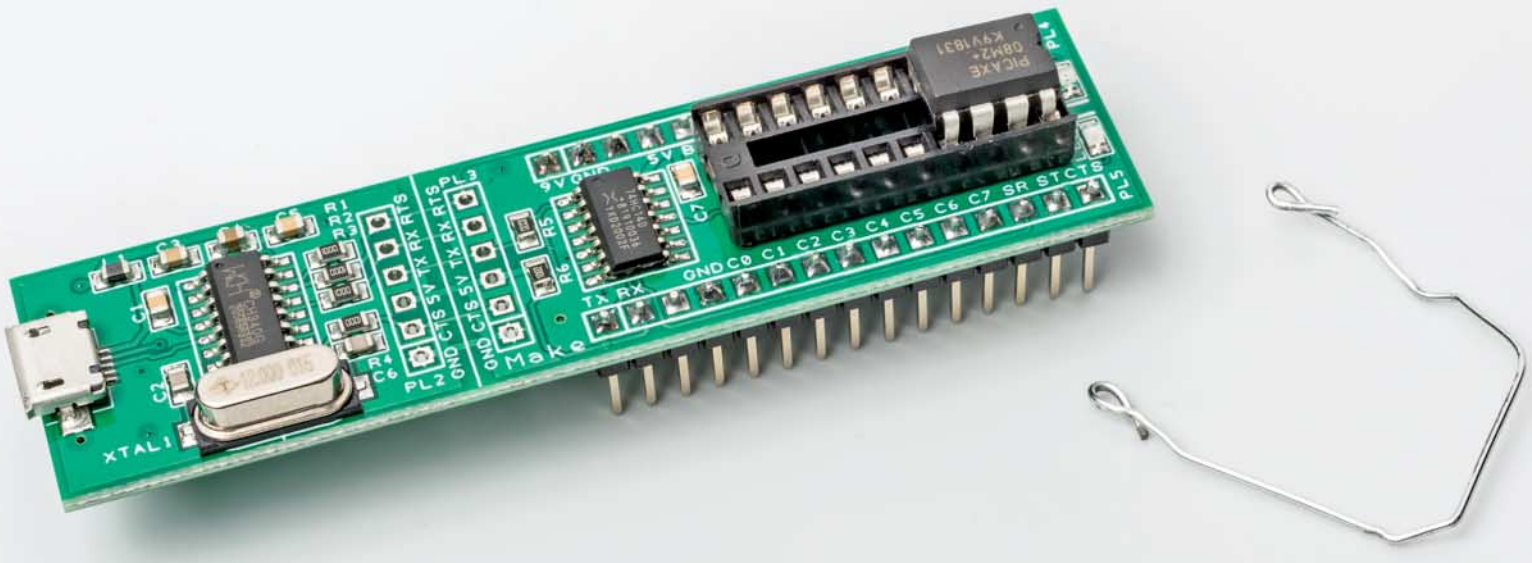
Einstieg mit LED

Mit einer einzigen LED kann man schon viel anstellen. In unseren ersten Experimenten lassen wir sie aufleuchten und lernen dabei mehr über unsere Pins, das Programmieren und wie Signale weitergegeben werden.

von Helga Hansen



Alles zum Artikel im Web unter *make-magazin.de/x8c3*

EINSTIEG MIT LED

Die LED blinkt

Zum Einstieg lassen wir die LED auf dem Nano-Axe-Board blinken

Für unser erstes Projekt benötigen wir das Nano-Axe-Bord, den Picaxe-Chip und ein USB-Kabel sowie einen Rechner mit der Picaxe-Software zur Programmierung. Dann gilt es, mit einem einfachen Programm zu überprüfen, ob tatsächlich alles funktioniert. Dafür lässt man beim Basteln mit Hardware meist eine LED aufleuchten – auf dem Nano-Axe-Board ist daher extra eine LED verbaut.

Mit dieser Übung lassen sich mehrere Fliegen mit einer Klappe schlagen. Blinkt am Ende die LED auf, sind alle Komponenten in Ordnung, die Kommunikation zwischen Rechner und Chip funktioniert und im Programm wurden alle Befehle richtig eingegeben. Außerdem hat man das Grundprinzip der Programmierung gemeistert und kann beginnen, eigene Ideen auszuprobieren. Bleibt die LED dagegen dunkel, muss man sich auf die Suche nach der Ursache begeben. Und auch wenn das frustrierend erscheint, lernt man beim Überprüfen möglicher Fehlerquellen ebenfalls sehr viel.

Auf der Softwareseite brauchen wir nur wenige Zeilen Code, um die LED leuchten zu lassen. Auch auf der Hardwareseite sind keine weiteren Bauteile und Schaltungen nötig, da wir eine LED auf dem Nano-Axe-Board nutzen. Die Abkürzung LED steht übrigens für *light emitting diode*, auf Deutsch Leuchtdiode. Dioden sind elektronische Bauteile, die Strom nur in eine Richtung passieren lassen. In die andere Richtung ist der Stromfluss gesperrt, was sich in vielen Schaltungen ausnutzen lässt. Die Besonderheit von Leuchtdioden ist, dass sie aufleuchten, wenn Strom durch sie fließt. Je nachdem, welches Material in der LED genutzt wird, kann die Farbe des entstehenden Lichts unterschiedlich sein. Auf dem Nano-Axe-Board haben wir eine blaue LED, die frei ansteuerbar ist, sowie eine orange Status-LED.

An und aus

Die LEDs leuchten, solange Strom fließt. Da die Status-LED nicht ansteuerbar ist, leuchtet sie immer, wenn der Mikrocontroller mit dem Rechner oder einer Batterie verbunden ist. Wann und wie die blaue LED leuchtet, können wir dagegen ändern. Dazu müssen wir „es fließt Strom“ mit „es fließt kein Strom“ abwechseln. Im Programmcode können wir dies mit zwei Zuständen angeben: High (Strom fließt) und Low (kein Strom).

