Brick 'R' knowledge

ARDUINO

Arduino Coding Set

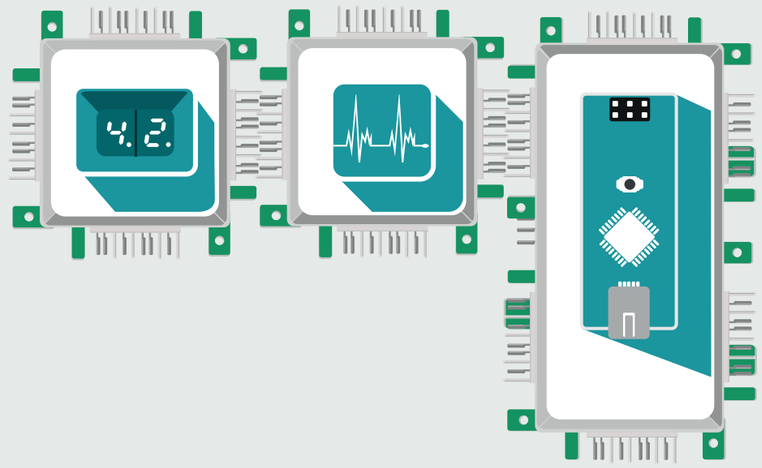
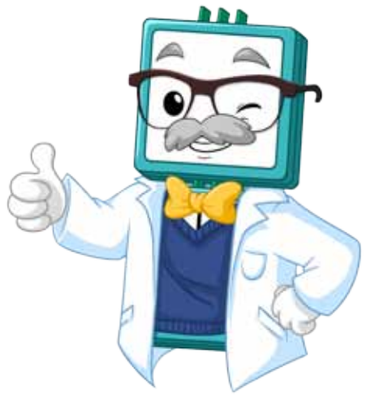


Table of contents

1. Preface

The Brick‘R‘knowledge was presented first time at the HAM radio conference 2014 by DM7RDK (callsign of Rolf-Dieter Klein) .
Now here we can present the Arduino coding set.
The special thing about the bricks is that both sides of the connectors contain a hermaphrodite, which can connect to itself and also can be bending while maintaining contact with a maximum of 6.3 amperes at 110 Volt. Of course we work with low voltages around 9 Volt to be safe.
Two contacts are reserved for the ground connection, allowing easy to document circuits, which can be uploaded by taking a photo to any social network for example. Even a 3D stackup is possible for complex circuits.

Have fun with the set and may ideas.
Rolf-Dieter Klein



1.1 Safety information

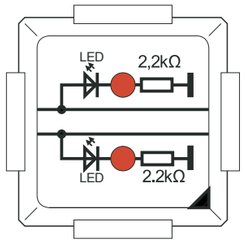
Note: Never connect the bricks directly to the mains power supply (115V/230V). There might be danger to life.

Please only use the included power supply-bricks. The voltage of our power-supply modules is 9V, which is not a health hazard. Please also ensure, that no open wires are in contact with the mains power outlets. Otherwise there might be a danger of hazardous electric shocks. Never look straight into LEDs, since this may damage your eye retina.
Please connect the included polarized capacitors (tantalum/electrolytic) only with the positive side to the plus side of the power supply. If those polarized capacitors are connected not correctly, they can be destroyed and even explode!

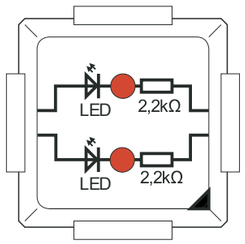
Please remove the power supply brick every time you finished experimenting, to avoid the risk of an electric fire.

2. Overview

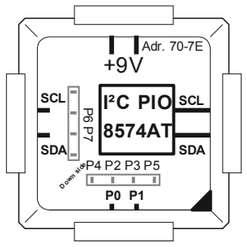
2.1 Bricks (components) of the Arduino Set



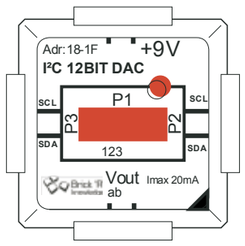
This brick contains two LEDs (light emitting diode). The LEDs are in series with a 2.2kOhm resistor to limit the maximum current. They are optimized for a 2mA current flow.
Both resistors are connected to ground, this allows connecting the brick directly to the output of a Arduino Nano brick. The two signals are also connected to the opposite site for feeding through the signals.



This brick contains two LEDs (light emitting diode). The LEDs are in series with a 2.2kOhm resistor to limit the maximum current. They are optimized for a 2mA current flow.
Both resistors connected to the opposite site. This allows a floating power level. To be similar to the above LED brick, a ground brick needs to be connected to the cathode. The cathode is at the resistor side, the LED symbol indicates this with a small bar.

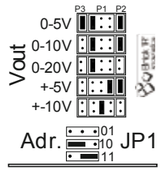


The Arduino has a special bus called I2C which allows connection of additional bricks via this serial bus. The 8574 is an integrated circuit which is used for extending the number of IO ports. Therefore 8 lines are provided which can be used as input or output port. The I2C bricks uses 3 address lines which can be set directly at the bottom of the brick for a maximum of 8 bricks at one bus. There is another IC called 8574T which can be used for additional 8 addresses. Both allow of 8x8+8x8=128 different IO ports to be used. All IO ports are available at the connectors of the brick, but P4-P7 at the bottom connector.

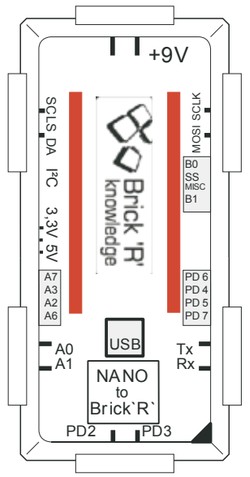


D/A converter brick: Using the I2C bus the output value can be set with 0..4095 different levels. This is assigned to a propotional voltage.
The bricks can be set for three different I2C addresses, allowing a maximum of three bricks at the same I2C bus.
In addition different output voltage ranges can be set via jumpers at the front site. Be careful when changing the jumpers.
The output allows for a maximum of 20mA.

The back side has a printed version of the voltage assignment for the jumpers and for the address jumpers.



Arduino NANO: This is the heart of our set. The processor is connected to the top side into the connectors. The programming is done like for other Arduino products using a PC and the Arduino development system, which can be downloaded from the homepage of the Arduino products.
For programming the Nano is also supplied via the USB-Port, also possible is an additional 9V supply. The USB Port can be disconnected after programming is done.
Attention never directly connect the bricks contacts of any IO to the 9V supply. Though there is a small protection circuit it won't last very long. The IOs are used with a 5V level.

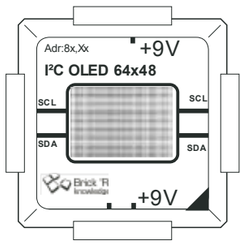


IO assignment
PD0 - 0
PD1 - 1
PD2 - 2
PD3 - 3
PD4 - 4
PD5 - 5
PD6 - 6
PD7 - 7

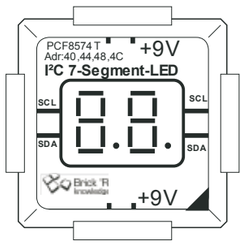
PB0 - 8
PB1 - 9
PB2 - 10
PB3 - 11
PB4 - 12
PB5 - 13

```
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
```

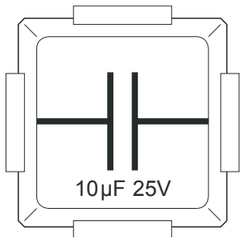
```
port definitions
shortname   port  parameter
SCK         PB5   #13
MOSI        PB3   #11
B1          PB1   #9
SS          PB2   #10
MISO        PB4   #12
B0          PB0   #8
```



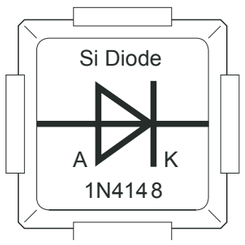
OLED: Organic Light Emitting Diode. Especially there is a matrix of 64x48 LEDs. Each of the can be addressed as a single LED by use of commands via the I2C bus. This allows for multi line text output as well as simple monochrom grafics.



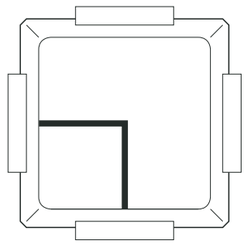
7-Segment-Display: Its using 7 light emitting bars and a single dot. Each of them contains a LED. The LEDs are addressed with the help of two 8574T circuits (16 ouput ports are used for the two displays). On the backside there is a small switch array, with which you can set the I2C address. A maximum of four such bricks can be operated on a single I2C bus.



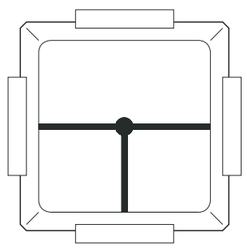
This capacitor has a capacitance of 10µF. A charging voltage of 1V is achieved already after 10µs, when loaded with a current of 1A. Capacitors must not exceed their specific maximum voltage.



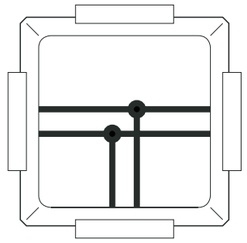
The special feature of this diode is that it can rectify voltages up to 100V and can also switch very high frequencies of up to 100MHz. It is operated in forward direction. In reverse direction the current flow is very low.



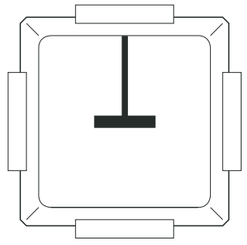
The corner-brick connects two bricks with a 90° angle.



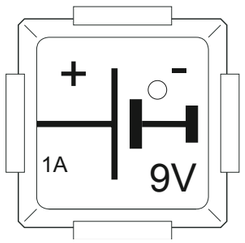
The T-bricks is used for junctions. It can also be used instead of a corner-brick.



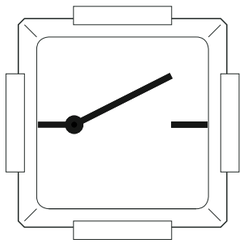
With a double T-brick also junctions can be implemented. In this case both middle contacts are separated.



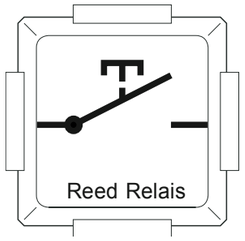
The ground-brick closes a circuit. This allows a much easier usage and saves the number of bricks needed to complete a circuit. The ground-brick connects the two center contacts with the two outer contacts used as ground connection.



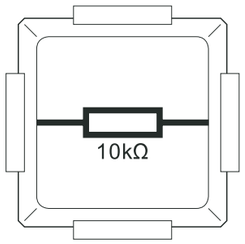
The power-supply-brick is the second way to power a circuit. It provides a stabilized DC voltage of 9V with a maximum short-circuit-proof current of 1A. The ground is connected to the minus pole, so that no further ground-bricks must be used to complete the circuit. A green LED signals the correct insertion of the brick. To reduce the risk of electronic malfunctions the power-supply-brick should be disconnected after an experiment.



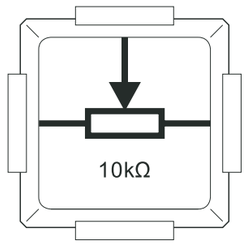
The push-button-brick is an electromechanical switch that allows a conductive connection only during pressing and holding the button. At the moment of release the connection will open again and the button returns to its inital position.



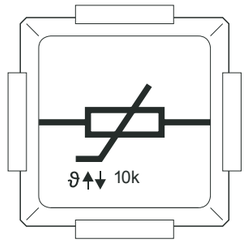
A reed contact switch (thin tube) is activated by an outside magnetic field. This magnetic field can be generated by a permanent magnet or an electromagnet. Reed switches have a low remanent magnetization.
They turn on whenever an external magnetic field is added in the same direction and turn off when the external magnetic field is removed. They are used as a proximity switch.



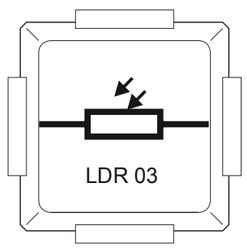
This brick has an electrical resistance of 10k. This value corresponds to a current flow of 100µA at a voltage of 1V. As all our resistors, the resulting power dissipation should not exceed 0.125 watt.



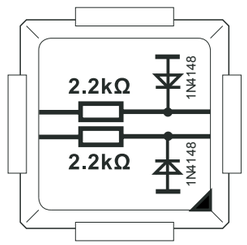
The potentiometer is a manually variable resistor. The third contact (wiper) changes the amount of electrical resistance by moving it from left to right. Can be adjusted in the range of 0 to 10k. If the wiper or one of the other contacts is connected directly to the power supply, a short circuit will appear. This must be avoided! The potentiometer has a maximum power of approximately 0.125 watt.



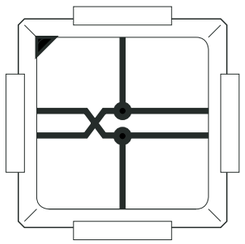
A NTC-resistor (Negative-Temperature-Coefficient) follows the temperature change with a positive gradient. For example if the temperature increases, the electrical resistance decreases.
The resistance value as on the circuit symbol is 10k at standard 25°C room temperature.
The NTC resistor is well suited as a temperature sensor.



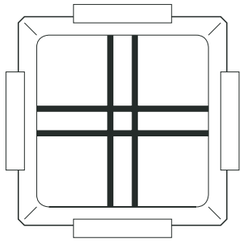
The LDR 03 is a light-dependent resistor. The more light falls on the sensor, the smaller the resistance will be. The values vary from a few 100 in bright environment and several kilo Ohm in the dark. The change of the resistance value is continuous.



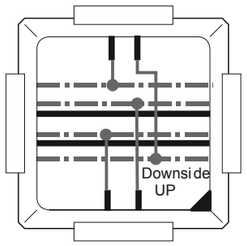
Double resistor with protection diodes. This can be used especially to protect the analog input of the Arduino Nano brick. The diodes must point towards the analog inputs of the Nano brick. They protect against negative voltages. The Nano bricks itself also contains two resistors with around 100 Ohms each. The 2.2k Ohm resistors in this brick can also protect the input against a 9V supply but don‘t connect them for a prolonged time, the analog inputs are limited to a 0-5V range.



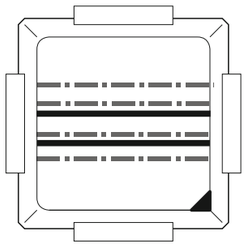
This brick connects the separate center contacts. By seperating and crossing the lines, the connection can be changed.



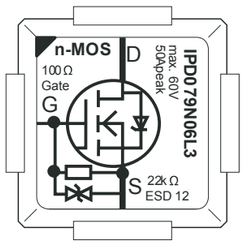
Not connected double crossing. In this case the two opposite middle contacts are connected to each other. The center is isolated against shorts.



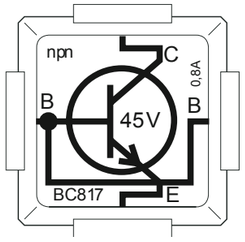
A very special bricks, designed for using the Nano bricks. In addition to the contacts at the upper side, also contacts at the bottom are available. With the help of this bricks those 4 contacts are connected to the upper side with 2 each. This allows normal bricks to be used for connecting the lower connector.



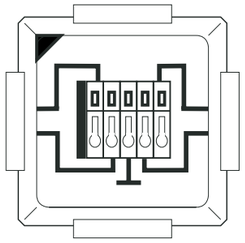
Connects the two contacts at the top level from one side to the other but also the four contacts at the bottom side which are independant from each other.



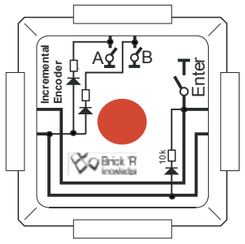
Our field effect transistor controls the current flow between drain and source via the voltage applied at the gate. The special thing about this device is that the connection between the gate and source is very high impedance. Field effect transistors are also called MOSFET (Metal Oxid Semi-channel Field Effect Transistor) or referred to as “MOS“.
There are different types of MOS, our brick is an enhancement type n channel which is normally in off condition. This means that a positive voltage above a certain threshold voltage needs to be applied to the gate, so the MOS begins conduct between drain and source.
This MOSFET can be used with up to 60V between source and drain. A peak current of 60A is also possible, of course not both at the same time. Attention: the contacts of the brick system can withstand a maximum of 6.3A per contact.
The gate input is protected by an ESD diode and an additional 22kOhm resistor to the source contact.



This brick contains the same npn transistor. It controls the flow of the current between the collector (C) and Emitter (E) via its smaller current into its base contact (B).
The base should be positive in relation to the emitter for normal operation. In addition this brick offers a connection of the base to the left side and to the right side. This allows a much easier construction of complex circuits.



This allows connecting wires or components directly into the circuit. Pressing a small screwdriver in the slots above opens the contact and a wire can be inserted from the side. After release of the screwdriver pressure the wire is fixed.



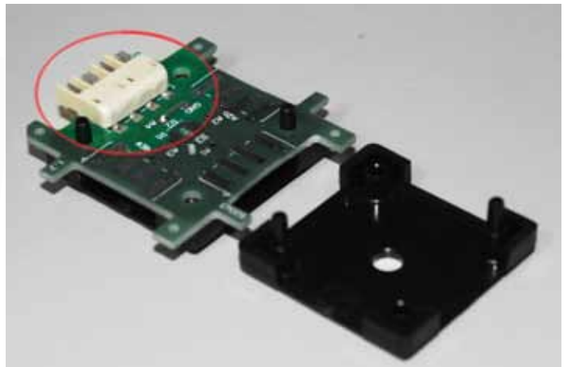
Rotary encoder.
Such elements are often used on operating panels. There are two contacts with a 90 degree phase shift. The CPU can use this information to get the rotation direction. Pressing the button closes an additional contact. Depending on the used rotary encoder type, 18 to 36 contact closes per revolution are activated.

3. Basic Operation

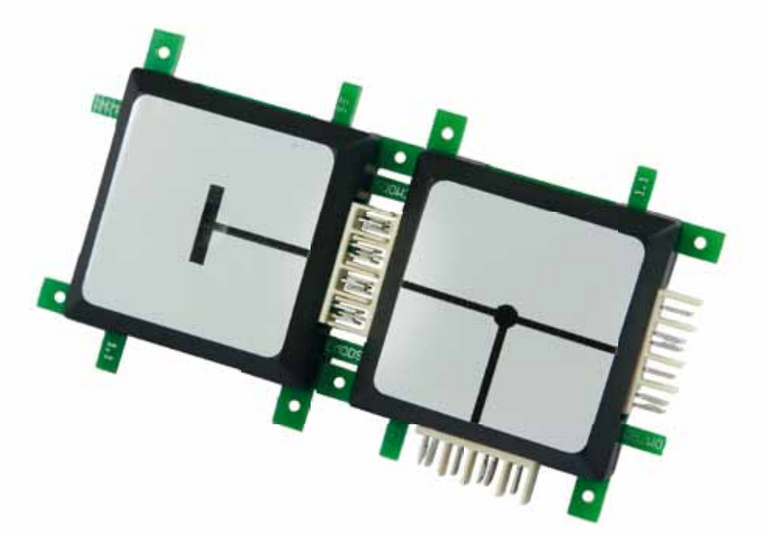
3.1 The Brick'R'knowledge system

The ground brick is a special brick in our system. It saves the use of additional connector bricks or wires. This shows the secret of the four contacts. The two middle contacts are used for signal transmission, according to the print on top. The outer two contacts are used for closing the circuit, that means allowing the current to flow back to the power source. Its called ground brick, because the technicans often call it ground or 0V level.

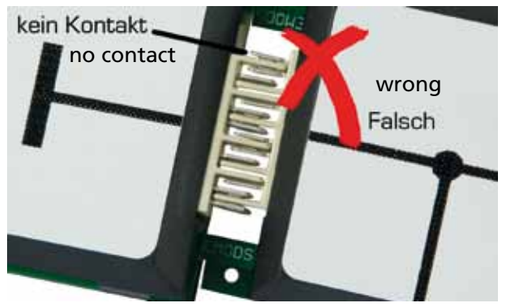
This ground simplifies the electronic circuit by using a symbol for the return path. Our ground brick also connects both inner contacts to ground that means the outer contacts. All bricks connect the outer two contacts together even if it's not printed on the label.



Be careful when connecting the bricks together. If shifted, an open or short circuit can result. They must fit exactly together.



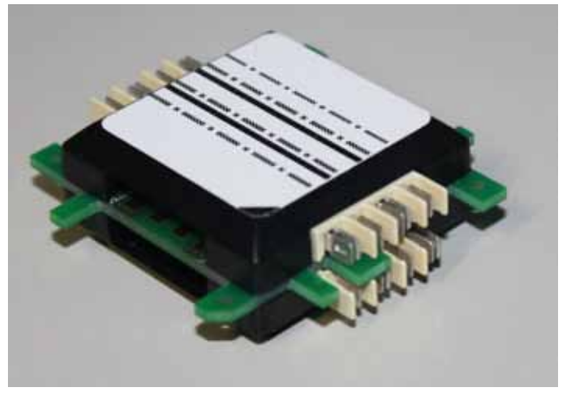
This is an example of a correctly connected brick. The connection has small contacts and also isolating parts which fit exactly together. The isolating parts are used to separate the contacts and avoiding shorts.



Here the connector is shifted by one position, the contacts are all isolated, and therefore no contact is done. If shifted once more a short can result. Carefully check all connections before you power up a brick circuit.

Attention: its really important to have a good and right adjusted contacts. Shorts can also lead to destruction of bricks and even result in fire to the system.

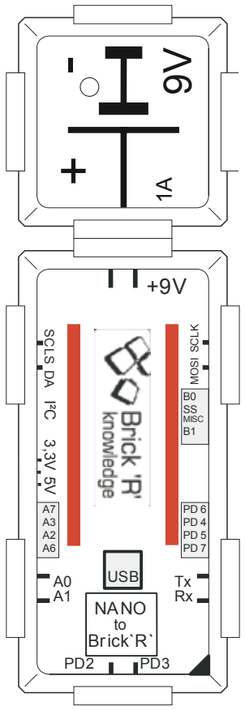
Our Arduino Set has an additional contact level. The contacts on the bottom allow for additional 4 signals per side. When you connect such bricks, be very careful when you plug them together. An additional nose is used to guide the connection. This nose also prevents the bricks from bending up and down which is possible with normal bricks.

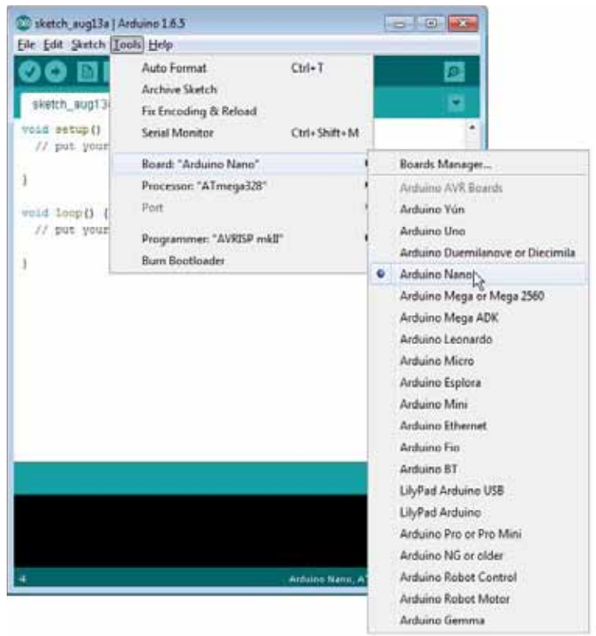


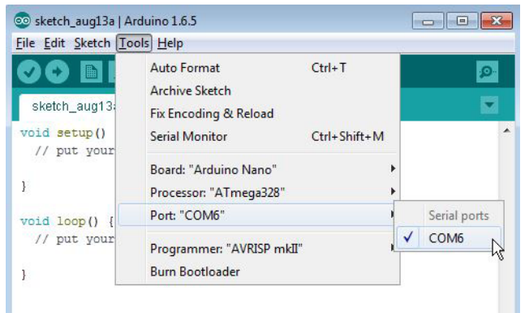
Important: Always check your connections for precise contacts.

3.2 Arduino Nano Brick - First steps

The Arduino Nano brick can be supplied with either the USB port or by an external power supply. You can use our 9V power supply for this purpose. But be very careful. Never connect this brick to any other of the contacts except for the power supply input, where 9V is written on the label. This might destroy the Nano brick.
If you get a plain Nano brick, there might be already a test program running which lets the LED blink on the Nano board. To use your own program, you need to connect the brick via an USB cable to your PC. You should have installed the Arduino Development Software. Then you select the Arduino Nano as board and select the according COM port for the communication.



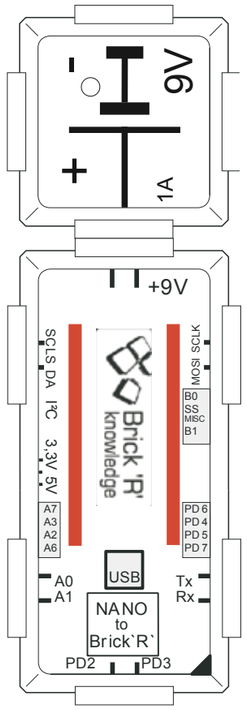




4. LEDs & Push button

4.1 On Board LEDs

The Arduino Nano has a lot of in- and output ports, some of them a wired to our contacts on the Nano board interface. There is a LED on the printed circuit board of the Nano itself, which is wired to port 13 (=PB3).
With our simple programming example you can let it blink.




```
// EN_1
#define PORTLED 13 // this defines a symbol with the value 13

// only executed once after startup of the arduino
void setup() {
  pinMode(PORTLED,OUTPUT); // Port 13 assigned as output
}

// this loop is called in repetition by the arduino
void loop() {
  // set output to a high level
  digitalWrite(PORTLED,HIGH); // high means 5V in our case
  delay(1000); // 1 second delay = 1000 millisecond
  digitalWrite(PORTLED,LOW); // set low level on output
  delay(1000); // wait for an additional second
} // end of loop
```

What happens? The red LED on the printed circuit board of the NANO should blink with around 1 time per second.

You can adjust the on/off time by changing the parameter at the delay command.

Below we printed a table of all IO Port assignments (0-13 : logical number) and the according physical names which are split into Port B and D from the Arduino processor.

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

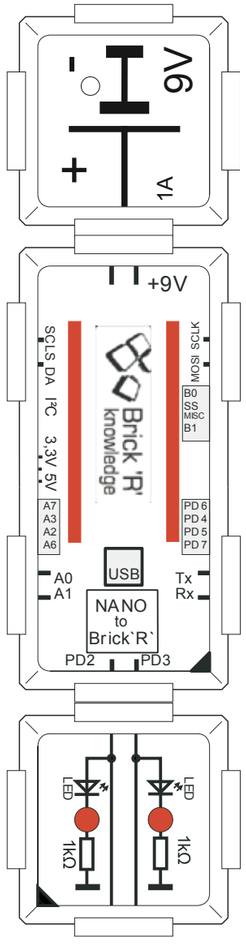
PB0 - 8
PB1 - 9
PB2 - 10
PB3 - 11
PB4 - 12
PB5 - 13

Some of those ports are at the bottom side of the brick. You can get access to these ports by using our special brick, which has wired those bottom connections to the top side. PD2 and PD3 for example are accessible directly on the top side of the brick. PB0 and some other lines have a multiple meaning, which is listed below. Examples will follow.

```
port definitions
shortcut    port   parameter
SCK         PB5    #13
MOSI        PB3    #11
B1          PB1    #9
SS          PB2    #10
MISO        PB4    #12
B0          PB0    #8
```

4.2 Double LED

Now let's directly connect one of our bricks to the Nano. Therefore we use one the double LED brick with internal ground connection. The Nano uses two of the inner contacts as separate IO, which can now be directly assigned to the LEDs. The outer two contacts are used for the ground connection which is not shown in the schematic. Here we connect the LEDs to the ports PD2 and PD3, which will be programed as outputs. 2 and 3 is also the numbering used in the Arduino sketch. When keeping the connection via USB the 9V brick is not necessary in this case. After programming successfully the LEDs are blinking one or the other changing. The LED bricks have different colors, which you can choose as you like. Be careful to use the LED bricks with the ground symbol inside. We will show later how you connect the other bricks.

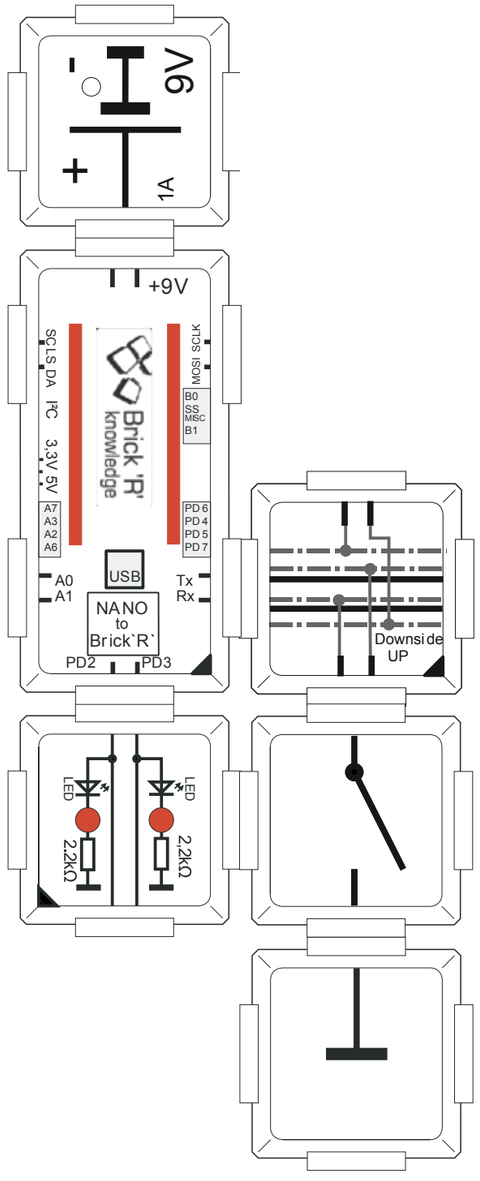



```
// EN_2 double LED

#define PORTLED2 2 // Symbol gets the value 2
#define PORTLED3 3 // and here same for 3

// this loop is called in repetition by the arduino
void setup() {
  pinMode(PORTLED2,OUTPUT); // Port 2 to be output
  pinMode(PORTLED3,OUTPUT); // Port 3 to be output
}

// this loop is called in repetition by the arduino
void loop() {
  digitalWrite(PORTLED2,HIGH); // set the output to a high level
  digitalWrite(PORTLED3,LOW); // set the output to a low level
  delay(1000); // wait one second
  digitalWrite(PORTLED2,LOW); // set the output to a low level
  digitalWrite(PORTLED3,HIGH); //  set the output to a high level
  delay(1000); // wait one second
}
```

What happens? The two LEDs blink in a one second interval alternating.

4.3 Push button & LED

Now we inserted an additional brick with a push button. When pressing the botton a connection to ground is done. If the button is open, the input port of the Nano would be undefined. To solve this usually a pullup resistor towards 5V (not 9V!) is necessary, but the Adrunio Nano has a nice option called INPUT_PULLUP. This sets an internal pullup resistor and eliminates the need to wire an external resistor. The internal resistor is around 20kOhm towards 5V. The push button cannot be connected directly to the Ports 4 or 5, as they are contacted at the bottom side. Therefore we use our special brick. This wires the bottom contacts to the upper side.
In our case we connect both ports PD4 and PD5 at the same time to the contacts of the push button, which is no problem as long as both ports are defined to be input. PD6 and PD7 are now also on the upper side but not used.

