Arduino verstehen und anwenden

Teil 10: Richtig auf unerwartete Ereignisse reagieren – Interrupts und Polling

Der zehnte Artikel der Beitragsreihe „Arduino verstehen und anwenden“ dreht sich um den Bereich „Interrupts und Polling“. Dabei stehen die folgenden Themen im Vordergrund:

- Was ist ein Interrupt und wozu wird er benötigt?
- Was ist der Unterschied zwischen Interrupts und „Polling“
- Welche Interruptquellen stehen auf einem Mikrocontroller zur Verfügung

Wie in dieser Serie üblich, wird wieder besonderer Wert auf praktische Anwendungen gelegt. Dazu werden die folgenden Praxisübungen durchgeführt:

- Tasterabfrage mit Polling
- LED-Steuerung mit Interrupts
- Wirkung verschiedener Interrupt-Modi
- „Multitasking“: simultanes Blinken und Schalten von LEDs
- Blinken ohne Delay!

Wozu benötigt man Interrupts?

Gerade in der Mikrocontroller-Technik ist es oftmals erforderlich, dass ein Programm auf unerwartete Ereignisse schnell und sicher reagiert. Ein klassisches Beispiel ist die Auslösung von Alarmen beim Vorliegen eines Fehlers. Hier ist eine schnelle und unverzügliche Reaktion erforderlich. Aber auch bei Eingaben über Tastaturen will der Anwender nicht warten, bis das laufende Programm zu irgendeinem Zeitpunkt in die Tastenabfrage springt, sondern jede Eingabe soll unverzüglich übernommen werden.

Zudem liefern Sensoren oftmals unvorhersehbare Werte. Soll z. B. beim Überschreiten einer bestimmten Temperatur ein Gerät abgeschaltet werden, so muss der Mikrocontroller (µC) unverzüglich reagieren.

Ist er gerade mit komplizierten Berechnungen ausgelastet, kann das betroffene Gerät bereits überlastet sein, bis der Controller einen bestimmten Eingang abfragt und die vorprogrammierte Aktion startet.

Prinzipiell könnte man einen Taster oder auch einen Sensorwert in regelmäßigen Abständen über

```
digitalRead(pushButton);
```

abfragen. Dieses Abfragen eines Schaltzustandes in regelmäßigen Abständen wird als „Polling“ (engl. für „Abfragen“) bezeichnet.

Wird der betreffende Taster allerdings gedrückt, während das Anwendungsprogramm mit anderen Aufgaben beschäftigt ist, bleibt der Tastendruck ohne Wirkung. Natürlich kann ein solches „Verschlucken“ von Eingabewerten in der Praxis nicht toleriert werden.

Auf rein softwaretechnischem Weg ist das Problem nicht zu beheben. Hier ist eine zusätzliche controllerinterne Hardware-Schaltung erforderlich, die kontinuierlich und ohne Unterbrechung einen Schaltpegel an einem bestimmten Pin überwachen kann.

Da dieses Problem sehr häufig auftritt, wurde dafür eine spezielle Lösung implementiert, die sogenannte Interruptsteuerung. Interrupts unterbrechen unmittelbar das Hauptprogramm und springen in eine sogenannte Interruptroutine. Erst nach dem Abarbeiten dieser speziellen Befehlsfolge erfolgt wieder ein Rücksprung in das Hauptprogramm.

Damit kann z. B. ein Tastendruck zu keinem Zeitpunkt mehr verloren gehen. Bild 1 veranschaulicht diesen Vorgang.

