ALLNET®

Brick 'R' knowledge

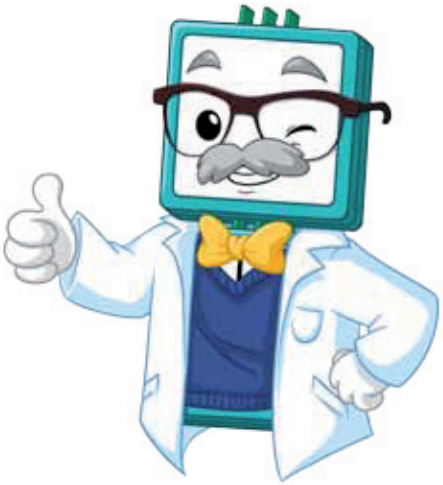
Arduino coding Set

Auf die Plätze fertig los
Arduino Anwendungen im Handumdrehen erstellen

Inhaltsverzeichnis

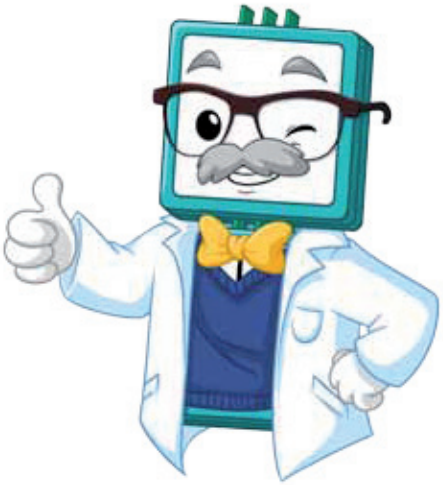
1. Vorwort

Das Brick'R'knowledge System wurde zum ersten mal auf HAM Radio Ausstellung am 28.06.2014 von DM7RDK (Rufzeichen) vorgestellt.
Hier liegt nun das neue Arduino Coding Set vor.
Das Besondere an unserem Elektronikset ist, dass die einzelnen Bausteine über ein Stecker-System verbunden werden, bei dem die zusammenzugefügenden Teile baugleich sind (Hermaphrodite).
So können auch knifflige Stromkreise realisiert werden. Auch das Zusammenstecken der einzelnen Bausteine in verschieden Winkeln ist möglich! Für die Rückführung der null Volt (0V, Masse) sind gleich zwei Kontakte vorhanden!
Damit lassen sich kompakte Schaltungen aufbauen, bei der die 0V-Rückführung für eine stabile Spannungsversorgung der Bausteine sorgt. Eine weitere Besonderheit ist, dass man solche Schaltungen sehr leicht erklären und dokumentieren kann.
Viel Spass mit dem Set wünscht
Rolf-Dieter Klein



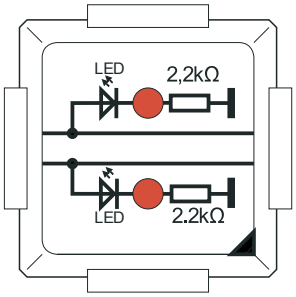
1.1 Sicherheitshinweise

Achtung, die Bausteine des Elektroniksets NIE direkt an das Stromnetz (230V) anschließen, andernfalls besteht Lebensgefahr!
Zur Spannungsversorgung (9V) ausschließlich das mitgelieferte Netzteil (Batteriebaustein) verwenden. Die Versorgungsspannung beträgt hier gesundheitsungefährliche 9 Volt bei einem Stromfluss von ca. 1 Ampere. Bitte tragen Sie auch Sorge dafür, dass offen herumliegende Drähte nicht in Berührung oder Kontakt mit Steckdosenleisten (gewöhnliche Zimmerverteiler) kommen bzw. in diese hineinfallen, auch hier besteht andernfalls die Gefahr eines gesundheitsgefährlichen Stromschlags bzw. elektrischen Schocks. Schauen Sie niemals direkt in eine Leuchtdiode (LED), da hier die Gefahr besteht, die Netzhaut zu schädigen (Blenden). Die Netzhaut befindet sich im Auge und hat die Aufgabe, die einfallenden Lichtreize durch die auf ihr befindlichen Zapfen (das Farbsehen) und die ebenfalls auf ihr befindlichen Stäbchen (Hell-,Dunkelsehen) in für das Gehirn verwertbare Reize umzuwandeln. Es werden zwei LED-Bausteine mitgeliefert: LED „grün“(2mA) und LED „gelb“(2mA) mit einer Stromaufnahme von 2 Milli-Ampere. Den, im Elektronikset mitgelieferten gepolten Kondensator (Tantalkondensator/Elektrolytkondensator) (100µF) mit der Kapazität von 100 Mikro-Farad niemals mit dem mit „Plus“ gekennzeichneten Kontakt direkt oder indirekt an den mit „Minus“ gekennzeichneten Anschluss der Spannungsversorgung (9V) anschließen, sondern den mit „Plus“ gekennzeichneten Anschluss nur direkt oder indirekt an den mit „Plus“ gekennzeichneten Kontakt der Spannungsversorgung (9V) verbinden. Man spricht hier von der Polung! Ist der Tantal- oder Elektrolytkondensator falsch gepolt, also diese im vorangegangen beschriebene Regel nicht eingehalten, kann dieser zerstört werden - Explosionsgefahr! Es ist unbedingt darauf zu achten, dass mitgelieferte Netzteil (Batteriebaustein) nach den Versuchsaufbauten wieder von allen Bausteinen zu trennen, andernfalls besteht die Gefahr eines Elektrobrandes!
Bausteine oder andere Teile des Elektroniksets nicht verschlucken, andernfalls sofort einen Arzt hinzuziehen!

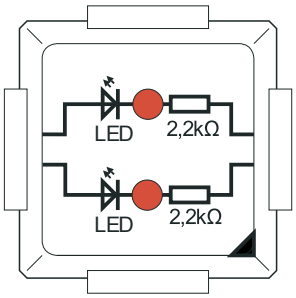


2. Übersicht

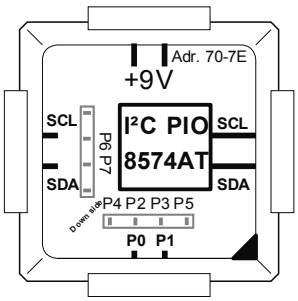
2.1 Bausteine



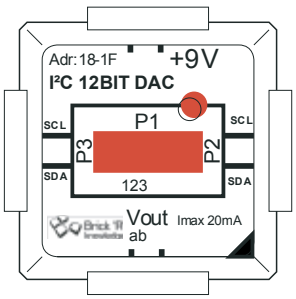
In dem Baustein sind zwei LEDs (Light Emitting Diode) untergebracht. Zusätzlich wird je ein Vorwiderstand von 2.2 KOhm verwendet, der die Dioden vor zu hohem Stromfluss schützt. Sie sind für 2mA Strom optimiert. Die beiden Widerstände sind mit Masse verbunden. So kann man den Baustein direkt an ein Ausgangspaar vom Arduino Nano Brick anschließen. Zusätzlich werden die beiden Signale an das andere Ende weitergeleitet, so dass sich dort weitere Steuerungen durchführen lassen.



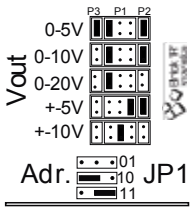
In dem Baustein sind zwei LEDs (Light Emitting Diode) untergebracht. Zusätzlich wird je ein Vorwiderstand von 2.2 KOhm verwendet, der die Dioden vor zu hohem Stromfluss schützt. Sie sind für 2mA Strom optimiert. Die Widerstände sind mit der anderen Seite des Bricks verbunden und getrennt geführt, so dass man da auch Schaltungsvarianten bauen kann. Wenn man die Widerstände mit einem Masse-baustein an 0V koppelt lässt er sich direkt mit den Ausgängen des Nano betreiben. Generell ist darauf zu achten, dass die Kathode negativ gegenüber der Anode angeschlossen werden muss. Die Kathode ist mit einem Querstrich am LED-Dioden-symbol gekennzeichnet.



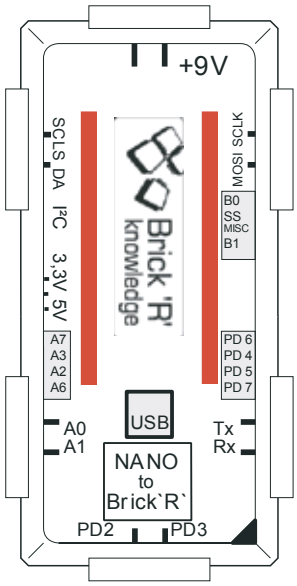
Der Arduino besitzt einen speziellen Bus, genannt I2C, mit dem man weitere Bau-steine anschliessen kann. Der 8574 ist ein Baustein mit dem man zusätzlich 8 Ein-oder Ausgabeports bekommt und so die Anzahl der IO-Ports beim Arduino einfach erweitern kann. Der I2C verwendet Adressen für die Bausteine, die sich direkt im Brick einstellen lassen, um so maximal 8 dieser Bausteine verwenden zu können. Wir nutzen den 8574AT als Baustein. Es gibt noch den 8574T der 8 andere Adressen belegt. Beide zusammen könnten dann 8x8+8x8 = 128 IO-Ports verfügbar machen. Bei unserem Brick sind alle IO-Ports herausgeführt, die Ports P4-P7 jedoch auf der Unterseite der Platine.



D/A Umsetzer Brick: Über den I2C Bus kann man einen Wert zwischen 0..4095 ausgeben, der dann in eine proportionale Spannung umgesetzt wird. Der Baustein kann für zwei unter-schiedliche Adressen konfiguriert werden, so dass man maximal zwei dieser Bausteine anschließen kann. Das Besondere ist auch die Steckleiste auf der Vorderseite, über die sich unterschiedliche Spannungsbereiche einstellen lassen. Maximal können 20mA entnommen werden, dabei ändert sich aber die Ausgangsspan-nung. Auf der Rückseite ist die Belegung der Spannungseinstel-lung abgedruckt.



Arduino NANO: Dies ist das Herz unseres Sets. Der Prozessor wird oben auf die Steckleisten gesetzt. Die Programmierung erfolgt wie bei Arduino üblich, über den PC mit dem Arduino Entwicklungssystem, dass man sich von der Homepage entsprechend runterladen kann. Zur Programmierung wird er auch über den USB-Port versorgt, man kann auch eine 9V Quelle zur Versorgung verwenden, und dann nach der Programmierung vom USB lösen. Achtung. Die Eingänge des Arduino sollten nie in direkten Kontakt mit der 9V Quelle kommen, da sie trotz Schutzschaltungen auf der Platine nur für 5V ausgelegt sind.



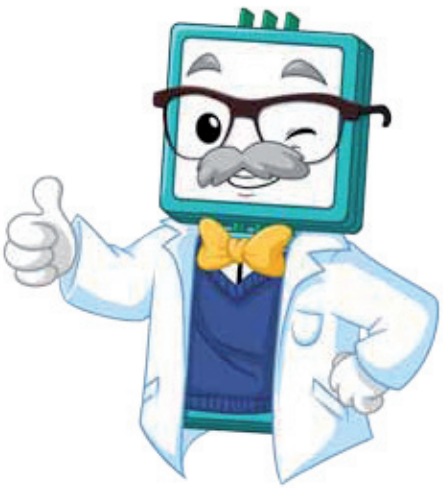
IO Zuordnung

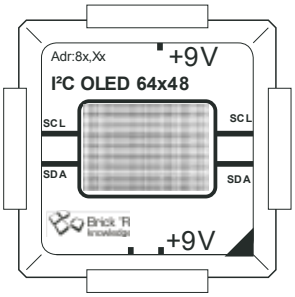
PD0 - 0
PD1 - 1
PD2 - 2
PD3 - 3
PD4 - 4
PD5 - 5
PD6 - 6
PD7 - 7

PB0 - 8
PB1 - 9
PB2 - 10
PB3 - 11
PB4 - 12
PB5 - 13

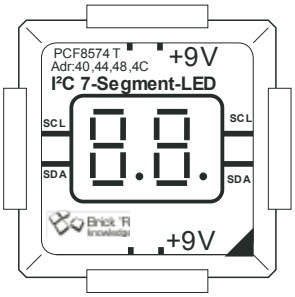
```
A0 - analog 0
A1 - analog 1
A2 - analog 2
A3 - analog 3
A4  (SDA) - analog 4
A5 (SCL) - analog 5
A6 - analog 6

A7 - analog 7
```

```
Portbelegung
Kürzel      Port  Parameter
SCK         PB5   #13
MOSI        PB3   #11
B1          PB1   #9
SS          PB2   #10
MISO        PB4   #12
B0          PB0   #8
```

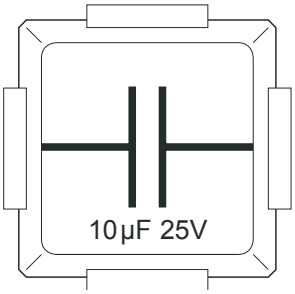




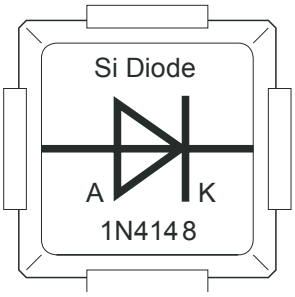
OLED: Organische Leuchtdiode. Genaugenommen sind 64x48 Leuchtdioden in einer Matrix angeordnet und können einzeln mit Hilfe von Befehlen durch den I2C angesteuert werden. Damit lassen sich mehrzeilige Texte darstellen, aber auch einfache monochrome Grafiken.



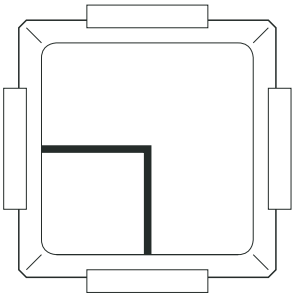
7-Segment-Anzeige: Sie besteht aus 7 Leuchtbalken und einem Punkt. Dahinter verbirgt sich je eine Leuchtdiode. Die LEDs werden mit Hilfe zweier 8574T Bausteine angesteuert (16 Ausgangsleitungen an die LEDs). Auf der Rückseite lässt sich mit kleinen Schaltern die I2C Adresse einstellen. Maximal können vier solcher Bausteine an einem I2C Bus betrieben werden.



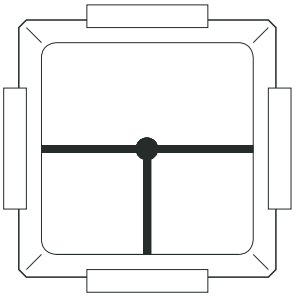
Dieser Kondensator hat eine Kapazität von 10µF. Eine Ladespannung von 1V wird somit schon nach 10µs erreicht, wenn er mit einem Strom von 1A geladen wird. Der Kondensator darf die aufgedruckte Maximalspannung von 25V nicht überschreiten!



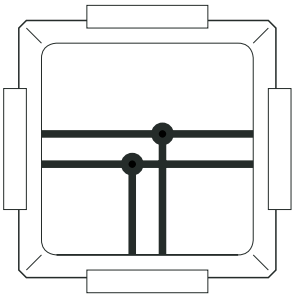
Die besondere Eigenschaft dieses Dioden-Bricks liegt darin, dass er Spannungen bis zu 100V gleichrichten und sehr hohe Frequenzen von bis zu 100MHz schalten kann (bitte nicht ausprobieren - 100V sind Lebensgefährlich).
Er wird in Durchlassrichtung betrieben. In Sperrrichtung kommt praktisch kein Stromfluss zustande. In Flussrichtung kann maximal 200mA verwendet werden. In Sperrrichtung fließt nur ein Strom von 25nA (Nano-Ampere 25nA = 25E-9A) bei 20V. Der Spannungsabfall in Flussrichtung liegt bei ca 0.6V-1V je nach durchflossenem Strom.



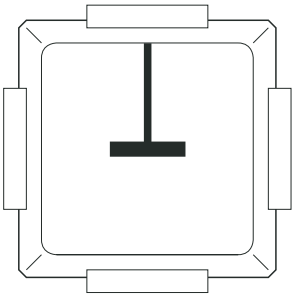
Mit dem Eck-Brick werden zwei angrenzende Seiten miteinander verbunden.



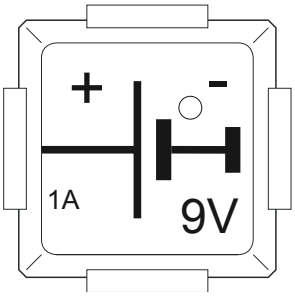
Mit dem T-Brick werden Abzweigungen hergestellt.
Der T-Brick kann auch anstelle einer Ecke verwendet werden.



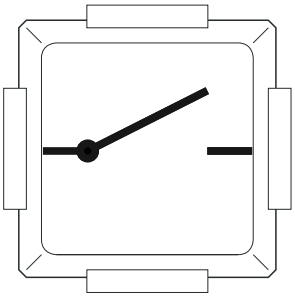
Mit dem Doppelt-T-Brick werden Abzweigungen hergestellt.
Dabei sind die beiden Mittelleiter getrennt.



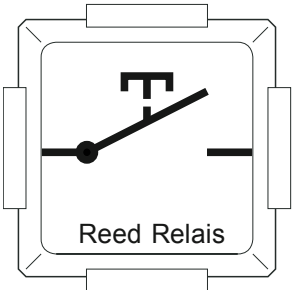
Der Masse-Brick hilft Schaltungen übersichtlicher zu gestalten. Die Masse oder auch Ground genannt, ist der Referenzpegel einer Schaltung, bei uns 0V. Damit lässt sich ein Stromkreis schliessen. Die Leitungen sind sozusagen unsichtbar miteinenander im inneren des Bricks verbunden und bilden ein 0V Netz.
Es können mehrere Masse-Bricks in einen Versuchsaufbau eingefügt werden. Der Masse-Brick verbindet die beiden mittleren Kontakte des Anschlusses mit den beiden außen liegenden Kontakten, die so eine Masseleitung bilden.



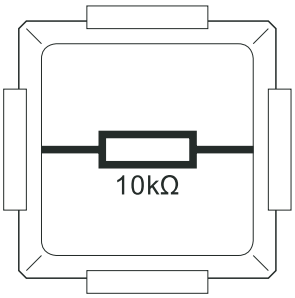
Ein weiterer Versorgungs-Brick ist der abgesicherte Netzteiladapter. Er liefert eine stabilisierte Gleichspannung von 9V und einen maximal möglichen, kurzschlusssicheren Stromfluss von 1A. Die Masse ist auf der Seite der Anode, dem Minuspol, durchverbunden, so dass kein weiterer Masse-Brick verwendet werden muss. Eine Kontroll-LED leuchtet, sobald der Netzteil-Brick Spannung bereitstellt. Das Netzteil ist am Ende der Versuchsdurchführung sofort vom Stromnetz zu trennen! Das Netzteil sollte bei längerer Inaktivität vom Stromnetz getrennt werden. Ausserdem sollte das Netzteil bei längerer Inaktivität vom Stromnetz getrennt werden.



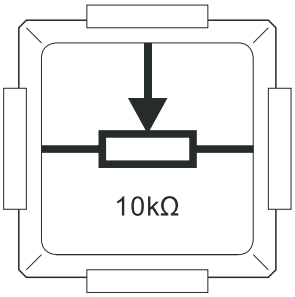
Der Taster-Brick ist ein elektromechanisches Bedienelement, das eine leitende Verbindung nur während des Gedrückthaltens ermöglicht. Im Moment des Loslassens öffnet sich diese wieder und der Taster kehrt in seine Ausgangslage zurück.



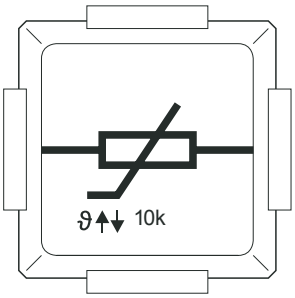
Ein Reed-Kontaktschalter (Reed, englisch für dünnes Rohr) wird durch ein von außen herangebrachtes Magnetfeld betätigt, sobald dieses stark genug ist. Das Magnetfeld kann durch einen Dauer- oder Elektromagneten erzeugt werden. Reedschalter haben eine geringe Eigenmagnetisierung. Sie schalten immer dann, wenn das äußere Magnetfeld in gleicher Richtung addiert wird und trennen, wenn es die Eigenmagnetisierung aufhebt, es also entgegengesetzt gerichtet ist. Sie finden als Näherungsschalter Verwendung.



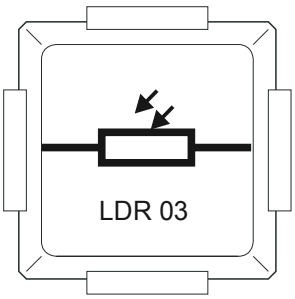
Der Widerstand (hier als Beispiel der 10 kOhm Widerstand abgebildet) wird bei uns meist zur Begrenzung des Stroms verwendet oder er dient als Spannungsteiler bei der Beschaltung der anlogen Eingänge des Nano-Bricks.
Die Widerstands-Bricks sind, wenn nicht anders beschriftet mit maximal 1/8 Watt belastbar.



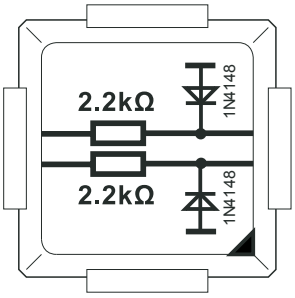
Das Potentiometer ist ein manuell veränderbarer Widerstand. Hier fährt ein dritter Kontakt (Schleifer) die Länge des Widerstandes ab und ändert so die Höhe des elektrischen Widerstandswertes an seinem Anschluss. Er ist im Bereich 0 bis 10k einstellbar. Ist der Schleifer oder auch Mittelkontakt genannt oder einer der anderen Kontakte direkt mit der Spannungsversorgung verbunden, so kommt es zu einem Kurzschluss. Dies ist unbedingt zu vermeiden! Das Potentiometer hat eine maximale Leistung von ca. 1/8 W.



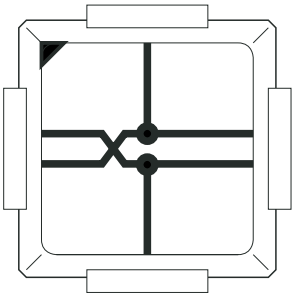
Ein NTC-Widerstand (Negative-Temperature-Coeffizient) ist ein temperaturabhängiger Widerstand. Die Wertänderung des Widerstandes erfolgt hierbei entgegengesetzt zum Temperaturgradienten, so sinkt der elektrische Widerstand bei ansteigender Temperatur. Im Deutschen werden NTC-Widerstände daher auch Heißleiter genannt. Bei Raumtemperatur beträgt der Widerstandswert 10k. Er eignet sich gut als Temperatursensor.



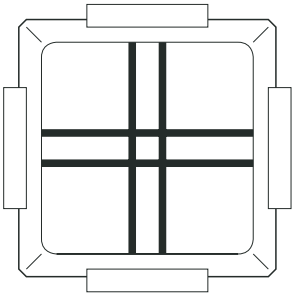
Der LDR 03 (Light-Dependent-Resistor) ist ein lichtabhängiger Widerstand. Je mehr Licht auf den Sensor fällt, desto kleiner ist der Widerstand. Die Werte variieren von einigen 100 bei Helligkeit und mehreren Kilo Ohm bei Dunkelheit. Die Veränderung des Widerstandswertes ist kontinuierlich.



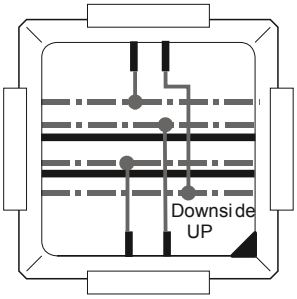
Doppelwiderstand mit Schutzdioden. Speziell zum Schutz der analogen Eingänge des Arduino Nano. Die Dioden müssen in Richtung des Arduino Nano Bricks angeschlossen werden, sie schützen vor negativen Spannungen. Im Nano Brick selbst sind auch nochmal zwei Widerstände mit ca. 100Ohm untergebracht. Die 2.2KOhm Widerstände können die Eingänge auch vor 9V Spannung schützen (bitte nicht dauernd anlegen, der Analogeingang ist für 0-5V ausgelegt).



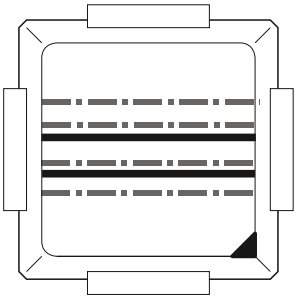
Mit diesem Brick werden separat belegte Kontakte in der Mitte gentrennt kontaktiert. Durch die Trennung und Überkreuzung der Leitungen können diese zum Wechseln der Verbindungen genutzt werden. Oben und unten sind beide Mittelkontakte miteinander verbunden.



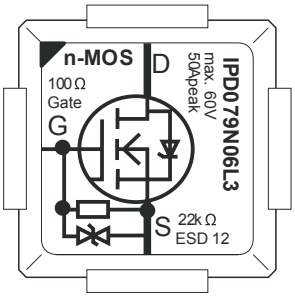
Doppelte Kreuzung nicht verbunden Verbindet die jeweils gegenüberliegenden Mittelkontakte einzeln. Im Zentrum sind die Leitungen isoliert.



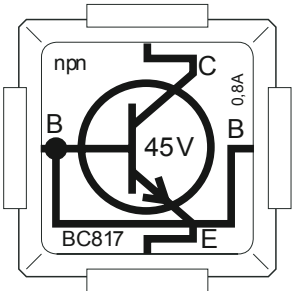
Ein Spezialbaustein, der für den Nano Brick bereit steht. Er besitzt zusätzlich zu den Kontakten auf der oberen Ebene auch noch vier unabhängige Kontakte auf der Unterseite der Platine. Diese werden dann an den Seiten auf die obere Ebene geführt, dort können normale Bricks verwendet werden, um an die Signale heranzukommen.



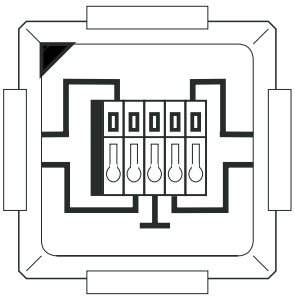
Verlängert die vier Kontakte oben (inklusive der Masse an den Außenkontakten) und die vier Kontakte auf der unteren Ebene, die unabhängig voneinander sind.



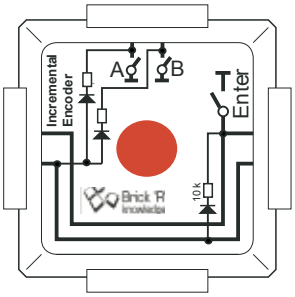
Unser Feldeffekt-Transistor steuert den Stromfluss zwischen Drain und Source über die am Gate angelegte Spannung. Das Besondere an diesem Bauelement ist, dass die Verbindung zwischen Gate und Source sehr hochohmig ist. Feldeffekttransistoren werden als „MOSFET“ (Metall-Oxid-Semileitender Feldeffekt-Transistor) bezeichnet oder kurz als „MOS“.
Es gibt unterschiedliche Arten von MOS, dieser ist ein normal sperrender n-Kanal. Dies bedeutet, dass die Schwellspannung am Gate (Tor) anliegen muss, damit ein Stromfluss zwischen Drain (Abfluss) und Source (Quelle) erfahren wird. Die Spannung muss am Gate positiv zur Source sein. Der MOSFET kann bis zu 60V Source - Drain- Spannung vertragen und einen Spitzenstrom von 60A (nicht beides gleichzeitig versteht sich). Achtung. Die Kontakte des Brickssystems vertragen maximal 6.3A pro Kontakt. Der Gate-Eingang ist mit einer ESD 12V geschützt, sowie einem Widerstand von 22kOhm.



Dieser Brick enthält einen npn-Transistor. Er steuert den Stromfluss zwischen Kollektor (C) und Emitter (E) über den wesentlich kleineren Strom an seinem Basisanschluss (B). Die Basis ist bei einem npn-Transistor gegenüber dem Emitter positiv anzusteuern. Es ergibt sich eine Basisemitterspannung UBE von ca 0.7 Volt. Man sollte dem Transistor nicht mehr als 500mA abverlangen, obwohl die Spitzenstromstärke ca. 800mA ist, je nach Herstellermodell. Die Verlustleistung darf je nach genauem Typ nicht mehr als 1/4 Watt betragen. Dies ist wichtig, wenn der Transistor nicht als Schalter verwendet wird und einen hohen Spannungsabfall zwischen Kollektor und Emitter hat.



Damit können Leitungen oder Bauteile befestigt und an die Schaltung angeschlossen werden. Mit einem kleinen Schraubendreher
drückt man dazu auf den Schlitz oben. Es öffnet sich dann der Kontakt und die Leitung kann seitlich davon eingeführt werden. Beim Loslassen des Schraubendrehers sitzt die Leitung fest.

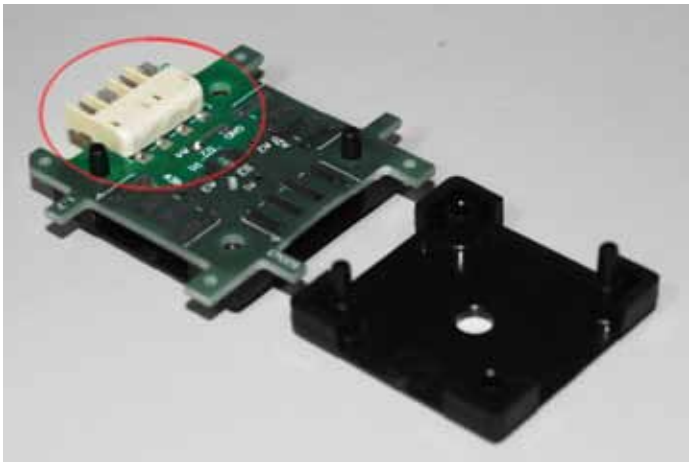


Inkremental Drehgeber. Solche Bausteine werden gerne bei Bedienteilen verwendet. Durch zwei phasenverschobene Kontakte lässt sich z.B. mit einem Prozessor die Drehrichtung bestimmen. Wenn man auf den Knopf drückt wird zusätzlich ein Tastkontakt geschlossen, mit dem man auch eine beliebige Aktion auslösen kann. Je nach Bautyp werden pro Umdrehung ca. 18 bis 36 Kontaktvorgänge ausgelöst.

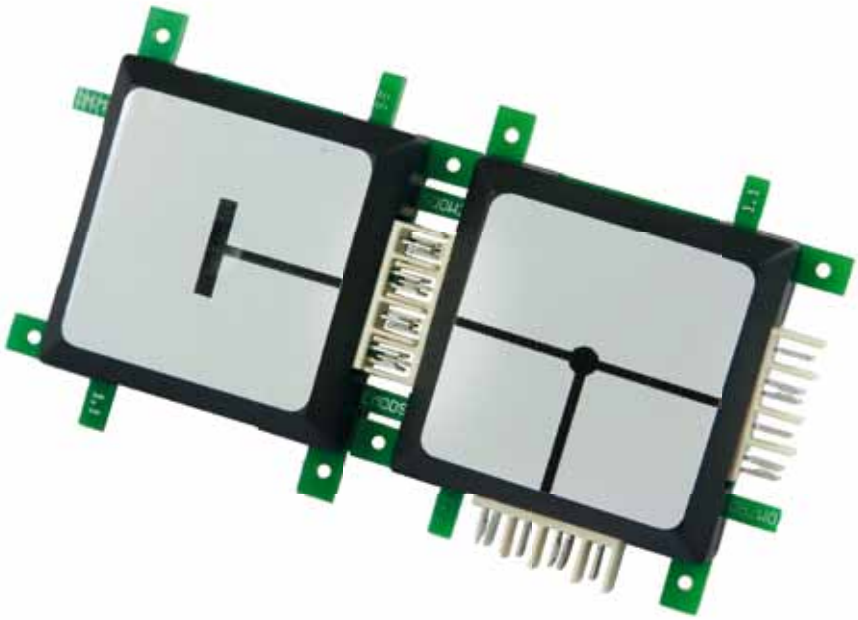
3. Grundlagen

3.1 Das Brick'R'knowledge System

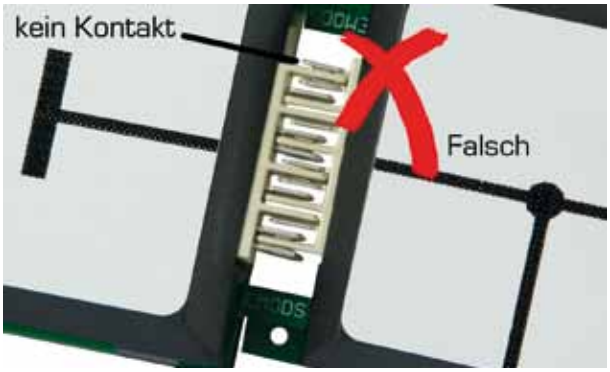
Der Masse-Brick ist ein besonderer Baustein des Brick'R'knowledge Systems. Er spart zusätzliche Verbindungen mit Hilfe anderer Bricks oder Leitungen. Hier wird das Geheimnis unserer vierpoligen Verbinder offenbart. Die mittleren zwei Kontakte sind für die Signalübertragung reserviert, so wie es der Aufdruck verrät. Die äußeren Kontakte werden zum Schließen des Stromkreises, also der Rückführung des Stromflusses zur Spannungsquelle benutzt. Das realisiert der Masse-Brick. Dieser Brick heißt deshalb Masse-Brick, weil in der Elektronik mit der Bezeichnung „Masse" nicht das träge Gewicht des Gegenstandes selbst beschrieben ist, sondern das Vergleichspotential zu dem alle anderen Potentiale bezeichnet sind. Unser Masse-Brick stellt also genau diese Verbindung zu 0V her.
In unserer Schaltung sind das 9 Volt gegenüber 0 Volt: Man spricht einfach nur „Neun Volt". Man erstellt in der Elektronik Schaltungen so, dass nachdem alle Bauelemente in ihrer Funktionsweise in die mehr oder weniger komplexen Stromkreise eingebracht sind, diese mit der „Masse" verbunden werden. Schaltpläne sind nur so zu lesen.
Unser Masse-Brick verbindet die beiden mittleren Kontakte mit den beiden äußeren. Wir verursachen damit keinen Kurzschluss, denn der Strom durchfließt noch die Bauelemente im Inneren der Bricks.



Beim Zusammenstecken der Bricks muss darauf geachtet werden, dass sich die Kontakte richtig berühren, da sonst die Gefahr von Unterbrechungen oder sogar Kurzschlüssen besteht!



Hier ist ein Beispiel einer richtig gesteckten Verbindung. Die Verbindung besteht jeweils aus kleinen Stiften, die sich mechanisch verklemmen und dabei ebenfalls elektrisch leiten. Um eine Isolation zwischen den Kontakten zu gewährleisten und einen Kurzschluss zu verhindern sind Stege aus Kunststoff eingebracht, die den elektrischen Strom nicht leiten.



Ein Beispiel einer fehlerhaften Verbindung ist in diesem Bild zu sehen. Hier sind noch Abstände zwischen den Kontakten, die einen sicheren Stromfluss nicht gewährleisten können. Der Stromkreis bleibt „offen“ oder ist instabil und die Funktion der Schaltung nicht gegeben.

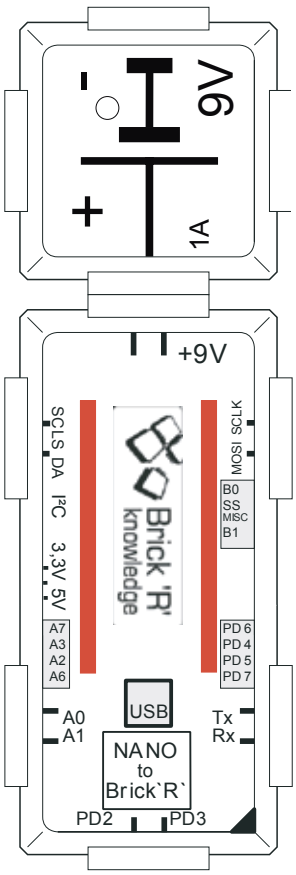
Achtung: Es ist wichtig, grundsätzlich immer den richtigen Sitz der Kontaktstifte zu kontrollieren. Weichen diese zu weit voneinander ab, kann es zu einem Kurzschluss kommen. Dann findet der Stromfluss nicht durch unsere Bauelemente mit der erhofften Wirkung statt, sondern sucht sich den kürzesten Weg zurück zur Spannungsquelle. Ein Kurzschluss führt zum Maximalstromfluss, da der einzige Widerstand der den elektrischen Strom überwinden muss, der Innen-Widerstand der Spannungsquelle ist. Dieser Widerstand ist anschaulich sehr klein, so kann der Kurzschlussstrom bei längerer Dauer zur Überhitzung führen. Es besteht Brandgefahr!

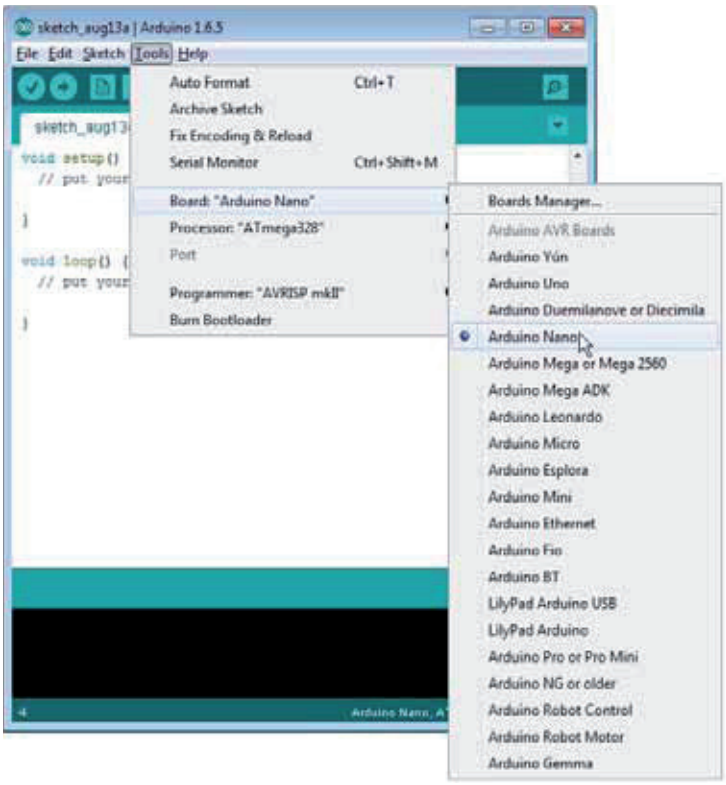
In unserem Set gibt es noch eine Besonderheit: Manche Bricks haben auch Kontakte auf der Unterseite. Hier muss man beim Zusammenstecken besondere Vorsicht walten lassen, eine Schutznase verhindert auch, dass man die Bricks von oben stecken kann, um Kurzschlüsse beim Steckvorgang zu vermeiden. Die Kontakte unten ermöglichen vier weitere Signale gleichzeitig zu verbreiten. Besonders beim Arduino Nano wird von diesen Kontakten intenstiv Gebrauch gemacht.

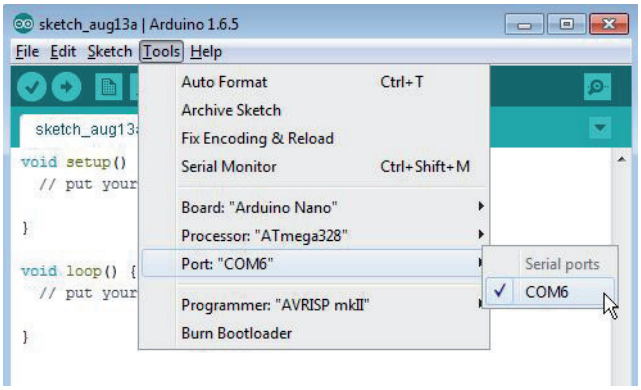
Wichtig: Immer die richtige Stellung der Kontakte überprüfen!

3.2 Arduino Nano Brick - Erste Schritte

Den Ardunio Nano kann man entweder über den USB-Port oder durch eine extra Versorgungsspannungsquelle mit der benötigten Energie versorgen. Dafür eignet sich das beiliegende Netzteil. Achtung den 9V Block niemals an die anderen Kontakte des Arduino Nano-Bricks anschließen, dies könnte den Baustein zerstören.
Der Arduino Nano wird bereits vom Hersteller mit einem Default Programm versehen, deshalb wird wahrscheinlich eine LED auf dem Baustein blinken. Zum Einspielen eines eigenen Programms muss der Brick über den USB-Anschluss des Nano-Bricks mit dem PC verbunden werden, um die Programmierumgebung zu erreichen. Anschließend wird im Board Manager „Arduino Nano“ als zu verwendendes Gerät ausgewählt und der gewünschte Port festgelegt.



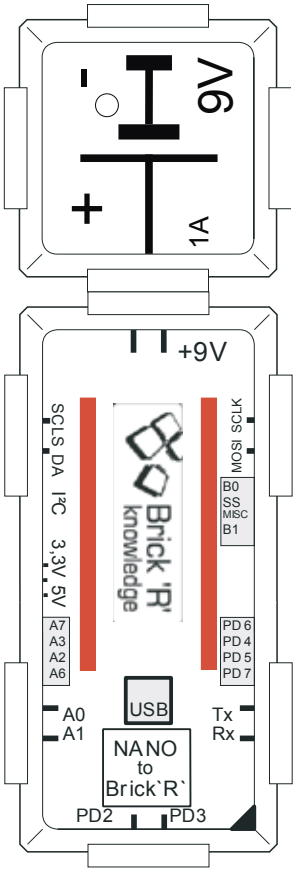




4. LEDs & Tasten

4.1 On Board LEDs

Der Arduino Nano besitzt eine Vielzahl von Ein- Ausgabepins, einige davon sind bereits auf der Nano Platine verdrahtet. Die LED auf dem Arduino ist an Port 13 verdrahtet (=PB5). In unserem einfachen Beispiel werden wir sie blinken lassen.



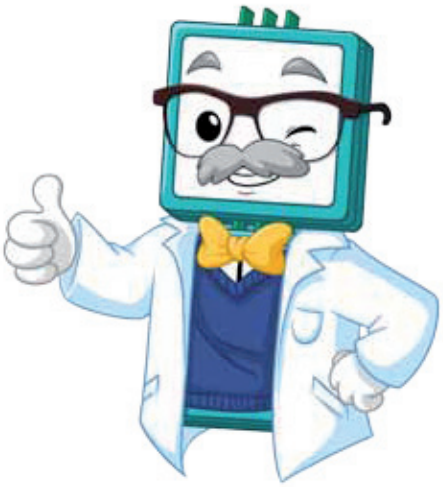

```
// DE_1    -- Nummerierung der Programme 1..xxx DE=Deutsch
#define PORTLED 13 // damit definiert man ein Symbol mit dem Wert 13

// wird nur einmal bei Start des Arduinos ausgefuehrt
void setup() {
  pinMode(PORTLED,OUTPUT); // Port 13 als Ausgang schalten
}

// Arduino fuehrt die Schleife wiederholt aus.
void loop() {
  // Ausgang auf hohen Pegel schalten
  digitalWrite(PORTLED,HIGH); // High bedeutet hohen Pegel
  delay(1000); // 1 Sekunde Verzoegerung = 1000 ilisekunden
  digitalWrite(PORTLED,LOW); // Ausgang auf niedrigen Pegel schal-
ten
  delay(1000); // Eine weitere Sekunde warTen
} // Schleifenende
```

Was passiert? Die rote LED auf der Platine des NANO sollte im Sekundentakt blinken wenn alles geklappt hat.
Die Zeit kann man durch den Parameter beim delay() entsprechend anpassen.
Unten eine Tabelle der Zuordnung aller IO Ports 0-13 (logische Nummern) und den physikalischen Anschlüssen die in Port D und B des Arduino Prozessors aufgeteilt sind.

IO Zuordnung
PD0 - 0
PD1 - 1
PD2 - 2
PD3 - 3
PD4 - 4
PD5 - 5
PD6 - 6
PD7 - 7

PB0 - 8
PB1 - 9
PB2 - 10
PB3 - 11
PB4 - 12
PB5 - 13

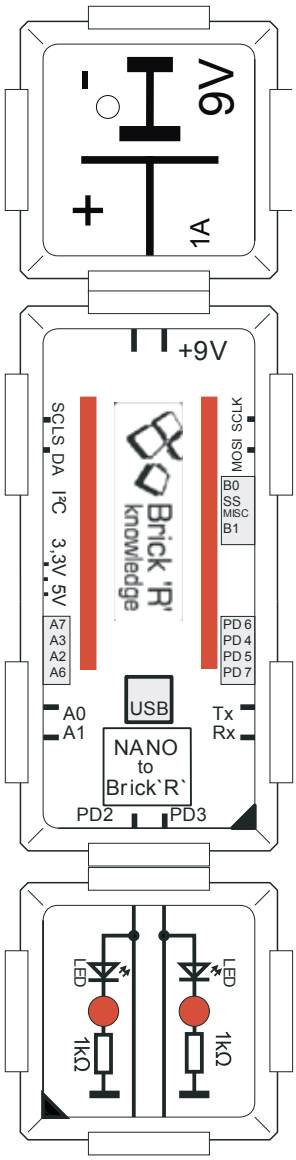
Einige der Ports sind auf der Unterseite der Brick-Platine. Man erreicht sie mit unseren Spezial-Bricks, die Leitungen nach oben bringen können. PD2 und PD3 sind zum Beispiel direkt auf der Oberseite erreichbar, PB0 und PB1 auf der Unterseite. Die anderen Signale sind mehrfach belegt und können unterschiedliche Bedeutung haben wie unten gezeigt:



```
Portbelegung
Kürzel      Port  Parameter
SCK         PB5   #13
MOSI        PB3   #11
B1          PB1   #9
SS          PB2   #10
MISO        PB4   #12
B0          PB0   #8
```

4.2 Doppelte LED

Nun schliessen wir einen unserer Bricks direkt an den NANO an. Dazu gibt es im Set einen Baustein mit zwei Leuchtdioden. Der NANO-Brick verwendet neben den üblichen Masseanschlüssen (die äußeren Kontakte des Brick-Bausteins) die beiden inneren Kontakte getrennt. Es gibt beim diesem Brick auch Kontakte die unten liegen, doch dazu später mehr. Die beiden LEDs sind mit den Portausgängen PD2 und PD3 verbunden, dies entspicht auch der Nummerierung 2 und 3 im Sketch. Ist der Nano-Brick noch über den USB-Port mit dem PC verbunden, kann der Netzteil-Brick auch weggelassen werden. Die LEDs blinken hier abwechseln, da immer paarweise 2 auf high und 3 auf low und dannach 2 auf low und 3 auf high gesetzt werden. Die LEDs unterscheiden sich je nach Baustein in der Farbe.

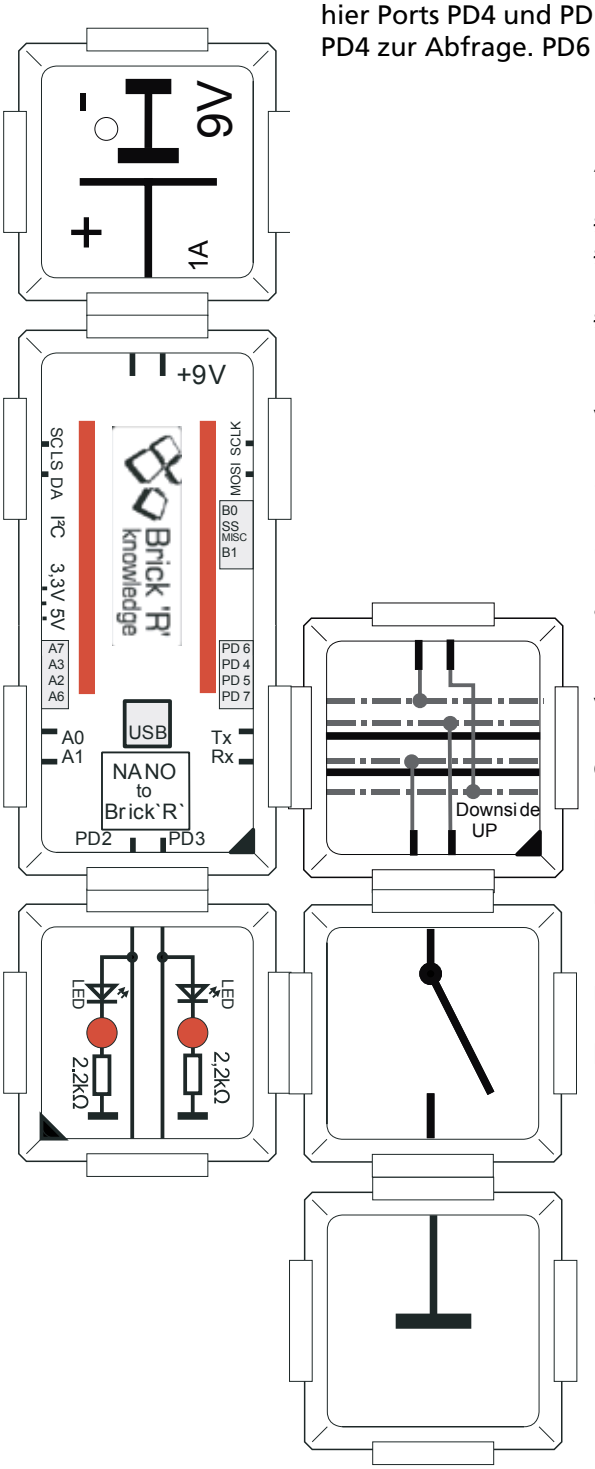



```
// DE_2 Doppelte LED

#define PORTLED2 2 // definiert das Symbol PORTLED2 mit 2
#define PORTLED3 3 // entsprechend fuer PORTLED3

// Ausführen am Anfang
void setup() {
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
}

// Schleife wird wiederholt ausgefuehrt:
void loop() {
  digitalWrite(PORTLED2,HIGH); // Ausgang auf hohen Pegel schalten
  digitalWrite(PORTLED3,LOW); // Ausgang auf niedrigen Pegel
schalten
  delay(1000); // Eine weitere Sekunde warten (=1000 ms)
  digitalWrite(PORTLED2,LOW); // Ausgang auf niedrigen Pegel
schalten
  digitalWrite(PORTLED3,HIGH); // Ausgang auf hohen Pegel schalten
  delay(1000); // Eine weitere Sekunde warten
}
```

Was passiert? Die beiden LEDs blinken im Sekundentakt abwechselnd.

4.3 Taste & LED

Hier haben wir zusätzlich eine Taste eingebaut. Sie schließt einen Port auf 0V wenn man die Taste drückt. Bei offener Taste, wäre der Zustand des Eingangs undefiniert. Man kann da z.B. einen Pullup Widerstand an +5V (nicht 9V!) schalten, oder was einfacher ist, es gibt beim Arduino Sketch einen INPUT_PULLUP Modus, der bewirkt, dass ein im Prozessor vorhandener 20 kOhm Widerstand nach +5V geschaltet wird, und wir brauchen nicht soviel Aufwand bei der Beschaltung, wie ohne einen solchen Befehl. Wenn man die Taste drückt, blinken die LEDs abwechselnd, beim Loslassen wird ein Blinkzyklus noch abgeschlossen, dann bleibt der Status erhalten. Den Tasten-Brick kann man nicht ohne weiteres direkt den freien Port 4 bzw 5 anschliessen, er würde normalerweise nur die Kontakte auf der Oberseite kontaktieren. Dazu gibt es unseren Spezialbrick: Er verbindet die Signale der Unterseite des Nano-Bricks auf die Oberseite entsprechend dem Schaltsymbol. Die Taste überbrückt hier Ports PD4 und PD5 gleichzeitig, was aber nicht weiter stört. Wir verwenden Port PD4 zur Abfrage. PD6 und PD7 werden hier nach oben weiter geführt.

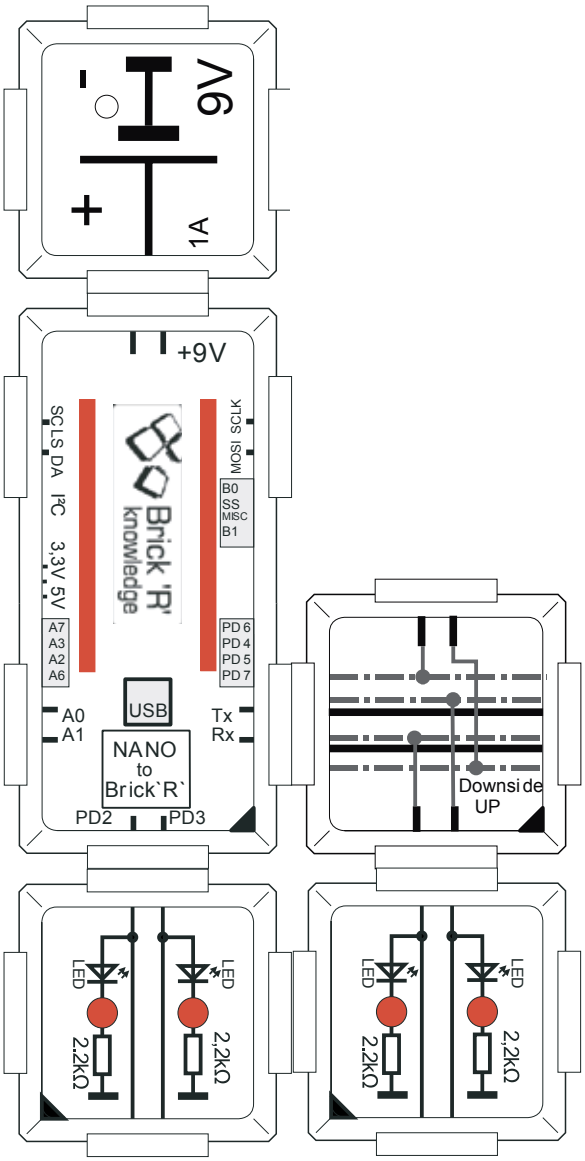



```
// DE_3 Taste & LED

#define PORTLED2 2 // Ausgang wird Port 2
#define PORTLED3 3 // und Port 3 dort die LED

#define PORTSWTTCH4 4 // SChalter an Port 4 (und
5)

void setup() {
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang
schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang
schalten
  pinMode(PORTSWTTCH4,INPUT_PULLUP); // Port 4
als Eingang mit internem Pull Up
}

void loop() {
  if (digitalRead(PORTSWTTCH4)==LOW) { // nur bei
gedrückter Taste
    digitalWrite(PORTLED2,HIGH); // Ausgang auf
hohen Pegel schalten
    digitalWrite(PORTLED3,LOW); // Ausgang auf
niedrigen Pegel schalten
    delay(1000); // Eine weitere Sekunde warten
    digitalWrite(PORTLED2,LOW); // Ausgang auf
niedrigen Pegel schalten
    digitalWrite(PORTLED3,HIGH); // Ausgang auf
hohen Pegel schalten
    delay(1000); // Eine weitere Sekunde warten
  } // ende Abfrage fuer gedrueckte Taste
} // Ende der Schleife
```

Was passiert? Die beiden LEDs blinken im Sekundentakt abwechselnd, während man die Taste gedrückt hält.