Bisher haben wir immer über Strom gesprochen, der durch unsere Dioden fließt. Meistens verändert man aber die Spannung, um die Zustände festzulegen. Low bedeutet keine Spannung (0 Volt) und High eine höhere Spannung (etwa 5 Volt). Die Grafik zeigt das Grundprinzip von digitalen Signalen: Es wechselt zwischen 0 und 5 Volt und bleibt immer eine bestimmte Zeit in einem der beiden Zustände. Wenn das Signal eine LED steuert, leuchtet sie bei 5 Volt auf und geht bei 0 Volt wieder aus.

Das Nano-Axe-Board bringt hier allerdings eine Besonderheit mit. Die LED ist mit einem Bein mit der Spannungsversorgung VCC verbunden und funktioniert daher als digitaler Ausgang quasi verkehrt herum. Durch den Anschluss an VCC liegt an der LED grundsätzlich Spannung an – selbst wenn der Mikrocontroller ein Low-Signal liefert. Statt die LED auszuschalten, schaltet Low diese LED an. Beim Programmieren müssen wir dies berücksichtigen.

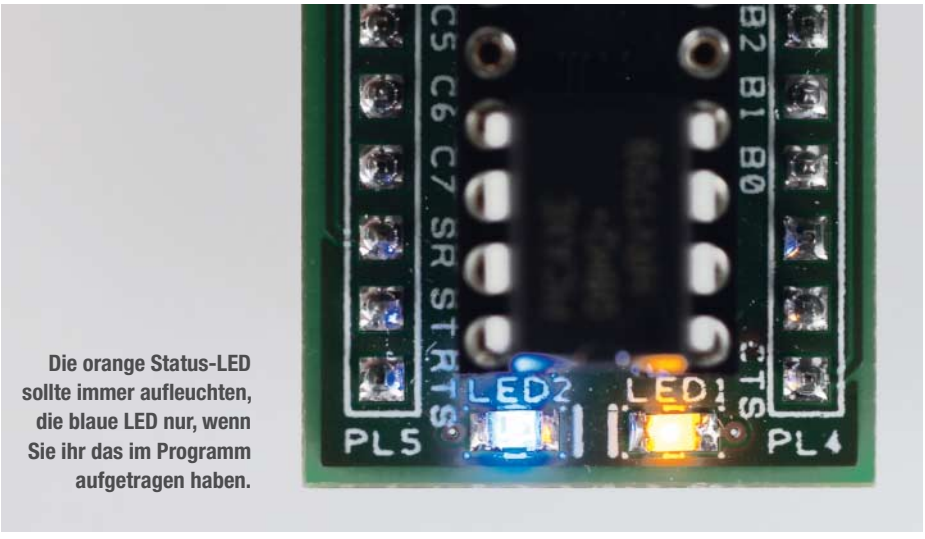
Das erste Programm

Unser BASIC-Programm ist nur sechs Zeilen lang. Es wird grundsätzlich von oben nach unten abgearbeitet und wiederholt sich unendlich lange – zumindest solange der Picaxe-Chip mit Strom versorgt wird. In der ersten Zeile vergeben wir mit `main:` ein sogenanntes Label (manchmal auch als Sprungmarke bezeichnet), das den Beginn unseres Programms markiert. Anschließend wechseln sich die Befehle für High und Low ab, damit die LED blinkt. In der letzten Zeile verweisen wir mit `goto main` wieder auf das Label am Anfang, damit das Programm von vorn beginnt (oder dorthin springt). So blinkt es nicht nur einmal, sondern dauerhaft.

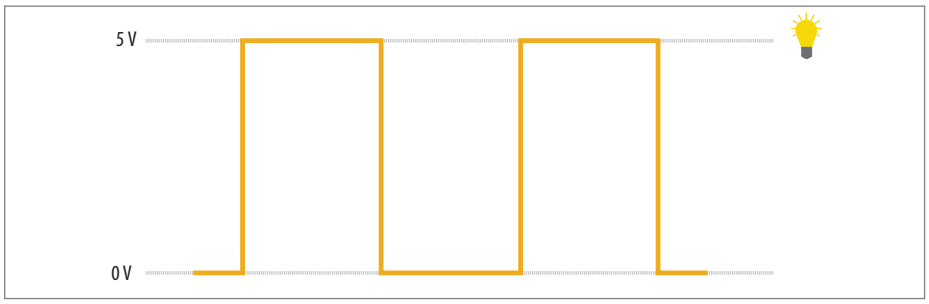
blink.bas

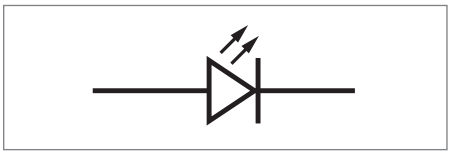
```
1 main:
2       low C.2
3       pause 1000
4       high C.2
5       pause 1000
6 goto main
```

Schauen wir uns die Befehle etwas genauer an. Das Label zu Beginn des Programms markieren wir mit einem Doppelpunkt. Unseres heißt `main:`, was Englisch für „Haupt-“ ist und darauf hinweist, dass nun unsere Hauptaufgabe folgt. Prinzipiell kann aber fast jedes Wort als Label genutzt werden. Außerdem können auch Ziffern und der Unterstrich _ darin auftauchen. Label in einem Computerprogramm haben die gleiche Aufgabe wie Label auf Marmeladengläsern oder Getränkeflaschen – sie zeigen möglichst auf einen Blick an, was drinsteckt.



Die orange Status-LED sollte immer aufleuchten, die blaue LED nur, wenn Sie ihr das im Programm aufgetragen haben.

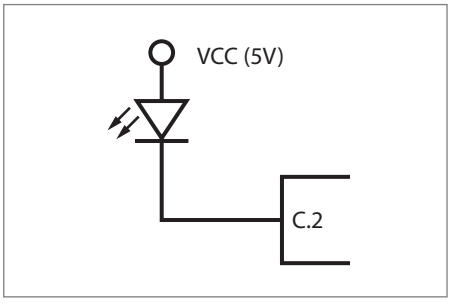




Das LED-Symbol für Schaltpläne zeigt an, in welche Richtung der Strom fließen kann – und dass Licht abgegeben wird.