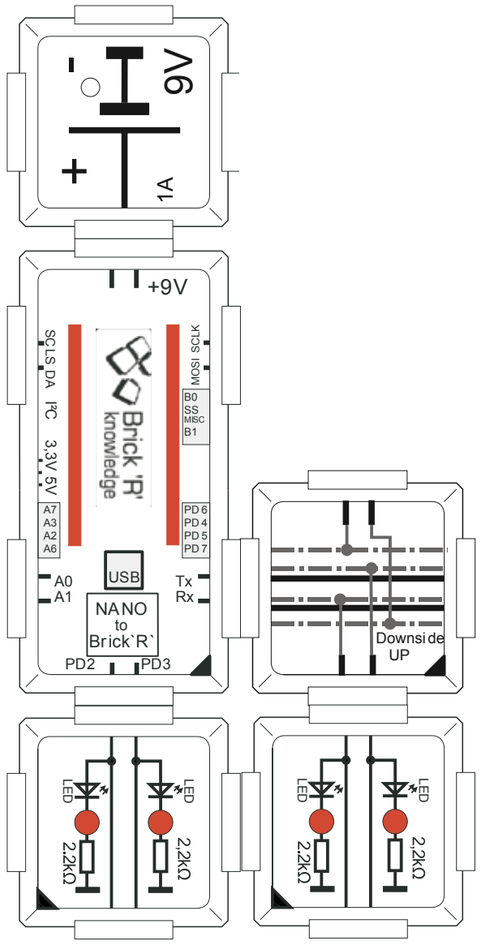



```
// EN_3 Double LED and Button

#define PORTLED2 2 // Symbol gets the value 2
#define PORTLED3 3 // and here same for 3

#define PORTSWTTCH4 4 // push button to  Port 4
(und 5)

void setup() {
  pinMode(PORTLED2,OUTPUT); // Port 2 to be out-
put
  pinMode(PORTLED3,OUTPUT); // Port 3 to be out-
put
  pinMode(PORTSWTTCH4,INPUT_PULLUP); // Port 4 as
input with pullup resistor
}

void loop() {
  if (digitalRead(PORTSWTTCH4)==LOW) { // only
when the button is pressed
    digitalWrite(PORTLED2,HIGH); // set the out-
put to a high level
    digitalWrite(PORTLED3,LOW); // set the output
to a low level
    delay(1000); //  wait one second
    digitalWrite(PORTLED2,LOW); // Ausgang auf
niedrigen Pegel schalten
    digitalWrite(PORTLED3,HIGH); // set the out-
put to a high level
    delay(1000); //  wait one second
  } // on of polling the button
} // end of loop
```

What happens? The two LEDs blink in a one second interval alternating as long as the button is pressed.

4.4 Multiple LEDs are blinking

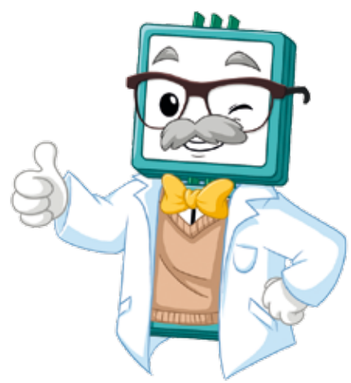
The LEDs show a binary code of a counter. This counter runs from 0 to 15. For this to happen, single bits of the counter variable are checked and then assigned to the different LEDs, which can be on or off. Attention the LEDs have the strange sequence 2, 3, 5 and 4, which is also handled in the programming code.
This time we use a new term „static“ which means that a variable keeps its value also between calls. This is like a global variable.
Below we listed the binary code. The counter is implemented by the assignment „counter = counter + 1“. But this will lead to an overflow, depending whether the variable is a 16 bit integer or a 32 bit integer. Usually this is not a problem, but it’s more clear to use a special instruction to prevent this. We use the „&“-operation which is a logical AND. The value of 0xf is also called mask value. With the assignment „counter = (counter +1) & 0xf“ the counter is incremented first and then the AND operation is done, which only keeps the least for bits of the variable „counter“ and so prevents an overflow.



decimal	binary
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

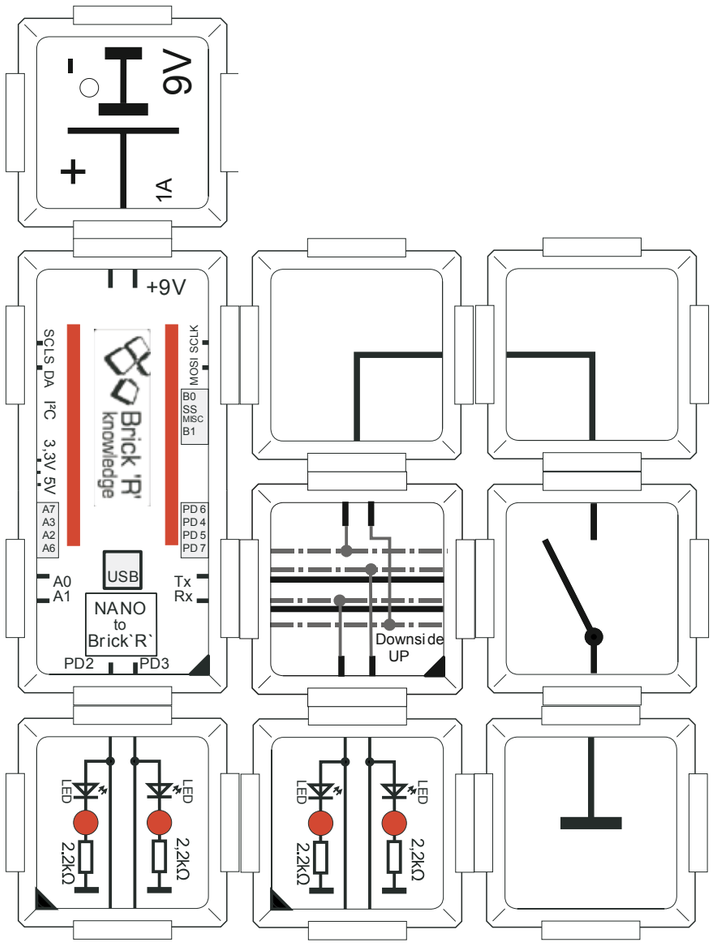
```
// EN_4 Multiple LEDs blink

#define PORTLED2 2 // now define 4 LEDs
#define PORTLED3 3 // use  PD2,PD3,PD4,PD5
#define PORTLED4 4 //
#define PORTLED5 5

void setup() { // start initializing
  pinMode(PORTLED2,OUTPUT); // Port 2 as output
  pinMode(PORTLED3,OUTPUT); // Port 3 as output
  pinMode(PORTLED4,OUTPUT); // Port 4 as output
  pinMode(PORTLED5,OUTPUT); // Port 5 as output
}

void loop() {
  static int counter = 0; // keep the counter value
    // because we use a static variable
    // withotu static it will be reset to 0
    // each time the loop is entered
    if (counter & 1) { // check bit 0
     digitalWrite(PORTLED4,HIGH); // right  LED
    } else digitalWrite(PORTLED4,LOW); //
    if (counter & 2) { // check bit 1
     digitalWrite(PORTLED5,HIGH); // left  LED
    } else digitalWrite(PORTLED5,LOW); // next
    if (counter & 4) { // check bit 2
     digitalWrite(PORTLED3,HIGH); //  // set to 1
    } else digitalWrite(PORTLED3,LOW); //
    if (counter & 8) { // check bit 3
     digitalWrite(PORTLED2,HIGH); // now last bit 3
    } else digitalWrite(PORTLED2,LOW); //
    delay(1000); // wait fro 1sec = 1000ms
    counter = (counter + 1)& 0xf; // this keeps the counter between 0..15
} // end of the loop
```

What happens? The four LEDs show the binary count from 0000 (all off), then 0001, 0010, 0011 to 1111 (all on). Then all repeats.



4.5 Binary counter - controlled by a push button

Counting is done every second. Now we can build a very simple stopwatch by adding a push button. The stopwatch counts only if the button is pressed.
Below a table helping for conversion. Later on we will implement a stopwatch which a decimal output and much more comfort.
The button is connected to both ports 6 and 7. We use port 7 for polling the button only, the short to port 6 does not matter because it defaults to an input port without pullup.
Now the counter should keep counting as long as the button is pressed. But it overflows to zero after reaching the maximum value of 1111=15.



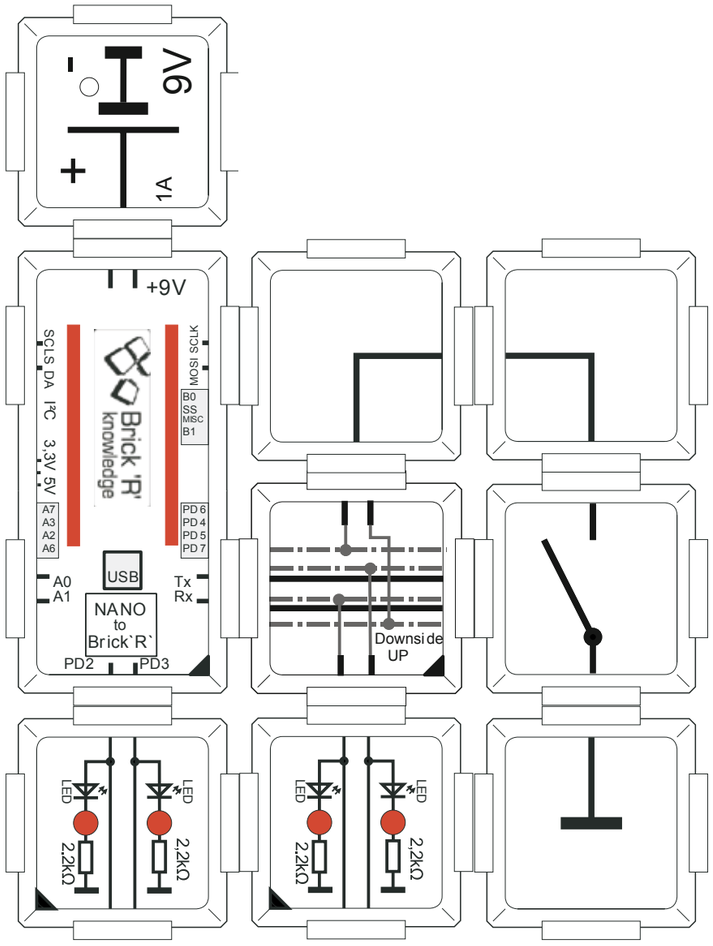
decimal	binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

```
// EN_5 Multiple LEDs and control by button

#define PORTLED2 2 // now define 4 LEDs
#define PORTLED3 3 // use  PD2,PD3,PD4,PD5
#define PORTLED4 4 //
#define PORTLED5 5

#define SWITCH7 7 // the button is wired to PD7

void setup() { // define all inputs and outputs
  pinMode(PORTLED2,OUTPUT); // Port 2 as output
  pinMode(PORTLED3,OUTPUT); // Port 3 as output
  pinMode(PORTLED4,OUTPUT); // Port 4 as output
  pinMode(PORTLED5,OUTPUT); // Port 5 as output
  pinMode(SWITCH7,INPUT_PULLUP); // Port 7 as input with pullup
  // a pullup is an internal resistor, which is programmable
  // if the button is not pressed and undefine condition at the
  // input would result. With the pullup the return of a high value
  /// that is a 1 value is returned.
}

void loop() {
  static int counter = 0; // important the use of static
    if (counter & 1) { // check bits 0..3 one after the other
     digitalWrite(PORTLED4,HIGH); // right  LED
    } else digitalWrite(PORTLED4,LOW); //
    if (counter & 2) { // check bit 1
     digitalWrite(PORTLED5,HIGH); // left  LED
    } else digitalWrite(PORTLED5,LOW); //
    if (counter & 4) { // check bit 2
     digitalWrite(PORTLED3,HIGH); //
    } else digitalWrite(PORTLED3,LOW); //
    if (counter & 8) { // check bit 3
     digitalWrite(PORTLED2,HIGH); //
    } else digitalWrite(PORTLED2,LOW); //
    // define the delay time for maximum count speed
    delay(1000); //wait for a second
    // now check teh button at this point
    if (digitalRead(SWITCH7) == LOW) { // coutn if pressed
      counter = (counter + 1) & 0xf; // 0..15 cyclic counter
    } // only if pressed, also hold the  counter value
} // end of loop
```

What happens? If the button is pressed and only then the LEDs show the binary value of the counter from 0000, 0001....to 1111 (all on), then 0000, 0001 in a cycle manner. If the button is released the currrent counter value is displayed and does not change.

4.6 Control the speed of counting

With a push button you can do other things also. For example control the counting speed. There is only a minimal change in the program. Instead of enabling the counter as before, now the parameter of the delay() instruction is changed. For a pressed button it's using 200ms and for a non pressed button its using 1000ms = 1sec as a delay. Important to know is that the count speed is not exactly what is used in the delay, as the execution time of instructions in-between must be added to the delay time. The delay is very precise using the crystal oscillator of the CPU. To adjust you can try a value little bit lower, but there are other programming ways to implement this more precise (and more complicated).

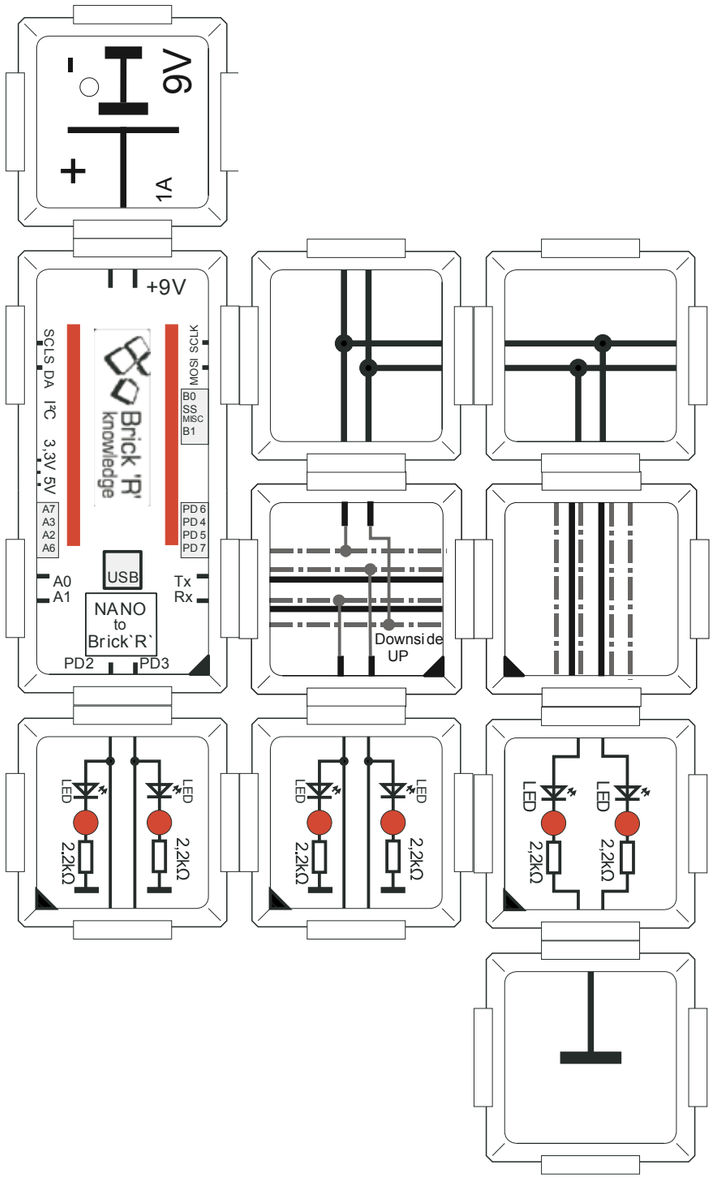


```
// EN_6 speed of counting is controled

#define PORTLED2 2 // now define 4 LEDs
#define PORTLED3 3 // use  PD2,PD3,PD4,PD5
#define PORTLED4 4 //
#define PORTLED5 5

#define SWITCH7 7 // the button is wired to PD7

void setup() { // IO definieren
  pinMode(PORTLED2,OUTPUT); // Port 2 as output
  pinMode(PORTLED3,OUTPUT); // Port 3 as output
  pinMode(PORTLED4,OUTPUT); // Port 4 as output
  pinMode(PORTLED5,OUTPUT); // Port 5 as output
  pinMode(SWITCH7,INPUT_PULLUP); // Port 7 as input with pullup.
}

void loop() {
  static int counter = 0; // important the use of static
  if (counter & 1) { // check bits 0..3 one after the other
   digitalWrite(PORTLED4,HIGH); // right  LED
  } else digitalWrite(PORTLED4,LOW); //
  if (counter & 2) { // check bit 1
   digitalWrite(PORTLED5,HIGH); // left  LED
  } else digitalWrite(PORTLED5,LOW); //
  if (counter & 4) { // check bit 2
   digitalWrite(PORTLED3,HIGH); //
  } else digitalWrite(PORTLED3,LOW); //
  if (counter & 8) { // check bit 3
   digitalWrite(PORTLED2,HIGH); //
  } else digitalWrite(PORTLED2,LOW); //
  //
  if (digitalRead(SWITCH7) == LOW) { // if pressed
    delay(200); // wait only 200ms inbetween
  } else { // else count slow
    delay(1000); //with 1 sec inbetween
  } // then continue
  // always count up
  counter = (counter + 1)& 0xf; // 0..15 counter cyclic
}
```

What happens? If the button is pressed the counter is incremented every 200ms, that is the LEDs show the counter state 0000, 0001 ... 1111 (all on) much faster then if the button is not pressed and a change occurs every second.

4.7 Simple chasing light

We can enhance our circuit to a chasing light. This will use an addtional double LED brick. Also special bricks are used to connect the bottom contact side of the bricks to the upper and another special bricks is just used as a wire through for the upper side to connect the PD6 and PD7 to the LEDs. The bottom side is left unconnected. Be careful when inserting those special bricks.
Attention also for the assignment of Ports PD6 and Ports PD7. If the double T is inserted in a different orientation, the chasing light no longer chases, but you can try for you own.

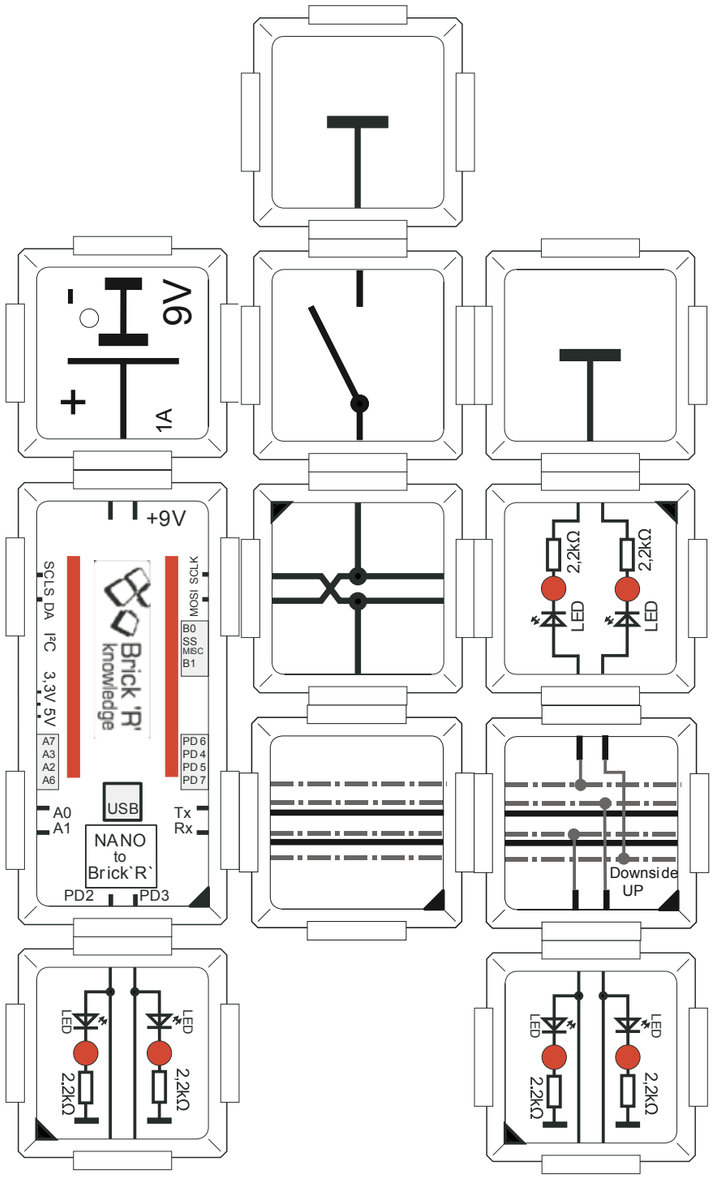


```
// EN_7 chasing lights using  6 LEDs

#define PORTLED2 2 // define all LED ports
#define PORTLED3 3 // ports PD2 to PD7
#define PORTLED4 4 // are all used
#define PORTLED5 5 // please check the exact
#define PORTLED6 6 // assignment at the
#define PORTLED7 7 // bricks

void setup() { // IO definitions
  pinMode(PORTLED2,OUTPUT); // Port 2 as output
  pinMode(PORTLED3,OUTPUT); // Port 3 as output
  pinMode(PORTLED4,OUTPUT); // Port 4 as output
  pinMode(PORTLED5,OUTPUT); // Port 5 as output
  pinMode(PORTLED6,OUTPUT); // Port 6 as output
  pinMode(PORTLED7,OUTPUT); // Port 7 as output
}

void loop() {
  static int shiftreg = 1; // use 6 bits for the chasing light
    if (shiftreg & 1) { // check bit 0 to bit 5
     digitalWrite(PORTLED6,HIGH); // right LED
    } else digitalWrite(PORTLED6,LOW); //
    if (shiftreg & 2) { // check bit 1
     digitalWrite(PORTLED7,HIGH); // left LED
    } else digitalWrite(PORTLED7,LOW); //
    if (shiftreg & 4) { // check bit 2
     digitalWrite(PORTLED4,HIGH); //
    } else digitalWrite(PORTLED4,LOW); //
    if (shiftreg & 8) { // check bit 3
     digitalWrite(PORTLED5,HIGH); //
    } else digitalWrite(PORTLED5,LOW); //
    if (shiftreg & 0x10) { // check bit 4
     digitalWrite(PORTLED3,HIGH); //
    } else digitalWrite(PORTLED3,LOW); //
    if (shiftreg & 0x20) { // check bit 5
     digitalWrite(PORTLED2,HIGH); //
    } else digitalWrite(PORTLED2,LOW); //
    //
    delay(300); // 300 ms wait time
    shiftreg = shiftreg << 1; // 1,2,4,8,16,32  shift instruction of C
    if (shiftreg > 32) shiftreg = 1; // then  start again
} // end of loop
```

What happens? There is always one LED of the 6 possible in on condition. The light chases from right to left. Every 300ms a change is done. After this all is repeated in a cycle.

4.8 Chasing light - speed control

An additional push button allows for controlling the program. In this case the speed of chasing. For this we use one of the B-ports which has several meanings. This one is also called MOSI and is identically to Port PB3, it uses the parameter #11 for programming. MOSI is the short for „master out slave in" used for the internal SPI interface (MOSI+MISO+SCJ). But those pins can be used as normal I/O ports also, like in our example.
If the button is pressed, we use a delay time of 50ms and therefore the chasing light speed is much faster than with a non pressed button with 300ms.



port assignment

shortcut	port	parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

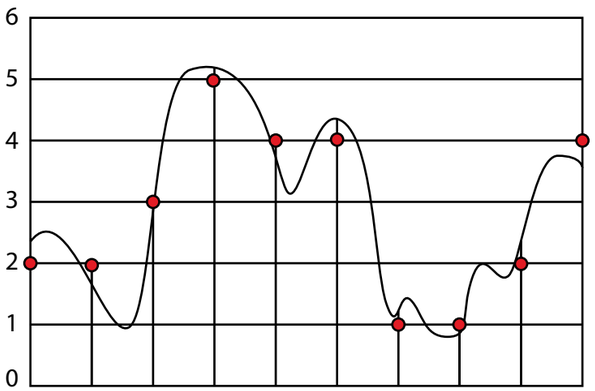
What happens? There is always one LED of the 6 possible in on condition. The light chases from right to left. As long as the button is pressed the LEDs changes every 50ms which is then very fast and if not pressed every 300ms like in the example before.

```
// EN_8 chasing light with button - control the speed

#define PORTLED2 2 // define all LED ports
#define PORTLED3 3 // ports PD2 to PD7
#define PORTLED4 4 // are all used
#define PORTLED5 5 // please check the exact
#define PORTLED6 6 // assignment at the
#define PORTLED7 7 // bricks

#define SWITCHB3MOSI 11 // we use PB3 as n input
// its assigned as no 11 and uses the name MOSI
// as it has a double meaning.

void setup() { // define all IOs
  pinMode(PORTLED2,OUTPUT); // Port 2 as output
  pinMode(PORTLED3,OUTPUT); // Port 3 as output
  pinMode(PORTLED4,OUTPUT); // Port 4 as output
  pinMode(PORTLED5,OUTPUT); // Port 5 as output
  pinMode(PORTLED6,OUTPUT); // Port 6 as output
  pinMode(PORTLED7,OUTPUT); // Port 7 as output
  pinMode(SWITCHB3MOSI,INPUT_PULLUP); // MOSI is used
}

void loop() {
  static int shiftreg = 1; //  use 6 bits for the chasing light
   if (shiftreg & 1) { // check bit 0 to bit 5
    digitalWrite(PORTLED6,HIGH); // right LED
   } else digitalWrite(PORTLED6,LOW); //
   if (shiftreg & 2) { // check bit 1
    digitalWrite(PORTLED7,HIGH); // left LED
   } else digitalWrite(PORTLED7,LOW); //
   if (shiftreg & 4) { // check bit 2
    digitalWrite(PORTLED4,HIGH); //
   } else digitalWrite(PORTLED4,LOW); //
   if (shiftreg & 8) { // check bit 3
    digitalWrite(PORTLED5,HIGH); //
   } else digitalWrite(PORTLED5,LOW); //
   if (shiftreg & 0x10) { // check bit 4
    digitalWrite(PORTLED3,HIGH); //
   } else digitalWrite(PORTLED3,LOW); //
   if (shiftreg & 0x20) { // check bit 5
    digitalWrite(PORTLED2,HIGH); //
   } else digitalWrite(PORTLED2,LOW); //
   //
   if (digitalRead(SWITCHB3MOSI)==LOW) { // means PORT PB3 #11
     delay(50); // 50  ms which is fast
   } else {
     delay(300); // 300 ms much slower here
   }
   shiftreg = shiftreg << 1; // 1,2,4,8,16,32  then again  1,2,3,4...
   if (shiftreg > 32) shiftreg = 1; // starts shifting from the beginning
} // end of the loop.
```

5. Analog/Digital-converter

5.1 A/D-converter - principle construction

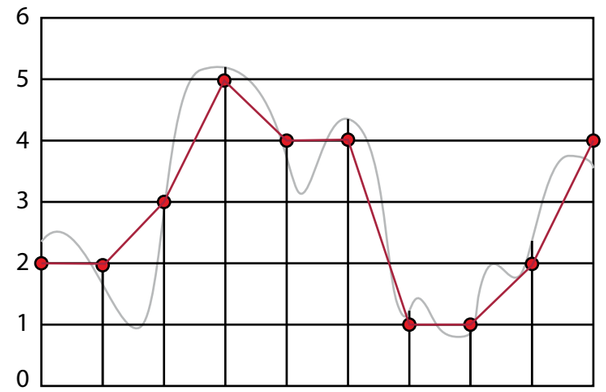
A/D means analog to digital. An A/D-converter should convert an analog signal like a voltage to a digital representation. This digital value can be handled by a computer for further processing. A standard computer cannot use analog signals directly. There are two important steps to get a digital value out of an analog signal. First the analog level needs to be quantized and second there is a quantization in time if the analog value is changing over time. What does this mean?
First quantization of the amplitude. An analog voltage with a range of 0-5V for example can have any value in between. For example 2.3V or 2.31V. The question is what should be the precision you need and that means how many steps are acceptable. For example we you like to digitize the voltage range of 0 to 5V with 6 steps. Which digital value is assigned for 2.1V? In the diagram below you can see the assignment done in this case. The value of 2.1 is closer to 2 than to 3. This means the digital value of 2 is chosen. With an analog value of 2.5 the digital value of 3 would be used is rounding applies and so on.



In the diagram there is also a change of the analog signal over time. The y-axis might be used as voltage and the x-axis for seconds in this example.
To get a digital representation, now you also have to choose for the time steps in which you sample the signal For example we take one in a second, these are the vertical lines. If you round properly you might get the following number sequence:

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

There are some interesting effects. Information is lost from the amplitude and also from the time sampling. For the amplitude the first two values are assigned to 2. Using more precision would have conserved this.
Also information over time is lost. The sequence „...1,1...“ does not contain the small wave in-between.
You need to take care that no frequency part in the signal is higher than the half sample rate, this is call nyquist criteria, and can lead even to totally false interpretation of signals.
Now if you connect the red lines, you get the signal representation the computer can see. The original curve cannot be reconstructed totally, only part of it. If the number of sample points is increased (time or amplitude) things get better.

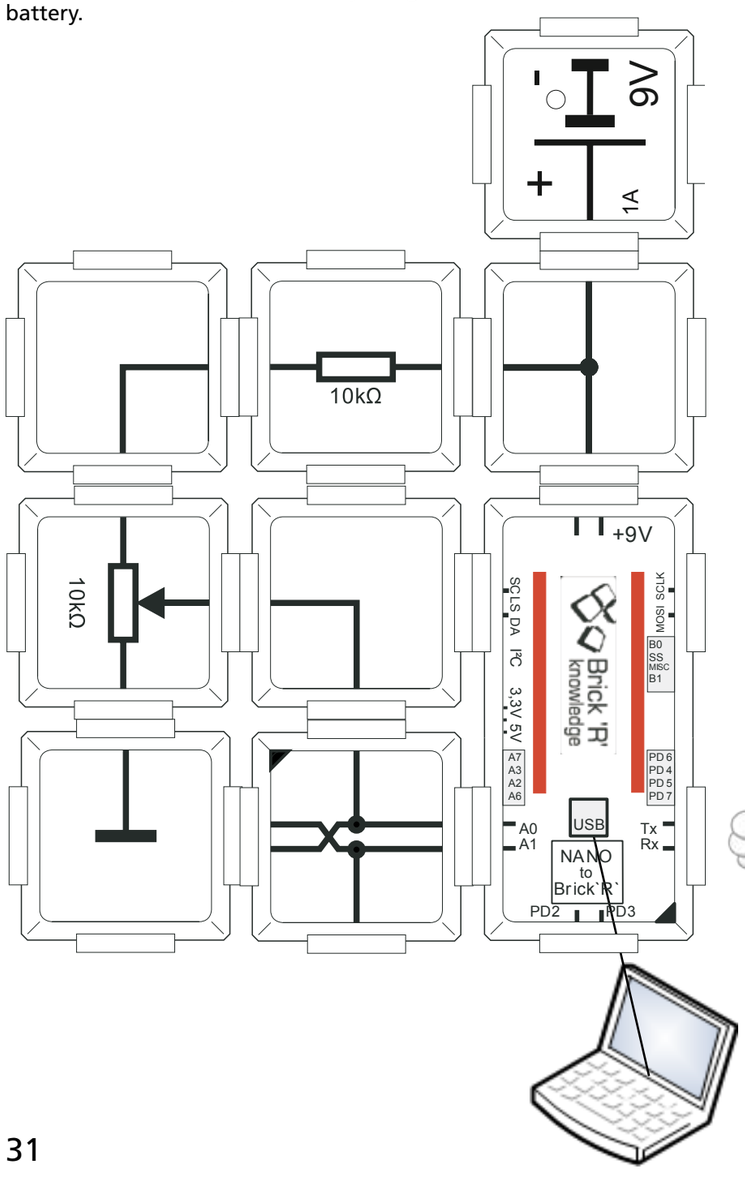


Above you see the reconstructed curve.
The resolution of the amplitude is determined by the number of bits used, which are assigned to a number value. The analog digital converter built into the Nano use 10 Bit. This is 2 raised to 10 different values, that means 1024 different ones. With a voltage range of 0 to 5V the smallest step size is 5V/1024 = 4.88mV.
The technicians are interested in the dynamic range, which could be calculated
= 20 * LOG (number of steps) in dB.
For us = 20*LOG(1024) = 60.2 dB. This is a rather good value, the human ear for example has a much higher dynamic, therefore if audio signals should be represented, for a HiFi quality you need much more bits. For 24 bits it results in 20 * LOG (2 raised to 24) = 20 * LOG (16777216)=144.5 dB.
The human hearing is about 120 dB in some of the frequency ranges.