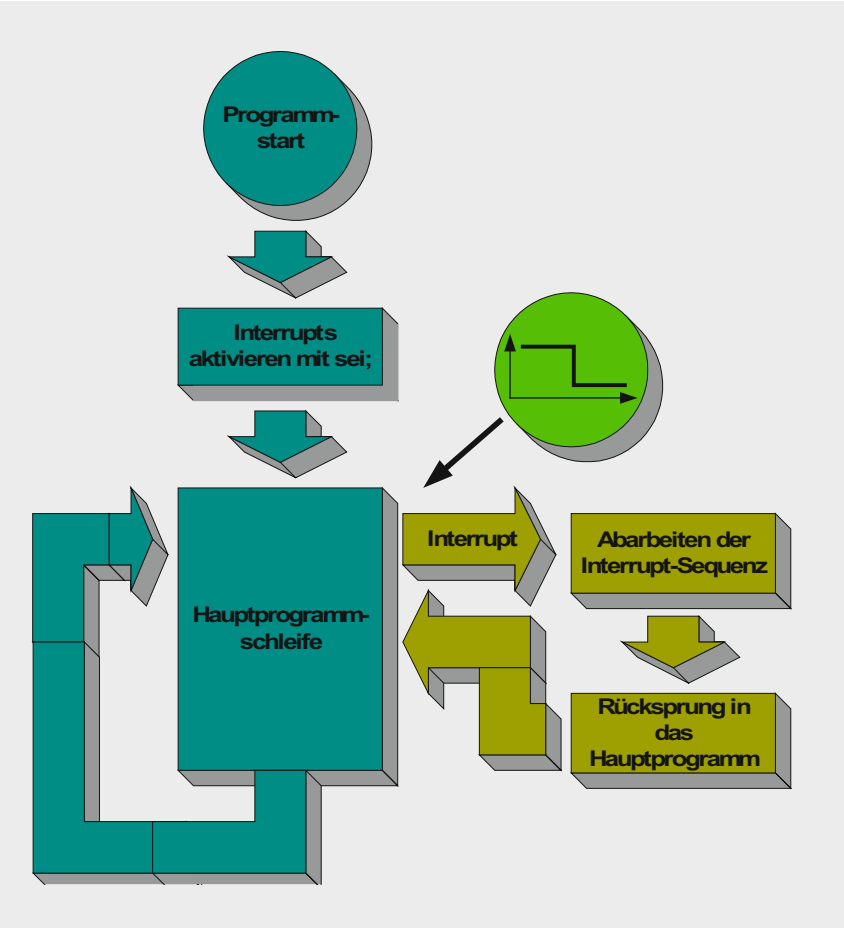

Bild 1: Programmablauf mit Interruptaufruf

Interruptquellen

Mikrocontroller besitzen meist mehrere Interruptquellen. Dadurch kann ein µC auf verschiedene unvorhergesehene Ereignisse reagieren.

Einzelne Interrupts werden über sogenannte Interruptvektoren angesprochen. Diese Vektoren sorgen dafür, dass die Ausführung immer an der korrekten Adresse startet. Deshalb muss man sich beim Programmieren mit Processing nicht um spezielle Adressierungen kümmern. Es ist vollkommen ausreichend, wenn man die richtigen Interruptvektoren kennt. Die wichtigsten Interruptquellen sind:

- Reset
- Externer Interrupt 0
- Externer Interrupt 1
- Timer-/Counter-Ereignisse
- UART-Zeichen empfangen

Für sogenannte externe Interrupts steht die Funktion

attachInterrupt(interrupt, function, mode)

zur Verfügung.

Auf dem Arduino stehen zwei Pins für Interruptzwecke bereit:

1. Für Interrupt0 der Digital Pin 2
2. Für Interrupt1 der Digital Pin 3

Die Parameter der Funktion attachInterrupt haben dabei die folgenden Bedeutungen:

interrupt: Nummer des Interrupts (0 oder 1)

function: Diese Funktion wird aufgerufen, wenn der Interrupt ausgelöst wurde, sie wird auch als Interrupt-Service-Routine (ISR) bezeichnet

mode: Legt den Auslösemodus fest

Für die Funktion „mode" sind 4 Varianten vordefiniert:

LOW: Auslösung, sobald der Interrupt-Pin auf LOW geht

CHANGE: Interrupt wird bei jedem Signalwechsel am Interrupt-Pin ausgelöst

RISING: Übergänge von LOW nach HIGH lösen den Interrupt aus

FALLING: Übergänge von HIGH nach LOW lösen den Interrupt aus

Der Programmcode innerhalb einer ISR sollte keine größeren Verzögerungen enthalten. Trifft nämlich ein neuer Interrupt ein, während der alte noch nicht abgeschlossen ist, so kann dieser nicht korrekt abgearbeitet werden. Zwar ist es prinzipiell möglich, mehrere Interruptebenen zuzulassen, jedoch sollte diese Technik aufgrund ihrer Fehlerträchtigkeit nur angewendet werden, wenn es keine anderen Möglichkeiten mehr gibt.