Damit unsere LED blinkt, benötigen wir wechselnde Spannungen. Im Programmcode legen wir zunächst den Befehl `low` fest. Dahinter müssen wir angeben, auf welchen Pin sich dieser Befehl bezieht. Unsere LED ist am Pin C.2 des Picaxe-08M2 angeschlossen, also ergänzen wir `C.2` hinter `high`. Wäre die LED mit einem anderen Pin verbunden, müssten wir die Bezeichnung entsprechend verändern. Auf einem anderen Board und in einer anderen Schaltung würde die LED jetzt ausgehen, aber auf dem Nano-Axe-Board wird sie nun angeschaltet.

Anschließend machen wir eine `pause`. Auch hier brauchen wir eine weitere Angabe, damit der Mikrocontroller unsere Wünsche umsetzen kann und nicht für immer pausiert. Hinter der Zahl `1000` verbergen sich 1000 Millisekunden, also eine Sekunde. Das ist für den Anfang ein guter Wert, damit die LED erkennbar blinkt und nicht zu lange oder zu kurz aufleuchtet.



Unsere LED hängt mit ihrer Anode (+) an der Spannungsversorgung VCC, statt wie üblicherweise am Pin. Der Pin C.2 ist stattdessen mit der Kathode (–) verbunden, die sonst an GND hängt.

Mit dem Befehl `high` schalten wir die Spannung wieder ab, damit die LED ausgeht. Wie bei `low` müssen wir angeben, an welchem Pin dies passieren soll. Würden wir uns vertippen und etwa `C.1` statt wieder `C.2` angeben, würde die LED einfach weiter leuchten. Etwas Spannung liegt auch jetzt noch an, solange der Mikrocontroller selbst über das USB-Kabel oder Batterien mit Spannung versorgt wird. Allerdings reicht sie nicht mehr aus, um in der LED Licht zu erzeugen. Damit die LED eine erkennbare Zeit dunkel bleibt, befehlen wir dem Mikrocontroller eine weitere `pause` von 1000 Millisekunden. Wer nicht das Nano-Axe-Board nutzt, sollte jetzt eine leuchtende LED sehen.

Damit sind wir am Ende des Programms angekommen. Der Befehl `goto` erklärt dem Picaxe, zu welchem Label er nun gehen soll – in unserem Fall zu `main`. Damit beginnt das Programm von vorn. Der Pin C.2 wird wieder auf `low` gesetzt und beginnt zu leuchten. Dann wartet der Mikrocontroller eine Sekunde, bevor er den Pin auf `high` setzt, wieder wartet und wieder zum Anfang springt. Ohne den Hinweis `goto main` wäre das Programm zu Ende. Mit dem Hinweis läuft es so lange, wie der Picaxe mit Spannung versorgt wird und alle Komponenten heil sind.

Auf unserem Cheat-Sheet sind die Befehle als `high pin, {,pin, pin ...}` und `low pin, {,pin, pin ...}` aufgeführt. Die geschweiften Klammern {} zeigen an, dass ein Befehl um bestimmte Angaben erweitert werden kann. Wäre auf dem Nano-Axe-Board eine dritte LED an Pin C.2, könnten wir sie zusammen mit der bekannten LED mit `high C.2, C.3` ansprechen. Die Klammern selbst dürfen nicht ins Programm geschrieben werden.

Ab auf den Picaxe

Mit einem Klick auf den Button *Prüfen* können wir die sogenannte Syntax des Programms überprüfen. Damit werden die Regeln bezeichnet, die man einhalten muss, damit der Mikrocontroller das Programm versteht. So müssen die Befehle ohne Rechtschreibfehler geschrieben sein. In unserem Programm kommen außerdem der Doppelpunkt hinter unserem Label sowie die notwendigen Angaben zu den verwendeten Pins und der Zeit für unsere Pause hinzu. Viel Raum für Fehler ist das noch nicht, aber sicher ist sicher.

Wenn alles passt, spielen wir das Programm endlich mit einem Klick auf *Programmieren* auf den angeschlossenen Mikrocontroller. In der unteren Zeile des Picaxe-Editors gibt es dafür eine Statusanzeige, die ganz rechts eingeblendet wird. Zunächst sucht der Editor nach der Hardware und sendet dann erst den Programmcode und schließlich Daten. Hat alles geklappt, wird von Windows die Meldung „Abgeschlossen PICAXE-08M2 v4.A erfolgreich“ eingeblendet.

Auf dem Nano-Axe-Board sollte die blaue LED nun rhythmisch aufblinken, während die orange LED dauerhaft leuchtet. Gratulation, die erste Hürde ist geschafft! Wer gleich noch weiterarbeiten möchte, sollte sich die Pausen im Programmcode vornehmen. Statt einer Sekunde lassen sich andere, größere oder kleinere Werte eingeben. Mit einem Klick wird das veränderte Programm neu auf den Chip geladen.

Hilfe, es geht nicht!

Wenn die LED nicht aufleuchtet, kann dies eine Reihe von Gründen haben, die man nacheinander überprüfen sollte. Fangen wir bei der Software an. In dem Kasten *Arbeitsbereich Übersicht* links im Picaxe-Editor lassen sich die gewählten Einstellungen überprüfen. Hier müssen der Chip-Typ (PICAXE-08M2) und der COM-Port (USB-SERIAL CH340) stimmen. Praktischerweise lässt sich die COM-PORT-Liste mit einem Klick aktualisieren und auch der Gerätemanager ist verlinkt.

Stimmen alle Software-Einstellungen, kommt die Überprüfung der Hardware. So muss der Picaxe-Chip richtig im Sockel sitzen. Ist die Einkerbung auf dem Chip an der gleichen Stelle wie die Einkerbung auf dem Sockel? Stecken alle Beinchen im Sockel oder ist vielleicht eines abgerissen? Wenn der Chip falsch oder unvollständig eingesetzt ist, können die Pins nicht richtig angesprochen werden und ihre Aufgabe nicht erfüllen. Außerdem sollte die Spannungsversorgung zwischen 4,5 bis 5,5 Volt bereitstellen – liegt zu wenig Spannung an, kann es sein, dass dies einfach nicht ausreicht, um die LED zum Leuchten zu bringen. Dagegen kann zu viel Spannung den Chip sogar zerstören. Solange das Nano-Axe-Board über das USB-Kabel am Computer angeschlossen ist, sollte die richtige Spannung gewährleistet sein. Für weitere Schaltungen empfiehlt Revolution Education, drei AA-Batterien zu nutzen. In seltenen Fällen ist auch das USB-Kabel das Problem, wenn es über keine Datenleitung verfügt oder sehr lang ist.