But how can such signals be converted to digital ones. There are many different approaches which will quickly explode the manual here. But some search keywords should be mentioned here: delta sigma, successive approximation (SAR), parallel converter, sawtooth method, multiple ramps for some of them.
The A/D-converter of the Arduino Nano is built into the Atmel 328 processor. The converter can handle a sample rate of up to 15kSPS (with 10 bit) that means 15000 samples per seconds.
The inputs are multiplexed, that means only one of the multiple inputs assigned for analog signals can be used for conversion at the same time. If multiple channels are used, the conversion rate is reduced accordingly, because the converter can only be used for one channel at the same time.
The conversion principle is a successive approximation.
The use of the Nano AD convertes is very simple, the Ardunio software library using the `analogRead(PINnumber) instead of` digitalRead(PINnumber) makes it possible for all pins assigned to a A/D channel. A value between 0..1023 is returned which corresponds to an analog value of 0..5V

5.2 A/D-converter and potentiometer

The value of the built in A/D-converter is read with the sketch command analogRead(). This results in a value between 0 and 1023. 0 is assigned to 0V and 1023 is equivalent to 5V. The values are printed via serial interface on a PC terminal. This can be done by using the key sequence CTRL-SHIFT-M when you have activated the Arduino programming window. The numbers are then listed in the terminal windows.
Why we only reach 920 instead of 1023. The 10kOhm resistor is in sequence with the 10kOhm value of the potentiometer and forms a voltage divider. Best you can have is therefore half of the battery voltage, for a 9V battery this means 4.5 Volt. The full range of the A/D-converter is 5V so a value less than 1023 is the result of the measurement.
Reverse its possible to calculate the battery voltage out of the measurement which is:
VBatt = (value*5)/1023.
For printout the line `Serial.print((value*10.0)/1023.0); Serial.print(„V „); is used.`
If the potentiometer is on one side you get the full battery voltage, otherwise you can turn it down to 0V.
Attention the Nano inputs are only designed for a maximum of 5V, so never connect them directly to the 9V battery.



Attention:
Activate the terminal

You need to press CTRL-SHIFT-M when the arduino software is active.
A window with the terminal pops into the way.
Attention: set the baudrate to 9600 if it does not show a readable text.

```
// EN_9 AD converter and poti

// CTRL-SHIFT-M AT PC for
// activating the terminal
// seriell monitor.
//


#define PORTAD0 0 // we use channel 0
// the putput is done at the terminal on the PC
// later on we will connect a display.
void setup() { // start
  Serial.begin(9600); // use this baudrate
  // with this data is transfered to the PC
  // 9600 baud = 9600 bits/second
}

void loop() { // start of the looop
  int value; // temp storage
  value = analogRead(PORTAD0); // read AD converter
  Serial.print((value*10.0)/1023.0);  // calc voltage
  Serial.print(„V „); // print Volt symbol
  Serial.println(value); // and then the value
}
```

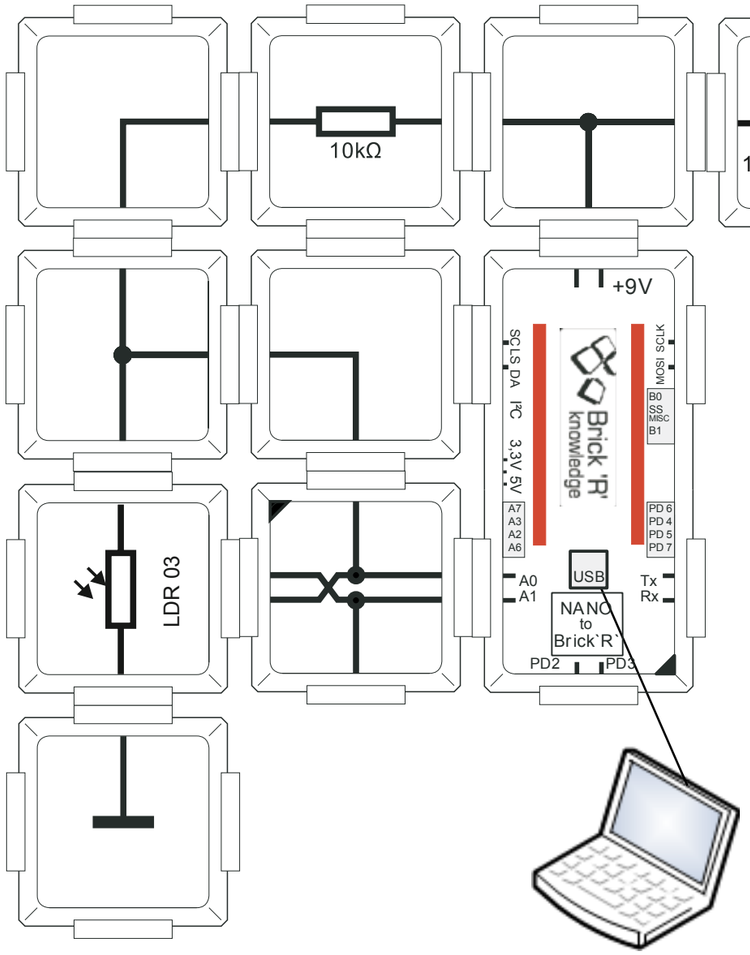
Example output at the terminal (after Ctrl-Sfiift-M)

```
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
```

What happens? If the terminal is activated you should see the following string „9.00V 921“ This means the voltage value after conversion and also the internal value which is returned by the A/D-converter. The list of number is increasing, when you turn the potentiometer the value should be between 0 and the maximum voltage of the battery.

5.3 A/D-converter and light sensitive resistor LDR

The LDR03 is a light sensitive resistor. The value is reduced the more light it gets. In the circuit it forms a voltage divider together with the 10kOhm resistor. The value at the A/D converter is:
Vad = Rldr/(Rldr+R10k)*VBatt
If the resistor is reduced, also the voltage at the A/D-converter is reduced. It's important that the Arduino has a limit of 5V maximum, which is reached when the sensor is darkened too much. Therefore we cannot measure the whole range. The terminal on the PC shows the recalculated resistor values of the LDR. Therefore the formula is first done for calculating the voltage and then an additional formula is used to calculate the resistor. Here we assume the battery voltage to be 9V, which of course affects the calculation. With a value of 12.5kOhm the 5V of the A/D-converter is reached, this means no higher values can be measure without a new circuit. The number of digits is a little bit too large for the precision we use, as the converter only has 10 bit, and 1/1024 is the smallest step. This is a small task for the reader to enhance the program, and show a reduced number of digits.



Attention:
Activate the terminal

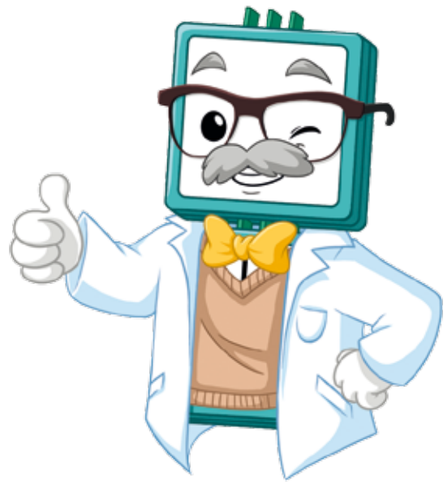
You need to press CTRL-SHIFT-M when the arduino software is active.
A window with the terminal pops into the way.
Attention: set the baudrate to 9600 if it does not show a readable text.

```
// EN_10 AD Converter and LDR03


// CTRL-SHIFT-M for
// serial terminal
// that means ctrl and SHIFT and M
// must be pressed together !

#define PORTAD0 0 // we choose channel 0
// The output gets to the PC, later
// we shall connect display bricks
void setup() { // start
  Serial.begin(9600); // the speed; the BAUDRate
  // for the communication with the PC
  // 9600 Baud = 9600 Bits/Second
}

void loop() { // loop start
  int value; // value from AD converter
  double Vdivider,Rldr; // voltage at the divider of resistance + LDR
  value = analogRead(PORTAD0); // read A0
  // convert in steps from 0..1023 to 5 V max
  Vdivider = value * 5.0 / 1023.0;
  // convert voltage at divider to resistance value R of LDR
  // measure battery before (!) Here we asume 9V;
  // if not 9V the value has to be adapted
  Rldr = (Vdivider*10000.0)/(9.0-Vdivider);
  Serial.print(Rldr); // show resistance at terminal
  Serial.println(„Ohm „); // denomination Ohm in double quotes
}
```

example output:

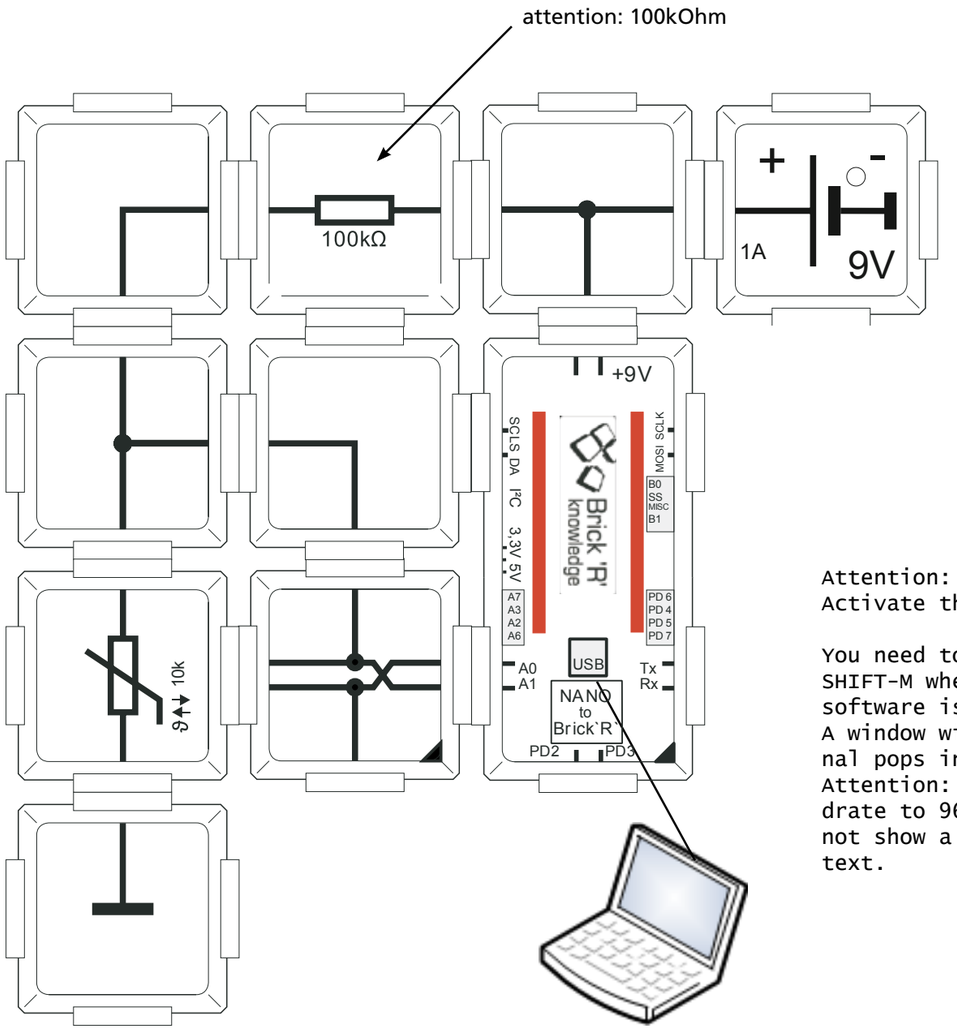
```
5719.65Ohm
5534.00Ohm
5391.17Ohm
5719.65Ohm
5481.76Ohm
5416.95Ohm
5746.54Ohm
5442.80Ohm
5442.80Ohm
5733.08Ohm
5404.05Ohm
5507.83Ohm
```

What happens? If the terminal is activated you should see a character sequence like „5719.65Ohm", on the terminal which is the measured value of the resistor of the LDR03. When darkening the LDR, the value increases, with additional light it should decrease.



5.4 A/D converter - measure of temperature using a NTC

When using a NTC (high-temperature conductor) for the variable resistor, the circuit can be used for measuring temperatures. The resistor value can be determined as we did in the previous example. But now an additional formula is needed to determine the resulting temperature. For NTC it can be approximated:
$R_T = R_N * e^{B(1/T-1/TN)}$ which result as: $R= B *T_N /(B+\ln(R_T/R_N)*T_N)$
T_N is usually 298.15K the reference temperature. This results in a resistor R_N of 10kOhm as measured at the manufacturer. Usually the room temperature of 25 degree C, calculated in K is used.
B is a value given by the manufacturer (usually between 2000K and 4000K), but this can also be determined by a reference measurement. R_T is the measured temperature in K which results in B (also measured with the unit in K). This time we use a resistor of 100kOhm for the voltage divider to expand the measurement range to 0 degree centigrade. For example melting ice cubes can be used as reference for 0 deg C. This allows to get the B value, which is also dependent on the temperature a little bit. With a given table for B the value on the center for the measurement is chosen.



Attention:
Activate the terminal

You need to press CTRL-SHIFT-M when the arduino software is active.
A window with the terminal pops into the way.
Attention: set the baudrate to 9600 if it does not show a readable text.

```
// EN_11 AD converter and NTC

// CTRL-SHIFT-M for
// serial terminal
// that means ctrl and SHIFT and M
// must be pressed together !

#define PORTAD0 0 // we choose channel 0
// The output is shown at the PC, later
// we shall connect display bricks
void setup() { // start
  Serial.begin(9600); // the speed; the BAUDRate
  // for the communication with the PC
  // 9600 Baud = 9600 Bits/Second
}


void loop() { // loop start
  int value; // value from AD converter
  double Vdivider,RNtc; // voltage at the divider at Rntc
  value = analogRead(PORTAD0); // read A0
  // convert in steps from 0..1023 to 5 V max
  Vdivider = value * 5.0 / 1023.0;
  // convert voltage at divider to resistance value R
  // measure battery before (!) Here we asume 9V
  double VBatt = 9.0;
  // if not 9V the value has to be adapted
  double Rdivider = 100000.0; // 100k resistor
  RNtc = (Vdivider*Rdivider)/(VBatt-Vdivider); // resistance
  // calculate temperature
  double B = 3800.0; // for every NTC type different; please investigate
  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);          // show temperature
  Serial.print(„ Grad C „); // denomination in double quotes
  Serial.print(RNtc);       // also show the resistance value
  Serial.println(„Ohm „);   // denomination in double quotes
}
```

What happens? If the terminal is activated, two lines should be displayed. One line with the temperature in degree centigrade and the next line with the measured resistor value. If you heat up the NTC (by hand), the displayed temperature should increase and the resistor value should decrease.

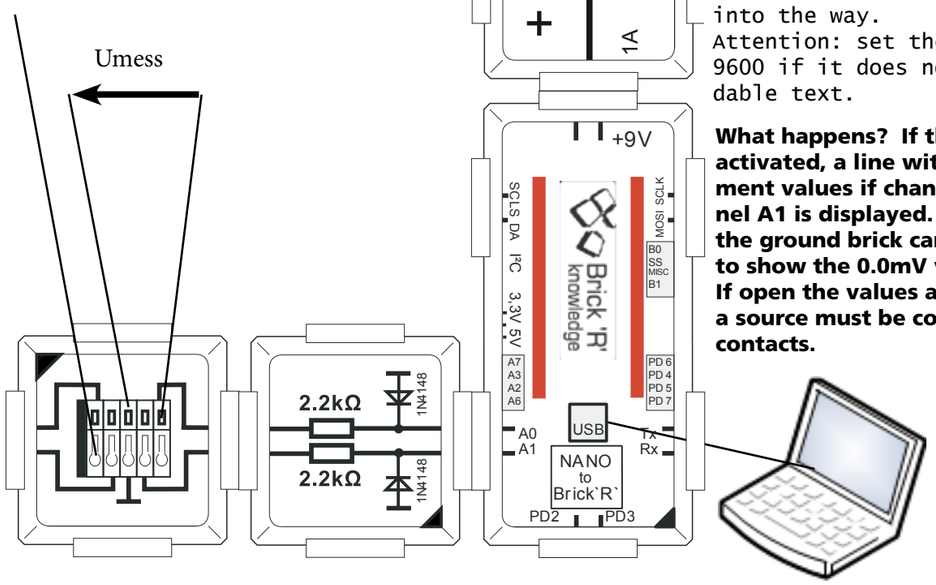
5.5 Voltmeter using an A/D converter - PC interface

Our brick is also very well suited to become a voltmeter. Only some measurement cables need to be connected. Therefore we can use the universal contact brick. Here we use both A/D-converter channels A0 and A1. The program calculates the measured voltage and sends it to teh PC for display. We use an additional new brick, which contains two resistors and two protection diodes. The resistor limits the current if a voltage higher than 5V is connected to the port (of course not higher than a certain maximum, e.g. 9V). The Diodes protect against voltages less than 0V by clamping them to 0V. This will happen if you reverse the measured voltage at Umess. The Nano brick contains an additional 100Ohm resistor at each input. Together with the external diodes it helps that only the external diodes are activated on a negative voltage and not the diodes which are also present internal in the Atmet processor chip. With 9V a maximum current of 4.1mA flows through the 2.2kOhm reistsor which is safe for the input. The protection also changes the measurement a little bit which depends on the impedance of the measured object. This might be taken into account. This also makes it necessary to do a calibartion if you need more exact results.
For the calibration you need a defined voltage source. Then a correction factor can be calculated, which is introduced into the formula. You need a defined voltage source or an adjustable voltage source and a calibarted voltmeter to do this. If for example a 2000mV source shows a value of 2100mV, a correction factor of 2/2.1 is needed, which is then multiplied into the formula for voltage1= or voltage2=. For example

voltage1 = value1 * 5000.0 / 1023.0 * 2.0 / 2.1;

It's also possible to precalculate the constants into one, but usually the compiler does this for us and we have a more readable source code. Now question is about the resolution. We have a 10 bit A/D-converter, for one bit this is 1/1024. This means for the lowest possible voltage rounded: Umin = 5.0V / 1024 = 4.9mV

Universal contact brick: Pressing a small screwdriver in the slots above opens the contact and a wire can be inserted from the side. After release of the screwdriver pressure the wire is fixed.




```
Attention:
Activate the terminal

You need to press CTRL-SHIFT-
M when the arduino software is
active.
A window with the terminal pops
into the way.
Attention: set the baudrate to
9600 if it does not show a rea-
dable text.
```

What happens? If the terminal is activated, a line with the measurement values if channel A0 and channel A1 is displayed. For a quick test the ground brick can be used instead to show the 0.0mV values as result. If open the values are undefined and a source must be connected to the contacts.


```
// EN_12 AD converter as Voltmeter

// CTRL-SHIFT-M for
// serial terminal
// that means ctrl and SHIFT and M
// must be pressed together !

#define PORTAD0 0 // channel 0 at A0
#define PORTAD1 1 // cahnnel 1 at A1

// The output is shown at the PC, later
// we shall connect display bricks
void setup() { // start
  Serial.begin(9600); // the speed; the BAUDRate
  // for the communication with the PC
  // 9600 Baud = 9600 Bits/Second
}

void loop() { // loop start
  int value1,value2; // values from two AD converter channels
  double Voltage1,Voltage2; // values vonverted to voltage
  value1 = analogRead(PORTAD0); // read A0
  value2 = analogRead(PORTAD1); // read A1
  // convert value to voltage
  // convert in steps from 0..1023 to 5000 max
  Voltage1 = value1 * 5000.0 / 1023.0; // if necessary use
  Voltage2 = value2 * 5000.0 / 1023.0; // correction factor
  Serial.print(Voltage1); // show voltage at channel 0
  Serial.print(„mV „);    // denomination in double quotes
  Serial.print(Voltage2); // show voltage at channel 1
  Serial.println(„mV „);  // denomination in double quotes
}
```

Output:

```
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
```

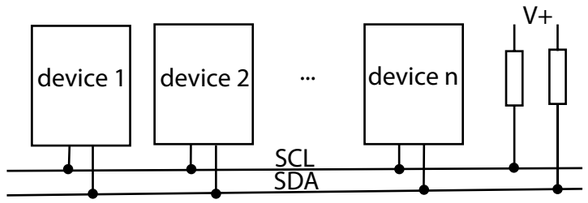
6. I2C BUS

6.1 I2C bus principle and commands

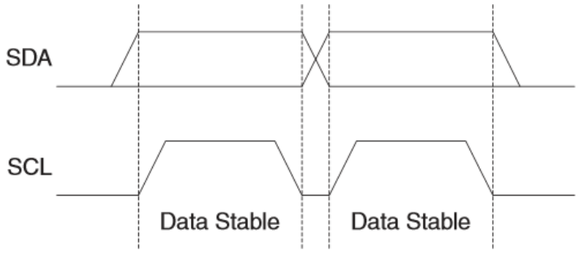
The I2C bus is a serial interface using only two wires: the clock line called SCL and the data line called SDA. These lines are used in a bidirectional way. The direction defines a master and a slave. In our case the Nano is the master and the other bricks are used as slave.
The I2C bricks use I2C addresses for activation. There are 128 possible different devices on one I2C bus. Some of the devices use several addresses. Some of our bricks have a small DIL switch at the bottom side after lifting the case where the exact address range can be set. This allows multiple bricks with the same device to be used on one I2C bus.
The I2C bus can handle different speeds, which depends on the specific device itself:

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.

Most microcontroler can only handle the first two speeds, as for example the Nano, some also the third. Same applies for the slave device, they must fit together. It's always possible to use a reduced speed but not the other way round. The master defines the clock with the SCL line and the data is transfered via the SDA line from either master to slave or reverse.

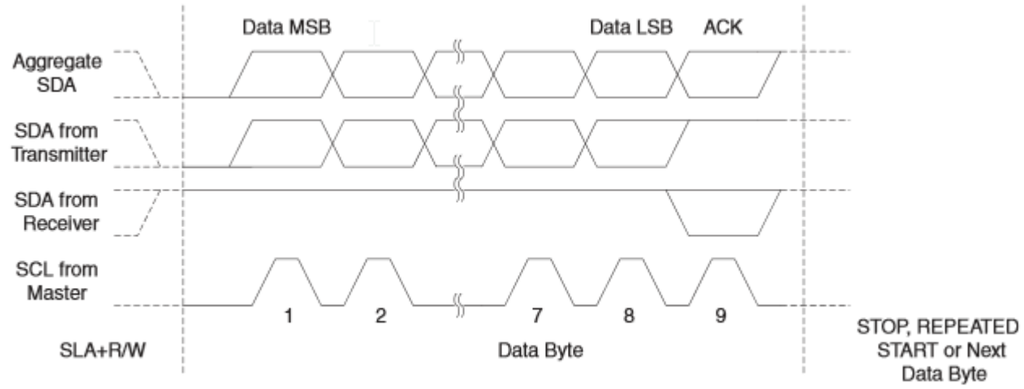


If each device uses one address a maximum of 128 different devices is possible. Here shown with device 1 to n. All devices are wired together at SCL and SDA. There are two resistors to pull up the signal, to either 5V as with the Nano or 3.3V or any other different voltage (depends on the I2C system). The resistors can be around several kiloohms, they are built into the Nano brick already also a level converter for switching between 3.3V and 5V.



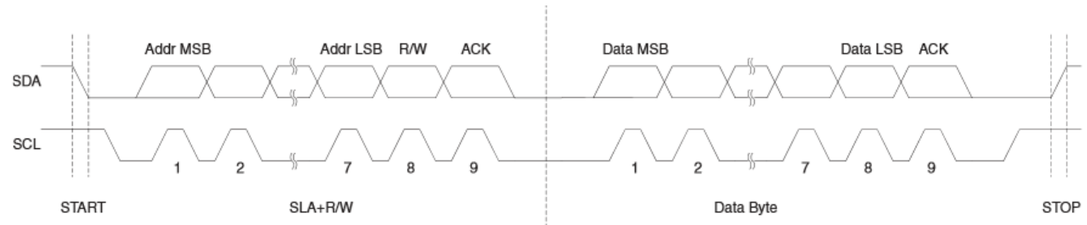
from the Atmel 328 datasheet

The clock defines when the data is stable. Above you can see that this is the case, when there is a high level. The receiver can read the data at this point and use it. The master defines the clock, and either master set the data, or it checks the values.



from the Atmel 328 datasheet

The above diagram shows a split of master, transmitter and receiver. The master assignes the clock. The synchronisation is important. The receiver, if either the master or slave, sets an ACK-signal at the end by pulling low the data line. Because this is a so called „wired or", the response of one slave is sufficient to define the signal.
The SDA as on the wire is shown as aggregate SDA above.

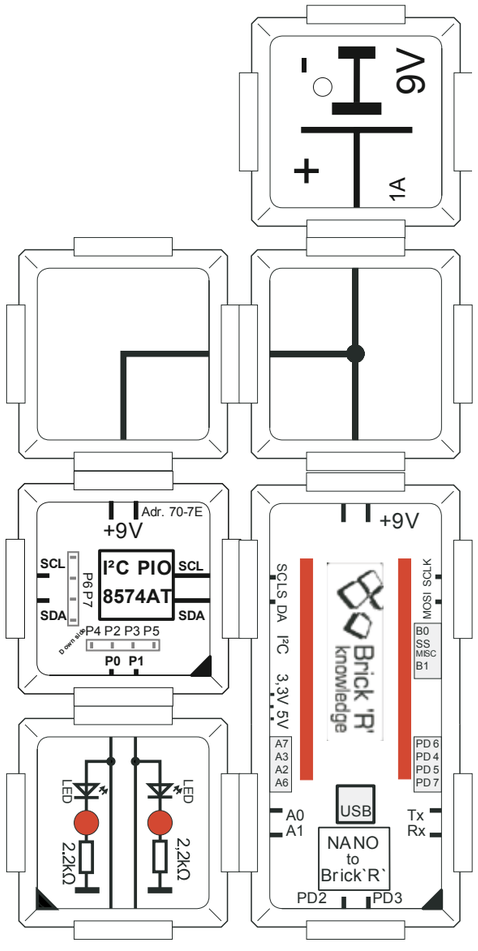


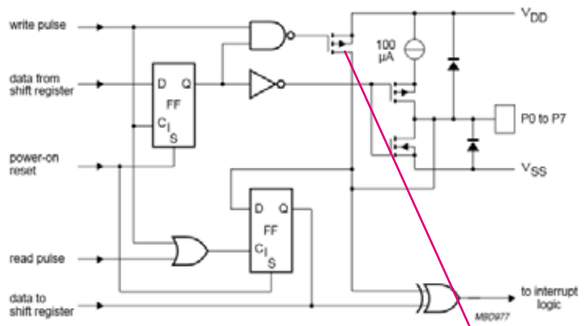
from the Atmel 328 datasheet

Here the example of a full datatransfer. First a paket is sent containing the address. The address uses 7 bits. In addition to the 7 bits for the address, there is one bit at position 0 with the read/write flag. This defines if the slave starts a read or a write cycle. The receiver compares the address with its internal address and if there is a match, the slave responds with an ACK signal. This means the SDA is pulled to low at the ACK position. Now the datatransfer starts. After the datatransfer a stop cycle is added. For this purpose the clock line is pulled high (e.g. released due to the active low logic), and then the SDA line released. Now other masters can take over the bus, as the I2C is a multi master bus. If both signals are high, SDA and SCL the bus is not occupied and can be used.
The I2C is easy to use with the Arduino library, as you will see in the next examples. Important is the sequence of calles.

6.2 I2C bus and IO Port

The PCF8574 is a so called I/O extender. It can handle 8 I/O ports, which can be programmed either as output but also be used as input. There are two versions of the PCF8574, the PCF8574T (40-4E) and the PCF8574AT (70-7E), they can use different address ranges. If you open the bottom side of the brick, there is DIL switch to assign the sub address which is possible for the device for example 40,42,44,46,48,4A,4C and 4E for the standard device. If you remove both covers, be careful on the orientation, when later placing it back again. There is a small triangle printed on the top which shows the correct orientation and should be aligned to the chamfered side of the pcb board below.
For the I2C we use a 8 bit counting. The I2C only uses 7 bit, of which the lower is a write bit. Therefore our address ranges are counted in steps of two. The DIL switch has four positions, but the number 4 position is not used. With the remaining 3 positions that means 8 different addresses to be chosen.
The output stage of the PCF8574 has a special construction which makes it bidirectional. There is a open drain stage, which can handle quite huge loads towards ground, but only 100 uA towards 5V. In our example the LEDs are not very bright. If turning to high a small pulse drives the load towards high, this allows the port to be used as input port without any additional register for the direction.
To drive the LEDs brighter a different approch will be shown next.





Copied from the datasheet of the PCF8574.
Important is the logic at the right side with the both MOSFETs and the 100uA current source. The single p-MOSFET shortly connects the output to VDD (5V) which allows for a fast output change when the register is set to 1. But due to the weak current source, it can be also used as input.