4.4 Mehrere LEDs binär blinken lassen

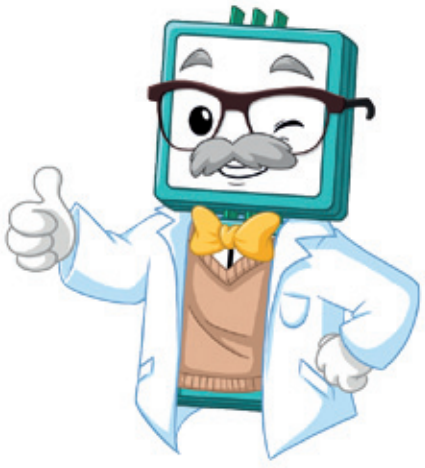
Genaugenommen zeigen die LEDs einen Binärcode an. Dazu wird im Programm ein Zähler von 0 bis 15 verwendet. Die einzelnen Bits werden abgefragt und den LEDs dann entsprechend zugeordnet. Zu beachten ist, dass die LEDs in der Reihenfolge 2, 3, 5, 4 liegen, was im Code auch entsprechend zugeordnet wird.
Im Code ist die Variable „counter" als static deklariert, damit der Inhalt zwischen zwei Schleifen-Durchläufen erhalten bleibt, wie bei einer globalen Variable.
Der Binärcode ist unten nochmal dargestellt. Der Zähler wird im Programm durch counter = counter + 1 realisiert. Nun käme es so zu einem Überlauf, da entweder 16 oder 32 Bit zur Verfügung stehen, je nach Implementierung. Mit dem &-Symbol wird eine logische UND-Verknüpfung erreicht, mit dem Wert 0xf ist der dezimale Wert 15 gemeint, dieser bewirkt, dass der maximale Wert, der dem Counter zugewiesen werden kann, 15 ist.



Dezimal	Binär
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

```
// DE_4 Mehrere LEDs binaer blinken lassen

#define PORTLED2 2 // Definieren nun 4 LEDs
#define PORTLED3 3 // fuer  PD2,PD3,PD4,PD5
#define PORTLED4 4 //
#define PORTLED5 5

void setup() { // Startet die Initialisierung
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
  pinMode(PORTLED4,OUTPUT); // Port 4 als Ausgang schalten
  pinMode(PORTLED5,OUTPUT); // Port 5 als Ausgang schalten
}

void loop() {
  static int counter = 0; // der Zaehlerstand bleibt erhalten
    // da eine statische Variable verwendet wird
    // Ohne static waere der Wert bei jedem Schleifendurchlauf
    // auf 0 gesetzt
    if (counter & 1) { // Bit 0 pruefen
     digitalWrite(PORTLED4,HIGH); // rechte LED
    } else digitalWrite(PORTLED4,LOW); //
    if (counter & 2) { // Bit 1 pruefen
     digitalWrite(PORTLED5,HIGH); // linke LED
    } else digitalWrite(PORTLED5,LOW); // naechste
    if (counter & 4) { // Bit 2 pruefen
     digitalWrite(PORTLED3,HIGH); //  // auf 1 setzen
    } else digitalWrite(PORTLED3,LOW); // Reihenfolge !
    if (counter & 8) { // Bit 3 pruefen
     digitalWrite(PORTLED2,HIGH); // Dann erst LED 2 bei Bit 4
    } else digitalWrite(PORTLED2,LOW); //
    delay(1000); // Eine weitere Sekunde warten = 1000ms
    counter = (counter + 1)& 0xf; // 0..15 Zaehler dann wieder auf 0 durch
Maske
} // Schleifenende
```

Was passiert? Die vier LEDs zählen im Sekundentakt binär von 0000 (alle aus), dann 0001, 0010, 0011 bis 1111 (alle an).



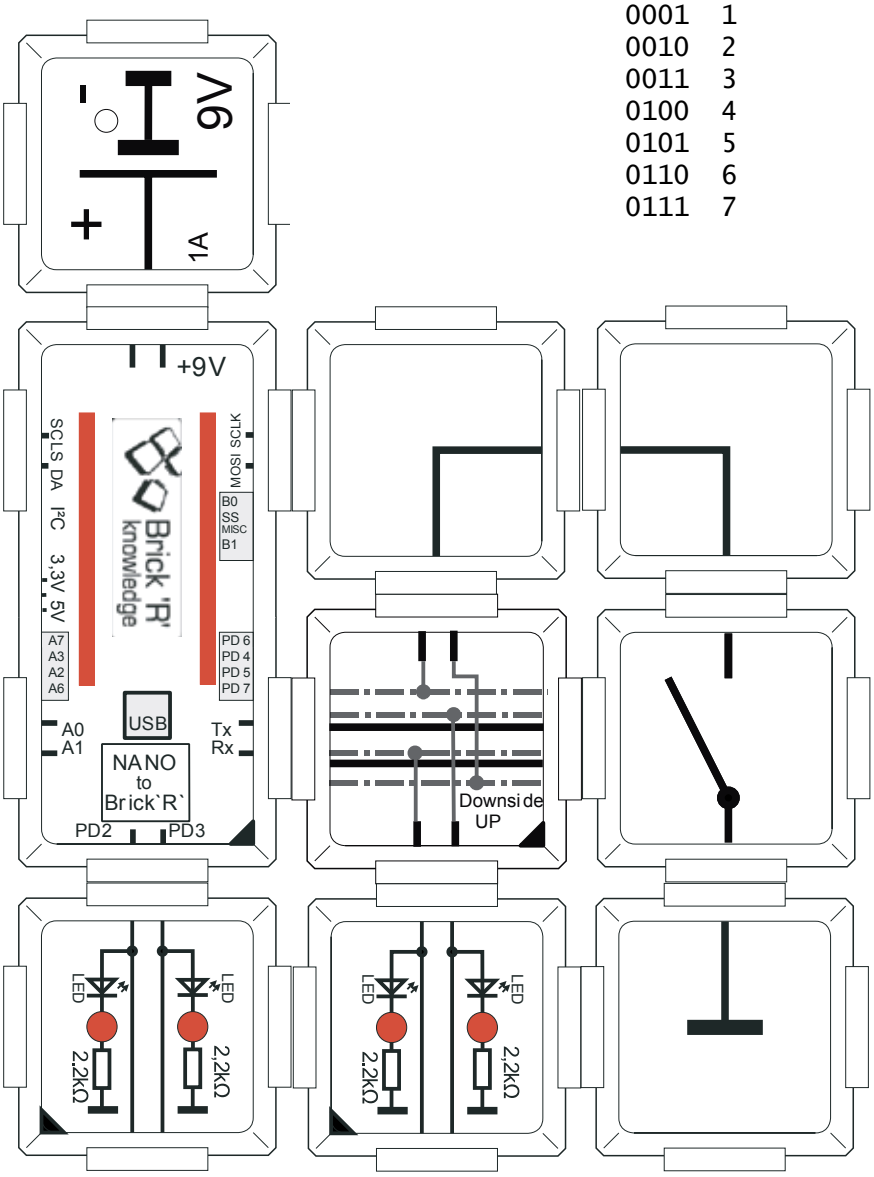
4.5 Binäres Zählen - durch Taste gesteuert

Das Zählen geschieht im Sekundentakt. Man kann nun eine primitive Stoppuhr bauen, indem man einen Taster zur Freigabe des Zählvorgangs verwendet.

Unten noch eine kleine Tabelle, die beim Umrechnen hilft. Wir werden später noch eine komfortablere Stoppuhr mit dezimaler Ausgabe bauen.
Die Taste ist diesmal mit Port 6 und 7 verbunden, wir verwenden hier Port 7, der „Kurzschluss“ zu Port 6 stört nicht weiter, da er auch als Eingang geschaltet ist (ohne Pullup).
Der Counter zählt nun solange, wie die Taste gedrückt gehalten wird.

Tabelle Umrechnung Anzeige nach Dezimal:

binär	dezimal	binär	dezimal
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

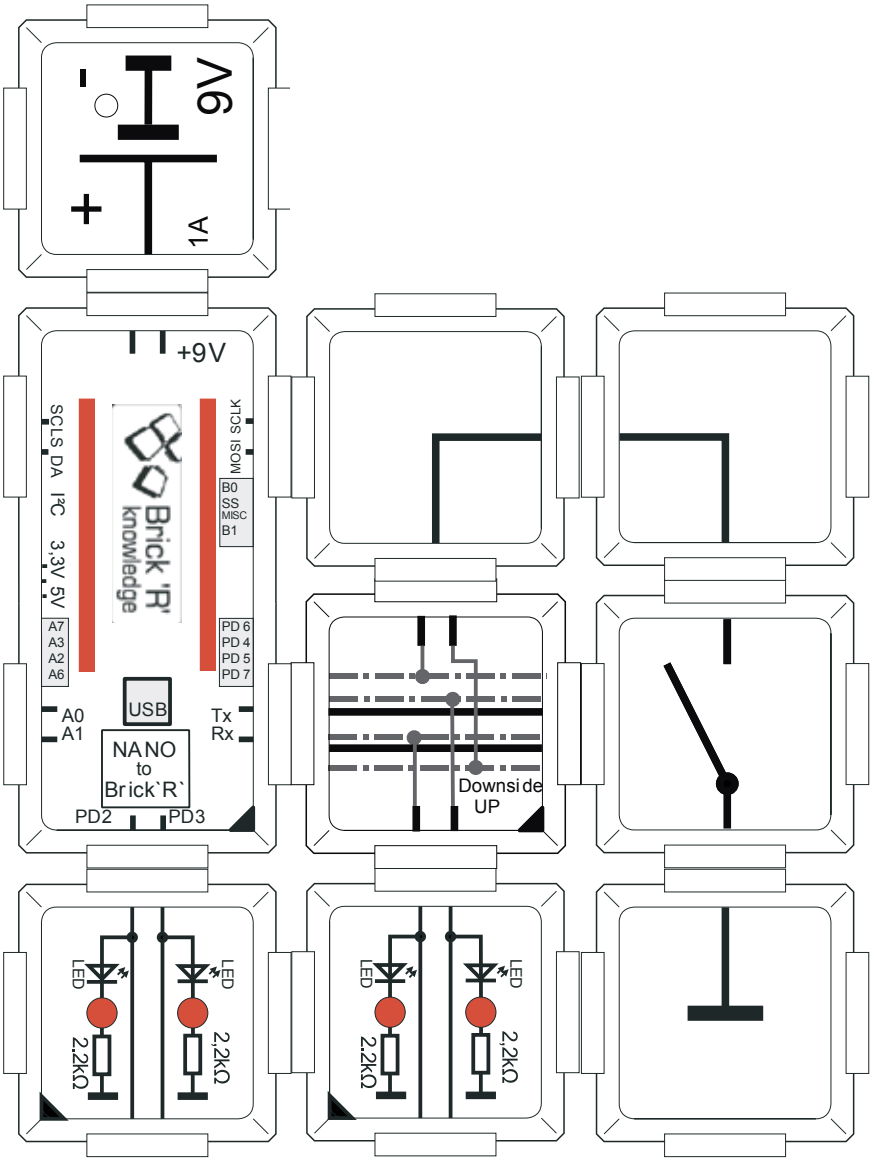


```
// DE_5 Binaeres Zaehlen wird durch Taste gesteuert

#define PORTLED2 2 // LEDs an PD2..PD5
#define PORTLED3 3 // damit Zuordnung
#define PORTLED4 4 // definieren
#define PORTLED5 5

#define SWITCH7 7 // Die Taste diesmal an Port PD7

void setup() { // alle Ein- und Ausgaenge hier definieren
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
  pinMode(PORTLED4,OUTPUT); // Port 4 als Ausgang schalten
  pinMode(PORTLED5,OUTPUT); // Port 5 als Ausgang schalten
  pinMode(SWITCH7,INPUT_PULLUP); // Port 7 als Eingang mit Pullup
  // Ein Pullup ist eine Art interner Widerstand. Wenn man die Taste nicht drueckt
  // ist der Eingang normalerweise offen und nicht definiert. Durch den Pullup
  // liefert ist er High und liefert ein 1-Signal
}

void loop() {
  static int counter = 0; // wichtig wieder Definition als Static
    if (counter & 1) { // Bit 0 pruefen 0..3 nacheinander
     digitalWrite(PORTLED4,HIGH); // rechte LED
    } else digitalWrite(PORTLED4,LOW); //
    if (counter & 2) { // Bit 1 pruefen
     digitalWrite(PORTLED5,HIGH); // linke LED
    } else digitalWrite(PORTLED5,LOW); //
    if (counter & 4) { // Bit 2 pruefen
     digitalWrite(PORTLED3,HIGH); //
    } else digitalWrite(PORTLED3,LOW); //
    if (counter & 8) { // Bit 3 pruefen
     digitalWrite(PORTLED2,HIGH); //
    } else digitalWrite(PORTLED2,LOW); //
    // Nun die STOppzeit definieren
    delay(1000); // Eine weitere Sekunde warten =1000ms
    // *** Tastenabfrage hier:
    if (digitalRead(SWITCH7) == LOW) { // zaehlen solange gedrueckt
      counter = (counter + 1) & 0xf; // 0..15 Zaehler im Kreis
    } // nur wenn gedrueckt sonst nicht zaehlen
} // Schleifenende
```

Was passiert? Wenn man die Taste gedrückt hält, dann und nur dann zählen die vier LEDs im Sekundentakt binär von 0000 (alle aus), dann 0001, 0010, 0011 bis 1111 (alle an). Bei losgelassener Taste bleibt der letzte Anzeigezustand erhalten.

4.6 Zählgeschwindigkeit steuern

Nun kann man mit der Taste auch noch mehr tun, zum Beispiel die Geschwindigkeit des Zählvorgangs einstellen. Die Änderung im Programmcode ist minimal. Anstatt den Zähler freizugeben, wird nun der delay() geändert. Hier als Beispiel zwischen schnellem Zählen mit 200ms und dem Zählen mit 1000ms = 1sec.
Insgesamt ist es wichtig, bei dieser Vorgehensweise zu wissen, dass die Zählgeschwindigkeit nicht ganz genau ist. Der Delay verzögert die Ausführung um die vorgegebene Zeit, i.A. quarzgenau, aber die restlichen Befehle bis zum nächsten delay() müssen auch berücksichtigt werden. Will man ein exakteres Ergebnis, muss eine andere Methoden gewählt werden. Der Quarz auf dem Prozessor des NANO gibt den Takt an. Auch er selbst ist nicht ganz exakt, aber viel genauer als unser Abfrage-Verfahren.

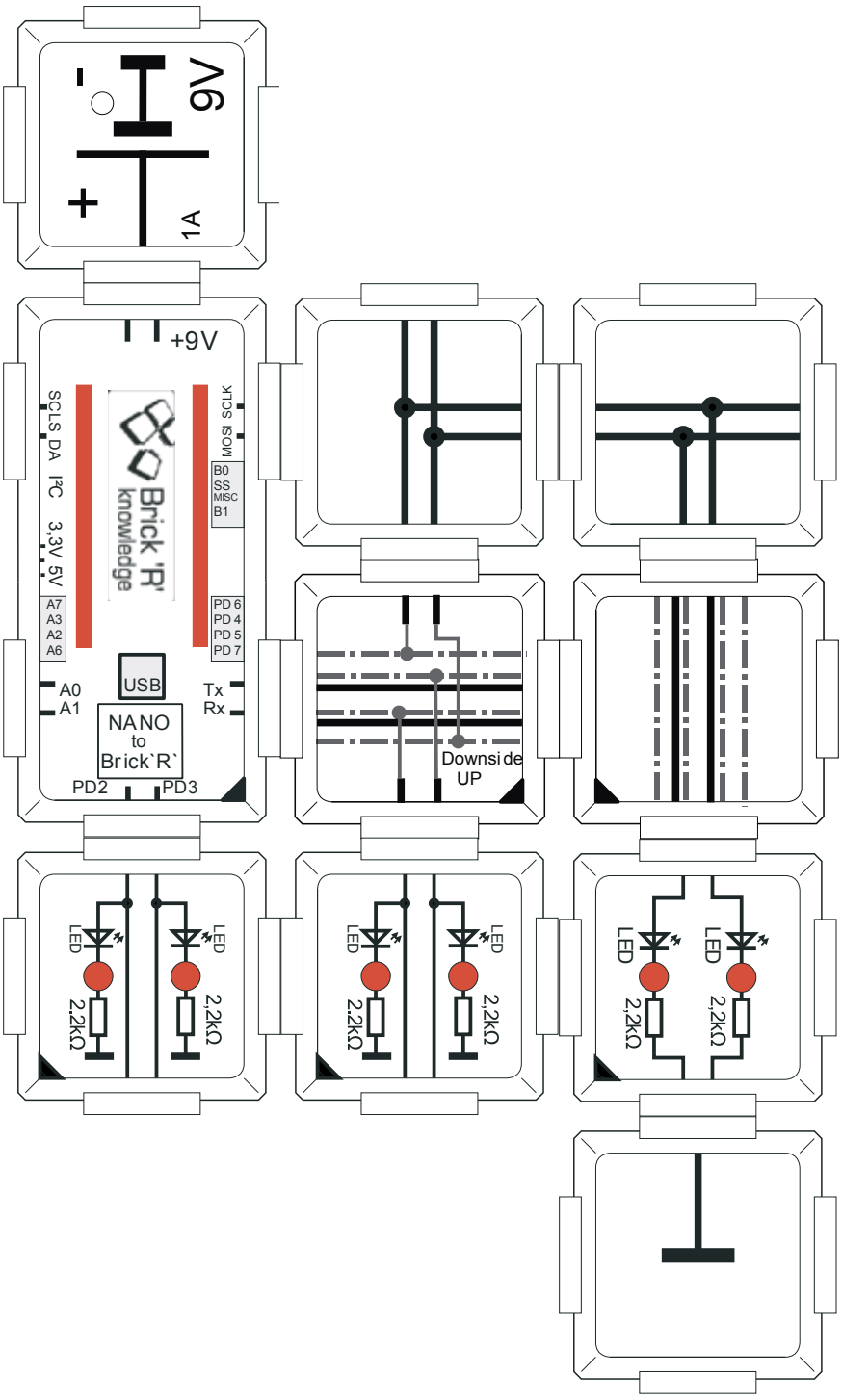


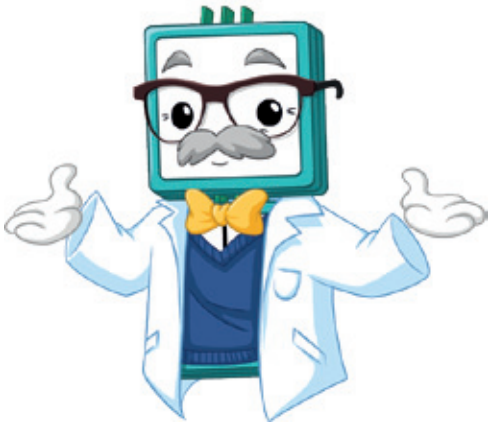
```
// DE_6 Zaehlgeschwindigkeit einstellen

#define PORTLED2 2 // PORTs PD2-PD5
#define PORTLED3 3 // An symbolische
#define PORTLED4 4 // Namen zuordnen
#define PORTLED5 5

#define SWITCH7 7 // Taster an PD7

void setup() { // IO definieren
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
  pinMode(PORTLED4,OUTPUT); // Port 4 als Ausgang schalten
  pinMode(PORTLED5,OUTPUT); // Port 5 als Ausgang schalten
  pinMode(SWITCH7,INPUT_PULLUP); // Port 7 als Eingang mit Pullup
}

void loop() {
  static int counter = 0; // whichtig muss static sein
  if (counter & 1) { // Bit 0 pruefen dann 1,2,3
   digitalWrite(PORTLED4,HIGH); // rechte LED
  } else digitalWrite(PORTLED4,LOW); //
  if (counter & 2) { // Bit 1 pruefen
   digitalWrite(PORTLED5,HIGH); // linke LED
  } else digitalWrite(PORTLED5,LOW); //
  if (counter & 4) { // Bit 2 pruefen
   digitalWrite(PORTLED3,HIGH); //
  } else digitalWrite(PORTLED3,LOW); //
  if (counter & 8) { // Bit 3 pruefen
   digitalWrite(PORTLED2,HIGH); //
  } else digitalWrite(PORTLED2,LOW); //
  //
  if (digitalRead(SWITCH7) == LOW) { // schnell wenn gedrueckt
    delay(200); // 200 ms warten = 200Milisekunden
  } else { // Sonst langsamer im Sekundentakt
    delay(1000); // 1000 ms warten
  } // Dann weiter
  // es wird immer gezaehlt.
  counter = (counter + 1)& 0xf; // 0..15 Zaehler
}
```

Was passiert? Die vier LEDs zählen bei gedrückter Taste alle 200ms sonst im Sekundentakt binär von 0000 (alle aus), dann 0001, 0010, 0011 bis 1111 (alle an).

4.7 Einfaches Lauflicht

Unsere Schaltung kann auch zu einem Lauflicht ausgebaut werden. Dazu wird hier noch ein weiterer Doppel-LED-Brick dazugesetzt. Hier kommt noch ein Spezialbaustein zum Einsatz, der auch an der Unterseite Kontakte hat. Diese bleiben aber in diesem Fall ungenutzt und dienen nur zur Weiterleitung der beiden LED Signale von Port 6 und 7 auf der Oberseite.
Man beachte die Zuordnung von Port 6 und 7 über die Leitung. Wird das Doppel-T rechts anders eingebaut, kann sich die Richtung des Lauflichts für die beiden LED 6 und 7 umgekehren! Man beachte, dass LED 6 dem Bit 0 zugeordnet ist und LED 7 dem Bit 1. Dem Leser bleibt es hier überlassen, andere Reihenfolgen auszuprobieren.





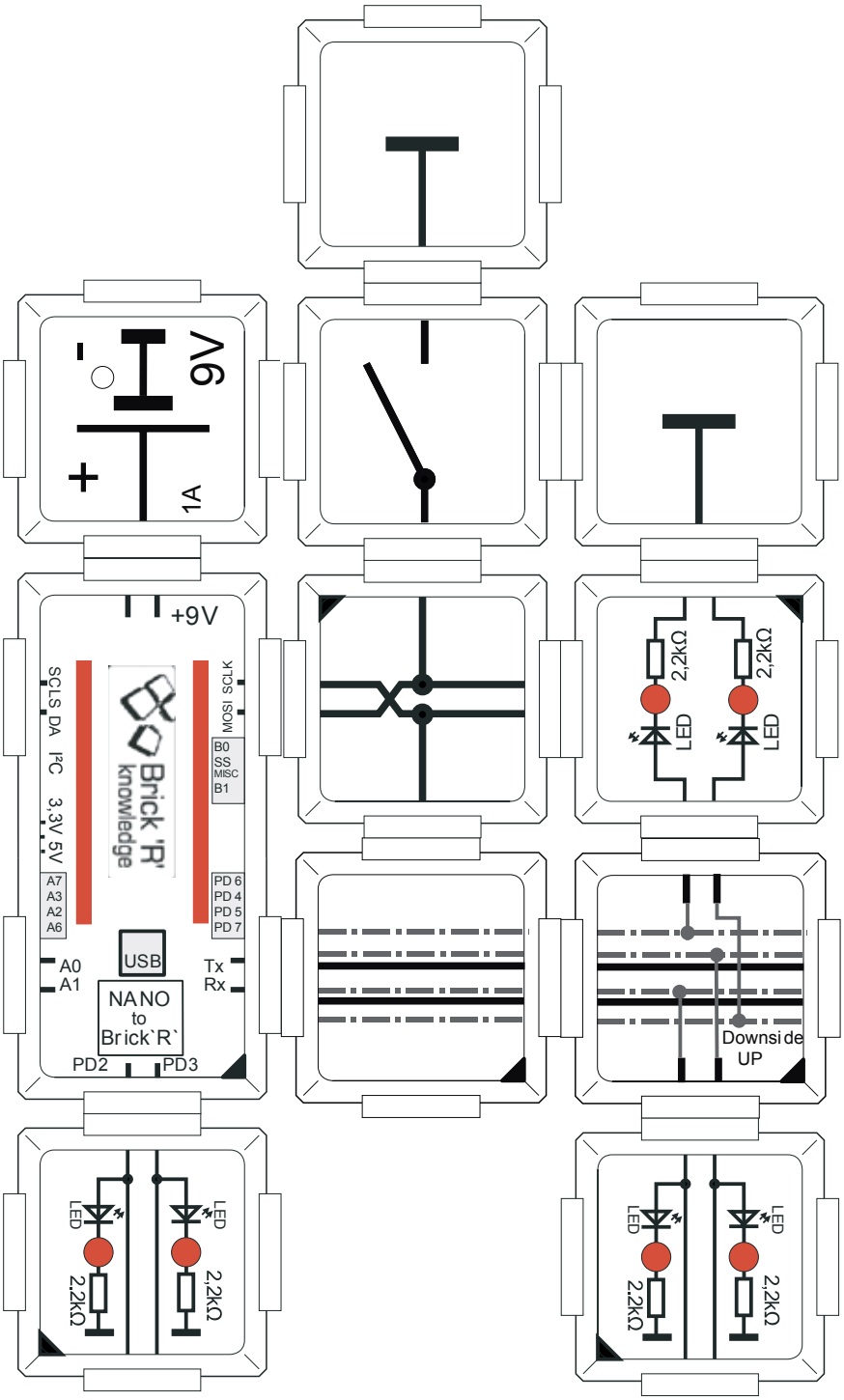
```
// DE_7 Lauflicht mit 6 LEDs

#define PORTLED2 2 // Alle LEDs definieren
#define PORTLED3 3 // Ports PD2 bis PD7
#define PORTLED4 4 // werden verwendet
#define PORTLED5 5 // Beim Brick auf die
#define PORTLED6 6 // genaue Zuordnung achten
#define PORTLED7 7

void setup() { // IO definieren
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
  pinMode(PORTLED4,OUTPUT); // Port 4 als Ausgang schalten
  pinMode(PORTLED5,OUTPUT); // Port 5 als Ausgang schalten
  pinMode(PORTLED6,OUTPUT); // Port 6 als Ausgang schalten
  pinMode(PORTLED7,OUTPUT); // Port 7 als Ausgang schalten
}

void loop() {
  static int shiftreg = 1; // Lauflicht Startbit 6 Bits werden verwendet
    if (shiftreg & 1) { // Bit 0 pruefen bis Bit 5 diesmal
     digitalWrite(PORTLED6,HIGH); // rechte LED
    } else digitalWrite(PORTLED6,LOW); //
    if (shiftreg & 2) { // Bit 1 pruefen
     digitalWrite(PORTLED7,HIGH); // linke LED
    } else digitalWrite(PORTLED7,LOW); //
    if (shiftreg & 4) { // Bit 2 pruefen
     digitalWrite(PORTLED4,HIGH); //
    } else digitalWrite(PORTLED4,LOW); //
    if (shiftreg & 8) { // Bit 3 pruefen
     digitalWrite(PORTLED5,HIGH); //
    } else digitalWrite(PORTLED5,LOW); //
    if (shiftreg & 0x10) { // Bit 4 pruefen
     digitalWrite(PORTLED3,HIGH); //
    } else digitalWrite(PORTLED3,LOW); //
    if (shiftreg & 0x20) { // Bit 5 pruefen
     digitalWrite(PORTLED2,HIGH); //
    } else digitalWrite(PORTLED2,LOW); //
    //
    delay(300); // 300 ms warten definiert die Geschwindigkeit
    shiftreg = shiftreg << 1; // 1,2,4,8,16,32  Schiebebefehl C-Compiler
    if (shiftreg > 32) shiftreg = 1; // dann wieder von Vorne. Damit es rund laeuft
} // Ende der Schleife
```

Was passiert? Von den 6 LEDs geht immer genau eine an, die andern sind dunkel. Dabei wandert das Licht von rechts nach links durch. Der Wechsel zur nächsten LED geschieht alle 300ms. Danach wiederholt sich der ganze Zyklus.

4.8 Lauflicht Geschwingkeit steuern

Mit einer zusätzlichen Taste soll die Lauflichtgeschwindigkeit umgeschaltet werden können. Dazu verwenden wir einen der B-Ports, der multifunktional belegt ist. Hier bezeichnet als das Signal MOSI, welches von Port B3 kommt und mit #11 als Parameter angesprochen werden kann. MOSI steht für „Master Out Slave In“ und wird eigentlich für die eingebaute SPI Schnittstelle verwendet (bestehend aus MOSI+MISO+SCK). Man kann diese Pins aber ganz normal als IO-Ports verwenden, wie in unserem Beispiel.

Wenn man die Taste gedrückt hält so wird der Lauflichzyklus schneller.



Portbelegung

Kürzel	Port	Parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

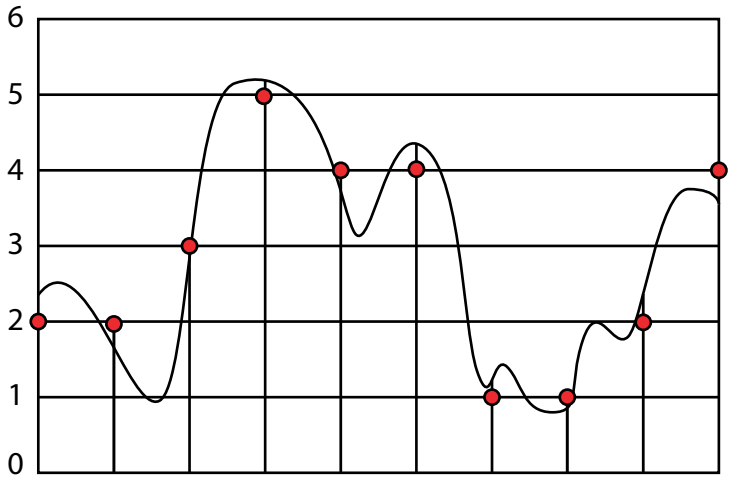
Was passiert? Von den 6 LEDs geht immer genau eine an, die anderen sind dunkel. Dabei wandert das Licht von rechts nach links durch. Der Wechsel zur nächsten LED geschieht nun in Abhängigkeit der Taste unterschiedlich schnell. Entweder beim Drücken alle 50ms sonst alle 300ms. Danach wiederholt sich der ganze Zyklus.

```
// DE_8 Lauflicht mit Taste - Geschwindigkeit steuern

#define PORTLED2 2 // LEDs and POrt PD2-PD7
#define PORTLED3 3
#define PORTLED4 4
#define PORTLED5 5
#define PORTLED6 6
#define PORTLED7 7

#define SWITCHB3MOSI 11 // Nun Port PB3 =#11 als Eingang
// er traegt den Namen MOSI auf unserem Brick da er eine
// doppelte Bedeutung hat.


void setup() { // IOs definieren
  pinMode(PORTLED2,OUTPUT); // Port 2 als Ausgang schalten
  pinMode(PORTLED3,OUTPUT); // Port 3 als Ausgang schalten
  pinMode(PORTLED4,OUTPUT); // Port 4 als Ausgang schalten
  pinMode(PORTLED5,OUTPUT); // Port 5 als Ausgang schalten
  pinMode(PORTLED6,OUTPUT); // Port 6 als Ausgang schalten
  pinMode(PORTLED7,OUTPUT); // Port 7 als Ausgang schalten
  pinMode(SWITCHB3MOSI,INPUT_PULLUP); // MOSI verwenden
}

void loop() {
  static int shiftreg = 1; // Lauflicht Startbit 6 Bits werden verwendet
    if (shiftreg & 1) { // Bit 0 pruefen bis Bit 5
     digitalWrite(PORTLED6,HIGH); // rechte LED
    } else digitalWrite(PORTLED6,LOW); //
    if (shiftreg & 2) { // Bit 1 pruefen
     digitalWrite(PORTLED7,HIGH); // linke LED
    } else digitalWrite(PORTLED7,LOW); //
    if (shiftreg & 4) { // Bit 2 pruefen
     digitalWrite(PORTLED4,HIGH); //
    } else digitalWrite(PORTLED4,LOW); //
    if (shiftreg & 8) { // Bit 3 pruefen
     digitalWrite(PORTLED5,HIGH); //
    } else digitalWrite(PORTLED5,LOW); //
    if (shiftreg & 0x10) { // Bit 4 pruefen
     digitalWrite(PORTLED3,HIGH); //
    } else digitalWrite(PORTLED3,LOW); //
    if (shiftreg & 0x20) { // Bit 5 pruefen
     digitalWrite(PORTLED2,HIGH); //
    } else digitalWrite(PORTLED2,LOW); //
    //
    if (digitalRead(SWITCHB3MOSI)==LOW) { // bzw. PORT PB3 #11
      delay(50); // 50  ms warten also sehr schnell
    } else {
      delay(300); // 300 ms warten langsamer
    }
    shiftreg = shiftreg << 1; // 1,2,4,8,16,32  dann wieder 1,2,3,4...
    if (shiftreg > 32) shiftreg = 1; // dann wieder von Vorne. auf 1 setzen
} // Ende der Schleife
```

5. Analog-Digital Umsetzer

5.1 AD Umsetzer - prinzipieller Aufbau

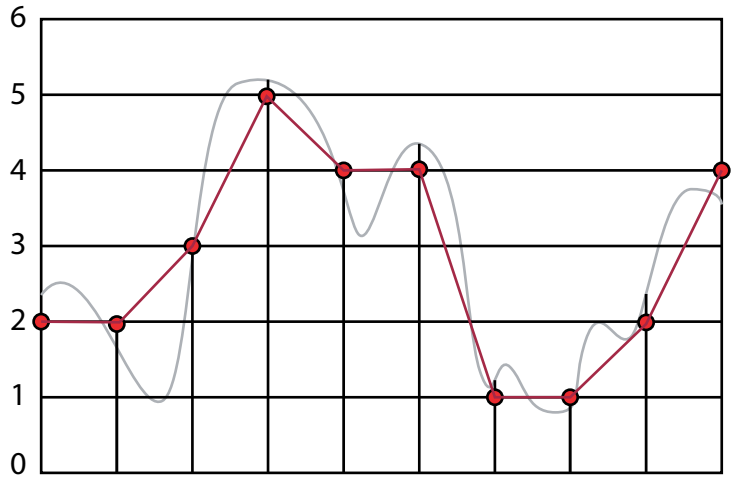
AD steht für Analog - Digital. Ziel ist es, analoge Werte, wie zum Beispiel eine beliebige Spannung, in einen digitalen Wert also Zahlen umzusetzen. Diese Zahlen kann ein Computer dann weiterverarbeiten. Mit analogen Werten kann ein normaler Computer nichts direkt anfangen.
Zwei wichtige Schritte werden dabei durchgeführt, eine Quantisierung der Amplitude im Falle einer Spannung und eine Quantisierung der Zeit bei einem zeitlich veranderbarem Verlauf des analogen Wertes.
Was ist damit gemeint?
Die Quantisierung in der Amplitude kann man leicht verstehen.
Eine analoge Spannung z.B. zwischen 0 und 5V kann jeden beliebigen Wert annehmen. Also 2.3V oder 2.31V oder 2.315 Volt ... usw. Es bleibt die Frage, wo will man mit der Genauigkeit hin oder wieviele Stufen soll es geben. Die Zahlenwerte muss man schließlich für die Verarbeitung in einem digitalen Rechensystem aufbereiten.
Beispiel: Wir wollen einen Spannungsbereich von 0 bis 5V in 6 Stufen digital darstellen. Welchen digitalen Wert wird man dann für 2.1V vergeben? Im Diagramm unten kann man die Zuordnung ablesen. Der Wert 2.1 liegt näher bei 2 als bei 3, so wird man den digitalen Wert 2 wählen. Bei einem Wert von 2.5 kann man 3 als digitalen Wert zuordnen, wenn man die Zahl 2.5 normal rundet.



Hier haben wir den zeitliche Verlauf eines Spannungssignals aufgezeichnet (Hier kann man sich in der y-Achse die Spannung in V aufgetragen denken und in x-Richtung die Zeit z.B. in Sekunden).
Bei der Umwandlung des Signals in eine digitale Zahlenfolgen, wird man einmal pro Sekunde einen Messwert abfragen (senkrechte Linien), und dann die RUndung auf eine Stelle durchführen. Damit ergibt sich dann die folgende Zahlenreihe:

2,2,3,5,4,4,1,1,2,4

Es gibt dabei zwei interessante Effekte. Wir verlieren Information in der Amplitude, die beiden ersten Werte sind jedesmal 2 als Beispiel. Mit einer höheren Auflösung bei der Wandlung, z.B. einer weiteren Nachkommastelle hätte man mehr Information des ursprünglichen Signales erhalten.
Und der zweite Effekt, wir verlieren auch zeitlich relevante Informationen. Die Sequenz 1,1 beinhaltet einen kleinen Schwinger der in der Zahlereihe nicht mehr sichtbar ist. Die Fachleute sagen dazu dass das sogenannte Nyquist-Kriterium verletzt wird, denn man muss mindestens mit der doppelten Frequenz abtasten, die im Signal enthalten ist. Bei den kleinen Schwingungen ist das Prinzip verletzt. Wenn man nun die roten Punkte verbindet, dann erhält man dass für den Computer verwertbare Signal, die ursprüngliche Kurve ist nicht mehr genau rekonstrierbar. Wenn man die Zahl der Abtastpunkte erhöht, wird dass besser.



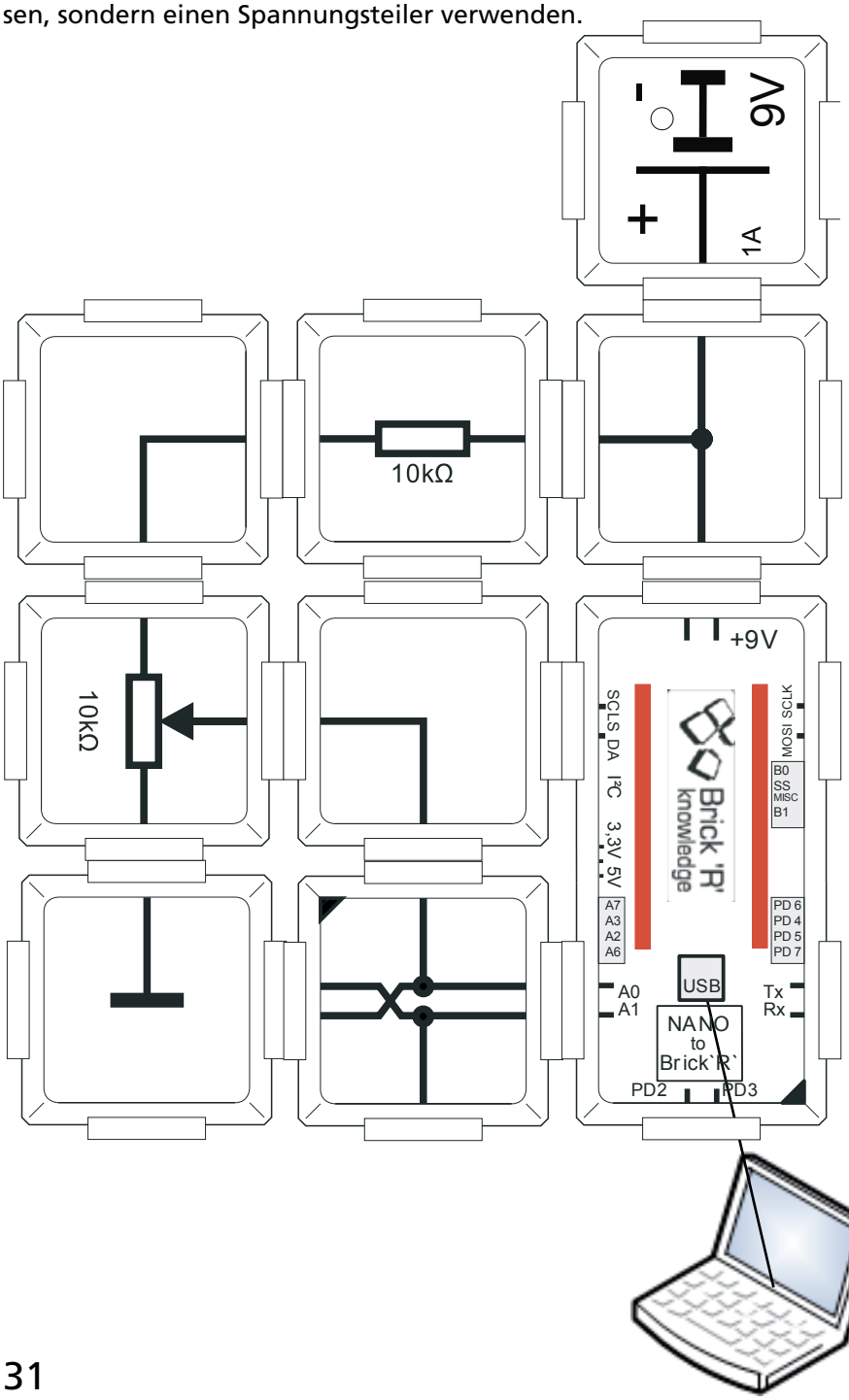
Oben sieht man die rekonstruierte Kurve.
Die Auflösung für die Amplitude wird durch die Anzahl der Bits, die einem Zahlenwert zugeordnet werden bestimmt. Bei dem Analog-Digital-Umsetzer im Arduino NANO sind dies 10 Bit. Das entspricht 2 Hoch 10 Werten, also mathematisch umgerechnet 1024 Stufen. Bei einem Spannungsbereich von 0 bis 5V entspricht die kleinste Stufe einer Spannung von 5V / 1024 = 4.88mV.
Der Techniker interessiert sich auch für den sogenannten Dynamikbereich, den berechnet man aus dem Dynamic Range = 20 * LOG (Anzahl der Stufen) in dB.

Bei uns = 20*LOG(1024) = 60.2 dB. Dies ist ein ganz guter Wert. Das menschliche Gehört hat aber eine höhere Dynamik, daher muss man Audiosignale für eine HIFI Qualität auch mit mehr Bits digitalisieren, zum Beispiel bei 24 Bit ergeben sich: 20 * LOG (2Hoch24) = 20 * LOG (16777216)=144.5 dB.
Das menschliche Gehör kann in etwa 120 dB in bestimmten Hörbereichen wahrnehmen.

Wie kann man nun solche Signale in digitale umwandeln. Dazu gibt es unterschiedliche Verfahren, die den Rahmen hier schnell sprengen würden. Aber ein paar Stichworte zum Suchen seien hier genannt: Delta-Sigma, Sukzessive Approximation (SAR): Parallelwandler, Sägezahnverfahren, Mehrrampenverfahren, um nur einige zu nennen.
Der A/D-Umsetzer des Arduino Nano ist in dem Atmel 328 Prozessor eingebaut. Der Wandler kann im Prinzip mit 15kSPS digitalisieren (bei maximaler Auflösung von 10 Bit), also 15.000 mal pro Sekunde. Die Eingänge sind gemultiplexed, das heißt nur je einer der 8 analogen Eingänge wird an den internen Wandler geschaltet. Werden mehre Kanäle verwendet, sinkt die maximale Digitalisierungsrate entsprechend, da man den Wandler nur für einen Kanal gleichzeitig verwenden kann.
Das Wandelprinzip ist, das der sukzessiven Approximation. Die Arduino Software-Bibliothek macht das Ganze aber einfach, man muss nur den Befehl digitalRead(PINNO...) verwenden und der Wandler wird aktiviert und nach der Umwandlung aktiviert. So bekommt man den Zahlenwert zwischen 0 und 1023, was einer Spannung zwischen 0 und 5V entspricht.

5.2 A/D Umsetzer und Potentiometer

Hier wird der Wert des eingebauten A/D-Umsetzer mit dem Sketch-Befehl analogRead() eingelesen. Der Wandler liefert einen Wert zwischen 0 und 1023, dabei 0 für 0V und 1023 für 5V. Die Ausgabe erfolgt über die serielle Schnittstelle am PC. Dazu muss man in der Arduino Programmierumgebung den seriellen Monitor aktivieren (CTRL-SHIFT-M), dann sieht man die Zahlenwerte vorbeilaufen. Wir erreichen mit der Schaltung einen maximalen Wert von ca. 920 statt 1023. Warum?: Durch den Spannungsteiler, gebildet durch den 10kOhm Vorwiderstand und den 10KOhm des Potis (10Kohm + 10Kohm in der Mitte angezapft), liegt am analogen Eingang maximal die halbe Spannung der Batterie an, also bei 9V entspricht dies 4.5V und nicht 5V, die für den vollen Bereich nötig wären. Man kann nun umgekehrt aus dem ausgegebenen Wert die Batteriespannung berechnen und den Wert ausgeben lassen:
VBatt = (value*5)/1023.
Dazu die Zeile: `Serial.print((value*10.0)/1023.0); Serial.print(„V „);`
vor die Zeile Serial.println(...) einbauen. Man erhält nun die Batteriespannung, wenn man den Regler ganz aufdreht. Der Eingang des Wandlers verträgt nur maximal 9V, daher sollte man die Batterie nicht direkt anschliessen, sondern einen Spannungsteiler verwenden.



ACHTUNG:
Terminal aktivieren!

Dazu beim Arduino Programm, CTRL und SHIFT und M drücken (CTRL=STRG je nach Tastatur).
Dann poppt ein Fenster hoch in dem die Infos stehen.
Ggf. muss man die Baudrate im Terminalfenster auf 9600 einstellen, so daß diese mit unserer im Programm übereinstimmt.

```
// DE_9 AD Umsetzer und POTI

// CTRL-SHIFT-M fuer
// seriell Port Monitor
// bzw. STRG und SHIFT und M
// druecken am PC !

#define PORTAD0 0 // wir nehmen Kanal 0
// Die Ausgabe erfolgt am PC, spaeter
// werden wir noch Anzeige Bricks anschliessen
void setup() { // start
  Serial.begin(9600); // die sogenannte BAUDRate
  // damit werden die Daten an den PC übertragen
  // 9600 Baud = 9600 Bits/Sekunde
}

void loop() { // start der Schleife
  int value; // Zwischenspeicher
  value = analogRead(PORTAD0); // A0 einlesen WANDLUUNG
  Serial.print((value*10.0)/1023.0);  // Umrechnen in Volt
  Serial.print(„V „); // Volt Zeichen in Aufführungszeichen setzen !
  Serial.println(value); // Danaben auch den Wert direkt
}
```

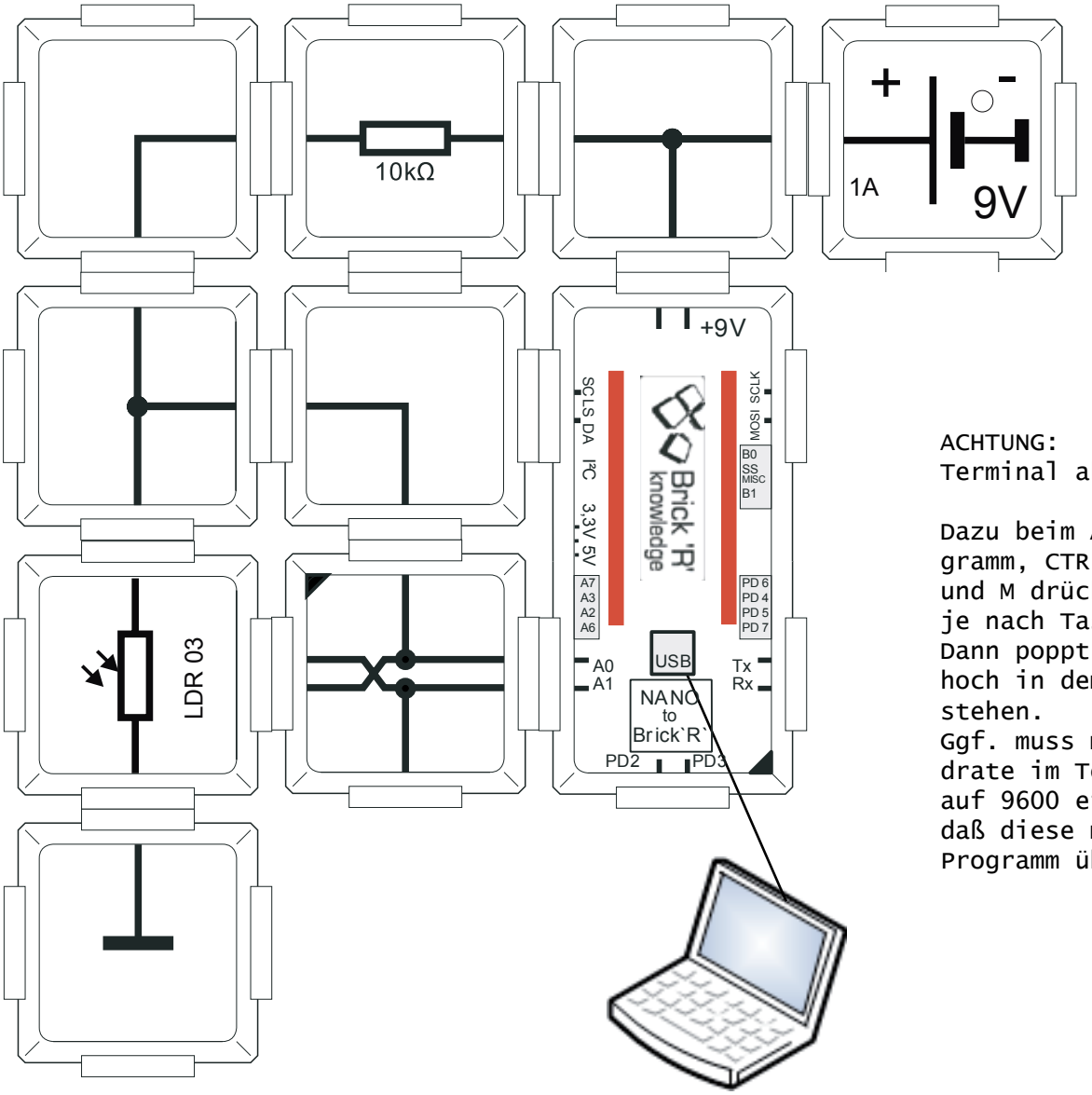
Beispiel einer Ausgabe am seriellen Monitor:

```
9.00V 921
8.99V 920
9.00V 921
9.00V 921
9.00V 921
8.99V 920
9.00V 921
8.99V 920
9.00V 921
9.00V 921
```

Was passiert? Wenn das Terminal aktiviert ist, sollte man dort eine Zeichenfolge auftauchen sehen z.B. „9.00V 921“ also erst der Spannungswert, dann eine Zahl die dem eingelesenen AD-Wert entspricht. Die Liste wird dann rasch länger. Durch Drehen am Poti, kann der Wert von 0 bis zur maximalen Spannung der Batterie eingestellt werden.

5.3 A/D Umsetzer und lichtempfindlicher Widerstand LDR

Der LDR03 ist ein lichtempfindlicher Widerstand. Er verringert seinen Widerstand umso mehr, je heller er beleuchtet wird. In der Schaltung bildet er zusammen mit dem 10 KOhm Widerstand einen Spannungsteiler. Am A/D-Umsetzer steht ein Wert von Vad = Rldr/(Rldr+R10k)*VBatt an. Wird der Widerstand kleiner, so wird auch die Spannung am A/D-Umsetzer kleiner. Zu beachten ist noch, dass der A/D-Umsetzer des Arduinos maximal 5V messen kann, bzw. die Spannung auf diesen Wert begrenzt. So fehlt ein kleiner Bereich wenn man den LDR verdunkelt. Auf der Konsole werden die umgerechneten Widerstandswerte des LDR03 ausgegeben. Dazu wird zunächst die Spannung berechnet und dann der Widerstand durch Umstellen der Standardformel (U=I*R). Den Wert für die Batteriespannung kann man aus dem vorherigen Versuch übernehmen (bei uns 9.0V angenommen). Ab einem gemessenem Wert von 12.5k sind 5.0V am Wandlereingang erreicht, daher kann man keine grösseren Werte messen und die Zahl der dargestellten Stellen ist natürlich übertrieben, die müsste noch angepasst werden (Aufgabe für den Leser). Der Wandler hat „nur“ 10 Bit Auflösung, damit ist 1/1024 der kleinste Schritt, daraus ergibt sich die Anzahl der anzeigbaren Stellen.



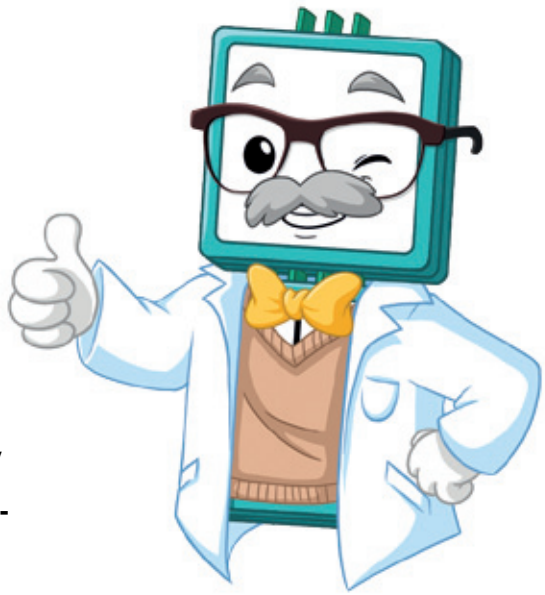
ACHTUNG:
Terminal aktivieren!