Das Blinken verändern

Ein improvisierter Schalter hilft uns, das LED-Blinken zu verändern

Mit kurzen und längeren Pausen lässt sich bereits die Frequenz des Blinkens variieren, doch das ist nur ein Teil der Möglichkeiten für Veränderungen. Nun experimentieren wir mit dem digitalen Signal, dessen Zustände High und Low wir bisher im Programm fest vorgegeben haben. Sie sollen sich in Abhängigkeit von Eingangssignalen ändern.

Dazu schauen wir uns an, wie digitale Eingänge funktionieren, wie man ihre Signale speichern kann und wie wir mit einer Büroklammer einen einfachen Schalter für das Nano-Axe-Board bauen können. Dabei lernen wir die Debugging-Möglichkeiten des Picaxe-Systems kennen, mit dem man Problemen einfacher auf die Spur kommt. Neben digitalen Signalen gibt es übrigens auch analoge Signale – mit ihnen beschäftigen wir uns danach.

Wie funktioniert ein Eingang?

Wir haben bereits gelernt, dass digitale Signale aus zwei Zuständen bestehen: High (Spannung an) und Low (Spannung aus). Das klingt nicht nach viel, aber bedeutet immerhin, dass wir Fragen beantworten können, die nur ein Ja oder Nein als Antwort benötigen. Auch viele Schalter haben nur die zwei Zustände „an“ und „aus“. Eine Information, die zwei Zustände enthalten kann, wird Bit genannt und ist eine Grundeinheit der Informatik.

Unser digitales Signal haben wir bisher über Pin C.2, als Ausgang, gegeben und die LED an- und ausgeschaltet. Nun kommt ein Eingang hinzu, der ebenfalls mit digitalen Signalen arbeitet. Dafür nutzen wir Pin C.3, der nur als Eingang arbeiten kann (vgl. Pin-Übersicht auf S. 6). In unserem Programm PinAbfrage.bas überprüfen wir erst den Zustand unseres Pins und lassen dann, je nach Zustand, unser erstes Unterprogramm laufen.

Die ersten drei Zeilen beinhalten das Hauptprogramm. Darin fragen wir mit dem Befehl `if` ab, ob der Pin C.3 gerade angeschaltet ist. Wenn dies der Fall ist, soll das Unterprogramm `Blink` ausgeführt werden. Ist es nicht der Fall, beginnt das Hauptprogramm einfach von neuem. Das Unterprogramm `Blink` ist uns schon bekannt. Es schaltet den Pin C.2 und damit die LED für eine Sekunde an und anschließend wieder aus.

PinAbfrage.bas

```
1 main:
2       if pinC.3 = 1 then blink
3 goto main
4
5 blink:
6       low C.2
7       pause 1000
8       low C.2
9       high 1000
10 goto main
```

Das Programm übertragen wir gleich auf den Picaxe-Chip. Was aber kommt eigentlich am Pin C.3 an? Wir nutzen die elektrostatische Ladung der Luft und der Haut aus, um zufällige Signale zu erzeugen. Der Pin des Picaxe hat über die Pinleiste des Nano-Axe-Boards quasi eine kleine Antenne. Nimmt man das Board in die Hand, um über die Pins zu streichen, oder pustet es an, blinkt die LED wie zufällig auf. Um den Effekt noch deutlicher zu machen, kann man eine Büroklammer dazunehmen. Dazu falten wir sie auf und machen an einem Ende eine kleine Schlinge um unseren Eingangspin C.2 (Beschriftung C.7). Ist die Schlinge zu lose, kann man sie mit einer kleinen Zange fest zusammendrücken. Schon haben wir eine größere Antenne.

Status-Updates

Mit einem anderen, kleinen Programm (Debug.bas) können wir uns genauer anschauen, welche Informationen gerade über den Eingang eintrudeln. Dazu speichern wir das Signal erst ab und inspizieren dann den Speicher.

Wir benötigen für unser Vorhaben eine Variable: in `b0` sollen die Werte abgelegt werden, die über den `pinC.3` eingehen. `b0` kann eine begrenzte Menge an Daten des Typs Integer, also ganzzahligen Zahlen, speichern. Das B in `b0` steht für Byte, die Größe des Speichers. Ein Byte besteht aus acht Bits, die jeweils nur zwei Werte speichern können, 0 und 1. Die Kombination von acht Bits ergibt einen Werteraum von 2^8 oder genauer gesagt 0 bis 255. Mit `let` zeigt man an, dass eine Variable verändert wird – prinzipiell kann man darauf sogar verzichten, aber es macht den Code etwas lesbarer.

Debug.bas

```
1 main:
2       let b0 = pinC.3
3       debug
4 goto main
```

Mit dem Befehl `debug` öffnet der Picaxe-Editor automatisch den Code-Explorer, wenn das Programm erfolgreich übertragen wurde. Dann beginnt das Programm von vorn. Im Code-Explorer lässt sich nun in der binären und dezimalen Ansicht beobachten, wie sich der Wert von `b0` ändert und 0 oder 1 beträgt.

