~) zum Beispiel alle Bits eines Bytes invertiert werden. Die folgende Anweisung würde den Zustand aller Pins D0 bis D7 umschalten, so ließe sich ein einfacher Wechselblinker realisieren.

```
1  PORTD = ~PORTD;
```

2.3 Sketch

Widmen wir uns nun also dem Sketch (Programmcode), welcher von den Portregistern Gebrauch macht, um unseren LED-Würfel zu realisieren. In den Downloads zum Buch finden Sie als Extra auch einen (selbsterklärenden) Sketch, der für den Pin-Zugriff die herkömmlichen Funktionen nutzt.

```
1  byte Bitmuster;
2  byte Zahlen[] = {                          // (1)
3    B00000000,        // alle aus
4    B00000100,        // 1
5    B00100001,        // 2
6    B00100101,        // 3
7    B01100011,        // 4
```

```
  B01100111,        // 5
  B01111011         // 6
};

void setup() {
  DDRD  = B11100000;                  // (2)
  DDRB  = B00001111;
  PORTD = B00010000;                  // (3)
}

void loop() {
  while(PIND & B00010000);                  // (4)

  randomSeed(analogRead(1)+micros());       // (5)

  for(byte i = 0; i < 30; i++)              // (6)
  {
    Bitmuster = Zahlen[random(6)+1];        // (7)
    PORTD = (Bitmuster << 5 | B00010000);   // (8)
    PORTB = Bitmuster >> 3;                 // (9)
    delay(10 + i*5);                    // (10)
  }

  Bitmuster = Zahlen[random(6)+1];          // (11)

  for(byte i = 0; i < 5; i++)               // (12)
  {
    byte AlleAus = Zahlen[0];
    PORTD = (AlleAus << 5 | B00010000);
    PORTB = AlleAus >> 3;
    delay(200);

    PORTD = (Bitmuster << 5 | B00010000);
    PORTB = Bitmuster >> 3;
    delay(300);
  }
}
```

(1) Im Array `Zahlen[]` sind den Würfelwerten eins bis sechs die entsprechenden Würfelaugen (LEDs) zugeordnet, und zwar jeweils ein Byte (also acht Bit) pro Würfelwert. Für die sieben LEDs werden nur sieben Bit benötigt, das jeweils achte Bit ist ohne Funktion und daher willkürlich auf null gesetzt. Zur einfacheren Handhabung wurde ein zusätzlicher Eintrag vorangestellt, der einfach alle LEDs ausschaltet. Dieser erhält den Index 0, so dass die Indizes der folgenden

Einträge genau deren Würfelwerten entsprechen. Die geplante Zuordnung der Bits zu den Pins kann folgender Tabelle entnommen werden, sie wird später im Programm umgesetzt. Die Zuordnung der Pins zu den konkreten LEDs ist in Abb. 2.4 ersichtlich. Einer gewürfelten 1 ist im Array der Wert B00000100 zugeordnet, dadurch wird alleinig Pin D7 auf HIGH geschaltet, welcher die mittlere LED leuchten lässt, und so weiter.

MSB PORTB							LSB	MSB PORTD							LSB
--	--	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
			0	X	X	X	X	X	X	X					
			MSB	Zahlen[]						LSB					

Tab. 2.3 geplante Zuordnung der sieben relevanten Zahlen[]-Bits.

(2) Die beiden Data-Direction-Register werden genutzt, um die Pins D5 bis D11 als Ausgang (binär 1) zu definieren und Pin D4 als Eingang (binär 0). Die anderen Pins sind nicht relevant, wir setzen sie willkürlich auf 0, um sie als Eingänge zu konfigurieren.

(3) Im Port-Register setzen wir das Bit für den Pin D4 auf 1. Dieser Pin wurde im DDRD als Input konfiguriert, somit sorgt die binäre 1 im Port-Register dafür, dass der Pull-Up-Widerstand aktiviert wird. Dies ist nötig, da wir einfach nur einen Taster gegen Masse (GND) anschließen. Im unbetätigten Zustand hätte der Pin ohne Pull-Up kein definiertes Potential.

(4) Diese leicht zweckentfremdete `while`-Schleife enthält keine Anweisungen, denn hinter der schließenden Klammer folgt direkt das Semikolon statt eines Schleifenrumpfes. Das ist aber auch gar nicht nötig: Das Programm soll an dieser Stelle einfach so lange verharren, bis die angegebene Bedingung nicht mehr erfüllt ist. In der Bedingung wird das Register PIND mit einer Bitmaske verknüpft, welche das für D4 zuständige Bit selektiert. Die Bedingung ist also so lange erfüllt, wie an D4 ein HIGH-Pegel anliegt. Das ist aufgrund des internen Pull-Up-Widerstandes so lange der Fall, bis der Taster gedrückt und dadurch der Pin mit Masse (LOW) verbunden wird. Wenn dies eintritt, ist die Bedingung nicht mehr erfüllt (denn B00000000 zählt als `false`) und der Programmablauf wird fortgesetzt.

(5) Die später zum Würfeln genutzte Funktion `random()` liefert *pseudo*zufällige Zahlen. Diese wirken in ihrer Abfolge zwar zufällig, folgen aber einer festen mathematischen Funktion und sind somit vorhersagbar. Als Resultat würde der Würfel nach jedem Einschalten immer die gleiche Ergebnisreihe liefern. Abhilfe schafft `randomSeed()`. Mit dieser Funktion kann dem „Zufallsgenerator" ein anderer Startwert gegeben werden – die pseudozufällige Zahlenreihe folgt dann zwar immer noch einer festen Vorschrift, allerdings nutzt diese Vorschrift nun bei jedem Würfeln andere Parameter. Als Übergabewert sollte daher eine möglichst schwer vorhersagbare Zahl genutzt werden. Im Beispiel bedienen wir uns der Anzahl der seit Programmstart vergangenen Mikrosekunden (`micros()`) und addieren den eingelesenen Analogwert des Pins A1. Da an diesen Pin nichts angeschlossen ist („floating"), wird sein Potential von zufälligen Faktoren, wie beispielsweise elektrischen Feldern in der Umgebung, bestimmt. Beide Werte sind somit für den Anwender unmöglich genau vorhersehbar und damit ideal als Startwerte für einen Pseudo-Zufallsgenerator.

(6) Die `for`-Schleife simuliert den Würfelvorgang, indem sie in schneller Abfolge 30 zufällige Würfelergebnisse zeigt.