DIl switch

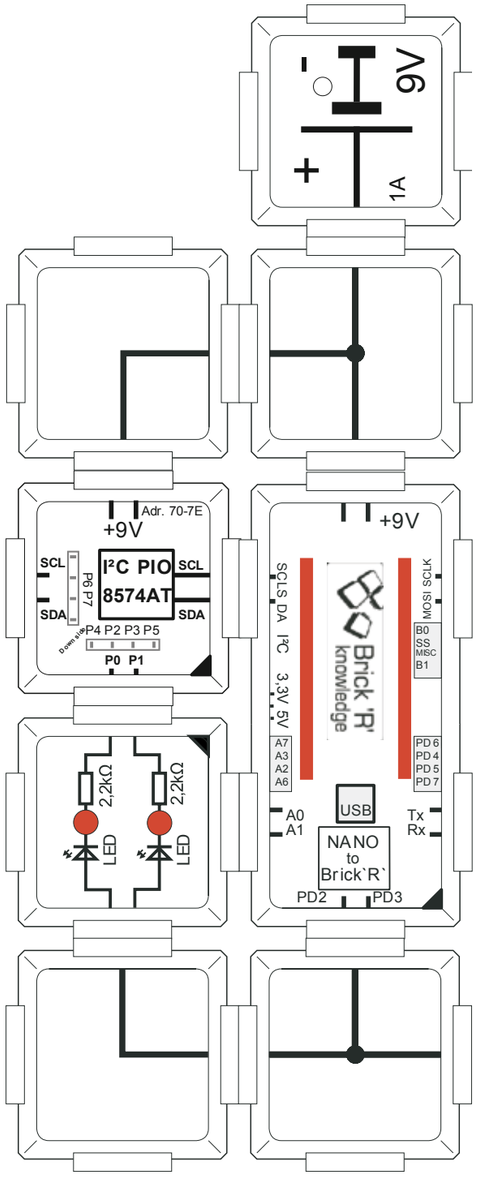
1234	8574T	8574AT
000x	40h	70h
001x	42h	72h
010x	44h	74h
011x	46h	76h
100x	48h	78h
101x	4Ah	7Ah
110x	4Ch	7Ch
111x	4Eh	7Eh

```
// EN_13 I2C - IO Port 8574AT and 8574T

#include <Wire.h> // load definitions for I2C
#include <avr/pgmspace.h> // further definitions

// all adresses for the 8574xx bricks:
#define i2cIO8574_0 (0x40>>1) // trick for smart work with bytes
#define i2cIO8574_1 (0x42>>1) // instead of 7 bits
#define i2cIO8574_2 (0x44>>1) // last bit is the R/W,
#define i2cIO8574_3 (0x46>>1) // which is added by the Arduino library
#define i2cIO8574_4 (0x48>>1) //
#define i2cIO8574_5 (0x4A>>1) // We define here all the ranges,
#define i2cIO8574_6 (0x4C>>1) // which can be set by the bricks.
#define i2cIO8574_7 (0x4E>>1) //

#define i2cIO8574A_0 (0x70>>1) // The series PCF8474AT
#define i2cIO8574A_1 (0x72>>1) // starts at address
#define i2cIO8574A_2 (0x74>>1) // 0x70 = 70 sedecimal
#define i2cIO8574A_3 (0x76>>1) // 01110000 binary
#define i2cIO8574A_4 (0x78>>1) // or internal 0111000
#define i2cIO8574A_5 (0x7A>>1) // though 0111xxx
#define i2cIO8574A_6 (0x7C>>1) // with x for the
#define i2cIO8574A_7 (0x7E>>1) // DIL or DIP switch positions

// ATTENTION: set here the assignment according to switch positions

#define myi2cIOadr i2cIO8574A_0 // ENTER HERE THE ACCORDING ADDRESS

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

void loop() {
  Wire.beginTransmission(myi2cIOadr); // starting sequence for the I2C address
  Wire.write(0x55) ;                  // alternate IO Ports to 01010101
  Wire.endTransmission();             // set stop condition for I2C
  delay(100);                         // 100 ms delay.
  Wire.beginTransmission(myi2cIOadr); // once again the same address
  Wire.write(0xaa) ;                  // alternate IO Port at 10101010
  Wire.endTransmission();             // set stop condition (stop bit)
  delay(100);                         // 100 ms delay
}
```

What happens? The LED blink alternating with a delay of 1 second. If they don'nt alternate check the address settings at the DIL switch first!

6.3 I2C Bus IO port using active low

As the 8574 uses open drain drivers with a very weak pullup (100uA) to +5V it's better to change the circuit for an active low pulling. For this the cathode of the LEDs are now connected to the driver output. Therefore we use the dual LED brick with the series LEDs. But now the Anode needs a +5V to switch on. Do not use the 9V source, it can dammage the 8574. We use a small trick to get the 5V, we just use the output port from the Nano brick programmed as pullup and set to high. Two of them can source enough current for the two LEDs.



DIL switch:

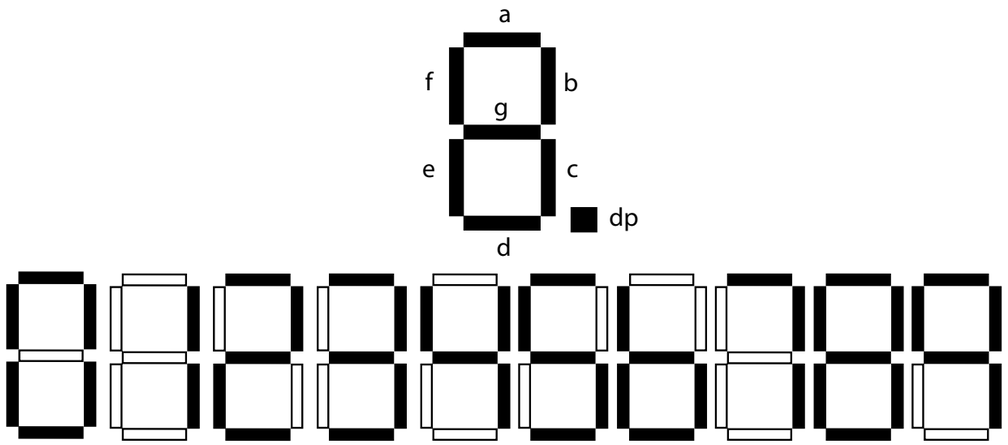
1234	8574T	8574AT
000x	40h	70h
001x	42h	72h
010x	44h	74h
011x	46h	76h
100x	48h	78h
101x	4Ah	7Ah
110x	4Ch	7Ch
111x	4Eh	7Eh

What happens? The LED blink alternating with a delay of 1 second. If they don'nt alternate check the address settings at the DIL switch first!

```
// EN_14 I2C - IO Port 8574AT and 8574T Lowpulse

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

#define i2cIO8574_0 (0x40>>1) // trick for smart work with bytes
#define i2cIO8574_1 (0x42>>1) // instead of 7 bits
#define i2cIO8574_2 (0x44>>1) // last bit is the R/W,
#define i2cIO8574_3 (0x46>>1) // which is added by the Arduino library
#define i2cIO8574_4 (0x48>>1)
#define i2cIO8574_5 (0x4A>>1) // We define here all the ranges,
#define i2cIO8574_6 (0x4C>>1) // which can be set by the bricks.
#define i2cIO8574_7 (0x4E>>1)

#define i2cIO8574A_0 (0x70>>1) // The series PCF8474AT
#define i2cIO8574A_1 (0x72>>1) // starts at address
#define i2cIO8574A_2 (0x74>>1) // 0x70 = 70 sedecimal
#define i2cIO8574A_3 (0x76>>1) // 01110000 binary
#define i2cIO8574A_4 (0x78>>1) // or internal 0111000
#define i2cIO8574A_5 (0x7A>>1) // though 0111xxx
#define i2cIO8574A_6 (0x7C>>1) // with x for the
#define i2cIO8574A_7 (0x7E>>1) // DIL or DIP switch positions

// ATTENTION: set here the assignment according to switch positions

#define myi2cIOadr i2cIO8574A_0 // we take the 0x70

#define PULLUP2 2 // small trick at port 2 und 3
#define PULLUP3 3 // we get 5V (40mA max!)

void setup() {
  pinMode(PULLUP2, OUTPUT);   // we need PD2 and PD3 as output
  pinMode(PULLUP3, OUTPUT);
  // they have to be set to high to deliver 5V
  digitalWrite(PULLUP2,HIGH); // quick, quick, otherwise short circuit
  digitalWrite(PULLUP3,HIGH);
  Wire.begin();               // enable I2C
}

void loop() {
  Wire.beginTransmission(myi2cIOadr); // starting sequence for the I2C address
  Wire.write(0x55) ;                  // alternate IO Ports to 01010101
  Wire.endTransmission();             // end: set stop condition for I2C
  delay(100);                         // 100 ms delay.
  Wire.beginTransmission(myi2cIOadr); // once again the same address
  Wire.write(0xaa) ;                  // alternate IO Port at 10101010
  Wire.endTransmission();             // and again stop bit
  delay(100);                         // again 100 ms delay for blinking effect
}
```

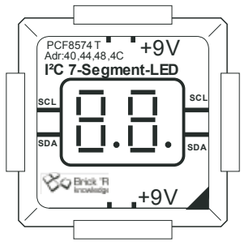
6.4 The seven segment display - principle

At the beginning of computing, it was a goal to display numbers. The simplest form was to use 10 bulbs numbered from 0 to 9. Later the bulbs were placed behind some glas plates in which there were drills in the shape of numbers 0..9. At the same time the so called Nixi tubes arrised. The numbers were formed by wires and with a high voltage in a protective gas, they numbers illuminate.
Soon the idea arrises to split the numbers into segments. With at least 7 segements it's possible to display all numbers between 0 and 9. The first display uses small bulbs which glow, but then the LEDs come up and things get easier. Each segment contains a small LED that illuminates the bar. The segments are names from „a" to „h". Optional a single dot LED is added for the decimal point.



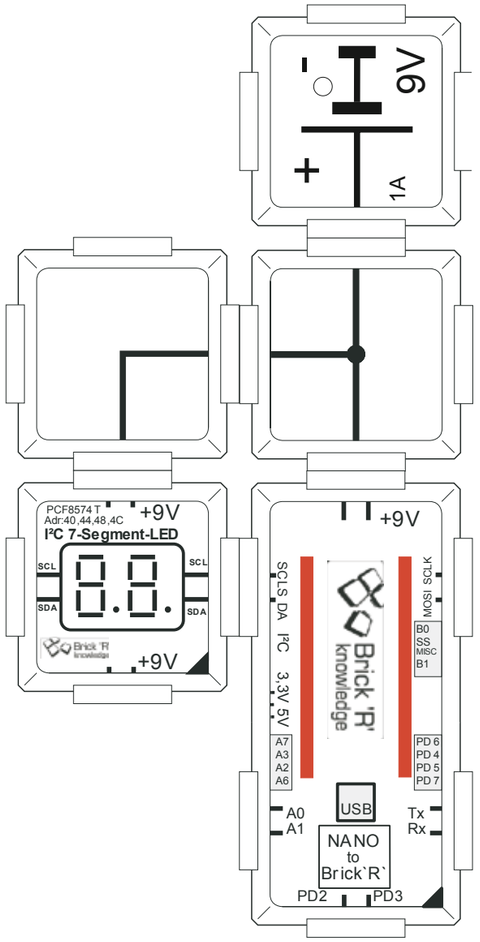
Here the numbers from 0 to 9. Its also possible to display alphanumerics with some phantasy (try it , A,E,L,o then it gets cryptic).
There are also 14 and 16 segment display for better alphanumeric display, but they become very rare, as there are much more elegant ways for such displays for example with our OLED.

Our brick contains two such 7-segment-displays.
There are two 8574T used to control the LEDs, that means it uses 16 bits as each can control 8 I/O ports. With a small switch you can set the I2C address range, for a maximum of 4 such bricks on one single I2C bus.



6.5 Seven segment display - using I2C - principle

Let's implement as simple circuit using the two 7-segment-displays. Attention, it might be necessary to adjust the preset address range. The switches are on the bottom side of the bricks after removing the cover (if set back, please carfully check the orientation).
We have prepared a small library for the Nano brick, in which all assignments to the segments are stored in a table. In addition to the numbers 0-9 also all letters from A-Z get a symbol as good as possible with seven segments. The table is addressed via an index which is according to the character code. But we have prepared some subroutines for your convenience. With display_seg1x() a signle segment is displayed. The subprogram gets the I2C address as paramater and the charcater code in ASCII. Using display_seg1xbin () single segments can also be displayed using a binary code. As we use two parallel ports for the two 7-segment-displays, two addresses of the I2C are used for a total display of 2 characters. Take a look at our programming examples to understand the usage.
The example program just displays the found I2C address block and shows it. So its easier to check the switch position. Therefore all possible combinations are tried.

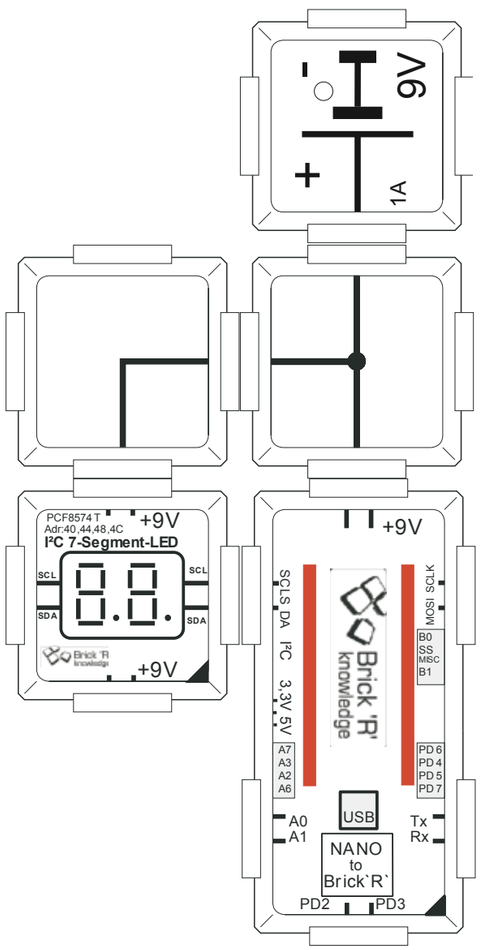


What happens? On the display, two letters are displayed. They are the address of the I2C used for the display. It could be 40, 44, 48, 4C or in the range 70, 74, 78, 7C depending on the chip set used (8574T or 8574AT). The exact address depends also on the DIL switch position at the bottom of the brick (after removing the cover).

```
// EN_15 SevenSegementDisplay as I2C Brick
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1) // 7 bit
#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)

// optional built in: 8574AT
#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)

// setup of the 7 segment assignments to the
// bits, 0x80 is for the DOT
//  ************************
//          01
//       20    02
//          40
//       10    04
//          08
//                80
//  ************************

// conversion table ASCII -> 7 segment
// OFFSET Asciicode 32..5F corresponds to
// Space up to 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
};

// convert ASCII Code to7-Segment table
index
unsigned int get_7seg(unsigned char
asciicode)
{
    // convert 0..255 to
    // 7 segment table index
    // only digits an capital letters
    // 20..5F
    // the rest will be mapped to the above
    asciicode = asciicode & 0x7f;     // mask 7 bit only
    if (asciicode < 0x20) return (0); // no special characters
    if (asciicode >= 0x60) asciicode = asciicode - 0x20; // map lower case letters
    return((~siebensegtable[asciicode-0x20])&0xff);      // return table index
}

// Display a single ASCCII character, which will be
// handed over as a character. Output over the 7 segment Brick
// In addition handover the segment addess as parameter.
void display_seg1x(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C address begin
  Wire.write(get_7seg(ch1)); // take table index and write it
  Wire.endTransmission();    // end I2C
}

// Output without conversion, if own characters shall be used.
// Parameter is the binary code
void display_seg1xbin(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C address begin
  Wire.write(ch1); // write binary code direct to port
  Wire.endTransmission(); // end I2C
}

// Start only needed once
void setup() {
  Wire.begin(); // initialize I2C library
}

void loop() {
   // write all potential addresses of display 8574T,
   // to see which address is occupied
  display_seg1x(i2cseg7x2amsb1,'4'); // write own address
  display_seg1x(i2cseg7x2alsb1,'0'); //
  display_seg1x(i2cseg7x2bmsb1,'4'); // they are always COUPLES
  display_seg1x(i2cseg7x2blsb1,'4'); // two commands for one Brick
  display_seg1x(i2cseg7x2cmsb1,'4');
  display_seg1x(i2cseg7x2clsb1,'8'); // from 40 up to 4C
  display_seg1x(i2cseg7x2dmsb1,'4');
  display_seg1x(i2cseg7x2dlsb1,'C');
  // if 8574AT exists, write also to that
  display_seg1x(i2cseg7x2amsb2,'7'); // write own address
  display_seg1x(i2cseg7x2alsb2,'0');
  display_seg1x(i2cseg7x2bmsb2,'7'); // here
  display_seg1x(i2cseg7x2blsb2,'4'); // from 70 up to 7C
  display_seg1x(i2cseg7x2cmsb2,'7');
  display_seg1x(i2cseg7x2clsb2,'8');
  display_seg1x(i2cseg7x2dmsb2,'7');
  display_seg1x(i2cseg7x2dlsb2,'C');
}
```

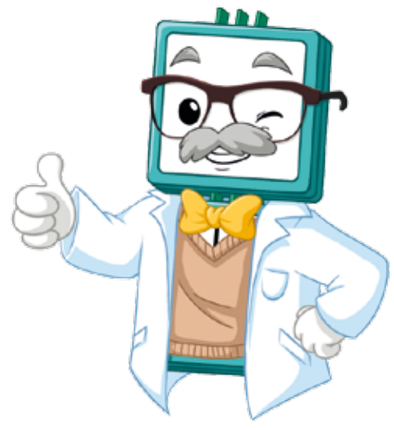
6.6 Seven segment display - simple counter

Let's implement a simple counter in this example. We have a variable called „counter" for saving the counter value. Every 500ms the counter value is increased. With one display brick we can show a maximum of „99" as value. For this reason the counter value is compared to 99 and if larger then a value of 0 is set to the counter variable. The counter starts again with 0 in this case.
The loop takes a little bit longer than 500ms, though the call „delay(500)" is quite exact, but the other instructions do sum up, especially the communication with the I2C. For higher precision a timer might be needed, but this is just a hint.




```
// EN_16 SevenSegmentDisplay count --  display on I2C Brick

#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
..............
see appendix or previous example EN_15 for the code inbetween
..............

// Start only needed once
void setup() {
  Wire.begin(); // initialize I2C library
}

void loop() {             // start loop
  char buffer[10];        // use characterbuffer
  static int counter = 0; // static variable for counter starting at 0
  sprintf(buffer,"%02d",counter++); // convert integer to character
  if (counter >99) counter = 0;     // counter running from 0..99 then start again
  // show counter as two digits from buffer 0 and 1
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb (most significant Byte) character
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb character
  delay(500); // count up every 500ms (approximately)
} // end of the loop
```

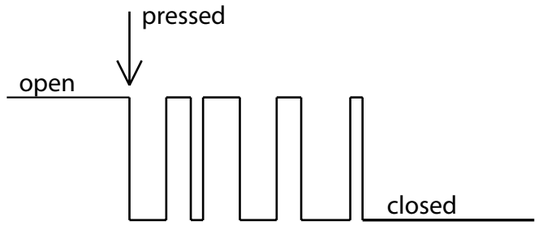
What happens? The 7-segment-display shows the sequence 00,01,02...99. Every 1/2 second the counter value is changed, after 99 the value of 00 is displayed again.



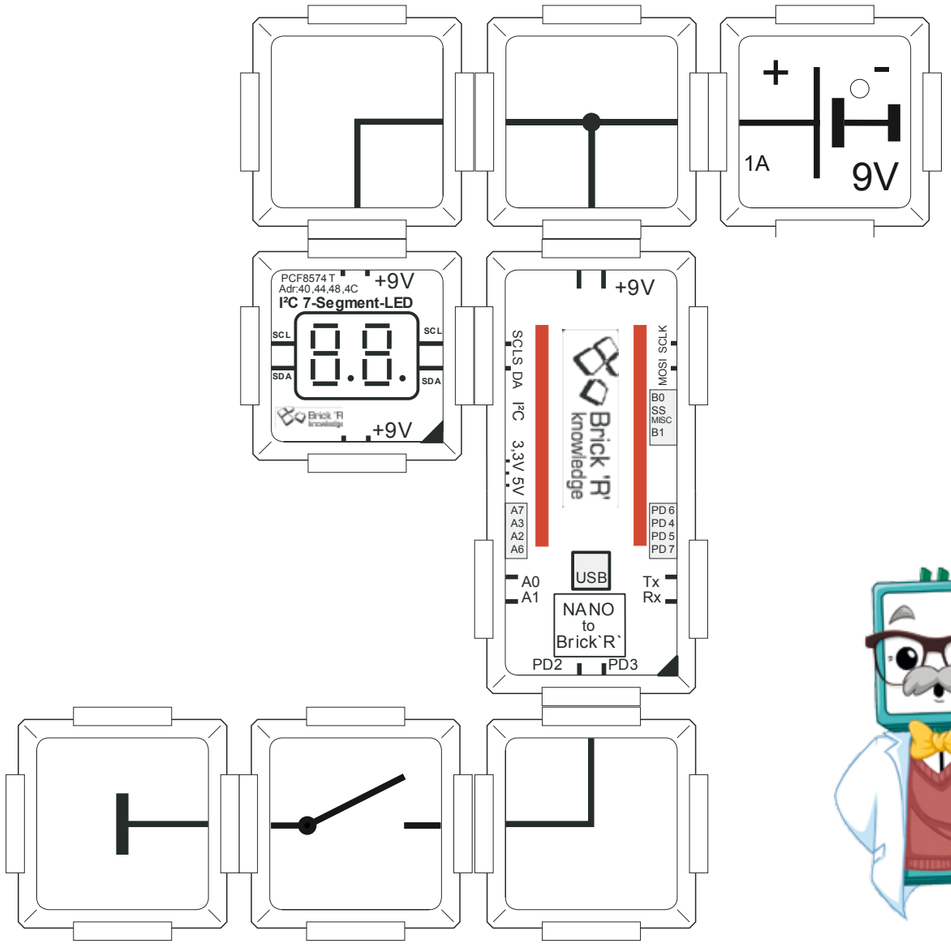
7. Push buttons & bouncing

7.1 Push buttons can bounce

When you press a mechanical push button, the contacts bounce back and forth. This results in multiple open and close contacts until finally the contact is in a close position. Same happens when opening such a contact.



This program is sensitive to contact bouncing. Sometimes if you press the button the counter increments more than once. This depends very much on the button currently built into the brick. Sometimes the contacts are so fast bouncing that the processor is not registering this, then you can take an external wire to try it. The bounce time lasts only some miliseconds.

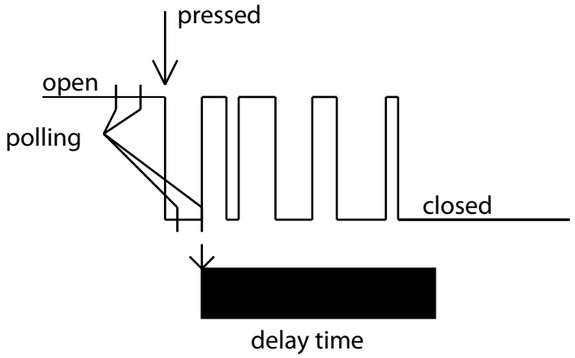


```
// EN_17 buttons can bounce; count it
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... as before  ...


// new code:
#define PORTTASTE 2 // connect button to PD2

// initialize I2C and define button
void setup() {
  Wire.begin(); // initialize I2C
  pinMode(PORTTASTE,INPUT_PULLUP); // connect button with pullup
}

void loop() {             // start loop
  char buffer[10];        // buffer for count value in ASCII
  static int counter = 0; // counter
  sprintf(buffer,"%02d",counter); // convert counter to ASCII and write to buffer
  // read button, it can bounce
  if (digitalRead(PORTTASTE)==LOW) { // if button is pushed signal high -> low
    while (digitalRead(PORTTASTE)==LOW) { // loop while buttonn is pressed
     // that means wait until Low -> High ransition, which means button is released
!
     // as long as button is hold nothing happens
    }
   counter++; // count not until released - but bouncing is counted also
  }
  if (counter > 99) counter = 0;     // counter running from 0..99 then start again
  // show counter as two digits from buffer 0 and 1
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb (most significant byte) character
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb character
} // end of the loop
```

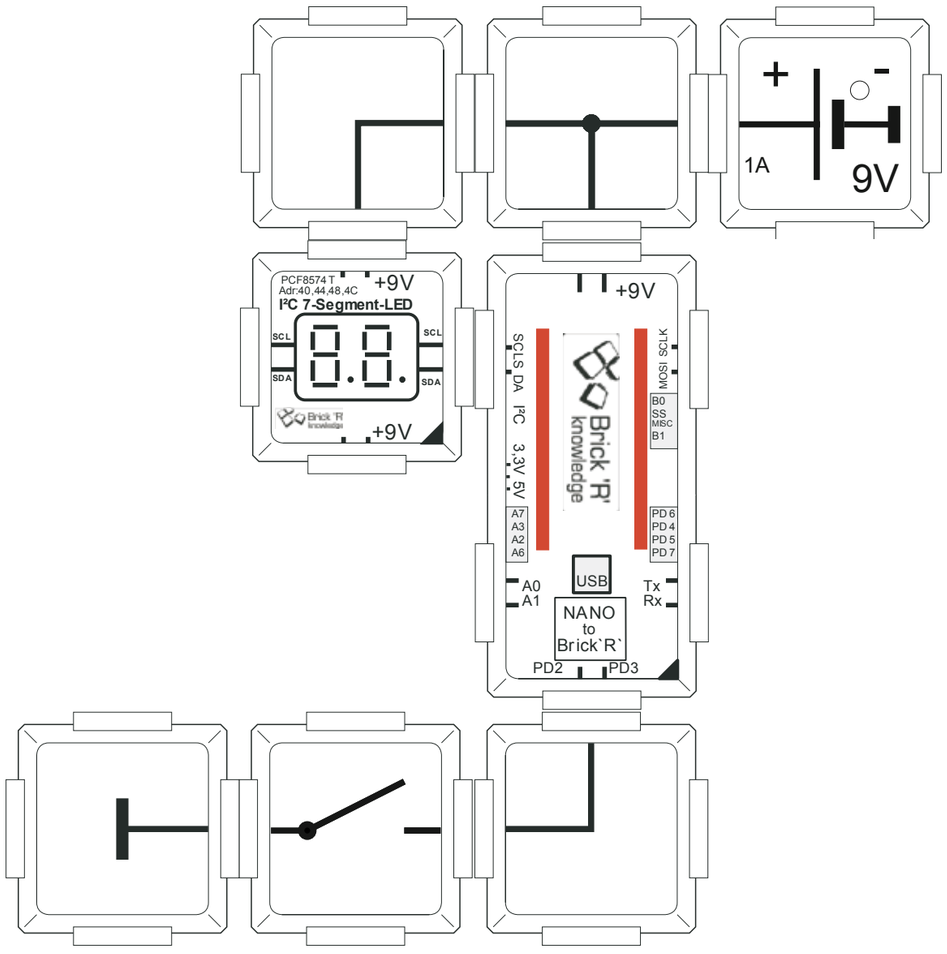
What happens? If the push button is pressed, the counter increments, but multiple counts are possible for example: 00, 03, 04, 07, 09 ... Depends very much on the quality of the push button. A wire can be used to short the button if necessary to see the effect.

7.2 Debouncing of mechanical buttons by software

For debouncing a delay can used when polling the contacts. Therefore a small delay time is waited before checking again the state of the button.



Debouncing algorithm:
First the signal is checked for the high to low transistion, then there is a wait until the signal returns to high again (that means looks like the button is released). But then a wait time is introduced for example 40ms until the next poll is done again for a high to low transisition. The delay time is dependant of the used push button and must be checked by experiment. The delay prevents polling to early and thus prevents a false counting.

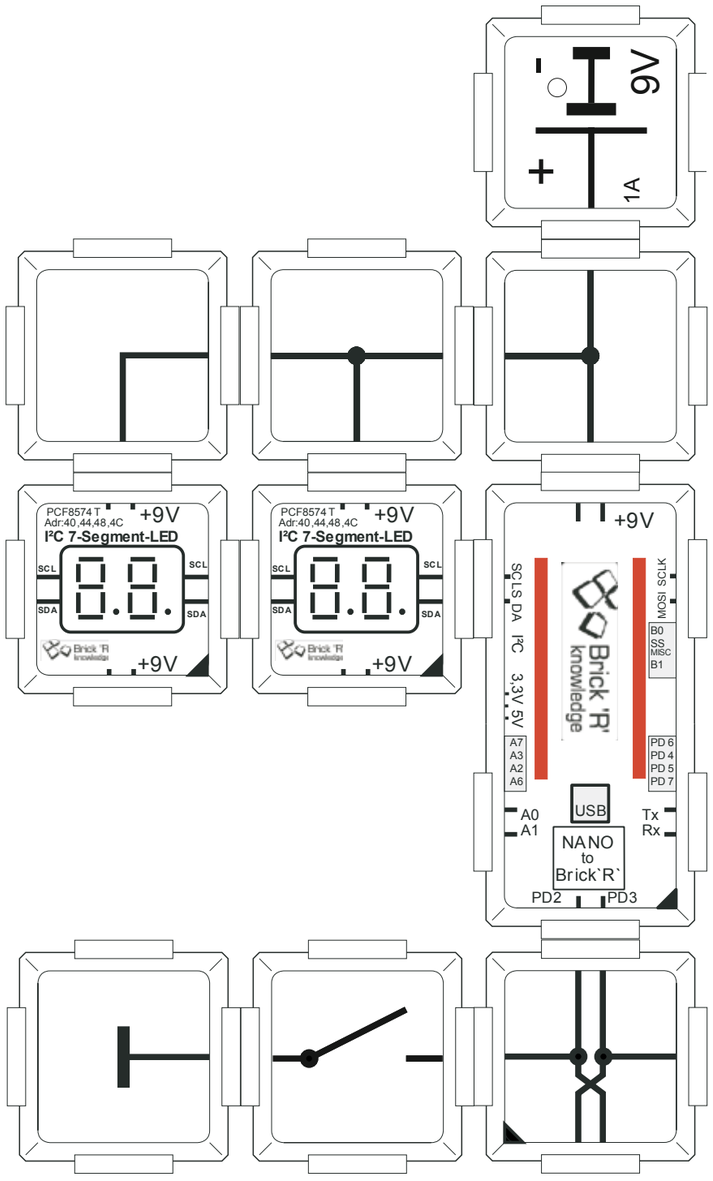



```
// EN_18_buttons_debounce_I2C - 7 segment display as I2C Brick
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... as before ...


// new code:
#define PORTTASTE 2 // connect button to PD2

void setup() { // initialize (non-recurring)
  Wire.begin(); // initialize I2C
  pinMode(PORTTASTE,INPUT_PULLUP); // connect button with pullup
}

void loop() { // start loop
  char buffer[10];        // buffer for count value in ASCII
  static int counter = 0; // static variable counter 0..99
  sprintf(buffer,"%02d",counter); // convert counter to two digit ASCII and write to
buffer
  // read button; it can bounce
  if (digitalRead(PORTTASTE)==LOW) { // if button is pushed signal high -> low
    while (digitalRead(PORTTASTE)==LOW) { // loop while buttonn is pressed
      // that means wait for low -> high ransition, which means button is released !
      // as long as button is hold nothing happens
      // could be done after delay, but is not critical
    }
    counter++; // first increment the counter
    delay(40); // but then wait some time until bouncing ends
  } // end of the test for high -> low transition
  if (counter > 99) counter = 0; // counter running from 0..99 then start again
  // show counter as two digits from buffer 0 and 1
  display_seg1x(i2cseg7x2amsb1,buffer[0]); // msb (most significant byte) character
  display_seg1x(i2cseg7x2alsb1,buffer[1]); // lsb character
} // end of the loop
```

What happens? If the push button is pressed, the counter value should increase only by one - for each time the button is pressed. That is a sequence of 00, 01, 02 ... should be displayed.

7.3 Sevensegment display - enhanced counter

Now let's extend our counter for counting from 0 to 9999. Therefore we now use two 7-segment-display bricks. Not much code change is necessary for this. The counter value is converted to the 4 digits using the „sprintf()" call, but this time with the format „%04d" to get 4 digits with a zero displayed for pre zeros. Now the button is also polled, but this time a little bit different from the previous example. The delay is split into two halfs, this gets more savety into the debouncing. The counter increases towards 9999 but then resets to 0000 again. The if condition is also adjusted for the new overflow.
Attention: be sure the 2nd display has 42 as new I2C address and be on the left side. Otherwise a wrong display results.