Dazu beim Arduino Programm, CTRL und SHIFT und M drücken (CTRL=STRG je nach Tastatur).
Dann poppt ein Fenster hoch in dem die Infos stehen.
Ggf. muss man die Baudrate im Terminalfenster auf 9600 einstellen, so daß diese mit unserer im Programm übereinstimmt.

```
// DE_10 AD Umsetzer und LDR03


// CTRL-SHIFT-M fuer
// seriell Port Monitor
// bzw. STRG und SHIFT und M
// druecken am PC !

#define PORTAD0 0 // wir nehmen Kanal 0
// Die Ausgabe erfolgt am PC, spaeter
// werden wir noch Anzeige Bricks anschliessen
void setup() { // start
  Serial.begin(9600); // die sogenannte BAUDRate
  // damit werden die Daten an den PC übertragen
  // 9600 Baud = 9600 Bits/Sekunde
}

void loop() { // Schleifenstart
  int value; // AD Umsetzer Wert zwischenspeichern
  double Vteiler,Rldr; // Spannung am Teiler, und Wid. LDR
  value = analogRead(PORTAD0); // A0 einlesen
  // Umrechnen value in Vteiler 5V und 1024 Schritte 0..1023
  Vteiler = value * 5.0 / 1023.0; // dann Spannung am Teiler
  // Umrechnen in Widerstandwert
  // Batterie vorher messen (!) hier 9V angenommen
  Rldr = (Vteiler*10000.0)/(9.0-Vteiler); // R LDR
  Serial.print(Rldr); // Wert auf dem Terminal ausgeben
  Serial.println(„Ohm „); // Text Ohm in doppelten Anfuehrungszeichen
}
```

Beispielausgabe:

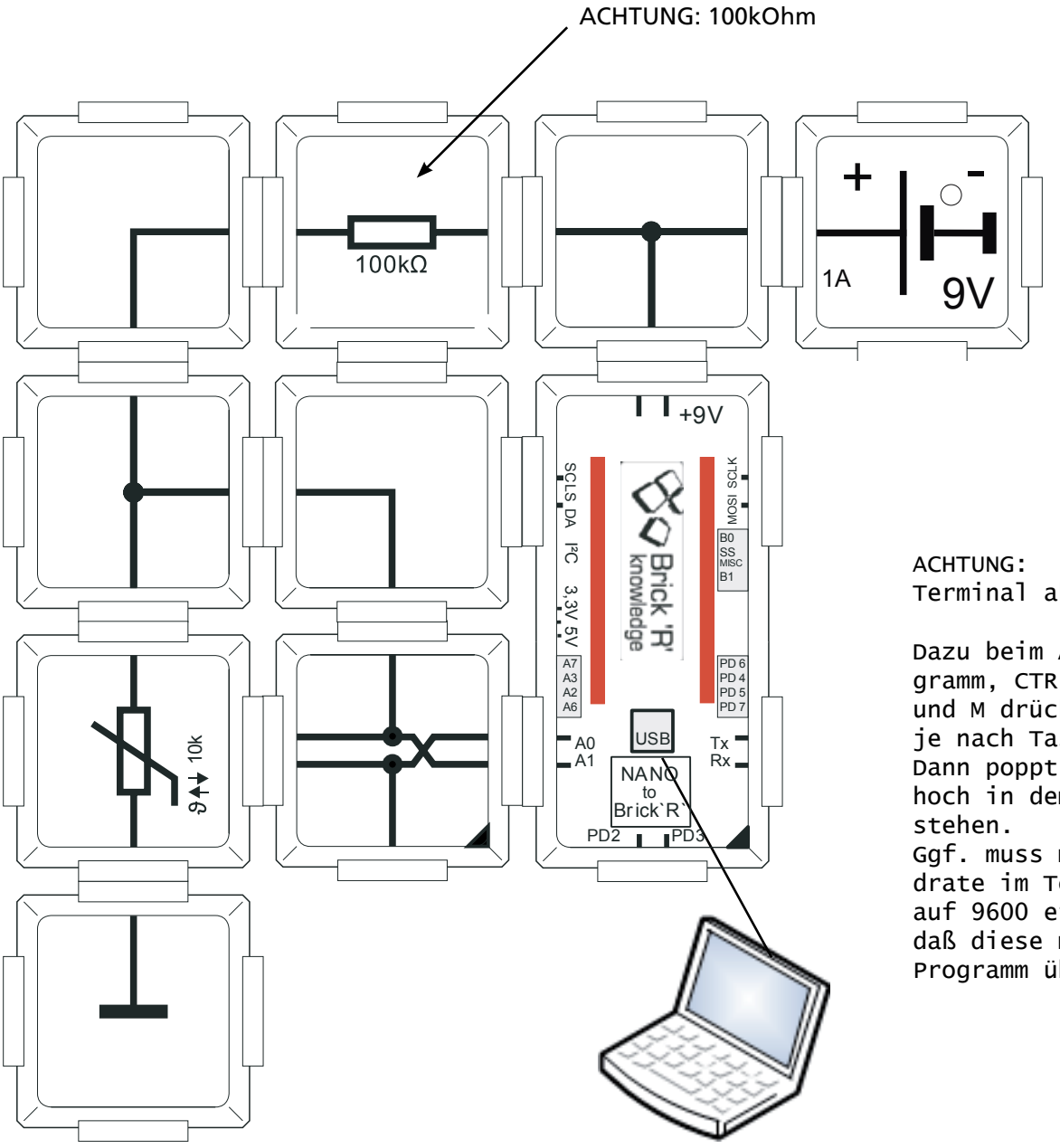
```
5719.65Ohm
5534.00Ohm
5391.17Ohm
5719.65Ohm
5481.76Ohm
5416.95Ohm
5746.54Ohm
5442.80Ohm
5442.80Ohm
5733.08Ohm
5404.05Ohm
5507.83Ohm
```



Was passiert? Wenn das Terminal aktiviert ist, sollte man dort eine Zeichenfolge auftauchen sehen z.B. „5719.65Ohm“, die den gemessenen Widerstandswert des LDR03 anzeigt. Der angezeigte Wert vergrößert sich bei geringerem Lichteinfall (Abdunkeln) und sinkt bei stärkerem Lichteinfall (Taschenlampe)

5.4 AD Umsetzer - Temperatur messen mit NTC

Verwendet man einen NTC (Heißleiter) als variablen Widerstand, kann man mit der Anordnung auch Temperaturen messen. Zunächst läßt sich der Widerstand wie gehabt ermitteln. Dann benötigt man aber eine komplexere Formel zur Bestimmung der Temperatur. Für Heißleiter gilt näherungsweise:
$R_T = R_N * e^{B(1/T-1/TN)}$ daraus läßt sich folgendes ableiten: $R= B *T_N /(B+\ln(R_T/R_N)*T_N)$
T_N ist i.A: 298.15K die Nenntemperatur. Der Nennwiderstand R_N liegt bei uns bei 10kOhm so wie er beim Hersteller gemessen wurde. Es wird eine Raumtemperatur von 25 Grad C, in K umgerechnet, verwendet.
B ist vom Hersteller vorgegeben (zwischen 2000K und 4000K), kann man durch eine Referenzmessung auch ermitteln. R_T ist die gemessene Temperatur (in K) daraus ergibt sich T (in K) . Als Widerstand zum Spannungsteiler ist hier ein 100kOhm Widerstand verwendet, damit man auch bei 0 Grad C messen kann. Zur Kalibrierung kann man nun z.B. Schmelzwasser verwenden, und damit den genauen Wert für B ermitteln, der ist aber leider in Wirklichkeit auch geringfügig temperaturabhängig, so gibt es unterschiedliche B-Werte je nach Temperatur. Man sucht sich am Besten die Temperatur für B heraus, in deren Bereich man messen will.

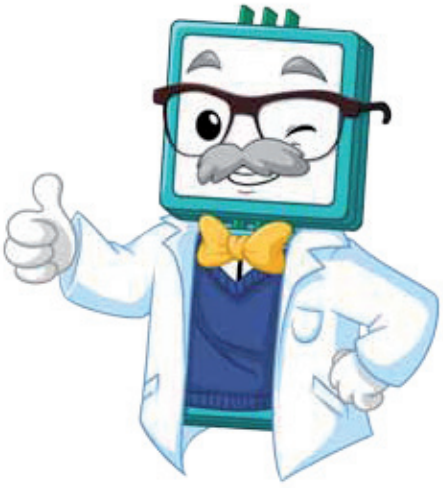


ACHTUNG:
Terminal aktivieren!

Dazu beim Arduino Programm, CTRL und SHIFT und M drücken (CTRL=STRG je nach Tastatur).
Dann poppt ein Fenster hoch in dem die Infos stehen.
Ggf. muss man die Baudrate im Terminalfenster auf 9600 einstellen, so daß diese mit unserer im Programm übereinstimmt.

```
// DE_11 AD Umsetzer und NTC

// CTRL-SHIFT-M fuer
// seriell Port Monitor
// bzw. STRG und SHIFT und M
// druecken am PC !

#define PORTAD0 0 // wir nehmen Kanal 0
// Die Ausgabe erfolgt am PC, spaeter
// werden wir noch Anzeige Bricks anschliessen
void setup() { // start
  Serial.begin(9600); // die sogenannte BAUDRate
  // damit werden die Daten an den PC übertragen
  // 9600 Baud = 9600 Bits/Sekunde
}


void loop() { // Schleife
  int value; // Wert vom AD Umsetzer
  double Vteiler,RNtc; // Spannung am Teiler Rntc
  value = analogRead(PORTAD0); // A0 einlesen
  // Umrechnen value in Vteiler
  Vteiler = value * 5.0 / 1023.0; // in Volt
  // Umrechnen in Widerstandwert
  // Batterie vorher messen (!)
  double VBatt = 9.0; // MUSS MAN Anpassen !
  double RTeiler = 100000.0; // 100k Teiler
  RNtc = (Vteiler*RTeiler)/(VBatt-Vteiler); //Widerstand
  // Dann Temperatur ermitteln.
  double B = 3800.0; // je nach NTC unterschiedlich !! ermitteln
  double RN = 10000.0; // 10kOhm
  double TN = 298.15; // 25 Grad Celsision in Kelvin
  double T = (B*TN)/(B+log(RNtc/RN)*TN)-273.15; // T in Celsius
  Serial.print(T); // Temperatur ausgeben
  Serial.print(„ Grad C „); // Text in doppelten Anfuehrungszeichen
  Serial.print(RNtc); // auch den Widerstandswert ausgeben
  Serial.println(„Ohm „); // Text in doppelten Anfuehrungszeichen
}
```

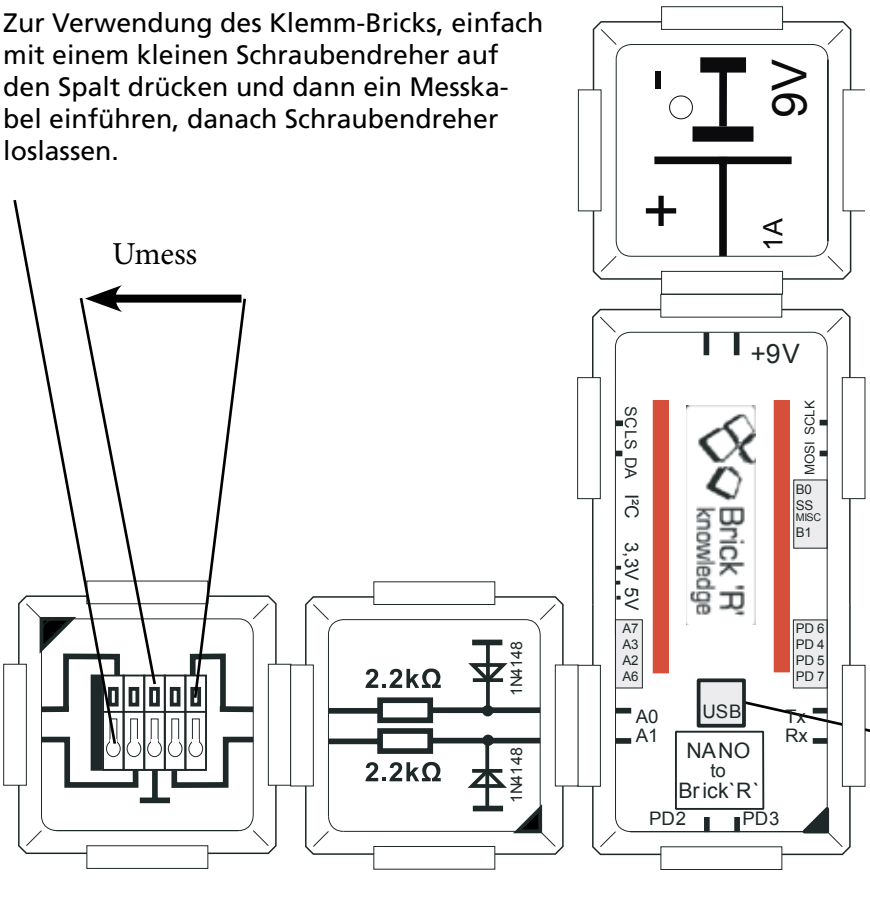
Was passiert? Wenn das Terminal aktiviert ist, sollte man dort je zwei Zeilen sehen, eine mit dem Temperaturwert in Grad Celsius, die nächste mit dem gemessenen Widerstandswert. Wenn man den NTC erhitzt (Handwärme reicht), sollte sich die angezeigte Temperatur erhöhen und der Widerstand verkleinern.



5.5 Voltmeter mit AD Umsetzer und Kalibrierung - PC Schnittstelle

Unser Baustein ist auch hervorragend als Voltmeter geeignet. Man muss dazu nur noch Messleitungen anschliessen. Der Klemm-Brick eignet sich dazu sehr gut und ist in der Schaltung eingebaut. Nun kann man auch beide A/D-Umsetzer an den Ports A0 und A1 verwenden. Im Programm wird die Spannung aus dem digitalisierten Wert berechnet und ausgegeben. Als einfachen Schutz haben wir hier einen Doppelwiderstand eingebaut. Er dient als Strombegrenzung, wenn man Spannungen größer als 5V anlegt und die Dioden schützen vor Spannungen kleiner 0 Volt (z.B. beim Verpolen an Umess). Im Brick ist noch je ein 100 Ohm Widerstand am Eingang: Damit wird dann gesichert, dass nur die externen Dioden bei negativen Spannungen ansprechen und nicht die im Chip eingebauten Dioden. Man sollte dies aber trotzdem vermeiden. Über den 2.2k Ohm Widerstand fliesst bei 9V ein maximaler Strom von 4.1mA. Die Schutzmaßnahmen bewirken auch eine Veränderung des Messergebnis, daher muss man für genaue Messungen eine Kalibrierung durchführen.
Zur Kalibrierung benötigt man eine definierte Spannungsquelle. Daraus kann man dann einen Korrekturfaktor berechnen. Den bringt man dann in die Formel ein.
Die Spannung muss genau bekannt sein, entweder durch ein genau einstellbares Netzteil, oder man misst eine Spannungsquelle mit einem genauen Voltmeter und verwendet dann diese. Wird z.B. bei einer 2V Quelle der Wert 2100mV bei uns angezeigt, so benötigt man einen Korrekturfaktor von 2/2.1 den man in die Umrechnungsformel für voltage1= oder voltage2= mit reinmultipliziert.
Also Voltage1 = value1 * 5000.0 / 1023.0 * 2.0 / 2.1; die Konstanten kann man auch ausmultiplizieren, so ist das Program aber leichter lesbar. Nun kommt noch die Frage nach der Auflösung. Bei uns liegt ein 10 Bit A/D-Umsetzer vor, also hat er eine Auflösung von 1/1024, die einem Bit entspricht. Damit ist die kleinste Spannung, die man gerade noch messen kann, gerundet ca. Umin = 5.0V / 1024 = 4.9mV

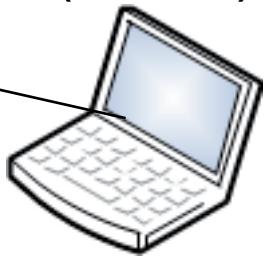
Zur Verwendung des Klemm-Bricks, einfach mit einem kleinen Schraubendreher auf den Spalt drücken und dann ein Messkabel einführen, danach Schraubendreher loslassen.



ACHTUNG:
Terminal aktivieren!

Dazu beim Arduino Programm, CTRL und SHIFT und M drücken (CTRL=STRG je nach Tastatur). Dann poppt ein Fenster hoch in dem die Infos stehen.
Ggf. muss man die Baudrate im Terminalfenster auf 9600 einstellen, so daß diese mit unserer im Programm übereinstimmt.

Was passiert? Wenn das Terminal aktiviert ist, erscheint eine Zeile mit der gemessenen Spannung von Kanal A0 und daneben von Kanal A1. Für einen schnellen Test kann man die statt des Klemm-Bricks einen Mass-Brick einsetzen. Dann sollte der Wert fast 0mV sein. Offen ist er natürlich undefiniert und man muss erst eine Spannungsquelle anschliessen (z.B. Batterie).



```
// DE_12  AD Umsetzer als Voltmeter

// CTRL-SHIFT-M fuer
// seriell Port Monitor

#define PORTAD0 0 // Kanal 0 bei A0
#define PORTAD1 1 // Kanal 1 bei A1

// Die Ausgabe erfolgt am PC, spaeter
// werden wir noch Anzeige Bricks anschliessen
void setup() { // start
  Serial.begin(9600); // die sogenannte BAUDRate
  // damit werden die Daten an den PC übertragen
  // 9600 Baud = 9600 Bits/Sekunde
}

void loop() {
  int value1,value2; // diesmal gleich zwei analoge Werte speichern
  double Voltage1,Voltage2; // und beide in Spannung umrechnen
  value1 = analogRead(PORTAD0); // A0 einlesen
  value2 = analogRead(PORTAD1); // A1 einlesen
  // Umrechnen value in Vteiler
  Voltage1 = value1 * 5000.0 / 1023.0; // ggf Korrekturfaktor
  Voltage2 = value2 * 5000.0 / 1023.0; // verwenden
  Serial.print(Voltage1); // Ausgabe der Spannung vom Kanal 0
  Serial.print(„mV „); // Text in doppeltem Anfuehrungszeichen
  Serial.print(Voltage2); // Ausgabe der Spannung vom Kanal 1
  Serial.println(„mV „); // Text in doppelten Anfuehrungszeichen
}
```

Ausgabe:

```
943.30mV 953.08mV
943.30mV 957.97mV
938.42mV 953.08mV
938.42mV 953.08mV
938.42mV 948.19mV
938.42mV 943.30mV
933.53mV 943.30mV
933.53mV 943.30mV
928.64mV 938.42mV
913.98mV 923.75mV
```

6. I2C BUS

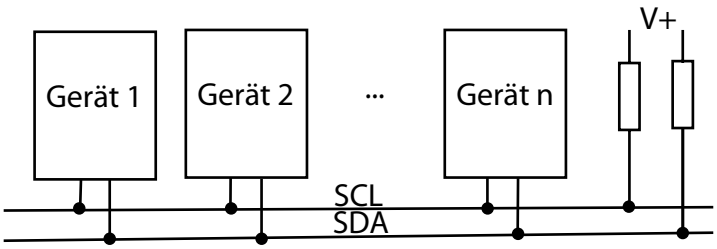
6.1 I2C Bus Aufbau Prinzip und Befehle

Der I2C Bus ist eine serielle Schnittstelle, die mit zwei Leitungen auskommt. Der Taktleitung SCL und der Datenleitung SDA. Die Leitungen arbeiten bidirektional, man unterscheidet zwischen Master und Slave. Der NANO ist in unserem Fall der Master und die anderen Bricks sind Slaves. Die Bausteine werden über I2C Adressen angesprochen, 128 sind pro Bus möglich. Dabei können einzelne Bausteine auch mehrere Adressen belegen. Manche Bausteine haben auf der Rückseite kleine DIL-Schalter, so daß man die Adressbereiche umschalten kann, wenn mehrere Bausteine am gleichen Bus verwendet werden.

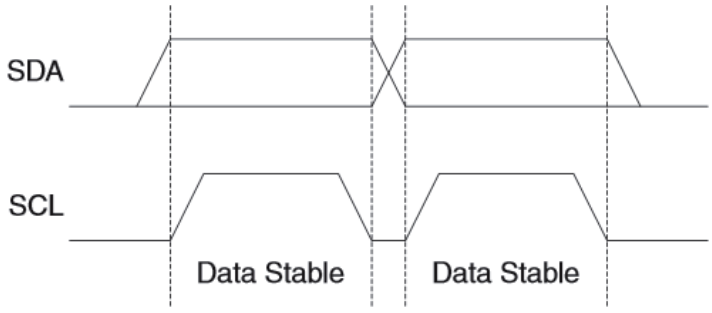
Der Bus kann im Prinzip mit unterschiedlichen Geschwindigkeiten betrieben werden:

Standard Mode (Sm)	0.1MBit/sec
Fast Mode (Fm)	0.4MBit/sec
High Speed Mode (HS-mode)	1.0MBit/Sec
Ultra Fast-Mode (UFm)	5.0Mbit/Sec.

Viele Microcontroller können nur die ersten beide Modi (zum Beispiel die CPU des Arduino Nano) und manchmal noch den dritten Mode handeln. Das gleiche gilt für die Peripheriebausteine. Die Modi müssen natürlich zusammenpassen. Der Master, also meist der Microcontroller, gibt dabei den Takt vor, dazu dient die Leitung SCL (Serial Clock). Über SDA (Serial Data) werden die eigentlichen Daten übertragen.

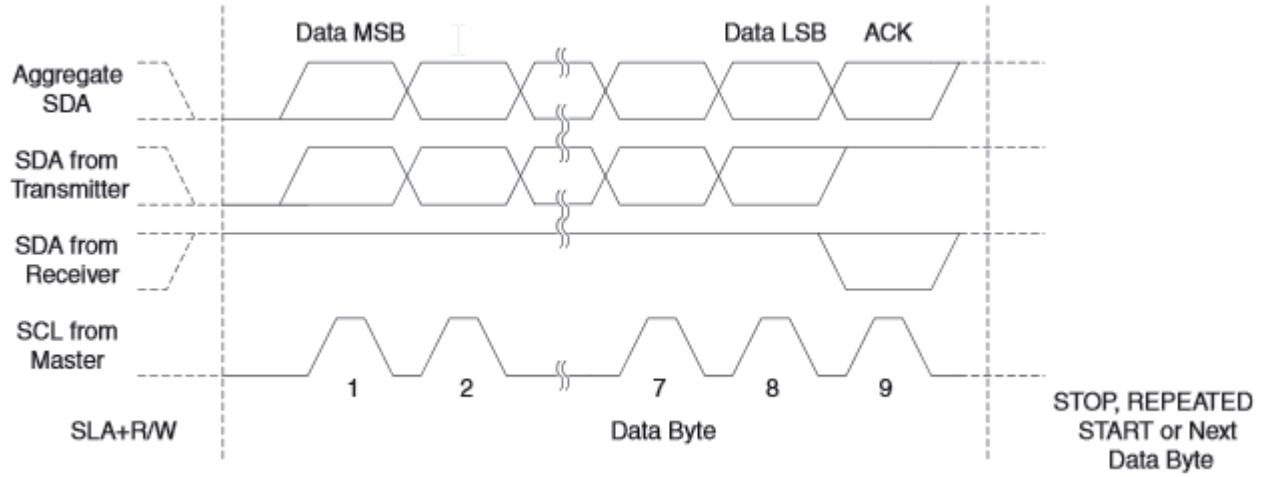


Man kann maximal 128 Geräte an den Bus anschließen, wenn jedes der Geräte nur eine Adresse belegt, sonst entsprechend weniger. Hier mit Gerät 1 bis n angedeutet. Die Geräte sind alle über zwei Leitungen verbunden. Zwei Widerstände zur Versorgungsspannung (Arduino meist 5V manchmal auch 3.3V), der Wert liegt im Kiloohmbereich. Die Widerstände sind bei uns schon im Nano-Brick eingebaut.



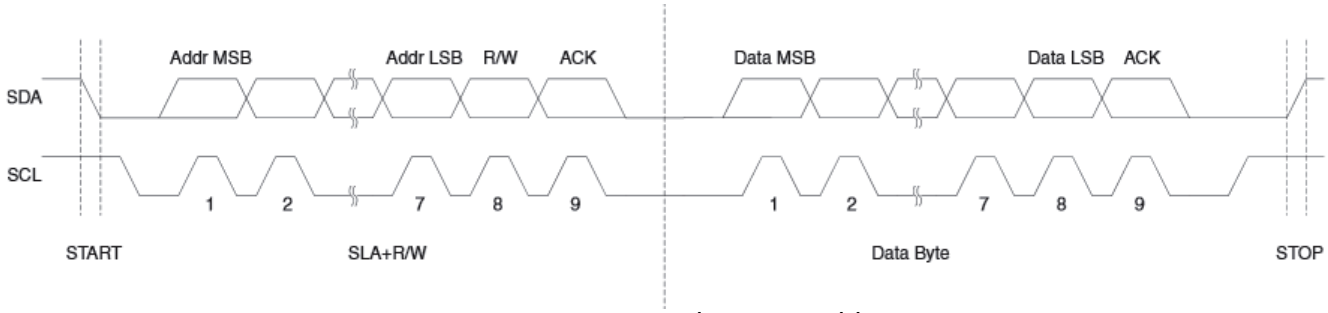
Auszug aus Atmel 328 Datenblatt

Der Takt gibt dabei immer an, wann stabile Daten anliegen, oben im Diagramm sieht man, dass dies immer beim High-Pegel der Fall ist. Der Empfänger kann dort die Daten abtasten und auswerten. Der Master gibt dabei den Takt vor, er legt dann entweder selbst Daten an oder erwartet sie an den entsprechenden Abfragepunkten.



Auszug aus Atmel 328 Datenblatt

Im oberen Diagramm sieht man einen Datentransfer zerlegt in Master, Transmitter und Receiver als Sender und Empfänger. Der Master gibt den Takt vor. Wichtig ist die Synchronisation. Der Empfänger (egal ob Master oder Slave) sendet am Schluss jedes Datenpakets ein ACK-Signal, indem er die Leitung auf Low zieht. Da dies ein Wire OR ist, reicht es wenn ein Slave antwortet. Oben zeigt sich das SDA-Signal als Gesamtheit, also so wie es auf der SDA-Leitung anliegt.

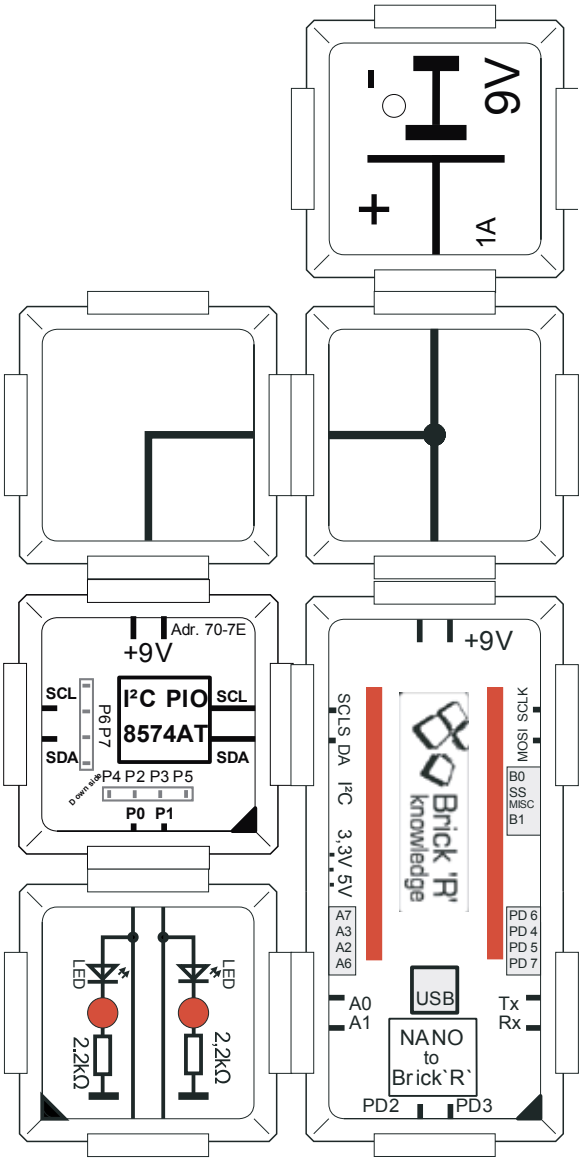


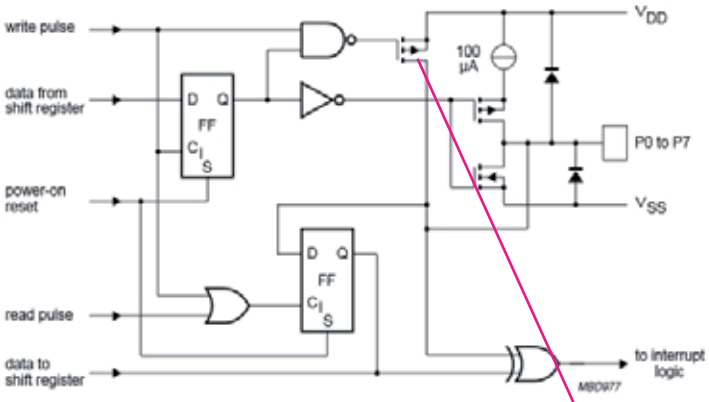
Auszug aus Atmel 328 Datenblatt

Hier das Beispiel einer gesamten Datenübertragung. Zunächst wird ein Paket mit der Adresse gesendet. Die Adresse besteht aus 7 Bits. Der Empfänger vergleicht dies dann mit seiner internen Adresse. Es kommt noch ein weiteres mit R/W benanntes Bit hinzu. Es definiert ob ein Schreib oder Lesezyklus vom Slave gestartet werden soll. Stimmt die Adresse überein, so antwortet der Slave mit ACK, legt die Leitung an der entsprechend vorgesehenen Stelle auf einen Low Pegel, bei uns 0V. Danach kann der Datentransfer beginnen. Am Schluss wird ein Stop-Zyklus eingeleitet. Dazu wird der Takt auf High gesetzt (wegen der Low-Activen Logik nicht mehr aktiviert), und dann die SDA Leitung freigegeben. Nun kann auch ein anderen Master, falls man mehrere auf dem Bus hat, den Zyklus neu starten. Die Leitungen SDA und SCL sind beide auf High, das bedeutet dass der I2C Bus frei verwendbar ist.
Der Bus ist einfach nutzbar, dazu stellt die Arduino Bibliothek mehrere Befehle bereit, die wir im folgenden verwenden werden, um I2C Bausteine an den Nano-Brick anzuschließen.

6.2 I2C Bus und IO Port Baustein

Der Bausteine PCF8574 ist ein sogenannter IO Extender. Der beinhaltet 8 IO Ports, die man als Ausgang aber auch als Eingang verwenden kann. Den Baustein PCF8574 gibt es in zwei Ausführungen, den PCF8574T (40-4E) und PCF8574AT (70-7E), dabei haben die beiden einen unterschiedlichen Adressbereich. Nach dem Öffnen des Deckels, gibt es auf der Oberseite noch einen DIL-Schalter mit dem sich die Unteradresse einstellen lässt. (40,42,44,46,48,4A,4C,4E). Achtung: Beim Aufsetzen des Deckels auf die Orientierung achten. Das kleine Dreieck auf der Oberseite kennzeichnet die entsprechende abgeschrägte Ecke bei der Platine! Wir nutzen bei den Adressen die 8 Bit Zählweise. Beim I2C werden aber nur 7 Bit verwendet, wobei das untere das Schreibbit ist. Daher werden die Adressen unten in der Tabelle in zweier Schritten gezählt. Der DIL-Schalter hat 4 Positionen, wobei bei uns Position 4 unbelegt ist, somit sind bis zu 8 unterschiedliche Stellungen möglich.
Die Ausgänge vom PCF8574 haben aber eine Besonderheit, sie sind quasi bidirectional. Als open drain können sie recht große Lasten auf 0V ziehen, aber auf high nur mit 100 uA nach +5V. Daher blinken die LEDs in diesem Beispiel nur schwach. Immer wenn man ein hIgh Signal sendet, wird der Ausgang ganz kurz auf Versorgungs-Spannungslevel hochgezogen, dann aber der Stromquelle überlassen. Dies ermöglicht einen quasi bidirektionalen Betrieb ohne zusätzliches Richtungsregister. Will man die LEDs heller leuchten lassen, muss man eine andere Beschaltung verwenden.



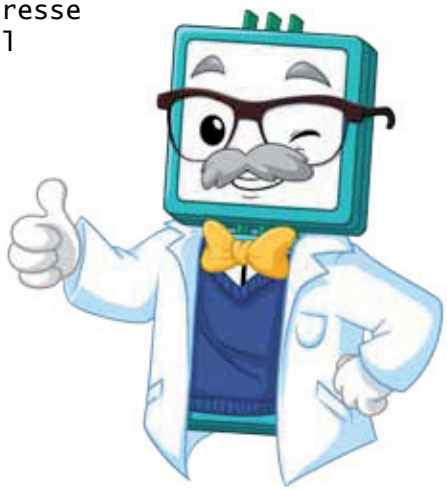


Auszug aus dem Phillips Datenblatt zum PCF8574.
Wichtig ist die Logik rechts mit den beiden MOSFETS und der 100µA Stromquelle, sowie dem einzelnen p-MOSFET der kurzzeitig beim Setzen der Register auf 1 den Ausgang auf VDD (=High) setzt

DIL Schalter:

1234	8574T	8574AT
000x	40h	70h
001x	42h	72h
010x	44h	74h
011x	46h	76h
100x	48h	78h
101x	4Ah	7Ah
110x	4Ch	7Ch
111x	4Eh	7Eh

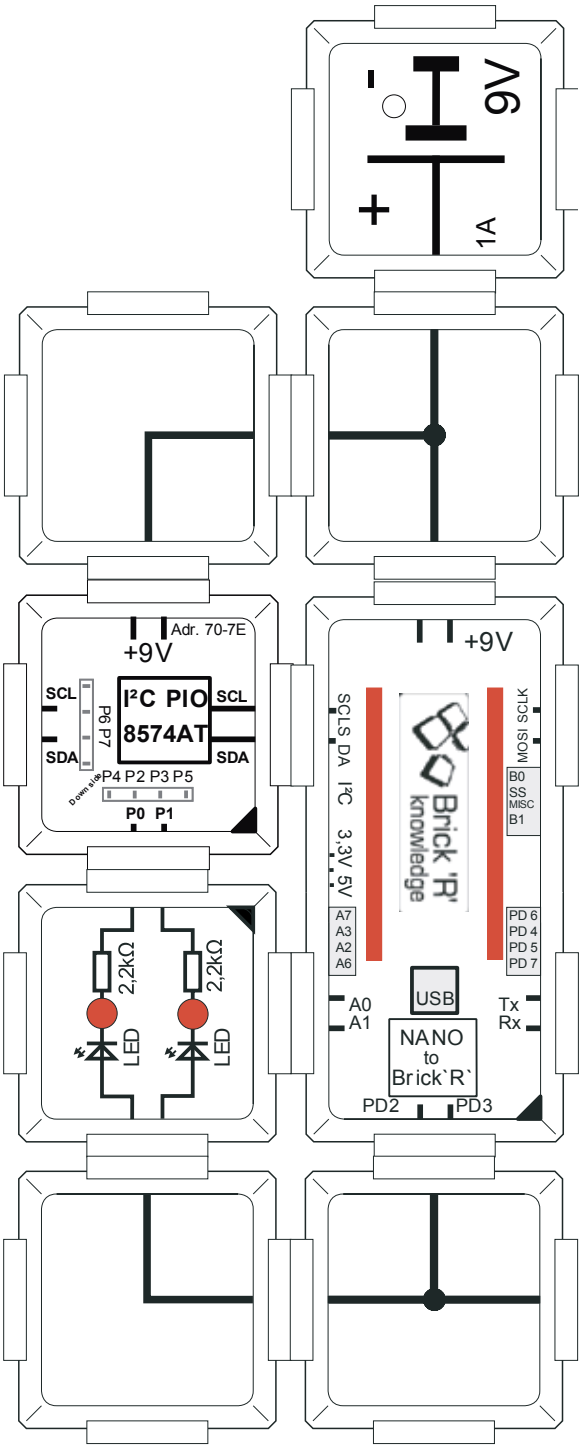
```
// DE_13 I2C - IO Port 8574AT und 8574T

#include <Wire.h> // Definitionen laden fuer I2C
#include <avr/pgmspace.h> // weitere Definitionen

// Alle Adressen bei den 8574xx bricks:
#define i2cIO8574_0 (0x40>>1) // Trick um elegant mit Bytes
#define i2cIO8574_1 (0x42>>1) // zu arbeiten statt in 7 Bit
#define i2cIO8574_2 (0x44>>1) // das letzte Bit ist das R/W
#define i2cIO8574_3 (0x46>>1) // dass von der Arduino
#define i2cIO8574_4 (0x48>>1) // Bibliothek dazugefuegt wird
#define i2cIO8574_5 (0x4A>>1) // wir definieren hier alle
#define i2cIO8574_6 (0x4C>>1) // Bereiche die man mit den
#define i2cIO8574_7 (0x4E>>1) // Bricks einstellen kann,.

#define i2cIO8574A_0 (0x70>>1) // Die Serie PCF8474AT
#define i2cIO8574A_1 (0x72>>1) // beginnt bei Adresse
#define i2cIO8574A_2 (0x74>>1) // 0x70 = 70 sedezimal
#define i2cIO8574A_3 (0x76>>1)  // 01110000 binaer
#define i2cIO8574A_4 (0x78>>1) // oder intern 0111000
#define i2cIO8574A_5 (0x7A>>1) // dabei 0111xxx
#define i2cIO8574A_6 (0x7C>>1) // mit x fuer die
#define i2cIO8574A_7 (0x7E>>1) // Dilschalterposition

// ACHTUNG: hier Zuordnung setzen je nach Schalterstellung

#define myi2cIOadr i2cIO8574A_0 // HIER PASSENDE ADRESSE EINTRAGEN

void setup() {
  Wire.begin(); // I2C aktivieren !
}

void loop() {
  Wire.beginTransmission(myi2cIOadr);// Startvorgang I2C Adresse
  Wire.write(0x55) ; // IO POrts auf 01010101 abwechseln
  Wire.endTransmission(); // Stop Kondition setzen bei I2C
  delay(100); // 100 ms verzoegern.
  Wire.beginTransmission(myi2cIOadr); // Nochmal die Adresse
  Wire.write(0xaa) ; // IO POrts auf 10101010 im Wechsel
  Wire.endTransmission(); // Stop Kondition setzen
  delay(100); // 100 ms verzoegern.
}
```



Was passiert? Die LEDs blinken abwechselnd im Sekundentakt.
Wenn sie nicht blinken, zuerst die Adress-Einstellungen an den DIL-Schaltern kontrollieren !

6.3 I2C Bus und IO Port Baustein LOW Activ

Der 8574 hat open drain Ausgänge mit einem ganz weichen pullup (100uA) nach +5V. Daher ist es eigentlich besser, Lasten nach 0V ziehen zu lassen, und nicht wie vorher auf einen hohen Pegel. Hier eine trickreiche Schaltung dazu. Die LEDs sind mit der Kathode an die Ausgänge des Treibers geschaltet. Nun benötigt man noch eine positive Versorgungsspannung für die Anoden. Dazu werden zwei Ausgänge des Nano verwendet, die auf High programmiert sind und damit 5V liefern. Achtung: nicht die 9V der Spannungsquelle zur Versorgung der LEDs verwenden, dies könnte den 8574 beschädigen.



DIL Schalter:

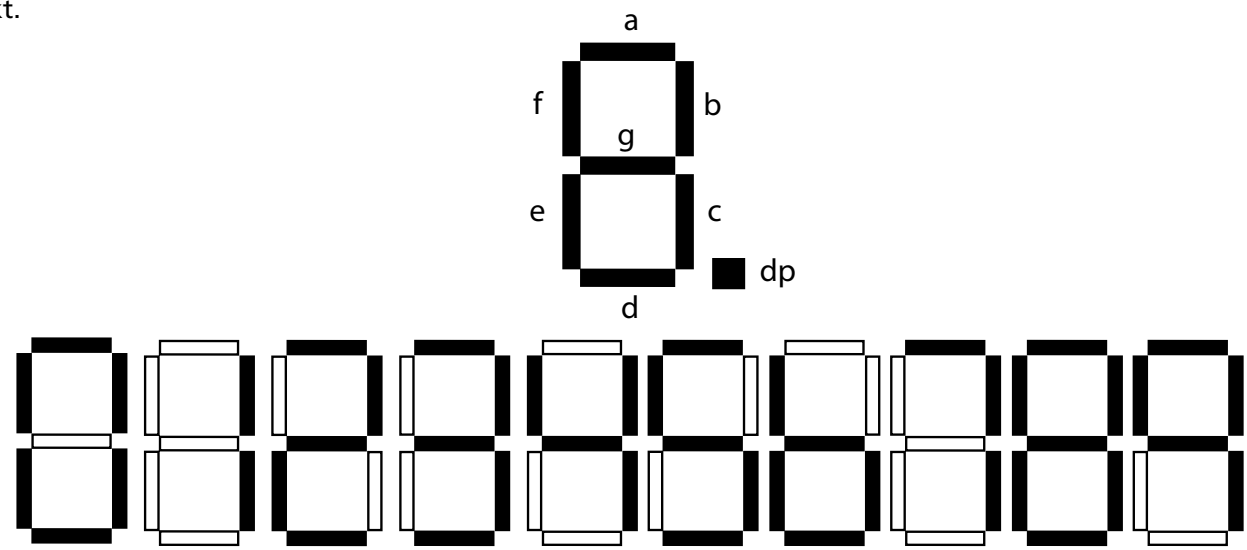
1234	8574T	8574AT
000x	40h	70h
001x	42h	72h
010x	44h	74h
011x	46h	76h
100x	48h	78h
101x	4Ah	7Ah
110x	4Ch	7Ch
111x	4Eh	7Eh

Was passiert? Die LEDs blinken abwechseln im Sekundentakt.
Wenn sie nicht blinken, zuerst die Adress-Einstellungen an den DIL-Schaltern kontrollieren !

```
// DE_14 I2C - IO Port 8574AT und 8574T Lowpulse

#include <Wire.h>
#include <avr/pgmspace.h>

#define i2cIO8574_0 (0x40>>1) // Trick um elegant mit Bytes
#define i2cIO8574_1 (0x42>>1) // zu arbeiten statt in 7 Bit
#define i2cIO8574_2 (0x44>>1) // das letzte Bit ist das R/W
#define i2cIO8574_3 (0x46>>1) // dass von der Arduino
#define i2cIO8574_4 (0x48>>1) // Bibliothek dazugefuegt wird
#define i2cIO8574_5 (0x4A>>1) // wir definieren hier alle
#define i2cIO8574_6 (0x4C>>1) // Bereiche die man mit den
#define i2cIO8574_7 (0x4E>>1) // Bricks einstellen kann,.

#define i2cIO8574A_0 (0x70>>1) // Die Serie PCF8474AT
#define i2cIO8574A_1 (0x72>>1) // beginnt bei Adresse
#define i2cIO8574A_2 (0x74>>1) // 0x70 = 70 sedezimal
#define i2cIO8574A_3 (0x76>>1) // 01110000 binaer
#define i2cIO8574A_4 (0x78>>1) // oder intern 0111000
#define i2cIO8574A_5 (0x7A>>1) // dabei 0111xxx
#define i2cIO8574A_6 (0x7C>>1) // mit x fuer die
#define i2cIO8574A_7 (0x7E>>1) // Dilschalterposition

// ACHTUNG: hier Zuordnung setzen je nach Schalterstellung

// ACHTUNG: hier Zuordnung setzen je nach DILSCHALTER !

#define myi2cIOadr i2cIO8574A_0 // wir nehmen den 0x70

#define PULLUP2 2 // Kleiner Trick am Port 2 und 3
#define PULLUP3 3 // bekommen wir 5V (40mA maximal!)

void setup() {
  pinMode(PULLUP2, OUTPUT); // PD2 und PD3 brauchen wir als Ausgang
  pinMode(PULLUP3, OUTPUT); // muss man dann noch auf High stellen
  digitalWrite(PULLUP2,HIGH); // damit sie beide 5V liefern
  digitalWrite(PULLUP3,HIGH); // schnell schnell, da sonst Kurzschluss
  Wire.begin(); // I2C aktiviern
}

void loop() {
  Wire.beginTransmission(myi2cIOadr); // Dann wie gehabt starten
  Wire.write(0x55) ; // IO Ports auf 01010101 abwechseln
  Wire.endTransmission(); // Ende
  delay(100); // 10ms Verzoegerung
  Wire.beginTransmission(myi2cIOadr); // ncohmal starten
  Wire.write(0xaa) ; // IO Ports auf 10101010 im Wechsel
  Wire.endTransmission(); // und wieder Stopbit
  delay(100); // nochmal 100ms Warten fuer Blinkeffekt
}
```

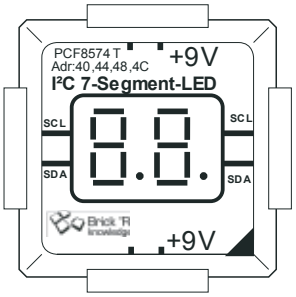
6.4 Die Siebensegment-Anzeige - Prinzip

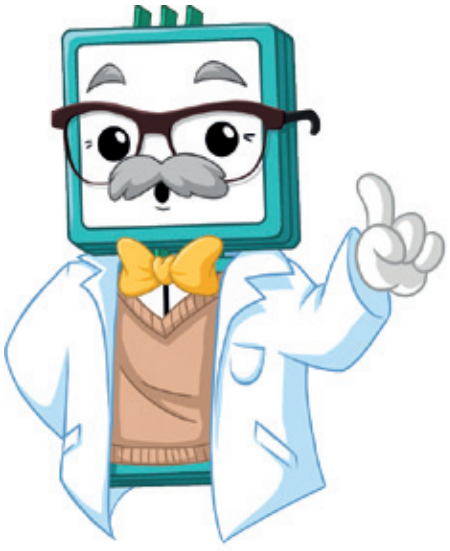
In den Anfängen der Computertechnik machte man sich Gedanken, wie man Ziffern darstellen kann. Am einfachsten waren damals 10 Lampen, die von 0 bis 9 beschriftet wurden.
Später hat man dann die Lampen zum Beleuchten kleiner Glasplatten mit entsprechenden Bohrungen verwendet. Zur gleichen Zeit kamen die sogenannten Nixi-Röhren auf, die Ziffern waren aus Draht geformt, und bei Anlegen einer höheren Spannung unter einem Schutzgas, begannen die Ziffern zu leuchten.
Dann kam man auf die Idee, die Zahlen in Segmente zu zerlegen. Mit sieben Banken kann man alle Ziffern zwischen 0 und 9 darstellen. Die ersten Anzeigen verwendeten noch Glühdrähte, aber mit dem Aufkommen der LEDs wurde es einfacher. Hinter jedem Segment verbirgt sich eine LED, die den Balken beleuchtet. Die einzelnen Segmente werden häufig mit a bis h bezeichnet. Optional gibt es noch eine einzelne LED für den Dezimalpunkt.



Die Ziffern 0 bis 9. Man kann sogar Buchstaben mit etwas Phantasie darstellen (Aufgabe für den Leser die Buchstaben E,A,L,klein O - dann wird es phantasievoll). In der Praxis gibt es noch 16 Segmentanzeigen, die aber leider nur schlecht verfügbar sind. Wir werden bei der OLED dann noch elegantere Formen kennenlernen, wenn man Buchstaben oder Grafiken braucht.

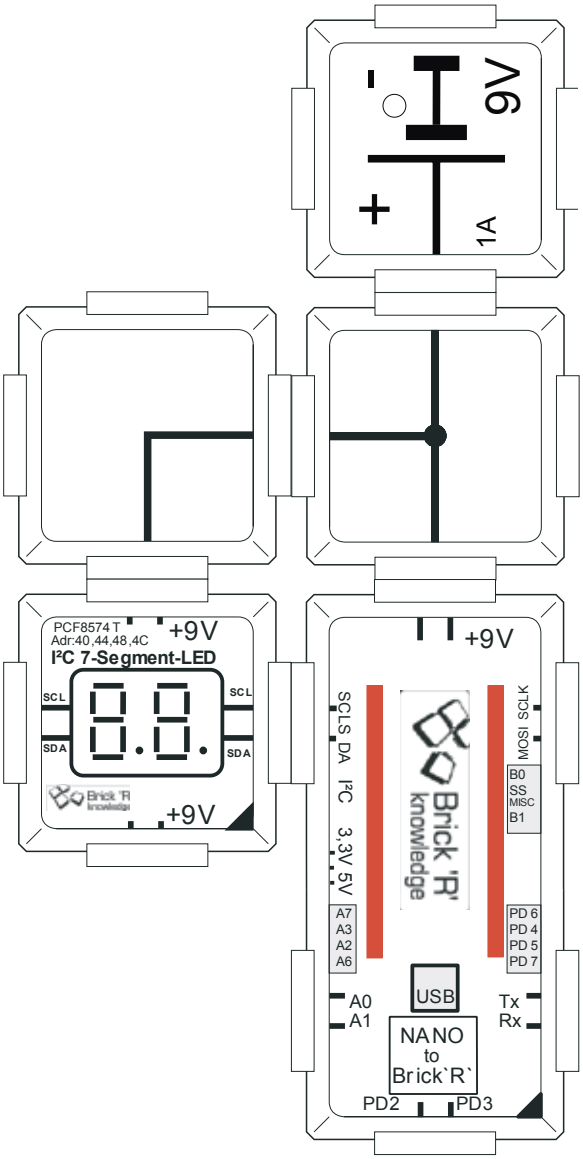
Bei unserem Brick sind zwei solcher 7-Segment-Anzeigen untergebracht.
Die LEDs werden mit Hilfe zweier Bausteine 8574T angesteuert (16 Ausgangsleitungen an die LEDs). Auf der Rückseite lässt sich mit kleinen Schaltern die I2C Adresse einstellen. Maximal sind vier solcher Bausteine an einem I2C Bus nutzbar.



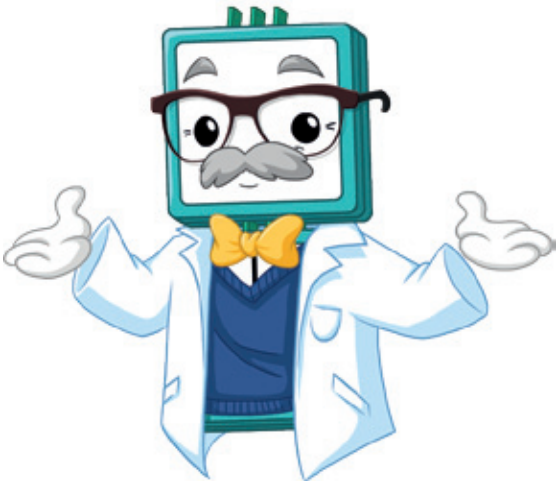


6.5 Siebensegment-Anzeige als I2C Brick - Aufbau und Adressen

Wir realisieren nun eine einfache Schaltung mit einer der beiden 7-Segment-Anzeigen. Achtung, die Adressen sind voreingestellt. Unter Umständen muss die Einstellung kontrolliert oder anpasst werden. Die Schalter sind auf der Rückseite der Platine, dazu muss man den Deckel vorsichtig öffnen und beim Wiederverschließen unbedingt drauf achten, dass er korrekt eingesetzt wird (Orientierung beachten).
Wir haben eine kleine Bibliothek für den Arduino Nano vorbereitet, in dem alle Segmente kodiert sind. Dies geschieht mit Hilfe einer so genannten Zeichentabelle. Den Ziffern und sogar allen Buchstaben von A-Z wird mit einem Byte die Kombination der Segmente in einer Tabelle zugewiesen. Die Tabelle wird dann vom Programm über den Index, dem Code der Ziffer angesprochen und so umgerechnet. Dazu haben wir schon ein paar Unterprogramme vorbereitet.
Die Prozedur display_seg1x() zeigt ein einzelnes Segment an. Dazu übergibt man die I2C Adresse des entsprechenden Treibers. Die Routine get_7seg () übernimmt dabei die Umrechnung des ASCII-Codes in den Index der Tabelle mit den Segmentzuordnungen. Mit display_seg1xbin () können die Segmente auch direkt angesprochen werden. Zwei solcher Parallel-Port-Treiber gibt es für die beiden Ziffern die aufsteigende Adressen besitzen. Dazu am besten aber unser Programmbeispiel genauer studieren und einfach mal damit experimentieren.
Das Beispielprogramm gibt die gefundene I2C Adresse auf dem Display aus, so kann man sich den Wert leicht notieren.



Was passiert? Auf der Siebensegment-Anzeige erscheinen zwei Zahlen, die der Adresse des I2C Busses entsprechen. Also entweder 40, 44, 48, 4C oder aus der Gruppe 70, 74, 78, 7C. Die Adresse hängt von der Einstellung des DIL-Schalters auf der Rückseite und vom Bausteintyp (8574T oder 8574AT) ab.