(7) Dafür wird zunächst ein zufälliges Bitmuster (von 1 bis 6) aus dem `Zahlen[]`-Feld ausgewählt und im Byte `Bitmuster` zwischengespeichert. Die Funktion `random()` liefert eine zufällig Zahl, die größer gleich 0 und kleiner als der übergebene Wert ist. `random(6)` liefert als Zahlen von 0 bis 5. Durch Addition von 1 kommen wir auf den typischen Würfel-Zahlenbereich.

(8) Tab. 2.3 zeigte bereits, dass sich die Bits aus der Variablen `Bitmuster` nicht direkt auf die Portregister zuordnen lassen. Dafür gibt es jedoch einfache Operatoren. Der hier genutzte <<-Operator verschiebt alle Bits um die angegebene Zahl an Schritten nach links. Bits, die dabei „herausgeschoben" werden, gehen verloren. Von rechts wird mit Nullen aufgefüllt. Nach einer Verschiebung um fünf Schritte kann der Wert ins Register PORTD kopiert werden, um die Pins D7, D6 und D5 zu setzen. Allerdings darf dabei nicht D4 auf 0 gesetzt werden, denn dieser Eintrag sorgt für den Pull-Up-Wider-

stand am Taster. Aus diesem Grund wird nach der Bitverschiebung per bitweise-ODER wieder das Bit für D4 auf 1 gesetzt.

(9) Für PORTB ist das Vorgehen ähnlich, hier ist eine Verschiebung um 3 Schritte nach rechts nötig. Auch hier gehen „herausgeschobene" Bits verloren und von links wird mit Nullen aufgefüllt. Da wir D12 und D13 nicht nutzen, ist dies nicht von Belang.

(10) Mit steigender Anzahl der Schleifendurchläufe wird die Verzögerungszeit immer weiter erhöht, um den Effekt eines „ausrollenden" Würfels zu erzielen.

(11) Nach der Animation des Würfelvorganges wird nun das Endergebnis ermittelt.

(12) Diese Schleife lässt das Endergebnis 5-mal aufblinken. Dafür wird einfach im Wechsel der Zahlenwert 0 (alle LEDs aus) und der Ergebniszahlenwert nach dem eben erläuterten Muster auf die Ports gegeben.

Anschließend beginnt die `loop()`-Schleife von vorn und die bereits erläuterte `while()`-Anweisung wartet auf den nächsten Tastendruck.

2.4 Aufbau

Dieses Projekt ist aus elektrischer Sicht recht simpel und lässt sich mühelos nach Vorbild des Stromlaufplans auf einer Experimentierplatine zusammenlöten. Wenn der LED-Würfel eine Familie mit Kindern begeistern soll und daher etwas stabiler ausgeführt werden muss, lässt sich mit etwas Geschick auch ein passendes Holzgehäuse gestalten – zum Beispiel in Würfelform!

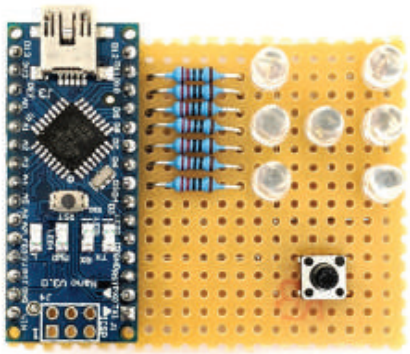

Abb. 2.6 Frontseite.

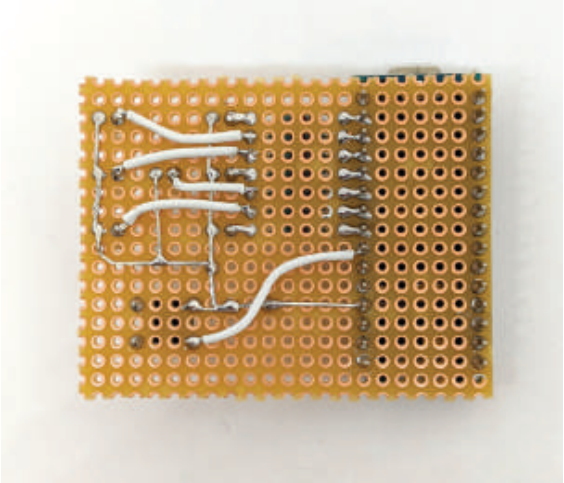

Abb. 2.7 Rückseite – die Verbindungen können am besten am Stromlaufplan nachvollzogen werden.

Damit schließen wir unser erstes Projekt ab. Vielleicht passen Sie unseren LED-Zufallsgenerator ja noch etwas an Ihre Bedürfnisse an: Wie wäre es mit einer automatischen Abschaltung der Anzeige nach einer Minute? Oder möchten Sie Ihre Mitspieler ärgern und dafür sorgen, dass eine Sechs viel seltener auftritt als alle anderen Zahlen? Sie allein entscheiden, wie viel Fairness Sie dem Spiel gönnen – die Möglichkeiten sind zahlreich. Aber denken Sie daran: Wer schummelt, spielt bald allein. Das ändert sich auch in der digitalen Welt nicht. Doch auch ohne Schummeln haben Sie schon jetzt gewonnen: Die Erkenntnis, wie man Portregister benutzt. Gratulation zum ersten Projekt!

Kapitel 3

Ventilator mit Touch-Bedienung

Unser nächstes Projekt soll ein kleiner Tischventilator sein, welcher sich über vier berührungsempfindliche Flächen steuern lässt. Derartige Touch-Bedienungen haben den Vorteil, dass sie bereits auf leichte Berührungen reagieren und somit oft eleganter wirken als herkömmliche Taster oder Schalter.