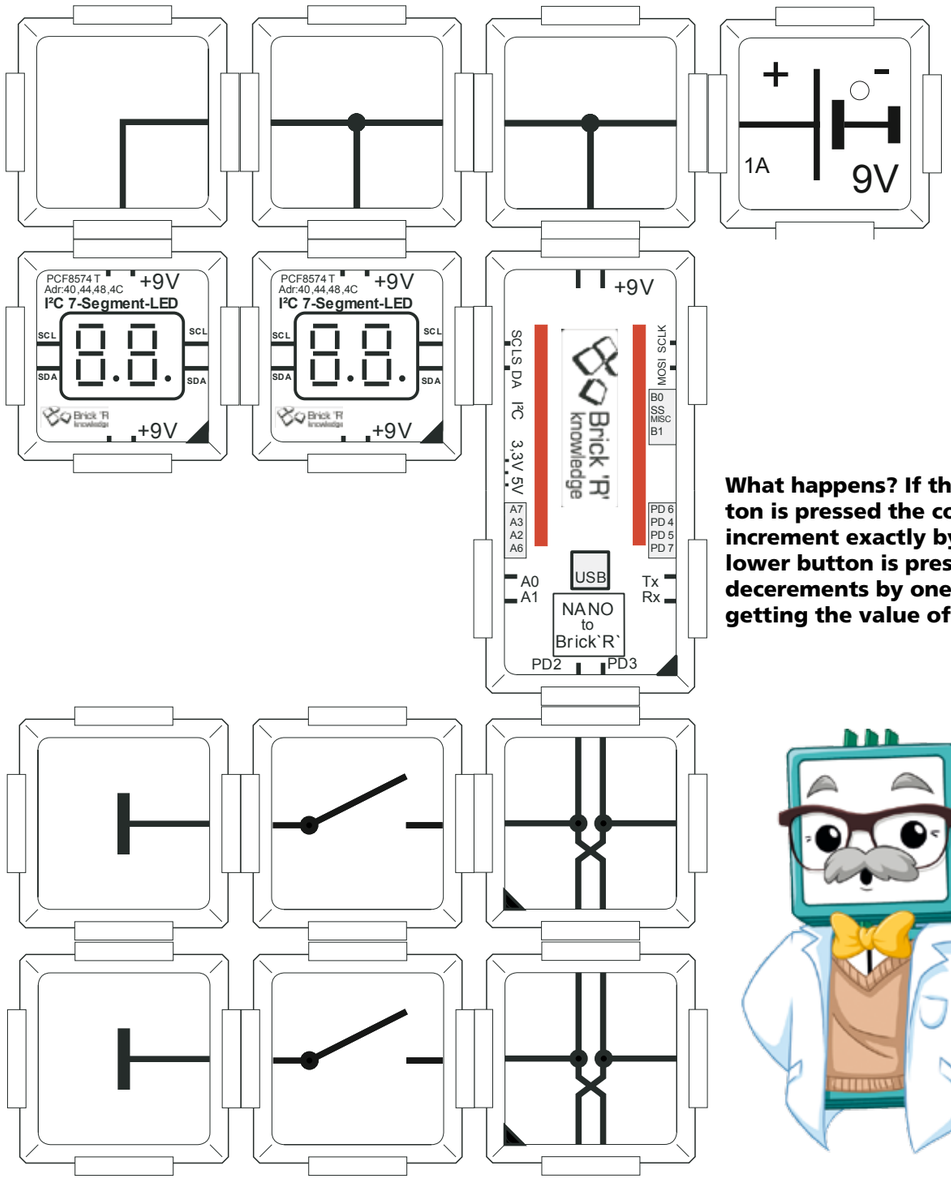


```
// EN_19 Seven segment counter extended.
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... as before ...


// new code:
#define PORTTASTE 2 // connect button to PD2

// initialize (non-recurring)
void setup() {
  Wire.begin(); // initialize I2C
  pinMode(PORTTASTE,INPUT_PULLUP); // connect button with pullup resistance
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCII
  static int counter = 0; // static variable counter 0..9999
  sprintf(buffer,"%04d",counter);
    // convert counter to 4 digit ASCII and write to buffer
  // read button, it can bounce
  if (digitalRead(PORTTASTE)==LOW) { // if button is pushed signal high -> low
    delay(20); // in this example wait 20 ms
    // already here until button is not bouncing any more
    // it depends on used button if time is enough
    while (digitalRead(PORTTASTE)==LOW) { // loop while buttonn is pressed
      // that means wait for low -> high ransition,
      // which means button is released !
      // as long as button is hold nothing happens
    } // go further after button is released
    counter++; // after release increment the counter
    delay(20); // and wait again (not important, but sum of both delays
    // at activate and release of the button should be 40ms)
  }
  if (counter > 9999) counter = 0;
     // counter running from 0..99999 then start again
  // show counter as 4 digits from buffer 0..3
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb (most significant byte) charac-
ter to most left display
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // third position from left
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb (least significant byte) cha-
racter to most right display
} // end of the loop
```

What happens? If the button is pressed, the counter increments by one from 0000, 0001, 0002 ... 9999 and then again with 0000.

7.4 Seven segment display - with enhanced up- and downcounter

With this circuit you can count up and down. Therefore two push buttons are used, which are polled in the processor. This is very similar to the previous example, but there is a 2nd case checked if the down button is pressed, then the counter is decremented by one. If the counter gets a value smaller than 0000 then the counter resets to 9999 for a cyclic counting (other solutions are possible).

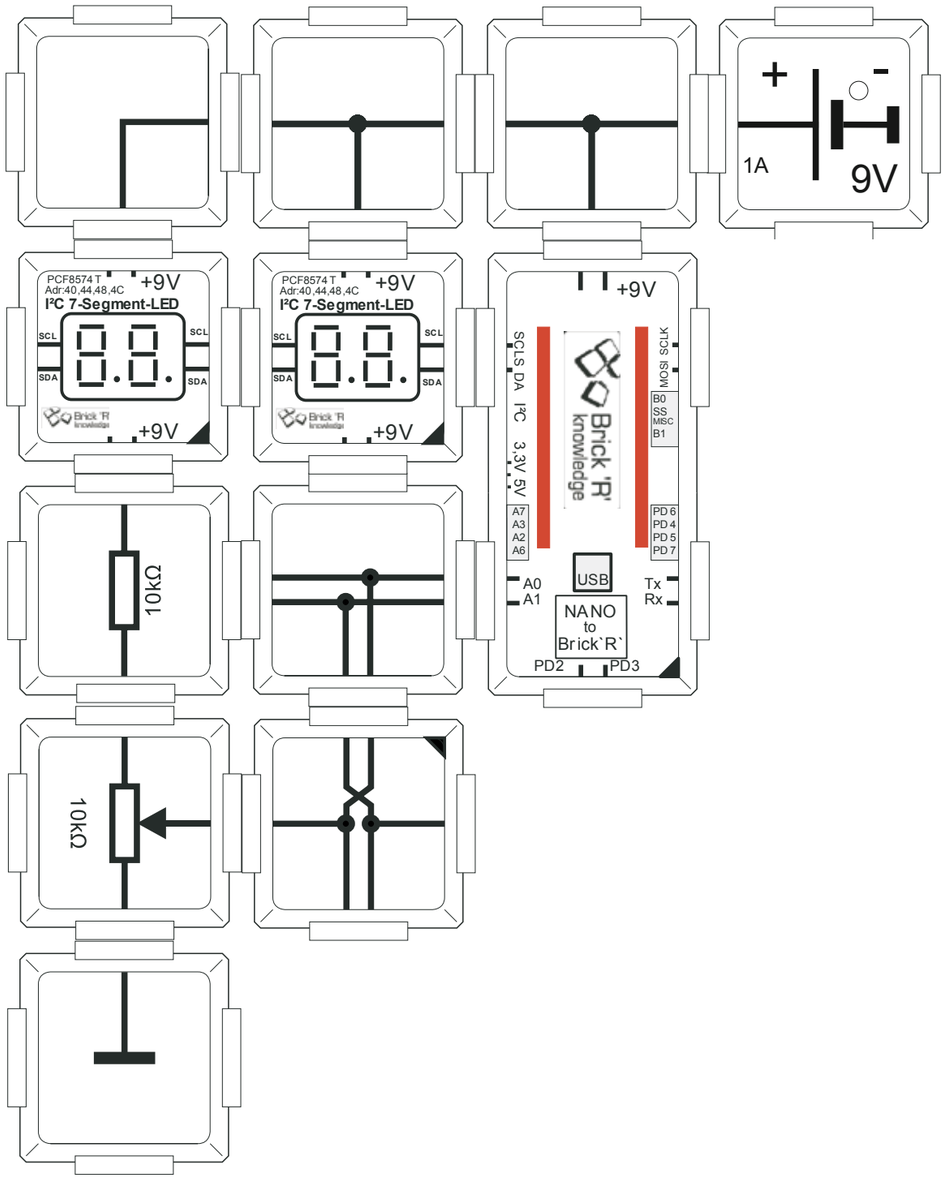


What happens? If the upper button is pressed the counter should increment exactly by one, if the lower button is pressed the counter decerements by one. From 0000 its getting the value of 9999.


```
// EN_20_SevenSegment_Counter_updown_I2C
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... as before ...

// new code:
#define PORTTASTEUP 2   // connect button to PD2 for upcount
#define PORTTASTEDOWN 3 // connect button to PD2 for downcount

void setup() { // initialize (non-recurring)
  Wire.begin(); // initialize I2C
  pinMode(PORTTASTEUP,INPUT_PULLUP); // connect both buttons
  pinMode(PORTTASTEDOWN,INPUT_PULLUP); // with pullup resistances
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCII
  static int counter = 0; // static variable counter
  sprintf(buffer,“%04d“,counter); // // convert counter to 4 digit ASCII and
write to buffer
  // read up button, it can bounce
  //-------------------------------
  if (digitalRead(PORTTASTEUP)==LOW) { // if button is pushed signal high -> low
    delay(20); // wait 20 ms until button is not bouncing any more
    // it depends on used button if time is enough
    while (digitalRead(PORTTASTEUP)==LOW) { // loop while buttonn is pressed
      // that means wait for low -> high
      // as long as button is hold nothing happens
    } // continue after button is released
    counter++; // after release increment the counter
    delay(20); // and wait again to sum up 40ms
  }
  // read down button, it can bounce
  // it follows the same as we did for the up button
  if (digitalRead(PORTTASTEDOWN)==LOW) {
    delay(20);
    while (digitalRead(PORTTASTEDOWN)==LOW) {
    }
    // but now we count down
    counter--;
    delay(20); // same delay
  }
  if (counter < 0) counter = 9999; // counter to 0 then start again with 9999
  if (counter > 9999) counter = 0; // counter to 9999 then start again with 0
  // show counter as 4 digits from buffer 0..3
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb (most significant byte)
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // third position from left
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb (least significant byte) c
} // end of the loop
```

7.5 A/D converter using a seven segment display for a voltage meter

The value of the A/D converter is read and then the resulting milivolt calculated. This will have a range between 0mV and 5000mV. The range might be needed to calibrate (in a previous chapter we showed this). The voltage is the voltage at the taper of the potentiometer. With a 9V battery and the protection resistor of 10kOhm the maximum voltage should be around 4500mV.
The protection resistor is important to avoid voltages larger than 5V at the input of the Nano brick.



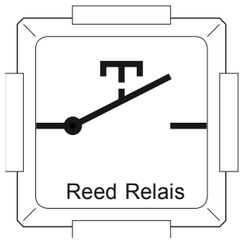

```
// EN_21_SevenSegment_Voltmeter_I2C
#include <Wire.h>

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

... as before ...



// new code:

void setup() { // initialize (non-recurring)
  Wire.begin(); // initialize I2C
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCII
  int poti = analogRead(A0); // read a1,a2,a3 0..1023 from A0
  double milivolt = 0;       // intermediate variable
  milivolt = ((double)poti*5000.0)/1024.0; // convert to max 5000 militvolts
  sprintf(buffer,"%04d",(int)milivolt); // convert to ASCII string with 4 digits
  delay(50); // to avoid the display to jump too fast
  // show counter as 4 digits from buffer 0..3
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // msb (most significant byte)
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // third position from left
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // lsb (least significant byte)
} // end of the loop
```

What happens? The display shows a value between 0000 and 5000. By turning the potentiometer a value close to 0000 can be reached and in the other way round a value of around 4500mV depending on the battery voltage (1/2 of).

8. Relais

8.1 Reed relais

Relais are still used today if either large loads have to be switched or the circuits have to be isolated.
The information whether a contacts should be closed or not is done via a magnetic field.
A special version of a relais the the reed-relais, with special contacts the so called reed contacts.



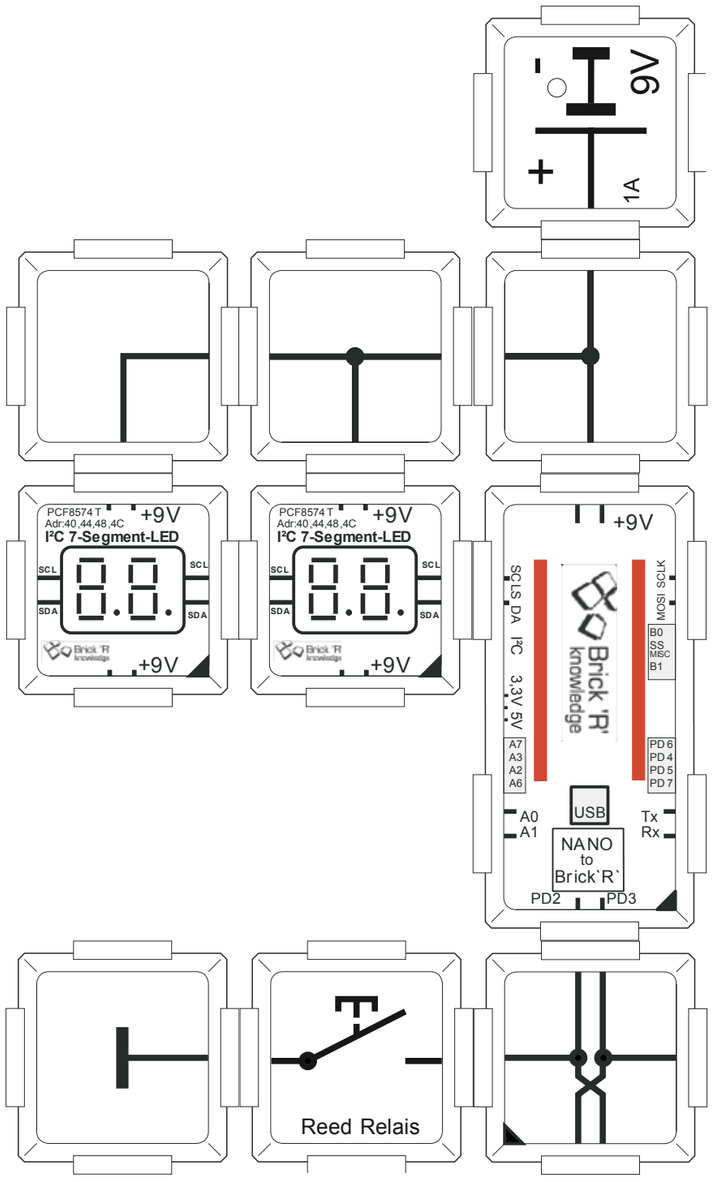
The reed contact is activated via the magnetic field. For our reed relais a small magnet is used which should be brought aside to the reed contact in the longer direction. You can see the reed contact, as its within a small glas tube. If the magnetic field is in the right direction and strong enough the contact will close.
This allows implementing contactless switches. For example they can be used for theft protection.
The reed contact is mounted at the door frame and the magnet at the door. If the door is closed the contact should be closed also. Then a circuit can be closed as long as the door is closed. Now you can check at a different location if the door is closed or open. If the door is open the circuit is interrupted, no current can flow and you can build an indicator for this to do an action for example.
Same applies for window contacts, the contacts are mounted at the windows frame and the magnets at the window. When the window is closed the contact should be closed also.
All contact information can be brought together at a central place and so be checked if all relevant doors / windows are closed for example. To get an intruder alert you need to do some processing and maybe add other sensors.
This is a really great opportunity for implementing own ideas.

Other applications for reed contacts come from industry. For example build a revolution counter.

8.2 Space for notes

8.3 Reedrelais controls the display

The reed relais controls the display. When approching the magnet and rmove it the counter is incremented by one. For a revolution counter, the magnet can be added to a wheel and the contact counts the revolutions. Usually its better to debounce also reed contacts, though they are much faster for the bounce time.
We use our normale counting program for the implementation.

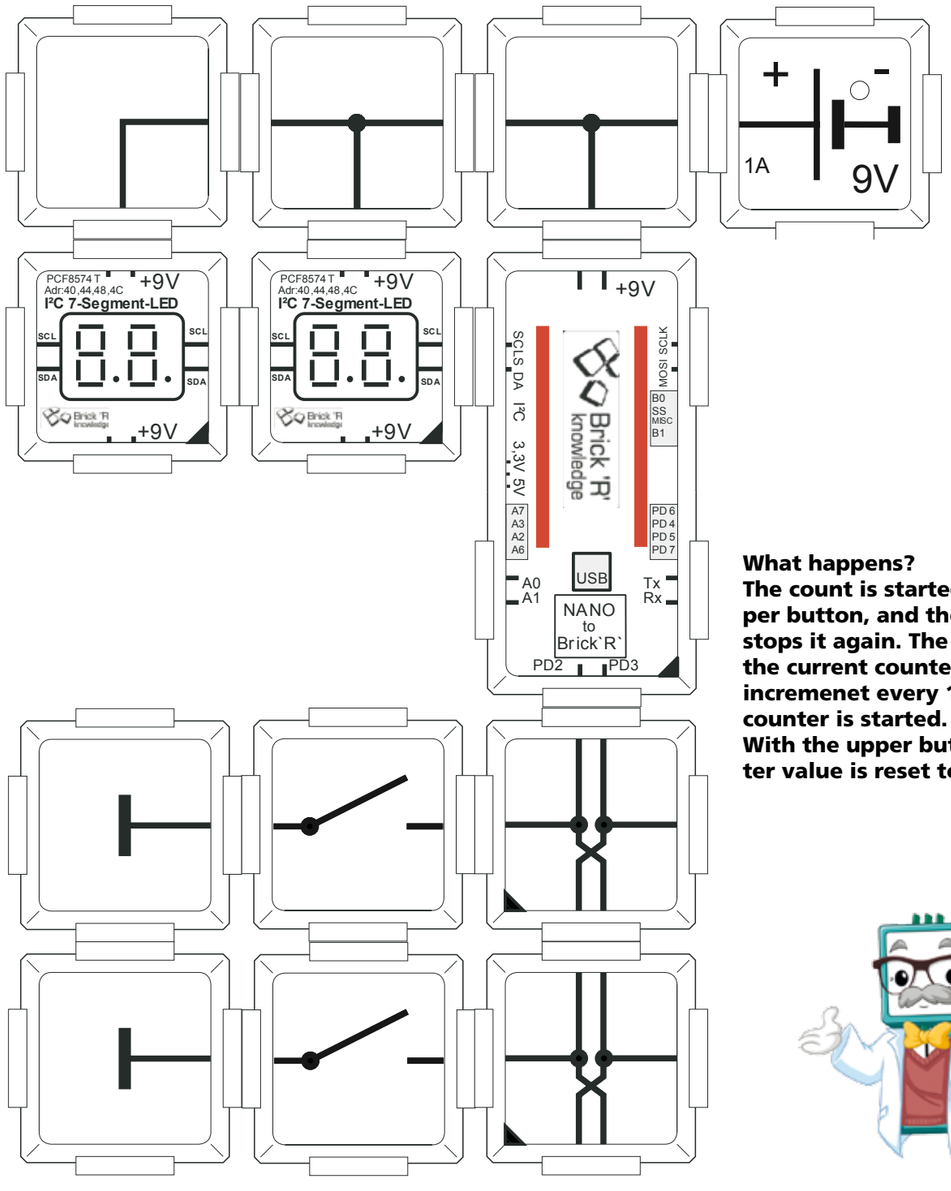



```
// EN_22 Seven Segment Counter with Reed Relais.
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1)
#define i2cseg7x2amsb1 (0x42>>1)

..... as before ...


// new code:

#define PORTRELAIS 2 // connect Reed-Relais to PD2

// initialize (non-recurring)
void setup() {
  Wire.begin(); // initialize I2C
  pinMode(PORTRELAIS,INPUT_PULLUP); // use pullup resistor for the reed-contact
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCII string
  static int counter = 0; // static count variable
  sprintf(buffer,"%04d",counter); // convert integer counter value to 4 digit ASCII
  // read reed-contact; it can bounce also
  if (digitalRead(PORTRELAIS)==LOW) { // contact closes; signal high -> low
    delay(20); // wait 20ms until bouncing ends
    while (digitalRead(PORTRELAIS)==LOW) { // loop while contact is closed
     // wait until contact opens !
    }
    counter++; // increment the counter
    delay(20); // wait some time until bouncing after contact opening ends
  }
  if (counter > 9999) counter = 0; // counter running from 0..9999 then start again
  // show counter as 4 digits from buffer 0..3
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // left most character
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1x(i2cseg7x2amsb1,buffer[2]); // third position from left
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // last character to most right display
} // end loop
```

What happens? If a magnet is approched and removed from the reed contact (it must be strong enough and in the right direction), the counter is incremented and shown on the display.

8.4 Stopwatch with seven segment displays

Now we have a stopwatch with two buttons for start and stop and a resolution of 1/10 sec = 100ms.
Until then we just counted how often the button was pressed or how long. The classic stopwatch is very similar to this and we can easily change the program structure for this. One button activates the counting and the other stops counting. We use here an addtional static varible named „startflag“. If set to one the count occurs otherwise it‘s on hold. One of the buttons sets the variable to 1 the other to 0.
We don‘t need to debounce the buttons in this case as multiple start or stop in sequence does not change the result. With „delay(100)“ we are close to 100ms but the additional instruction time is not taken into account. We will see later one a different method to get a precise counting result. For the moment you can try to adjust the parameter a little bit if necessary.

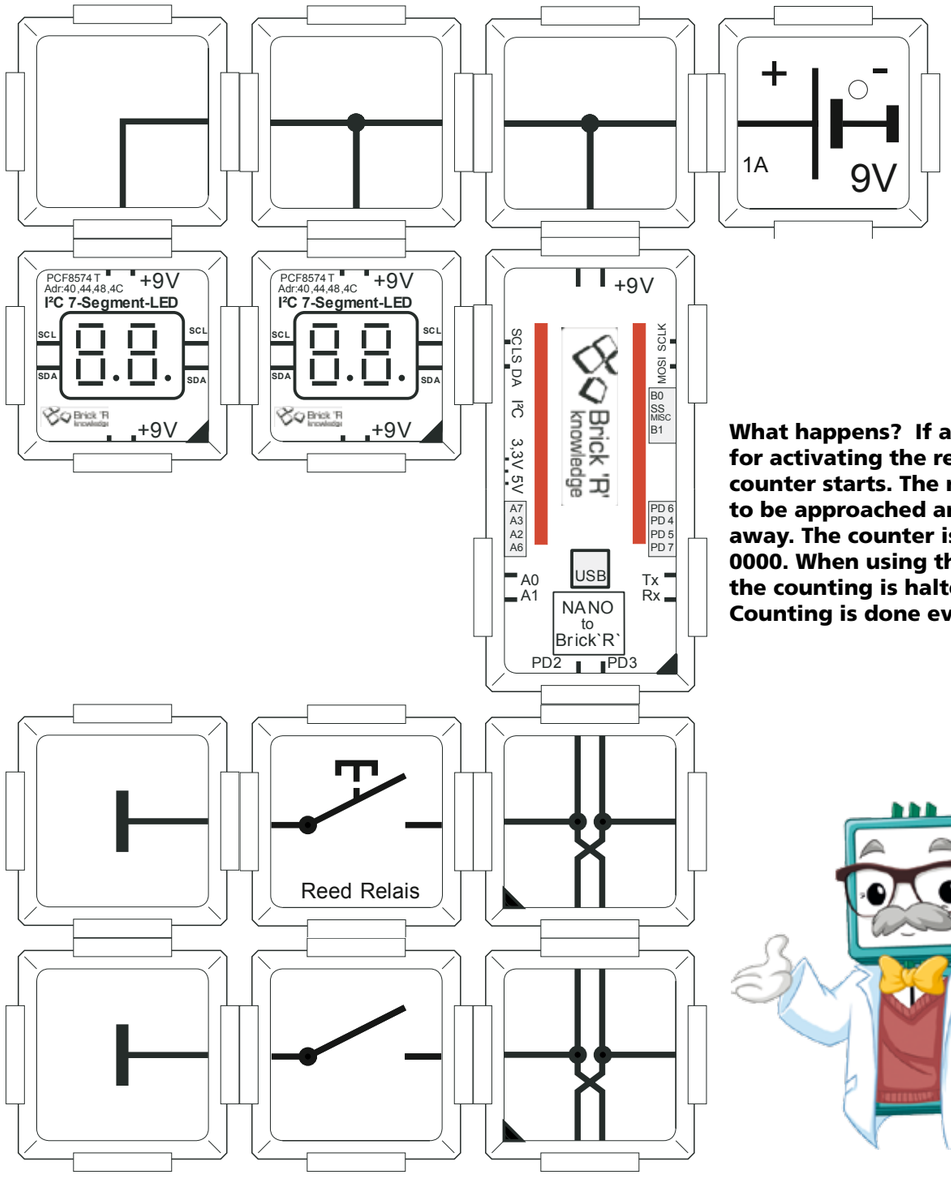


What happens?
The count is started with the upper button, and the lower button stops it again. The display shows the current counter value. It will incremenet every 100ms if the counter is started.
With the upper button, the counter value is reset to 0000.


```
// EN_23 7 segment counter Stop Watch
#include <Wire.h>

..... as before ...

// new code:

#define PORTSTART 2 // connect PD2 to Start
#define PORTSTOP  3 // conenct PD3 to Stop

// the setup function runs once when you press reset or power the board
// initialize (non-recurring)
void setup() {
  Wire.begin(); // initialize I2C
  pinMode(PORTSTART,INPUT_PULLUP); // use pullup resistor for start at PD2
  pinMode(PORTSTOP,INPUT_PULLUP);  // use pullup resistor for start at PD3
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCIIn string
  static int startflag = 0; // 0=stop 1=watch is running
  static int stopzeit = 0;  // time counter
  sprintf(buffer,“%04d“,stopzeit); // convert counter value to 4 digit ASCII
  // read start contact
  // ------------------
  // it can bounce also
  if (digitalRead(PORTSTART)==LOW) { // start watch contact closes
    delay(20); // wait 20ms until bouncing ends
    while (digitalRead(PORTSTART)==LOW) { // loop while contact is closed
     // wait until contact opens !
    }
    startflag = 1; // time counting starts
    stopzeit = 0;  // start at time zero (is also watch reset)
    delay(20);     // wait some time until bouncing after contact opening ends
  } // end of read start contact
  // read stop contact
  // ------------------
  // debounce the same as we did for the start contact
  // attention BUTTON has to be hold longer ! > 1/10 sec
  if (digitalRead(PORTSTOP)==LOW) {  // activate watch stop
    delay(20);
    while (digitalRead(PORTSTART)==LOW) {
    }
    startflag = 0; // time counting stops
    delay(20); // wait some time until bouncing after contact opening ends
  } // end of read stop contact
  if ((startflag == 1) && (stopzeit < 9999)) { // counts until 9999
    stopzeit++; // if startflag=1 then increment
    delay(100); // actually < 100ms !! HAS TO BE ACURATE
    // can be ralized more acurate with CPU timer
  } // end test startflag
  // show counter as 4 digits from buffer 0..3
  //-----------------------------------------
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // left most character t
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1xbin(i2cseg7x2amsb1,get_7seg(buffer[2]) & 0x7f); // third position
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // last character
} // end loop
```

8.5 Stopwatch with reed relais trigger

This is a version where the start is triggered with the reed-relais. A magnet getting close in the right direction towards the reed-relais will switch it on, and the stopwatch is started.
Usually two of the reed-relais would make a perfect arangement, but one is good for several applications also.
The algorithm is the same.
Attention: the same code EN_23 is used here as in the previous example.



What happens? If a magnet is used for activating the reed-relais, the counter starts. The magnet needs to be approached and then brought away. The counter is also reset to 0000. When using the stop button the counting is halted. Counting is done every 100ms.


```
// EN_23 7 segment counter Stop Watch
#include <Wire.h>

..... as before ...

// new code:

#define PORTSTART 2 // connect PD2 to Start
#define PORTSTOP  3 // conenct PD3 to Stop

// the setup function runs once when you press reset or power the board
// initialize (non-recurring)
void setup() {
  Wire.begin(); // initialize I2C
  pinMode(PORTSTART,INPUT_PULLUP); // use pullup resistor for start at PD2
  pinMode(PORTSTOP,INPUT_PULLUP);  // use pullup resistor for start at PD3
}

void loop() { // start loop
  char buffer[10]; // buffer for count value in ASCIIn string
  static int startflag = 0; // 0=stop 1=watch is running
  static int stopzeit = 0;  // time counter
  sprintf(buffer,"%04d",stopzeit); // convert counter value to 4 digit ASCII
  // read start contact
  // ------------------
  // it can bounce also
  if (digitalRead(PORTSTART)==LOW) { // start watch contact closes
    delay(20); // wait 20ms until bouncing ends
    while (digitalRead(PORTSTART)==LOW) { // loop while contact is closed
     // wait until contact opens !
    }
    startflag = 1; // time counting starts
    stopzeit = 0;  // start at time zero (is also watch reset)
    delay(20);     // wait some time until bouncing after contact opening ends
  } // end of read start contact
  // read stop contact
  // ------------------
  // debounce the same as we did for the start contact
  // attention BUTTON has to be hold longer ! > 1/10 sec
  if (digitalRead(PORTSTOP)==LOW) {  // activate watch stop
    delay(20);
    while (digitalRead(PORTSTART)==LOW) {
    }
    startflag = 0; // time counting stops
    delay(20); // wait some time until bouncing after contact opening ends
  } // end of read stop contact
  if ((startflag == 1) && (stopzeit < 9999)) { // counts until 9999
    stopzeit++; // if startflag=1 then increment
    delay(100); // actually < 100ms !! HAS TO BE ACURATE
    // can be ralized more acurate with CPU timer
  } // end test startflag
  // show counter as 4 digits from buffer 0..3
  //----------------------------------------
  display_seg1x(i2cseg7x2bmsb1,buffer[0]); // left most character t
  display_seg1x(i2cseg7x2blsb1,buffer[1]); // second position from left
  display_seg1xbin(i2cseg7x2amsb1,get_7seg(buffer[2]) & 0x7f); // third position
  display_seg1x(i2cseg7x2alsb1,buffer[3]); // last character
} // end loop
```

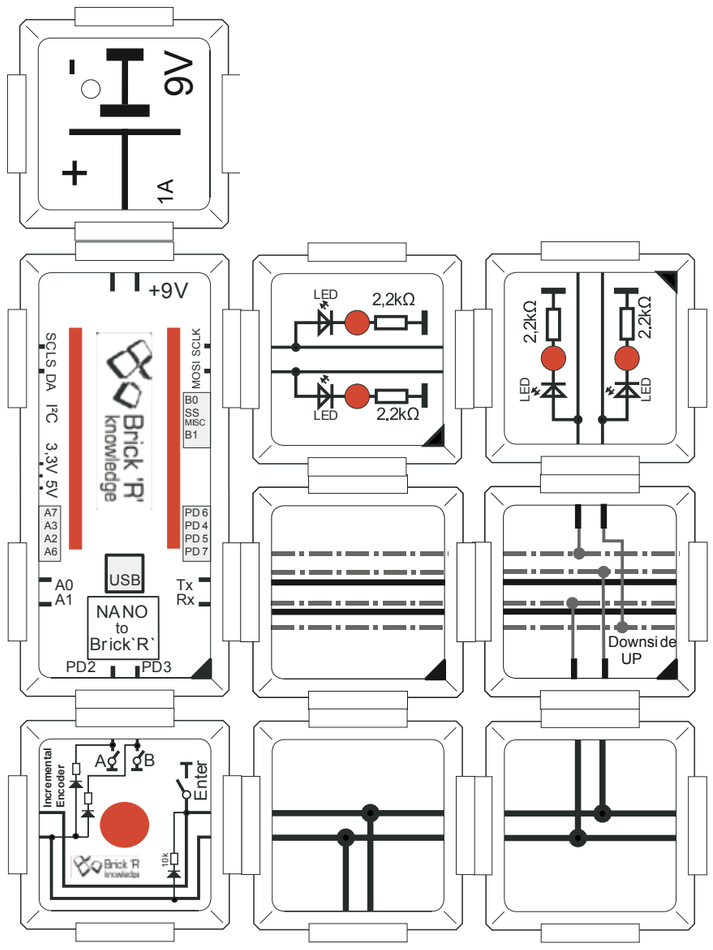
9. Rotary encoders

9.1 The rotary encoder brick

Incremental encoders are used in many industrial devices but also for consumer parts, like mp3 players, cameras etc. Using two phase shifted contacts the processor can determine the revolution direction. There is an additional button which can be pressed. This can be used to trigger any action.
According to the built in type 18 or 36 contact closes per full revolution can happen. The outpus of the main contacts are named „A“ and „B“. The contacts are closed in a specific way if the knob is rotated

angle	A	B
0	0	0
1	1	0
2	1	1
3	0	1

The sequence of A and B determines the rotation direction. Only one of the two contacts is closed or opened at one step. The contacts A and B don‘t need to be debounced.



port assignment

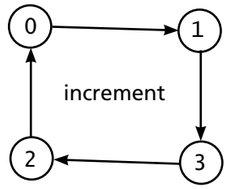
shortcut	port	parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

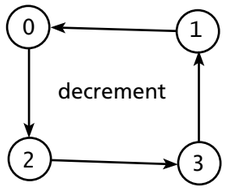
```
// EN_24 incremental encoder

#define SWITCHA 2 // encoder contact A
#define SWITCHB 3 // encoder contact B
#define SWITCHT 4 // encoder button contact
#define PULLUP 5  // only for pullup
#define PORTLED6 6 // display A
#define PORTLED7 7 // display B
#define PORTMOSILED 11 // encoder button status
#define PORTSCLKLED 13 // complemantary encoder button status

// the setup function runs once when you press reset or power the board
void setup() {
  // all buttons with pullup
  pinMode(SWITCHA,INPUT_PULLUP);
  pinMode(SWITCHB,INPUT_PULLUP);
  pinMode(SWITCHT,INPUT_PULLUP);
  pinMode(PULLUP,OUTPUT);      // switch port 5 to output
  digitalWrite(PULLUP,HIGH);   // additional define pullup
  pinMode(PORTLED6,OUTPUT);    // switch port 6 to output
  pinMode(PORTLED7,OUTPUT);    // switch port 7 to output
  pinMode(PORTMOSILED,OUTPUT); // switch PORTMOSILED to output
  pinMode(PORTSCLKLED,OUTPUT); // switch PORTSCLKLED to output
}

void loop() { // start loop
  int a= digitalRead(SWITCHA);   // read encoder contact A
  int b= digitalRead(SWITCHB);   // read encoder contact B
  int sw = digitalRead(SWITCHT); // read encoder button contact

  // show status of buttons with LEDs
  //---------------------------------
  if (a==HIGH) {
    digitalWrite(PORTLED6,HIGH); // LED 6 shows encoder contact A
  } else {
    digitalWrite(PORTLED6,LOW);  // switch to 0 if A=0
  }
  if (b==HIGH) {
    digitalWrite(PORTLED7,HIGH); // LED 7 shows encoder contact B
  } else {
    digitalWrite(PORTLED7,LOW);  // switch to 0 if B=0
  }
  if (sw==HIGH) { // also the encoder button shall do something
    digitalWrite(PORTMOSILED,HIGH); // one LED to 1
    digitalWrite(PORTSCLKLED,LOW);  // the other to 0
  } else {
    digitalWrite(PORTMOSILED,LOW);  // and vice versa
    digitalWrite(PORTSCLKLED,HIGH); // if pushed
  }
} // end loop
```

What happens? When turning the knob very very slowly the LEDs connected to ports 6 and 7 show the so called graycode of the rotary coder output. If the button on the knob is pressed two other LEDs alternate they are connected to port MOSI and SCLK (left horizontal group).