TIPP:
Interrupt-Service-Routinen sollten immer möglichst kurz und einfach gehalten werden!

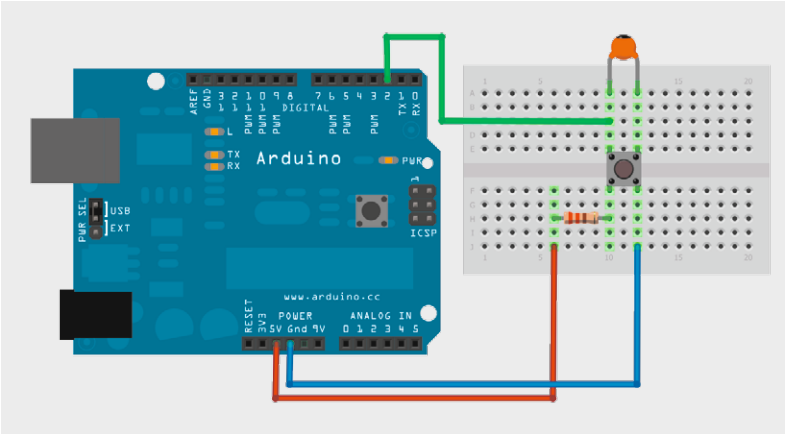

Bild 2: Aufbau zur Interruptsteuerung

Das folgende Programm zeigt, wie eine LED über einen Interrupt gesteuert werden kann:

```
// LED control via interrupt

int ledpin = 13;
volatile int state = LOW;

void blink()    // Change state of LED
{ state = !state;
  digitalWrite(ledpin, state);
}

void setup()
{ pinMode(ledpin, OUTPUT);
  attachInterrupt(0, blink, RISING);
}

void loop()
{
}
```

Die LED 13 kann nun über den Drucktaster ein- und ausgeschaltet werden.

Das Programm weist eine Besonderheit auf: Die Hauptprogrammschleife ist völlig leer!

Dies demonstriert, dass die Interruptaufrufe absolut unabhängig von einem Hauptprogramm erfolgen können. Wenn die Interruptbehandlung mit attachInterrupt erst einmal initialisiert ist, werden die Interrupt-Pins völlig unabhängig von anderen Programmteilen überwacht.

Den zugehörigen Aufbau zeigt Bild 2. Der 100-nF-Kondensator parallel zum Taster reduziert das sogenannte „Tasterprellen". Wird der Kondensator entfernt, dann kann es bei einem einzelnen Tastendruck zu mehreren Umschaltvorgängen kommen. Weitere Details zu diesem meist unerwünschten Effekt werden in einem späteren Beitrag zu dieser Artikelserie behandelt.

Wirkung der verschiedenen Interrupt-Modi

Ersetzen Sie den Interrupt-Modus „RISING" nacheinander durch die anderen drei Betriebsarten:

- LOW
- CHANGE
- FALLING

und beobachten Sie, wann genau die LED jeweils geschaltet wird.