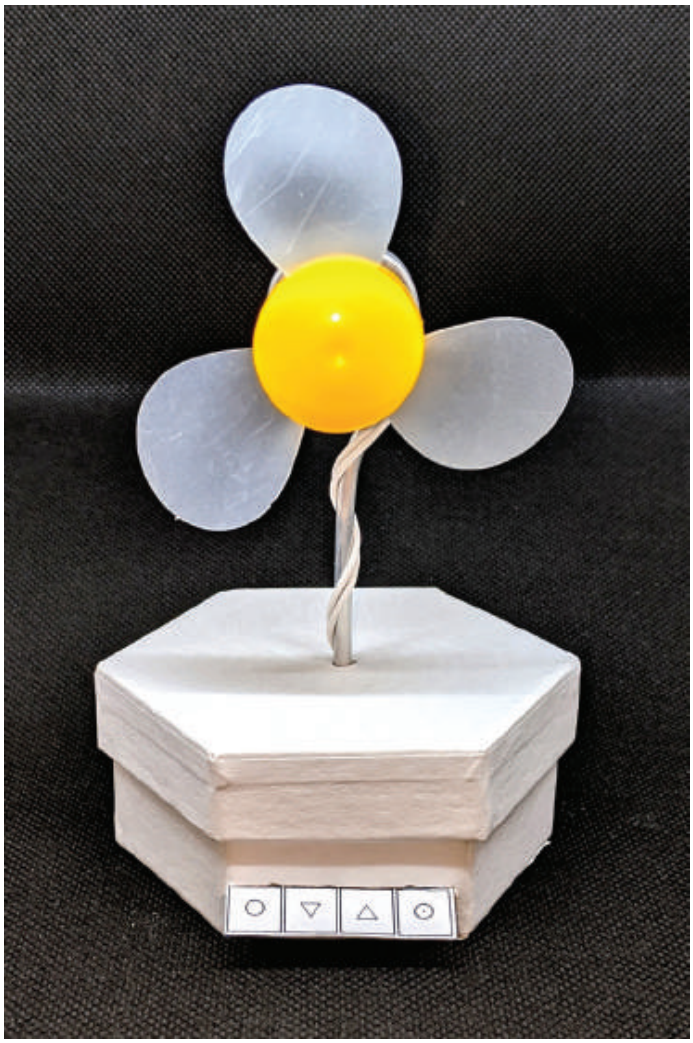

Abb. 3.1 Die vier berührungsempfindlichen Flächen an der Vorderseite des Ständers erlauben die Steuerung des Ventilators.

3.1 Kapazitive Sensoren

Üblicherweise werden für berührungsempfindliche Flächen kapazitive Sensoren verwendet. Der Name weist bereits darauf hin, dass der entscheidende Faktor die Kapazität eines Kondensators ist. Als elektronisches Bauteil besteht er in seiner einfachsten Form aus zwei leitfähigen Platten, die voneinander isoliert sind. Eine angelegte Spannung sorgt dafür, dass zwischen den Platten ein elektrisches Feld aufgebaut wird – der Kondensator lädt sich auf. Die Kapazität ist ein Maß für das „Speichervermögen" eines Kondensators. Je größter die Kapazität, desto länger dauert das Aufladen (bei ansonsten gleichen Rahmenbedingungen).

Allerdings tritt dieses Verhalten nicht nur in elektronischen Bauteilen auf, sondern jede leitende Fläche (und damit auch unsere Haut) kann selbst als Kondensator wirken. Die dabei entstehenden Kapazitäten sind allerdings sehr klein. Dennoch kann man sie messen, und genau das machen sich kapazitive Sensoren zunutze. Um deren Funktionsprinzip besser zu verstehen, können wir mit einem Breadboard und nur wenigen Bauelementen experimentell selbst einen solchen Sensor bauen.

Benötigt werden dafür ein Arduino (UNO oder Nano genügt), ein relativ großer Widerstand (1 MΩ oder größer) und ein relativ kleiner Kondensator (beispielsweise mit einer Kapazität von 22 pF). Die Idee: Wir schalten einen Ausgangspin von LOW auf HIGH. An diesem Pin befinden sich in Reihe der Widerstand und der Kondensator. Der Widerstand soll den Ladevorgang verzögern, sonst wäre der (sehr kleine) Kondensator augenblicklich voll aufgeladen. Mit einem Eingangspin (ohne Pull-Up) messen wir die Spannung am Kondensator. Nach einer bestimmten Ladezeit wird diese Spannung hoch genug sein, damit dieser Eingang als HIGH gewertet wird (etwa ab 3 V). Die bis dahin vergangene Zeit ist ein Maß für die Kapazität des Kondensators. Berührt man nun mit der Haut den markierten Kontakt, wirkt der menschliche Körper wie ein parallel geschalteter Kondensator. Dessen Kapazität addiert sich zum bereits vorhandenen Kondensator und sorgt dafür, dass die benötigte Ladezeit ansteigt.

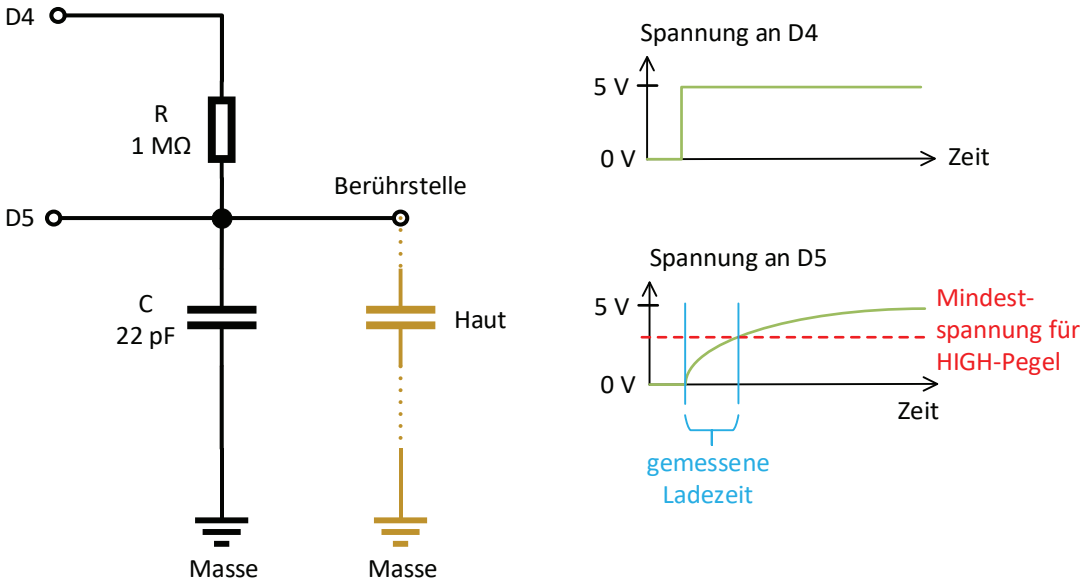

Abb. 3.2 Die Funktionsweise eines kapazitiven Sensors kann über eine einfache Schaltung simuliert werden.