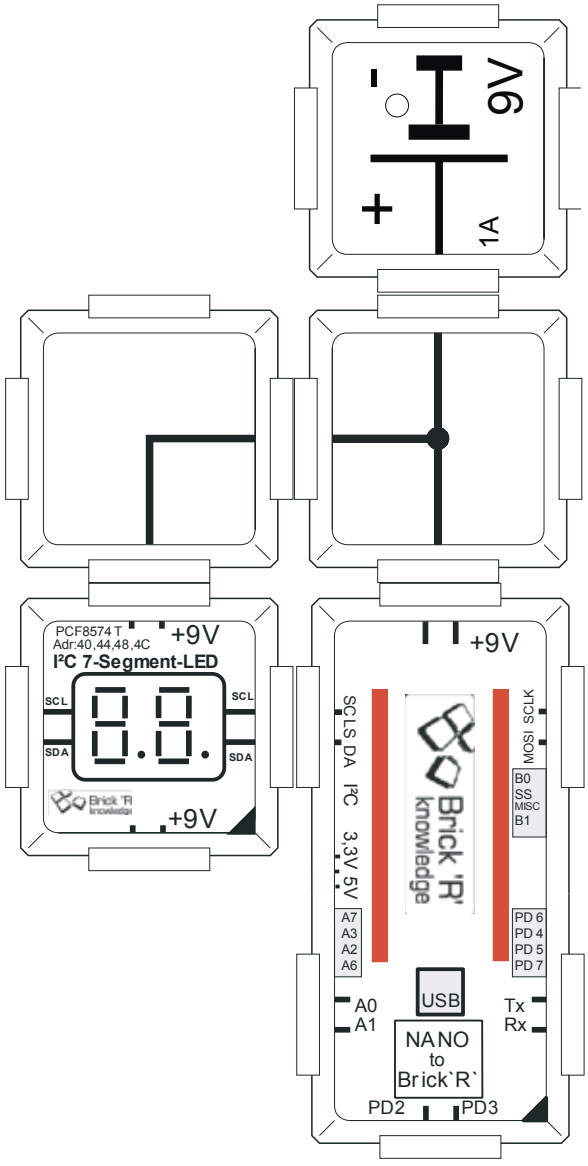


```
// DE_15 7segment Anzeige als I2C Brick
#include <Wire.h>

// 8574T standardmaessig verbaut
#define i2cseg7x2alsb1 (0x40>>1) // 7Bit
#define i2cseg7x2amsb1 (0x42>>1) //
#define i2cseg7x2blsb1 (0x44>>1)
#define i2cseg7x2bmsb1 (0x46>>1)
#define i2cseg7x2clsb1 (0x48>>1)
#define i2cseg7x2cmsb1 (0x4A>>1)
#define i2cseg7x2dlsb1 (0x4C>>1)
#define i2cseg7x2dmsb1 (0x4E>>1)

// 8574AT optional
#define i2cseg7x2alsb2 (0x70>>1)
#define i2cseg7x2amsb2 (0x72>>1)
#define i2cseg7x2blsb2 (0x74>>1)
#define i2cseg7x2bmsb2 (0x76>>1)
#define i2cseg7x2clsb2 (0x78>>1)
#define i2cseg7x2cmsb2 (0x7A>>1)
#define i2cseg7x2dlsb2 (0x7C>>1)
#define i2cseg7x2dmsb2 (0x7E>>1)

// Aufbau der Segmente Zuordnung zu den
// Bits, 0x80 ist fuer den DOT
// *************************
//          01
//       20    02
//          40
//       10    04
//          08
//                80
// *************************

// Umrechnungstabelle ASCII -> 7 Segment
// OFFSET Asciicode 32..5F entspricht Space
// bis Z
const unsigned char siebensegtable[] =
{
   0, // 20 Space
   0x30, // 21 !
   0x22, // 22 „
   0x7f, // 23 #
   0, // 24 $
   0, // 25 %
   0, // 26 &
   0x02, // 27 ,
   0x39, // 28 (
   0x0f, // 29 )
   0, // 2A *
   0x7f, // 2B +
   0x04, // 2C ,
   0x40, // 2D -
   0x80, // 2E .
   0x30, // 2F /
   0x3f, // 30 0
   0x06, // 31 1
   0x5b, // 32 2
   0x4f, // 33 3
   0x66, // 34 4
   0x6d, // 35 5
   0x7c, // 36 6
   0x07, // 37 7
   0x7f, // 38 8
   0x67, // 39 9
   //
   0, // 3A :
   0, // 3B ;
   0, // 3C <
   0x48, // 3D =
   0, // 3E >
   0, // 3F ?
   0x5c, // 40 @
   0x77, // 41 A
   0x7c, // 42 B
   0x39, // 43 C
   0x5e, // 44 D
   0x79, // 45 E
   0x71, // 46 F
   0x67, // 47 G
   0x76, // 48 H
   0x06, // 49 I
   0x86, // 4A J
   0x74, // 4B K
   0x38, // 4C L
   0x37, // 4D M
   0x54, // 4E N
   0x5c, // 4F O
   0x73, // 50 P
   0xbf, // 51 Q
   0x50, // 52 R
   0x6d, // 53 S
   0x70, // 54 T
   0x3e, // 55 U
   0x1c, // 56 V
   0x9c, // 57 W
   0x24, // 58 X
   0x36, // 59 Y
   0x5b, // 5A Z
   0x39, // 5B [
   0x30, // 5C
   0x0f, // 5D ]
   0x08, // 5E _
   0 // 5F OHNE
};

// Umrechnen ASCII Code in Tabellenindex
unsigned int get_7seg(unsigned char
asciicode)
{
    // Umrechnen 0..255 auf
    // 7 seg Tabellenindex
```

```
    // Dabei nur Zahlen und Grossbuchstaben
    // 20..5F
    // Rest wird auf diese gemappt
    asciicode = asciicode & 0x7f; // 7 bit only
    if (asciicode < 0x20) return (0); // Sonderzeichen nicht
    if (asciicode >= 0x60) asciicode = asciicode - 0x20; // KLeinbuchstaben
    return((~siebensegtable[asciicode-0x20])&0xff); // Index zurueck
}

// Anzeige eines einzelnen ASCII Zeichen, dass wird als
// char übergeben. Ausgabe ueber den 7 Segmentbrick
// Dazu die Segmentadresse als Parameter uebergeben
// Ohne DezimalPunkt ausgeben.
void display_seg1x(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C Adresse
  Wire.write(get_7seg(ch1)); // Tabellenidnex nehmen und dann ausgeben
  Wire.endTransmission(); // Ende I2C
}

// Ausgabe ohne Umrechnung, wenn eigene Zeichen verwendet werden
// sollen. Parameter ist der Bianry Code
void display_seg1xbin(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C Adresse
  Wire.write(ch1); // Binaercode direkt am Port ausgeben
  Wire.endTransmission(); // Ende I2C
}

// Start
void setup() {
  Wire.begin(); /// I2C Initialisieren
}

void loop() {
   // Anzeigen 8574T alle potentiellen Addressen ausgeben
   // Man kann so sehen welche Adresse man besetzt hat
  display_seg1x(i2cseg7x2amsb1,‘4‘); // eigene Adresse
  display_seg1x(i2cseg7x2alsb1,‘0‘); // ausgeben
  display_seg1x(i2cseg7x2bmsb1,‘4‘); // sind immer PAARE
  display_seg1x(i2cseg7x2blsb1,‘4‘); // Zwei Befehle fuer ein BRICK
  display_seg1x(i2cseg7x2cmsb1,‘4‘);
  display_seg1x(i2cseg7x2clsb1,‘8‘); // von 40-4C
  display_seg1x(i2cseg7x2dmsb1,‘4‘);
  display_seg1x(i2cseg7x2dlsb1,‘C‘);
  // falls 8574AT vorhanden dann auch diese ausgeben
  display_seg1x(i2cseg7x2amsb2,‘7‘); // eigene Adresse
  display_seg1x(i2cseg7x2alsb2,‘0‘); // ausgeben
  display_seg1x(i2cseg7x2bmsb2,‘7‘); // hier
  display_seg1x(i2cseg7x2blsb2,‘4‘); // von
  display_seg1x(i2cseg7x2cmsb2,‘7‘); // 70..7C
  display_seg1x(i2cseg7x2clsb2,‘8‘);
  display_seg1x(i2cseg7x2dmsb2,‘7‘);
  display_seg1x(i2cseg7x2dlsb2,‘C‘);
}
```

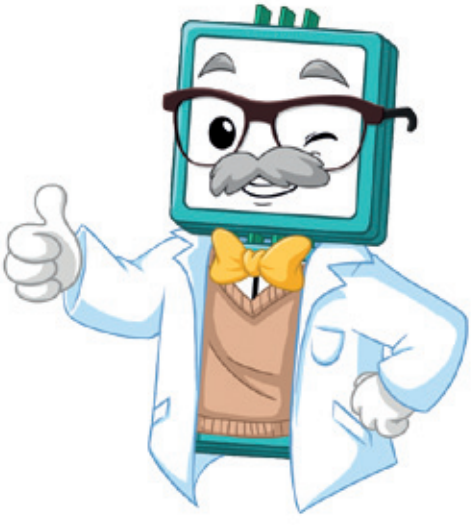
6.6 Siebensegment-Anzeige - zählen

In dem Beispiel wollen wir einen einfachen Zähler realisieren. In der Variable „counter“ wird der Zählerstand gespeichert. Das Programm erhöht den Wert alle 500ms um eins. Auf einer zweistelligen Siebensegment-Anzeige kann man aber nur maxmial den Wert 99 darstellen. Daher wird in einer Schleife der Zählerstand von „counter“ mit 99 verglichen, wenn dieser höher sein sollte, wird er wieder auf 0 zurückgesetzt. Damit zählt „counter“ von 0 bis 99 und beginnt dann wieder von vorne.
Zu beachten ist, dass die Schleife etwas länger als 500ms braucht, der Befehl „delay(500)“ wartet 500ms, aber es addieren sich die Ausführungszeiten der anderen Befehle dazu. Will man präziser arbeiten, muss man Timer verwenden und diesen abfragen.



```
// DE_16 7segmentanzeige zaehlen --  Anzeige als I2C Brick

#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
...............
siehe auch Anhang -- oder aus dem vorherigen Beispiel DE_16
...............

// Start einmal, nur I2C Initialisieren
void setup() {
  Wire.begin(); // I2C Library
}

void loop() { // In der Schleife
  char buffer[10]; // Zeichenbuffer verwenden
  static int counter = 0; // Zaehler statisch bei 0 beginnend
  sprintf(buffer,"%02d",counter++); // Umrechnen Integer auf Zeichen
  if (counter >99) counter = 0; // Zaehler soll nur zwischen 0..99 laufen
   // Counter ausgeben als zwei Ziffern, daher Buffer 0 und 1
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb Zeichen
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb Zeichen
  delay(500); // Ca. alle 500ms hochzaehlen.
} // Ende der Schleife
```

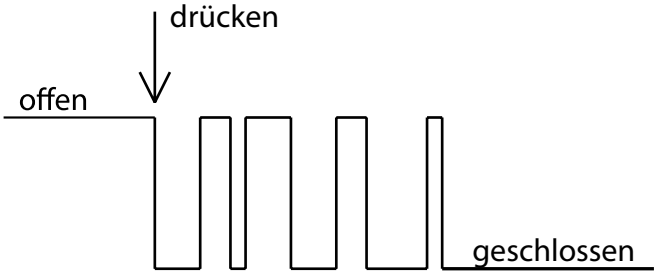
Was passiert? Auf der Siebensegment-Anzeige erscheint nacheinander die Zahlenfolge 00, 01, ... 99. Alle 1/2 Sekunde erfolgt ein Wechsel. Nach 99 kommt wieder der Wert 00.



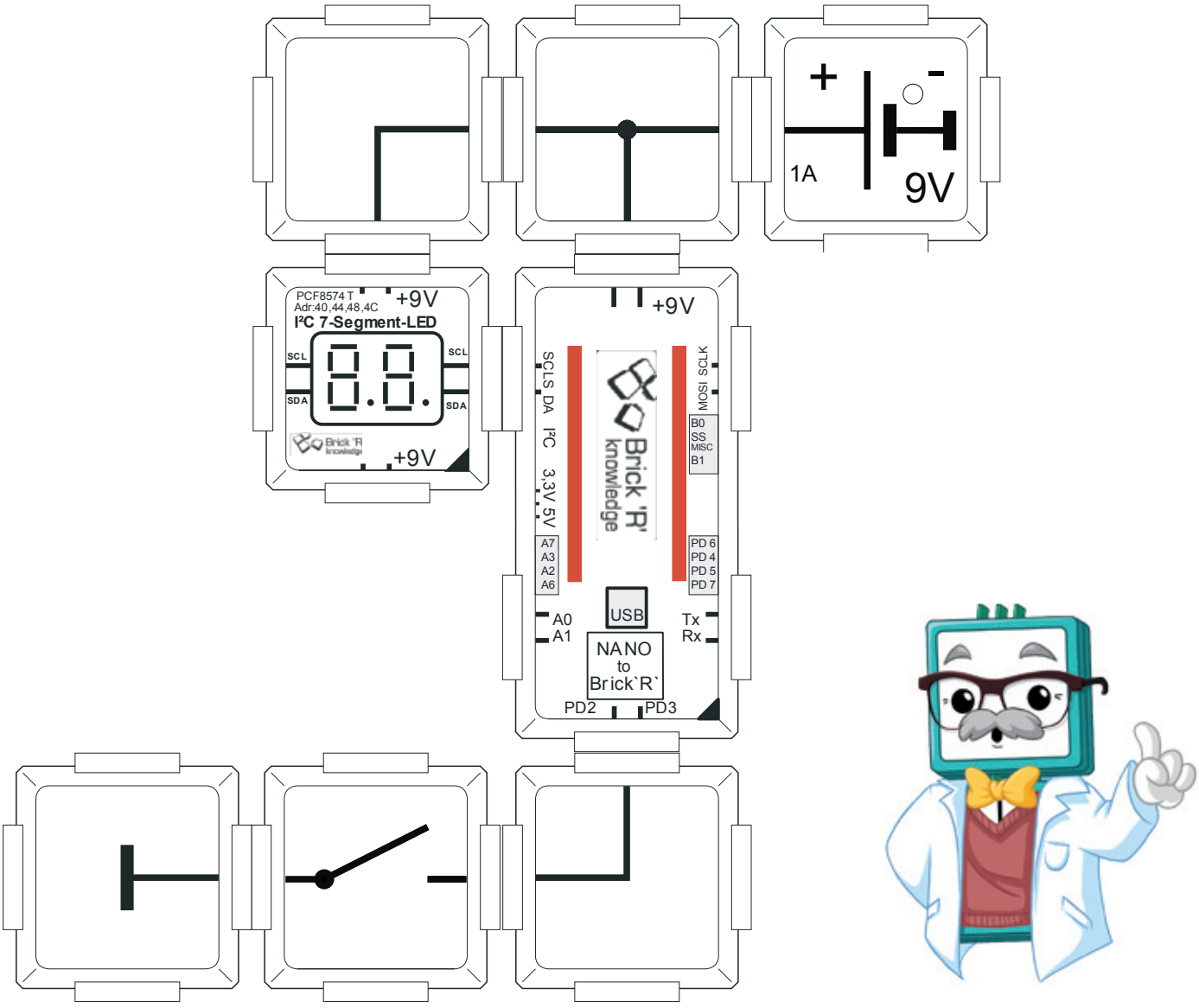
7. Tasten & Prellen

7.1 Tasten können prellen

Wenn man einen mechanischen Taster betätigt, so kann es zu sogenanntem Kontaktprellen kommen. Beim Drücken federn die Kontakte etwas, so dass die Taste mehrfach auslöst, also den Kontakt schliesst, dann erneut öffnet und wieder schließt...

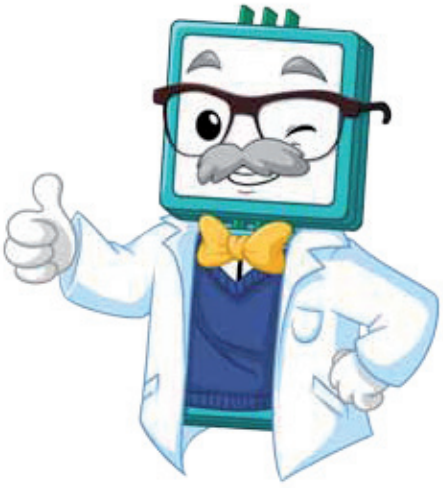


In dem Programm kann man das explizit testen. Drückt man die Taste, wird manchmal mehr als nur einmal gezählt. Manche Tasten sind allerdings auch so schnell, dass der Prozessor das nicht mitbekommt. Man muss mit der Schaltung unten ggf. etwas experimentieren, oder den Schluss nach Masse mit einem Draht statt der Taste durchführen, um den Effekt zu sehen. Die Zeitdauer des Prellens liegt in etwa im Millisekundenbereich.



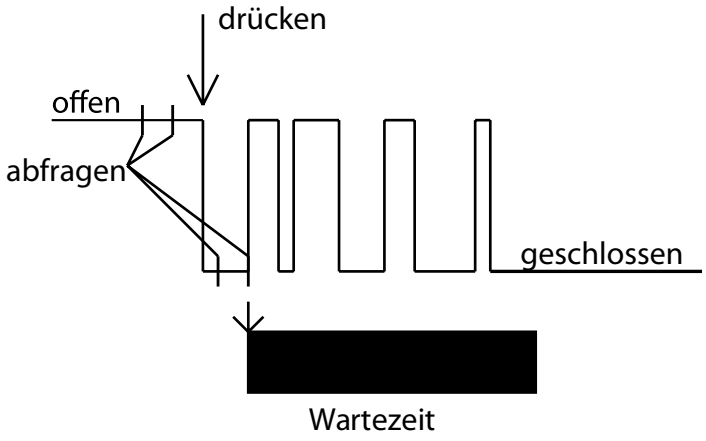
```
// DE_17 Tasten koennen Prellen
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...


// Code neu:
#define PORTTASTE 2 // hier an PD2 die Taste anschliessen

// I2C initialisieren und Taste definieren
void setup() {
  Wire.begin(); // I2C
  pinMode(PORTTASTE,INPUT_PULLUP); // Taste mit Pullup
}

void loop() { // Schleife beginnt hier
  char buffer[10]; // Dort komtm Zaehlerstand in ASCII
  static int counter = 0; // Der Testcounter
  sprintf(buffer,"%02d",counter); // Umwandlen in ASCII
  // Taste abfragen, kann prellen
  if (digitalRead(PORTTASTE)==LOW) { // wenn auf 0 geht
    while (digitalRead(PORTTASTE)==LOW) { // Solange gedrueckt
     // warten bis losgelassen !
    } // Sonst gehts gar nicht
    counter++; // dann erst zaehlen - aber auch Prellen wird gezaehlt
  }
  if (counter > 99) counter = 0; // wie vorher 0..9 zaehlern dann von vorne
   // Counter ausgeben
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb zeichen
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb zeichen
} // Ende der Schleife
```

Was passiert? Wenn man die Taste drückt wird der Zählerstand erhöht. Hier kann es aber passieren, dass mehr als einmal gezählt wird also z.B. 00,03,04,07,09 ... usw.
Das hängt hauptsächlich von der Qualität des Tasters ab. Man kann bei dem Versuch auch die Taste mit einem Stück Draht kurzzeitig überbrücken, um den Effekt deutlicher zu sehen.

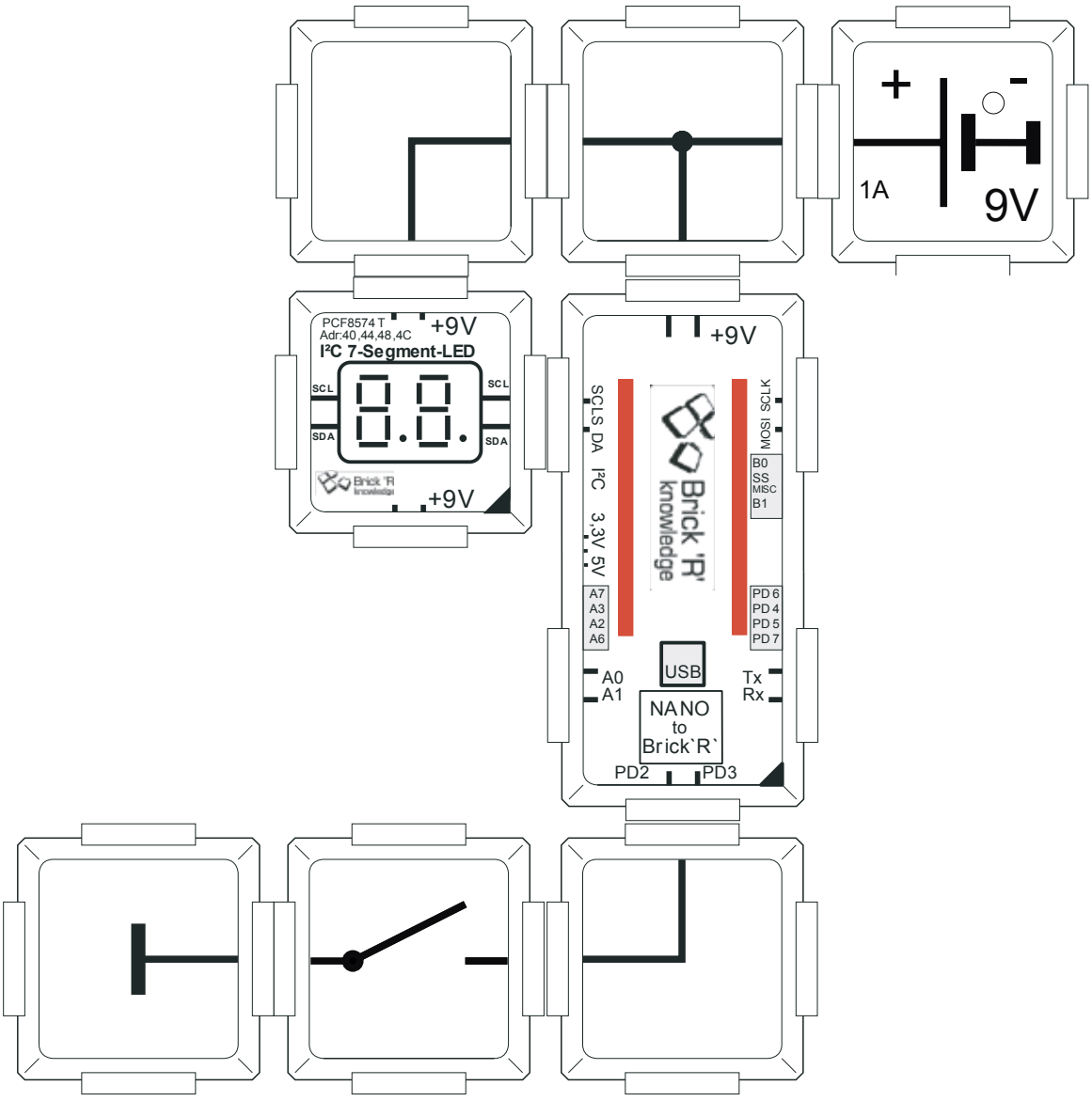


7.2 Entprellen von mechanischen Tasten per Software

Das Entprellen von Tasten kann man einfach über Delays durchführen. Man muss dazu die Zeit warten, die einen Prellzyklus dauert.



Algorithmus zum Entprellen:
Das Signal wird erstmal auf den Übergang nach Low abgefragt, dann wird wie im vorherigen Programm gewartet bis es auf High geht (also die Taste vermeindlich losgelassen wird), danach muss man aber nun warten (hier 40 ms), bis eine neue Abfrage auf einen High->Low Übergang durchführt wird. Die Dauer ist von der verwendeten Tastenart abhängig und muss ggf. experimentell ermittelt werden. So wird verhindert, dass die Prellimpulse mit ausgewertet werden.

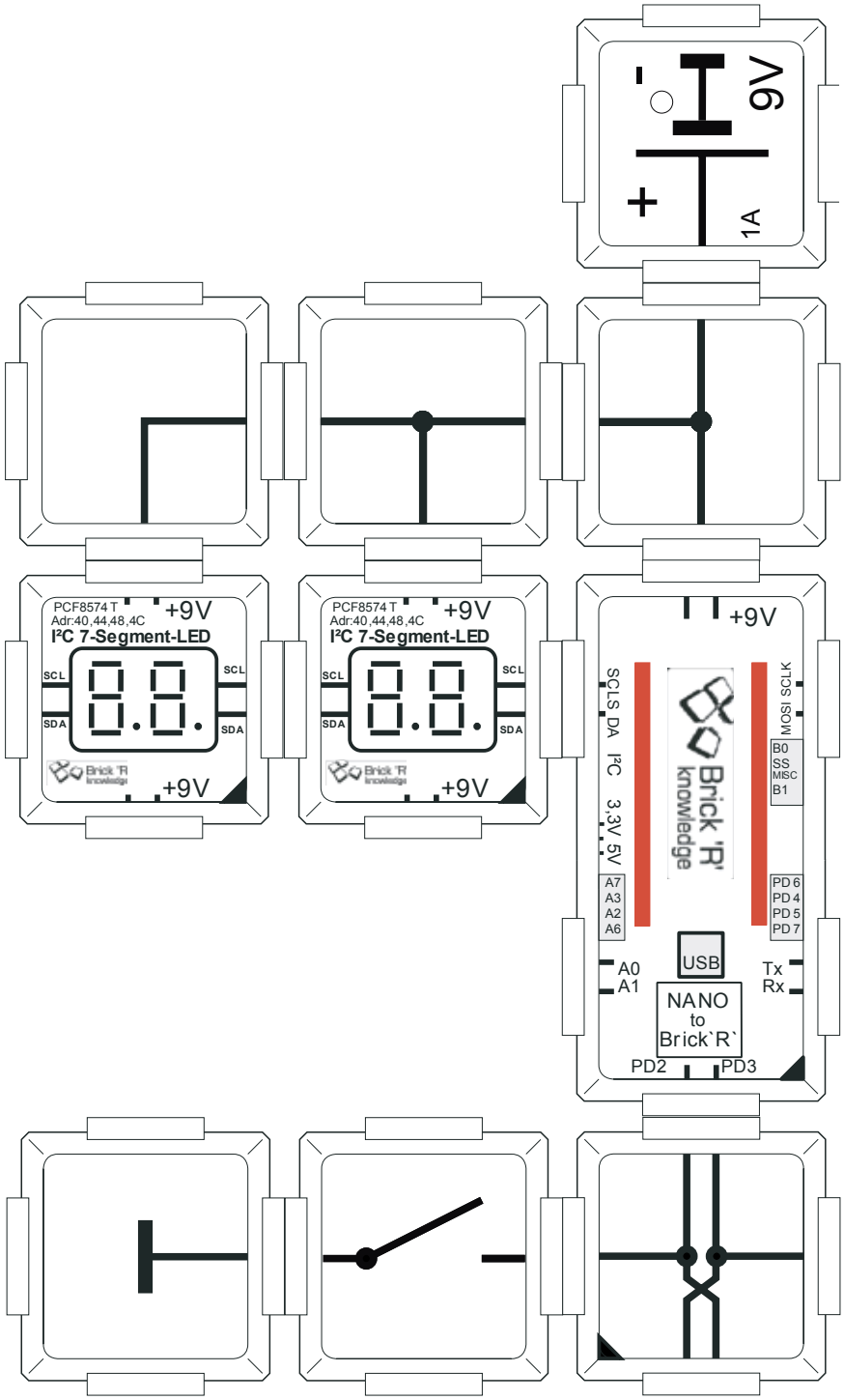


```
// DE_18 Tasten Entprellen - 7segment Anzeige als I2C Brick
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...


/// Code neu:
#define PORTTASTE 2 // Dort Taste anbringen

void setup() {
  Wire.begin(); // I2C Initialisierung
  pinMode(PORTTASTE,INPUT_PULLUP); // Taste mti Pullup versehen
}

void loop() { // Schleife
  char buffer[10];
  static int counter = 0;
  sprintf(buffer,"%02d",counter);
  // Taste abfragen, kann prellen
  if (digitalRead(PORTTASTE)==LOW) { // Pruefn auf High->low
    // Dann noch den den Low-High Übergang abwarten
    // Koennte man auch nach dem Delay machen, ist aber nicht
    // kritisch.
    while (digitalRead(PORTTASTE)==LOW) {
     // warten bis losgelassen ! bzw prellt.
    }
    counter++; // den Counter kann man noch hochzaehlen
    delay(40); // dann aber warten damit die Abfrage
    // Nicht zu schnell erneut erfolgt
  } // Ende der Abfrage High->Low
  if (counter > 99) counter = 0; // 0..99 zaehlen zum Test
   // Counter ausgeben
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb zeichen
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb zeichen
} // Ende der Schleife
```

Was passiert? Wenn man die Taste drückt, sollte bei diesem Versuch die Ausgabe auf der Anzeige bei jedem Tastendruck um genau eins erhöht werden also 00,01,02 .... usw. bei jedem Tastendruck.

7.3 Die Siebensegment-Anzeige - erweiterter Zähler

Mit zwei Siebensegment-Anzeigen kann man schon von 0 bis 9999 zählen. Dabei muss man am vorherigen Programm nicht viel ändern. Die Umrechnung des Zählerstands geschieht wieder mit dem „sprintf()“ Befehl, diesmal aber im Format %04d um vier Stellen mit Vornullen zu erzeugen. Die Taste wird auch wieder abgefragt, diesmal ist die Abfrage etwas anders aufgebaut. Der Delay wird in zwei Hälften zerlegt, was ggf. etwas mehr Sicherheit beim Abfragen des Prellvorgangs bietet. Solange man die Taste drückt, wird der Zähler hochgezählt. Die Abfrage des Überlaufs ist diesmal mit 9999 als Grenzwert eingebaut.

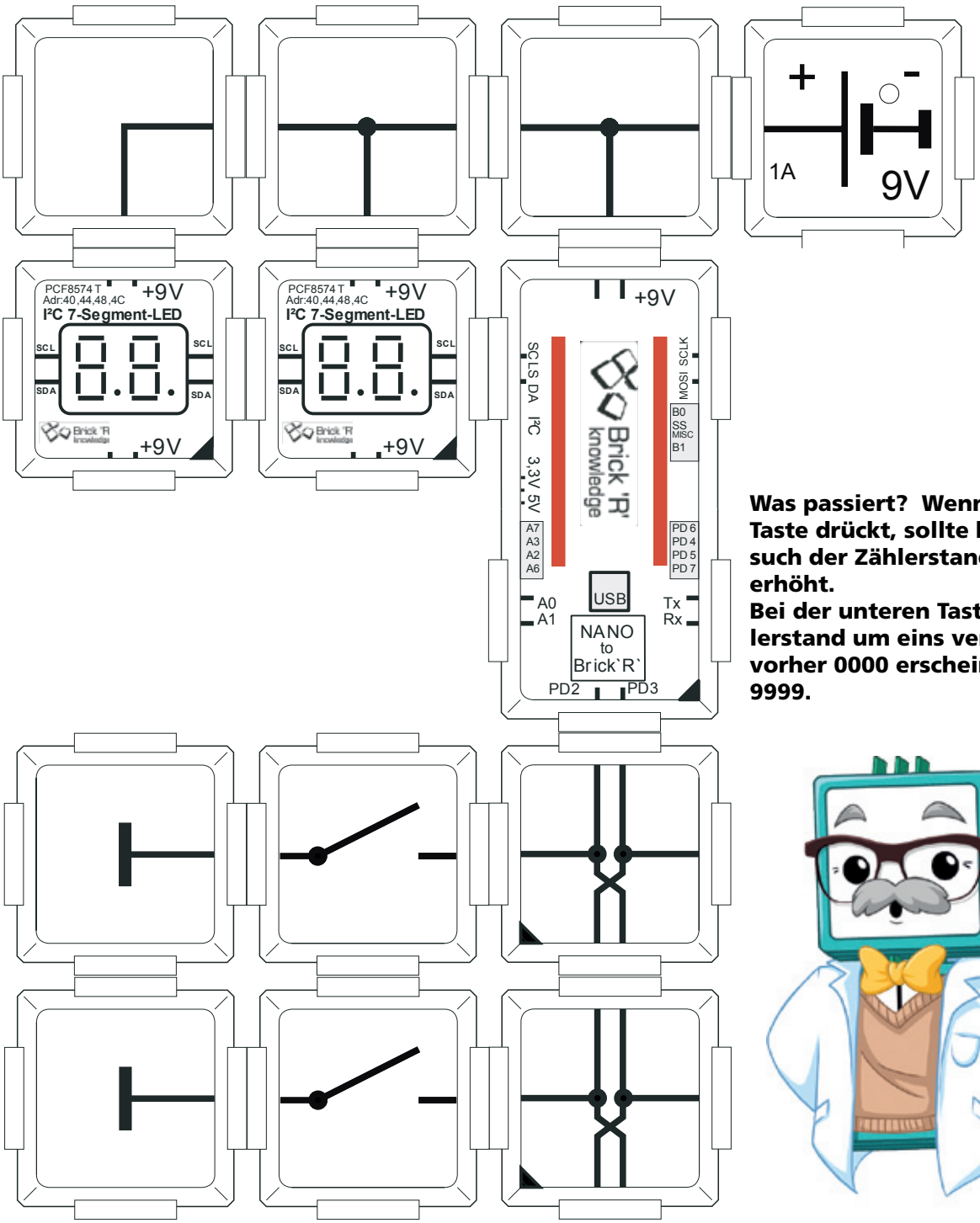


```
// DE_19 7segment Zaehler erweitert.
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...


// Code neu:
#define PORTTASTE 2 // Taste an PD2

void setup() { // Einmalige Initialisierung
  Wire.begin(); // I2C initialisieren
  pinMode(PORTTASTE,INPUT_PULLUP); // Taste mit pull-up Widerstan versehen
}

void loop() { // Schleife
  char buffer[10]; // Buffer fuer ASCII Umrechnung
  static int counter = 0; // statischer Zaaehler 0..9999
  sprintf(buffer,"%04d",counter); // nun 4 Stellen ausgeben
  // Taste abfragen, kann prellen
  if (digitalRead(PORTTASTE)==LOW) { // Uebergang High->Low
    delay(20); // 20ms warten bis stabil diemsl hier schon
    // haengt von der verwendeten Taste ab
    // ob die Zeit ausreicht.
    while (digitalRead(PORTTASTE)==LOW) { // Dann Low->High abwarten
     // warten bis losgelassen !
    } // Dann erst weiter
    counter++; // zaehlen
    delay(20); // auch nach dem loslassen nochmal warten (nicht wichtig aber
Summe 40ms)
  }
  if (counter > 9999) counter = 0; // 0..9999 zaehlen
   // Counter ausgeben
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb Zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // mittlere Stelle links
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // mittlere Stelle rechts
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb Zeichen
}
```

Was passiert? Wenn man die Taste drückt, sollte bei diesem Versuch der Zählerstand um genau eins erhöht. Diesmal 4-stellig, also 0000, 0001, 0002 ... 9999 und dann wieder von vorne.

7.4 Siebensegment-Anzeige - mit erweitertem up und down Zähler

Mit diesem Zähler kann man auf- und abzählen. Dazu werden zwei Taster verwendet, die im Prozessor abgefragt werden. Im Prinzip die gleichen Routinen wie vorher nur mit einem weiteren Zweig für das Runterzählen. Dabei wird bei einem Zählerstand < 0 wieder mit 9999 begonnen. Also nur der Bereich 00000 bis 9999 angezeigt.

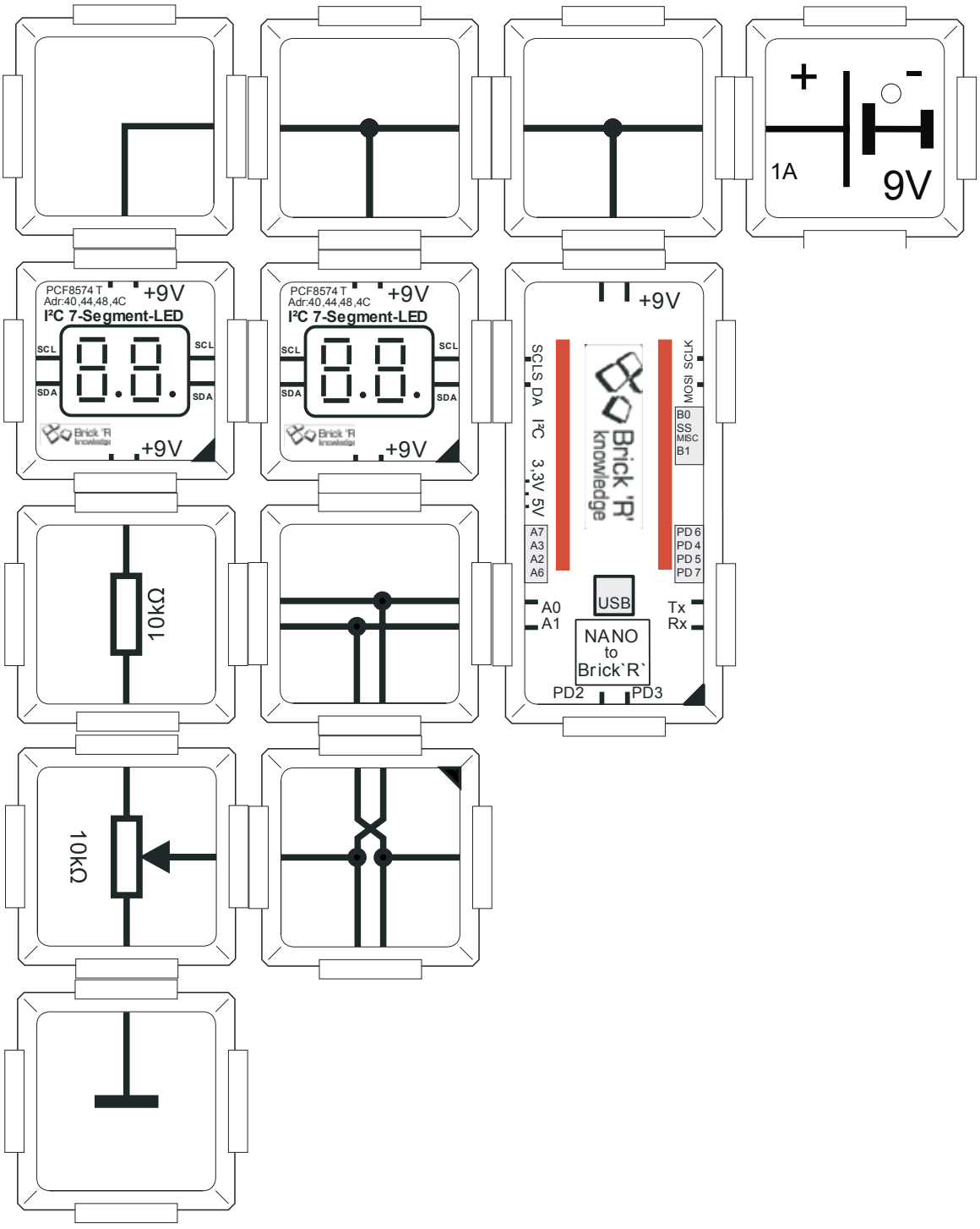


Was passiert? Wenn man die obere Taste drückt, sollte bei diesem Versuch der Zählerstand um genau eins erhöht.
Bei der unteren Taste wird der Zählerstand um eins verringert. War er vorher 0000 erscheint danach eine 9999.

```
// DE_20 7segment Zaehler Up und Down.
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...


//
// Code neu:
#define PORTTASTEUP 2 // PD2 fuers hochzaehlen
#define PORTTASTEDOWN 3 // PD3 fuers runterzaehlen

void setup() {
  Wire.begin(); // I2C wird initialisiert
  pinMode(PORTTASTEUP,INPUT_PULLUP); // beide Tasten mit
  pinMode(PORTTASTEDOWN,INPUT_PULLUP); // pullup Widerstand
}

void loop() { // Schleifenanfang
  char buffer[10]; // Buffer fuer Ergebnis in ASCII-Code
  static int counter = 0; // Up- und Down Counter
  sprintf(buffer,"%04d",counter); // Umrechnen Integer auf ASCII Sequenz
  // Taste abfragen, kann prellen
  if (digitalRead(PORTTASTEUP)==LOW) { // Uebergang High->Low
    delay(20); // 20ms warten bis stabil
    // haengt von der verwendeten Taste ab
    // ob die Zeit ausreicht.
    while (digitalRead(PORTTASTEUP)==LOW) { // dann Low auf High abwarten
     // warten bis losgelassen !
    }
    counter++; /// Hier Zaehler hochzaehlen
    delay(20); // auch nach dem loslassen nochmal warten
  }
  if (digitalRead(PORTTASTEDOWN)==LOW) {  // High->Low bei Downtaste
    delay(20); // 20ms warten bis stabil
    // haengt von der verwendeten Taste ab
    // ob die Zeit ausreicht.
    while (digitalRead(PORTTASTEDOWN)==LOW) { // warten auf Low->high
     // warten bis losgelassen !
    }
    counter--; // Diesmal runterzaehlen
    delay(20); // auch nach dem loslassen nochmal warten
  }
  if (counter < 0) counter = 9999;
  if (counter > 9999) counter = 0;
   // Counter ausgeben mit aktuellem Stand
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // ..
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // ..
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb zeichen
}
```

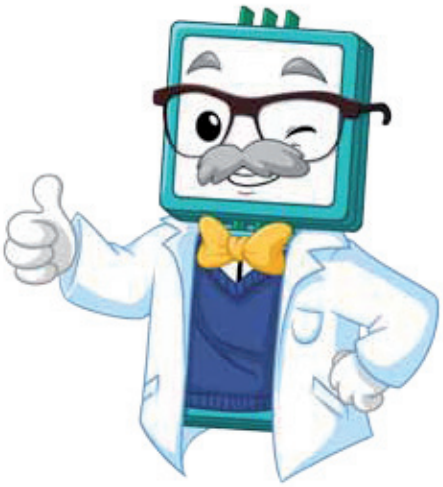
7.5 AD Umsetzer und Display mit Siebensegment-Anzeige als Voltmeter

Der Wert des A/D-Umsetzers wird eingelesen und im Programm in Milivolt umgerechnet. Es ergibt sich ein Bereich von 0 bis 5000mV. Dabei müsste man den Bereich noch kalibrieren. Es wird die Spannung am Schleifer des Potentiometers angezeigt. Den Wert von 5000 wird man hier aber nicht erreichen, wenn man eine 9V Batterie verwendet. Der Spannungsteiler mit 10kOhm vom einzelnen Widerstand und den 10kOhm vom Potentionmeter läßt nur einen kleineren Bereich von 0 bis 1/2 der Batteriespannung zu, also i.A. 4.5Volt. Da die Eingänge des Arduino Nano nicht für mehr als 5V geeignet sind, ist diese Schutzmaßnahme nötig.



```
// DE_21 7segment Voltmeter
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...



// Code neu:

void setup() { // Setup einmal
  Wire.begin(); //I2C wird gebraucht
}

void loop() { // Schleifenstart
  char buffer[10]; /// Umrechnung auf ASCII
  int poti = analogRead(A0); // a1,a2,a3 0..1023 Hier A0
  double milivolt = 0; // ZWischenvariable
  milivolt = ((double)poti*5000.0)/1024.0; // Umrechnen Wert in Militvolt
  sprintf(buffer,"%04d",(int)milivolt); // Umrechnen in ASCII Code 4stellig
  delay(50); // damit die Anzeige nicht so springt, sonst zu schnell
  // Counter ausgeben
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // ..
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // ..
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb zeichen
}
```

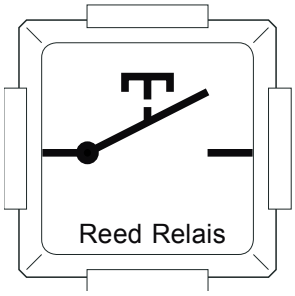
Was passiert? Im Display wird ein Wert zwischen 0000 und maximal 5000 angezeigt. Wenn man das Poti dreht, sollte man fast auf 0000 kommen und maximal auf 5000. Dabei liegt bei einer Batteriespannung von 9000mV der maximale Anzeigewert bei 4500mV (halbe Spannung!).



8. Relais

8.1 Reed Relais

Relais werden auch heute noch eingesetzt, wenn entweder große Leistungen geschalten werden oder es auf eine gute Trennung zwischen dem Anrege- und Schaltkreis ankommt. Die Übertragung der Information ob ein Relaiskontakt schließt oder nicht, geschieht über ein magnetisches Feld.
Eine Spezialausführung des normalen Relais ist das Reed-Relais, mit den speziell dort verwendeten Reedkontakten.



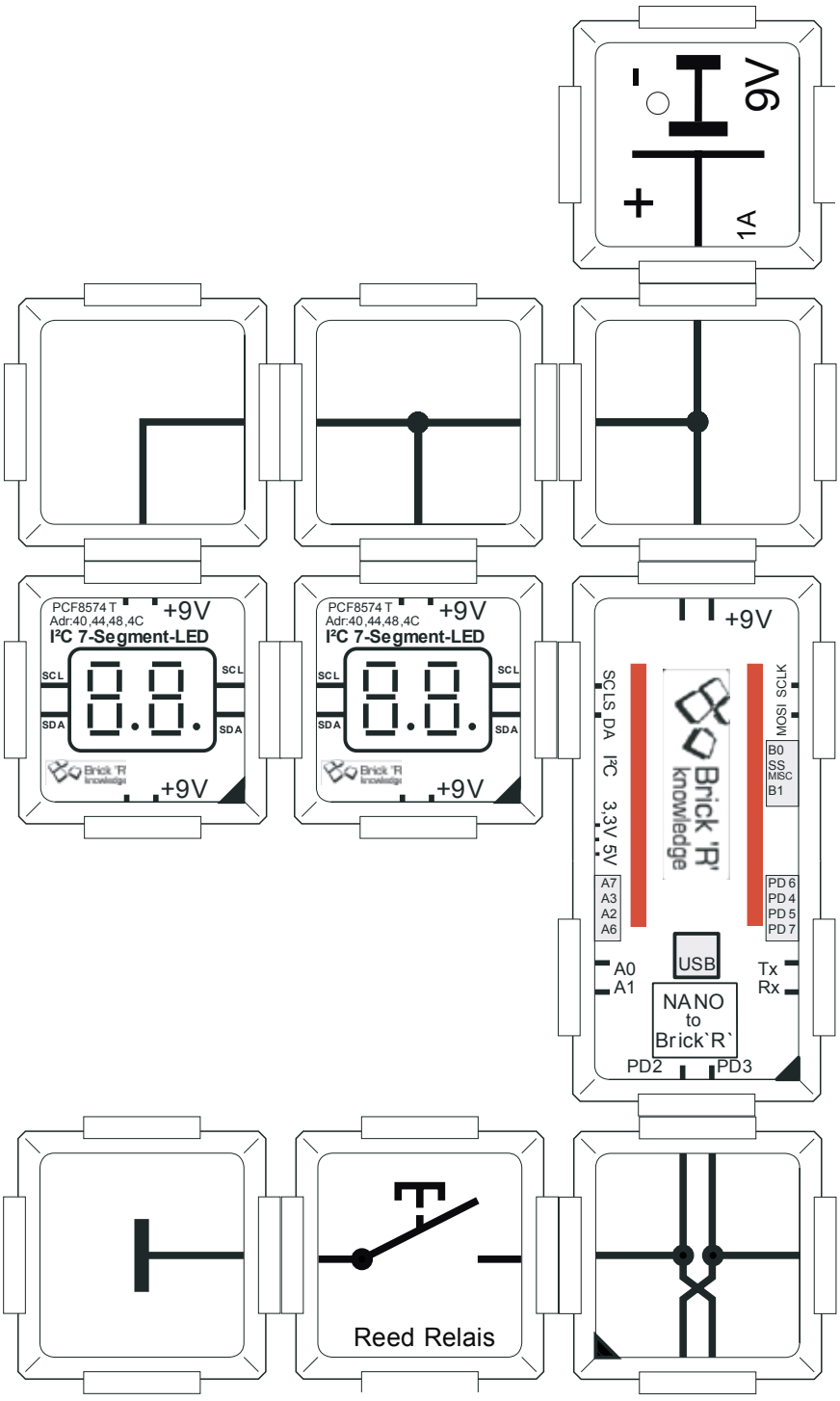
Der Reedkontakt wird über ein Magnetfeld ausgelöst, bei unserer Ausführung genügt ein kleiner Stabmagnet den man in Längsrichtung zum Reedkontakt anlegt. Der Reedkontakt ist bei diesem Brick sichtbar, in einem kleinen Glasröhrchen untergebracht. Wird ein Magnetfeld in der richtigen Position und Stärke zum Kontakt angesetzt, ziehen sich die beiden Kontakte an und der Kontakt schließt.
Man kann nun auf einfache Weise berührungslose Schalter bauen. Gerne werden solche Kontakte zum Beispiel zur Diebstahlsicherung verwendet.
Am Türrahmen bringt man den Reedkontakt an, die Leitungen führen dann zur Auswerteelektronik. Die Türe bekommt einen kleinen Magneten. Ist die Türe geschlossen, so muss der Magnet am Reed-Kontakt anliegen, so dass dieser auslöst. Der Stromkreis ist geschlossen, wenn man ihn entsprechend beschaltet, so kann auch feststellt werden, ob die Türe geschlossen ist.
Wird die Türe geöffnet, löst sich auch der Reed-Kontakt wieder und die Trennung kann gemessen und eine Aktion ausgelöst werden.. Das gleiche gilt beispielsweise auch für Fensterkontakte. Der Reedkontakt sitzt am Fensterrahmen und der Magnet an dem beweglichen Teil des Fensters. So lässt sich feststellen, ob das Fenster geschlossen ist.

In einer Zentrale könnte man so alle Kontakte zusammenführen und überprüfen, ob eine sicherheitsrelevante Türe oder Fenster nach Arbeitsschluss noch geöffnet ist. Für einen Einbruchalarm reicht das allerdings noch nicht, hier müsste man beispielsweise noch Bruchdetektoren oder IR-Bewegungsmelder einsetzen. Schließlich wird ein Einbrecher das Fenster von Außen nicht auf herkömmlichen Weg öffnen. Gesicherte Türen lösen auf die gleiche Weise aus und erhöhen so die Sicherheit innerhalb des Gebäudes.
Hier ist ein großer Spielraum.
Weitere Anwendungen kommen aus der Industrie, zum Beispiel kann man damit Umdrehungsmesser bauen. Der Kontakt sitzt aussen, auf dem Rad ist ein Magnet angeordnet, jede Umdrehung löst dann den Kontakt aus.

8.2 Platz für Notizen:

8.3 Reedrelais steuert Anzeige

Mit dem Reedrelais kann man die Anzeige steuern. Hier wird bei einer Annäherung des Magnets der Zählerstand um eins erhöht. Wird der Magnet an einem Rad befestigt, lässt sich so die Zahl der Umdrehungen anzeigen. Reedkontakte muss man prinzipiell entprellen, auch wenn die Kontakte meist sehr gut konstruiert sind und durch die magnetischen Eigenschaften die Presszeit extrem kurz sein kann. Hier ist unser normales Zählprogramm eingesetzt.

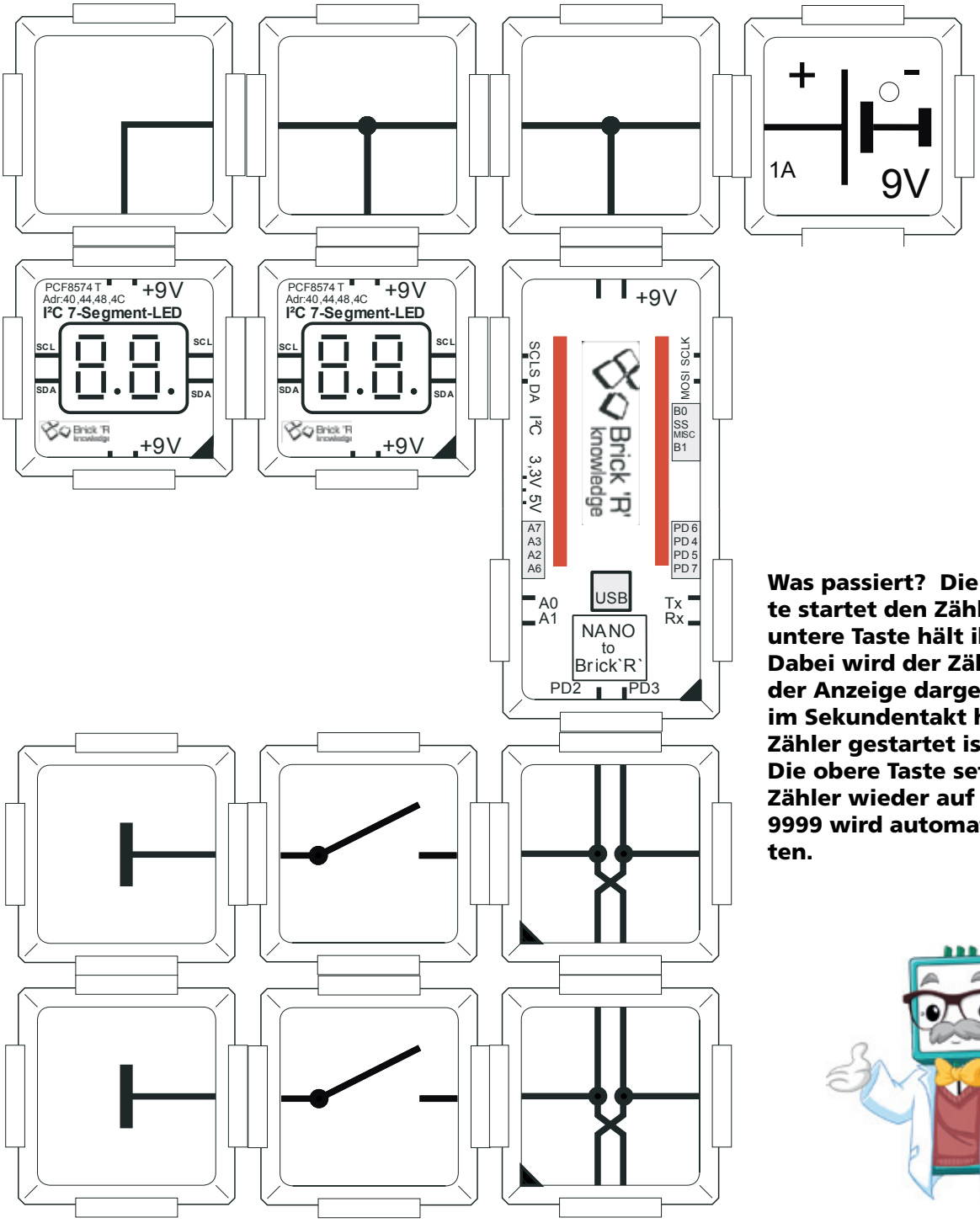


```
// DE_22 7segment Zaehler mit Reed Relais.
#include <Wire.h>

// 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... wie vorher ...



// Code neu:
#define PORTRELAIS 2 // dort das Reedrelais an PD2

void setup() {
  Wire.begin(); //I2C initialisieren
  pinMode(PORTRELAIS,INPUT_PULLUP); // PD2 mit Pullup fuer den Kontakt
}

void loop() {
  char buffer[10]; // Ausgabe des Zaehlerstandes in ASCII
  static int counter = 0; // statischer Zaehler
  sprintf(buffer,"%04d",counter); // Umrechnen INT in ASCII Code
  // Reed Relais abfragen, kann prellen
  if (digitalRead(PORTRELAIS)==LOW) { // wenn high->low
    delay(20); // 20ms warten bis stabil // entprellen
    // auch Reed Relais koennen prellen
    while (digitalRead(PORTRELAIS)==LOW) { // dann warten low->high
     // warten bis losgelassen !
    }
    counter++; // dann Zaehler hoch
    delay(20); // auch nach dem loslassen nochmal warten
  }
  if (counter > 9999) counter = 0; // Range ist 0..9999
   // Counter ausgeben
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // ..
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // ..
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb zeichen
}
```

Was passiert? Wenn man einen Magneten, der stark genug ist an das Reedrelais annähert, wird der Zählerstand in der Anzeige um eins erhöht. Soll der Zähler nochmal weiterspringen, so muss man den Magneten erst wieder weit genug entfernen und erneut annähern.

8.4 Stoppuhr mit 7-Segment-Anzeigen

Die 7-Segment-Anzeige als Stoppuhr mit zwei Tastern Start/Stopp und einer Auflösung von 1/10 Sec = 100ms. Bisher haben wir immer Tastendrücke oder die Tastendruckdauer gezählt. Die klassische Stoppuhr ist aber auch nicht weiter kompliziert. Eine Taste löst den Zählvorgang aus, die andere hält ihn wieder an. Dazu gibt es eine weitere statische Variable „startflag“ genannt. Wenn diese auf 1 ist, wird gezählt, sonst nicht. Eine Taste setzt den Wert auf 1, die andere auf 0. Entprellen müsste man eigentlich in dem Beispiel nicht, es ist aber eine gute Angewohnheit, Tasten immer zu entprellen, wenn diese eine andere Funktion übernehmen sollen. Die Zeitnahme ist so noch nicht genau, denn unser „delay(100)“ bestimmt die 100ms zwar recht genau, aber die weiteren Befehle werden nicht berücksichtigt. Man müsste den delay etwas kleiner als 100ms angeben, dann ist die Uhr aber ggf. schon zu schnell. Es gibt aber noch eine elegantere Möglichkeit, Zeitabschnitte zu bestimmen, doch dazu später mehr.