9.2 rotary encoder with display of values

The program is extended now, a counter is incremented and decremented according to the rotary encoder and the result is displayed on the terminal on the PC. Therefore the different phases of the encoder must be used to calculate the direction. A so called state diagram is used for this purpose. The program uses two additional variables for storing the current and past state.
The variable „state“ saves the current value of A and B with the encoded values 0,1,2 and 3 and „oldstate“ stores the last state previously. This determines the rotation direction. The transistion between the states is used for incrementing and decrementing the counter variable.

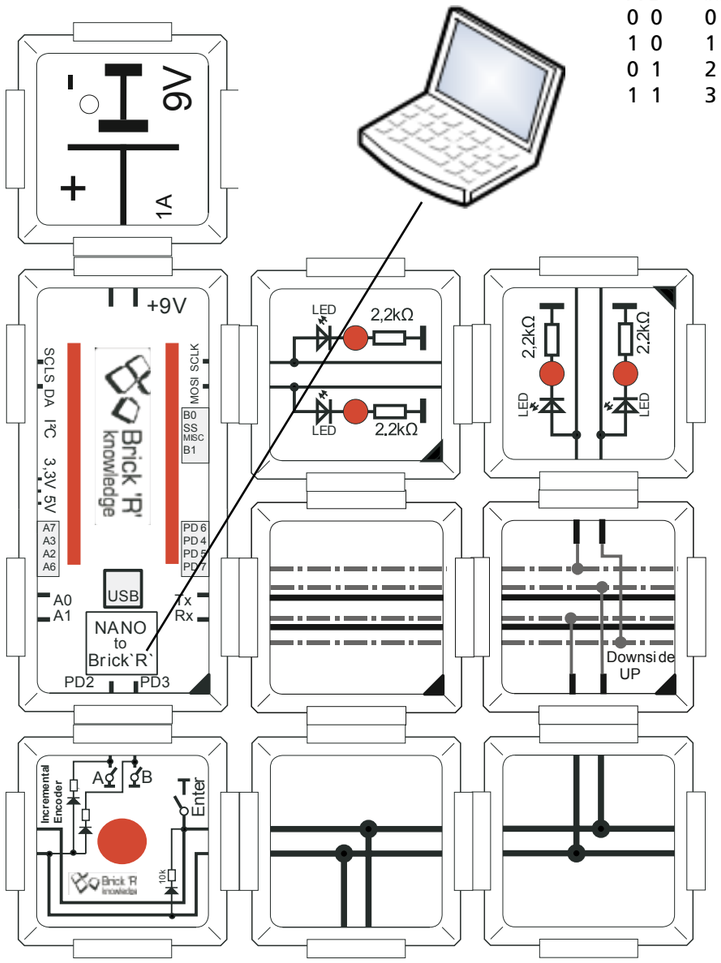




A	B	state
0	0	0
1	0	1
0	1	2
1	1	3

Attention:
Activate the terminal

You need to press CTRL-SHIFT-M when the arduino software is active.
A window with the terminal pops into the way.
Attention: set the baudrate to 9600 if it does not show a readable text.



port assignment

shortcut	port	parameter
SCK	PB5	#13
MOSI	PB3	#11
B1	PB1	#9
SS	PB2	#10
MISO	PB4	#12
B0	PB0	#8

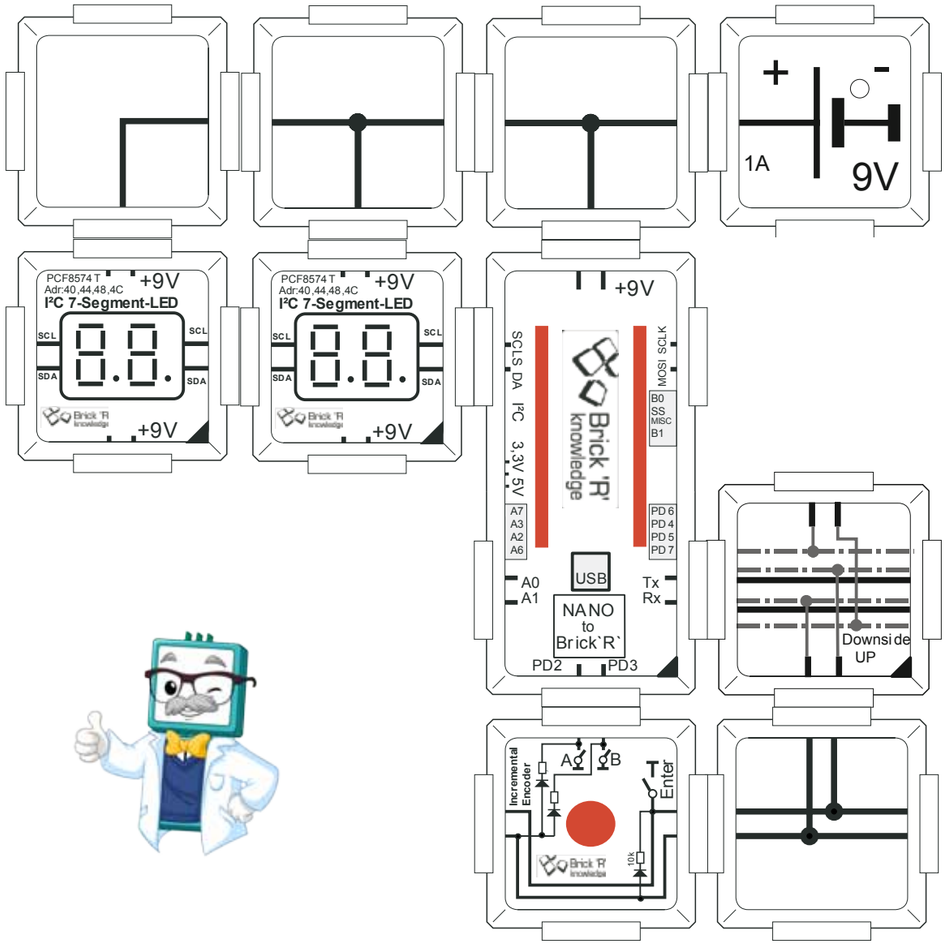
What happens? The LEDs behave like before but the terminal shows the additional output of the counter value which is incremented or decremented when turning the knob.

```
// EN_25_incrementalValueOutput to console

#define SWITCHA 2 // encoder contact A
#define SWITCHB 3 // encoder contact B
#define SWITCHT 4 // encoder button contact
#define PULLUP 5  // only for pullup
#define PORTLED6 6 // display A
#define PORTLED7 7 // display B
#define PORTMOSILED 11 // button status 1
#define PORTSCLKLED 13 // button status 2
void setup() { // the setup function runs
once
  Serial.begin(9600); // initialize with 9600
baud !
  // all contacts with pullup
  pinMode(SWITCHA,INPUT_PULLUP);
  pinMode(SWITCHB,INPUT_PULLUP);
  pinMode(SWITCHT,INPUT_PULLUP);
  pinMode(PULLUP,OUTPUT); // port 5 to output
  digitalWrite(PULLUP,HIGH); // define pullup
  pinMode(PORTLED6,OUTPUT); // output
  pinMode(PORTLED7,OUTPUT); // output
  pinMode(PORTMOSILED,OUTPUT); // PORTMOSILED
to output
  pinMode(PORTSCLKLED,OUTPUT); // PORTSCLKLED
to output
}

void loop() { // start loop
  static int counter = 0; // initialize  0
  int a= digitalRead(SWITCHA);// read A
  int b= digitalRead(SWITCHB);   // read B
  int state  = 0; // actual encoder state
  static int oldstate = 99; // old state
  int sw = digitalRead(SWITCHT);
  // read contact
  // set status of encoder   // set state
  // contacts A,B output to LEDs 6,7
    if (a==HIGH) { // first for contact A
    digitalWrite(PORTLED6,HIGH);
    state = 1;
  } else {
    digitalWrite(PORTLED6,LOW);
    state = 0;
  }
  if (b==HIGH) { // then for contact B
    digitalWrite(PORTLED7,HIGH);
    state += 2; // state +2 is a trick,
  } else { //  which delivers 2 or 3
    digitalWrite(PORTLED7,LOW);
    state += 0; // only for beauty
  }
  // increment or decrement a counter value
  // compare actual state with saved oldstate
  // turned right or left; according
   switch(state) { // the actual encoder
state
    case 0: // AB=00
     if (oldstate == 1)  { // if old
state was 1
        counter--; // then encoder tur-
ned backwards: decrement counter
     } else if (oldstate == 2) {
        counter++; // if old state was 2
then increment
     }
     break;
    // for the other states 1..2 the
code is structural the same; only other
transitions
    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; // save in oldstate
for comparison in next loop
  // read encoder button and show it
  // (button has no other function)
  if (sw==HIGH) {
    digitalWrite(PORTMOSILED,HIGH);
    digitalWrite(PORTSCLKLED,LOW);
  } else {
    digitalWrite(PORTMOSILED,LOW);
    digitalWrite(PORTSCLKLED,HIGH);
  }

  // IMPORTANT: output to console
  //----------------------------
  Serial.println(counter);
}
```

9.3 Rotary encoder and seven segment display for the output

Now with a little bit more comfort, the counter value is displayed on our 7-segment-displayed and the terminal application is no longer needed. The counting is done as in the previous example using the state machine. But the output is done on our display. Attention as the code is very large, we omitted the code part of the 7-segment-displays on the right side. Use the complete file or combine the listings with the appendix.
The use of the variable „oldcounter“ is a little bit special. Its value is compared to the new value of „counter“ and only if changed the 7-segment-display is updated. This saves processor time, as the polling of the rotary encoder is very time critial and doing both tasks can result in a very slow response or even dropping rotations. Now if you press the button, this resets the counter value.

What happens? The value of the coutner is displayed. It is incremented or decremented according to the turning direction of the knob.




```
// EN_26_Seven Segment Incremental
// Encoder with output on 7 segment
// display over I2C
#include <Wire.h>

// new code:
#define SWITCHA 2 // contact A
#define SWITCHB 3 // contact B
#define SWITCHT 4 // encoder button
#define PULLUP 5  // extra

void setup() {
  Wire.begin(); // initialize I2C
  pinMode(SWITCHA,INPUT_PULLUP);
  pinMode(SWITCHB,INPUT_PULLUP);
  pinMode(SWITCHT,INPUT_PULLUP);
}

void loop() {      // start loop
  char buffer[10]; // ascii  7 segment
  static int counter = 0;
  static int oldcounter = -999; // old
  int a= digitalRead(SWITCHA); // read
A
  int b= digitalRead(SWITCHB); // read
B
  int state  = 0;  // actual encoder
state
  static int oldstate = 99; // old sta-
te
  int sw = digitalRead(SWITCHT); //
read
  // use encoder button to reset the
counter !
  if (sw == LOW) counter = 0;
  // set encoder state to 0..3 accor-
ding
  // to encoder contacts A,B
  if (a==HIGH) {  // first A
    state = 1;
  } else {
    state = 0;
  }
  if (b==HIGH) { // then B
    state += 2;
  } else {
    state += 0;
  }
  // according to the change of encoder
status 0..3 increment or decrement a
counter value
  switch(state) {
    case 0: // AB=00
     if (oldstate == 1) { // if old
state was 1
        counter--; // then encoder turned
backwards: decrement counter
     } else if (oldstate == 2) { // if old
state was 2 then increment
        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; // save in oldstate
  // show counter as 4 digits from buffer 0..3
  sprintf(buffer,“%04d“,counter); // convert
integer counter
  if (oldcounter != counter) { // only if
changed
    display_seg1x(i2cseg7x2bmsb1,buffer[0]);
//
    display_seg1x(i2cseg7x2blsb1,buffer[1]);
//
    display_seg1x(i2cseg7x2amsb1,buffer[2]);
//
    display_seg1x(i2cseg7x2alsb1,buffer[3]);
//
    oldcounter = counter;     // for the next
loop !
  }
}
```

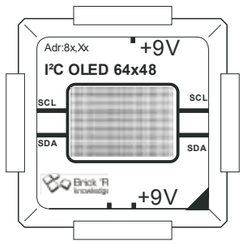
10. OLED

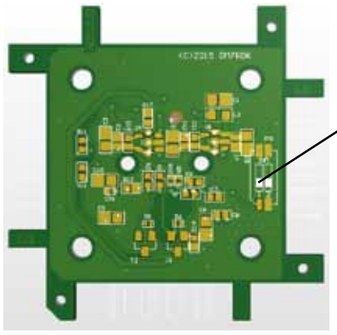
10.1 Graphic display principle

The 7-segment-displays usually are only used for numbers and with a lot of restrictions also for some characters. Mit a 14- and 16-segment-display characters can also be displayed. After this the first generic raster displays come to the market. With a matrix of 5x7 pixel, its possible to show characters in a very reasonable way, and also the first graphic symbols are possible. Originally LED matrixes were used for the displays, later on also LCD (liquid crystal displays) come to the market, which consume much less power. Now LEDs come back, but as OLEDs (organic light emitting diode), they can be manufactured in different colors and also need very low power. Our set contains a OLED wird 64x48 pixel. To display characters, a character table is needed like with the assignment table for the 7-segment-display. Also graphic display is possible in a simple way. To store the character set quite an amount of memory is needed, for 5x7 pixel per character, this is around 5 bytes per character. With the complete ASCII-character set, which is 128 characters including space, 128x5 Bytes = 640 Bytes is needed as storage. We did prepare such a character table for the Nano-brick, which is constructed a little bit different. The font is a little bit larger than the 5x7 set and we use a variable font width, so called proportional font. To generate such a table, we used the BitFontCreator PRO v3.0. It allows for different font types and sizes, and generates all necessary tables automatically from different processor types.

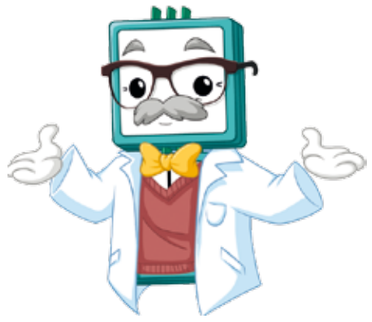
Our OLED brick uses one I2C address for communication, either 0x78 or 0x7A, which depends on the switch SW1 on the backside of the PCB (after opening the cover). The inner switch determines the address, usually a 1-sign should be there. The OLED is programmable and uses its own small microcontroler.

We built a small library, with procedure calls to display characters, texts, lines and many more in an easy way, which we will show in the following examples.





address 78 or 7A
switch to ON position
means 78



```
// part of our code

// single character codes

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,
...

};

// table with offsets into the character table
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 */
};


// width table
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 library

We prepared a lot of commands, so it's easier to access the OLED than with the direct I2C commands. They are based on the internal controller and allow to set a single pixel, set intensity, and many more. Also more complex functions to draw lines, show characters and text, which are not directly available in the controller are implemented in our small library.

The most important command:

```
i2c_oled_initall(i2coledssd);
```

This initialize the OLED with timing, size etc. and starts the display process. This sould be done in the setup function of the Arduino Code. The parameter is the I2C address used:

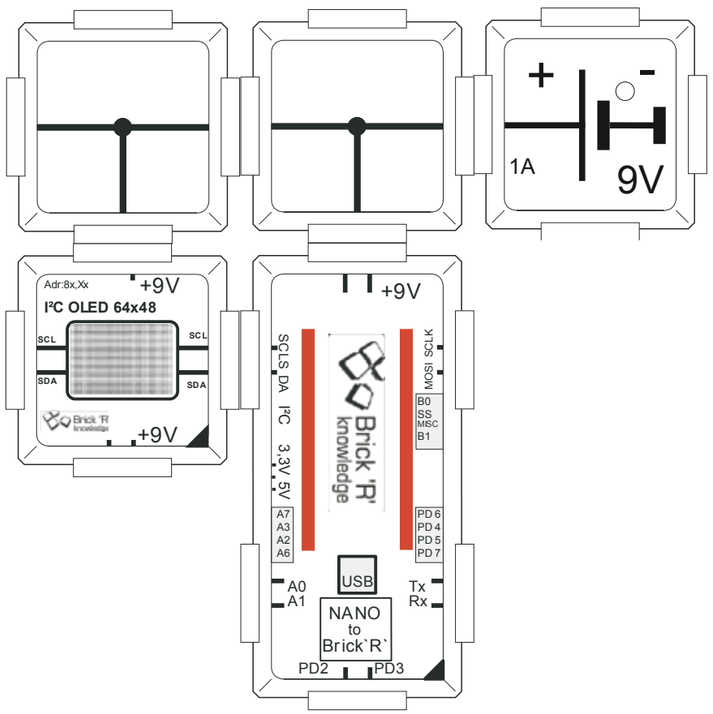
```
#define i2coledssd (0x7A>>1)  // Default
```

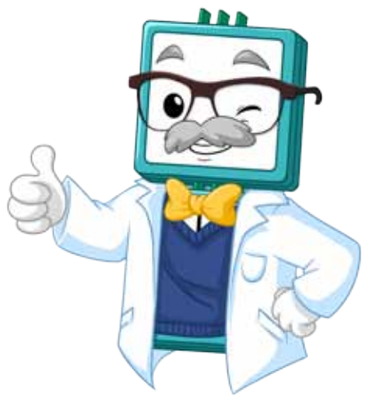
or

```
#define i2coledssd (0x78>>1)  // for an optional 2nd OLED display brick
```

AN internal buffer (lcdbuffer[]) is used for storage of all pixel states. The contents is then transfered as fast as possible to the display. This is called double buffering and allows flicker free operation.

```
// Black and white pixels without greylevels for this display
#define COLOR_BLACK 0        // makes code more readable
#define COLOR_WHITE 1        // OLED Pixel = On (if not the inverse mode is activated)
```



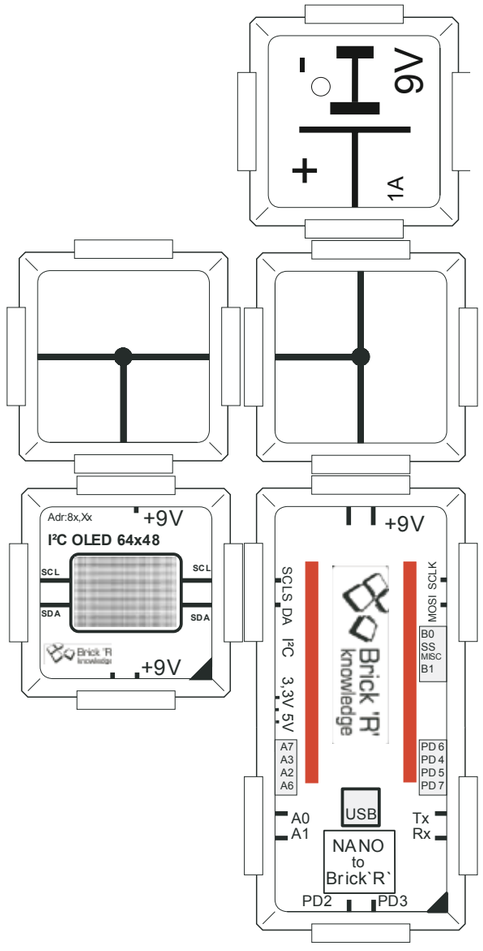


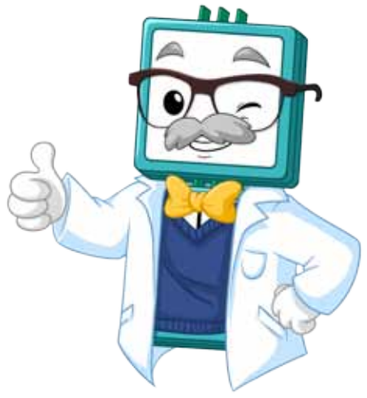

```
void i2c_oled_write_command(unsigned char i2cbaseadr, unsigned char cmdvalue)
      used internally to write I2C data

void i2c_oled_entire_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Switch on/off the OLED. 1=on and 0=off

void i2c_oled_display_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Can also be used to switch the display on or off

void i2c_oled_setbrightness(unsigned char i2cbaseadr, unsigned char wert)
      Set the intensity of the OLED 0..255, 0=dark

void i2c_oled_inverse_onoff(unsigned char i2cbaseadr, unsigned char onoff)
      Display is inverted. 1=invers

unsigned char i2c_oled_write_top(unsigned char i2cbaseadr, int zeile,
                   int bytes,  unsigned char barray[],signed int sh1106padding)
      Used internally to write a group of lines

void disp_lcd_frombuffer()
      transfers the contents of the buffer (lcdbuffer[]) to the display

void disp_buffer_clear(unsigned short data)
      clears the pixelbuffer (lcdbuffer[]) with data (=0 oder =1)

void disp_setpixel(int x, int y, unsigned short col1)
      set a single pixel at x,y where x list left and y is the top
      col1 defines the color (0 or 1)

unsigned short disp_setchar(int x, int y,  unsigned char chidx1, unsigned short color)
      sets a character at x,y with the code chidx1 in ASCII and color (0,1)

int disp_print_xy_lcd(int x, int y, unsigned char *text, unsigned short color, int
chset)
      prints a text (ASCII at char *text) at x,y with the characterset
      chset (ignore in this library), starting at the upper left.

void  disp_line_lcd(int x0, int y0, int x1, int y1, unsigned short col)
      draws a line form x0,y0 to x1,y1 with color col (0,1)

void disp_rect_lcd(int x1, int y1, int x2, int y2, unsigned short col)
      draws a rectangle at x1,y1 to x2,y2 with color col (0,1)

void disp_filledrect_lcd(int x1, int y1, int x2, int y2, unsigned short col)
      draws a filled rectangle at x1,y1 to x2,y2 with color col (0,1)
```

10.3 OLED using I2C

We use a sine function to show it on the display. After each frame, the position of the sine is changed a little bit by offseting the phase. Then its looks like the sine function is scrolling.
We use some praktical elements from our library. With disp_buffer_clear() the display buffer is cleared. The color is used as parameter, of course only have two different values, either black or white.
The buffer is in the memory of the Nano brick not on the OLED. At this time the current image is still displayed on the OLED. This is also called double buffer, the second buffer is the memory intergrated into the OLED.
Then a horizontal line is drawn. For this purpose we can use the disp_line_lcd(). It used a start coordinate x,a and a destination coordinate which is included when drawing the line. X has a range from 0 to 63 and y is from 0 to 47. The coordinate 0,0 is on the upper left side of the display and y points downwards.
Now the sin() function must be adjusted to the display range. Y is calculated between 0..47. But the sine-function has a range of -1.0 to +1.0. A scaling and offset adjusts the value in the formula.
Now disp_setpixel() is used to set all single pixels of the function. The function uses x,y as destination coordinate and a color which we use white. The parameter is then „47-y“ to mirror the coordinate system, because y=0 is on top of the display.

Attention: We use the display at I2C address 7A - change the DIL switch position to OFF if necessary!



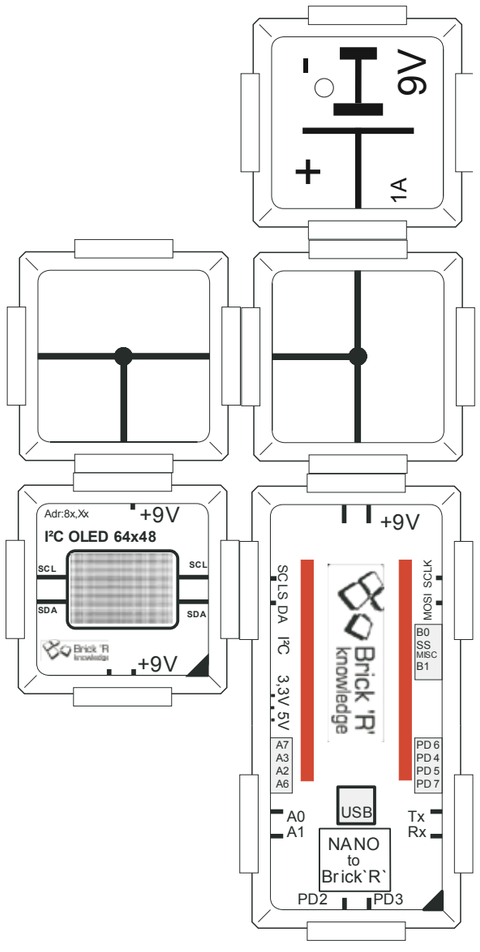


```
// EN_27 OLED example - pixel routines

#include <Wire.h>         // I2C library
#include <avr/pgmspace.h> // access to ROM


// -------------------------OLED ---------------------------------------------
// GLO066-D-M2005 -- SSD 1306 driver
// 011110sr  s=sa r=rw bei ssd1306 sa = adressbit to set optional
// if necessary adapt here the address  to 78 or 7A acording to switch
#define i2coledssd (0x7A>>1)  // default is 7A

..... use library code from appendix

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



// the setup function runs once when you press reset or power the board
void setup() {
  Wire.begin();                // initialize I2C
  i2c_oled_initall(i2coledssd); // initialize OLED
}


void loop() {  // start loop
  // 64x48 pixel OLED
  static int phase=0; // use oahse as scroll offset
  disp_buffer_clear(COLOR_BLACK); // clear virtal buffer
  disp_line_lcd (0,24,63,24, COLOR_WHITE); // x0,y0,x1,y1 to middle
  for (int i=0; i<63;i++) { // for each 64 rows
    int y=0; // sine curve calculation
    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); //0,0 is left upper adjust for -y
  } // all columns
  disp_lcd_frombuffer(); // now update display
  phase++; // next phase
  if (phase>=360)phase=0; // keep within 0..359 deg
}
```

What happens? After starting the program a sine-function is displayed. The curve slowly scrolls in horizontal direction.

10.4 OLED and character set

For displaying more than dots, you can use our character display procedures.
With disp_print_xy_lcd() text can be shown on the display. It uses the left upper dot as coordinate (x,y). The character size is around 10 pixels. If multiple lines are written then the new position of the next line must be calculated accordingly. In our example we write multiple text lines. The very famous sentence „the quick brown fox jumps over the lazy dog“ contains all characters of the alphabet and is used very often for this purpose.
An additional variable yoffset is used which is decremented for each loop interation. It‘s used for the text position. This results in a scroll effect, the text scrolls upwards. If an offset of -60 is reached, the screen is empty and the offset is reset to 50 so the first line appears at the bottom line.