Der nächste Sketch demonstriert ein einfaches „Multitasking". Während im Hauptprogramm eine LED kontinuierlich blinkt, kann eine weitere LED unabhängig davon ein- und ausgeschaltet werden. Die zweite LED muss dazu an Port 12 angeschlossen werden (über einen geeigneten Vorwiderstand).

```
// LED control via interrupt
// Blink a LED and simultaneously switch
// another one
// Interrupt 0 on DigitalPin 2

int led1pin = 13;
int led2pin = 12;
volatile int state = LOW; // Status LED_1

void switch_led1() // Change state of LED
{ state = !state;
  digitalWrite(led1pin, state);
}

void setup()
{ pinMode(led1pin, OUTPUT);
  pinMode(led2pin, OUTPUT);
  attachInterrupt(0, switch_led1, RISING);
}

void loop()
{ digitalWrite(led2pin, HIGH);  // LED on
  delay(100);
  digitalWrite(led2pin, LOW);   // LED off
  delay(100);
}
```

„Flüchtige" Variablen

Eventuell ist Ihnen in den beiden obenstehenden Programmen der Zusatz „volatile" bei der Variablendeklaration bereits aufgefallen. „Volatile" bedeutet „veränderlich, flüchtig".

Nun sind Variablen naturgemäß veränderlich, was ist also das Besondere an einer als „volatile" definierten Variablen?

Wird eine Variable nur innerhalb einer Interruptroutine verändert, so ist diese Veränderung im Hauptprogramm nicht sichtbar.

Je nach Compilereinstellung wäre es bei der Übersetzung des Quellcodes möglich, dass die Variable durch eine Speicherplatz sparende Konstante ersetzt wird. Im Normalfall würde dies zu einem kompakteren und schnelleren Code führen.

Ist aber eine Interruptroutine vorhanden, so muss dies ausgeschlossen werden, damit das Programm korrekt arbeitet. Der Zusatz „volatile" bei einer Variablendefinition sorgt dafür, dass immer eine „echte" Variable erzeugt wird, auch wenn diese scheinbar im Hauptprogramm gar nicht verändert wird.

Das Schlüsselwort „volatile" teilt dem Compiler also mit, dass die damit deklarierte Variable durch Ereignisse außerhalb der Kontrolle des Hauptprogramms verändert werden kann.

Praxisübung: Blinken ohne Delay!

In diesem Beispiel wird eine weitere Interruptquelle vorgestellt. Nicht nur externe Ereignisse, wie der Pegelwechsel an einem Pin, können Interrupts auslösen, sondern auch controllerinterne Auslösungen sind möglich.

Das folgende Beispiel zeigt, wie eine LED zum Blinken gebracht werden kann, ohne dass die Delay-Funktion zum Einsatz kommt.

```
// Blink without using delay-function

#include "TimerOne.h"   // include timer lib
int ledpin = 13;
double tc = 1000000;
// timeCounter = 1000000 µs = 1 s
volatile int state = LOW;

void setup()
{ pinMode(ledpin, OUTPUT);
  Timer1.initialize(tc);
  Timer1.attachInterrupt(switch_led);
}

void switch_led() // Change state of LED
{ state = !state;
  digitalWrite(ledpin, state);
}

void loop()
{ // main loop empty!
}
```

Der Interrupt wird hier durch einen µC-internen Timer ausgelöst. Durch den Befehl

```
Timer1.attachInterrupt(switch_led);
```

wird festgelegt, dass nach Ablauf des Timers die Interruptroutine „switch_led" aufgerufen wird. Diese Routine sorgt dann wie im vorhergehenden Beispiel dafür, dass die LED ein- bzw. ausgeschaltet wird. Dies hat den großen Vorteil, dass der Controller auf diese Weise nicht blockiert wird. Wird die „delay()"-Funktion verwendet, kann der Arduino in dieser Verzögerungsphase keine anderen Aufgaben ausführen. Er ist praktisch vollständig mit „Warten" ausgelastet. In einem einfachen Blink-Programm spielt das keine große Rolle. Bei komplexeren Anwendungen möchte man den Controller aber selbstverständlich nicht mit Wartebefehlen auslasten.

Man erkennt, dass auch hier die Hauptschleife vollkommen leer ist. Diese Programmebene kann also weiterhin unabhängig genutzt werden.

In dieser Anwendung kommt auch bereits ein Counter zum Einsatz. Die Verwendung dieser nützlichen controllerinternen Funktionseinheiten wird im nächsten Artikel dieser Reihe näher erläutert werden.