Komfortabfrage im Terminal

Noch etwas übersichtlicher lässt sich der Zustand der Variablen über das serielle Terminal anzeigen. Der Befehl `sertxd` unterstützt

Bits und Bytes

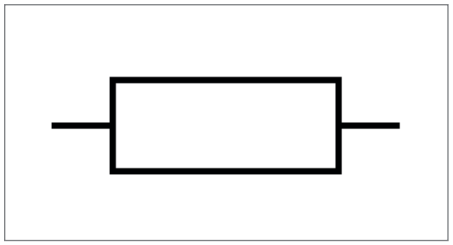
Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

PinSertxd.bas

```
1 main:
2       let b0 = pinC.3
3       sertxd ("Der Wert von Pin C.3 ist ", #b0, 13, 10)
4       pause 500
5 goto main
```

PinPullup.bas

```
1 pullup on
2 pullup %00001000
3
4 main:
5       let b0 = pinC.3
6       sertxd ("Der Wert von Pin C.3 ist ", #b0, 13, 10)
7       pause 500
8 goto main
```

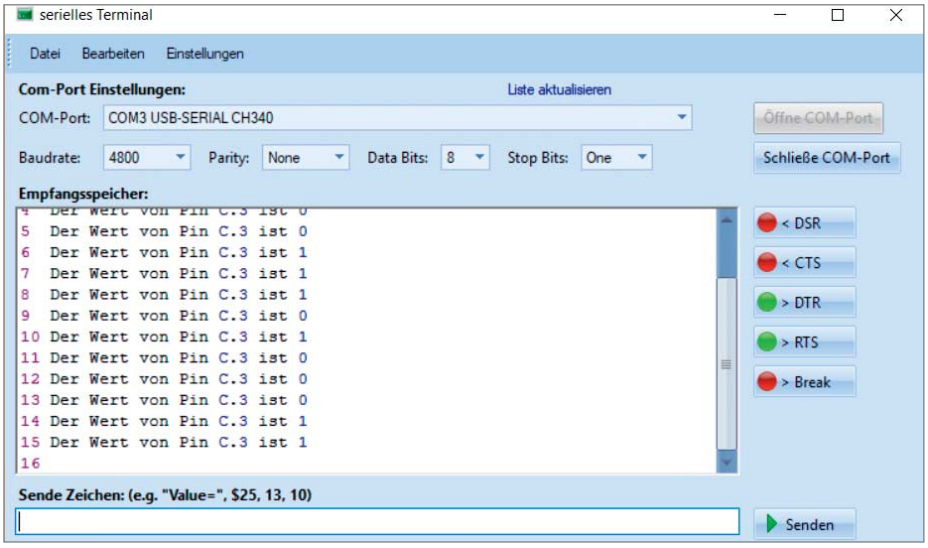


Widerstände werden in Schaltplänen als einfache Rechtecke dargestellt.

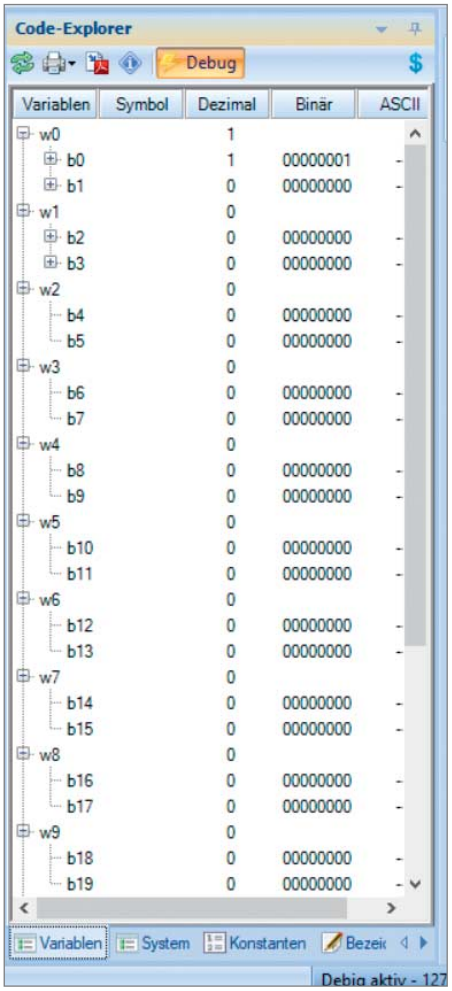
das Debugging über die serielle Schnittstelle. Der Prozess läuft über den Serial Out Pin (C.0) und schickt die gewünschten Daten über das Programmierkabel zurück an den Rechner. Dafür müssen wir nur eine Zeile in unserem Listing austauschen (siehe PinSertxd.bas).

Mit `sertxd` lassen sich Daten des Typs String ausgeben. Strings sind eine endliche Folge an Zeichen und können Zahlen, Buchstaben und Sonderzeichen umfassen. Text, der auf dem Terminal angezeigt werden soll, wird zwischen den Anführungszeichen "" eingetragen. Die Eingangssignale sind in den Variablen als Datentyp Integer, also ganzzahlige Werte, gespeichert. Damit sie auf dem Terminal angezeigt werden können, wandeln wir sie in Strings um, indem wir das Zeichen # vor den Variablennamen setzen. Statt einer Zahl wie 124 werden nun einzelne Ziffern übertragen: 1 2 4. Schließlich fügen wir noch `13, 10` an, womit bei der Ausgabe jeweils eine neue Zeile erzwungen wird. Mit einer kleinen Pause verhindern wir, dass die Signale so schnell durchrauschen, dass wir nicht mehr mitlesen können.

Wenn wir das Programm übertragen haben, müssen wir das Terminal öffnen. Dieses finden wir über den Tab *PICAXE* unter *Werkzeuge*. Die Ausgabe aktualisiert sich von alleine. Wie zuvor sollte sie zwischen 0 und 1 springen, wenn man mit dem Finger an den Pins entlangfährt. Über den Button *Schließe COM-Port* kann man die Übertragung anhalten. Achtung: Solange das Terminal-Fenster offen ist, kann der Picaxe nicht neu programmiert werden.



Die Ausgabe der zufälligen Werte an Pin C.3 auf dem seriellen Terminal



Der Code-Explorer

Es mag verlockend erscheinen, `debug` oder `sertxd` in jedes Programm einzubauen. Beide Prozesse sorgen aber dafür, dass der Picaxe langsamer arbeitet, da er neben seiner eigentlichen Aufgabe noch Daten an den Computer sendet. Beim Anschauen unserer Variablen ist das kein Problem, bei zeitkritischen Aufgaben ist es dagegen äußerst ungünstig.

Pullup!