```
// EN_28 OLED example - characterset

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

// set the according address
#define i2coledssd (0x7A>>1)  // default is 7A

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

..... use library code from appendix

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

void setup() {
  Wire.begin(); // I2C initializsation
  i2c_oled_initall(i2coledssd);  // OLED init after
}

void loop() {  // main loop
  // 64x48 Pixel OLED
  char buffer[40]; // buffer for character output
  static int yoffset = 50; // scroll offset
  disp_buffer_clear(COLOR_BLACK); // clear double buffer
  // output single lines. and use 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();  // then transfer the buffer to OLE
  delay(10); // more safe for the timming
  yoffset = yoffset - 1; // scroll line by line
  if (yoffset < -60) yoffset = 50; // then start again
} // end of loop.
```

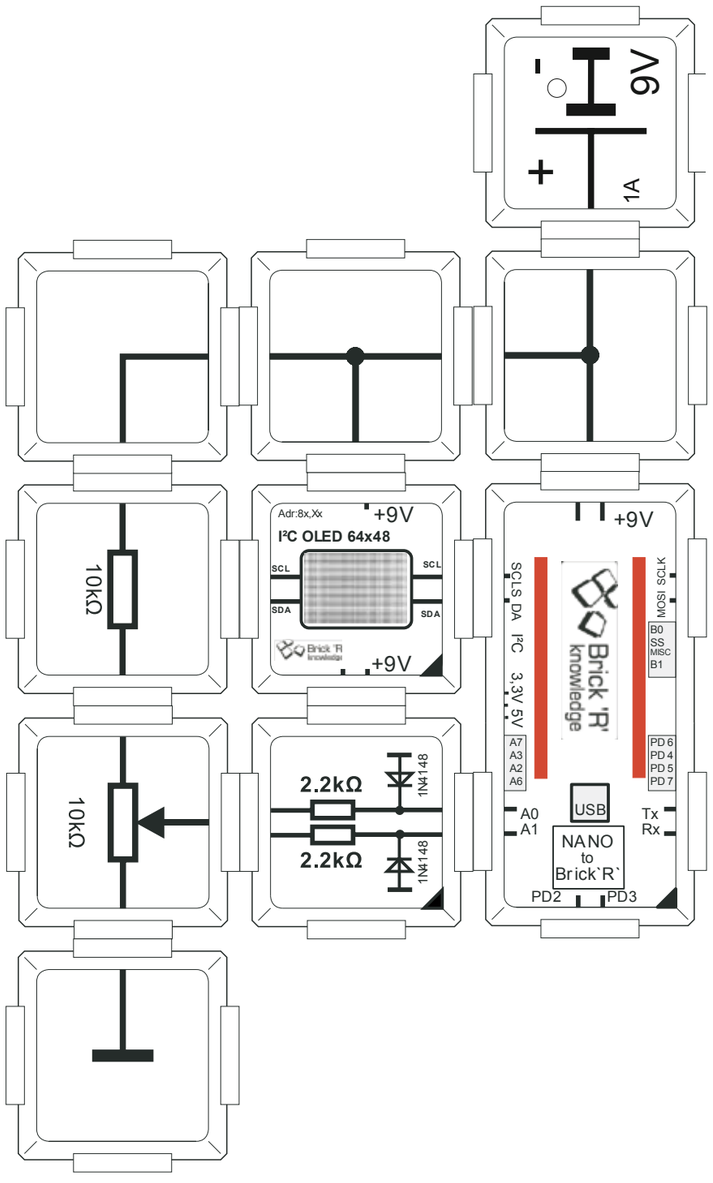
What happens? The text „the quick brown fox jumps over the lazy dog 0123456789“ is shown on the display and scrolls splitted into several lines in vertical direction. The whole repeats.

10.5 OLED Display with A/D-converter as voltmeter

We used already the 7-segment-display for this purpose, but with the OLED much more comfort is possible now. Here the values is displayed in mV - milivolts.
We use the digitized value of channel A0 which results in the digital range 0..1023. Now this must be adjusted to get the voltage range from 0 to 5000mV with following formula:

```
double mverg = (double)poti * 5000.0 / 1023.0;
```

If a more precise display is wanted, the value of „5000.0“ in the formula can be adjusted. Therefore a defined voltgae source is need or an external Voltmeter. The new value is then measured and a correction factor calculated.



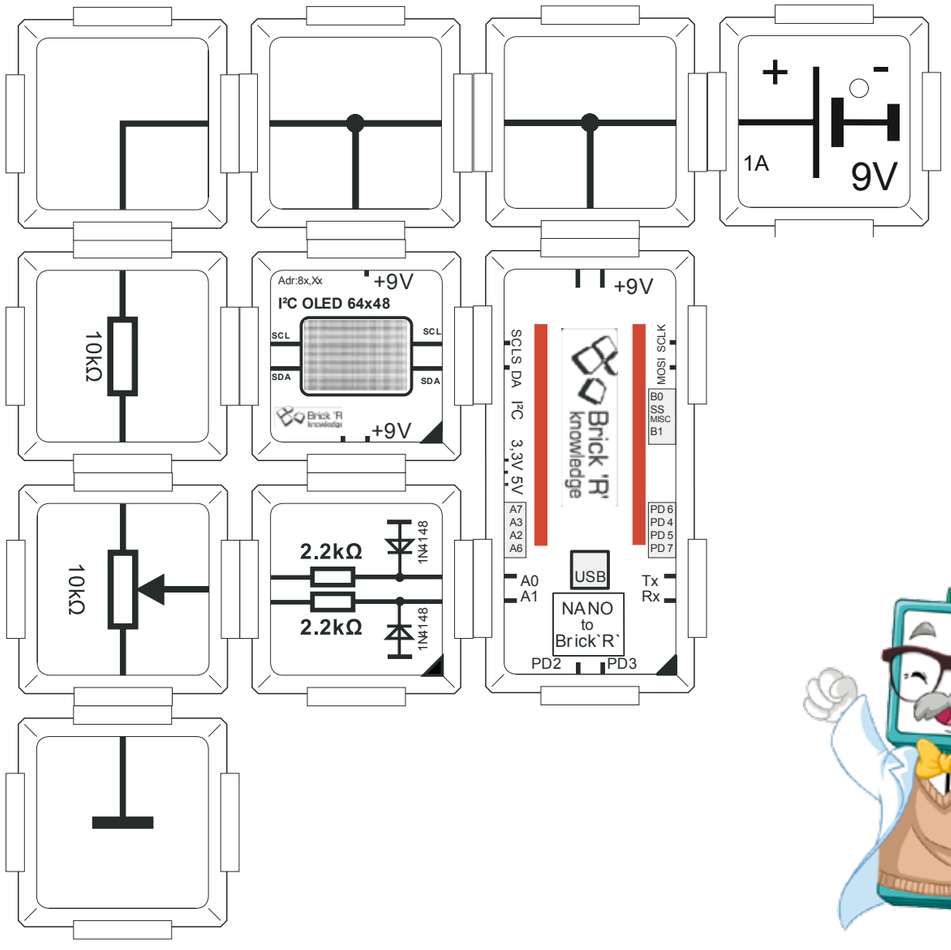

```
// DE_29 OLED Beispiele - AD Voltmeter

#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 ---------------------------------------------

..... use library code from appendix


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


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

void loop() {  // loop
  // 64x48 Pixel OLED
  int poti = analogRead(A0); // read A/D convert
  char buffer[40]; // double buffer
  disp_buffer_clear(COLOR_BLACK); // clear virtual buffer
  double mverg = (double)poti * 5000.0 / 1023.0; // adjust!
  int mvint = (int) mverg; // optional adjustment
  sprintf(buffer,“%4d mV“,mvint); // display in mV
  disp_print_xy_lcd(10, 20, (unsigned char *)buffer, COLOR_WHITE, 0);
  disp_lcd_frombuffer();  // now display
  delay(10); // more safe for the timming
}
```

What happens? The measued voltage is displayed in mV on the OLED. If you turn the potentiometer in a range from 0mV till 5000mV can be shown, but usually only half of the voltage source, in our case 1/2 of 9V = 4500mV (depends on the battery status).

10.6 OLED Display with A/D-converter as simple mini-oscilloscope

When saving the measurement values in a buffer, a small mini oscilloscope can be implemented. Of course the maxmim reachable frequency is not very high if the image is displayed after each measurement like we do. On top the voltage is shown and below the time dependent change of the signal.

Left you can see a commercial oscilloscope from Rigol. There are a lot off settings possible, like amplitude, time base and trigger settings. But the principle is always the same. They can show the time dependant changes of an input signal in a graphic way. In earlier times electron guns very used with high coltages, today its much easier and with more comfort done using digital electronic. But still such scopes can be very complex devices. The maximum frequency is one of the major features for example. Currently the limitation for a real time display is around 100 GHz (LeCroy Teledyne LeCroy LabMaster 10-100zi 100GHz, 240GS/s oscilloscope with around 1 Mio dollar). At www.TheSignalPath.com you can take a look at the working principle of such scopes. Out mini scope is of course at the other end, but much more cheaper.

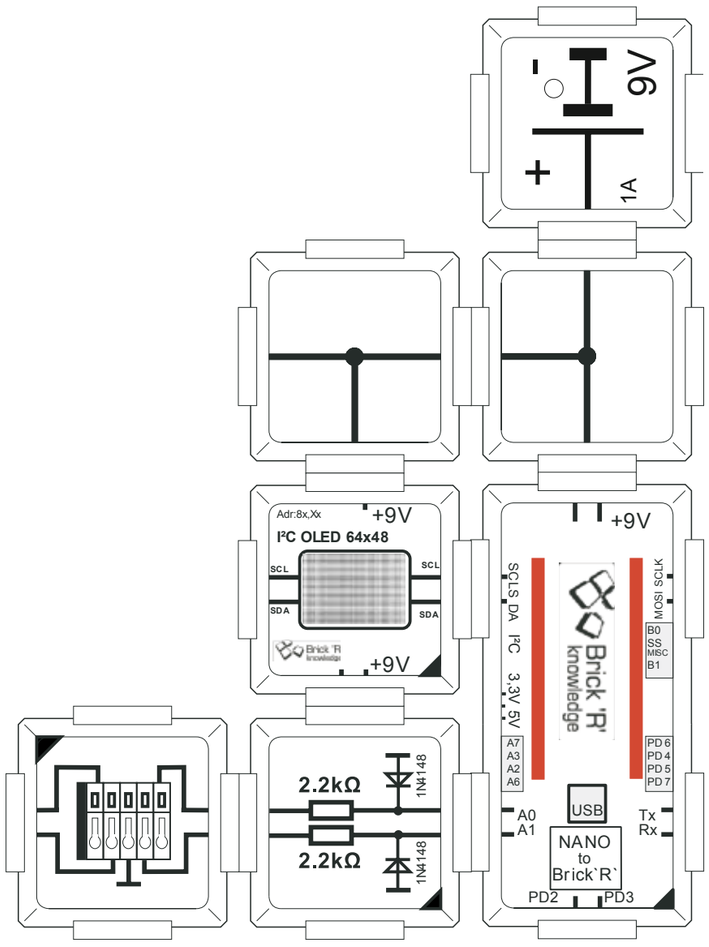


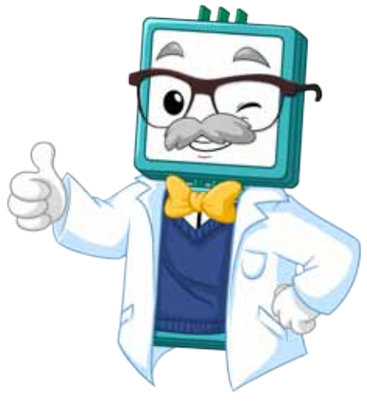
```
// EN_30 OLED example - AD Mini Oscilloscope

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

// set the according address
#define i2coledssd (0x7A>>1)  // default is 7A

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

..... use library code from appendix

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

char advalbuf[64]; // cyclic memory for ad values

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

void loop() { // loop
  // 64x48 Pixel OLED
  static int cxx = 0; // cyclic counter
  int poti11 = analogRead(A0); // get A/D value
  char buffer[40]; // text output
  disp_buffer_clear(COLOR_BLACK); // clear first
  double  p1 = (poti11*5000.0)/1023.0;  // to mV
  int y1 =0; // temp variable for y
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000); // output
  y1 = 47 - (p1 * 30.0)/5000.0; // 5V Max -> 30 pixel
  advalbuf[cxx++] = y1; // result  0..4xx V +-128  cylic buffer
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // start from 0.
  int yold  = advalbuf[(cxx+1)%64]; // last value
  for (i=0; i<63; i++) { // for all columns
   y1 =advalbuf[(cxx+1+i)%64]; // new value
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);  // connect
   yold = y1; // save the last to connetc the lines
  } // for all pixels
  if (cxx >63) cxx =0;  // counter from 0l..63
  disp_lcd_frombuffer();  // to display
  delay(10); // defines sample rate
}
```

What happens? The measured voltage is displayed in mV on the OLED. If you turn the potentiometer the Voltage is shown with a maximum of 5000mV (usually around 4500mV). At the same time you can see the change of time in form of a graphic display below the voltage.
By turning the potentiometer back and forward you can see the effect very good.

10.7 OLED display with A/D-converter as dual voltmeter

This time we use two channels: A0 and A1, they are both polled and the values are displayed on the OLED in Volts.
We use our universal brick, so its easy to connect two analog sources and do measurement. The voltage range is between 0V and 5V. Attention we use our protection brick, but though the two 2.2kOhm resistors and the reverse protection diodes, never connect voltages out of the recommened voltage range which is 0V to 5V.
This time we show the result in Volts with three digits after the comma. Because its not recommened to use floating point numbers within the printf command (you need different Arduino library in some cases), and which is also slow, its easier to use integer number with a small trick. The number before the digit separator is a value divided by 1000 (without rounding), if you get the value in mV. And the digits with the fractional parts can be determined by using the modulo function: „% 1000“. Printing is done with „%d“ before the fractional part and with „%03d“ for the fractional part. The „0“ fills up the number wird zeros. Inbetween you can set the dot as digit seperator.





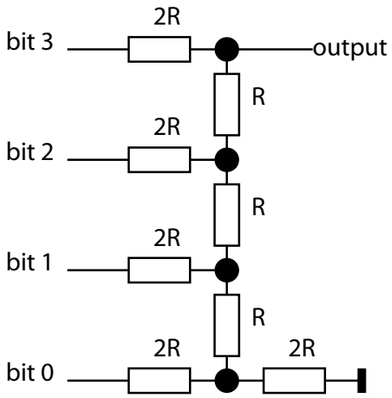
```
// EN_31 OLED example - AD Dual

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

// set the according address
#define i2coledssd (0x7A>>1)  // default is 7A

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


..... use library code from appendix


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


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

void loop() {     // loop
  // 64x48 Pixel OLED
  int poti11 = analogRead(A0); // read channel 0
  int poti12 = analogRead(A1); // read channel 1
  char buffer[40]; // text output
  disp_buffer_clear(COLOR_BLACK); // clear display
  double  p1 = (poti11*5000.0)/1023.0; // Voltage A0
  double  p2 = (poti12*5000.0)/1023.0; // Voltage A1
  // use a small trick to print a float by using int
  // values - split them in two parts
  sprintf(buffer, „A0=%d.%03dV“,(int)p1/1000,(int)p1%1000); // output
  disp_print_xy_lcd(2, 8, (unsigned char *)buffer, COLOR_WHITE, 0);
  sprintf(buffer, „A1=%d.%03dV“,(int)p2/1000,(int)p2%1000); // output
  disp_print_xy_lcd(2, 28, (unsigned char *)buffer, COLOR_WHITE, 0);
  disp_lcd_frombuffer();    // send to OLED
  delay(10);  // 10ms sample time
} // End of loop
```

What happens? Channels A0 and A1 are both samples and the resulting value is display in the unit of Volts on the display at two line positions. The allowed range for the voltage source is 0V to 5V. Attention don’t connect negative voltages to the input!

11. Digital/Analog-converter

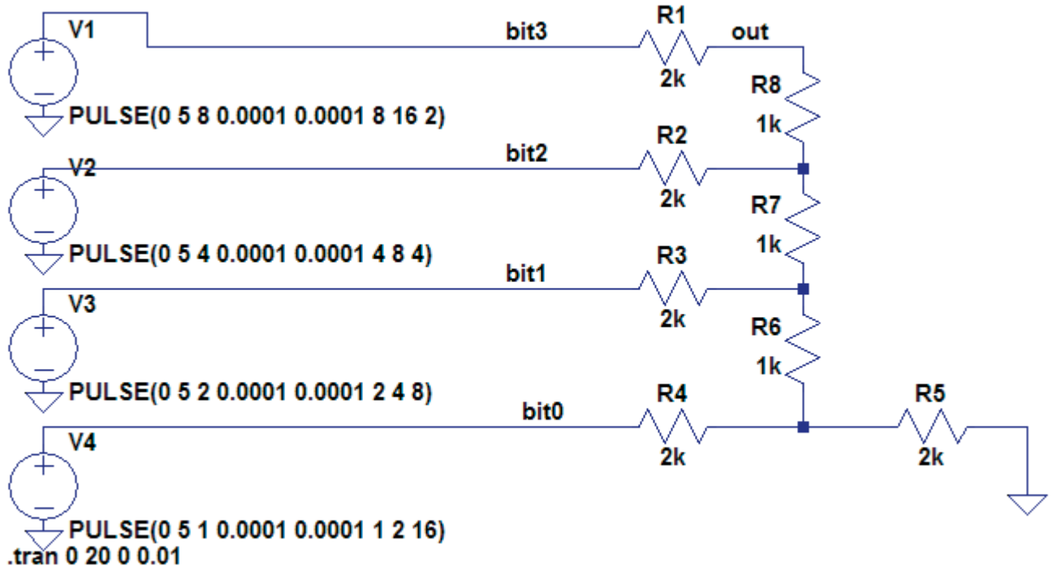
11.1 Digital/Analog-converter and principle

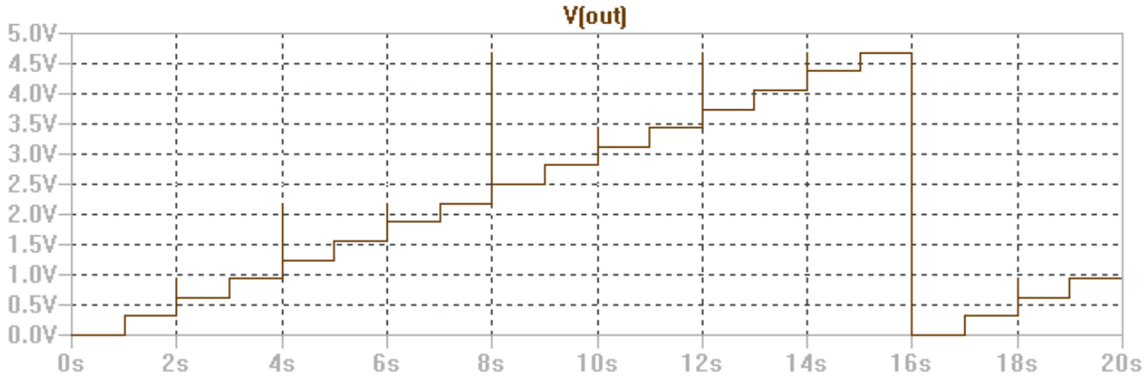
Until now we translated analog signals as voltages using an analog/digital-converter to digital values. But the reverse is also possible to translate digital values back into analog signals.
This converter is called digital/analog converter or with the shortcut D/A-converter



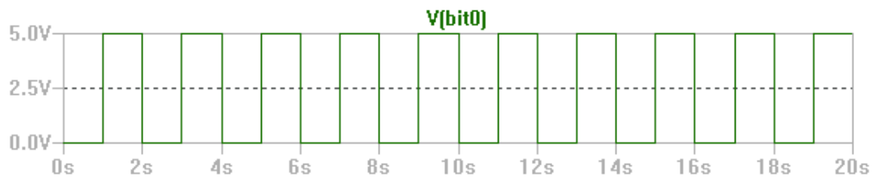
example: 0=0V 1=5V R=1000Ohm

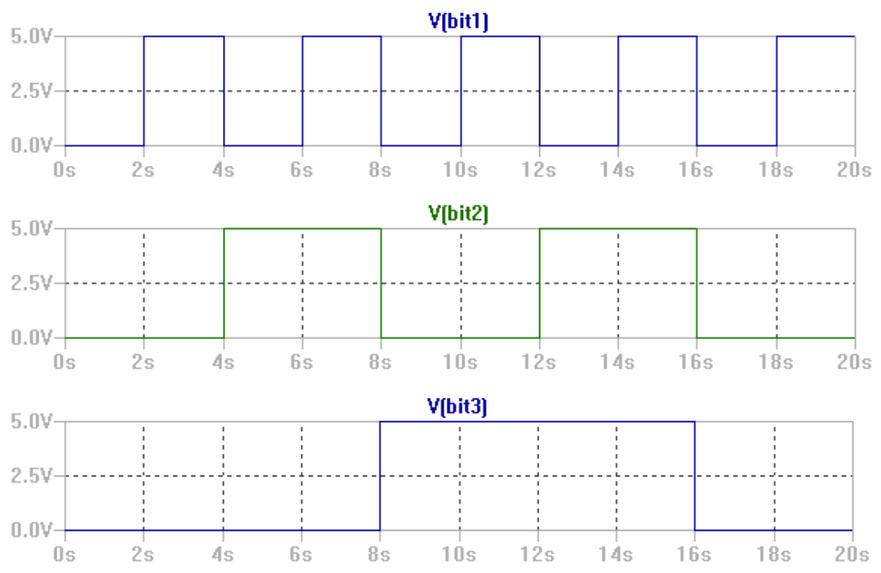
Above a very simple 4-bit-D/A-converter, which you can try to build yourself. Its called R2R ntwork. This is because only resistors of the value R oder 2R that is the double of the value R are used. You can use 1kOhm for R and 2kOhm vor 2R for example (could be a series of 2x 1kOhm). The resistor needs to be very precise, otherwise the total precision is not good enough.
The least significant bit is bit 0 and only changes very less in the output voltage. Bit 3 instead has the most influence on the total voltage at the output. Each bit can have either 0V or 5V for example.
Some special cases are easy to calculate. All bits are at 0V, then the output has 0V. If all bits are 1, e.g. have 5V, the output is not exactly 5V because of the single 2R resitor to ground.
Below you find a schematic for the free circuit simlator LTSpice (from Linear Technology) to see the effects.





Above are the simlation results. The simulated counter counts from 0 to 15 in cyclic way- The D/A-converter generates the proportional voltage at the output.
The small spikes you can see are the result of the not ideal components which have some delaytime like in the real world.



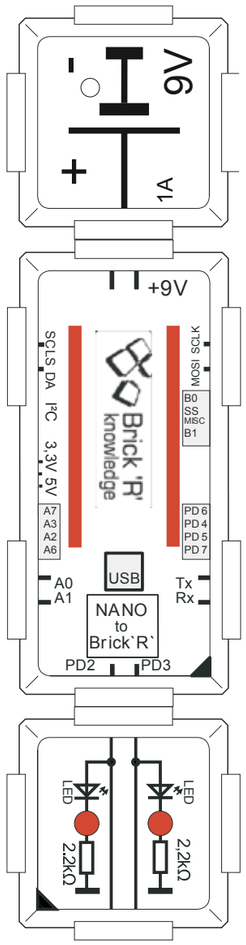


The voltage sources are programmed to output cyclic waves. Together the form a binary counter.

11.2 Simple D/A converter using PWM

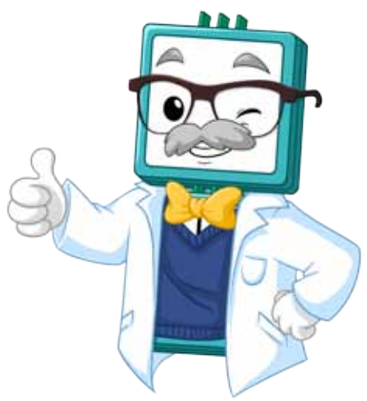
A simple D/A converter is already possible with the Nano brick directly. It's using the pulse wide moulation. The signal is still digital but the width of each pulse is proportional to a given value, which then can be converter to an analog signal.
With the help of a capacitor the pulswidth can be converted to an analog signal. Here we use a simple LED, in this case our eye is doing the conversion because it cannot follow the fast pulses and sees a mean value.

Not all ports can be used for the PMW, Nano allows it for the ports 3,5,6,9,10 and 11. The instruction „analogWrite()" is then used to do the conversion.



ports 3,5,6,9,10,11 PWM

```
// EN_32 DA example PWM

#include <Wire.h> // I2C library
#include <avr/pgmspace.h> // extra

#define DA3 3 // Port PD3

void setup() {
  pinMode(DA3, OUTPUT); // as output
}

void loop() { // loop all
  static int brightness=0; // LED brightness
  analogWrite(DA3, brightness);
  delay(20); // 20ms
  brightness++; // increment slow
  if (brightness>255) brightness = 0; // 0..255
}
```



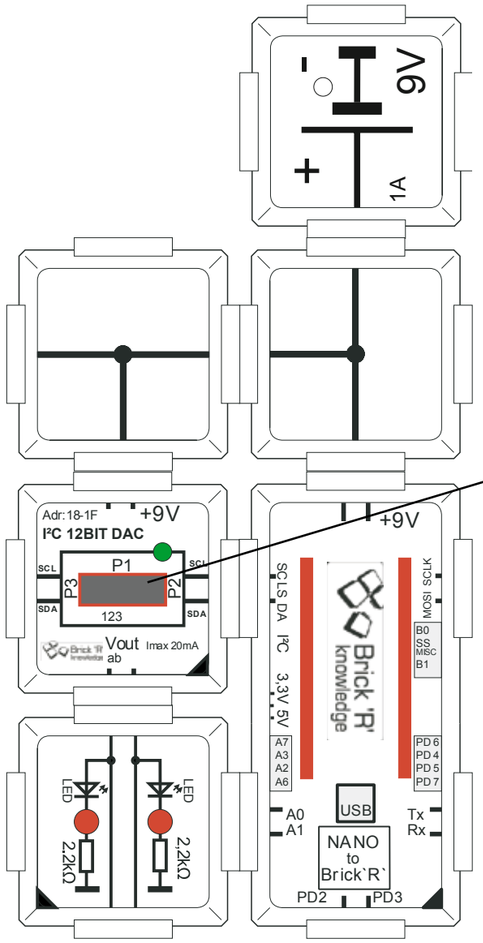
What happens? The LED at port PD3 starts being dark. Then the brightness is increasing to a final maximum value. After this all starts in a cycle. It will take some seconds (>256 * 20msec).

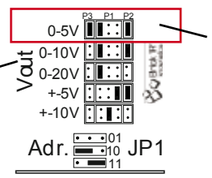
11.3 D/A-converter brick controlled by I2C

Our D/A converter bricks has a much higher resolution than the PWM can do. We have 4096 steps, that is 12 bit. With this example circuit also the brightness of the LEDs can be controlled.
The LEDs start lighting after a certain voltagelevel is reached, which depends on the color and technology of the LEDs. Its needs around 1.8Volt. Usually a current regulation would be better. But with the 2.2kOhm resistors the voltage is converted to a current as soon as the level is reached.
The formula applies:
I = (Uda - Uled) / 2200Ohm (*1).
If Uda is larger than Uled, the trigger level of the LED, the translation to current works quit well. If Uda < Uled, not current flows into the LED it remains dark.

Attention: The D/A converter bricks has internal voltage generators for -10V and +10V, which can be used by the proper jumper configuration on top. Here we use 0-5V for the output range, but also 0-10V can be used in this case. The output current is limited to 20mA, but always carefully check the jumper configuration as folloow up circuits can be damaged by inproper voltage ranges.
A print on the bottom side shows the possible combinations. If you open the brick, you can also set several addresses for the I2C, three of them can be used, 18 is disabled.

*1 Uda > ULed and Uled = minimum threshold level for the LED to be on





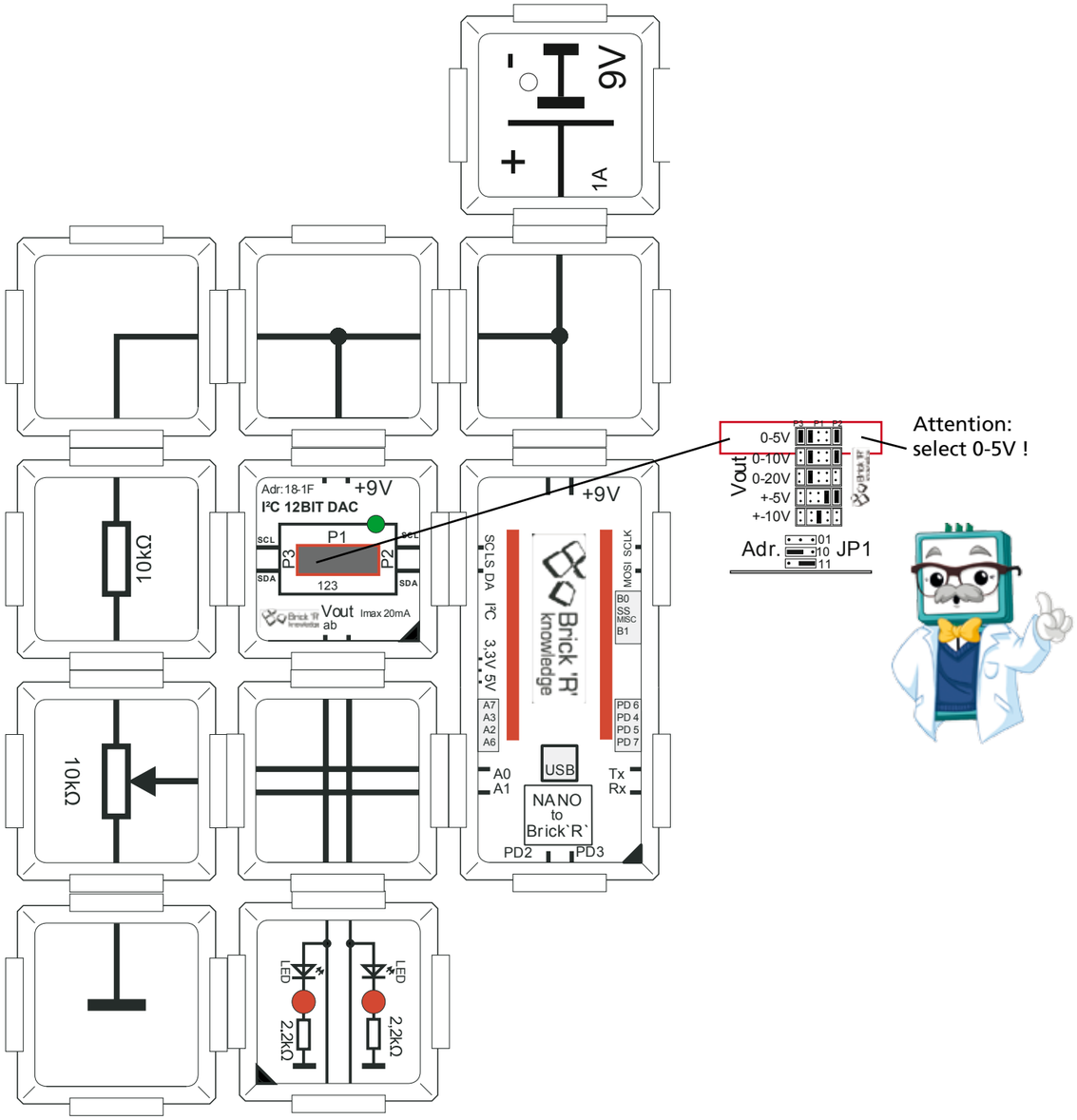
Attention:
Select 0-5V

```
/// EN_33 DA example D/A Brick I2Cs

#include <Wire.h> // I2C library
#include <avr/pgmspace.h> // extra

// 1E=GND,1c=Open,1a=VCC AD5622 open jumpers = 1Ch
// 0001 1aa0
// but only aa=01 10 11 are possible I2C addresses

#define i2cdasel1 (0x1a>>1)  // first of 1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // second of 1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // third of 1a,1c,1e


void i2c_da_write_command(unsigned char i2cbaseadr, unsigned short cmdvalue)
{
  // BIt15,14=0 13,12=pd (std=0) then 11..0 = DA Value
  cmdvalue = cmdvalue & 0xFFF; // 12 Bits valide // 0..4095 ranee
  Wire.beginTransmission(i2cbaseadr); // I2C Start
  Wire.write((cmdvalue>>8)&0xff); //  then MSB first
  Wire.write(cmdvalue&0xff);  //   LSB
  Wire.endTransmission(); // I2C end
}

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

void loop() {
  static int daval=0; // static value counted up
  // use all possible I2C addresses of the D/A bricks
  i2c_da_write_command(i2cdasel1,daval); // one after the other
  i2c_da_write_command(i2cdasel2,daval); //
  i2c_da_write_command(i2cdasel3,daval); // det
  delay(1); // 1ms for savety not to be too fast
  daval++; // next DA value
  if (daval>4095) daval = 0; // avoid overflow  0..4095
}
```

What happens? Both LEDs are dark at the beginning. Suddenly the brightness increases until a maximum is reached. Takes some seconds. One LED might be delayed a little bit if the voltage level is higher for this LED than for the other.

11.4 D/A converter brick and potentiometer

In this example a A/D converter is used to sample a value at the potentiomater and then sets this value to the D/A converter after recalculating the new range. The A/D converter of the Nano has a range of 0..1023 and the D/A converter uses 12 bit with 0..4095. That it we need to multiply the values of the A/D converter by 4 to get a full range of the D/A converter. Here we use the shift operation „<<“. The Value of 4095 is not exactly reached, only 4092 which is ok in this case.
Now when turning the potentiometer the brightness of the LEDs follow. You can also see the different voltage level for the LEDs to switch on. This depends which color and technology was used.