Die für diesen Sketch erforderliche Bibliothek TimerOne kann unter

http://code.google.com/p/arduino-timerone/downloads/list

kostenlos aus dem Internet geladen werden. Die Details zur Installation von Bibliotheken wurden bereits im Arduino-Artikel Teil 5 „Nutzung und Erstellung von Programmbibliotheken" erläutert (ELV-journal 4/2014).

Praxisprojekt: Fahrradrücklicht mit Interruptsteuerung

Zum Abschluss soll hier noch das Modell eines Fahrradrücklichts mit verschiedenen Betriebsmodi vorgestellt

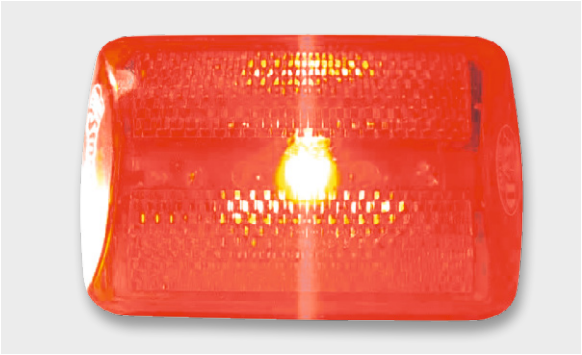

Bild 3: Fahrradrücklicht mit verschiedenen Betriebsmodi

werden. Moderne akkubetriebene Fahrradrücklichter verfügen über mehrere LEDs. Diese können meist in verschiedene Betriebsmodi geschaltet werden.

Neben einem einfachen Dauerlicht sind auch unterschiedliche Blink- und Lauflichteffekte aktivierbar. Bild 3 zeigt ein solches Rücklicht in Aktion.

Das Modell eines solchen Rücklichts eignet sich bestens als Demonstrationsbeispiel für den Einsatz von Interrupttechniken.

Bild 4 zeigt einen Aufbauvorschlag für die Arduino-Hardware dazu.

Der Wert für den eingezeichneten Kondensator kann zwischen 100 nF und ca. 10 µF liegen. Der Kondensator hat hier wieder die Aufgabe, das Tastenprellen zu reduzieren. Den optimalen Wert für den Kondensator bestimmt man am besten experimentell. So können bei bestimmten Tastern bereits mit Werten im Nanofarad-Bereich gute Ergebnisse erzielt werden, während stark prellende Taster Kondensatoren von bis zu einigen Mikrofarad erfordern.

Das zugehörige Programm „Bike-Light" zeigt, wie ein passender Sketch dazu aussehen kann. Hierbei wurde zunächst auf den Einsatz von Interrupts verzichtet. Stattdessen wird am Anfang der Endlosschleife void loop() der aktuelle Tasterstatus über checkSwitch abgefragt (sogenanntes „Polling"). Stellt das Programm fest, dass der Taster gedrückt ist, wird die Variable „mode" zyklisch in der Form

0 -> 1 -> 2 -> 3 -> 0

weitergeschaltet. Entsprechend werden die Betriebsmodi:

- Alle LEDs aus
- Alle LEDs an
- Alle LEDs blinken
- Lauflichteffekt

aktiviert.