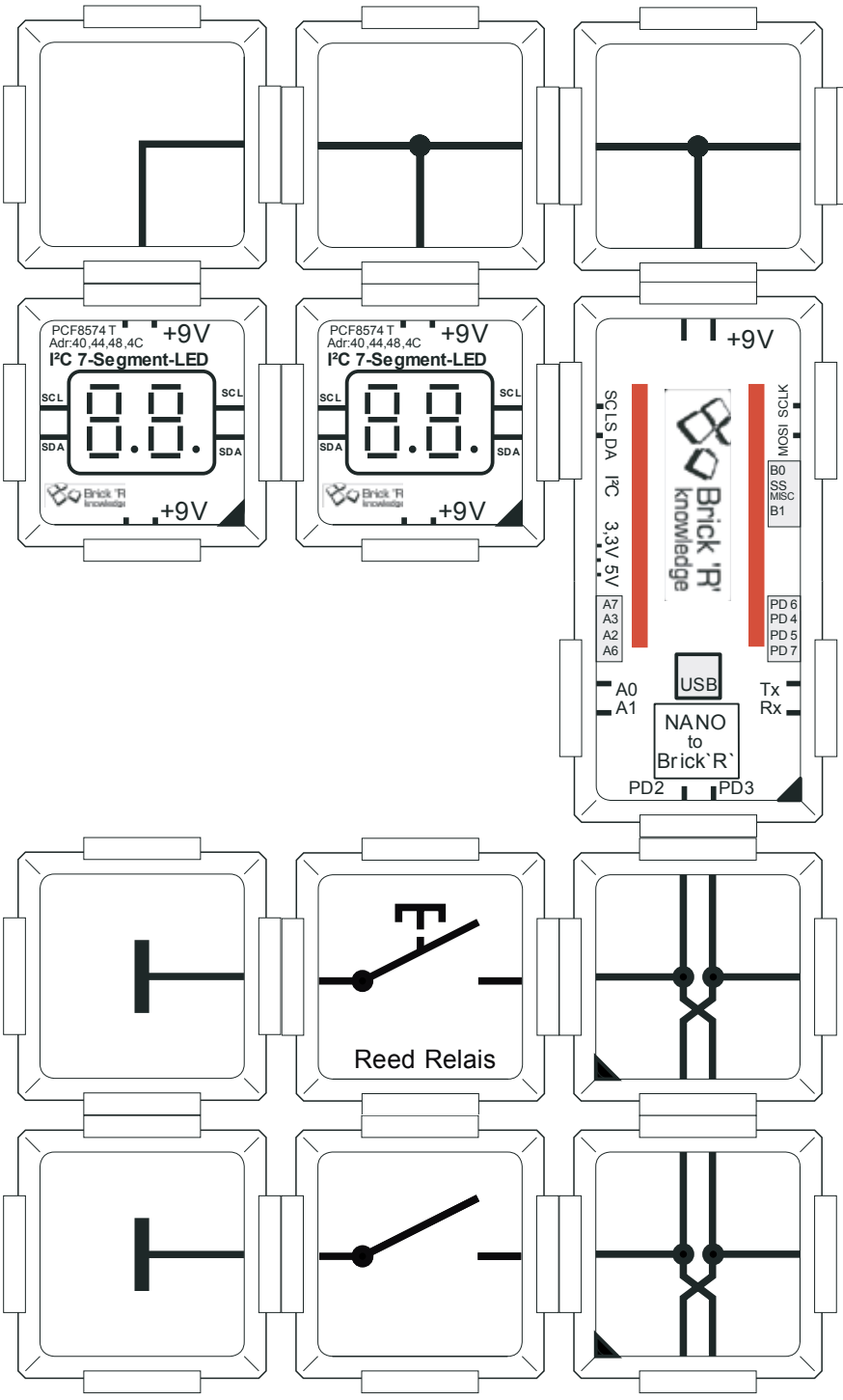


Was passiert? Die obere Taste startet den Zählvorgang, die untere Taste hält ihn wieder an. Dabei wird der Zählerstand auf der Anzeige dargestellt und zählt im Sekundentakt hoch, wenn der Zähler gestartet ist.
Die obere Taste setzt auch den Zähler wieder auf 0 zurück. Bei 9999 wird automatisch angehalten.

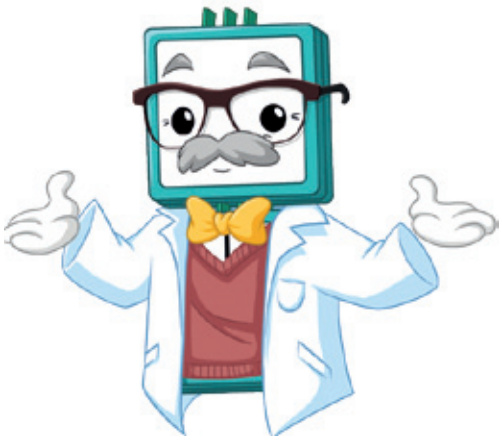
```
// DE_23 7segment Anzeige Stoppuhr
#include <Wire.h>
..... wie vorher ...
// Code neu:
#define PORTSTART 2 // PD2 fuer Start
#define PORTSTOP  3 // PD3 fuer Stop

void setup() {
  Wire.begin(); // I2C initialisieren
  pinMode(PORTSTART,INPUT_PULLUP); // PD2 Pullup
  pinMode(PORTSTOP,INPUT_PULLUP); // PD3 Pullup
}

void loop() { // Schleifenstart
  char buffer[10]; // BUffer fuer Umrechnung nach ASCII
  static int startflag = 0; // 0=stop 1=Uhrlaeuft
  static int stopzeit = 0; // der Zeitzaehler
  sprintf(buffer,"%04d",stopzeit); // Umrechnen in ASCII
  // Reed Relais abfragen, kann prellen
  if (digitalRead(PORTSTART)==LOW) { // Low->High
    delay(20); // 20ms warten bis stabil
    // auch Reed Relais koennen prellen
    while (digitalRead(PORTSTART)==LOW) { // und wieder High->Low
     // warten bis losgelassen !
    }
    startflag = 1; // dann beginnt der Zaehlvorgang
    stopzeit = 0; // Bei 0 starten, also gleichzeitig Reset
    delay(20); // auch nach dem loslassen nochmal warten
  } // Abfrage Start Ende
  // Achtung TASTE muss laenger gehalten werden ! > 1/10 sec
  if (digitalRead(PORTSTOP)==LOW) {  // Stop auslösen
    delay(20); // 20ms warten bis stabil
    // auch Reed Relais koennen prellen
    while (digitalRead(PORTSTART)==LOW) { // und High-Low warten
     // warten bis losgelassen !
    }
    startflag = 0;
    delay(20); // auch nach dem loslassen nochmal warten
  } // Ende Abfrage Stop der Zeitnahme
  if ((startflag == 1) && (stopzeit < 9999)) { // Zaehlt bis 9999
    stopzeit++; // wenn startflag=1 dann erhoehren
    delay(100); // eigentlich <100 100ms !! MUSS GENAU SEIN
    // mit CPU timer genauer zu realisieren
  } // Ende Abfrage startflag
  // Counter ausgeben
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // ..
  display_seg1xbin(i2cseg7x2amsb1,get_7seg(buffer[2]) & 0x7f); // Mit PUNKT
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb zeichen
} // Ende Schleife
```

8.5 Stoppuhr mit Reedrelais Auslösung

Diese Variante der Stoppuhr lässt sich mit einem Reedrelais anhalten. Dazu wird der Magnet über das Reed-Relais geführt. Mit so einer Anordnung kann man auch Zeiten gut stoppen, wenn ein Gegenstand vorbeikommt. Idealerweise wird in der Praxis dann mit zwei Reed-Relais gearbeitet.
Hier kann man sich einmal unterschiedliche Möglichkeiten der Zeitnahme ausdenken.
Das Programm ist identisch zu dem vorherigen, denn am Algorithmus ändert sich nichts.



Was passiert? Die Reedrelais startet den Zählvorgang. Dazu muss man einen Magneten annähern und wieder entfernen (erst dann wird hochgezählt). Gleichzeitig wird der Zähler auf 0000 gesetzt.
Die untere Taste hält den Zähler wieder an. Dabei wird der Zählerstand auf der Anzeige dargestellt und zählt im Sekundentakt hoch, wenn der Zähler gestartet ist.
Bei 9999 wird automatisch angehalten.



```
// DE_23 7segment Anzeige Stoppuhr
#include <Wire.h>
..... wie vorher ...
// Code neu:
#define PORTSTART 2 // PD2 fuer Start
#define PORTSTOP  3 // PD3 fuer Stop

void setup() {
  Wire.begin(); // I2C initialisieren
  pinMode(PORTSTART,INPUT_PULLUP); // PD2 Pullup
  pinMode(PORTSTOP,INPUT_PULLUP); // PD3 Pullup
}

void loop() { // Schleifenstart
  char buffer[10]; // BUffer fuer Umrechnung nach ASCII
  static int startflag = 0; // 0=stop 1=Uhrlaeuft
  static int stopzeit = 0; // der Zeitzaehler
  sprintf(buffer,"%04d",stopzeit); // Umrechnen in ASCII
  // Reed Relais abfragen, kann prellen
  if (digitalRead(PORTSTART)==LOW) { // Low->High
    delay(20); // 20ms warten bis stabil
    // auch Reed Relais koennen prellen
    while (digitalRead(PORTSTART)==LOW) { // und wieder High->Low
     // warten bis losgelassen !
    }
    startflag = 1; // dann beginnt der Zaehlvorgang
    stopzeit = 0; // Bei 0 starten, also gleichzeitig Reset
    delay(20); // auch nach dem loslassen nochmal warten
  } // Abfrage Start Ende
  // Achtung TASTE muss laenger gehalten werden ! > 1/10 sec
  if (digitalRead(PORTSTOP)==LOW) {  // Stop auslösen
    delay(20); // 20ms warten bis stabil
    // auch Reed Relais koennen prellen
    while (digitalRead(PORTSTART)==LOW) { // und High-Low warten
     // warten bis losgelassen !
    }
    startflag = 0;
    delay(20); // auch nach dem loslassen nochmal warten
  } // Ende Abfrage Stop der Zeitnahme
  if ((startflag == 1) && (stopzeit < 9999)) { // Zaehlt bis 9999
    stopzeit++; // wenn startflag=1 dann erhoehren
    delay(100); // eigentlich <100 100ms !! MUSS GENAU SEIN
    // mit CPU timer genauer zu realisieren
  } // Ende Abfrage startflag
  // Counter ausgeben
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb zeichen/ LINKE Anzeige
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // ..
  display_seg1xbin(i2cseg7x2amsb1,get_7seg(buffer[2]) & 0x7f); // Mit PUNKT
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb zeichen
} // Ende Schleife
```

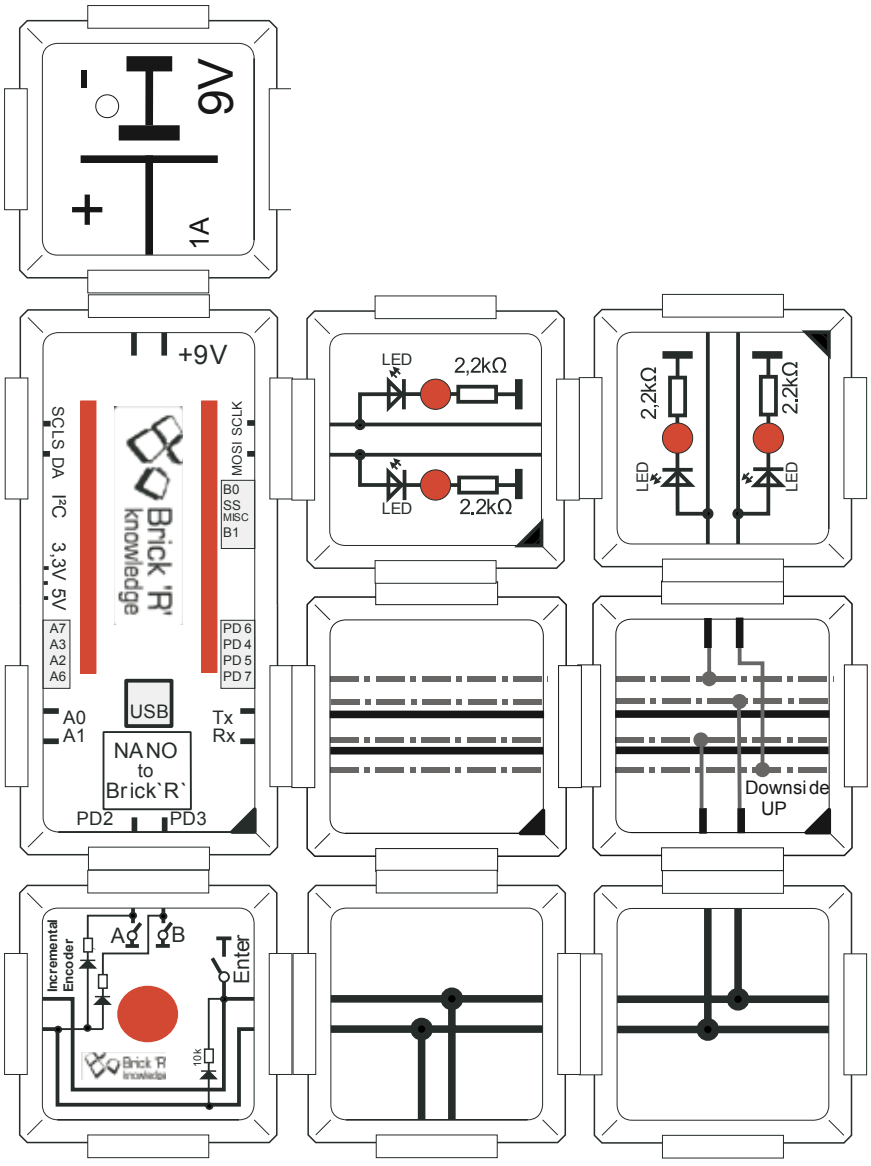
9. Rotationsgeber

9.1 Der Incrementalgeber Brick

Incremental Drehgeber, oder englisch incremental encoder. Solche Bausteine werden gerne bei Bedienteilen verwendet. Durch zwei phasenverschobene Kontakte lässt sich z.B. mit einem Prozessor die Drehrichtung bestimmen. Durch den Knopfdruck, wird zusätzlich ein Tastkontakt geschlossen, mit dem man auch eine beliebige Aktion auslosen kann. Je nach Bautyp werden pro Umdrehung ca. 18 bis 36 Kontaktvorgänge ausgelöst. Der Geber hat zwei Ausgänge die mit A und B bezeichnet werden. Wenn man nun an dem Geber dreht, dann werden die Kontakte auf bestimmte Weisse geschlossen

Winkel	A	B
0	0	0
1	1	0
2	1	1
3	0	1

Anhand der Reihenfolge kann man nun Vor- oder Rückrichtung eindeutig bestimmen. Es wird immer nur EIN Kontakt dazu- oder weggeschaltet, dadurch wird die Drehrichtung eindeutig.
Unser Brick hat zusätzlich einen Druckkontakt der, wie bei einer Taste, beliebig verwenden werden kann. Die Kontakte A und B muss man nicht entprellen !



Portbelegung

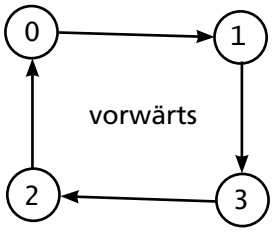
Kürzel	Port	Parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

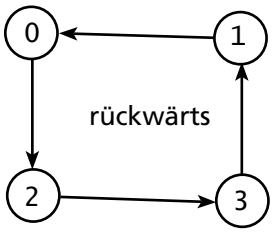
```
// DE_24 Incrementalgeber

#define SWITCHA 2 // Kontakt A
#define SWITCHB 3 // Kontakt B
#define SWITCHT 4 // Tastenkontakt
#define PULLUP 5 // Dient nur als Pullup
#define PORTLED6 6 // Anzeige von A
#define PORTLED7 7 // Anzeige von B
#define PORTMOSILED 11 // Tasteenzustand
#define PORTSCLKLED 13 // komplementaer

void setup() {
  pinMode(SWITCHA,INPUT_PULLUP); // Alle Tasten mit Pullup
  pinMode(SWITCHB,INPUT_PULLUP); // Alle Tasten mit Pullup
  pinMode(SWITCHT,INPUT_PULLUP); // Alle Tasten mit Pullup
  pinMode(PULLUP,OUTPUT); // Port 7 als Ausgang schalten
  digitalWrite(PULLUP,HIGH); // zusaetzlich pullup
  pinMode(PORTLED6,OUTPUT); // Port 6 als Ausgang schalten
  pinMode(PORTLED7,OUTPUT); // Port 7 als Ausgang schalten
  pinMode(PORTMOSILED,OUTPUT); // PORTMOSILED als Ausgang schalten
  pinMode(PORTSCLKLED,OUTPUT); // PORTSCLKLED als Ausgang schalten
}

void loop() {
  int a= digitalRead(SWITCHA); // A einlesen
  int b= digitalRead(SWITCHB); // B einlesen
  int sw = digitalRead(SWITCHT); // Taste einlesen
  if (a==HIGH) { // hier statisch ausgeben
    digitalWrite(PORTLED6,HIGH); // LED 6 zeigt A an
  } else {
    digitalWrite(PORTLED6,LOW); // und 0 wenn A=0
  }
  if (b==HIGH) {
    digitalWrite(PORTLED7,HIGH); // LED 7 zeigt B an
  } else {
    digitalWrite(PORTLED7,LOW); // und 0 wenn B=0
  }
  if (sw==HIGH) { // Auch die Taste soll was tun
    digitalWrite(PORTMOSILED,HIGH); // Eine LED auf 1
    digitalWrite(PORTSCLKLED,LOW); // dier andere 0
  } else {
    digitalWrite(PORTMOSILED,LOW); // und umgekehrt
    digitalWrite(PORTSCLKLED,HIGH); // wenn gedrueckt.
  }
}
```

Was passiert? Beim Drehen zeigen die LEDs an Port 6 und 7 den Graycode des Incrementalgebers an (rechte senkrechte LED-Gruppe). Man sollte dazu ggf. ganz langsam drehen, also auch zwischen dem Einrastvorgang. Dann kann man die Schaltvorgänge besser beobachten. Wenn man auf den Regler drückt, dann wechseln die beiden LEDs am Port MOSI und SCLK (linke horizontale LED-Gruppe).

9.2 Incrementalgeber mit Wertausgabe

Das Programm wird nun erweitert. Hier wird ein Zähler rauf- und runtergezählt und der Wert auf der Konsole des PC ausgegeben. Dazu müssen nun die einzelnen Phasen in Zähleraktionen umgerechnet werden. In der Praxis entwickelt man dazu ein Statediagramm. Im Programm gibt es zwei Variable, die den Durchlauf steuern - „state“ hält den augeblicklichen Status von A und B fest und hat die Werte 0,1,2,3 für alle Kombinationen. „oldstate“ merkt sich den Wert der vorherigen Abfrage. Damit kann man die Drehrichtung bestimmen. A=1 dann state =1 oder 3 und B=1 dann state=2 oder 3. Sagt noch nichts über die Drehrichtung aus. A und B können ja nur in bestimmter Reihenfolge ablaufen, wie wir vorher gesehen haben. Diese Code nennt man auch Gray-Code, er ist in der Technik ein sehr wichtiger Code und wird häufig verwendet.

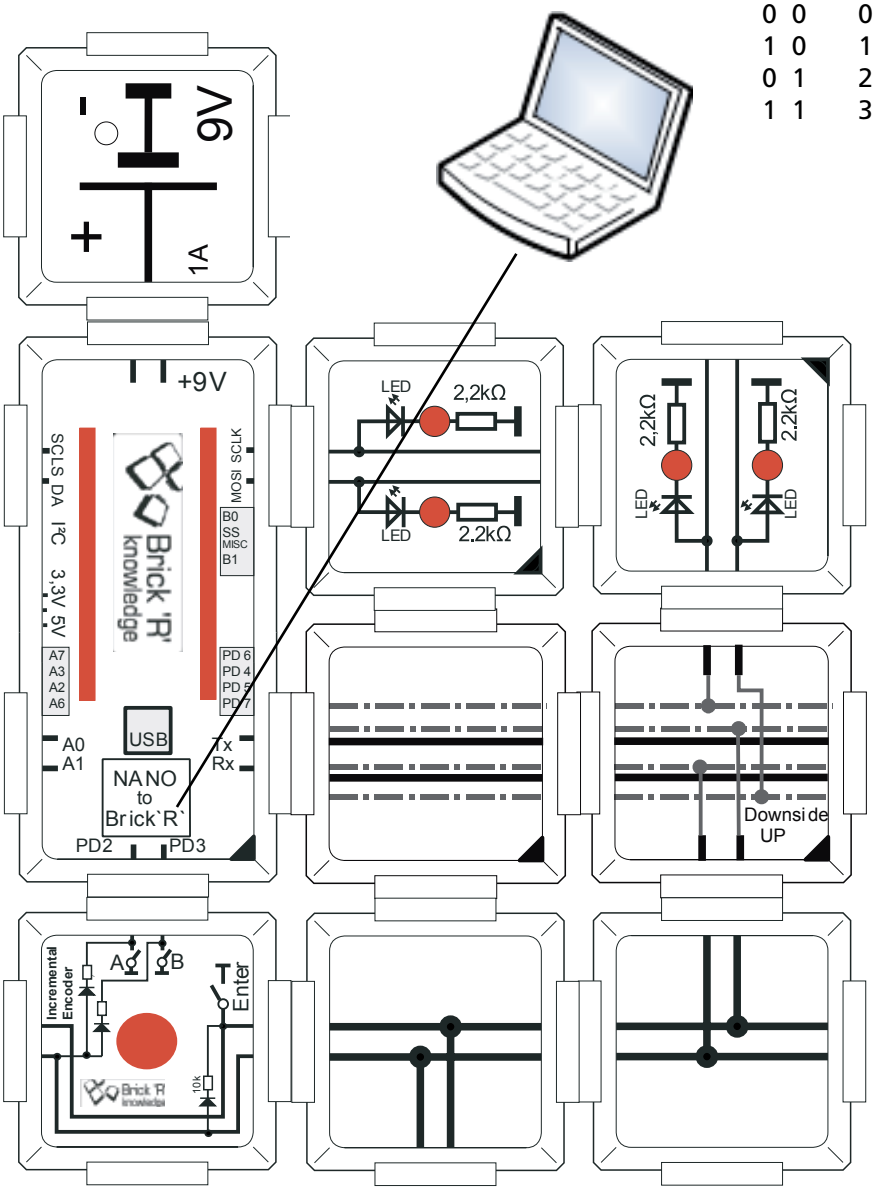




A	B	state
0	0	0
1	0	1
0	1	2
1	1	3

ACHTUNG:
Terminal aktivieren!

Dazu beim Arduino Programm, CTRL und SHIFT und M drücken (CTRL=STRG je nach Tastatur).
Dann poppt ein Fenster hoch in dem die Infos stehen.
Ggf. muss man die Baudrate im Terminalfenster auf 9600 einstellen, so daß diese mit unserer im Programm übereinstimmt.



Portbelegung

Kürzel	Port	Parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

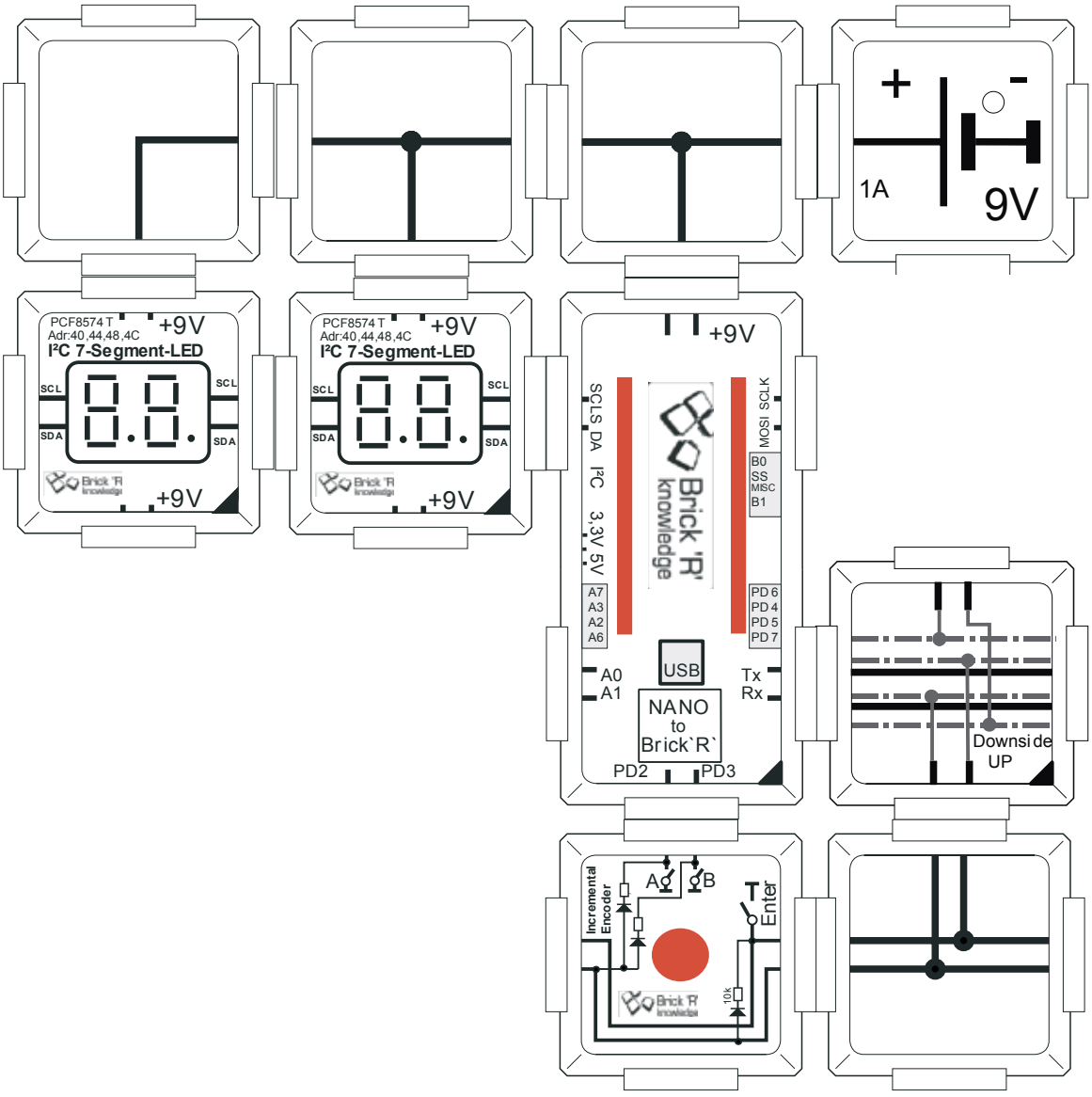
Was passiert? Das verhalten der LED bleibt gleich wie im vorherigen Beispiel. Auf dem Terminal des PCs sieht man aber zusätzlich einen Zählerstand, der je nach Drehrichtung erhöht oder verringert wird.

```
// DE_25 Incrementalgeber Wertausgabe Konsole

#define SWITCHA 2 // PD2 A
#define SWITCHB 3 // PD3 B
#define SWITCHT 4 // PD4 Taste
#define PULLUP 5 // optional
#define PORTLED6 6 // Kontrol
#define PORTLED7 7 // Ausgaben
#define PORTMOSILED 11 // Kontrolle
#define PORTSCLKLED 13 //f. Taste

void setup() {
  Serial.begin(9600); // Konsolte verwenden !
  pinMode(SWITCHA,INPUT_PULLUP); // Wie vor-
her
  pinMode(SWITCHB,INPUT_PULLUP); // alles mit
  pinMode(SWITCHT,INPUT_PULLUP); /// Pullups
  pinMode(PULLUP,OUTPUT); // Port 7 als Aus-
gang schalten
  digitalWrite(PULLUP,HIGH); // zusaetzlich
pullup
  pinMode(PORTLED6,OUTPUT); // Port 6 als
Ausgang schalten
  pinMode(PORTLED7,OUTPUT); // Port 7 als
Ausgang schalten
  pinMode(PORTMOSILED,OUTPUT); // PORTMOSILED
als Ausgang schalten
  pinMode(PORTSCLKLED,OUTPUT); // PORTSCLKLED
als Ausgang schalten
}

void loop() { // Schleife
  static int counter = 0; // Unser Zaehler
  int a= digitalRead(SWITCHA); // A lesen
  int b= digitalRead(SWITCHB); // B lesen
  int state  = 0; // State aktuell
  static int oldstate = 99; // vorheriges
99=neu
  int sw = digitalRead(SWITCHT); // Schalter
  if (a==HIGH) { // erst state definieren
    digitalWrite(PORTLED6,HIGH);
    state = 1;
  } else { // sonst 0
    digitalWrite(PORTLED6,LOW);
    state = 0;
  }
  if (b==HIGH) { // auch fuer Kontakt B
    digitalWrite(PORTLED7,HIGH);
    state += 2; // State +2 ist Trick gibt 2
oder 3
  } else {
    digitalWrite(PORTLED7,LOW);
    state += 0; // nur zur Schoenheit
  }
  switch(state) { // der neue Zustand
    case 0: // AB=00
     if (oldstate == 1)  { // wenn vor-
her 1
        counter--; // dann rueckwaerts
     } else if (oldstate == 2) {
        counter++; // Bei 2  vorwaerts
     } // sonst halten
     break;
    case 1: // AB=10
    if (oldstate == 0) { // wenn 0
       counter++;  // dann vorwaerts
    } else if (oldstate == 3) {
      counter--; // bei 3 Rueckwaerts
    } // sonst halten
    break;
    case 2: // AB=01
      if (oldstate == 3) {  // wenn 3
        counter++; // dann zaehlen vor-
waerts
      } else if (oldstate == 0) {
         counter--;  // wenn 0 dann
rueckwaerts
      } // sonst halten
     break;
    case 3: // AB=11
     if (oldstate == 2) { // wenn vorher
2
        counter--; // dann rueckwaerts
      } else if (oldstate == 1) {
        counter++; // wenn 1 dann vor-
waerts
      } // sonst halten
     break;
  }
  oldstate = state; // wird in oldstate
gespeichert
  if (sw==HIGH) { // Taste hat sonst
    digitalWrite(PORTMOSILED,HIGH); //
keine funktion
    digitalWrite(PORTSCLKLED,LOW); //
nur LEDs steuern
  } else {
    digitalWrite(PORTMOSILED,LOW); //
und umgekehrt
    digitalWrite(PORTSCLKLED,HIGH); //
zur Kontrolle
  }
  Serial.println(counter); // WICHTIG:
Ausgabe auf Konsole
}
```

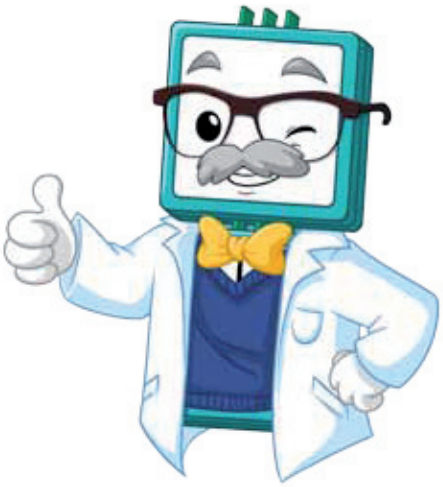
9.3 Incrementalgeber und Siebensegement-Anzeigen zur Ausgabe

Hier eine besonders komfortable Art, die Werte des Zählerstands auszugeben. Es werden einfach die Siebensegment-Anzeigen dazu verwendet. Dann brauchen wir auch die Konsolen-Ausgabe nicht mehr. Der Zählvorgang wird wie vorher über das Statesystem durchgeführt. Aber die Ausgabe erfolgt nun auf den beiden Siebensegment-Anzeigen. Achtung, der Code ist sehr umfangreich, daher haben wir den allgemeinen Teil mit der Bibliothek hier weggelassen, im Anhang ist er zu finden.
Eine Besonderheit ist die Verwendung von „oldcounter“. Die Ausgabe auf der Siebensegment-Anzeige wird nur dann durchgeführt, wenn sich der Zählerstand ändert. Damit wird erreicht, dass keine unnötigen Ausgaben erfolgen die Rechenzeit brauchen, da die Abfrage der AB Kontakte zeitkritisch ist. Bei schnellen Drehbewegungen kann das Programm unter Umständen den Änderungen nicht mehr folgen, da es bei der Abfrage zu langsam ist und Änderungen übersieht. Die Taste hat noch eine Zusatzfunktion, damit läßt sich der Zäherstand auf 0 setzen.

Was passiert? Hier wird der Zählerstand auf der 7-Segment-Anzeige ausgegeben. Dieser wird je nach Drehrichtung erhöht oder verringert.



```
// DE_26 7segment Incrementalgeber mit
Wertausgabe auf 7Segment
#include <Wire.h>

... wie gehabt bei der 7-Segmentanzei-
ge...

// Code neu:
#define SWITCHA 2 // A
#define SWITCHB 3 // B
#define SWITCHT 4 // Taste
#define PULLUP 5  // Extra

void setup() {
  Wire.begin(); // i2C Init
  pinMode(SWITCHA,INPUT_PULLUP);
  pinMode(SWITCHB,INPUT_PULLUP);
  pinMode(SWITCHT,INPUT_PULLUP);
}

void loop() { // Schleife
  char buffer[10]; // Ascii fuer 7Seg
  static int counter = 0; // Zaehler-
stand
  static int oldcounter = -999; // al-
ter Zaehler
  int a= digitalRead(SWITCHA); // A le-
sen
  int b= digitalRead(SWITCHB); // B le-
sen
  int state  = 0; // State merker
  static int oldstate = 99; // alter
State
  int sw = digitalRead(SWITCHT); //
Taste
  if (sw == LOW) counter = 0; // RESET
damit !
  if (a==HIGH) { // state definieren
    state = 1;
  } else {
    state = 0;
  }
  if (b==HIGH) { // auch fuer B
    state += 2;
  } else {
    state += 0;
  }
  switch(state) { // Statediagramm
    case 0: // wie vorher abarbeiten 00
     if (oldstate == 1)  {
        counter--;
     } else if (oldstate == 2) {
        counter++;
     }
     break;
    case 1: // AB = 10
    if (oldstate == 0) {
       counter++;
    } else if (oldstate == 3) {
      counter--;
    }
     break;
    case 2: // AB = 01
      if (oldstate == 3) {
        counter++;
      } else if (oldstate == 0) {
         counter--;
      }
     break;
    case 3: // AB = 11
     if (oldstate == 2) {
        counter--;
      } else if (oldstate == 1) {
        counter++;
      }
     break;
  }
  oldstate = state; // und wieder merken
  sprintf(buffer,"%04d",counter); // in ASCII
  // Counter ausgeben nur bei Aenderung
  if (oldcounter != counter) { // nur wenn
neuer Wert da
      display_seg1x(i2cseg7x2bmsb1,buffer[0]);
// msb zeichen/ LINKE Anzeige
      display_seg1x(i2cseg7x2blsb1,buffer[1]);
// ..
      display_seg1x(i2cseg7x2amsb1,buffer[2]);
      display_seg1x(i2cseg7x2alsb1,buffer[3]);
// lsb zeichen
      oldcounter = counter; // als Merker wich-
tig
    }
}
```



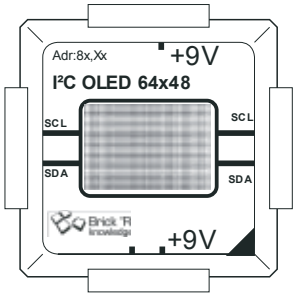
10. OLED

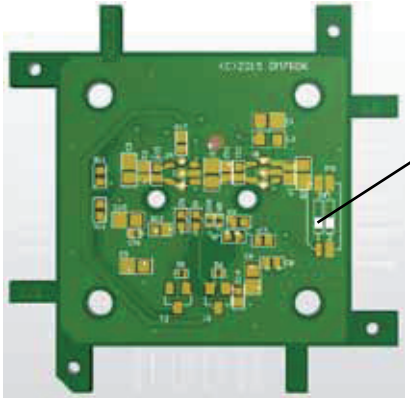
10.1 Aufbau einer grafischen Anzeige

Die 7-Segment-Anzeigen werden normalerweise nur für Ziffern verwendet und ganz eingeschränkt auch für Buchstaben. Mit 14- und 16-Segment-Anzeigen lassen sich auch Buchstaben brauchbar darstellen. Danach gab es dann die ersten Rasteranzeigen, mit einer Matrix von 5x7 Punkten konnte man die Schriften schon viel besser darstellen, aber auch erste graphische Symbole waren hiermit möglich. Die Anzeigen basierten zunächst auf LEDs, dann kamen LCDs (Liquid Crystal Displays) auf den Markt und neuerdings auch OLEDs (Organic Light-Emitting Diode), diese sind wieder den LEDs ähnlicher, da sie selbst leuchten können. Unser Set enthält ein monochromes OLED Display mit 64x48 Pixeln. Damit lassen sich nicht nur einzelne Zeichen sondern auch ganzer Text und einfache Grafiken darstellen. Damit man Zeichen in dieser Weise auf dem Display darstellen kann, benötigt man einen Zeichengenerator, bzw. Tabelle, so ähnlich wie die 7-Segmenttabelle. Der numerischen Code wird für die Segmente in den Zustand an und aus gesetzt. Der Zeichengenerator oder die Zeichensatztabelle benötigt ungleich mehr Speicher. Der Wert hängt von der Zahl der Pixel ab. Bei den kleinsten mit ca. 5x7 Pixel, werden etwa 5 Bytes pro Zeichen benötigt. Wenn man damit den ganzen ASCII-Satz (128 Zeichen inkl. Lücken) darstellen möchte, braucht man 128 x 5 Bytes = 640 Bytes. Für den Nano-Brick haben wir eine solche Zeichentabelle vorbereitet, die noch etwas anders aufgebaut ist. Damit wird es möglich auch Unicode Sonderzeichen, die nicht im ASCII-Satz enthalten sind, darzustellen. Um eine bessere Lesbarkeit zu erreichen, ist der Font etwas größer und der Schriftabstand variabel (Proportionalschrift). Zu Erzeugung wurde der BitFontCreator PO v3.0 verwendet, mit dem sich unterschiedliche Schriften in Raster für Mikrocontroller und damit auch den NANO generieren lassen.

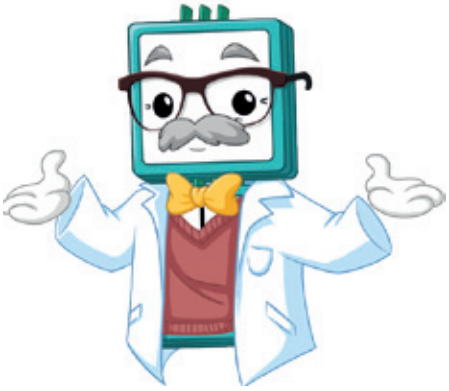
Unser Brick belegt eine I2C Adresse, die entweder 0x78 oder 0x7A ist, je nach Schalterstellung von SW1 auf der Rückseite der Platine. Der innere Schalter bestimmt die Adresse, normalerweise trägt er die Beschriftung „1“. Die OLED wird über I2C programmiert und hat einen eigenen Controller an Board.

In unserer kleinen Bibliothek gibt es Aufrufe, mit denen man Zeichen, Texte, Linien usw. leicht anzeigen kann, was die folgenden Beispiele noch verdeutlichen.





Adresse 78 oder 7A
Schalter auf ON dann
78



```
// Auszug aus unserem CODE:

// Einzelne Zeichen codiert

const unsigned char fontArial14h_data_tablep[] PROGMEM =
{

/* character 0x0020 (, ,): [width=3, offset= 0x0000 (0) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x0021 (,!‘): [width=2, offset= 0x000E (14) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x00, 0x80, 0x00, 0x00, 0x00,
...

};

// Tabelle mit offsets in die Zeichentabelle
const unsigned int fontArial14h_offset_tablep[] PROGMEM =
{
/*  offset      offsetHex  -  char    hexcode    decimal */
/*  =======     =========  -  ====    =======    ======= */
  0,         /*         0  -          0020       32      */
  14,        /*         E  -  !       0021       33      */
  28,        /*        1C  -  „       0022       34      */
...
  1876,      /*       754  -  extra address: the end of the last character‘s
imagebits data */
};


// Breitentabelle
const unsigned short fontArial14h_width_tablep[] PROGMEM =
{
/*  width    char    hexcode    decimal */
/*  =====    ====    =======    ======= */
  3,       /*          0020        32   */
  2,       /*  !       0021        33   */
  4,       /*  „       0022        34   */
...
  8,       /*  ?       2642      9794   */
  8,       /*  ?       266B      9835   */
};
```

10.2 OLED Brick Bibliothek

Wir haben zahlreiche Befehle vorbereitet, mit denen man die OLED einfacher ansprechen kann, als mit reinen I2C Befehlen. Die basieren auf dem internen Controller und man kann damit jedes Pixel ansteuern, Helligkeiten programmieren und vieles mehr. Ausgabe von Zeichen, Schriften, Linien usw. ist nicht im OLED vorgesehen, daher muss es dann extern programmiert werden, dabei hilft unsere eine kleine Bibliothek.

Der wichtigste Befehl ist:

```
i2c_oled_initall(i2coledssd);
```

Damit wird das OLED überhaupt erst in Betrieb genommen. Auch das Timing der OLED wird hier programmiert, so dass es für die interne Brick-Schaltung angepasst ist. Als Parameter wird die I2C Adresse angegeben:

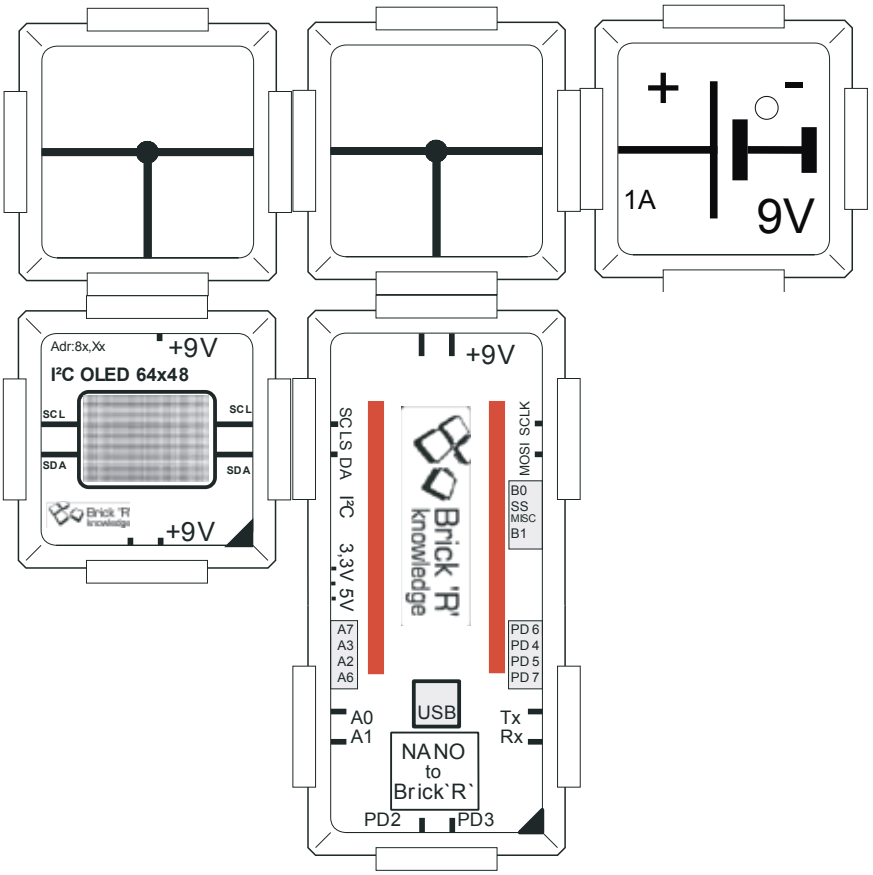
```
#define i2coledssd (0x7A>>1)  // Default
```

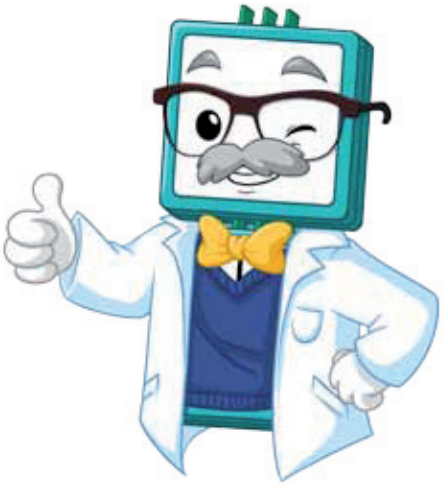
Oder

```
#define i2coledssd (0x78>>1)  // optional für zweites OLED
```

Es wird ein interner buffer (lcdbuffer[]) verwendet um alle Pixel zu speichern und diese dann auf einmal zu übertragen. Das vermeidet Flimmereffekte, die Double-Buffering genannt werden.

```
// Helle und dunkle Pixel, in dieser Ausführung gibt es keine Zwischenstufen.
#define COLOR_BLACK 0        // damit vereinfachte Parameterübergabe
#define COLOR_WHITE 1        // OLED Pixel = an (wenn nicht invers)
```



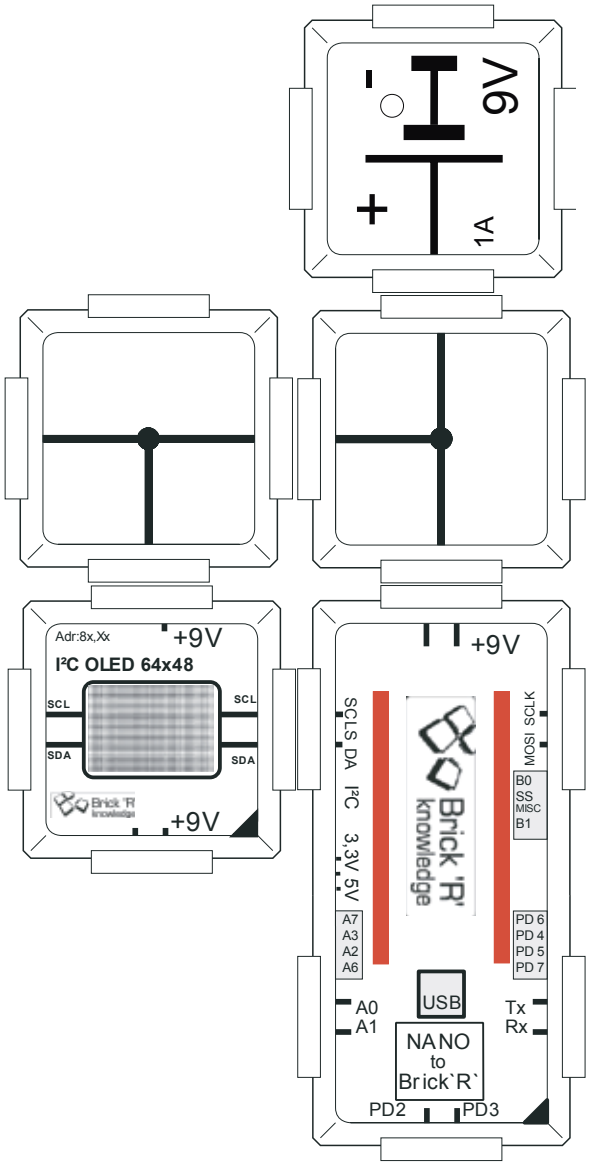


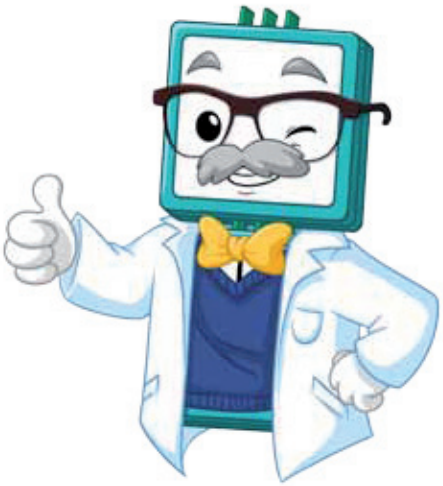
```
void i2c_oled_write_command(unsigned char i2cbaseadr, unsigned char cmdvalue)
      Intern verwendet um an den I2C Daten zu schreiben

void i2c_oled_entire_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Damit kann man die OLED an- oder ausschalten. 1=an 0=aus

void i2c_oled_display_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Auch zum an- und auschalten des Displays ander Variante

void i2c_oled_setbrightness(unsigned char i2cbaseadr, unsigned char wert)
      Helligkeit des OLEDs setzen 0..255 als Wert 0=dunkel

void i2c_oled_inverse_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Invertieren des Displays, 1=invers

unsigned char i2c_oled_write_top(unsigned char i2cbaseadr, int zeile,
                    int bytes,  unsigned char barray[],signed int sh1106padding)
      Intern verwendet um eine Zeilegruppe (8) zu schreiben.

void disp_lcd_frombuffer()
      Überträgt den Bufferinhalt (lcdbuffer[]) auf das Display.

void disp_buffer_clear(unsigned short data)
      Löscht den Pixelbuffer (lcdbuffer[]) mit data (=0 oder =1)

void disp_setpixel(int x, int y, unsigned short col1)
      Setzt ein Pixel bei x,y wobei x links und y oben ist.
      col1 definiert die Farbe (0 oder 1)

unsigned short disp_setchar(int x, int y,  unsigned char chidx1, unsigned short color)
      Setzt ein Zeichen bei x,y mit ASCII Coe chidx1 und color (=0 oder =1)

int disp_print_xy_lcd(int x, int y, unsigned char *text, unsigned short color, int
chset)
      Ausgabe eines Textes (ASCII bei char *text) an der Position x,y
      mit dem Zeichensatz chset (bei uns ignoriert), linke obere Ecke.

void  disp_line_lcd(int x0, int y0, int x1, int y1, unsigned short col)
      Linie von x0,y0 nach x1,y1 in Farbe col (0,1)

void disp_rect_lcd(int x1, int y1, int x2, int y2, unsigned short col)
      Rechteck bei x1,y1 aufgespannt nach x2,y2 in Farbe col(0,1)

void disp_filledrect_lcd(int x1, int y1, int x2, int y2, unsigned short col)
      Gefuelltes Rechteck bei x1,y1 aufgespannt nach x2,y2 in Farbe col(0,1)
```

10.3 OLED Display über I2C

Hier wird eine Sinus-Funktion auf dem Display dargestellt. Dabei wird nach jedem Bildaufbau die Phase des Sinus leicht geändert, so dass der Sinus durchzulaufen scheint.
Wir verwenden dazu ein paar praktische Elemente aus unserer Bibliothek. Der Aufruf disp_buffer_clear() löscht den Displaybuffer. Man kann eine Löschfarbe (Schwarz oder Weiss) als Parameter angeben. Dies geschieht aber nur in einem Buffer auf dem NANO nicht auf der OLED selbst. Die OLED zeigt zu dem Zeitpunkt noch das aktuelle Bild an. Man nennt das auch Double Buffer Technique. Danach wird eine horizontale Linie gezeichnet. Dazu gibt es den praktischen Befehl disp_line_lcd(). Als Parameter wird hier die Startkoordinate x,y gegeben und die Zielkoordinate (der Zielpunkt wird auch als Punkt gezeichnet!). X geht von 0 bis 63 und y von 0 bis 47. Der Ursprung liegt dabei links oben!
Die Funktion sin() muss noch in den Wertebereich angepasst werden, dazu wird y mit dem Range 0..47 berechnet, sin() hat ja bekanntlich den Wertebereich -1.0 .. 1.0.
Mit disp_setpixel() werden nun genau die einzelnen Punkte der Funktion gesetzt, die durchlaufen werden. Als Parameter gibt es x,y und die Farben weiß oder schwarz. Die Koordinate y wird vor der Ausgabe mit 47-y als Parameter an disp_setpixel() übergeben, da der Urspung bei der OLED für 0,0 oben links liegt.

ACHTUNG: Hier liegt die Anzeige auf der Adresse 7A, ggf. den Dilschalter auf der Rückseite auf OFF stellen !



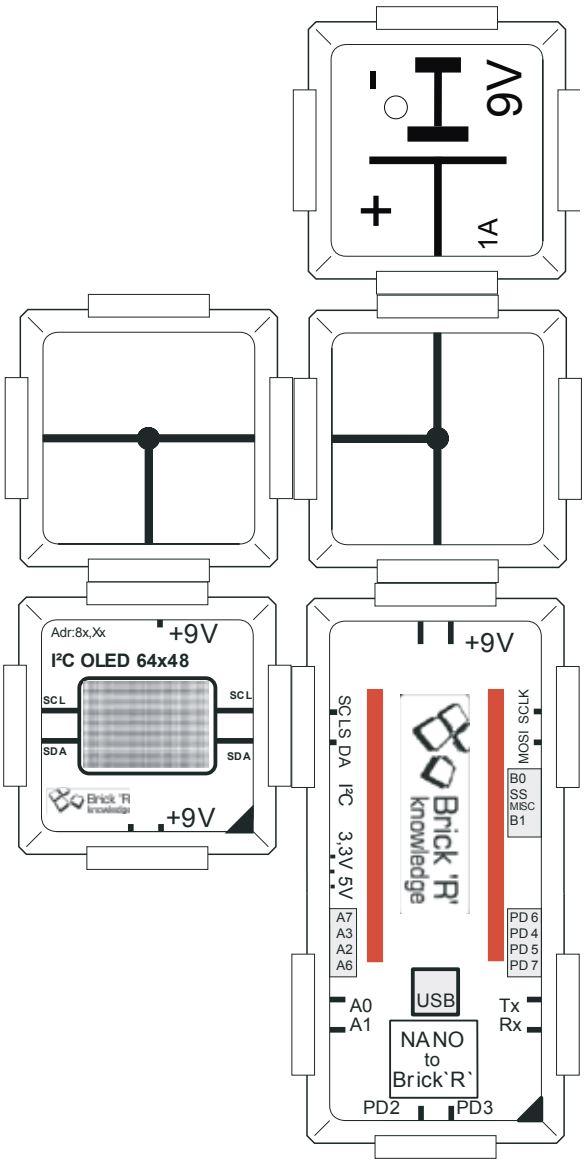


```
// DE_27 OLED Beispiele - Pixelroutinen

#include <Wire.h> // I2C Bibliothek
#include <avr/pgmspace.h> // Zugriff ins ROM

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A

// -------------------------OLED ---------------------------------------------


..... Bibliothek einkopieren.


// -------------------------END OLED -----------------------------------------

void setup() {
  Wire.begin(); // I2C Init
  i2c_oled_initall(i2coledssd); // OLED Init
}


void loop() {  // in der Schleife
  // 64x48 Pixel OLED
  static int phase=0; // Scrolleffekt hier statisch speichern
  disp_buffer_clear(COLOR_BLACK); // Virtuellen Buffer loeschen BLACK = dunkel
  disp_line_lcd (0,24,63,24, COLOR_WHITE); // x0,y0,x1,y1 Linie in Mitte
  for (int i=0; i<63;i++) { // nun fuer alle 64 Spalten (x)  des Displays
    int y=0; // Sinuskurve berechnen und in Radiant umrechnen von Phase
    y = (int)(23.0*sin(((double)i*3.141592*2.0)/64.0+phase/360.0*2.0*3.141592)+24.0);
    disp_setpixel(i, 47-y, COLOR_WHITE); // koordinate 0,0 ist links oben daher 47-y
  } // alle Spalten
  disp_lcd_frombuffer(); // dann erst updaten, double buffer flimmerfrei
  phase++; // beim nächsten Mal mit neuer Phase für den Sinus
  if (phase>=360)phase=0; // Immer 0 bis 359 Grad (als DEG)
}
```

Was passiert? Nach dem Start erscheint eine Sinuskurve auf dem Display. Dabei scrollt die Kurve langsam horizontal durch.