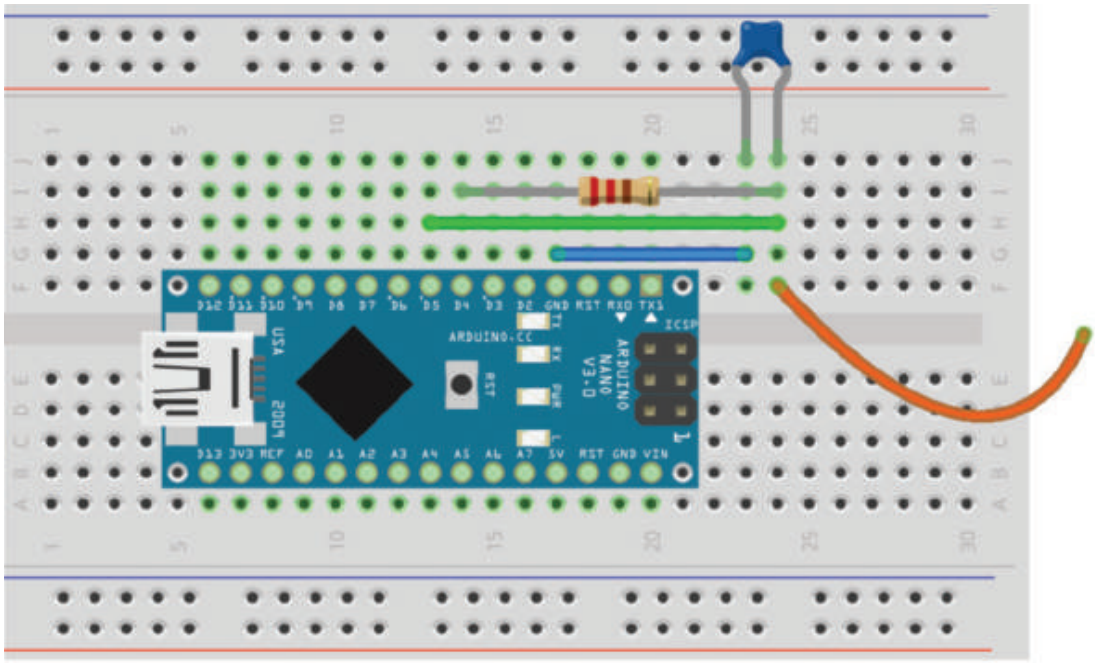

Abb. 3.3 Die Testschaltung lässt sich schnell auf einem Breadboard realisieren. Der freie orangefarbene Draht dient als Berührpunkt.

Die zu messende Ladezeit liegt im Bereich von wenigen Mikrosekunden. Es bietet sich daher an, unsere Kenntnisse über Portregister gleich

wieder anzuwenden, denn dieser Programmcode wird schneller ausgeführt als der herkömmliche Zugriff über Funktionen.

Der Sketch soll alle 100 Millisekunden eine Messung durchführen und dann das Messergebnis über die serielle Verbindung am Computer ausgeben. Zusätzlich soll die Onboard-LED (D13) aktiviert werden, wenn eine Berührung erkannt wurde.


```
long Startzeit;
long Endzeit;
long Ladezeit;

void setup() {
  DDRD = B00010000;                // D4 als Ausgang, D5 als
                                   // Eingang
  DDRB = B00100000;                // D13 als Ausgang
  PORTD = B00000000;               // Ausgang D4 LOW, Eingang D5
                                   // ohne Pull-Up
  Serial.begin(9600);
}

void loop() {

  Startzeit = micros();            // Beginn merken
  PORTD = B00010000;               // D4 auf HIGH setzen
  while(!(PIND & B00100000));      // warten, bis D5 HIGH wird
  Endzeit = micros();              // Endzeit merken
  Ladezeit = Endzeit - Startzeit;  // Zeitdifferenz berechnen
  PORTD = B00000000;               // D4 auf LOW, damit sich der
                                  // Kondensator wieder entladen
                                  // kann
  Serial.println(Ladezeit);

  if(Ladezeit > 50)                // Bei erkannter Berührung
                                   // wird
    PORTB = B00100000;             // die Onboard-LED
                                   // eingeschaltet,
  else
    PORTB = B00000000;             // sonst aus.

  delay(100);
}
```

Der Sketch ist selbsterklärend. Alle nicht relevanten Bits wurden in den Portregistern willkürlich auf 0 belassen. Eine Bitmaskierung ist hier

nicht notwendig, da die anderen Bits ungenutzt sind und ihr Zustand somit egal ist.

Zur Ausgabe am Computer eignet sich der „serielle Plotter“ noch besser als der serielle Monitor. Diese in der Arduino IDE unter „Werkzeuge“ zu findende Funktion stellt den empfangenen Zahlenwert direkt grafisch dar, wodurch sich die Funktion anschaulich testen lässt.