Soweit funktioniert das Fahrradrücklicht ganz zufriedenstellend. Allerdings stellt man bei genauerer Betrachtung fest, dass das Weiterschalten der Betriebszustände nicht ganz optimal abläuft. Insbesondere wenn der Modus „Lauflicht" aktiv ist, kommt es vor, dass ein Tastendruck „verschluckt" wird. Die Ursache für dieses Verhalten liegt darin, dass während des Ablaufs der Funktion „Lauflicht" keine Tasterabfrage erfolgt. Erst wenn der Programmablauf wieder den Anfang der Hauptschleife erreicht hat, kann der Tastendruck erkannt werden. Ein solches Verhalten ist in der Praxis natürlich nicht erwünscht.

```
// Bike-Light with polling

byte ledPin[5] = {8, 9, 10, 11, 12};    // LEDs on pins 8 to 12
int switchPin = 2;                       // switch on pin 2
int checkSwitch, switchState=0;          // variables for switch control
int mode = 0;                            // display mode

void setup()
{ for (int n=0; n<5; n++) { pinMode(ledPin[n], OUTPUT); }
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);            // activate internal pull-up
}

void loop()
{ checkSwitch = digitalRead(switchPin);    // check switch
  if (checkSwitch != switchState)       // switch status changed!
    { if (checkSwitch == LOW)
      { mode++; if (mode > 3) mode=0;
      }
    }
    switchState = checkSwitch;          // save switch status

  switch (mode)
  { case 0:
    { // all LEDs off
    for (int n=0; n<5; n++) { digitalWrite(ledPin[n], LOW); }
    } break;

    case 1:
    { // all LEDs on
    for (int n=0; n<5; n++) { digitalWrite(ledPin[n], HIGH); }
    } break;

    case 2:
    { // blink
      for (int n=0; n<5; n++) { digitalWrite(ledPin[n], HIGH); }delay(100);
      for (int n=0; n<5; n++) { digitalWrite(ledPin[n], LOW); }delay(100);
    } break;

    case 3:
    { // chaser
      for (int n=0; n<5; n++)
        { digitalWrite(ledPin[n], HIGH); delay(100);
          digitalWrite(ledPin[n], LOW);
        }
    } break;
  }
}
```

Abhilfe kann hier der Einsatz der Interruptfunktionalität des Controllers schaffen.

Der nächste Sketch zeigt, wie die Interrupttechnologie hier nutzbringend eingesetzt werden kann. Über die Programmzeile

```
attachInterrupt(0, switch_update, RISING);
```

wird hier wieder ein Interrupt aktiviert.

Damit wird, immer wenn der Taster betätigt wird, ein Interrupt ausgelöst. In der Interruptroutine

```
void switch_update()                    // Change mode
{ …
   mode++; if (mode > 3) mode=0;
  …
}
```

wird unverzüglich die Variable „mode“ um eins erhöht bzw. auf null zurückgesetzt. Um das Tasterprellen zu reduzieren, wurden zusätzlich während des Routineablaufs neue Interrupts deaktiviert.

Zusätzlich wurde in der Chaser-Schleife eine „mode“-Abfrage eingefügt. Nun reagiert das Programm wie gewünscht verzögerungsfrei auf jeden Tastendruck. Das „Verschlucken“ eines Tastendrucks ist jetzt ausgeschlossen.

Falls dagegen gelegentlich ein Modus übersprungen wird, muss der Kondensatorwert angepasst werden. Durch Verwendung der Interrupttechnik und nach Optimierung des Entprellkondensators kann ein sehr bedienerfreundliches Schaltverhalten erreicht werden.

Wie bereits weiter oben erwähnt, wird das Thema Entprellen und sicheres Schalten in einem späteren Beitrag zu dieser Serie nochmals eingehend diskutiert werden.

```
// Bike-Light with interrupt control

byte ledPin[5] = {8, 9, 10, 11, 12};    // LEDs on pins 8 to 12
int switchPin = 2;                      // switch on pin 2
volatile int mode = 1;                  // display mode

void switch_update()                    // Change mode
{ noInterrupts();
  mode++; if (mode > 3) mode=0;
  delay(100);
  interrupts();
}

void setup()
{ for (int n=0; n<5; n++) { pinMode(ledPin[n], OUTPUT); }
  pinMode(switchPin, INPUT);
  digitalWrite(switchPin, HIGH);        // activate internal pull-up
  attachInterrupt(0, switch_update, FALLING);
}

void loop()
{ while (mode == 1)  // all LEDs on
    { for (int n=0; n<5; n++)  digitalWrite(ledPin[n], HIGH);
    }

  while (mode == 2)  // blink
    { for (int n=0; n<5; n++) { digitalWrite(ledPin[n], HIGH); } delay(100);
      for (int n=0; n<5; n++) { digitalWrite(ledPin[n], LOW); } delay(100);
    }

  while (mode == 3)  // chaser
    { for (int n=0; n<5; n++)
        { digitalWrite(ledPin[n], HIGH); delay(100);
          digitalWrite(ledPin[n], LOW);
          if (mode != 3) break;
        }
    }

  while (mode == 0)  // all LEDs off
   {for (int n=0; n<5; n++) digitalWrite(ledPin[n], LOW); }
}
```