10.4 OLED Display und Zeichensatz

Will man Zeichen darstellen, so kann man auf unsere Zeichenroutinen zurückgreifen.
Mit disp_print_xy_lcd() lässt sich ein Text auf dem Display darstellen. Dazu wird als Parameter die Koordinate der linken oberen Ecke des ersten Zeichens angegeben (x,y). 0,0 links links oben. Die Zeichenhöhe kann man mit ca. 10 Pixeln annehmen, wenn man mehrere Zeilen schreibt, so muss man einen entsprechenden Offset eingeben. In dem Programmbeispiel werden mehrere Textzeilen ausgegeben. Der berühmte Satz „the quick brown fox jumps over the lazy dog“ enthält alle vorkommenden Zeichen des Alphabets, daher wird er gerne für solche Beispiele verwendet.
Das Programm verwendet zusätzlich eine Variable yoffset, die bei jedem Durchlauf der Schleife „loop“ um eins verringert wird. Dadurch entsteht ein Scrolleffekt, der Text läuft nach oben durch. Wenn der Offset von -60 erreicht ist, ist der Bildschirm leer und der Offset wird zurück auf 50 gesetzt, so daß die erste Zeile links unten erscheint (bei einem Offset von 0 würde der Text oben starten).



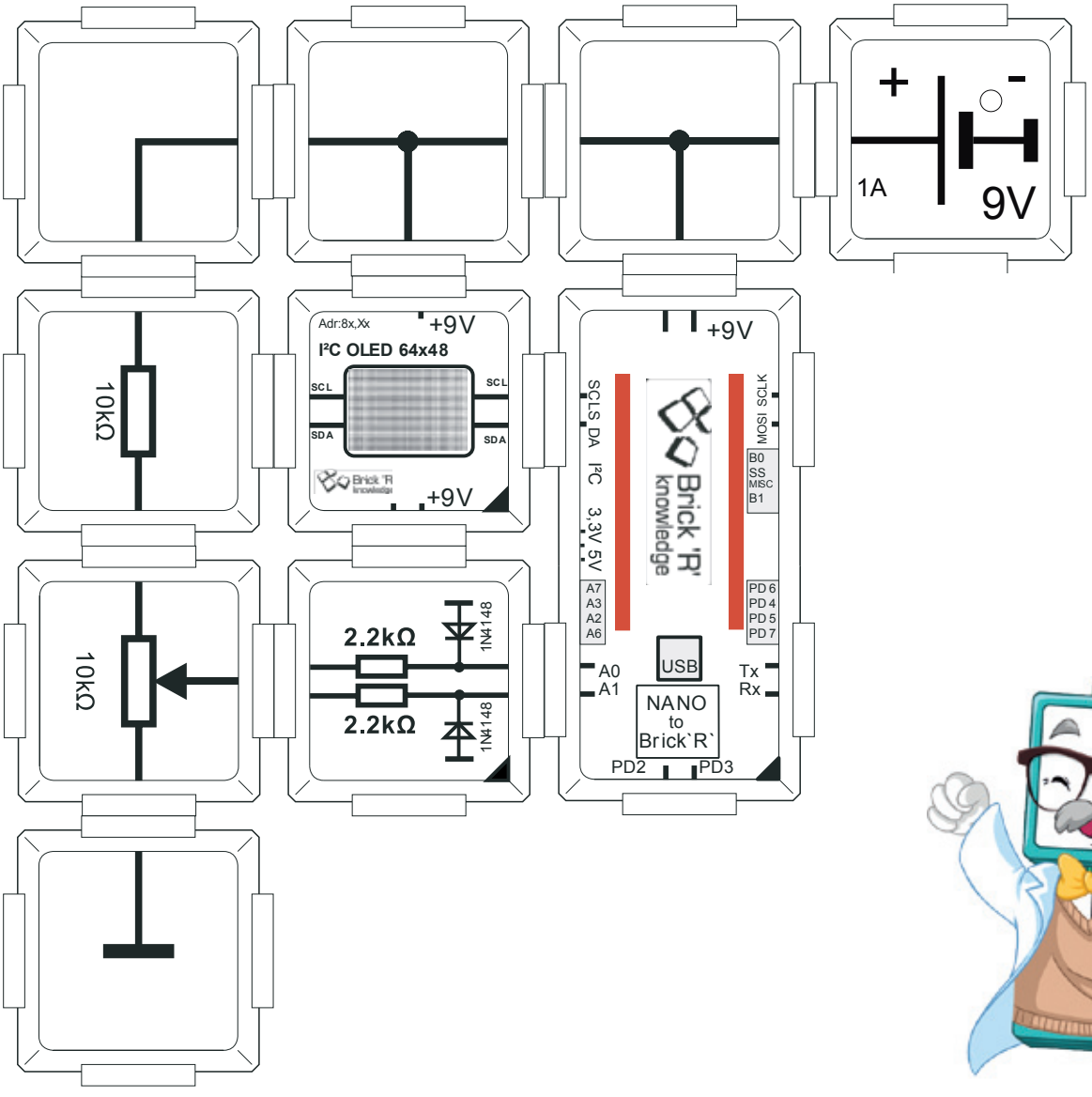
```
// DE_28 OLED Beispiele - Zeichensatz

#include <Wire.h>
#include <avr/pgmspace.h>

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A

// -------------------------OLED ---------------------------------------------

...

// -------------------------END OLED -----------------------------------------

void setup() {
  Wire.begin(); // I2C Initialisieren
  i2c_oled_initall(i2coledssd);  // OLED Initialisieren danach !
}

void loop() {  // Schleife
  // 64x48 Pixel OLED
  char buffer[40]; // Buffer für die Zeichenaushabe
  static int yoffset = 50; // Scrollt von unten rein.
  disp_buffer_clear(COLOR_BLACK); // Virtuellen Buffer loeschen
  // dann einzelne Zeilen ausgeben, dabei mit einem Offset yoffset.
  disp_print_xy_lcd(0, 0+yoffset, (unsigned char*)“the quick“, COLOR_WHITE, 0);
  disp_print_xy_lcd(0, 10+yoffset, (unsigned char*)“brown fox“, COLOR_WHITE, 0);
  disp_print_xy_lcd(0, 20+yoffset, (unsigned char*)“jumps over“, COLOR_WHITE, 0);
  disp_print_xy_lcd(0, 30+yoffset, (unsigned char*)“the lazy“, COLOR_WHITE, 0);
  disp_print_xy_lcd(0, 40+yoffset, (unsigned char*)“dog“, COLOR_WHITE, 0);
  disp_print_xy_lcd(0, 50+yoffset, (unsigned char*)“0123456890“, COLOR_WHITE, 0);
  disp_lcd_frombuffer();  // ann erste buffer ans OLED uebertragen.
  delay(10); // Eigentlich nicht noetig aber bei schnellen CPUs sicherer
  yoffset = yoffset - 1; // Zeilenweise scrollen 1 Pixel
  if (yoffset < -60) yoffset = 50; // wieder von vorne anfangen
} // Ende der Schleife
```

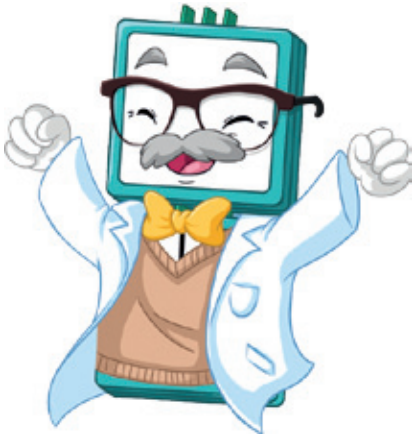
Was passiert? Der Text „the quick borwn fox jumps over the lazy dog 0123456789“ wird auf dem DIsplay ausgegeben und scrollt in mehreren Zeilen zerlegt, vertikal über den Bildschirm. Das Ganze wiederholt sich dann.

10.6 Das OLED Display mit A/D-Umsetzer als einfaches Mini-Oszilloskope

Wenn man die Messwerte in einem Buffer speichert, kann man ein kleines Mini-Oszilloskope aufbauen. Die maximale Frequenz ist in dem Fall ziemlich klein, wenn man wie hier nach jeder Messung das Bild ausgibt. Oben wird die Spannung in V angezeigt und unten ist das Bild des gemessenen Spannungsverlaufs zu sehen.

Links sieht man ein kommerzielles Oszillsokope von Rigol. Da gibt es natürlich noch viele andere Einstellmöglichkeiten, wie Amplitude, Frequenz und Trigger. Das Prinzip ist aber immer das Gleiche, es gilt den Zeitverlauf eines Signals graphisch darzustellen. Dazu hat man früher Elektronenstrahlen verwendet, heute geht das elegant mit der Digitaltechnik. Solche Scope oder im Jargon auch Oscis genannt können ziemlich aufwendig sein. Entscheidend ist zum Beispiel der Frequenzbereich. Rekordwerte für ein Scope mit Echtzeitdarstellung (also speichern des ganzen Signals) liegen aktuell bei 100 GHz (LeCroy Teledyne LeCroy LabMaster 10-100zi 100GHz, 240GS/s Oszilloskope ca. 1 Mio Dollar). Bei www.TheSignalPath.com kann man sich die genauere Funktionsweise solcher High-End-Geräte ansehen. Unser Miniscope liegt dagegen an der unteren Skala aber immerhin ist es deutlich preiswerter.



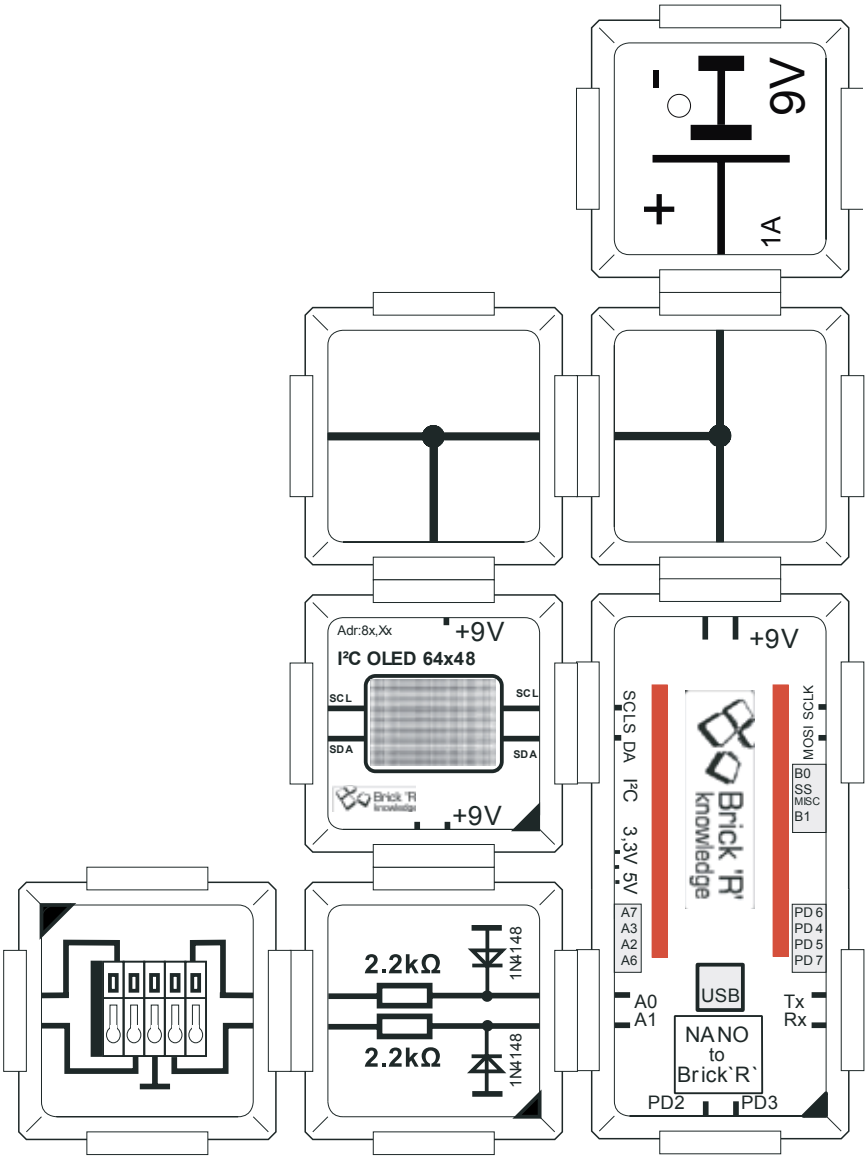


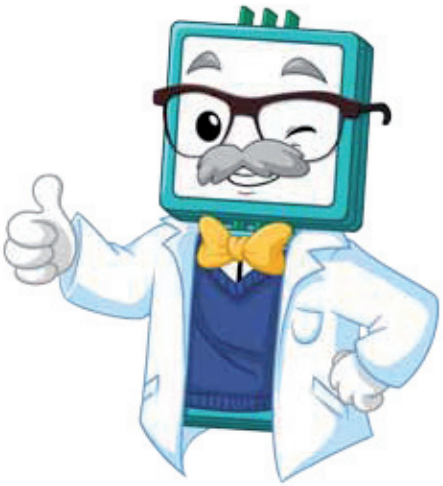
```
// DE_30 OLED Beispiele - AD Mini Oscilloscope

#include <Wire.h>
#include <avr/pgmspace.h>

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A

// -------------------------OLED ---------------------------------------------

...


// -------------------------END OLED -----------------------------------------
char advalbuf[64]; // Zyklischer Speicher fuer die Messwerte

void setup() {
  Wire.begin(); // I2C Initialisierung
  i2c_oled_initall(i2coledssd);  // OLED Initialisieriung
  for (int i=0; i<64; i++) advalbuf[i]=47; // Mittelwert als Default
}

void loop() { // Messen in der Schleife
  // 64x48 Pixel OLED
  static int cxx = 0; // zyklischer Zaehler fuer Buffer
  int poti11 = analogRead(A0); // Einlesen A/D-Umsetzer
  char buffer[40]; // Ausgabebuffer feur Voltangabe
  disp_buffer_clear(COLOR_BLACK); // erst mal loeschen
  double  p1 = (poti11*5000.0)/1023.0;  // Wert am AD Wandler in mV
  int y1 =0; // Temporaere Variable merkt die y Position
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000); // Ausgabe
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel Umrechnen in pixel
  advalbuf[cxx++] = y1; // merken 0..4xx V +-128  Zyklischer Buffer
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // Ausgabe des zyklschen Buffers.
  int yold  = advalbuf[(cxx+1)%64]; // Letzer Wert
  for (i=0; i<63; i++) { // alle Pixel durchlaufen (Spalten)
   y1 =advalbuf[(cxx+1+i)%64]; // Neue Werte ausgeben
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);  // Und verbinden
   yold = y1; // der alte wird zum neuen Wert, fuer Linienbildung
  } // alle Pixel durch
  if (cxx >63) cxx =0;  // Zaehler von 0l..63 fuer zyklischen Buffer
  disp_lcd_frombuffer();  // Dann alles Fertig zur Ausgabe
  delay(10); // Nicht unbedingt wichtig definiert auch die Abtastrate
}
```

Was passiert? Der gemessene Spannungswert wird auf dem Bildschirm in mV ausgegeben. Wenn man an dem Poti dreht kann man einen Bereich von ca. 0 mV bis maximal 5000mV überdecken. Dabei wird dann die Hälfte der Betriebsspanung als Maximalwert angezeigt, also normalweise ca. 4500 mV. Gleichzeitig wird in diesem Versuch auch der zeitliche Verlauf der Spannung angezeigt. Wenn man das Poti hin- und herbewegt, kann man den Verlauf gut sichtbar machen.

10.7 Das OLED Display und der A/D Umsetzer als Dual Voltmeter

Diesmal werden beide A/D-Umsetzer Kanäle A0 und A1 verwendet und sind auch getrennt ansprechbar. Die Werte werden in V auf dem OLED angezeigt.
Mit dem Universalbrick kann man zwei analoge Quellen anschliessen und messen. Der Spannungsbereich liegt zwischen 0V bis 5V. Achtung: Trotz der Schutzschaltung, bestehend aus dem 2.2k Ohm Widerstand und der Diode, deshalb bitte keine Spannung außerhalb des zulässigen Bereichs von 0-5V anschließen.
Diesmal wird das Ergebnis direkt in Volt, mit drei Nachkommstellen angezeigt. Da die Gleitkommaarithmetik beim sprintf Befehl nicht unbedingt verwendet werden kann und zudem auch langsam ist, kann man mit einem Trick auch Integerzahlen bei der Ausgabe verwenden. Dazu gibt man die Vorkommstallen mit Wert/1000 aus, z.B. wenn der Wert mit tausend skaliert wurde (mV nach Volt) und dann die Nachkommastellen mit Wert % 1000, also mit Hilfe der sogenannten Modulo-Funktion (siehe auch Modulo Arithmetik - nicht mit dem Begriff Modul zu verwechseln). Die eigentliche Ausgabe erfolgt vor dem Komma mit dem String „%d“ und nach dem Komma mit „%03d“. Das bedeutet, drei Stellen werden reserviert und die 0 steht für „mit Nullen auffüllen“. Dazwischen kann einfach ein Punkt oder auch Komma (Deutsche Lokalisierung) ausgegeben werden.





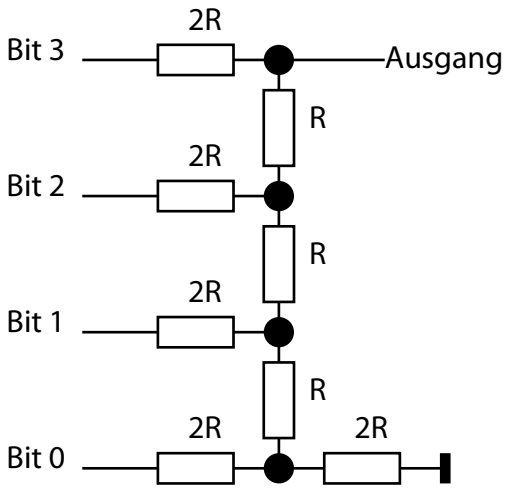
```
// DE_31 OLED Beispiele - AD Dual

#include <Wire.h>
#include <avr/pgmspace.h>

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A

// -------------------------OLED ---------------------------------------------


...


// -------------------------END OLED -----------------------------------------


void setup() {
  Wire.begin(); // I2C Initialisieren
  i2c_oled_initall(i2coledssd);  // OLED Initialisieren
}

void loop() {     // Schleife
  // 64x48 Pixel OLED
  int poti11 = analogRead(A0); // Einlesen A/D-Umsetzer 0
  int poti12 = analogRead(A1); // Einlesen A/D-Umsetzer 1
  char buffer[40]; // fuer die Ausgabe in ASCII
  disp_buffer_clear(COLOR_BLACK); // Display loeschen
  double  p1 = (poti11*5000.0)/1023.0; // Spannung A0 ausrechnen
  double  p2 = (poti12*5000.0)/1023.0; // Spannung A1  ausrechnen
  // Kleiner Trick um auch Nachkommastellen ohne Gleitkommabefehle
  // auszugeben ist sie in zwei INT Werte zu zerlegen:
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000); // ausgabe
  disp_print_xy_lcd(2, 8, (unsigned char *)buffer, COLOR_WHITE, 0);
  sprintf(buffer, „A1=%d.%03dV“,(int)p2/1000,(int)p2%1000); // Ausgabe
  disp_print_xy_lcd(2, 28, (unsigned char *)buffer, COLOR_WHITE, 0);
  disp_lcd_frombuffer();    // Dann auf das OLED Display uebertragen
  delay(10);  // 10ms damit nicht zu haeufige Updates
} // Ende Schleife
```

Was passiert? Hier werden die beiden Kanäle A0 und A1 gleichzeitig als Spannungswert in Volt angezeigt. An den Klemmbrick können dazu zwei verschiedene Spannungsquellen mit einem Bereich von 0 bis maximal 5V angeschlossen werden. Achtung: keine negativen Spannungen anlegen !

11. Digital-Analog Umsetzer

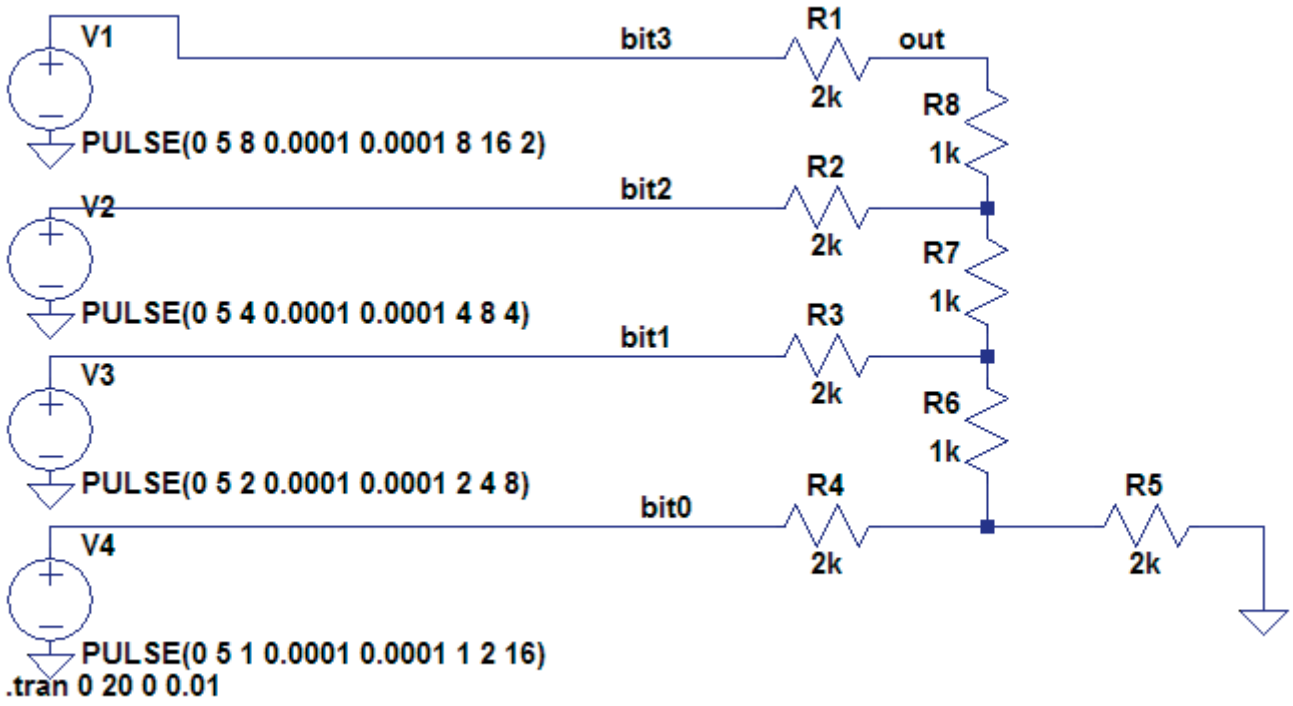
11.1 Digital Analog Umsetzer und Funktion

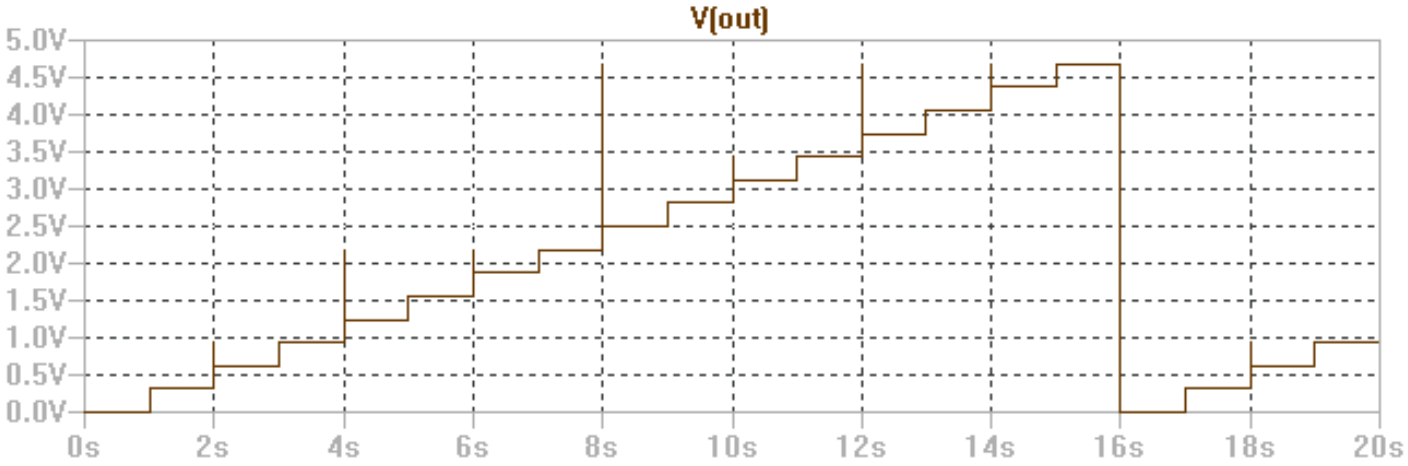
Bisher haben wir analoge Werte, wie Spannung mit einem Analog/Digital Umsetzer in binäre Werte übersetzt. Es geht aber auch umgekehrt, also digitale Werte in analoge Werte umzusetzen. Den Umsetzer nennt man Digital-Analog Umsetzer oder umgangssprachlich D/A-Wandler.
Zur Realisierung gibt es unterschiedliche Wege, den einfachsten erklären wir hier näher:



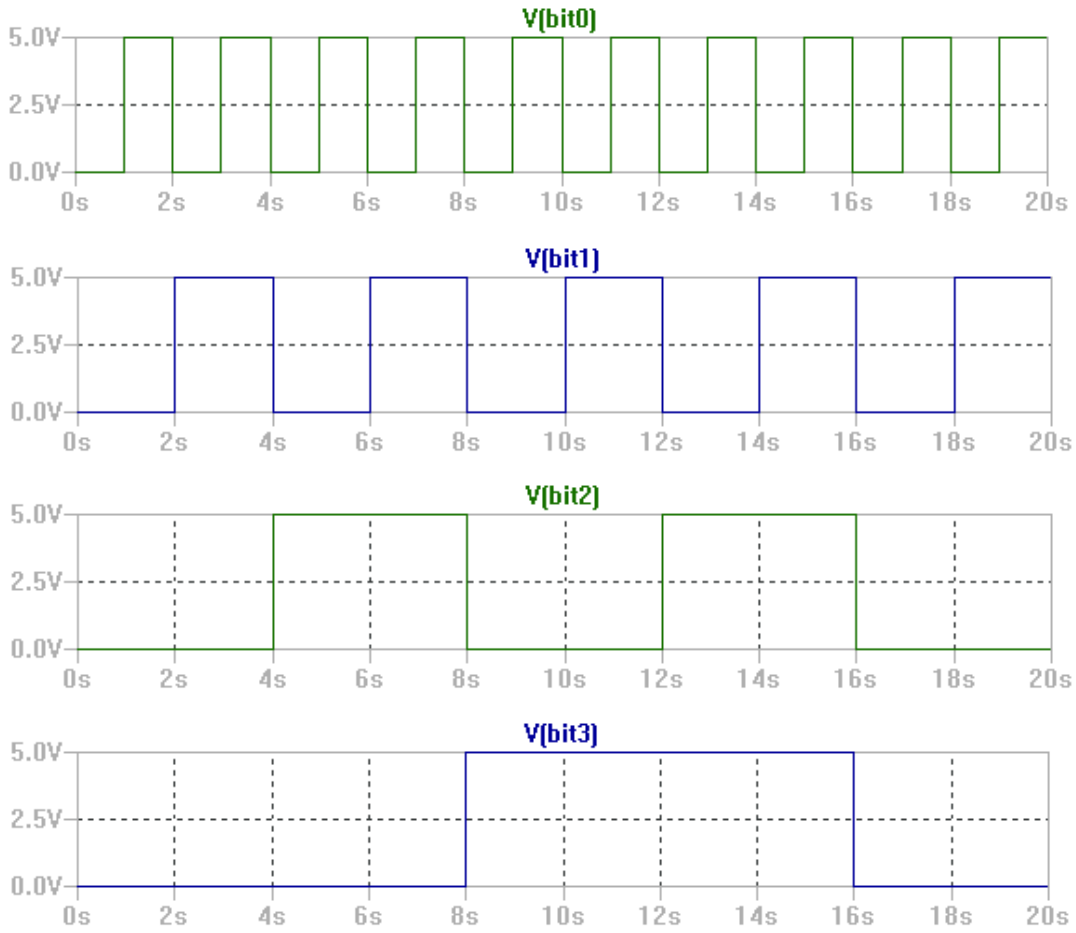
Beispiel: 0=0V 1=5V R=1000Ohm

Hier ein einfacher 4-Bit-DA-Umsetzer, den man sogar nachbauen kann. Man nennt die Anordnung R2R Netzwerk, da nur Widerstände der Wertigkeit R und 2R also dem doppelten Wert vorkommen. Wir verwenden 1kOhm für R und 2kOhm für 2R. Die Widerstände müssen sehr genau sein, damit das Ganze gut funktioniert. Bit 0 ist das sogenannte niederwertigste Bit, ist also dem niedrigsten Wert zugeorndet und ändert nur wenig an der Ausgangsspannung, da es ganz weit hinten am Spannungsteiler liegt. Bit 3 hat den stärksten Einfluss. Ein Sonderfall lässt sich leicht berechnen: sind alle Bits auf 0, dann liegt am Ausgang 0V an. Bei diesem Fall sind alle Bits auf 1 also 5V an den Anschlüssen, am Ausgang gibt es nicht genau 5V, da der letzte Widerstand immer auf 0V liegt. Untenstehend der Plan für eine Simulation mit LTSpice.





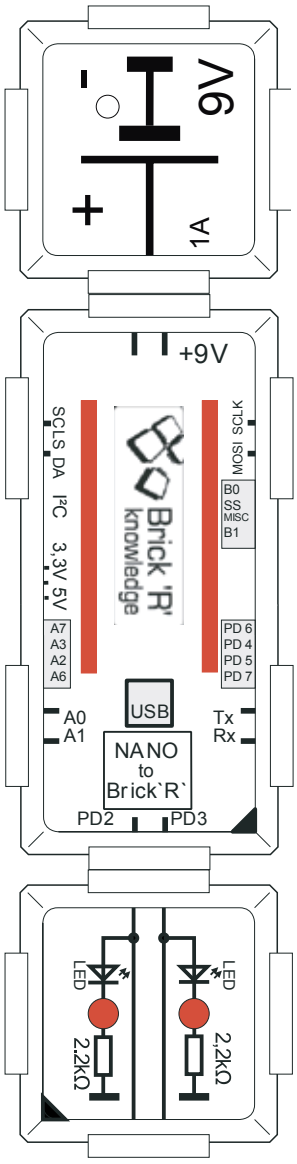
Oben sieht man die Simulationsergebnisse, der simulierte Counter zählt zyklisch von 0 bis 15 und der DA-Umsetzer liefert die entsprechende proportionale Spannung. Das verwendete Programm LTSpice ist kostenlos (Linear Technologies). Die kleinen Spitzen entstehen, wie in der Wirklichkeit, da die Umschaltungen nicht ideal verlaufen, sondern Anstiegs- und Abfallflanken angegeben haben.



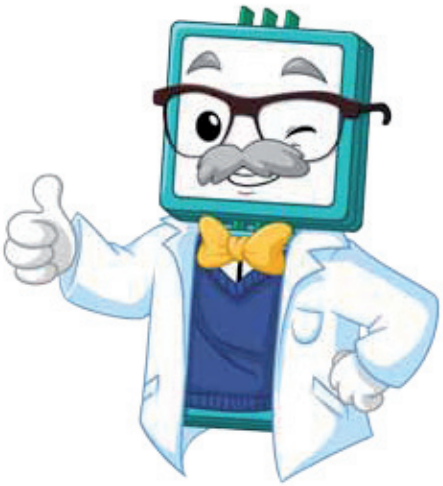
Der Spannungsverlauf der einzelnen Quellen, hier wurden Spannungsquellen verwendet, deren Verlauf per LTSpice programmierbar ist, so dass ein binärer Zähler entsteht.

11.2 Einfacher DA Umsetzer mit PWM

Der NANO hat selbst einen einfachen DA Umsetzer vorgesehen, er arbeitet mit der sogenannten Pulsbreitenmodulation. Das Signal ist eigentlich noch digital, die Pulsbreite entspricht aber dem analogen Wert. Mit einem Kondensator könnte man den Wert noch glätten und in einen „echten“ analogen Wert umwandeln. Wenn man eine einfache LED anschließt, kann man aber auch ohne Kondensator die Helligkeit der LED durch die Pulsbreitenmodulation steuern. Das Licht ist in Wirklichkeit gepulst, aber unser Auge ist zu träge um das Flackern noch gut auflösen zu können.
Hierfür können nicht alle Ports verwendet werden. Beim NANO sind es die Ports 3,5,6,9,10 und 11 die mit dem Befehl analogWrite() angesprochen werden können.
PWM steht als Abkürzung für Pulsweitenmodulation oder auch Pulsbreitenmodulation.

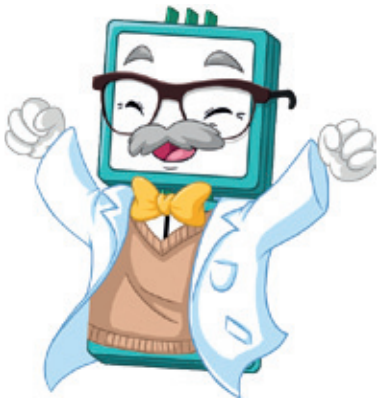



```
Ports 3,5,6,9,10,11 PWM
```



```
// DE_32 DA Beispiele PWM

#include <Wire.h> // I2C Bibliothek
#include <avr/pgmspace.h> // reserve

#define DA3 3 // Port PD3

void setup() {
  pinMode(DA3, OUTPUT); // Auch als Ausgang
}

void loop() { // Schleife
  static int helligkeit=0; // LED Helligkeit
  analogWrite(DA3, helligkeit);
  delay(20); // 20ms
  helligkeit++; // Hier aufsteigend
  if (helligkeit>255) helligkeit = 0; // 0..255
}
```

Was passiert? Die LED am Port PD3 ist am Anfang ganz dunkel, dann sieht man plötzlich die Helligkeit ansteigen, bis ein Maximum erreicht wird und alles von vorne beginnt. Der Vorgang dauert ein paar Sekunden (>256 * 20mSec).

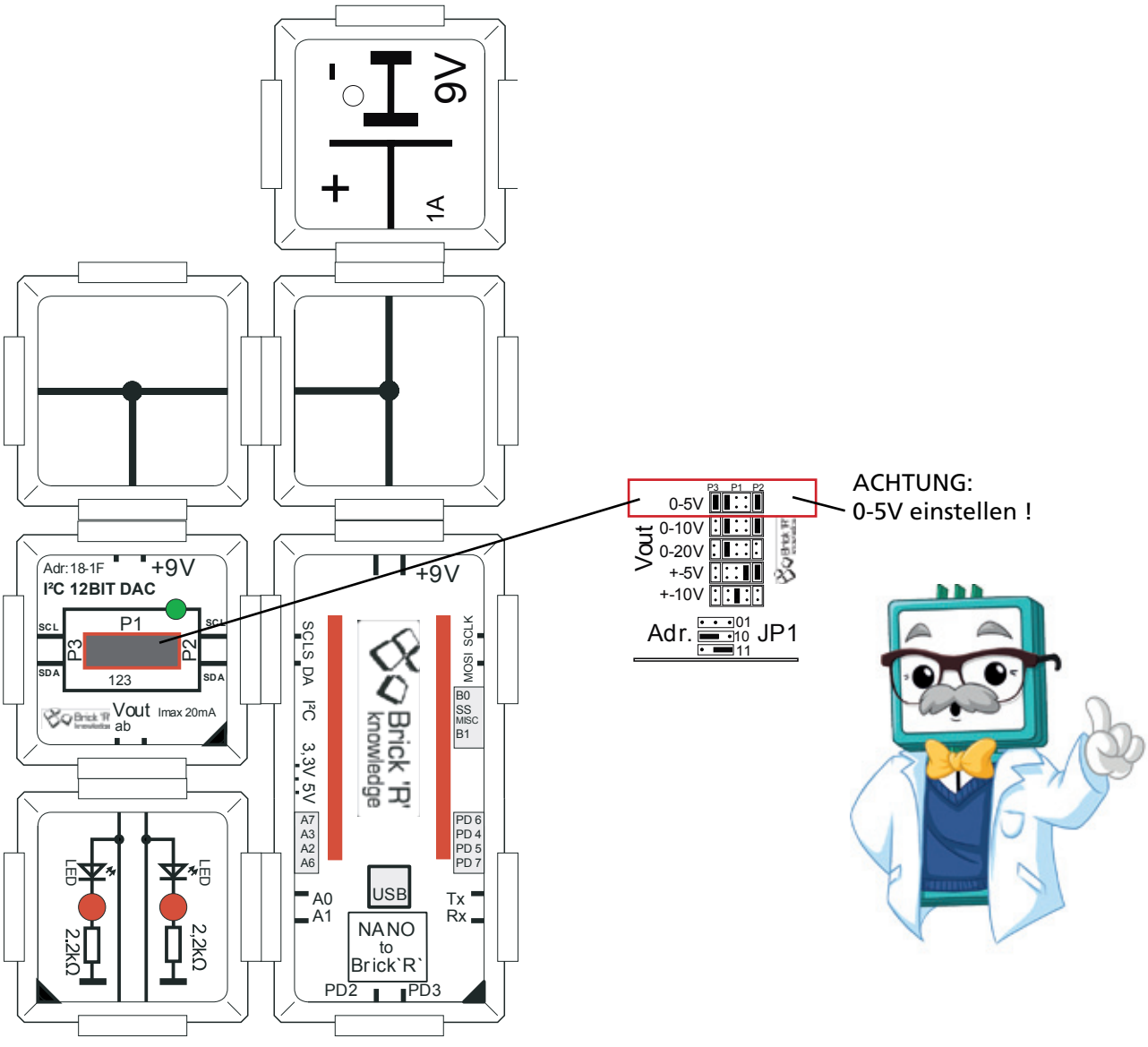


11.3 Der DA-Umsetzer Brick und Ansteuerung per I2C

Unser DA-Umsetzer hat eine höhere Auflösung als der über den PWM realisierte Ausgang, nämlich 4096 Stufen, also 12 Bit. Mit dieser Beispielschaltung kann auch die Helligkeit der LEDs gesteuert werden. Sie beginnen allerdings erst ab einem bestimmten Schwellwert zu leuchten, der je nach LED unterschiedlich sein kann. Deshalb leuchten die beiden LEDs unten auch nicht gleichmäßig. LEDs müsste man eigentlich mit einer Stromregelung ansteuern. Die 2.2k Ohm Widerstände übersetzen näherungsweise die Spannungsänderung in eine Stromänderung. Es gilt dabei die Formel I = (Uda - Uled) / 2200Ohm (*1). Wenn Uda groß gegenüber ULed ist, funktioniert die Übersetzung in die Stromstärke auch ganz gut. Wenn Uda < Uled ist, fließt kein Strom durch die LED und sie bleibt dunkel, damit wird Uda nicht mehr richtig angezeigt.

Achtung: Der DA-Umsetzer-Brick hat integrierte Spannungskonverter für -10V, man kann durch die Steckbrücken auf der Oberseite einstellen, welchen Bereich er hat. In dem Beispiel geht neben 0-5V auch noch 0-10V. Der Ausgangsstrom ist auf ca. 20 mA begrenzt, doch bitte trotzdem immer GENAU kontrollieren, wie die Steckbrücken gesetzt sind. Auf der Unterseite ist die Belegung auch nochmal abgebildet. Wenn man den Baustein öffnet, kann man eine der drei Adressen für den I2C Bus einstellen. Die vierte Adresse (18) lässt sich bei dem Baustein nicht setzen.

*1 Uda > ULed und Uled = die Mindest-Spannung damit die LED leuchtet.

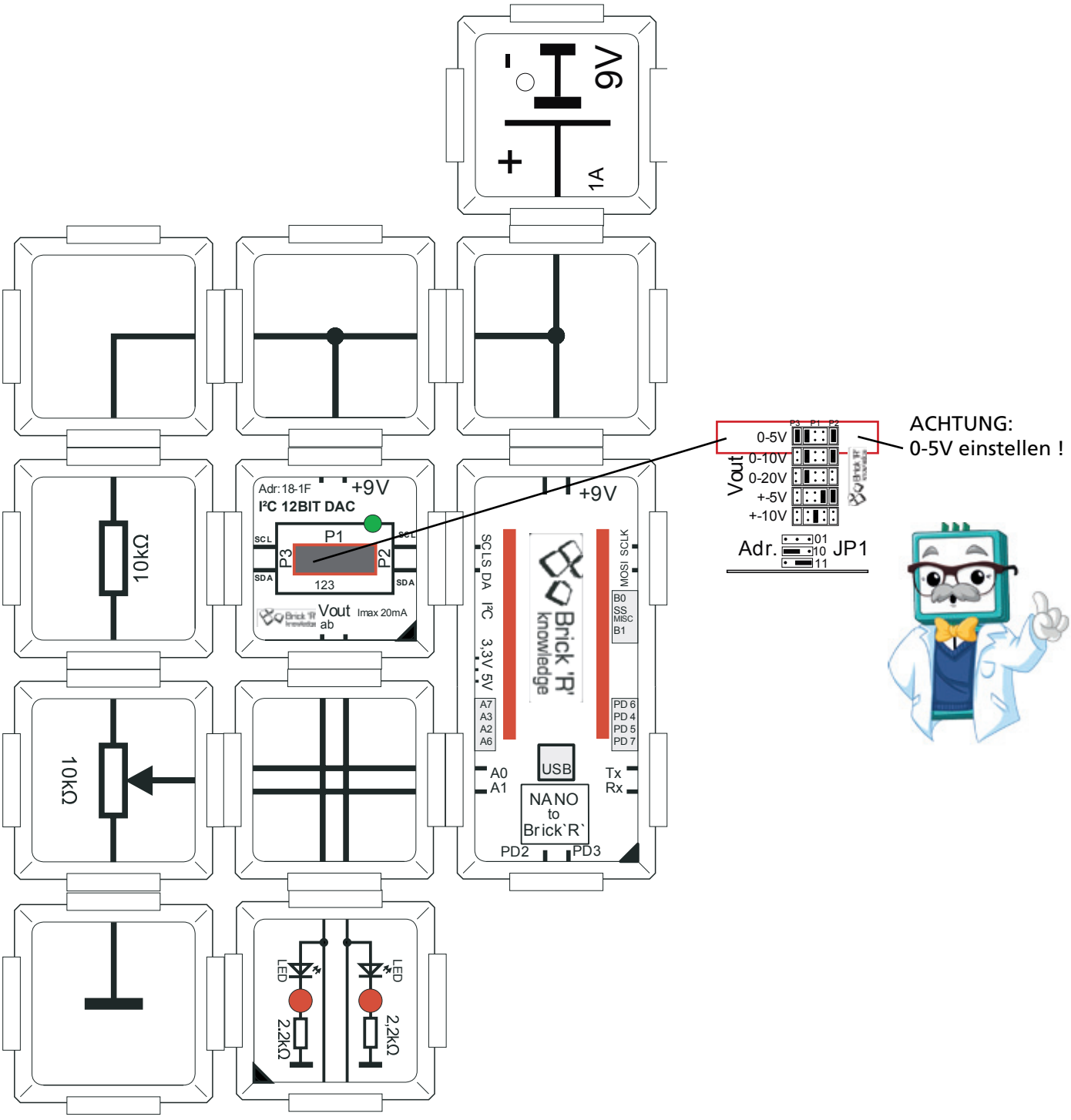


```
// DE_33 DA Beispiele Da Brick I2Cs

#include <Wire.h>
#include <avr/pgmspace.h>

// 1E=GND,1c=Open,1a=VCC AD5622  JUmper offen dann 1Ch
// 0001 1aa0
// aber nur aa=01 10 11 sind  moegliche Adressen

#define i2cdasel1 (0x1a>>1)  // ADRESSE Einstellen 1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // ADRESSE Einstellen 1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // ADRESSE Einstellen 1a,1c,1e


void i2c_da_write_command(unsigned char i2cbaseadr, unsigned short cmdvalue)
{
  // BIt15,14=0 13,12=pd (std=0) dann 11..0 = DA Value
  cmdvalue = cmdvalue & 0xFFF; // 12 Bits valide // 0..4095 Wertebereich
  Wire.beginTransmission(i2cbaseadr); // I2C Start senden
  Wire.write((cmdvalue>>8)&0xff); //  dann MSB zuerst
  Wire.write(cmdvalue&0xff);  // danach das LSB
  Wire.endTransmission(); // I2C beenden
}

void setup() {
  Wire.begin(); // i2C Initialisierung
}

void loop() {
  static int daval=0; // Wert statisch fuer tests wird hochgezaehlt
  // Alle moeglichen I2C Adresse ausgeben der Einfachheit halber
  i2c_da_write_command(i2cdasel1,daval); // Hier mal alle DA Umsetzer
  i2c_da_write_command(i2cdasel2,daval); // an sprechen die moeglich sind
  i2c_da_write_command(i2cdasel3,daval); // Nur einer wird aber verwendet
  delay(1); // 1ms nur zur Sicherheit damit nicht zu schnell und alles noch sichtbar
  daval++; // Neuer DA Wert
  if (daval>4095) daval = 0; // Ueberlauf verhindern 0..4095
}
```

Was passiert? Die beiden LEDs sind zu Beginn ganz dunkel, dann sieht man plötzlich die Helligkeit ansteigen, bis ein Maximum erreicht wird und alles von vorne beginnt. Der Vorgang dauert ein paar Sekunden.

11.4 Der DA Umsetzer Brick und Poti

In dem Beispiel wird der A/D-Umsetzer verwendet, um den Spannungswert am Poti einzulesen und dann in der Schleife an dem D/A-Umsetzer auszugeben. Nun hat der A/D-Umsetzer des Nano einen Wertebereich von 0..1023 also 10Bit, der D/A-Umsetzer aber 0..4095 also 12 Bit. Im Programm muss der Wertebereich umgerechnet werden. Dazu kann der Wert des A/D-Umsetzer mit 4 multipliziert werden, im Programm geschieht dies durch eine Schiebeoperation (<<2).
Wenn man das Poti dreht, kann man die Helligkeit der LEDs einstellen. Die gelbe LED reagiert zum Beispiel früher als die grüne LED, da sie eine geringere Flussspannung hat. Dies ist abhängig von der Bestückung des jeweilig eingesetzten Dual-LED-Bricks, welche LEDs vorhanden sind.



```
// DE_34 DA Beispiele Da Brick I2Cs und Poti

#include <Wire.h>
#include <avr/pgmspace.h>

// 1E=GND,1c=Open,1a=VCC AD5622  JUmper offen dann 1Ch
// 0001 1aa0
// aber nur aa=01 10 11 sind  moegliche Adressen

#define i2cdasel1 (0x1a>>1)  // ADRESSE Einstellen 1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // ADRESSE Einstellen 1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // ADRESSE Einstellen 1a,1c,1e


void i2c_da_write_command(unsigned char i2cbaseadr, unsigned short cmdvalue)
{
  // BIt15,14=0 13,12=pd (std=0) dann 11..0 = DA Value
  cmdvalue = cmdvalue & 0xFFF; // 12 Bits valide // 0..4095 Wertebereich
  Wire.beginTransmission(i2cbaseadr); // I2C Start senden
  Wire.write((cmdvalue>>8)&0xff); //  dann MSB zuerst
  Wire.write(cmdvalue&0xff);  // danach das LSB
  Wire.endTransmission(); // I2C beenden
}

void setup() {
  Wire.begin(); // i2C Initialisieren
}

void loop() {
  int daval=0; // Als Zwischenvariable verwendet
  int poti = analogRead(A0); // a1,a2,a3 kann man austauschen
  daval = poti << 2; // 0..1023 -> 0..4095  UMRECHNUNG
  // Alle moeglichen I2C Adresse ausgeben der Einfachheit halber
  i2c_da_write_command(i2cdasel1,daval); // Hier mal alle DA Umsetzer
  i2c_da_write_command(i2cdasel2,daval); // an sprechen die moeglich sind
  i2c_da_write_command(i2cdasel3,daval); // Nur einer wird aber verwendet
}
```

Was passiert? Die beiden LEDs lassen sich in der Helligkeit mit dem Poti einstellen. Es gibt dabei auch einen kleinen „toten“ Bereich durch die Schwellspannung der LEDs.

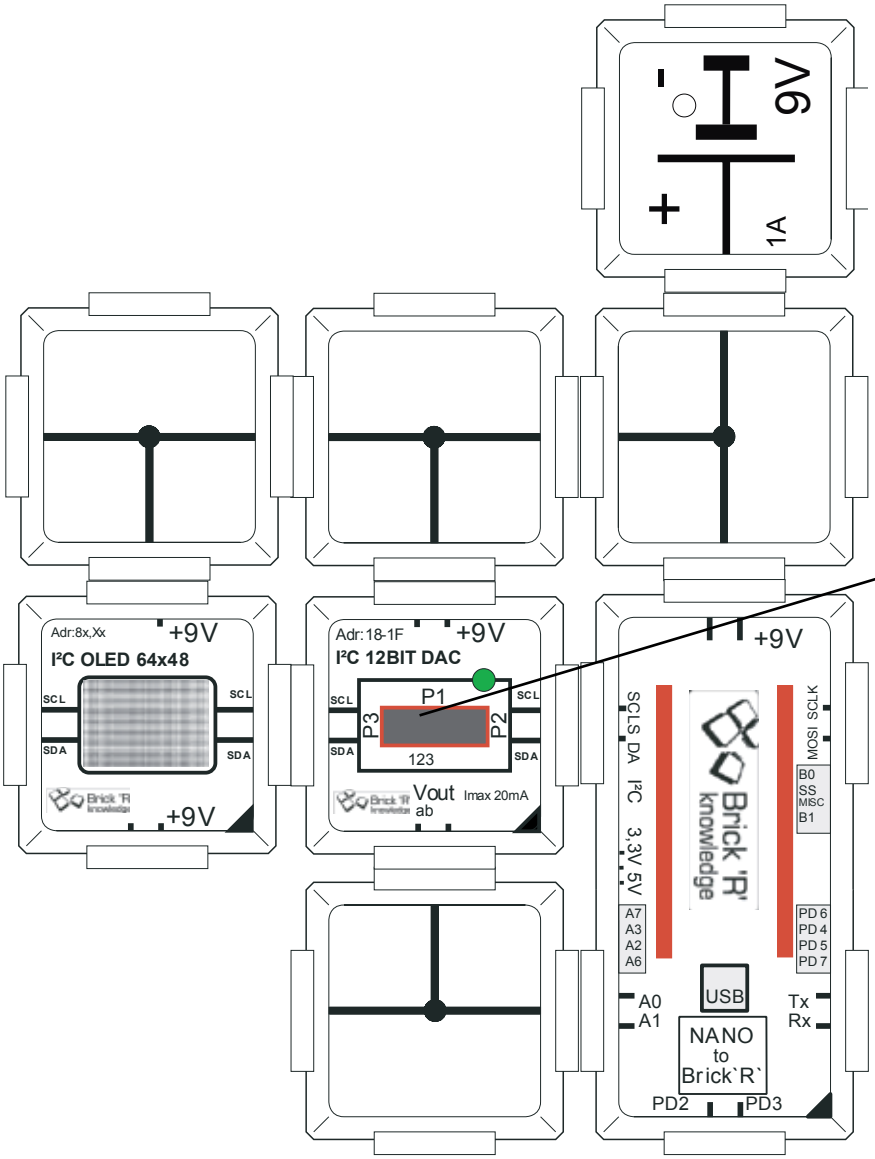
11.5 OLED und DA Umsetzer am AD Umsetzer

Wenn man den D/A-Umsetzer mit dem eingebauten A/D-Umsetzer und einer OLED kombiniert, kann man ganz interessante Experimente durchführen. Hier ist das Mini-Oszilloskope-Beispiel zur Darstellung eingesetzt. Der D/A-Umsetzer wird mit einer Rampe angesteuert, das heißt die Spannung steigt von 0V auf 5V an. Der Wert wird mit dem A/D-Umsetzer gemessen und als Text ausgegeben oder grafisch dargestellt. Die Auflösung des A/D-Umsetzers ist hier nicht so hoch, wie bei dem D/A-Umsetzer. Daher werden ein paar Werte identisch ausgegeben, obwohl der D/A-Umsetzer einen steigenden Wert ausgibt.

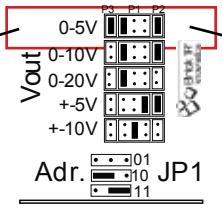
Das Beispiel kann auch leicht abgeändert werden, indem statt daval++ eine Formel eingesetzt wird und damit eine andere Wertesequenz ausgegeben wird. Der Phantasie sind hier keine Grenzen gesetzt.

Wichtig: Kontrolle der Jumperstellung! Der Ausgang des DA-Umsetzers ist hier ohne SCHUTZSCHALTUNG! an den AD Umsetzer geschaltet.
Die Jumperstellung muss unbedingt stimmen, da der Brick sonst Schaden nehmen kann. Nur 0 bis 5V sind als Eingangs-Bereich erlaubt, der DA-Umsetzer hat aber bei anderer Jumperstellung auch andere Ausgangsbereiche die für den DA-Umsetzer eine Gefahr darstellen können,

Man kann mit der Schaltung elegant die Schwellwerte von LEDs messen, Dazu testweise einmal die Dual-LED-Bricks am T-Stück links anschalten.



Was passiert? Auf der OLED sollte eine Rampe dargestellt werden, die dann langsam durchscrollt.