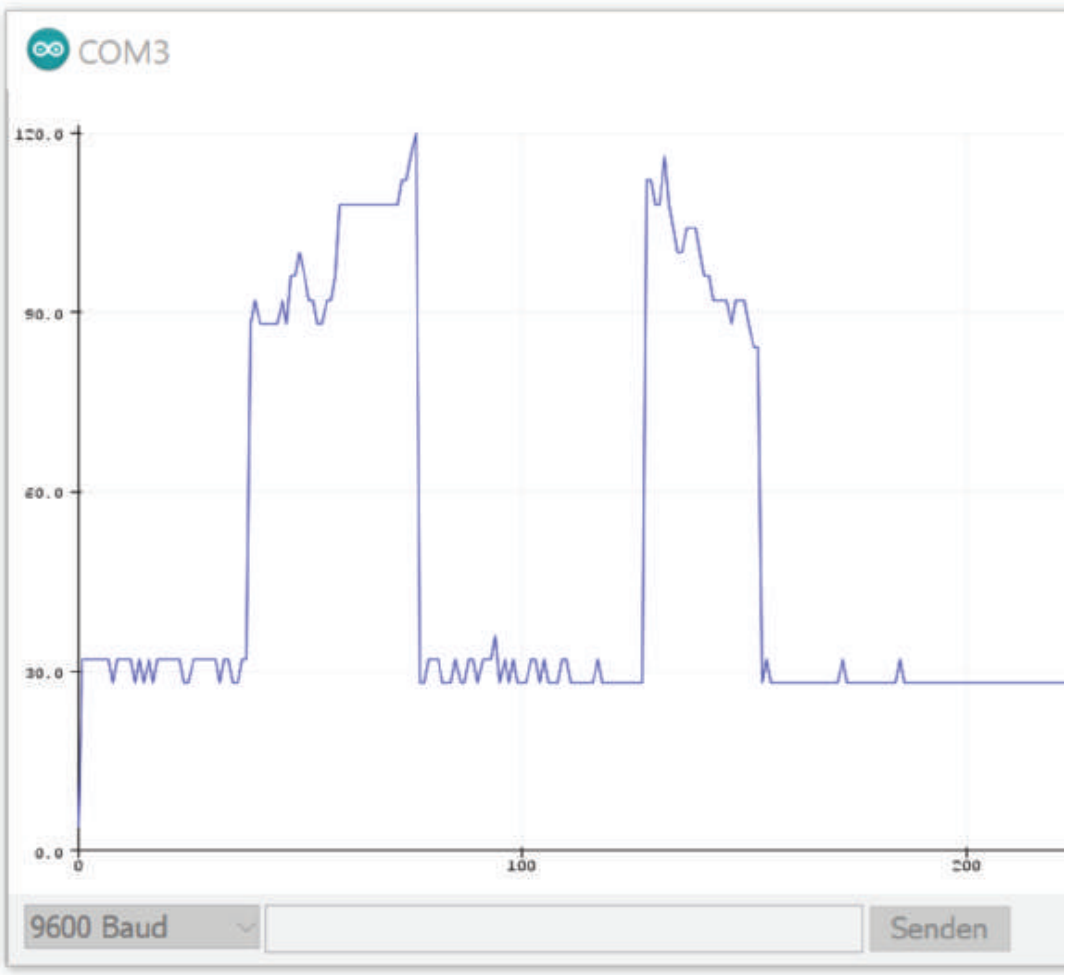

Abb. 3.4 Im seriellen Plotter sind die beiden Berührungen während des Testlaufs deutlich als Ausschlag zu erkennen.

Mit diesem Beispielaufbau liegt die gemessene Ladezeit ohne Kontakt bei etwa 30 Mikrosekunden. Bei Berührung steigt sie auf den dreifachen Wert, teils noch höher. Aufgrund dieser Auswertung wurde der Schwellenwert für die Aktivierung der Onboard-LED auf 50 gesetzt. Je nach verwendeten Bauelementen kann dieser Wert abweichen und

muss gegebenenfalls angepasst werden. Als Kontaktfläche kann anstatt des losen Drahtes auch eine beliebige Metallfläche genutzt werden – beispielsweise Alufolie oder eine Teelicht-Hülle. Für einen Ausschlag ist nicht einmal eine direkte Berührung mit der Haut notwendig – es funktioniert auch, wenn man die Kontaktfläche mit Papier abdeckt und einen Finger darauflegt.

Es sei erwähnt, dass der bei solchen Sensoren über die Haut fließende Ladestrom derart klein ist, dass jegliche gesundheitliche Gefahren ausgeschlossen werden können. Generell arbeiten wir bei allen Projekten mit einer Spannung von maximal 9 Volt, welche bei Berührung gesundheitlich völlig unbedenklich ist.

3.2 Materialien

Wir könnten nun unseren geplanten Ventilator also mit selbstgebauten Touch-Sensoren ausstatten. Um noch eine weitere Methode zu demonstrieren, greifen wir im Folgenden jedoch auf ein fertiges Modul mit vier Touch-Flächen zurück. Es verfügt über einen integrierten Schaltkreis, welcher auf die soeben kennengelernte Weise Berührungen erkennt.

Benötigt werden daher im Wesentlichen:

- 1 Gleichstrommotor mit Flügelrad
- 1 Touch-Modul mit 4 Berührflächen (TTP224)
- 1 Transistor NPN, z.B. BC548C
- 1 Vorwiderstand 1 kΩ
- 1 Silizium-Diode
- 1 Arduino Nano bzw. gleichwertiger Nachbau

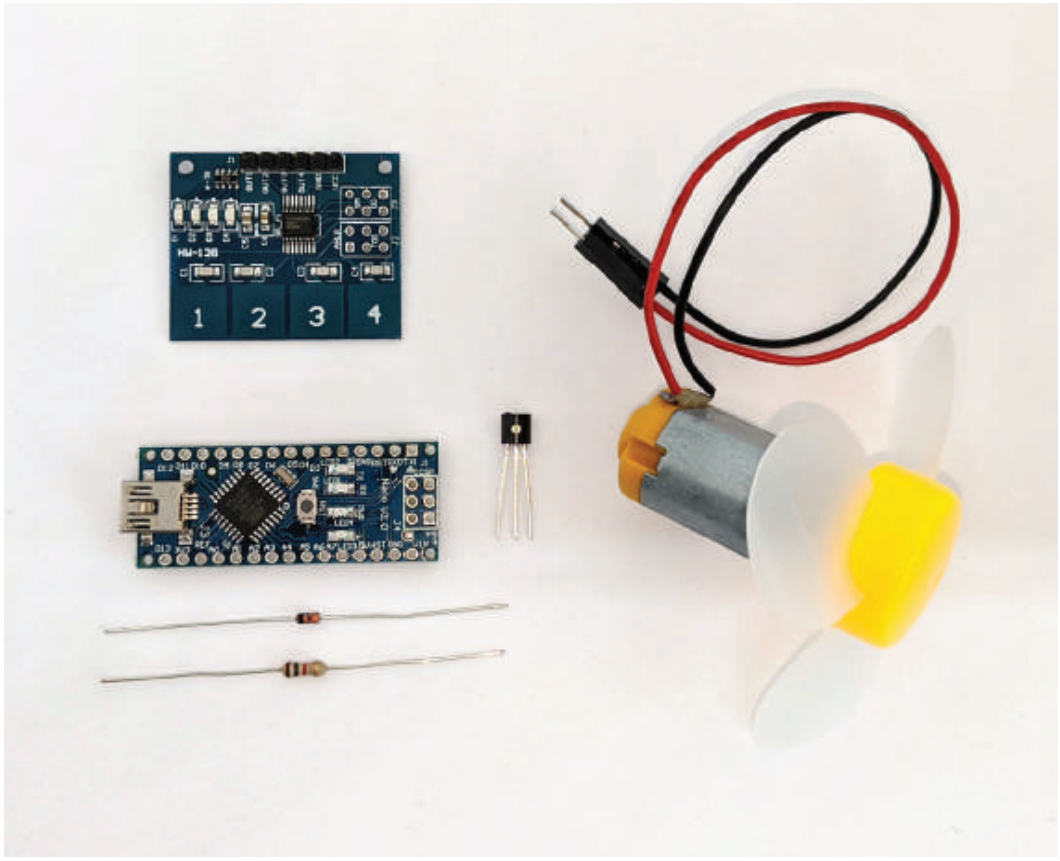

Abb. 3.5 Oben links befindet sich der Touch-Sensor mit den vier deutlich markierten Bedienflächen.