Ausblick

Nachdem in diesem Kapitel das Thema „Interrupts und Polling“ ausführlich behandelt wurde, wird im nächsten Beitrag der Einsatz von Timern und Countern erläutert.

Diese Technologien basieren zu einem großen Teil ebenfalls auf der Erzeugung von Interrupts. Wird etwa über einen controllerinternen Hardwarezähler in regelmäßigen Zeitabständen ein Interrupt ausgelöst,

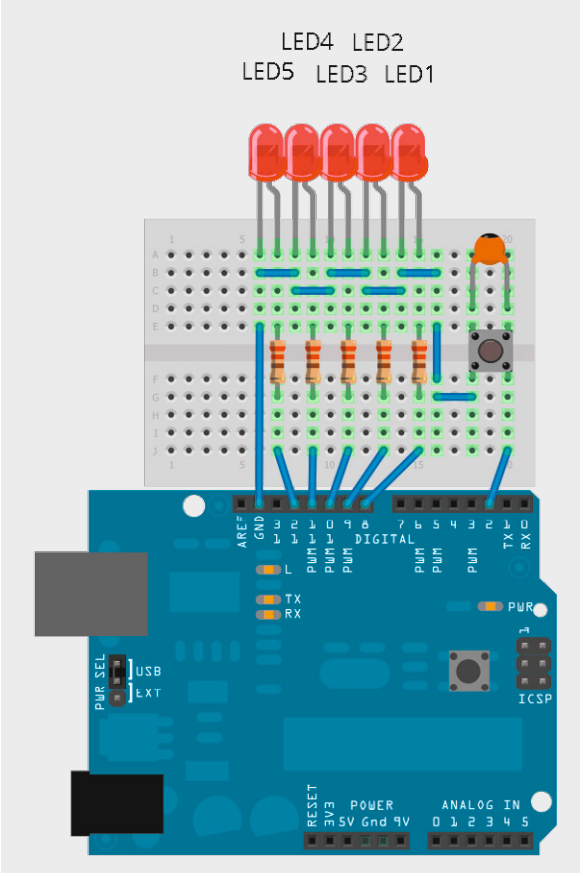

Bild 4: Arduino-basiertes Fahrradrücklicht mit verschiedenen Betriebsmodi

erhält man einen präzisen Timer, der den Aufbau von hochgenauen Frequenzzählern, Stoppuhren oder Digitalchronometern erlaubt.

Auch im nächsten Beitrag werden dazu, wie gewohnt, neben den theoretischen Grundlagen wieder viele Praxisprojekte vorgestellt. ELV

Weitere Infos:

- G. Spanner: Arduino – Schaltungsprojekte für Profis, Elektor-Verlag 2012, Best.-Nr. J9-10 94 45, € 39,80
- Grundlagen zur elektronischen Schaltungstechnik finden sich in der E-Book-Reihe „Elektronik!“ (www.amazon.de/dp/B000XNCB02)
- Buch „AVR-Mikrocontroller in C programmieren“, Franzis-Verlag 2012, Best.-Nr. J9-09 73 52, € 39,95

Preisstellung April 2015 – aktuelle Preise im Web-Shop

Empfohlene Produkte/ Bauteile:	Best.-Nr.	Preis
Arduino UNO	J9-10 29 70	€ 27,95
Mikrocontroller Online-Kurs	J9-10 20 44	€ 99,–

Alle Arduino-Produkte wie Mikrocontroller-Platinen, Shields, Fachbücher und Zubehör finden Sie unter: www.arduino.elv.de