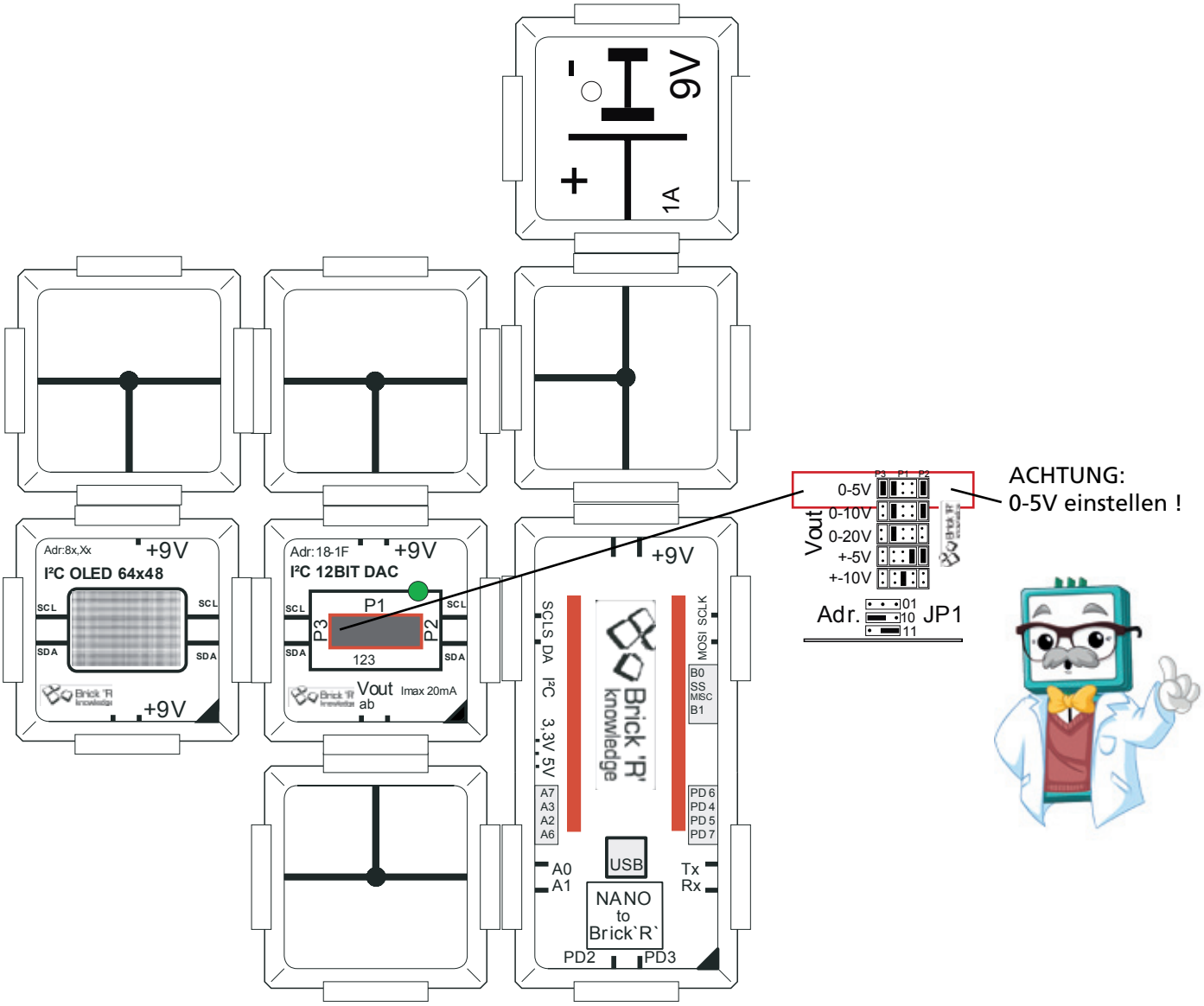


ACHTUNG:
0-5V einstellen
!

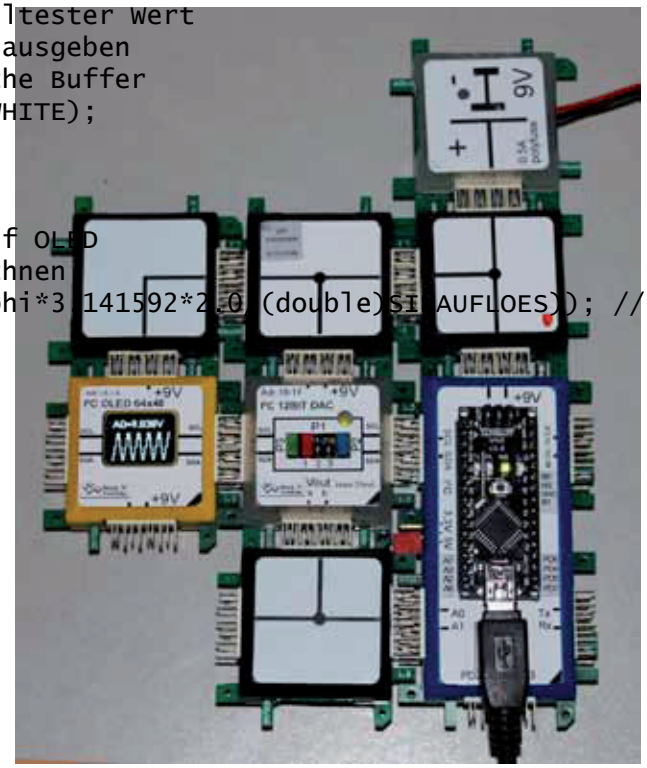
```
// DE_35 OLED Beispiele - DA Umsetzer am AD
Umsetzer

#include <Wire.h>
#include <avr/pgmspace.h>

// Hier ggf Adresse anpassen 78 oder 7A je
nach Schalter
#define i2coledssd (0x7A>>1)  // default ist
7A

// -------------------------OLED ------------
---------------------------------
...
// -------------------------END OLED --------
---------------------------------------


// 1E=GND,1c=Open,1a=VCC AD5622  JUmper offen
dann 1Ch
// 0001 1aa0
// aber nur aa=01 10 11 sind  moegliche Ad-
ressen

#define i2cdasel1 (0x1a>>1)  // ADRESSE Ein-
stellen 1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // ADRESSE Ein-
stellen 1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // ADRESSE Ein-
stellen 1a,1c,1e


void i2c_da_write_command(unsigned char i2c-
baseadr, unsigned short cmdvalue)
{
  // BIt15,14=0 13,12=pd (std=0) dann 11..0 =
DA Value
  cmdvalue = cmdvalue & 0xFFF; // 12 Bits va-
lide // 0..4095 Wertebereich
  Wire.beginTransmission(i2cbaseadr); // I2C
Start senden
  Wire.write((cmdvalue>>8)&0xff); //  dann
MSB zuerst
  Wire.write(cmdvalue&0xff);  // danach das
LSB
  Wire.endTransmission(); // I2C beenden
}

char advalbuf[64]; // loop buffer zyklisch

void setup() {
  Wire.begin(); // I2C Initialisieren
  i2c_oled_initall(i2coledssd);  // OLED ini-
talisieren
  for (int i=0; i<64; i++) advalbuf[i]=47; //
Zykl. Buffer
}

void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // zyklischer
Counter
  static int daval=0; // DA Wert fuer
Ausgabe
  int poti11 = analogRead(A0); // Einle-
sen A/D-Umsetzer
  char buffer[40]; // Textbuffer zusa-
etzlich
  disp_buffer_clear(COLOR_BLACK); //
Loeschen des Buffers
  double  p1 = (poti11*5000.0)/1023.0;
// Umrechnen in mV
  int y1 =0; // Ausgabeposition
  sprintf(buffer, „A0=%d.%03dV“,(int)
p1/1000,(int)p1%1000);
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V
Max -> 30 pixel
  advalbuf[cxx++] = y1; // merken 0..4xx
V +-128
  disp_print_xy_lcd(2, 0, (unsigned char
*)buffer, COLOR_WHITE, 0);
  int i=0; // Counter fuer Spalten
  int yold  = advalbuf[(cxx+1)%64]; //
Letzer Wert
  for (i=0; i<63; i++) { // Dann alle
Spalten ausgeben
   y1 =advalbuf[(cxx+1+i)%64]; // aus
zyk. Buffer
   disp_line_lcd (i, yold, i, y1, CO-
LOR_WHITE);
   yold = y1; // und wieder nach old
  }
  if (cxx >63) cxx =0; // 0..63 Ringbuf-
fer
  disp_lcd_frombuffer();   // Bildschirm
ausgeben
  // DA Umsetzer alle Werte an DA Umset-
zer
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval++; // Ergibt Saegezahn
  if (daval>4095) daval = 0;
}
```

11.6 OLED und DA Umsetzer am AD Umsetzer Sinus

Hier ein Beispiel für die Ausgabe eines Sinus-Signals über den D/A-Umsetzer. Die Anzahl der Punkte und damit auch die Frequenz definiert man über SINAUFLOES. Die Frequenz ist auch durch die Periodendauer der Schleife bestimmt, hier begrenzt die Ausgabe der OLED die maximale Frequenz, die im Hz Bereich liegt.



```
// DE_36 OLED Beispiele - DA Umsetzer am AD Umsetzer mit Sinus

#include <Wire.h>
#include <avr/pgmspace.h>

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A
// -------------------------OLED -------------------------------------------
...
// -------------------------END OLED -----------------------------------------
char advalbuf[64]; // loop buffer zyklisch

void setup() {
  Wire.begin(); // I2C Initialisieren
  i2c_oled_initall(i2coledssd);  // OLED initalisieren
  for (int i=0; i<64; i++) advalbuf[i]=47; // Zykl. Buffer
}

#define SINAUFLOES 10 // n Punkte
void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // zyklischer Zaehler
  static int phi=0; // Phase fuer Sinus
  int daval = 0; // Zwischenspeicher
  int poti11 = analogRead(A0); // Einlesen A/D-Umsetzer
  char buffer[40]; // Textbuffer fuer Ausgabe
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (poti11*5000.0)/1023.0; // mV umrechnen
  int y1 =0; // Y-Position
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000);
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel
  advalbuf[cxx++] = y1; // merken 0..4xx V +-128
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // Counter fuer Spalten
  int yold  = advalbuf[(cxx+1)%64]; // Aeltester Wert
  for (i=0; i<63; i++) { // Alle Spalten ausgeben
   y1 =advalbuf[(cxx+1+i)%64]; // Zyklische Buffer
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1; // ist neues old
  }
  if (cxx >63) cxx =0; // 0..63
  disp_lcd_frombuffer();   // Ausgeben auf OLED
  // DA Umsetzer Sinus ausgeben und berechnen
  daval = (int)(2048 + 2047*sin((double)phi*3.141592*2.0/(double)SINAUFLOES)); //
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  phi++; // Phase neu
  if (phi>SINAUFLOES) phi = 0;
}
```



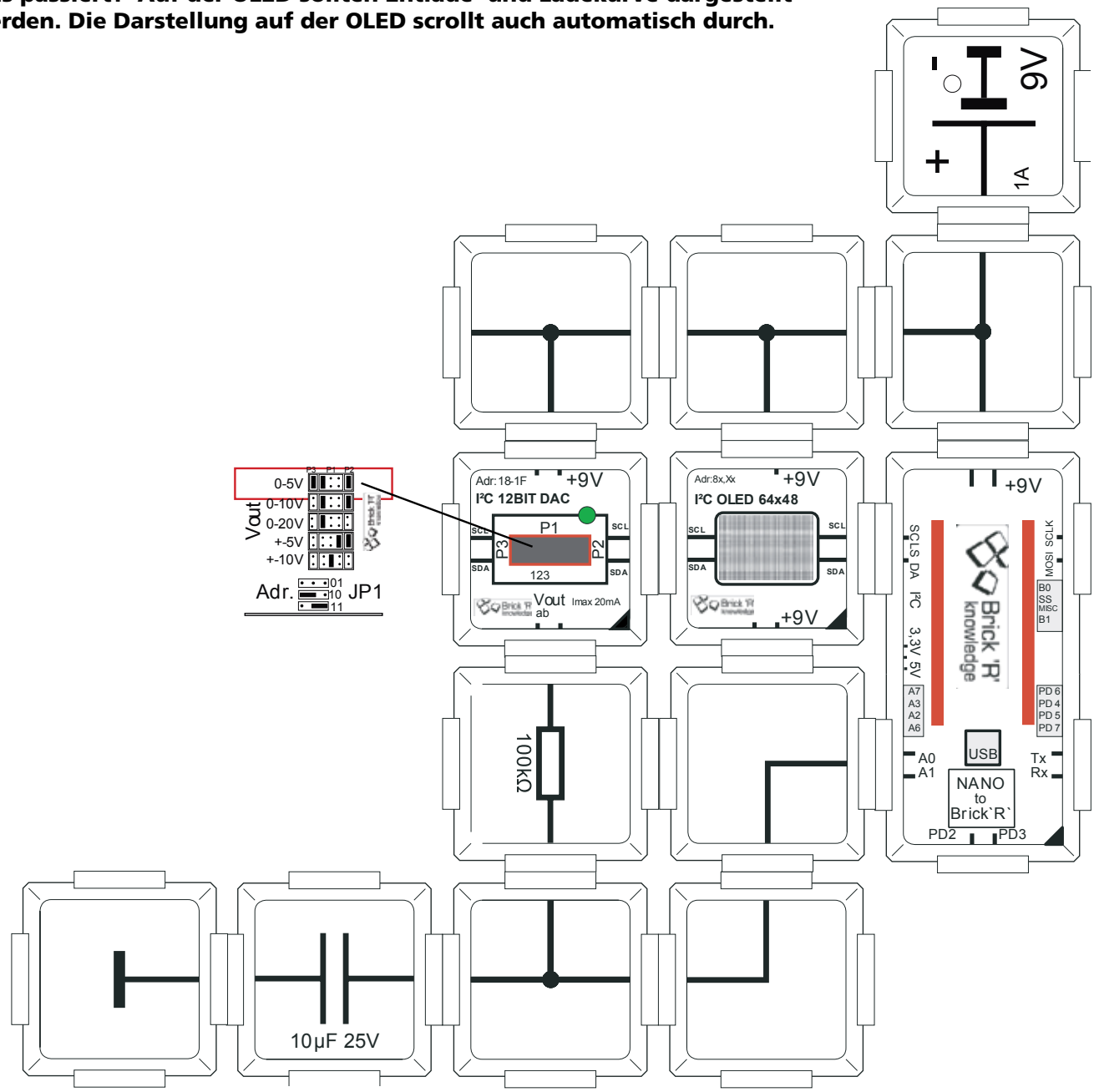
Was passiert? Auf der OLED sollte eine Sinuskurve dargestellt werden, die dann langsam durchscrollt.

12. Anwendungen

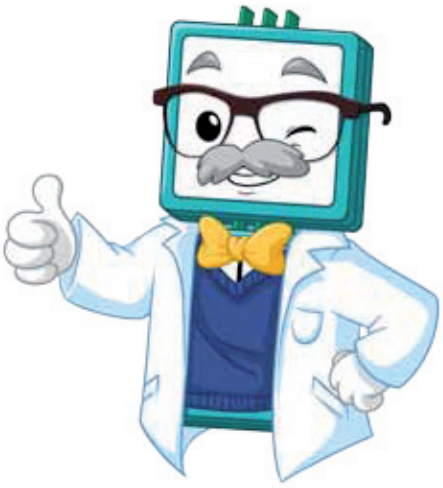
12.1 Entladekurve messen - Anzeige auf OLED

Der Kondensator wird alle 3 Sekunden im Wechsel geladen und dann wieder entladen. Auf dem Mini-Oszilloskop Display sieht man die Lade- und Entladekurve dargestellt. Zur Zeitnahme wird diesmal der eingebaute Timer millis() verwendet, der die abgelaufene Zeit seit dem Start des Programms in ms enthält (alle 54 Tage hat der Timer einen Überlauf auf 0). Die Messwerte werden wie früher beim Mini-Oszilloskop aufgezeichnet und ausgegeben. Will man das ganze beschleunigen, müssen Aufzeichnung und Wiedergabe von einander getrennt werden. In unserem Beispiel reicht die zeitliche Auflösung aber aus. Man erhält je nach CPU Version ca. 1-2 Entlade- und Ladekurven gleichzeitig auf dem OLED. Der D/A-Umsetzer ist für den Bereich 0-5V eingestellt, damit stimmt er auch mit dem Messbereich des A/D-Umsetzers von 0-5V überein.
Die Zeitkonstante kann man auch ausrechnen: t=R*C, bei uns 100000.0 Ohm * 10E-6 F = 1 sec. Dies kennzeichnet den Zeitpunkt, bei der der Kondensator 63% aufge- oder entladen wurde.

Was passiert? Auf der OLED sollten Entlade- und Ladekurve dargestellt werden. Die Darstellung auf der OLED scrollt auch automatisch durch.

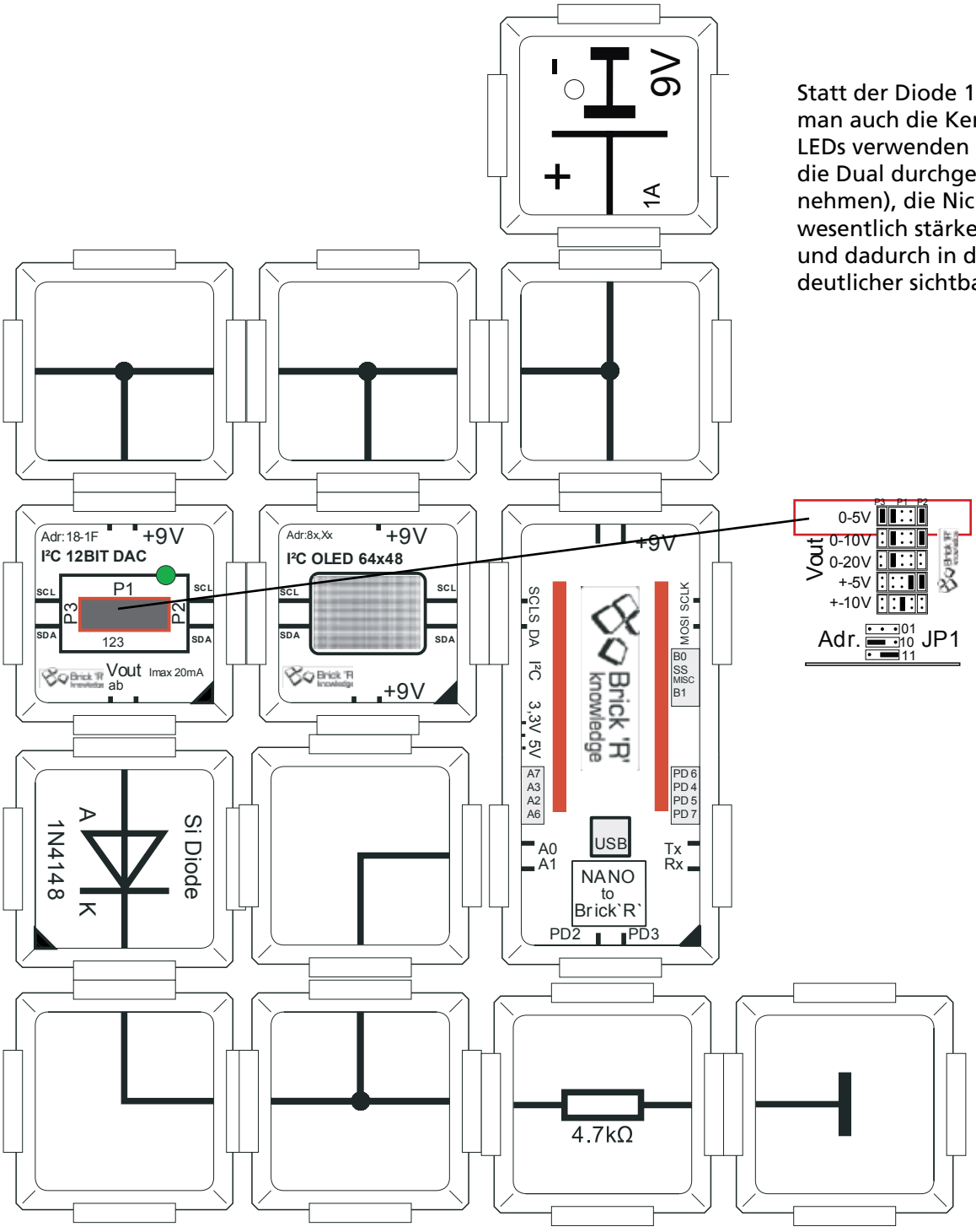


```
// DE 37 Anwendungen - Entladekurve
.....
int milisec=0; // merker fuer ms
void setup() {
  Wire.begin(); // I2C Initialisierung
  i2c_oled_initall(i2coledssd);  // OLED Initialsierung
  for (int i=0; i<64; i++) advalbuf[i]=47; // Default fuer Buffer
}

void loop() {  // Schleife
 // 64x48 Pixel OLED
  static int cxx = 0; // Cyklischer Zaehler
  static int state = 0; // State system verwenden fuer DA
  int ms = 0; // Messzeit ms
  static int daval = 0; // Ausgabewert
  int ana0 = analogRead(A0); // Einlesen A/D-Umsetzer
  char buffer[40]; // ASCII Buffer fuer V Anzeige
  disp_buffer_clear(COLOR_BLACK); //  BIldschirm loeschen
  double  p1 = (ana0*5000.0)/1023.0;  // in mV
  int y1 =0; // Position y
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000);
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel
  advalbuf[cxx++] = y1; // merken 0..4xx V +-128
  if (cxx >63) cxx =0;  // Zyklischer Zaehler
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // Fuer Spalten
  int yold  = advalbuf[(cxx+1)%64]; // Aktueller AD Wert
  for (i=0; i<63; i++) { // Fuer alle Spalten durchlaufen
   y1 =advalbuf[(cxx+1+i)%64]; // Aus dem Ringbuffer holen
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1; // Neues old
  }
  disp_lcd_frombuffer(); // Auf den Bildscirm
  // Kondensator aufladen und entladen
  // Dazu Sprungfunktion verwenden
  // Im festen Intervall messen
  //
  ms=millis();   // aktuelle Zeit
  if (ms > (milisec+3000)) { // 3sec warten
    switch(state) { // State abhaengeig
      case 0:
        daval = 0xfff; // Aufladen
        state = 1; // danach in State 1 nach 3sec
      break;
      case 1: // Dann wieder entlanden
        daval = 0; // nach 3 sec
        state = 0; // UNd state wieder auf 0
        break;
      default: // Im Falle eines Falles
        state = 0; // 0 immer sicher
    }
    milisec = ms; // Neue Zeit
  }
  // DA Umsetzer
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
}
```



12.2 OLED und einfache Diodenkennlinie

Man kann auch die Kennlinie eines Bauteils aufnehmen. Hier eine ganz einfache Methode für den positiven Teil der Kennlinie. Der Strom durch die Diode wird indirekt über den Spannungsabfall am Widerstand gemessen. Die Spannung wird über den D/A-Umsetzer zugeführt. Man erkennt die typische nichtlinieare Kennline der Diode in Durchlassrichtung. Diese Anordnung zur Messung kann man noch beliebig verfeinern. Mit einem Offset bei der Masse der Diode an der Messtrecke lassen sich auch die negativen Anteile messen. Der Phantasie sind hier keine Grenzen gesetzt. Der D/A-Umsetzer kann auch negative Spannungen erzeugen, wenn man die Steckbrücken entsprechend setzt. Aber Vorsicht: der A/D-Umsetzer verträgt nur Spannungen von 0V bis +5V.



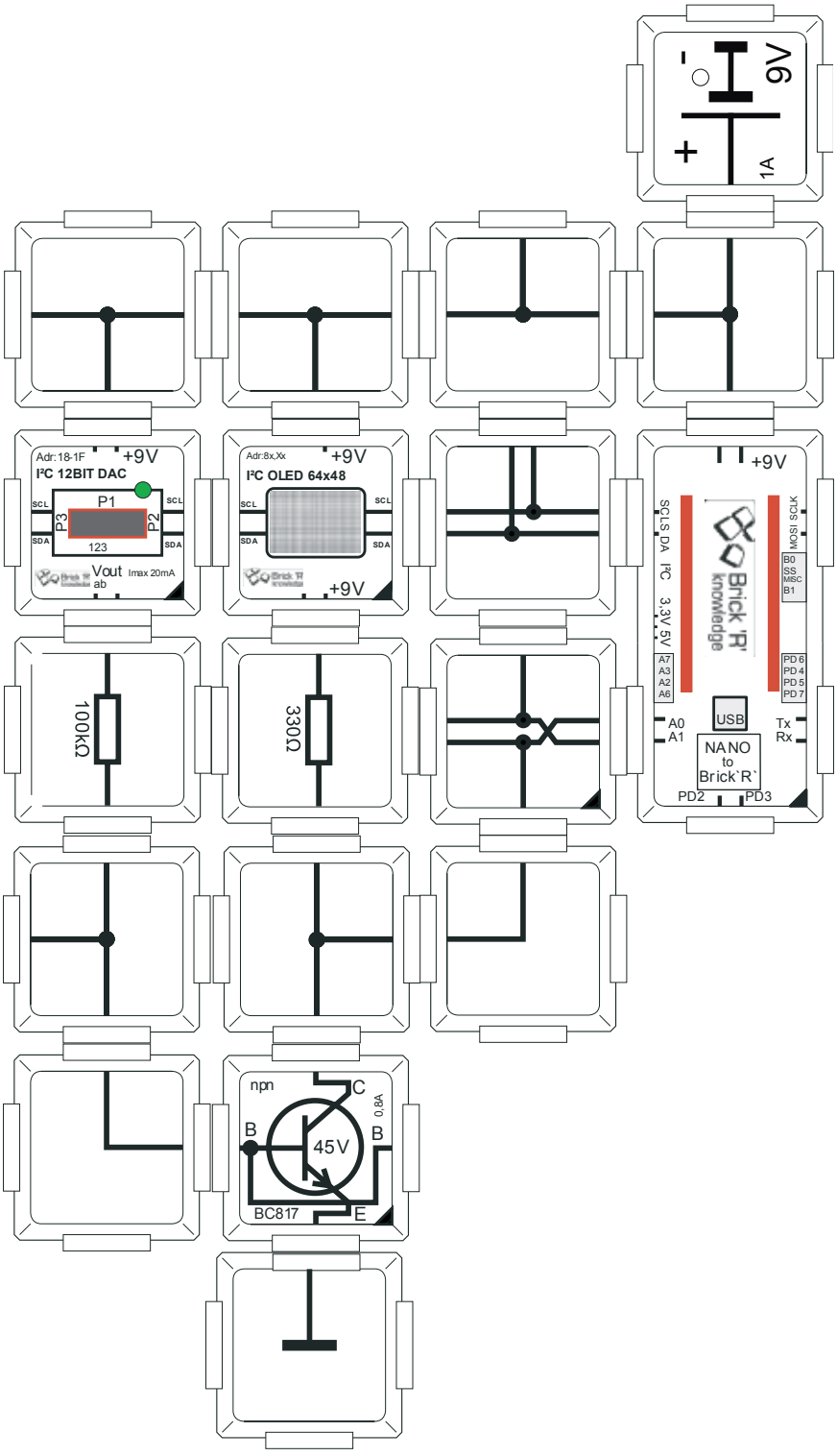
Statt der Diode 1N4148 kann man auch die Kennlinie von LEDs verwenden (einfach die Dual durchgehende LEDs nehmen), die Nichtlinearität ist wesentlich stärker ausgeprägt und dadurch in der Kennlinie deutlicher sichtbar.

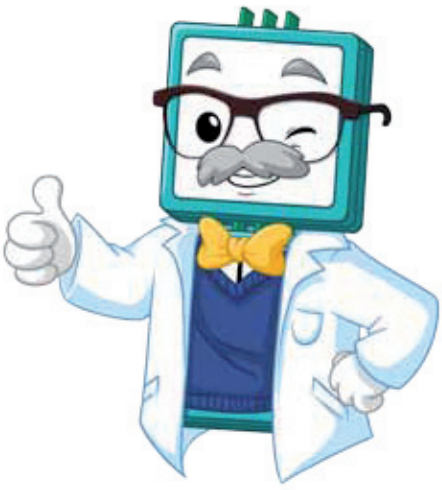
```
DE_38
... wie vorher

char advalbuf[64]; // loop

void setup() {
  Wire.begin(); // I2C Initialisierung
  i2c_oled_initall(i2coledssd);  // OLED Initialsierung
  for (int i=0; i<64; i++) advalbuf[i]=47; // Default fuer Buffer
}

void loop() {   // Schleife
 // 64x48 Pixel OLED
  static int cxx = 0; // Ringbuffer
  static int daval = 0;  // fuer Ausgabe
  int ana0 = analogRead(A0); // Einlesen A/D-Umsetzer
  char buffer[40]; // ASCII Textbuffer
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (ana0*5000.0)/1023.0;
  int y1 =0; // Y Position
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000);
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel
  advalbuf[cxx++] = y1; // merken 0..4xx V +-128
  if (cxx >63) cxx =0;  // Ringbuffer 0..63
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // Spalte
  int yold  = advalbuf[(cxx+1)%64];;
  for (i=0; i<63; i++) { // ALles ausgeben
   y1 =advalbuf[(cxx+1+i)%64];
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1;
  }
  disp_lcd_frombuffer();
  // DA Umsetzer Werte ausgeben Saegezahn !
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval  += (4096/40); // Kennlinie auf einen Schirm
  if (daval > 4095) daval = 0;
}
```

Was passiert? Auf der OLED sollte die typische Diodenkennlinie erkennbar sein, die auch hier wieder durchscrollt.

12.3 OLED und Transistor in Emitterschaltung

Die Aufnahme einer Kennlinie am Beispiel eines Transistors. Wichtig: Der Transistor wird hier mit der Spannung des D/A-Umsetzers angesteuert, die über den Widerstand zur Basis geleitet wird. Er bestimmt schließlich den Stromfluss in die Basis. Der Transistor verstärkt den Strom durch die Basis zum Emitter (je nach Typ z.B.100x) und bestimmt damit den Strom der zwischen Emitter und Kollektor fließt.
Das Programm zeigt die Kennlinie, wie bei einem Oszilloskop, als fortlaufende Kurve. Zusätzlich kann auch eine stehende Darstellung implementiert werden. (Aufgabe für den Leser!)



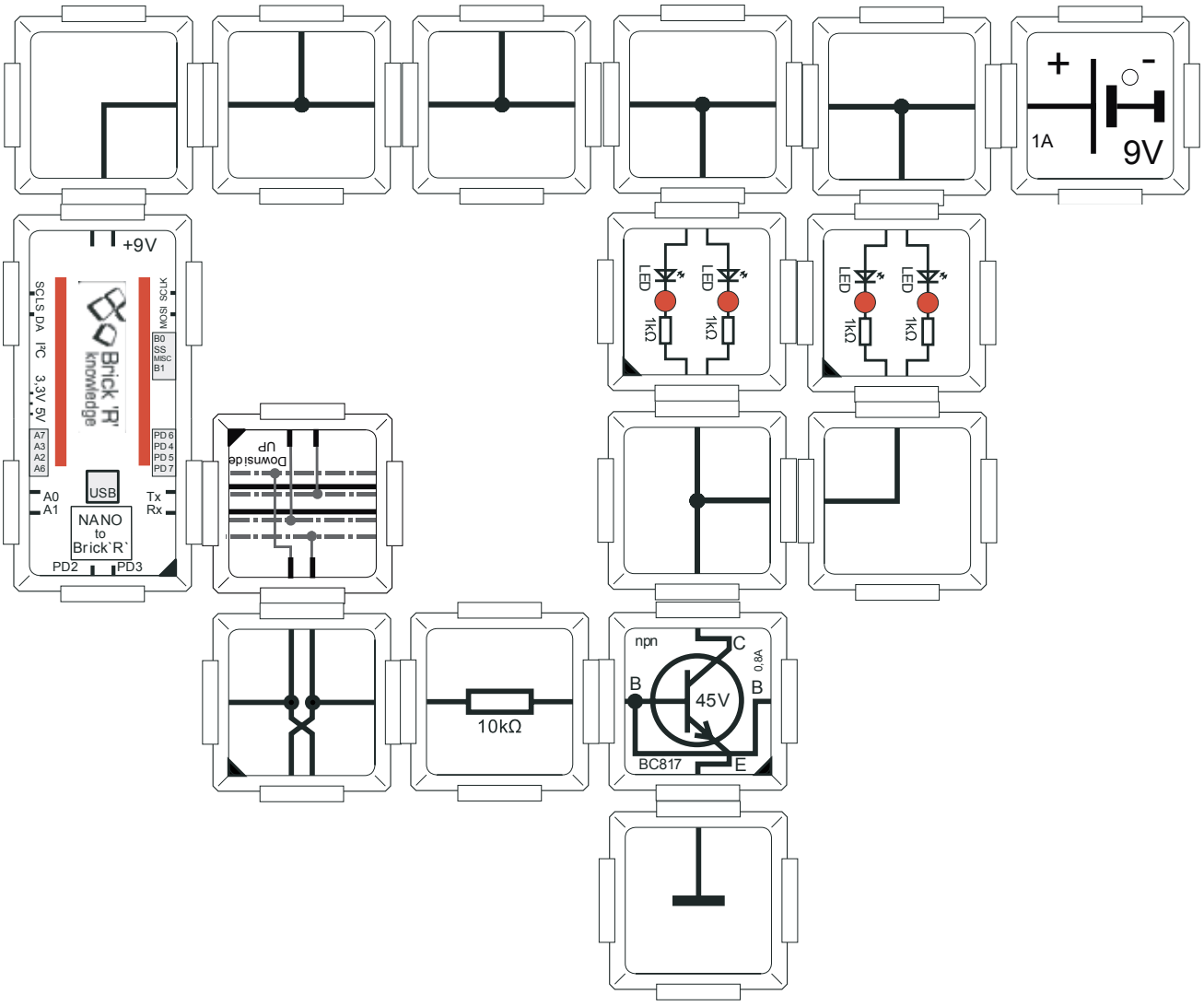


```
// DE_39 Anwendungen - Transistorkennlinie
... wie zuvor
//


char advalbuf[64]; // loop

void setup() {
  Wire.begin(); // I2C Initialisierung
  i2c_oled_initall(i2coledssd);  // OLED Initialsierung
  for (int i=0; i<64; i++) advalbuf[i]=47; // Default fuer Buffer
}

void loop() {   // Schleife
 // 64x48 Pixel OLED
  static int cxx = 0; // Ringbuffer
  static int daval = 0;  // fuer Ausgabe
  int ana0 = analogRead(A0); // Einlesen A/D-Umsetzer
  char buffer[40]; // ASCII Textbuffer
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (ana0*5000.0)/1023.0;
  int y1 =0; // Y Position
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000);
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel
  advalbuf[cxx++] = y1; // merken 0..4xx V +-128
  if (cxx >63) cxx =0;  // Ringbuffer 0..63
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // Spalte
  int yold  = advalbuf[(cxx+1)%64];;
  for (i=0; i<63; i++) { // ALles ausgeben
   y1 =advalbuf[(cxx+1+i)%64];
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1;
  }
  disp_lcd_frombuffer();
  // DA Umsetzer Werte ausgeben Saegezahn !
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval  += (4096/40); // Kennlinie auf einen Schirm
  if (daval > 4095) daval = 0;
}
```

Was passiert? Auf der OLED sollte die typische Transistorkennlinie in Emitterschaltung erkennbar sein, die ebenfalls durchscrollt.

12.4 Schalten von Lasten 1

Der Ausgang des NANOs sollte normalerweise mit nicht mehr als 20mA belastet werden. Will man Lasten schalten, die mehr Strom brauchen, wird ein elektronischer Schalter mit einer höheren Belastbarkeit benötigt. Dies kannn z.B. mit einem Transistor geschehen. Der BC817 kann normalerweise mit 500mA (Peak 800mA) belastet werden. Der Maximalwert sollte nicht auf Dauer erreicht werden. Dennoch liegt der Wert deutlich über dem des Arduino Nano Ausgangs und die nachfolgende Schaltung ist in der Lage mehr Last zu schalten. Dieses Beispiel zeigt eine einfache Last mit mehreren parallelen LEDs. Dazu ein Testprogramm, das die LEDs blinken lässt. Hier wird Port PD7 zur Ausgabe verwendet. Bitte auf die genaue Orientierung der Bricks achten, damit auch wirklich Port 7 an die Basis des Transistors gelangt. Der Widerstand von 10k begrenzt den Strom durch die Basis. Dieser kann auch kleiner gewählt werden, wenn größere Lasten geschalten werden sollen. Wird er zu groß gewählt, schaltet der Transistor nicht vollständig durch. Wenn der Widerstand zu klein gewählt wird, belastet der daraus resultierende Basisstrom den Ausgang des IO-Ports vom NANO mit einer zu hohen Stromstärke.

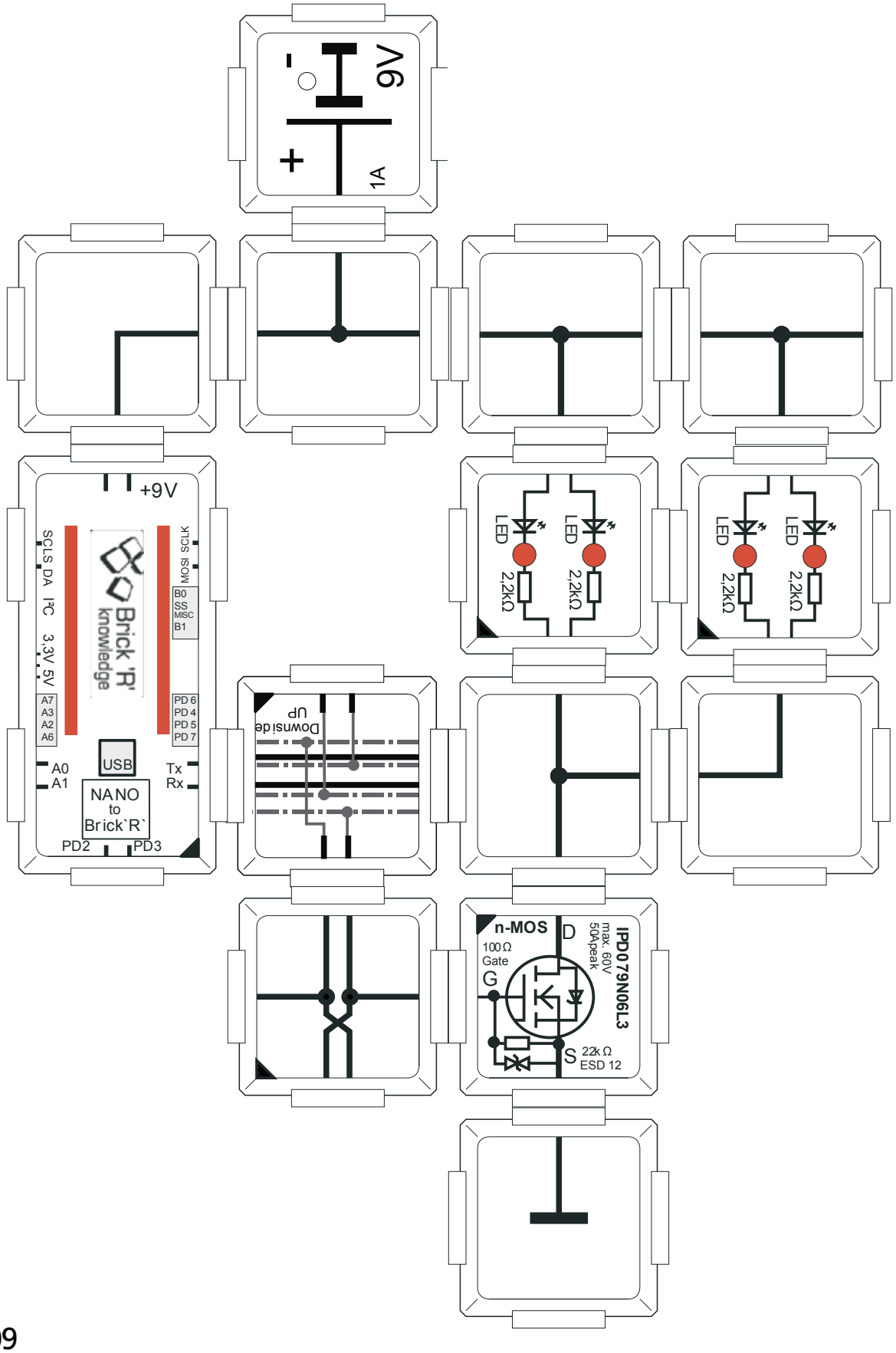


```
// DE_40 Schalten von Lasten 1

#define PORTLAST 7 // als  PD7 verwenden

void setup() {
  pinMode(PORTLAST,OUTPUT); // Port 13 als Ausgang schalten
}

void loop() { // Schleife
  digitalWrite(PORTLAST,HIGH); // Ausgang auf hohen Pegel schalten
  delay(1000); // 1 Sekunde Verzoegerung
  digitalWrite(PORTLAST,LOW); // Ausgang auf niedrigen Pegel schalten
  delay(1000); // Eine weitere Sekunde warten
}
```

Was passiert? Die LEDs sollten im Sekundentakt gemeinsam an- und ausgehen.

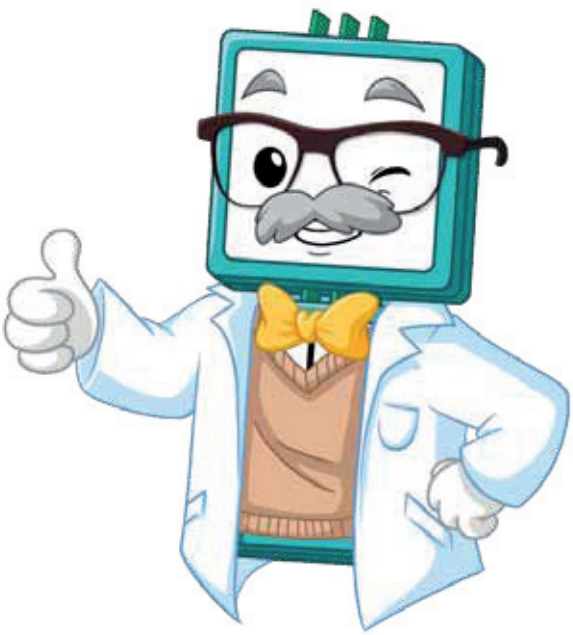
12.5 Schalten von Lasten 2

Mit MOSFETs kann man elegant große Lasten schalten, dabei wird nur mit einem Spannungspegel gearbeitet. Man braucht also das Gate nicht mit einem Widerstand zu schützen.



```
// DE_41 Schalten von Lasten 2

#define PORTLAST 7 // als PD7 verwenden

void setup() {
  pinMode(PORTLAST,OUTPUT); // Port 13 als Ausgang schalten
}

void loop() { // Schleife
  digitalWrite(PORTLAST,HIGH); // Ausgang auf hohen Pegel schalten
  delay(1000); // 1 Sekunde Verzoegerung
  digitalWrite(PORTLAST,LOW); // Ausgang auf niedrigen Pegel schalten
  delay(1000); // Eine weitere Sekunde warten
}
```

Was passiert? Die LEDs gehen im Sekundentakt gemeinsam an- und aus.



13. ANHANG

13.1 Listing - Sieben-Segment-Anzeige -- Bibliothek mit Beispiel

Hier der vollständige Code inklusive der allgemeinen Bibliothek für die Sieben-Segment-Anzeige. Der eigentliche Bibliothekscode ist in schwarz gedruckt, ein Beispielcode in Blau. Den schwarzen Bereich dann in die Beispiele auf den Experimentalseiten kopieren die dort nur mit „...“ angedeutet sind. Wir können hier leider nicht jedes Beispiel abdrucken, da es sonst den Umfang sprengen würde. Die kompletten Codebeispiele kann man aber auch von der BrickRKnowledge Homepage herunterladen.

```
// DE_15 7segment Anzeige als I2C Brick
#include <Wire.h>

// 8574T standardmaessig verbaut
#define i2cseg7x2alsb1 (0x40>>1) // 7Bit
#define i2cseg7x2amsb1 (0x42>>1) //
#define i2cseg7x2blsb1 (0x44>>1)
#define i2cseg7x2bmsb1 (0x46>>1)
#define i2cseg7x2clsb1 (0x48>>1)
#define i2cseg7x2cmsb1 (0x4A>>1)
#define i2cseg7x2dlsb1 (0x4C>>1)
#define i2cseg7x2dmsb1 (0x4E>>1)

// 8574AT optional
#define i2cseg7x2alsb2 (0x70>>1)
#define i2cseg7x2amsb2 (0x72>>1)
#define i2cseg7x2blsb2 (0x74>>1)
#define i2cseg7x2bmsb2 (0x76>>1)
#define i2cseg7x2clsb2 (0x78>>1)
#define i2cseg7x2cmsb2 (0x7A>>1)
#define i2cseg7x2dlsb2 (0x7C>>1)
#define i2cseg7x2dmsb2 (0x7E>>1)

// Aufbau der Segmente Zuordnung zu den
// Bits, 0x80 ist fuer den DOT
// *************************
//          01
//       20    02
//          40
//       10    04
//          08
//                80
// *************************

// Umrechnungstabelle ASCII -> 7 Segment
// OFFSET Asciicode 32..5F entspricht Space
// bis Z
const unsigned char siebensegtable[] =
{
   0, // 20 Space
   0x30, // 21 !
   0x22, // 22 „
   0x7f, // 23 #
   0, // 24 $
   0, // 25 %
   0, // 26 &
   0x02, // 27 ,
```

```
0x39, // 28 (
0x0f, // 29 )
0, // 2A *
0x7f, // 2B +
0x04, // 2C ,
0x40, // 2D -
0x80, // 2E .
0x30, // 2F /
0x3f, // 30 0
0x06, // 31 1
0x5b, // 32 2
0x4f, // 33 3
0x66, // 34 4
0x6d, // 35 5
0x7c, // 36 6
0x07, // 37 7
0x7f, // 38 8
0x67, // 39 9
//
0, // 3A :
0, // 3B ;
0, // 3C <
0x48, // 3D =
0, // 3E >
0, // 3F ?
0x5c, // 40 @
0x77, // 41 A
0x7c, // 42 B
0x39, // 43 C
0x5e, // 44 D
0x79, // 45 E
0x71, // 46 F
0x67, // 47 G
0x76, // 48 H
0x06, // 49 I
0x86, // 4A J
0x74, // 4B K
0x38, // 4C L
0x37, // 4D M
0x54, // 4E N
0x5c, // 4F O
0x73, // 50 P
0xbf, // 51 Q
0x50, // 52 R
0x6d, // 53 S
0x70, // 54 T
0x3e, // 55 U
0x1c, // 56 V
0x9c, // 57 W
0x24, // 58 X
0x36, // 59 Y
0x5b, // 5A Z
0x39, // 5B [
0x30, // 5C
0x0f, // 5D ]
```

```
   0x08, // 5E _
   0 // 5F OHNE
};

// Umrechnen ASCII Code in Tabellenindex
unsigned int get_7seg(unsigned char asciicode)
{
    // Umrechnen 0..255 auf
    // 7 seg Tabellenindex
    // Dabei nur Zahlen und Grossbuchstaben
    // 20..5F
    // Rest wird auf diese gemappt
    asciicode = asciicode & 0x7f; // 7 bit only
    if (asciicode < 0x20) return (0); // Sonderzeichen nicht
    if (asciicode >= 0x60) asciicode = asciicode - 0x20; // KLeinbuchstaben
    return((~siebensegtable[asciicode-0x20])&0xff); // Index zurueck
}

// Anzeige eines einzelnen ASCII Zeichen, dass wird als
// char übergeben. Ausgabe ueber den 7 Segmentbrick
// Dazu die Segmentadresse als Parameter uebergeben
// Ohne DezimalPunkt ausgeben.
void display_seg1x(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C Adresse
  Wire.write(get_7seg(ch1)); // Tabellenidnex nehmen und dann ausgeben
  Wire.endTransmission(); // Ende I2C
}

// Ausgabe ohne Umrechnung, wenn eigene Zeichen verwendet werden
// sollen. Parameter ist der Bianry Code
void display_seg1xbin(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C Adresse
  Wire.write(ch1); // Binaercode direkt am Port ausgeben
  Wire.endTransmission(); // Ende I2C
}

// Start
void setup() {
  Wire.begin(); /// I2C Initialisieren
}

void loop() {
   // Anzeigen 8574T alle potentiellen Addressen ausgeben
   // Man kann so sehen welche Adresse man besetzt hat
  display_seg1x(i2cseg7x2amsb1,'4'); // eigene Adresse
  display_seg1x(i2cseg7x2alsb1,'0'); // ausgeben
  display_seg1x(i2cseg7x2bmsb1,'4'); // sind immer PAARE
  display_seg1x(i2cseg7x2blsb1,'4'); // Zwei Befehle fuer ein BRICK
  display_seg1x(i2cseg7x2cmsb1,'4');
  display_seg1x(i2cseg7x2clsb1,'8'); // von 40-4C
  display_seg1x(i2cseg7x2dmsb1,'4');
  display_seg1x(i2cseg7x2dlsb1,'C');
  // falls 8574AT vorhanden dann auch diese ausgeben
  display_seg1x(i2cseg7x2amsb2,'7'); // eigene Adresse
  display_seg1x(i2cseg7x2alsb2,'0'); // ausgeben
  display_seg1x(i2cseg7x2bmsb2,'7'); // hier
  display_seg1x(i2cseg7x2blsb2,'4'); // von
```

```
  display_seg1x(i2cseg7x2cmsb2,'7'); // 70..7c
  display_seg1x(i2cseg7x2clsb2,'8');
  display_seg1x(i2cseg7x2dmsb2,'7');
  display_seg1x(i2cseg7x2dlsb2,'C');
}
```