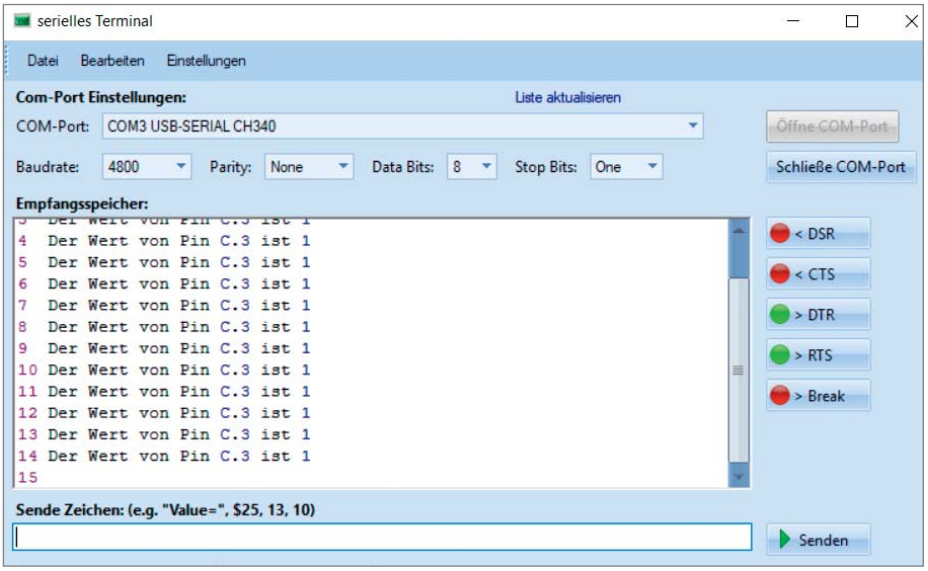
Um an unserem Eingang stabile statt zufällige Signale abfragen zu können, brauchen wir Pullup-Widerstände. Sie sorgen für ein definiertes Standardsignal. Beim Picaxe sind sie praktischerweise bereits eingebaut. Ein digitaler Eingang mit Pullup-Widerstand lie-

fert das Signal High, bzw. 1. Erst wenn er mit einem Low-Signal verbunden wird, ändert er seinen eigenen Zustand.

Integrierte Pullup-Widerstände sind allerdings nicht an allen Pins unseres Chips vorhanden, sondern nur an den Pins C.0 bis C.4. Wenn ein Pin als Ausgang genutzt wird, überbrückt der Picaxe den Widerstand von alleine. Ein zu hoher Widerstand vor unserer LED würde dazu führen, dass sie nicht mehr aufleuchten kann, weil nicht genug Spannung ankommt.

Unser Programm ergänzen wir nur um zwei Zeilen (zu PinPullup.bas). Sie stehen vor dem Hauptprogramm, damit sie nicht bei jedem Durchlauf des Programms wiederholt werden. Die erste Zeile schaltet Pullups an und die zweite Zeile legt fest, dass es um den Pin C.3 geht: `%00001000`. Gezählt wird dabei von hinten, von C.0 an. Der Befehl heißt Bitmask (auf Deutsch Bitfeld) und wird standardmäßig mit 8 Stellen, also 8 Bits geschrieben, auch wenn beim Picaxe-08M2 an dieser Stelle nur fünf Pins (C.0 bis C.4) zugeordnet werden können.

Auf dem Terminal sollten wir nun eine Änderung sehen: Der Eingang ist dauerhaft auf High geschaltet und zeigt eine 1. Um ein Low-Signal zu bekommen, müssen wir unseren Schalter aus der Büroklammer fertigbauen. Drücken wir das andere Ende zu einer Schlinge und legen es an einen GND-Pin, sollte das Terminal eine 0 ausgeben. Jetzt können wir unsere LED gezielt steuern und unterschiedliche Blink-Frequenzen einstellen.



Nun haben wir einen dauerhaften Wert an C.3.

Schalter anlegen

Zum Glück haben wir noch genau zwei Pins über, die wir dank eingebauter Pullup-Widerstände als stabile Eingänge nutzen können: C.1 und C.4. Jedem unserer drei Eingänge weisen wir in Pin-Zustandsabfrage.bas eine unterschiedliche Länge des Blinkens und der Pause zu. Als Schalter dient die Büroklammer, die GND mit einem der drei Pins verbindet. So sendet dieser Pin das Signal Low, während die anderen beiden Eingänge auf High liegen. In unserem Programm fragen wir die Zustände ab.

Zu Beginn legen wir wieder fest, an welchen Pins die Pullup-Widerstände aktiviert werden. Dann benötigen wir eine neue Variable. Für `waittime` werden wir später im Programm unterschiedliche Werte einsetzen, um das Blinken zu steuern. Vorher teilen wir dem Picaxe mit dem Befehl `Symbol` mit, dass `waittime` der Name unserer Variablen ist und dass er ihren Inhalt im Speicher `w0` ablegen soll. Diese sogenannte Wortvariable kann sich Zahlen des Typs Integer zwischen 0 und 65535 merken – also deutlich mehr als `b0`, das wir vorher genutzt haben. `b0` und `b1` sind

Pin-Zustandsabfrage.bas

```
 1 pullup on
 2 pullup %00001000
 3
 4 symbol waittime = w0
 5
 6 main:
 7      if pinC.1 = 0 then
 8           waittime = 2000
 9      elseif pinC.3 = 0 then
10           waittime = 4000
11      elseif pinC.4 = 0 then
12           waittime = 8000
13      else
14           waittime = 10000
15      endif
16
17      low C.2
18      pause waittime
19      high C.2
20      pause waittime
21
22 goto main
```

HINWEIS

Variablen werden im flüchtigen Speicher RAM abgelegt. Das heißt, dass sie gelöscht werden, wenn man den Strom abzieht. Dagegen legt der Picaxe den Programm-Code im nichtflüchtigen Speicher ab, damit er auch bei einem Neustart erhalten bleibt.

Byte-Variablen, die jeweils nur 8 Bit an Informationen enthalten. Man kann sie aber zusammen als 16-bittige Wortvariable nutzen.

Auch die Deklaration der Variablen müssen wir dem Picaxe nur einmal mitgeben und schreiben sie daher vor das Hauptprogramm. Bei der Vergabe von Variablennamen ist darauf zu achten, dass sich dahinter keine bereits vergebenen Befehle verbergen. Hier hilft ein Blick in das Handbuch des Herstellers. Außerdem überprüft der Editor dies vor dem Hochladen.