Außerdem kommt im Beispielaufbau noch Bastel-Equipment zum Einsatz:

- ca. 20 cm Aludraht (Durchmesser 3 mm)
- 1 Gehäuse/Schachtel als Standfuß

Die Bedienung soll intuitiv sein: Es gibt vier Bedienflächen – eine zum Ausschalten, eine zum Einschalten mit voller Geschwindigkeit und zwei zum Verringern bzw. Erhöhen der Geschwindigkeit.

Der Motor benötigt laut Datenblatt etwa 150 mA Strom. Somit kann er nicht direkt an einem Ausgangspin betrieben werden (Maximalstrom 20 mA). Stattdessen verwenden wir zur Ansteuerung einen Transistor, der wie ein elektronischer Schalter vom PWM-fähigen Ausgangspin D6 gesteuert wird.

3.3 Touch-Modul

Das genutzte Touch-Modul lässt sich mühelos anbinden. Es muss mit der Betriebsspannung versorgt werden und bietet für jede Berührfläche eine separate Signalleitung, welche standardmäßig auf 0 V (LOW) liegt. Wird eine Berührung erkannt, wird ein HIGH-Pegel ausgegeben. Die vier Signalleitungen können also direkt mit digitalen Arduino-Eingängen (ohne Pull-Up) verbunden werden. Im Handel gibt es diese Module in vielen Varianten – mit nur einer Bedienfläche oder sogar als Zahlenfeld mit 12 Elementen. Die Berührflächen selbst werden durch Leiterflächen auf der Platine gebildet. Sie können problemlos mit selbstgestalteten Beschriftungen beklebt werden, da selbst durch Papier (in normaler Stärke) hindurch die Berührungen zuverlässig erkannt werden. Auf der Platine befinden sich zudem vier LEDs, welche bei erkannter Berührung leuchten – dies macht den Funktionstest und die Fehlersuche besonders einfach:

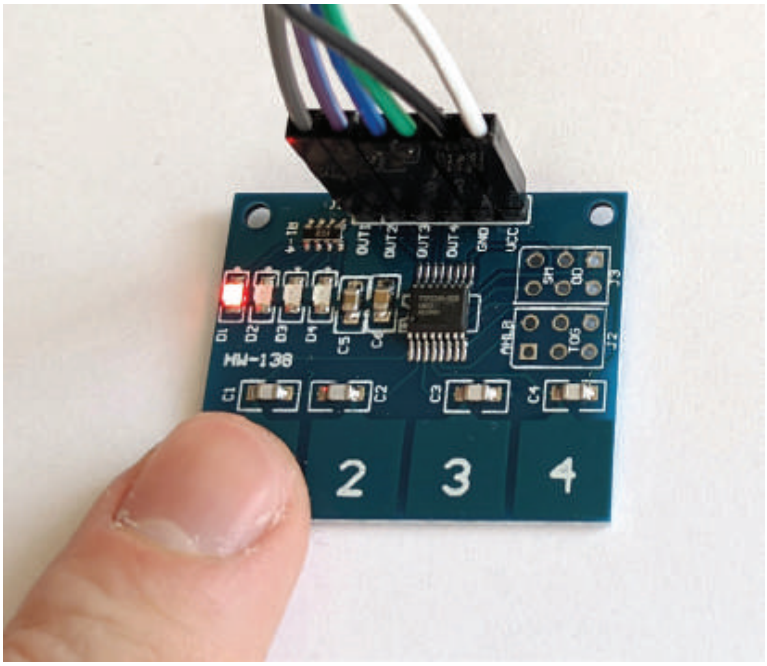

Abb. 3.6 Die Berührflächen sind bereits mit Schutzlack überzogen und können auch mit Papier beklebt werden.

Der Stromlaufplan ist daher leicht verständlich:

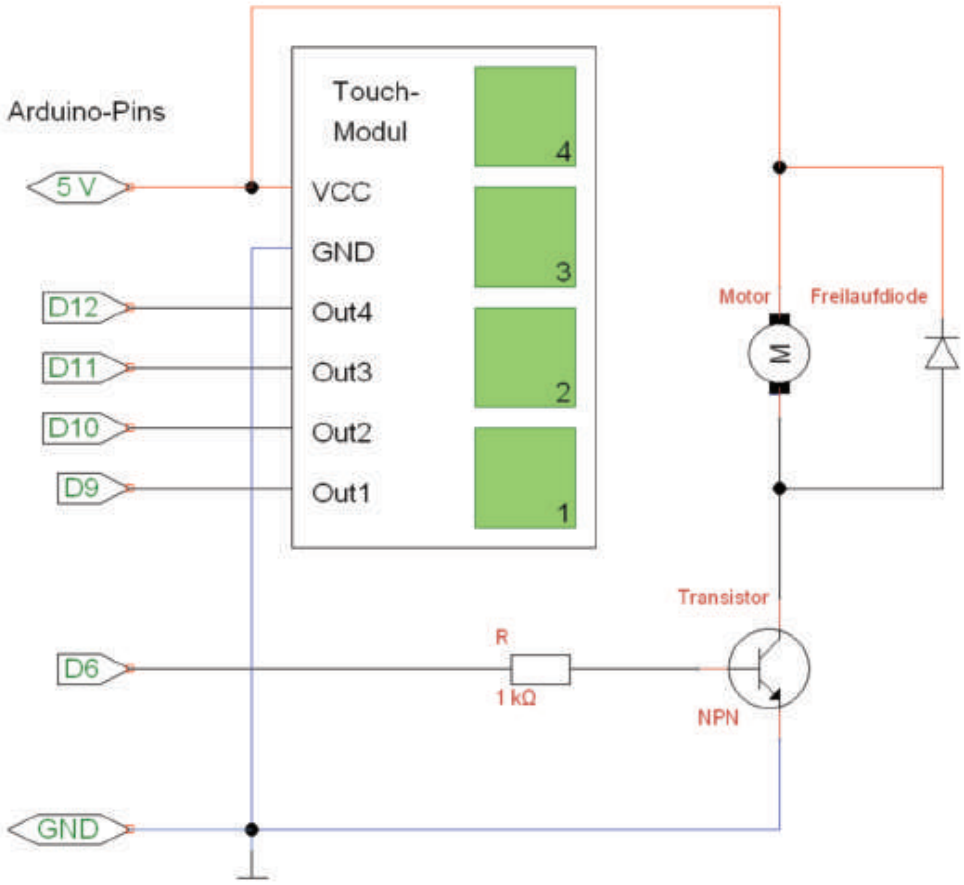

Abb. 3.7 Die Verbindung des Touch-Moduls kann direkt über 4 Eingangspins geschehen.