13.2 Listing OLED Bibliothek mit Beispiel

Hier der vollständige Code inklusive der Bibliothek für die OLED-Anzeige. Der eigentliche Bibliothekscode ist in schwarz gedruckt, das Beispiel in Blau. Entsprechende Teile dann in die Beispiele kopieren die dort nur mit „...“ angedeutet sind.

```
// DE_27 OLED Beispiele - Pixelroutinen

#include <Wire.h> // I2C Bibliothek
#include <avr/pgmspace.h> // Zugriff ins ROM

// Hier ggf Adresse anpassen 78 oder 7A je nach Schalter
#define i2coledssd (0x7A>>1)  // default ist 7A

// -------------------------OLED -----------------------------------------
// GLO066-D-M2005 -- SSD 1306 driver
// 011110sr  s=sa r=rw bei ssd1306 sa = adressbit optional zu setzen
// 0x78
// 0x78 und 0x7A je nach schalter...

//
// ******************************************************************
// RDK 2014 FONT Sets
//

/*******************************************************************************
*                                                                              *
* This file is generated by BitFontCreator Pro v3.0                           *
* by Iseatech Software   http://www.iseasoft.com/bfc.htm                       *
*                        support@iseatech.com                                  *
*                                                                              *
* Font name: Arial                                                             *
* Font width: 0 (proportional font)                                            *
* Font height: 27                                                              *
* Encode: Unicode                                                              *
*                                                                              *
* Data length: 8 bits                                                          *
* Invert bits: No                                                              *
* Data format: Big Endian, Row based, Row preferred, Unpacked                  *
*                                                                              *
* Create time: 13:31 12-01-2011                                                *
*******************************************************************************/
const unsigned char fontArial14h_data_tablep[] PROGMEM =
{

/* character 0x0020 (, ,): [width=3, offset= 0x0000 (0) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x0021 (,!‘): [width=2, offset= 0x000E (14) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x00, 0x80, 0x00, 0x00, 0x00,

/* character 0x0022 (,“‘): [width=4, offset= 0x001C (28) ] */
  0x00, 0x00, 0x00, 0xA0, 0xA0, 0xA0, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x0023 (,#‘): [width=6, offset= 0x002A (42) ] */
```

```
  0x00, 0x00, 0x00, 0x28, 0x28, 0xF8, 0x50, 0x50,
  0xF8, 0xA0, 0xA0, 0x00, 0x00, 0x00,

/* character 0x0024 (,$‘): [width=6, offset= 0x0038 (56) ] */
  0x00, 0x00, 0x00, 0x70, 0xA8, 0xA0, 0x70, 0x28,
  0x28, 0xA8, 0x70, 0x20, 0x00, 0x00,

/* character 0x0025 (,%‘): [width=10, offset= 0x0046 (70) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x62, 0x00,
  0x94, 0x00, 0x94, 0x00, 0x68, 0x00, 0x0B, 0x00,
  0x14, 0x80, 0x14, 0x80, 0x23, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,

/* character 0x0026 (,&‘): [width=7, offset= 0x0062 (98) ] */
  0x00, 0x00, 0x00, 0x30, 0x48, 0x48, 0x30, 0x50,
  0x8C, 0x88, 0x74, 0x00, 0x00, 0x00,

/* character 0x0027 (,‘‘): [width=2, offset= 0x0070 (112) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x0028 (,(,): [width=4, offset= 0x007E (126) ] */
  0x00, 0x00, 0x00, 0x20, 0x40, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x40, 0x20, 0x00,

/* character 0x0029 (,)‘): [width=4, offset= 0x008C (140) ] */
  0x00, 0x00, 0x00, 0x80, 0x40, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x40, 0x80, 0x00,

/* character 0x002A (,*‘): [width=4, offset= 0x009A (154) ] */
  0x00, 0x00, 0x00, 0x40, 0xE0, 0x40, 0xA0, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x002B (,+‘): [width=6, offset= 0x00A8 (168) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x20, 0x20, 0xF8,
  0x20, 0x20, 0x00, 0x00, 0x00, 0x00,

/* character 0x002C (,,‘): [width=3, offset= 0x00B6 (182) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x40, 0x40, 0x40, 0x00,

/* character 0x002D (,-‘): [width=4, offset= 0x00C4 (196) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0xE0, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x002E (,.‘): [width=3, offset= 0x00D2 (210) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x40, 0x00, 0x00, 0x00,

/* character 0x002F (,/‘): [width=3, offset= 0x00E0 (224) ] */
  0x00, 0x00, 0x00, 0x20, 0x20, 0x40, 0x40, 0x40,
  0x40, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x0030 (,0‘): [width=6, offset= 0x00EE (238) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x88, 0x88, 0x88,
```

```
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0031 (,1‘): [width=6, offset= 0x00FC (252) ] */
  0x00, 0x00, 0x00, 0x20, 0x60, 0xA0, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x00, 0x00, 0x00,

/* character 0x0032 (,2‘): [width=6, offset= 0x010A (266) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x08, 0x08, 0x10,
  0x20, 0x40, 0xF8, 0x00, 0x00, 0x00,

/* character 0x0033 (,3‘): [width=6, offset= 0x0118 (280) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x08, 0x30, 0x08,
  0x08, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0034 (,4‘): [width=6, offset= 0x0126 (294) ] */
  0x00, 0x00, 0x00, 0x10, 0x30, 0x50, 0x50, 0x90,
  0xF8, 0x10, 0x10, 0x00, 0x00, 0x00,

/* character 0x0035 (,5‘): [width=6, offset= 0x0134 (308) ] */
  0x00, 0x00, 0x00, 0x78, 0x40, 0x80, 0xF0, 0x08,
  0x08, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0036 (,6‘): [width=6, offset= 0x0142 (322) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x80, 0xF0, 0x88,
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0037 (,7‘): [width=6, offset= 0x0150 (336) ] */
  0x00, 0x00, 0x00, 0xF8, 0x10, 0x10, 0x20, 0x20,
  0x40, 0x40, 0x40, 0x00, 0x00, 0x00,

/* character 0x0038 (,8‘): [width=6, offset= 0x015E (350) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x88, 0x70, 0x88,
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0039 (,9‘): [width=6, offset= 0x016C (364) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x88, 0x88, 0x78,
  0x08, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x003A (,:‘): [width=3, offset= 0x017A (378) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00,
  0x00, 0x00, 0x80, 0x00, 0x00, 0x00,

/* character 0x003B (,;‘): [width=3, offset= 0x0188 (392) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x00, 0x00,
  0x00, 0x00, 0x80, 0x80, 0x80, 0x00,

/* character 0x003C (,<‘): [width=6, offset= 0x0196 (406) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x08, 0x70, 0x80,
  0x70, 0x08, 0x00, 0x00, 0x00, 0x00,

/* character 0x003D (,=‘): [width=6, offset= 0x01A4 (420) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF8, 0x00,
  0xF8, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x003E (,>‘): [width=6, offset= 0x01B2 (434) ] */
```

```
  0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0x70, 0x08,
  0x70, 0x80, 0x00, 0x00, 0x00, 0x00,

/* character 0x003F (,?‘): [width=6, offset= 0x01C0 (448) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x08, 0x10, 0x20,
  0x20, 0x00, 0x20, 0x00, 0x00, 0x00,

/* character 0x0040 (,@‘): [width=11, offset= 0x01CE (462) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1F, 0x00,
  0x60, 0x80, 0x4D, 0x40, 0x93, 0x40, 0xA2, 0x40,
  0xA2, 0x40, 0xA6, 0x80, 0x9B, 0x00, 0x40, 0x40,
  0x3F, 0x80, 0x00, 0x00,

/* character 0x0041 (,A‘): [width=8, offset= 0x01EA (490) ] */
  0x00, 0x00, 0x00, 0x10, 0x28, 0x28, 0x28, 0x44,
  0x7C, 0x82, 0x82, 0x00, 0x00, 0x00,

/* character 0x0042 (,B‘): [width=7, offset= 0x01F8 (504) ] */
  0x00, 0x00, 0x00, 0xF8, 0x84, 0x84, 0xFC, 0x84,
  0x84, 0x84, 0xF8, 0x00, 0x00, 0x00,

/* character 0x0043 (,C‘): [width=7, offset= 0x0206 (518) ] */
  0x00, 0x00, 0x00, 0x38, 0x44, 0x80, 0x80, 0x80,
  0x80, 0x44, 0x38, 0x00, 0x00, 0x00,

/* character 0x0044 (,D‘): [width=7, offset= 0x0214 (532) ] */
  0x00, 0x00, 0x00, 0xF0, 0x88, 0x84, 0x84, 0x84,
  0x84, 0x88, 0xF0, 0x00, 0x00, 0x00,

/* character 0x0045 (,E‘): [width=6, offset= 0x0222 (546) ] */
  0x00, 0x00, 0x00, 0xF8, 0x80, 0x80, 0xF8, 0x80,
  0x80, 0x80, 0xF8, 0x00, 0x00, 0x00,

/* character 0x0046 (,F‘): [width=6, offset= 0x0230 (560) ] */
  0x00, 0x00, 0x00, 0xF8, 0x80, 0x80, 0xF0, 0x80,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x0047 (,G‘): [width=8, offset= 0x023E (574) ] */
  0x00, 0x00, 0x00, 0x38, 0x44, 0x82, 0x80, 0x8E,
  0x82, 0x44, 0x38, 0x00, 0x00, 0x00,

/* character 0x0048 (,H‘): [width=7, offset= 0x024C (588) ] */
  0x00, 0x00, 0x00, 0x84, 0x84, 0x84, 0xFC, 0x84,
  0x84, 0x84, 0x84, 0x00, 0x00, 0x00,

/* character 0x0049 (,I‘): [width=2, offset= 0x025A (602) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x004A (,J‘): [width=5, offset= 0x0268 (616) ] */
  0x00, 0x00, 0x00, 0x10, 0x10, 0x10, 0x10, 0x10,
  0x90, 0x90, 0x60, 0x00, 0x00, 0x00,

/* character 0x004B (,K‘): [width=7, offset= 0x0276 (630) ] */
  0x00, 0x00, 0x00, 0x84, 0x88, 0x90, 0xB0, 0xD0,
```

```
  0x88, 0x88, 0x84, 0x00, 0x00, 0x00,

/* character 0x004C (,L‘): [width=6, offset= 0x0284 (644) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x80, 0xF8, 0x00, 0x00, 0x00,

/* character 0x004D (,M‘): [width=8, offset= 0x0292 (658) ] */
  0x00, 0x00, 0x00, 0x82, 0xC6, 0xC6, 0xAA, 0xAA,
  0xAA, 0x92, 0x92, 0x00, 0x00, 0x00,

/* character 0x004E (,N‘): [width=7, offset= 0x02A0 (672) ] */
  0x00, 0x00, 0x00, 0x84, 0xC4, 0xA4, 0xA4, 0x94,
  0x94, 0x8C, 0x84, 0x00, 0x00, 0x00,

/* character 0x004F (,O‘): [width=8, offset= 0x02AE (686) ] */
  0x00, 0x00, 0x00, 0x38, 0x44, 0x82, 0x82, 0x82,
  0x82, 0x44, 0x38, 0x00, 0x00, 0x00,

/* character 0x0050 (,P‘): [width=6, offset= 0x02BC (700) ] */
  0x00, 0x00, 0x00, 0xF0, 0x88, 0x88, 0x88, 0xF0,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x0051 (,Q‘): [width=8, offset= 0x02CA (714) ] */
  0x00, 0x00, 0x00, 0x38, 0x44, 0x82, 0x82, 0x82,
  0x9A, 0x44, 0x3A, 0x00, 0x00, 0x00,

/* character 0x0052 (,R‘): [width=7, offset= 0x02D8 (728) ] */
  0x00, 0x00, 0x00, 0xF8, 0x84, 0x84, 0xF8, 0x90,
  0x88, 0x88, 0x84, 0x00, 0x00, 0x00,

/* character 0x0053 (,S‘): [width=7, offset= 0x02E6 (742) ] */
  0x00, 0x00, 0x00, 0x78, 0x84, 0x80, 0x60, 0x18,
  0x04, 0x84, 0x78, 0x00, 0x00, 0x00,

/* character 0x0054 (,T‘): [width=6, offset= 0x02F4 (756) ] */
  0x00, 0x00, 0x00, 0xF8, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x00, 0x00, 0x00,

/* character 0x0055 (,U‘): [width=7, offset= 0x0302 (770) ] */
  0x00, 0x00, 0x00, 0x84, 0x84, 0x84, 0x84, 0x84,
  0x84, 0x84, 0x78, 0x00, 0x00, 0x00,

/* character 0x0056 (,V‘): [width=8, offset= 0x0310 (784) ] */
  0x00, 0x00, 0x00, 0x82, 0x82, 0x44, 0x44, 0x28,
  0x28, 0x10, 0x10, 0x00, 0x00, 0x00,

/* character 0x0057 (,W‘): [width=11, offset= 0x031E (798) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x84, 0x20,
  0x8A, 0x20, 0x4A, 0x40, 0x4A, 0x40, 0x51, 0x40,
  0x51, 0x40, 0x20, 0x80, 0x20, 0x80, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,

/* character 0x0058 (,X‘): [width=7, offset= 0x033A (826) ] */
  0x00, 0x00, 0x00, 0x84, 0x48, 0x48, 0x30, 0x30,
  0x48, 0x48, 0x84, 0x00, 0x00, 0x00,
```

```
/* character 0x0059 (,Y‘): [width=8, offset= 0x0348 (840) ] */
  0x00, 0x00, 0x00, 0x82, 0x44, 0x44, 0x28, 0x10,
  0x10, 0x10, 0x10, 0x00, 0x00, 0x00,

/* character 0x005A (,Z‘): [width=7, offset= 0x0356 (854) ] */
  0x00, 0x00, 0x00, 0x7C, 0x08, 0x10, 0x10, 0x20,
  0x20, 0x40, 0xFC, 0x00, 0x00, 0x00,

/* character 0x005B (,[,): [width=3, offset= 0x0364 (868) ] */
  0x00, 0x00, 0x00, 0xC0, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x80, 0xC0, 0x00,

/* character 0x005C (,\‘): [width=3, offset= 0x0372 (882) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x40, 0x40, 0x40,
  0x40, 0x20, 0x20, 0x00, 0x00, 0x00,

/* character 0x005D (,]‘): [width=3, offset= 0x0380 (896) ] */
  0x00, 0x00, 0x00, 0xC0, 0x40, 0x40, 0x40, 0x40,
  0x40, 0x40, 0x40, 0x40, 0xC0, 0x00,

/* character 0x005E (,^‘): [width=5, offset= 0x038E (910) ] */
  0x00, 0x00, 0x00, 0x20, 0x50, 0x50, 0x88, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x005F (,_‘): [width=6, offset= 0x039C (924) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0xFC, 0x00,

/* character 0x0060 (,`‘): [width=4, offset= 0x03AA (938) ] */
  0x00, 0x00, 0x00, 0x80, 0x40, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x0061 (,a‘): [width=6, offset= 0x03B8 (952) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x88, 0x78,
  0x88, 0x98, 0x68, 0x00, 0x00, 0x00,

/* character 0x0062 (,b‘): [width=6, offset= 0x03C6 (966) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0xB0, 0xC8, 0x88,
  0x88, 0xC8, 0xB0, 0x00, 0x00, 0x00,

/* character 0x0063 (,c‘): [width=6, offset= 0x03D4 (980) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x88, 0x80,
  0x80, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0064 (,d‘): [width=6, offset= 0x03E2 (994) ] */
  0x00, 0x00, 0x00, 0x08, 0x08, 0x68, 0x98, 0x88,
  0x88, 0x98, 0x68, 0x00, 0x00, 0x00,

/* character 0x0065 (,e‘): [width=6, offset= 0x03F0 (1008) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x88, 0xF8,
  0x80, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0066 (,f‘): [width=4, offset= 0x03FE (1022) ] */
  0x00, 0x00, 0x00, 0x20, 0x40, 0xE0, 0x40, 0x40,
```

```
  0x40, 0x40, 0x40, 0x00, 0x00, 0x00,

/* character 0x0067 (,g‘): [width=6, offset= 0x040C (1036) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x68, 0x98, 0x88,
  0x88, 0x98, 0x68, 0x08, 0xF0, 0x00,

/* character 0x0068 (,h‘): [width=6, offset= 0x041A (1050) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0xB0, 0xC8, 0x88,
  0x88, 0x88, 0x88, 0x00, 0x00, 0x00,

/* character 0x0069 (,i‘): [width=2, offset= 0x0428 (1064) ] */
  0x00, 0x00, 0x00, 0x80, 0x00, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x006A (,j‘): [width=2, offset= 0x0436 (1078) ] */
  0x00, 0x00, 0x00, 0x80, 0x00, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x80, 0x00, 0x00,

/* character 0x006B (,k‘): [width=5, offset= 0x0444 (1092) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x90, 0xA0, 0xC0,
  0xA0, 0xA0, 0x90, 0x00, 0x00, 0x00,

/* character 0x006C (,l‘): [width=2, offset= 0x0452 (1106) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x006D (,m‘): [width=8, offset= 0x0460 (1120) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0xBC, 0xD2, 0x92,
  0x92, 0x92, 0x92, 0x00, 0x00, 0x00,

/* character 0x006E (,n‘): [width=6, offset= 0x046E (1134) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0x88, 0x88,
  0x88, 0x88, 0x88, 0x00, 0x00, 0x00,

/* character 0x006F (,o‘): [width=6, offset= 0x047C (1148) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x88, 0x88,
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0070 (,p‘): [width=6, offset= 0x048A (1162) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0xB0, 0xC8, 0x88,
  0x88, 0xC8, 0xB0, 0x80, 0x80, 0x00,

/* character 0x0071 (,q‘): [width=6, offset= 0x0498 (1176) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x68, 0x98, 0x88,
  0x88, 0x98, 0x68, 0x08, 0x08, 0x00,

/* character 0x0072 (,r‘): [width=4, offset= 0x04A6 (1190) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0xA0, 0xC0, 0x80,
  0x80, 0x80, 0x80, 0x00, 0x00, 0x00,

/* character 0x0073 (,s‘): [width=6, offset= 0x04B4 (1204) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x88, 0x60,
  0x10, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x0074 (,t‘): [width=3, offset= 0x04C2 (1218) ] */
```

```
  0x00, 0x00, 0x00, 0x80, 0x80, 0xC0, 0x80, 0x80,
  0x80, 0x80, 0xC0, 0x00, 0x00, 0x00,

/* character 0x0075 (,u‘): [width=6, offset= 0x04D0 (1232) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x88, 0x88,
  0x88, 0x98, 0x68, 0x00, 0x00, 0x00,

/* character 0x0076 (,v‘): [width=6, offset= 0x04DE (1246) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x88, 0x50,
  0x50, 0x20, 0x20, 0x00, 0x00, 0x00,

/* character 0x0077 (,w‘): [width=10, offset= 0x04EC (1260) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x88, 0x80, 0x94, 0x80, 0x55, 0x00,
  0x55, 0x00, 0x22, 0x00, 0x22, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,

/* character 0x0078 (,x‘): [width=6, offset= 0x0508 (1288) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x50, 0x20,
  0x20, 0x50, 0x88, 0x00, 0x00, 0x00,

/* character 0x0079 (,y‘): [width=6, offset= 0x0516 (1302) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x88, 0x50,
  0x50, 0x20, 0x20, 0x20, 0x40, 0x00,

/* character 0x007A (,z‘): [width=6, offset= 0x0524 (1316) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0xF8, 0x10, 0x20,
  0x20, 0x40, 0xF8, 0x00, 0x00, 0x00,

/* character 0x007B (,{,): [width=4, offset= 0x0532 (1330) ] */
  0x00, 0x00, 0x00, 0x20, 0x40, 0x40, 0x40, 0x80,
  0x40, 0x40, 0x40, 0x40, 0x20, 0x00,

/* character 0x007C (,|‘): [width=2, offset= 0x0540 (1344) ] */
  0x00, 0x00, 0x00, 0x80, 0x80, 0x80, 0x80, 0x80,
  0x80, 0x80, 0x80, 0x80, 0x80, 0x00,

/* character 0x007D (,}‘): [width=4, offset= 0x054E (1358) ] */
  0x00, 0x00, 0x00, 0x40, 0x20, 0x20, 0x20, 0x10,
  0x20, 0x20, 0x20, 0x20, 0x40, 0x00,

/* character 0x007E (,~‘): [width=6, offset= 0x055C (1372) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xE8, 0xB0,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x007F (,□‘): [width=8, offset= 0x056A (1386) ] */
  0x00, 0x00, 0x00, 0xE0, 0xA0, 0xA0, 0xA0, 0xA0,
  0xA0, 0xA0, 0xE0, 0x00, 0x00, 0x00,

/* character 0x00A2 (,¢‘): [width=6, offset= 0x0578 (1400) ] */
  0x00, 0x00, 0x00, 0x10, 0x10, 0x70, 0xA8, 0xA0,
  0xA0, 0xA8, 0x70, 0x40, 0x40, 0x00,

/* character 0x00A3 (,£‘): [width=6, offset= 0x0586 (1414) ] */
  0x00, 0x00, 0x00, 0x30, 0x48, 0x40, 0x40, 0xE0,
```

```
  0x40, 0x60, 0x98, 0x00, 0x00, 0x00,

/* character 0x00A5 (,¥‘): [width=6, offset= 0x0594 (1428) ] */
  0x00, 0x00, 0x00, 0x88, 0x88, 0x50, 0x50, 0xF8,
  0x20, 0xF8, 0x20, 0x00, 0x00, 0x00,

/* character 0x00A7 (,§‘): [width=6, offset= 0x05A2 (1442) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x40, 0xE0, 0x90,
  0x48, 0x28, 0x10, 0x88, 0x70, 0x00,

/* character 0x00A9 (,©‘): [width=8, offset= 0x05B0 (1456) ] */
  0x00, 0x00, 0x00, 0x3C, 0x42, 0x9D, 0xA1, 0xA5,
  0x99, 0x42, 0x3C, 0x00, 0x00, 0x00,

/* character 0x00AC (,¬‘): [width=6, offset= 0x05BE (1470) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF8, 0x08,
  0x08, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00AE (,®‘): [width=8, offset= 0x05CC (1484) ] */
  0x00, 0x00, 0x00, 0x3C, 0x42, 0xB9, 0xA5, 0xB9,
  0xA5, 0x42, 0x3C, 0x00, 0x00, 0x00,

/* character 0x00B0 (,°‘): [width=4, offset= 0x05DA (1498) ] */
  0x00, 0x00, 0x00, 0xE0, 0xA0, 0xE0, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00B1 (,±‘): [width=6, offset= 0x05E8 (1512) ] */
  0x00, 0x00, 0x00, 0x00, 0x20, 0x20, 0xF8, 0x20,
  0x20, 0x00, 0xF8, 0x00, 0x00, 0x00,

/* character 0x00B2 (,²‘): [width=4, offset= 0x05F6 (1526) ] */
  0x00, 0x00, 0x00, 0xE0, 0x20, 0x40, 0xE0, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00B3 (,³‘): [width=4, offset= 0x0604 (1540) ] */
  0x00, 0x00, 0x00, 0xE0, 0x40, 0x20, 0xE0, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00B5 (,µ‘): [width=6, offset= 0x0612 (1554) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x88, 0x88,
  0x88, 0x88, 0xF8, 0x80, 0x80, 0x00,

/* character 0x00B9 (,¹‘): [width=4, offset= 0x0620 (1568) ] */
  0x00, 0x00, 0x00, 0x20, 0x60, 0x20, 0x20, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00BA (,º‘): [width=5, offset= 0x062E (1582) ] */
  0x00, 0x00, 0x00, 0x60, 0x90, 0x90, 0x60, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

/* character 0x00C4 (,Ä‘): [width=8, offset= 0x063C (1596) ] */
  0x00, 0x28, 0x00, 0x10, 0x28, 0x28, 0x28, 0x44,
  0x7C, 0x82, 0x82, 0x00, 0x00, 0x00,

/* character 0x00D6 (,Ö‘): [width=8, offset= 0x064A (1610) ] */
```

```
  0x00, 0x28, 0x00, 0x38, 0x44, 0x82, 0x82, 0x82,
  0x82, 0x44, 0x38, 0x00, 0x00, 0x00,

/* character 0x00DC (‚Ü'): [width=7, offset= 0x0658 (1624) ] */
  0x00, 0x28, 0x00, 0x84, 0x84, 0x84, 0x84, 0x84,
  0x84, 0x84, 0x78, 0x00, 0x00, 0x00,

/* character 0x00DF (‚ß'): [width=7, offset= 0x0666 (1638) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x88, 0x90, 0x98,
  0x84, 0xA4, 0x98, 0x00, 0x00, 0x00,

/* character 0x00E4 (‚ä'): [width=6, offset= 0x0674 (1652) ] */
  0x00, 0x00, 0x00, 0x50, 0x00, 0x70, 0x88, 0x78,
  0x88, 0x98, 0x68, 0x00, 0x00, 0x00,

/* character 0x00F6 (‚ö'): [width=6, offset= 0x0682 (1666) ] */
  0x00, 0x00, 0x00, 0x50, 0x00, 0x70, 0x88, 0x88,
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x00FC (‚ü'): [width=6, offset= 0x0690 (1680) ] */
  0x00, 0x00, 0x00, 0x50, 0x00, 0x88, 0x88, 0x88,
  0x88, 0x98, 0x68, 0x00, 0x00, 0x00,

/* character 0x0394 (‚?'): [width=8, offset= 0x069E (1694) ] */
  0x00, 0x00, 0x00, 0x10, 0x28, 0x28, 0x44, 0x44,
  0x44, 0x82, 0xFE, 0x00, 0x00, 0x00,

/* character 0x039B (‚?'): [width=8, offset= 0x06AC (1708) ] */
  0x00, 0x00, 0x00, 0x10, 0x28, 0x28, 0x44, 0x44,
  0x44, 0x82, 0x82, 0x00, 0x00, 0x00,

/* character 0x03A9 (‚O'): [width=8, offset= 0x06BA (1722) ] */
  0x00, 0x00, 0x00, 0x38, 0x44, 0x82, 0x82, 0x82,
  0x82, 0x44, 0xC6, 0x00, 0x00, 0x00,

/* character 0x03B5 (‚e'): [width=5, offset= 0x06C8 (1736) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x70, 0x80, 0x60,
  0x80, 0x80, 0x70, 0x00, 0x00, 0x00,

/* character 0x03B8 (‚?'): [width=6, offset= 0x06D6 (1750) ] */
  0x00, 0x00, 0x00, 0x70, 0x88, 0x88, 0xF8, 0x88,
  0x88, 0x88, 0x70, 0x00, 0x00, 0x00,

/* character 0x03BC (‚µ'): [width=6, offset= 0x06E4 (1764) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x88, 0x88, 0x88,
  0x88, 0x88, 0xF8, 0x80, 0x80, 0x00,

/* character 0x03C0 (‚p'): [width=9, offset= 0x06F2 (1778) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0xFF, 0x00, 0x24, 0x00, 0x24, 0x00,
  0x24, 0x00, 0x24, 0x00, 0x24, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00,

/* character 0x263A (‚?'): [width=11, offset= 0x070E (1806) ] */
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
  0x00, 0x00, 0x0F, 0x00, 0x30, 0x80, 0x28, 0x80,
  0x20, 0x80, 0x28, 0x80, 0x17, 0x00, 0x0E, 0x00,
  0x00, 0x00, 0x00, 0x00,

/* character 0x2640 (‚?‘): [width=8, offset= 0x072A (1834) ] */
  0x00, 0x00, 0x00, 0x3C, 0x42, 0x42, 0x42, 0x44,
  0x38, 0x10, 0x10, 0x3C, 0x00, 0x00,

/* character 0x2642 (‚?‘): [width=8, offset= 0x0738 (1848) ] */
  0x00, 0x00, 0x02, 0x0E, 0x06, 0x0A, 0x08, 0x78,
  0x44, 0x44, 0x44, 0x78, 0x00, 0x00,

/* character 0x266B (‚?‘): [width=8, offset= 0x0746 (1862) ] */
  0x00, 0x00, 0x00, 0x06, 0x1E, 0x22, 0x22, 0x22,
  0x26, 0x6E, 0xE4, 0x60, 0x00, 0x00,

};

/*****************************************************************************
******
   Offset table provides the starting offset of each character in the data table.

   If you can calculate the offsets by yourself, undefine USE_OFFSET_TABLE.

   Offset table provides the starting offset of each character in the data table.

   To get the starting offset of character ‚A‘, you can use the following expression:

     const USHORT index = GetIndex(‚A‘);
     const ULONG offset = offset_table[index];

*****************************************************************************
*****/
const unsigned int fontArial14h_offset_tablep[] PROGMEM =
{
/*  offset      offsetHex  -  char    hexcode    decimal */
/*  =======     =========  -  ====    =======    ======= */
  0,         /*         0  -         0020       32      */
  14,        /*         E  -  !      0021       33      */
  28,        /*        1C  -  „      0022       34      */
  42,        /*        2A  -  #      0023       35      */
  56,        /*        38  -  $      0024       36      */
  70,        /*        46  -  %      0025       37      */
  98,        /*        62  -  &      0026       38      */
  112,       /*        70  -  ‚      0027       39      */
  126,       /*        7E  -  (      0028       40      */
  140,       /*        8C  -  )      0029       41      */
  154,       /*        9A  -  *      002A       42      */
  168,       /*        A8  -  +      002B       43      */
  182,       /*        B6  -  ‚      002C       44      */
  196,       /*        C4  -  -      002D       45      */
  210,       /*        D2  -  .      002E       46      */
  224,       /*        E0  -  /      002F       47      */
  238,       /*        EE  -  0      0030       48      */
  252,       /*        FC  -  1      0031       49      */
```

```
266,    /*    10A  -  2    0032    50    */
280,    /*    118  -  3    0033    51    */
294,    /*    126  -  4    0034    52    */
308,    /*    134  -  5    0035    53    */
322,    /*    142  -  6    0036    54    */
336,    /*    150  -  7    0037    55    */
350,    /*    15E  -  8    0038    56    */
364,    /*    16C  -  9    0039    57    */
378,    /*    17A  -  :    003A    58    */
392,    /*    188  -  ;    003B    59    */
406,    /*    196  -  <    003C    60    */
420,    /*    1A4  -  =    003D    61    */
434,    /*    1B2  -  >    003E    62    */
448,    /*    1C0  -  ?    003F    63    */
462,    /*    1CE  -  @    0040    64    */
490,    /*    1EA  -  A    0041    65    */
504,    /*    1F8  -  B    0042    66    */
518,    /*    206  -  C    0043    67    */
532,    /*    214  -  D    0044    68    */
546,    /*    222  -  E    0045    69    */
560,    /*    230  -  F    0046    70    */
574,    /*    23E  -  G    0047    71    */
588,    /*    24C  -  H    0048    72    */
602,    /*    25A  -  I    0049    73    */
616,    /*    268  -  J    004A    74    */
630,    /*    276  -  K    004B    75    */
644,    /*    284  -  L    004C    76    */
658,    /*    292  -  M    004D    77    */
672,    /*    2A0  -  N    004E    78    */
686,    /*    2AE  -  O    004F    79    */
700,    /*    2BC  -  P    0050    80    */
714,    /*    2CA  -  Q    0051    81    */
728,    /*    2D8  -  R    0052    82    */
742,    /*    2E6  -  S    0053    83    */
756,    /*    2F4  -  T    0054    84    */
770,    /*    302  -  U    0055    85    */
784,    /*    310  -  V    0056    86    */
798,    /*    31E  -  W    0057    87    */
826,    /*    33A  -  X    0058    88    */
840,    /*    348  -  Y    0059    89    */
854,    /*    356  -  Z    005A    90    */
868,    /*    364  -  [    005B    91    */
882,    /*    372  -  \    005C    92    */
896,    /*    380  -  ]    005D    93    */
910,    /*    38E  -  ^    005E    94    */
924,    /*    39C  -  _    005F    95    */
938,    /*    3AA  -  `    0060    96    */
952,    /*    3B8  -  a    0061    97    */
966,    /*    3C6  -  b    0062    98    */
980,    /*    3D4  -  c    0063    99    */
994,    /*    3E2  -  d    0064    100   */
1008,   /*    3F0  -  e    0065    101   */
1022,   /*    3FE  -  f    0066    102   */
1036,   /*    40C  -  g    0067    103   */
1050,   /*    41A  -  h    0068    104   */
```

```
1064,     /*      428  -  i       0069       105    */
1078,     /*      436  -  j       006A       106    */
1092,     /*      444  -  k       006B       107    */
1106,     /*      452  -  l       006C       108    */
1120,     /*      460  -  m       006D       109    */
1134,     /*      46E  -  n       006E       110    */
1148,     /*      47C  -  o       006F       111    */
1162,     /*      48A  -  p       0070       112    */
1176,     /*      498  -  q       0071       113    */
1190,     /*      4A6  -  r       0072       114    */
1204,     /*      4B4  -  s       0073       115    */
1218,     /*      4C2  -  t       0074       116    */
1232,     /*      4D0  -  u       0075       117    */
1246,     /*      4DE  -  v       0076       118    */
1260,     /*      4EC  -  w       0077       119    */
1288,     /*      508  -  x       0078       120    */
1302,     /*      516  -  y       0079       121    */
1316,     /*      524  -  z       007A       122    */
1330,     /*      532  -  {       007B       123    */
1344,     /*      540  -  |       007C       124    */
1358,     /*      54E  -  }       007D       125    */
1372,     /*      55C  -  ~       007E       126    */
1386,     /*      56A  -          007F       127    */
1400,     /*      578  -  ¢       00A2       162    */
1414,     /*      586  -  £       00A3       163    */
1428,     /*      594  -  ¥       00A5       165    */
1442,     /*      5A2  -  §       00A7       167    */
1456,     /*      5B0  -  ©       00A9       169    */
1470,     /*      5BE  -  ¬       00AC       172    */
1484,     /*      5CC  -  ®       00AE       174    */
1498,     /*      5DA  -  °       00B0       176    */
1512,     /*      5E8  -  ±       00B1       177    */
1526,     /*      5F6  -  ²       00B2       178    */
1540,     /*      604  -  ³       00B3       179    */
1554,     /*      612  -  µ       00B5       181    */
1568,     /*      620  -  ¹       00B9       185    */
1582,     /*      62E  -  º       00BA       186    */
1596,     /*      63C  -  Ä       00C4       196    */
1610,     /*      64A  -  Ö       00D6       214    */
1624,     /*      658  -  Ü       00DC       220    */
1638,     /*      666  -  ß       00DF       223    */
1652,     /*      674  -  ä       00E4       228    */
1666,     /*      682  -  ö       00F6       246    */
1680,     /*      690  -  ü       00FC       252    */
1694,     /*      69E  -  ?       0394       916    */
1708,     /*      6AC  -  ?       039B       923    */
1722,     /*      6BA  -  O       03A9       937    */
1736,     /*      6C8  -  e       03B5       949    */
1750,     /*      6D6  -  ?       03B8       952    */
1764,     /*      6E4  -  µ       03BC       956    */
1778,     /*      6F2  -  p       03C0       960    */
1806,     /*      70E  -  ?       263A       9786   */
1834,     /*      72A  -  ?       2640       9792   */
1848,     /*      738  -  ?       2642       9794   */
1862,     /*      746  -  ?       266B       9835   */
```

```
  1876,      /*       754  -  extra address: the end of the last character‘s imagebits
data */
};

/*****************************************************************************
******
   Width table provides the width of each character. It‘s useful for proportional
font.
   For monospaced font, the widths of all character are the same.

   Generally speaking, the width table is not needed for monospaced font. you can get
   the width from the font header.

   If you do not need the width table, undefine USE_WIDTH_TABLE.

   To get the width of character ‚A‘, you can use the following expression:

     const USHORT index = GetIndex(‚A‘);
     const USHORT width = width_table[index];

*****************************************************************************
*****/
const unsigned short fontArial14h_width_tablep[] PROGMEM =
{
/*  width    char    hexcode    decimal */
/*  =====    ====    =======    ======= */
  3,       /*          0020        32   */
  2,       /*  !       0021        33   */
  4,       /*  „       0022        34   */
  6,       /*  #       0023        35   */
  6,       /*  $       0024        36   */
  10,      /*  %       0025        37   */
  7,       /*  &       0026        38   */
  2,       /*  ,       0027        39   */
  4,       /*  (       0028        40   */
  4,       /*  )       0029        41   */
  4,       /*  *       002A        42   */
  6,       /*  +       002B        43   */
  3,       /*  ,       002C        44   */
  4,       /*  -       002D        45   */
  3,       /*  .       002E        46   */
  3,       /*  /       002F        47   */
  6,       /*  0       0030        48   */
  6,       /*  1       0031        49   */
  6,       /*  2       0032        50   */
  6,       /*  3       0033        51   */
  6,       /*  4       0034        52   */
  6,       /*  5       0035        53   */
  6,       /*  6       0036        54   */
  6,       /*  7       0037        55   */
  6,       /*  8       0038        56   */
  6,       /*  9       0039        57   */
  3,       /*  :       003A        58   */
  3,       /*  ;       003B        59   */
```

```
  6,      /*  <       003C       60   */
  6,      /*  =       003D       61   */
  6,      /*  >       003E       62   */
  6,      /*  ?       003F       63   */
 11,      /*  @       0040       64   */
  8,      /*  A       0041       65   */
  7,      /*  B       0042       66   */
  7,      /*  C       0043       67   */
  7,      /*  D       0044       68   */
  6,      /*  E       0045       69   */
  6,      /*  F       0046       70   */
  8,      /*  G       0047       71   */
  7,      /*  H       0048       72   */
  2,      /*  I       0049       73   */
  5,      /*  J       004A       74   */
  7,      /*  K       004B       75   */
  6,      /*  L       004C       76   */
  8,      /*  M       004D       77   */
  7,      /*  N       004E       78   */
  8,      /*  O       004F       79   */
  6,      /*  P       0050       80   */
  8,      /*  Q       0051       81   */
  7,      /*  R       0052       82   */
  7,      /*  S       0053       83   */
  6,      /*  T       0054       84   */
  7,      /*  U       0055       85   */
  8,      /*  V       0056       86   */
 11,      /*  W       0057       87   */
  7,      /*  X       0058       88   */
  8,      /*  Y       0059       89   */
  7,      /*  Z       005A       90   */
  3,      /*  [       005B       91   */
  3,      /*  \       005C       92   */
  3,      /*  ]       005D       93   */
  5,      /*  ^       005E       94   */
  6,      /*  _       005F       95   */
  4,      /*  `       0060       96   */
  6,      /*  a       0061       97   */
  6,      /*  b       0062       98   */
  6,      /*  c       0063       99   */
  6,      /*  d       0064      100   */
  6,      /*  e       0065      101   */
  4,      /*  f       0066      102   */
  6,      /*  g       0067      103   */
  6,      /*  h       0068      104   */
  2,      /*  i       0069      105   */
  2,      /*  j       006A      106   */
  5,      /*  k       006B      107   */
  2,      /*  l       006C      108   */
  8,      /*  m       006D      109   */
  6,      /*  n       006E      110   */
  6,      /*  o       006F      111   */
  6,      /*  p       0070      112   */
  6,      /*  q       0071      113   */
  4,      /*  r       0072      114   */
```

```
  6,       /*  s       0073       115   */
  3,       /*  t       0074       116   */
  6,       /*  u       0075       117   */
  6,       /*  v       0076       118   */
  10,      /*  w       0077       119   */
  6,       /*  x       0078       120   */
  6,       /*  y       0079       121   */
  6,       /*  z       007A       122   */
  4,       /*  {       007B       123   */
  2,       /*  |       007C       124   */
  4,       /*  }       007D       125   */
  6,       /*  ~       007E       126   */
  8,       /*          007F       127   */
  6,       /*  ¢       00A2       162   */
  6,       /*  £       00A3       163   */
  6,       /*  ¥       00A5       165   */
  6,       /*  §       00A7       167   */
  8,       /*  ©       00A9       169   */
  6,       /*  ¬       00AC       172   */
  8,       /*  ®       00AE       174   */
  4,       /*  °       00B0       176   */
  6,       /*  ±       00B1       177   */
  4,       /*  ²       00B2       178   */
  4,       /*  ³       00B3       179   */
  6,       /*  µ       00B5       181   */
  4,       /*  ¹       00B9       185   */
  5,       /*  º       00BA       186   */
  8,       /*  Ä       00C4       196   */
  8,       /*  Ö       00D6       214   */
  7,       /*  Ü       00DC       220   */
  7,       /*  ß       00DF       223   */
  6,       /*  ä       00E4       228   */
  6,       /*  ö       00F6       246   */
  6,       /*  ü       00FC       252   */
  8,       /*  ?       0394       916   */
  8,       /*  ?       039B       923   */
  8,       /*  O       03A9       937   */
  5,       /*  e       03B5       949   */
  6,       /*  ?       03B8       952   */
  6,       /*  µ       03BC       956   */
  9,       /*  p       03C0       960   */
  11,      /*  ?       263A      9786   */
  8,       /*  ?       2640      9792   */
  8,       /*  ?       2642      9794   */
  8,       /*  ?       266B      9835   */
};


#define LCDWIDTH 64
#define LCDHEIGHT 48
#define LCDSIZE (LCDWIDTH * LCDHEIGHT)  // In PIXEL !!! muss durch 32 teilbar sein fuer
longs.
unsigned char lcdbuffer[LCDSIZE/8];
```

```
void i2c_oled_write_command(unsigned char i2cbaseadr, unsigned char cmdvalue)
{

  Wire.beginTransmission(i2cbaseadr);
  Wire.write(0x80); // 1000 0000 co=1 DC =0  ist commando oder parameter fuer letztes
kommando
  Wire.write(cmdvalue);
  Wire.endTransmission();
}



void i2c_oled_entire_onoff(unsigned char i2cbaseadr, unsigned char onoff)
{
  if (onoff == 1) { // anschalten alle PIXEL EIN
    i2c_oled_write_command(i2cbaseadr,0xA5);
  } else { // Daten aus dem RAM
    i2c_oled_write_command(i2cbaseadr,0xA4);
  }
}


void i2c_oled_display_onoff(unsigned char i2cbaseadr, unsigned char onoff)
{
  if (onoff == 1) { // anschalten display
    i2c_oled_write_command(i2cbaseadr,0xAF);
  } else { // ausschalten display
    i2c_oled_write_command(i2cbaseadr,0xAE);
  }
}

void i2c_oled_setbrightness(unsigned char i2cbaseadr, unsigned char wert)
{
  i2c_oled_write_command(i2cbaseadr,0x81); // cmd fuer brightness
  i2c_oled_write_command(i2cbaseadr,wert); // 2. wert dann brightness
}


void i2c_oled_inverse_onoff(unsigned char i2cbaseadr, unsigned char onoff)
{
  if (onoff == 1) { // inverse
    i2c_oled_write_command(i2cbaseadr,0xA7);
  } else { // ausschalten display
    i2c_oled_write_command(i2cbaseadr,0xA6);
  }
}

// INIT 64x48 display:


void i2c_oled_initall(unsigned char i2cbaseadr)
{
  // i2c_oled_display_onff(i2cbaseadr,0); // ERST mal aus.
  i2c_oled_write_command(i2cbaseadr,0xAE);   // display off
  //
```

```
  i2c_oled_write_command(i2cbaseadr,0x00); /*set lower column address*/
 i2c_oled_write_command(i2cbaseadr,0x12); /*set higher column address*/
 i2c_oled_write_command(i2cbaseadr,0x40); /*set display start line*/
 i2c_oled_write_command(i2cbaseadr,0xB0); /*set page address*/
 i2c_oled_write_command(i2cbaseadr,0x81); /*contract control*/
 i2c_oled_write_command(i2cbaseadr,0xff); /*128*/
 i2c_oled_write_command(i2cbaseadr,0xA1); /*set segment remap*/
 i2c_oled_write_command(i2cbaseadr,0xA6); /*normal / reverse*/
 i2c_oled_write_command(i2cbaseadr,0xA8); /*multiplex ratio*/
 i2c_oled_write_command(i2cbaseadr,0x2F); /*duty = 1/48*/
 i2c_oled_write_command(i2cbaseadr,0xC8); /*Com scan direction*/
 i2c_oled_write_command(i2cbaseadr,0xD3); /*set display offset*/
 i2c_oled_write_command(i2cbaseadr,0x00);
 i2c_oled_write_command(i2cbaseadr,0xD5); /*set osc division*/
 i2c_oled_write_command(i2cbaseadr,0x80);
 i2c_oled_write_command(i2cbaseadr,0xD9); /*set pre-charge period*/
 i2c_oled_write_command(i2cbaseadr,0x21);
 i2c_oled_write_command(i2cbaseadr,0xDA); /*set COM pins*/
 i2c_oled_write_command(i2cbaseadr,0x12);
 i2c_oled_write_command(i2cbaseadr,0xdb); /*set vcomh*/
 i2c_oled_write_command(i2cbaseadr,0x40);
 i2c_oled_write_command(i2cbaseadr,0x8d); /*set charge pump enable*/
 i2c_oled_write_command(i2cbaseadr,0x14);
 i2c_oled_write_command(i2cbaseadr,0xAF);   // enable display
  //
}

void i2c_oled_initalllarge(unsigned char i2cbaseadr)
{
  // i2c_oled_display_onff(i2cbaseadr,0); // ERST mal aus.
  i2c_oled_write_command(i2cbaseadr,0xd5); // divide ratio osc freq
  i2c_oled_write_command(i2cbaseadr,0x80); // f0 flackert werniger als 80
  //
  i2c_oled_write_command(i2cbaseadr,0xa8); // multiplex ratio mode:63
  i2c_oled_write_command(i2cbaseadr,0x3f);
  //
  i2c_oled_write_command(i2cbaseadr,0xd3); // set display offset
  i2c_oled_write_command(i2cbaseadr,0); // value 0
  //
  i2c_oled_write_command(i2cbaseadr,0x40); // set display startline (D5..D0 = line)
  //
  i2c_oled_write_command(i2cbaseadr,0x8D); // charge pump on + 14 + af
  i2c_oled_write_command(i2cbaseadr,0x14); // Enable charge pump
  //
  i2c_oled_write_command(i2cbaseadr,0xA1); // segment remap hor richtung a1 nach links
nach rechts a0 rechts nach links
  //
  i2c_oled_write_command(i2cbaseadr,0xC8); // c8 von oben nach unten c0 von unten nach
oben
  //
  i2c_oled_write_command(i2cbaseadr,0xda); // common pads hardware: alternative
  //
  i2c_oled_write_command(i2cbaseadr,0x12); // 12: OK, 32: dasselbe, 02: Datenmüll
  //
  i2c_oled_write_command(i2cbaseadr,0x81); // set brightness
```

```
  i2c_oled_write_command(i2cbaseadr,0xff); // 0..ff
  //
  i2c_oled_write_command(i2cbaseadr,0xD9);  // set precharge period
  i2c_oled_write_command(i2cbaseadr,0xF1);   // F1 flacher staerker, 11//22 weniger
stark
  //
  i2c_oled_write_command(i2cbaseadr,0xDB);   // COM Deselect level
  i2c_oled_write_command(i2cbaseadr,0x40);   // 0.83*VCC laut datenblatt von legendary
  //
  i2c_oled_write_command(i2cbaseadr,0xA4);    // Display on alle pixel ein
  //
  i2c_oled_write_command(i2cbaseadr,0xA6);   // set normal display a6=normal
a7=invertiert
  //
  i2c_oled_write_command(i2cbaseadr,0xAF);   // enable display
  //
}




// zeilen 0..7
unsigned char i2c_oled_write_top(unsigned char i2cbaseadr, int zeile,
                    int bytes,  unsigned char barray[],signed int sh1106padding)
{
  int i;


  i2c_oled_write_command(i2cbaseadr,0x20); // page address mode
  i2c_oled_write_command(i2cbaseadr,0x02); // page address mode
  // **
  //
  i2c_oled_write_command(i2cbaseadr,0xb0+(zeile & 7)); // B0..B7
  //
  i2c_oled_write_command(i2cbaseadr,0x00); // $00 lower nibble col + x3x2x1x0
  i2c_oled_write_command(i2cbaseadr,0x10); // $10 high  nibble col + x3x2x1x0
  //
  // DANN Daten senden Umschalten auf Daten Modus
  //

  // ACHTUNG bei Arduino anscheinend max 32 bytes data
  // Wirelib mxa 32 bytes buffer size !!
  // daher zerlegen noetig !!

  // AChtung col 1..64-seg95..seg32  row 1..48 com32..com55

  // zahl bytes / 8 teilbar !!
  int j=0;
  int k =0;

// MAX LIMIT daher zwei schleifen bei Arduino

    Wire.beginTransmission(i2cbaseadr);
```

```
    Wire.write(0x40); // 0100 0000 co=0 DC =1  ist data follow no cmd repeat
    for (i=0; i<16; i++) {
       Wire.write(0);
    }
    Wire.endTransmission();

    Wire.beginTransmission(i2cbaseadr);
    Wire.write(0x40); // 0100 0000 co=0 DC =1  ist data follow no cmd repeat
    for (i=0; i<16; i++) {
       Wire.write(0);
    }
    Wire.endTransmission();


  for (k=0; k<8; k++) {
    Wire.beginTransmission(i2cbaseadr);
    Wire.write(0x40); // 0100 0000 co=0 DC =1  ist data follow no cmd repeat
    for (i=0; i<bytes/8; i++) {
       Wire.write(barray[j++]);
    }
    Wire.endTransmission();
  }
  //
}

// ACHTUNG VERZAHNTes Display...
void disp_lcd_frombuffer() {
    // 132 fix an der Stelle !!
    // 64yx48
    // AChtung col 1..64-seg95..seg32  row 1..48 com32..com55
    // verzahnt com 0..23 nach 48..2 und com 32..55 nach row 47..1
    // offsets im transfer std 0..8
    i2c_oled_write_top(i2coledssd, 0, 64, &lcdbuffer[0], 0);
    i2c_oled_write_top(i2coledssd, 1, 64, &lcdbuffer[64], 0);
    i2c_oled_write_top(i2coledssd, 2, 64, &lcdbuffer[64*2], 0);
    i2c_oled_write_top(i2coledssd, 3, 64, &lcdbuffer[64*3], 0);
    i2c_oled_write_top(i2coledssd, 4, 64, &lcdbuffer[64*4], 0);
    i2c_oled_write_top(i2coledssd, 5, 64, &lcdbuffer[64*5], 0);

}

// Buffer loeschen mit farbe
// 0 oder ff
#define COLOR_BLACK 0
#define COLOR_WHITE 1      // sonderfall

void disp_buffer_clear(unsigned short data) {
  unsigned char  *ptr = (unsigned char*)&lcdbuffer[0];
  unsigned char datal = 0;
  if (data >0) datal = 0xffffffff; // alle an dann beim loeschen
  int i = 0;
  for (i = 0; i<LCDSIZE/8; i++) {
    *ptr++ = datal;
  }
}
```

```
// x,y
// dabei row col organisation
// 132 x 64 pixel dabei 8 pixel in y-richtung mit 0 beginnend auf einem byte
// x horizontal y vertikal

// 64 x 48 pixel !!
//
void disp_setpixel(int x, int y, unsigned short col1) {
    // COL =0 dunke =1hell
  // 1 pixel setzen (little endian)
  // col1 =0 hell =1 dunkel
  // Tabelle 8 bytes per zeile
  // eintrag 0 ist rechte pixelgruppe
  // y=0 ist unten links aber physicalisch von oben nach unten (row 7 bit 7 -> y=0)

  unsigned char  *dest;  // zielpointer lcdbuffer
  int yoff = 0;
  int ymod = 0;
  if (x < 0) return;  // CLIP
  if (x >= LCDWIDTH) return;  // CLIP
  if (y < 0) return;
  if (y >= LCDHEIGHT) return;
 #ifdef XXXX
  // SPEZIELL x-achse 0..63 aber y-achse abh von gerade oder ungerader zeile
  if ((y & 1) == 0) {
    // Gerade zeilen
    ymod = y & 0x7; // bitposition 0..7
    yoff = y >> 3; // offset der rows zeilen 0..7 (y>0 !!) bit 0..2 weg


  } else {
    // ungerade zeilen

  }
 #endif

  // berechnen row col und byte pos.
  // y = 63 - y; // damit y=0 gedreht y=0 link oben !!
  ymod = y & 0x7; // bitposition 0..7
  yoff = y >> 3; // offset der rows zeilen 0..7 (y>0 !!) bit 0..2 weg

  dest = &lcdbuffer[yoff * LCDWIDTH + x]; // DAMIT Byte definiert
  // nun bit 0..7 moeglichs effizienz  austauschen
  if (col1 == 0) { // bit loeschen
   *dest &= ~(1<<ymod);
  } else { // bit setzen
   *dest |= (1<<ymod);
  }
  //
}

unsigned short disp_setchar(int x, int y,  unsigned char chidx1, unsigned short color)
{ // breite als ergebnis fuer propschrift
  unsigned char chidx;
```

```
  chidx = chidx1;
  if (chidx <= 0x20) chidx = 0x20;
  chidx -= 0x20; // erster index
  unsigned short w = pgm_read_word_near(fontArial14h_width_tablep+chidx);
  unsigned long offset = pgm_read_word_near(fontArial14h_offset_tablep+chidx);
  int maxh = 14; // anzahl der zeilen
  int i = 0, v1 = 0, v2 = 0;
  int b1;
  //

  // breite <= 8 pixel dann ein byte >=8 dann zwei bytes (max width = 16 pixel)
  if (w <= 8) { // 1 byte max
    for (i = 0; i<14; i++) { // alle zeilen abarbeiten
      v1 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i); // 1 byte pro font
durchlaufen
      // alle bits durchlaufen
      for (b1 = 0; b1<w; b1++) {
        if (v1 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + b1, y + i, color);
        }
      } // alle bits durchlaufen
    }
  } else { // 2 byte zeichen
    for (i = 0; i<14; i++) { // alle zeilen abarbeiten
      v1 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i * 2); // 1 byte pro
font durchlaufen
      v2 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i * 2 + 1); // 1 byte
pro font durchlaufen
      // alle bits durchlaufen
      for (b1 = 0; b1<8; b1++) { // erste haelfte first
        if (v1 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + b1, y + i, color);
        }
      } // alle bits durchlaufen
      for (b1 = 0; b1<w; b1++) { // restl bits
        if (v2 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + 8 + b1, y + i, color);
        }
      } // alle bits durchlaufen
    }
  }
  return(w);
}


int disp_print_xy_lcd(int x, int y, unsigned char *text, unsigned short color, int
chset) { // 0=14 1=27
  int x1;
  x1 = x;
  if (text == 0) return(x);

    while (*text != 0) {
      x1 = x1 + disp_setchar(x1, y,  *text++, color);
    }
```

```
  return(x1);
}

// Linien Zeichnen

void  disp_line_lcd(int x0, int y0, int x1, int y1, unsigned short col) {
  // disp_setpixel(x+b1,y+i,color);

  int dx, dy, sx, sy, err, e2;
  dx = (x1 - x0);
  if (dx < 0) dx = -dx;
  dy = (y1 - y0);
  if (dy < 0) dy = -dy;
  if (x0 < x1) {
    sx = 1;
  } else {
    sx = -1;
  }
  if (y0 < y1) {
    sy = 1;
  } else {
    sy = -1;
  }
  //
  err = dx - dy;
  do {
    disp_setpixel(x0, y0, col);
    if ((x0 == x1) && (y0 == y1)) return;
    e2 = 2 * err;
    if (e2 > -dy) {
      err = err - dy;
      x0 = x0 + sx;
    }
    if (e2 < dx) {
      err = err + dx;
      y0 = y0 + sy;
    }
  } while (1==1);
}

// Rechteck Zeichnen



void disp_rect_lcd(int x1, int y1, int x2, int y2, unsigned short col) {
  // disp_setpixel(x+b1,y+i,color);
  disp_line_lcd(x1, y1, x2, y1, col);
  disp_line_lcd(x1, y2, x2, y2, col);
  disp_line_lcd(x1, y1, x1, y2, col);
  disp_line_lcd(x2, y1, x2, y2, col);
}

// filled rechteck zeichnen optionaler rahmen
//
void disp_filledrect_lcd(int x1, int y1, int x2, int y2, unsigned short col) {
```

```
  // disp_setpixel(x+b1,y+i,color);
  // schnelles fill
  //
  // x1,y1 und x2,y2 swappen ggf.
  unsigned short  *dest;    // zielpointer lcdbuffer mit 8 bytes per row !! (little
endian codiert)

  int x, y;
  if (x1 > x2) {
    x = x1;
    x1 = x2;
    x2 = x;
  }
  if (y1 > y2) {
    y = y1;
    y1 = y2;
    y2 = y;
  }
  if (x1 < 0) x1 = 0;  // CLIP
  if (x1 >= LCDWIDTH) x1 = LCDWIDTH - 1;  // CLIP
  if (y1< 0) y1 = 0;
  if (y1 >= LCDHEIGHT) y1 = LCDHEIGHT - 1;  // CLIP;
  // ACHTUNG rechnen row
  if (x2 < 0) x2 = 0;  // CLIP
  if (x2 >= LCDWIDTH) x2 = LCDWIDTH - 1;  // CLIP
  if (y2< 0) y2 = 0;
  if (y2 >= LCDHEIGHT) y2 = LCDHEIGHT - 1;  // CLIP;

  // noch nicht ganz effizient fuer sonderfaelle
  // koennte man in bytes zusammenfassen wenn eine ganze row betroffen ist
  // also startzeile, n*zeilen 8bit endzeilen z.b.
  // bzw drei masken berechnen dafuer...
  //
  for (y = y1; y<= y2; y++) {
    for (x = 0; x<=(x2 - x1); x++) {
      disp_setpixel(x, y, col);
    }
  }

}

// -------------------------END OLED ------------------------------------------

void setup() {
  Wire.begin(); // I2C Init
  i2c_oled_initall(i2coledssd); // OLED Init
}


void loop() {  // in der Schleife
  // 64x48 Pixel OLED
  static int phase=0; // Scrolleffekt hier statisch speichern
  disp_buffer_clear(COLOR_BLACK); // Virtuellen Buffer loeschen BLACK = dunkel
  disp_line_lcd (0,24,63,24, COLOR_WHITE); // x0,y0,x1,y1 Linie in Mitte
  for (int i=0; i<63;i++) { // nun fuer alle 64 Spalten (x)  des Displays
```

```
    int y=0; // Sinuskurve berechnen und in Radiant umrechnen von Phase
    y = (int)(23.0*sin(((double)i*3.141592*2.0)/64.0+phase/360.0*2.0*3.141592)+24.0);
    disp_setpixel(i, 47-y, COLOR_WHITE); // koordinate 0,0 ist links oben daher 47-y
  } // alle Spalten
  disp_lcd_frombuffer(); // dann erst updaten, double buffer flimmerfrei
  phase++; // beim nächsten Mal mit neuer Phase für den Sinus
  if (phase>=360)phase=0; // Immer 0 bis 359 Grad (als DEG)
}
```

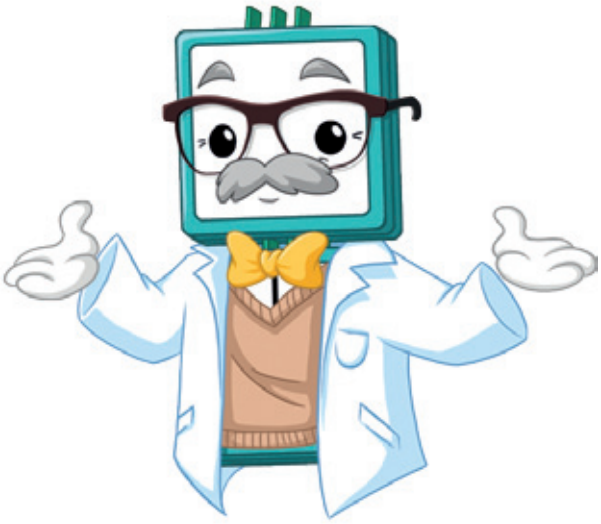
14. Ausblick

Das BrickRKnowledge-System wurde entwickelt und erfunden von Dipl. Ing. Rolf-Dieter Klein. Er hat das Amateurfunkrufzeichen DM7RDK (bzw. Ausbildungsrufzeichen DN5RDK - Seite Bundesnetzagentur, Mitglied DARC), daher taucht dieses kürzel auch öfters auf den Leiterplatten auf. Die Entwicklung des Brick'R'knowledge Systemes geschah 2014 im Rahmen der Nachwuchsförderung und Ausbildung im Bereich Amateurfunk und Industrie.

Rolf-Dieter Klein ist der Moderator der ehemaligen Computer Treff Sendung von BR-alpha sowie der Schöpfer des NDR-Klein-Computers aus den 90er Jahren (Neuere TV Sendungen bei www.computertrend-tv.com) sowie zahlreicher Bücher zum Thema Microcomputertechnik aus der Pionierzeit.

ALLNET GmbH
Maistrasse 2
D-82110 Germering
Tel.: +49 89 894 222-28
Fax: +89 89 894 222-33
www.brickrknowledge.de

email: info@brickrknowledge.de



ALLNET GmbH
Maistrasse 2
D-82110 Germering

Tel.: +49 89 894 222-22
Fax.:+49 89 894 222-33

www.brickrknowledge.de
email: info@brickrknowledge.de