In unserem Programm nutzen wir nun die bekannte `if`-Abfrage, um den Zustand des `PinC.1` zu bestimmen. Wenn wir dort den Schalter angelegt haben und der Zustand 0, also low, beträgt, wird der Wert unserer Variablen `waittime` auf 200 Millisekunden gesetzt. Ist das nicht der Fall, kontrolliert der Picaxe unsere zweite Frage, die mit `elseif` gekennzeichnet ist. Wäre `PinC.3` mit der Büroklammer verschaltet, sollte die Variable auf 400 Millisekunden gesetzt werden. Genauso funktioniert die Abfrage auch für `PinC.3`. Solange nichts davon eintritt (`else`) und alle Eingänge auf High liegen, wird die Zeit auf 1 Sekunde festgelegt.

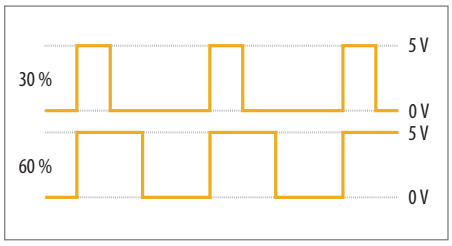
Schließlich sind die letzten Zeilen noch einmal unser allererstes Programm, mit einer kleinen Änderung. Statt einer festen Pausenzeit ist nur die Variable eingetragen, die bei jedem Durchlauf anders belegt werden kann. Jetzt noch das Programm auf den Picaxe laden und dann ist es geschafft. Mit einer Büroklammer können wir einstellen, wie schnell die LED auf dem Nano-Axe-Board blinkt.

LED dimmen

Im letzten Teil des Einstiegs variieren wir die Helligkeit der LED

Während digitale Signale nur zwei Zustände haben, sind analoge Signale deutlich komplexer und können etwa in Wellenform beschrieben werden. Da Mikrocontroller nur mit digitalen Signalen arbeiten, müssen wir also kreativ werden – mit ein wenig Geschick kann unser Picaxe-Chip immerhin so tun, als arbeite er mit analogen Signalen.

Zunächst probieren wir dies wieder an einem Ausgang. Wie die Umwandlung analoger Eingangssignale in digitale funktioniert, schauen wir uns später an. Konkret werden wir die Trägheit des Auges ausnutzen. Wenn sich Bilder und Lichter schnell genug ändern, nimmt das Auge die einzelnen Zustände nicht mehr wahr, sondern wir sehen eine kontinuierliche Veränderung. Den Effekt kennt jeder vom Fernsehen: Dort werden pro Sekunde 25 Bilder gezeigt, die wir aber nicht mehr einzeln erkennen können. Das gleiche Phänomen nutzen wir jetzt, um unsere LED zu dimmen.



Das Verfahren für die Umsetzung heißt Pulsweitenmodulation (PWM). Unser digitales Signal arbeitet dabei weiter mit den Pegeln High und Low, aber wir verändern die Dauer der Pausen und der Anschaltzeiten. Das Ziel ist ein schneller Wechsel zwischen High und Low, damit wir am Ende eine Spannung zwischen 0 und 5 Volt ausgeben. Statt nur an und aus zu gehen, kann unsere LED so heller oder weniger hell aufleuchten.

Die Funktionsweise ist in der Grafik zu erkennen: Wir belassen die Periodendauer immer gleich – also die Zeit, die während eines Zyklus aus High- und Low-Signal vergeht. Innerhalb dieser Periode verändern wir die Anschaltzeiten und damit auch die Pausen. Eine Angabe von 30% bedeutet etwa, dass 30 Prozent der Zeit einer Periode das High-Signal gesendet wird. Bei 60% wird das High-Signal deutlich länger gesendet. Bei Angaben von 0 oder 100% wird nur das Low- beziehungsweise High-Signal genutzt.

Pulsweite bestimmen

Die Picaxe-Chips bringen mit `pwmout` bereits einen Befehl mit, um Pulsweitenmodulation in einem Programm einzusetzen. Anders als andere Picaxe-Befehle läuft `pwmout` von alleine immer weiter, bis ein anderer Befehl geschickt wird. Um `pwmout` im Code zu nutzen, sind noch einige Angaben nötig. Zunächst wird wieder der Pin angegeben – dabei ist zu beachten, dass der Picaxe-08M2 nur am Pin C.2 Pulsweitenmodulation unterstützt. Praktischerweise ist das genau der Pin, an dem unsere LED anliegt.

PWM.bas

```
1 do
2      pwmout C.2, 99, 199
3 loop
```

Außerdem benötigt `pwmout` Angaben für `period` und `duty cycle`. `period` umfasst die Periodendauer und wird in Werten zwischen 0 und 255 angegeben. `duty cycle` ist die Anschaltzeit und eine 10-Bit-Variable. Es können also Werte zwischen 0 und 1023 eingetragen werden. Dabei ist `duty cycle` beim Picaxe auf den 4-fachen Wert der Periodendauer begrenzt. Höhere Angaben führen zu unberechenbaren Verhalten.

Jetzt müssen wir aus unseren Angaben wie „30 Prozent Anschaltzeit“ noch Werte berechnen, mit denen der Chip arbeiten kann. Praktischerweise gibt es im Picaxe-Editor ein Berechnungsprogramm, das uns diese Arbeit abnimmt. Aufrufen kann man es über den Tab *PICAXE*. Dort ist unter *Wizards* der *PWMOUT Wizard* zu finden. Für unser erstes Experiment müssen wir dort nur bei *Duty* die gewünschte Anschaltzeit in Prozent angeben. Alle weiteren Angaben wie die Frequenz sind bereits vorausgefüllt und können übernommen werden. Wenn wir uns an den Physikunterricht erinnern, sind Frequenzen der Kehrwert von Periodendauern. Die voreingestellte Frequenz von 10.000 Hertz bedeutet also, dass eine Periodendauer 0,0001 Sekunden lang ist.