Die Ansteuerung des Motors per Transistor bietet den Vorteil, dass auch der für Arduino-Verhältnisse relativ hohe Strom problemlos per Pulsbreitenmodulation (PWM) gesteuert werden kann. Allerdings besteht ein Motor im Wesentlichen aus Spulen, welche eine Induktivität aufweisen. In dieser kann sich der Strom nicht sprunghaft ändern. Wenn der Transistor also aufgrund des PWM-Signals zu einem konkreten Zeitpunkt von voll-leitend auf voll-sperrend schaltet, kann der Strom durch den Motor nicht sofort zum Stillstand kommen – ähnlich wie ein Schwungrad, welches nicht von einem Moment auf den nächsten sofort zum Halt gebracht werden kann. Die antiparallel geschaltete Diode ermöglicht es diesem Strom, auch bei sperrendem Transistor noch weiterzulaufen und langsam abzuklingen – wie eine Freilaufkupplung an einer mechanischen Welle. Aus diesem Grund wird die Diode auch Freilaufdiode genannt. Würde man sie weglassen, sorgte die Indukti-

vität bei Schaltvorgängen für sehr hohe Spannungen am sperrenden Transistor, wodurch dieser zerstört werden könnte.

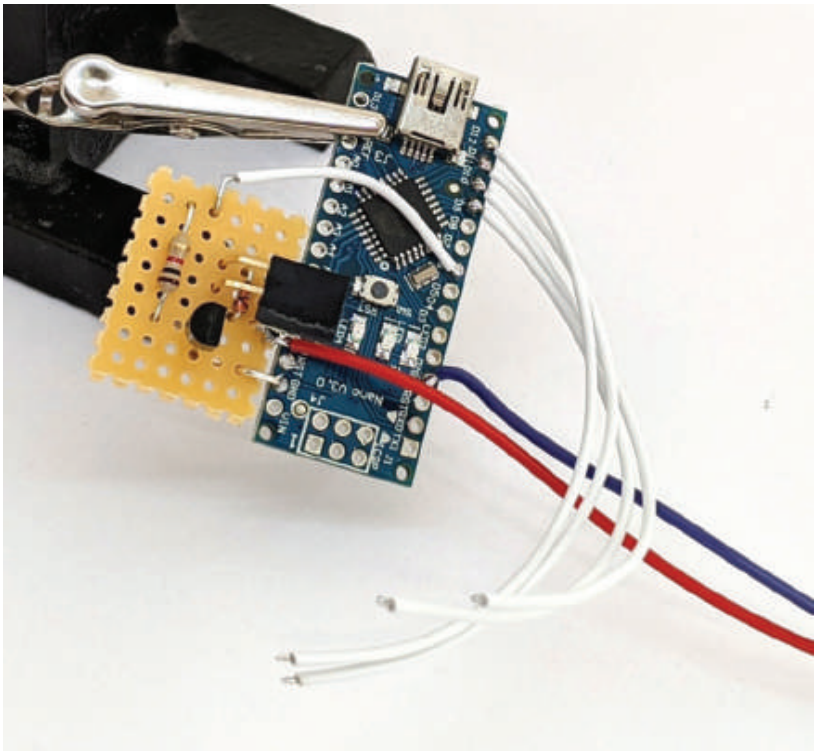

Abb. 3.8 Transistor, Vorwiderstand und Freilaufdiode wurden auf einer kleinen Hilfsplatine untergebracht. Für den Motoranschluss wurde die abgewinkelte Steckverbindung vorgesehen.

3.4 Sketch

Der Sketch ist weitestgehend selbsterklärend. Aufgrund der einfacheren Lesbarkeit wird im Folgenden statt der Portregister wieder auf die herkömmlichen Funktionen zum Portzugriff zurückgegriffen.

```
byte Geschwindigkeit = 0;

void setup() {
  pinMode(9, INPUT);          // 4 Eingangspins für das Touch-
                              // Modul
  pinMode(10, INPUT);         // (ohne Pull-Up, da das Modul
  pinMode(11, INPUT);         // jeden Pin aktiv auf LOW oder
  pinMode(12, INPUT);         // HIGH setzt)

  pinMode(6, OUTPUT);         // PWM-fähiger Pin als Ausgabe
```

```
}

void loop() {
  if(digitalRead(9))          // Ausschalten
    Geschwindigkeit = 0;

  if(digitalRead(12))         // Einschalten (volle
                              // Geschwindigkeit)
    Geschwindigkeit = 255;

  while(digitalRead(11) && Geschwindigkeit < 255)
  {                           // Geschwindigkeit erhöhen
    Geschwindigkeit++;
    analogWrite(6, Geschwindigkeit);
    delay(40);
  }

  while(digitalRead(10) && Geschwindigkeit > 0)
  {                           // Geschwindigkeit reduzieren
    Geschwindigkeit--;
    analogWrite(6, Geschwindigkeit);
    delay(40);
  }

  analogWrite(6, Geschwindigkeit);
  delay(10);
}
```

Die Byte-Variable `Geschwindigkeit` repräsentiert die aktuelle Soll-Geschwindigkeit, welche regelmäßig per `analogWrite()` am Pin D6 als PWM-Signal ausgegeben wird. Bei Berührung der vier Touch-Flächen wird der Motor ein- oder ausgeschaltet (D12 / D9) oder die Geschwindigkeit variiert, solang die Bedienfläche berührt wird (D11 / D10).

3.5 Aufbau

Bei der optischen Gestaltung des Ventilators ist neben handwerklichem Geschick auch Kreativität gefragt. Für die folgende Realisierung wurde der Motor mittels Aluminumdraht und Heißkleber an einem Standfuß befestigt, welcher von einer (stabilen) Papierschachtel gebildet wird. Diese Papierschachtel nimmt auch die komplette Elektronik auf. Sämtliche Verbindungen wurden gelötet. Die Berührflächen des Touch-Moduls wurden mit ausgedruckten Schaltflächen beklebt.