Ein Klick auf *Calculate* rechnet nun die Werte für den Picaxe aus. Ein weiterer Klick auf *Copy* übernimmt den Befehl in die Zwischenablage, damit man ihn im Quellcode einfügen kann. Mit seinen drei Zeilen ist unser Programm PWM.bas wieder sehr überschaubar. Die LED sollte nun etwas weniger hell aufleuchten als zuvor. Statt mit `main` starten wir dieses Programm mit `do`. Zusam-

RECHNUNG PER HAND

Natürlich kann man die Berechnung der Variablenangaben auch von Hand erledigen. Für die Umrechnung der Frequenz in `period` gilt:

1/Frequenz = (`period` + 1) × 4 ÷ Interner Takt

Umgestellt ist dies:

((Takt / Frequenz) ÷ 4) – 1 = `period`

Standardmäßig beträgt der interne Takt des Picaxe-08M2 vier Megahertz. Mit unserem Beispiel von 10.000 Hertz rechnen wir

((1/10.000 × 4.000.000) ÷ 4) – 1 = 99

Die Formel für die Anschaltzeit `duty cycle` lautet:

(1/Frequenz × Prozent/100) = `duty cycle` × Interner Takt

Für 50% ergibt dies:

(1/10.000 × 50/100) × 4.000.000 = 200

men mit `loop` in der dritten Zeile ergibt sich eine Schleife, die immer wieder ausgeführt wird. Sprunganweisungen wie `goto main`, das wir vorher genutzt haben, sind in den meisten Programmiersprachen zwar vorhanden, werden heute aber nicht mehr benutzt. Sprünge aus bestimmten Blöcken hinterlassen sogenannten *Garbage* im Speicher – Daten, die nicht mehr genutzt werden, aber Speicherplatz blockieren. Oft können statt Sprunganweisungen auch alternative Konstrukte wie `do`-Schleifen genutzt werden. Um unter bestimmten Bedingungen aus Schleifen herauszuspringen, ist `goto` bei der Arbeit mit dem Picaxe aber ein hilfreiches Werkzeug.

Dimmen mit Schalter

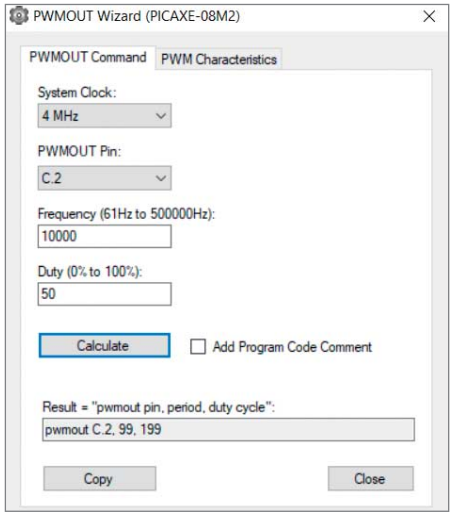
Schließlich wollen wir unser bekanntes Schalterprogramm mit dem Dimmeffekt verbinden. Je nachdem, welchen Pin wir mit der Büroklammer auf GND und damit 0 ziehen, soll die LED heller oder dunkler aufleuchten. Wie zuvor haben wir die drei Pins C.1, C.3 und C.4 als digitale Eingänge. Damit sie klare Signale liefern, statt Rauschen aus der Luft einzufangen, müssen wir auch wieder die Pullup-Widerstände aktivieren. Statt des Blinkens geben wir aber am Ende unterschiedliche Helligkeit aus. Außerdem ist dieses Programm ebenfalls in eine `do-loop`-Schleife eingebunden, die die Befehle `main` und `goto main` ersetzt.

Die Frequenz beträgt erneut 10.000 Hertz, beziehungsweise ein Durchlauf dauert 0,0001 Sekunden. Die unterschiedlichen Werte der Anschaltzeit werden jeweils in der Variablen `duty` gespeichert. Bei Pin C.1 beträgt sie 0 und damit 0 Sekunden. Die Werte für Pin C.3 und C.4 entsprechen 30 Prozent und 60 Prozent. Ohne angeschlossenen Pin wird der Wert auf 399 und damit 100 Prozent gesetzt. In den Argumenten von `pwmout` rufen wir schließlich den aktuellen Inhalt der Variablen ab.

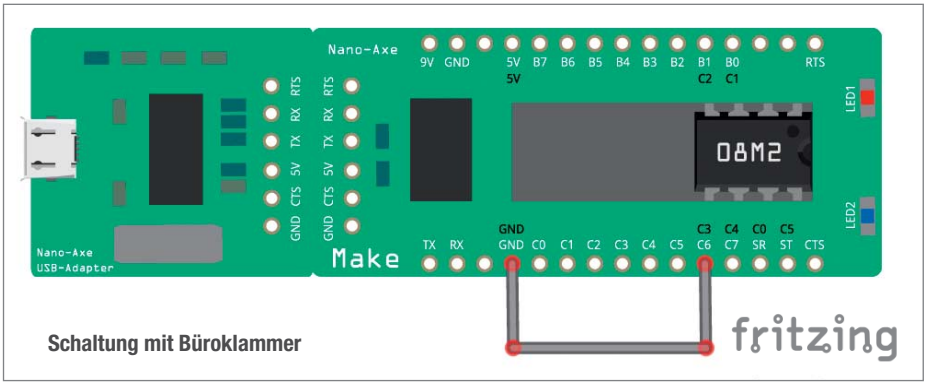
Auch hier gilt es wieder, die Besonderheit des Nano-Axe-Boards und seiner LED zu beachten: Bei 0 Prozent Anschaltzeit leuchtet die LED am hellsten. Bei 100 Prozent Anschaltzeit ist sie beinahe ausgeschaltet. In Schaltungen ohne das Board funktioniert dies genau anders herum. *—hch*

PWM-Abruf.bas

```
 1 pullup on
 2 pullup %11010
 3
 4 symbol duty = w0
 5
 6 do
 7       if pinC.1 = 0 then
 8            duty = 0
 9       elseif pinC.3 = 0 then
10            duty = 119
11       elseif pinC.4 = 0 then
12            duty = 239
13       else
14            duty = 399
15       endif
16
17       pwmout C.2, 99, duty
18
19 loop
```



Leider wurden die Hilfsprogramme wie der PWMOUT Wizard bei der Übersetzung vergessen und erscheinen weiterhin auf Englisch.



Schaltung mit Büroklammer