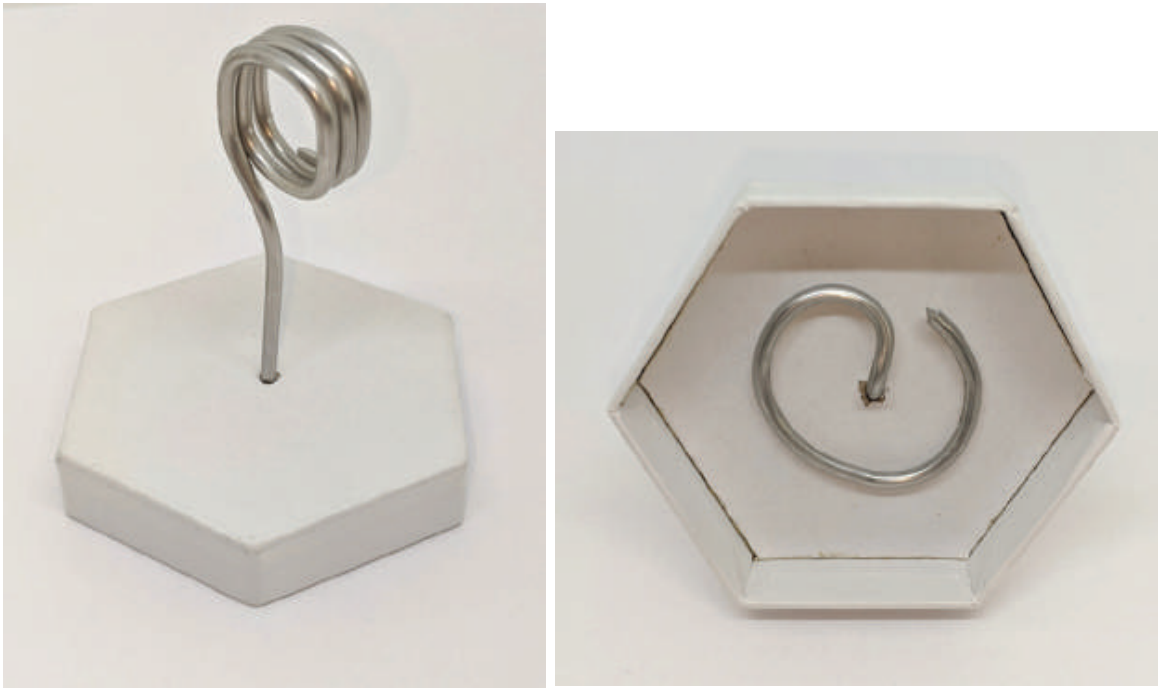

Abb. 3.9 Aludraht übernimmt die Fixierung des Motors. Der Draht kann im Deckel der Schachtel zu einem Standfuß geformt und mit Heißkleber befestigt werden.

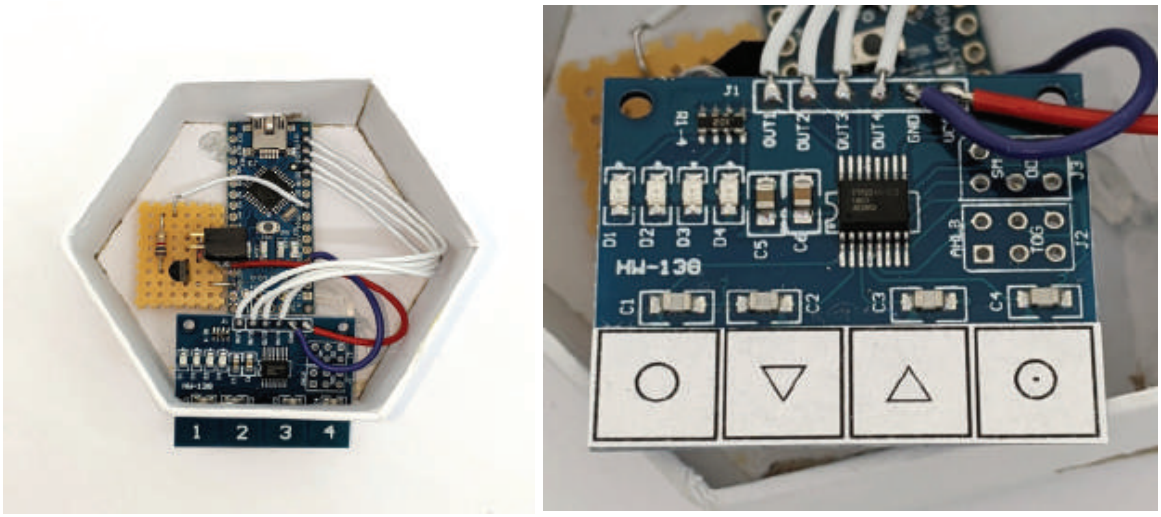

Abb. 3.10 Die komplette Elektronik passt in die als Fuß dienende Schachtel. Die Touch-Flächen können beliebig beklebt werden.

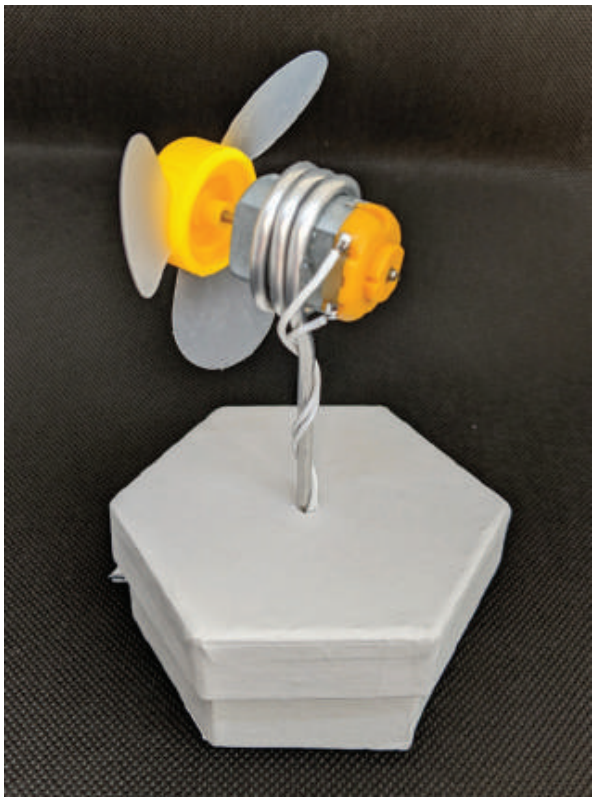

Abb. 3.11 Zur rutschfesten Montage des Motors hilft auch ein Tropfen Heißkleber.

Die Stromversorgung erfolgt bequem per USB direkt durch die rückseitig erreichbare USB-Buchse der Nano-Platine. Damit bringt dieser kleine Helfer auch im heißesten Sommer immer frischen Wind in den Bastelkeller!

Alle Programmcodes aus diesem Buch sind als PDF zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:
https://bmu-verlag.de/projekte-arduino-esp

Außerdem erhalten Sie die eBook-Ausgabe zum Buch im PDF-Format kostenlos auf unserer Website:

https://bmu-verlag.de/projekte-arduino-esp
Downloadcode: siehe Kapitel 28