```
// EN_34 DA example D/A Brick I2Cs with Poti

#include <Wire.h> // I2C library
#include <avr/pgmspace.h> // extra

// 1E=GND,1c=Open,1a=VCC AD5622 open jumpers = 1Ch
// 0001 1aa0
// but only aa=01 10 11 are possible I2C addresses

#define i2cdasel1 (0x1a>>1)  // first of 1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // second of 1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // third of 1a,1c,1e


void i2c_da_write_command(unsigned char i2cbaseadr, unsigned short cmdvalue)
{
  // BIt15,14=0 13,12=pd (std=0) then 11..0 = DA Value
  cmdvalue = cmdvalue & 0xFFF; // 12 Bits valide // 0..4095 ranee
  Wire.beginTransmission(i2cbaseadr); // I2C Start
  Wire.write((cmdvalue>>8)&0xff); //  then MSB first
  Wire.write(cmdvalue&0xff);  //   LSB
  Wire.endTransmission(); // I2C end
}

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

void loop() {
  int daval=0; // temp sotrage
  int poti = analogRead(A0); // a1,a2,a3 also possible
  daval = poti << 2; // 0..1023 -> 0..4095  range adjust
  // use all possible I2C addresses of the D/A bricks
  i2c_da_write_command(i2cdasel1,daval); // one after the other
  i2c_da_write_command(i2cdasel2,daval); //
  i2c_da_write_command(i2cdasel3,daval); // det
}
```

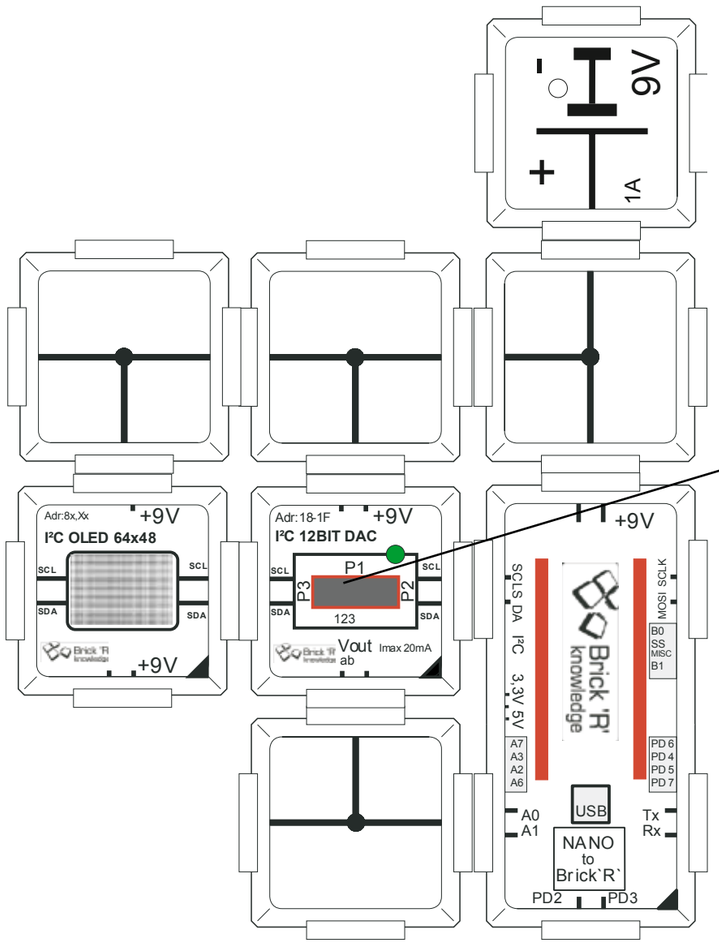
What happens? The brightness of both LEDs can be controlled by the position of the potentiometer. There is also a small dead area in which the LED voltage is below its threshold level.

11.5 OLED and D/A converter at the A/D converter

When combining the D/A-convert and the built in A/D-converter with an OLED some very interesting experiments can be done. Here for example we use the mini oscilloscope code and combine it with the rest. The D/A-converter is generating a ramp, that means a voltage from 0V to 5V. The value is measured with the A/D-converter and displayed as text or as a graphic. The resolution of the A/D-converter is not as high as the one from the D/A-converter, so some values are the same even the D/A-converter increased its value.
The example can be modified very easily, for example instead of „daval++“ a formula to derive the value of „daval“ can be used and therefore a total different sequence of values. Not limits to your phantasy.

Important: Check the jumper positions carefully, the output of the D/A-converter is directly connected to the A/D-converter without any additional protection. Bricks can suffer damage otherwise !!
Only 0V-5V is allowed for the inputs of the Nano bricks. The D/A-converter can output different ranges with different jumper settings.

Add some LEDs to the output to measure the threshold value of LEDs for example using our dual LED brick. You can connect it to the left side of the double T-brick.

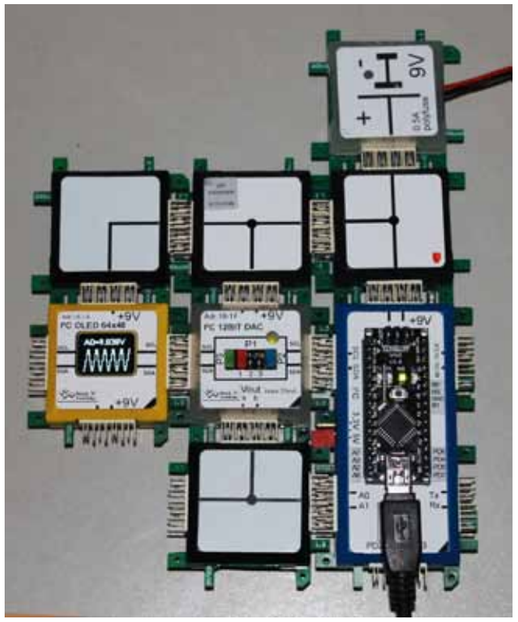


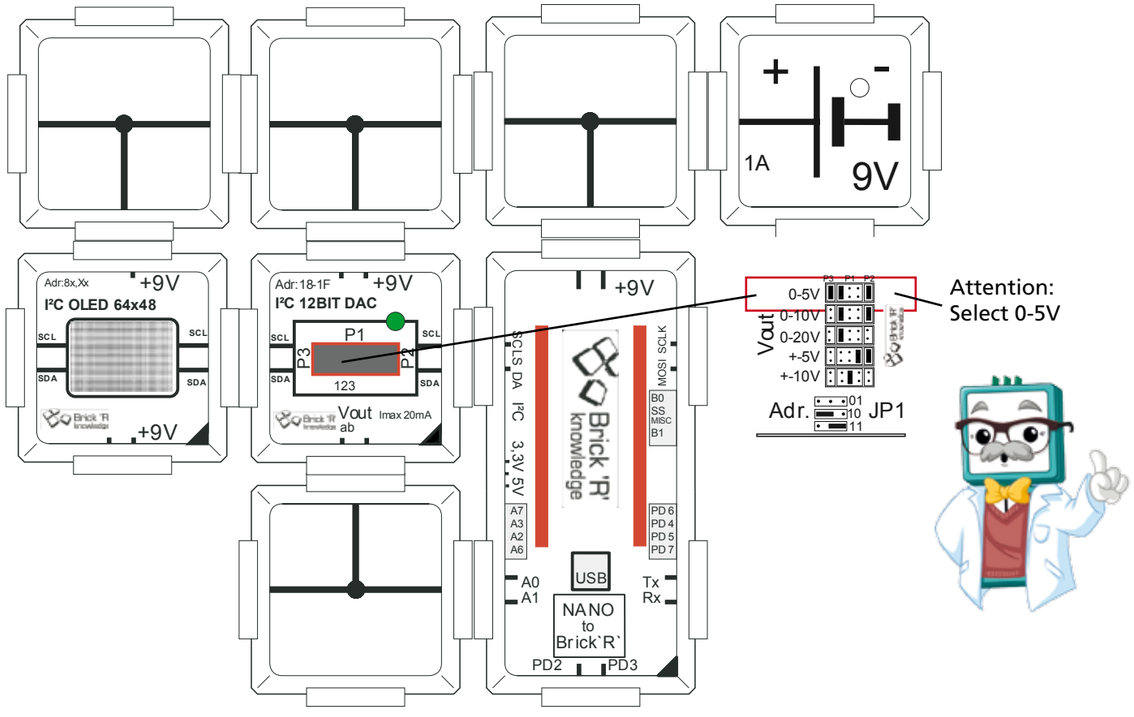
What happens? The OLED should display a ramp, which slowly scrolls through.

```
// EN_35 OLED example - DA converter and AD
converter

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

// set the according address
#define i2coledssd (0x7A>>1)  // default is 7A


// -------------------------OLED -----------
..... use library code from appendix
// -------------------------END OLED ------


// 1E=GND,1c=Open,1a=VCC AD5622 open jumpers
= 1Ch
// 0001 1aa0
// but only aa=01 10 11 are possible I2C
addresses

#define i2cdasel1 (0x1a>>1)  // first of
1a,1c,1e
#define i2cdasel2 (0x1c>>1)  // second of
1a,1c,1e
#define i2cdasel3 (0x1e>>1)  // third of
1a,1c,1e


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

char advalbuf[64]; // loop buffer cyclic

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


void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // cyclic counter
  static int daval=0; // DA value for
output
  int poti11 = analogRead(A0); // read
A/D
  char buffer[40]; //textbuffer
  disp_buffer_clear(COLOR_BLACK); //
clear it
  double  p1 = (poti11*5000.0)/1023.0;
//to mV
  int y1 =0; // output pos
  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; // store 0..4xx
V +-128
  disp_print_xy_lcd(2, 0, (unsigned char
*)buffer, COLOR_WHITE, 0);
  int i=0; // col counter
  int yold  = advalbuf[(cxx+1)%64]; //
last valuet
  for (i=0; i<63; i++) { // all columns
   y1 =advalbuf[(cxx+1+i)%64]; // cyc
buffer
   disp_line_lcd (i, yold, i, y1, CO-
LOR_WHITE);
   yold = y1; // save as old
  }
  if (cxx >63) cxx =0; // 0..63 ringbuf-
fer
  disp_lcd_frombuffer();   // to screenn
  // DA converter all values to D/A
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval++; // results in sawtooth
  if (daval>4095) daval = 0;
}
```

11.6 OLED and D/A converter at A/D converter with a sine signal

Here an example to generate a sine signal using the D/A-converter. The number of dots and therefore the frequency is determined by the variable „SINRESOL“. The frequency is also determined by the loop period. In this case the output rate of the OLED limits the frequency to be in the Hz region.






```
// EN_36 OLED example - DA converter and AD converter with sine

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

// set the according address
#define i2coledssd (0x7A>>1)  // default is 7A
// ------------------------OLED -----------------------------------------
..... use library code from appendix
// ++ DA code
char advalbuf[64]; // loop buffer cyclic

void setup() {
  Wire.begin(); // I2C init
  i2c_oled_initall(i2coledssd);  // OLED init
  for (int i=0; i<64; i++) advalbuf[i]=47; // clclic buffer
}

#define SINRESOL 10 // n dots
void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // cyclic counter
  static int phi=0; //phase for sine
  int daval = 0; // temp storage
  int poti11 = analogRead(A0); // read channel 0
  char buffer[40]; // textbuffer
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (poti11*5000.0)/1023.0; // get mV
  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; // store 0..4xx V +-128
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // counter for columns
  int yold  = advalbuf[(cxx+1)%64]; // oldest value
  for (i=0; i<63; i++) { // all columns
   y1 =advalbuf[(cxx+1+i)%64]; // cyclic bufferr
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1; // get new value
  }
  if (cxx >63) cxx =0; // 0..63
  disp_lcd_frombuffer();   // to OLED
  // DA converter gets sine
  daval = (int)(2048 + 2047*sin((double)phi*3.141592*2.0/(double)SINRESOL)); //
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  phi++; // new phsae
  if (phi>SINRESOL) phi = 0;
}
```

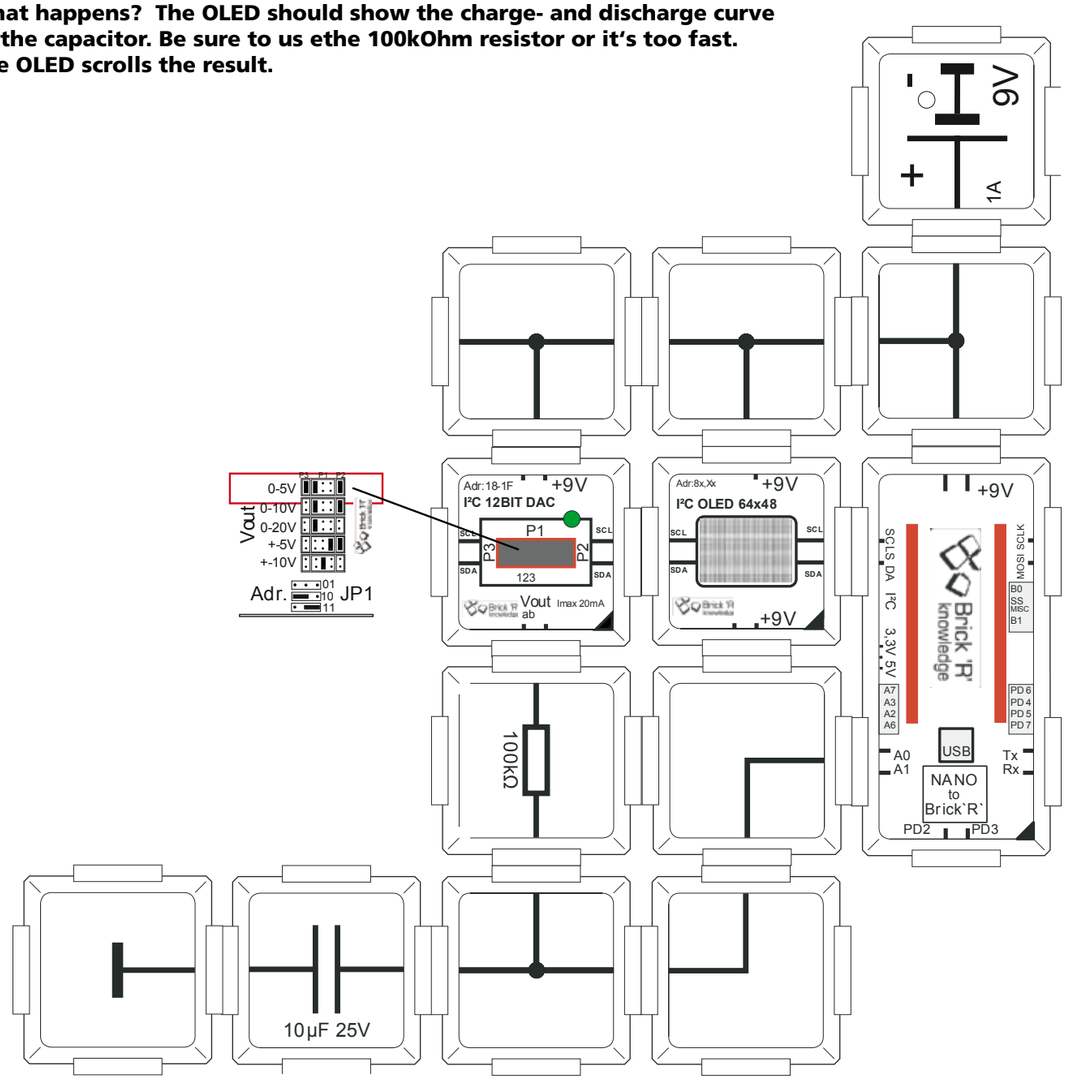
What happens? The OLED should display a sine curve slowly scrolling through.

12. Applications

12.1 Measurement of a discharge function - display on OLED

The capacitor is charged and discharged all 3 seconds. On our mini oscilloscope you can see the charge and discharge curve. To get an exact timestamp the function millis() is uses. This contains the time after starting the program in milliseconds (overflows every 54 days to zero). The values of the measurement are recorded like with the mini oscillsocope code before and also displayed in the same way.
To speed up the whole thing recording and display needs to be seperated. But in our example the speed is sufficient. Depending on the CPU around 1-2 charge and discharge cycles are shown on the OLED. The D/A-converter is set to 0-5V so it fits to the input range of the A/D-converter.
You can calculate the time constant for the charge and dischrge with the following formular:
t=R*C, we use 100000.0 Ohm * 10E-6 F = 1 sec. This is the time where the capacitor is charged for 63% or discharged.

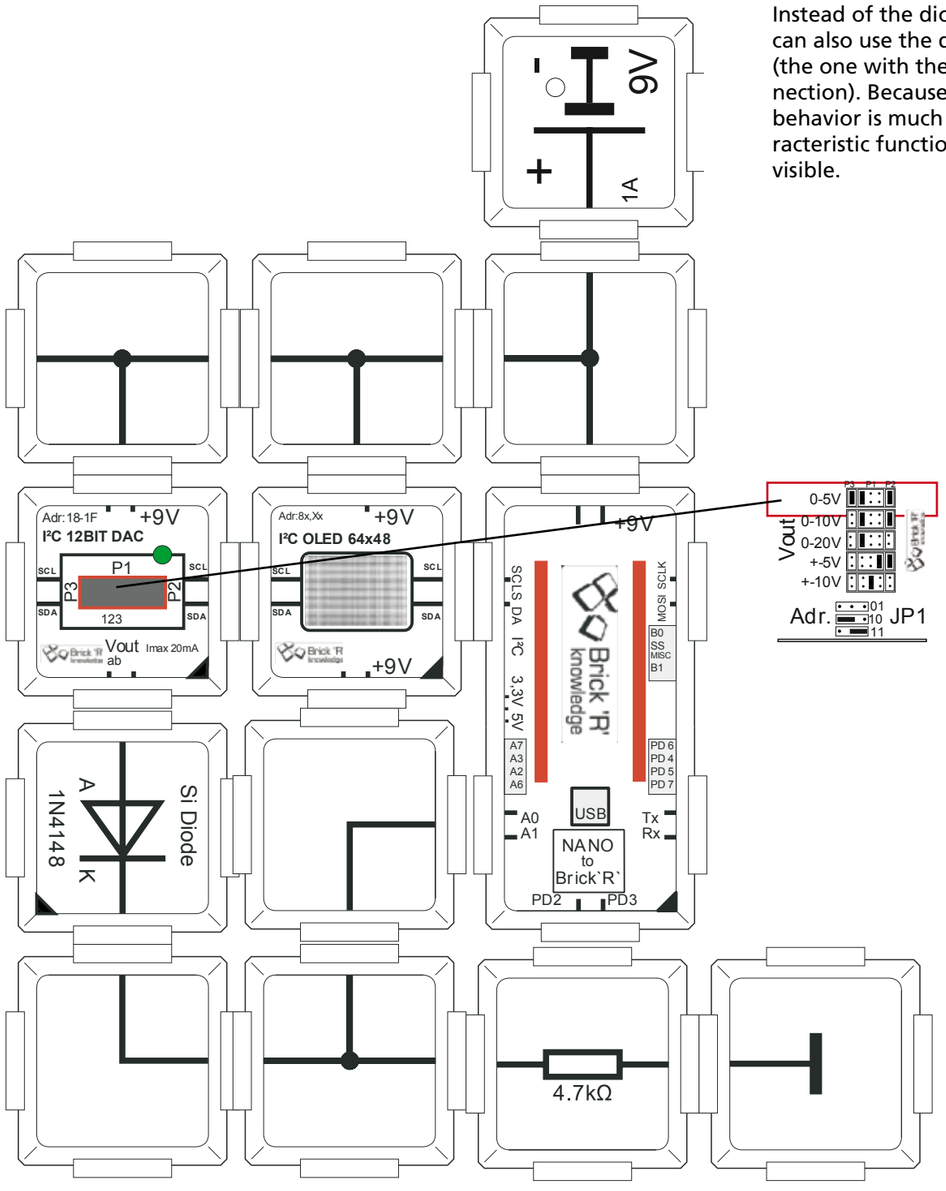
What happens? The OLED should show the charge- and discharge curve of the capacitor. Be sure to us ethe 100kOhm resistor or it's too fast. The OLED scrolls the result.




```
// EN 37 applications - dischargecurve
..... use library code from appendix and from previsous examples
int milisec=0; // save milisec
void setup() {
  Wire.begin(); // I2C init
  i2c_oled_initall(i2coledssd);  // OLED init
  for (int i=0; i<64; i++) advalbuf[i]=47; // default
}

void loop() {  //
 // 64x48 Pixel OLED
  static int cxx = 0; //cyclic counter
  static int state = 0; // use state system
  int ms = 0; // measurement time in ms
  static int daval = 0; // output value
  int ana0 = analogRead(A0); // read channel 0
  char buffer[40]; // textbuffer for Volt
  disp_buffer_clear(COLOR_BLACK); //  clear screen
  double  p1 = (ana0*5000.0)/1023.0;  // in mV
  int y1 =0; // pos 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; // store 0..4xx V +-128
  if (cxx >63) cxx =0;  // cyclic counter
  disp_print_xy_lcd(2, 0, (unsigned char *)buffer, COLOR_WHITE, 0);
  int i=0; // for columns
  int yold  = advalbuf[(cxx+1)%64]; // last value
  for (i=0; i<63; i++) { // for all columns
   y1 =advalbuf[(cxx+1+i)%64]; // from ringbuffer
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1; //set new value
  }
  disp_lcd_frombuffer(); // to screen
  // charge an ddischarge capacitor
  // use stair function
  // fixed interval measurement
  //
  ms=millis();   // current time
  if (ms > (milisec+3000)) { // wait 3sec
    switch(state) { // depends on state
      case 0:
        daval = 0xfff; // chagr
        state = 1; // then next state
      break;
      case 1: // then discharge
        daval = 0; // after 3sec
        state = 0; // repeat all
        break;
      default: // just in case
        state = 0; // 0 safe side
    }
    milisec = ms; // new time
  }
  // DA converter
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
}
```

12.2 OLED and simple diode characteristic diagram

Also a characteristic function of a electronic element can be determined using our measurement method. Here we show the positive part of a characteristic curve of the diode.
The current through the diode is measured using a resistor and measuring the voltage with the A/D-converter. The control voltage comes from the D/A-converter.
You can see the characteristic non linear curve of the diode for positive voltages.
Also negative voltages are possible, you need to reconstruct the circuit for example using an offset on the ground. Negative voltages also can be generated by the D/A-converter, but the A/D-converter only allows for positive voltages.



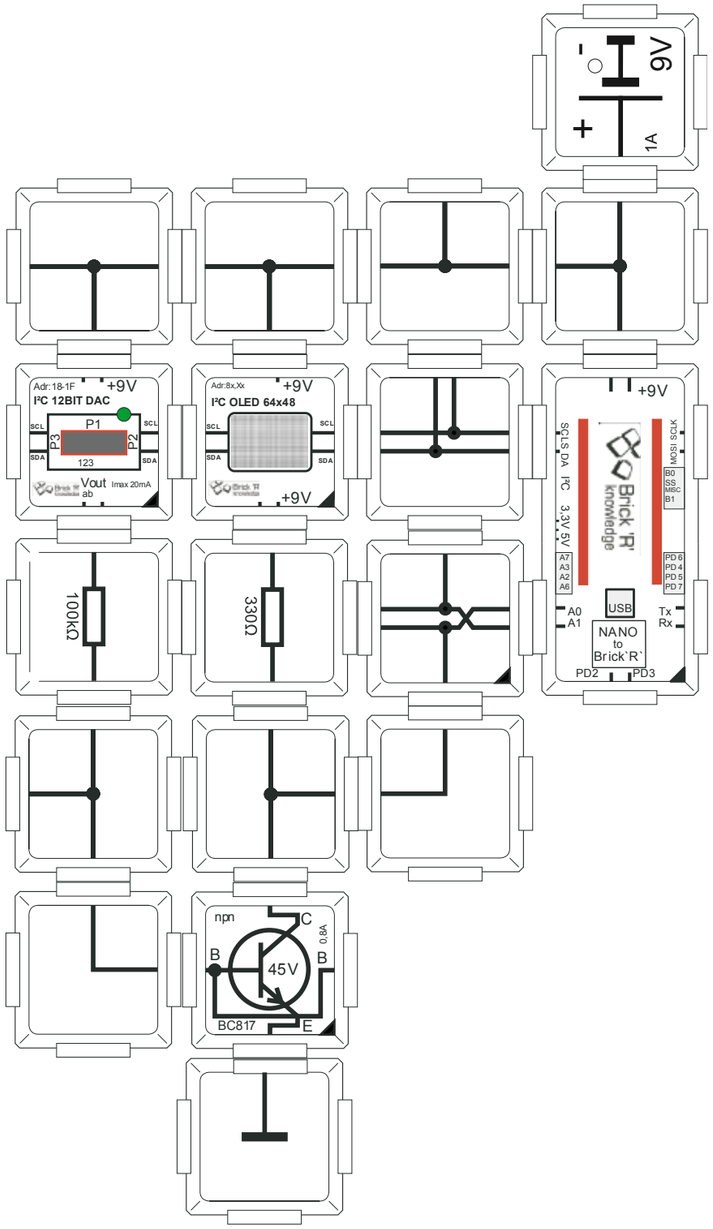
Instead of the diode 1N4148 you can also use the dual LED bricks (the one with the through connection). Because the non linear behavior is much stronger, the characteristic function is much better visible.

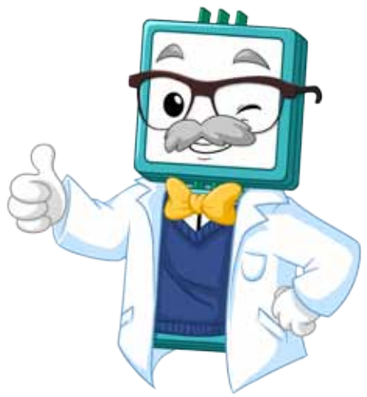
```
// EN_38 applications - simple diode characteristic
... as before

char advalbuf[64]; // loop

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

void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // cyclic buffer
  static int daval = 0;  // output
  int ana0 = analogRead(A0); // get channel a0
  char buffer[40]; // textbuffer
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (ana0*5000.0)/1023.0;
  int y1 =0; // Y pos
  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; // store 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++) { // all columns
   y1 =advalbuf[(cxx+1+i)%64];
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1;
  }
  disp_lcd_frombuffer();
  // DA sawtooth !
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval  += (4096/40); // to screen
  if (daval > 4095) daval = 0;
}
```

What happens? The OLED shows the characteristic positive part of a diode. It's also scrolling through.

12.3 OLED and transistor as common emitter circuit

This shows the characteristic curve of a transistor. Attention, the transistor uses a 100kOhm resistor at the base to limit the current. This current is amplified and a smaller resistors allows by measuring the voltage to see the current amplification.
The curve scrolls through, you can also implement a steady curve, by modifying the program.



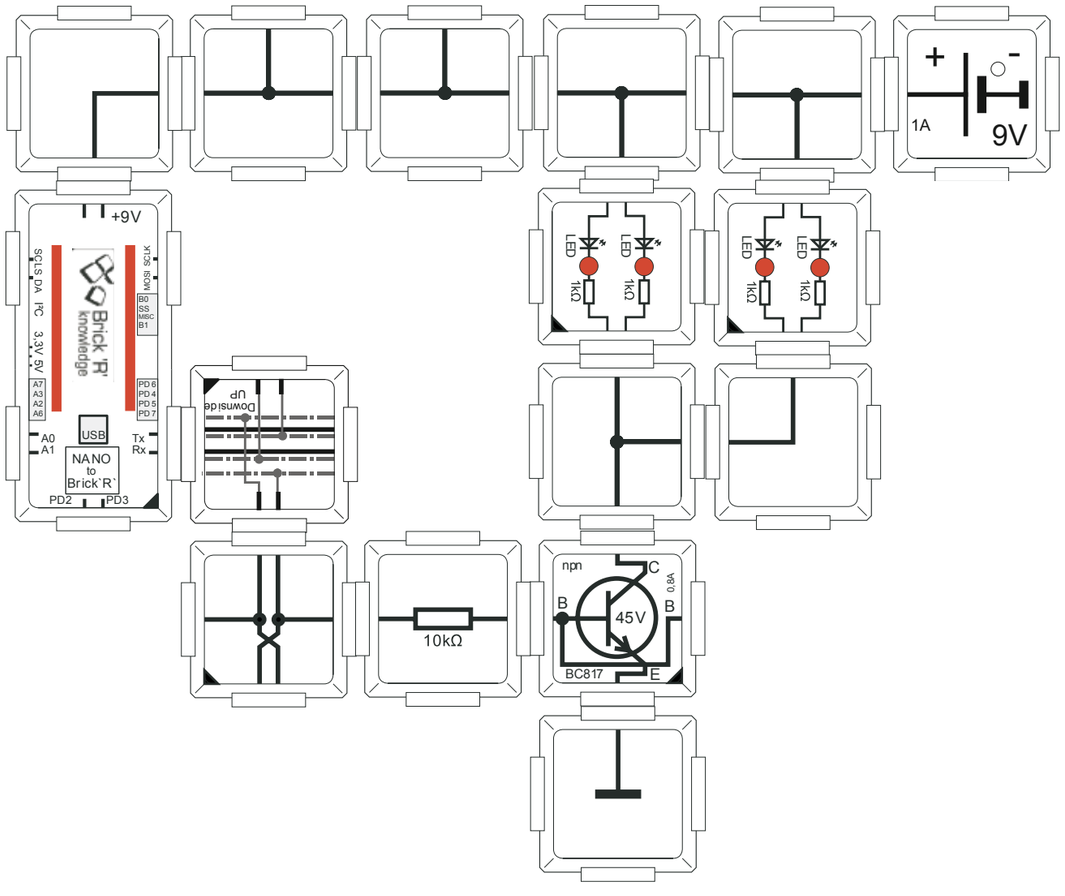


```
// EN_39 applications - transistor characteristic
... as before
//


char advalbuf[64]; // loop

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

void loop() {
 // 64x48 Pixel OLED
  static int cxx = 0; // cyclic buffer
  static int daval = 0;  // output
  int ana0 = analogRead(A0); // get channel a0
  char buffer[40]; // textbuffer
  disp_buffer_clear(COLOR_BLACK); //
  double  p1 = (ana0*5000.0)/1023.0;
  int y1 =0; // Y pos
  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; // store 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++) { // all columns
   y1 =advalbuf[(cxx+1+i)%64];
   disp_line_lcd (i, yold, i, y1, COLOR_WHITE);
   yold = y1;
  }
  disp_lcd_frombuffer();
  // DA sawtooth !
  i2c_da_write_command(i2cdasel1,daval);
  i2c_da_write_command(i2cdasel2,daval);
  i2c_da_write_command(i2cdasel3,daval);
  daval  += (4096/40); // to screen
  if (daval > 4095) daval = 0;
}
```

What happens? The OLED shows the typical characteristic curve of a transistor in common emitter circuit. Its scrolling through.

12.4 Switching of loads 1

Usually the output port of the Nano cannot drive more than 20mA. To switch loads which require a higher current an electronic switch is needed inbetween. This can be done using a transistor for example our BC817 which can usually handle around 500mA (peak 800mA), much more then the Nano itself.
Here some LED bricks are used as load. The transistor can turn them on or off. Therefore we use a small testprogram to let the LEDs blink.
Port PD7 is used as output, so we need the special brick to reach the contact on the bottom side. Check the orientation of this brick carefully.
The 10kOhm resistor at the base limits the current into the base.

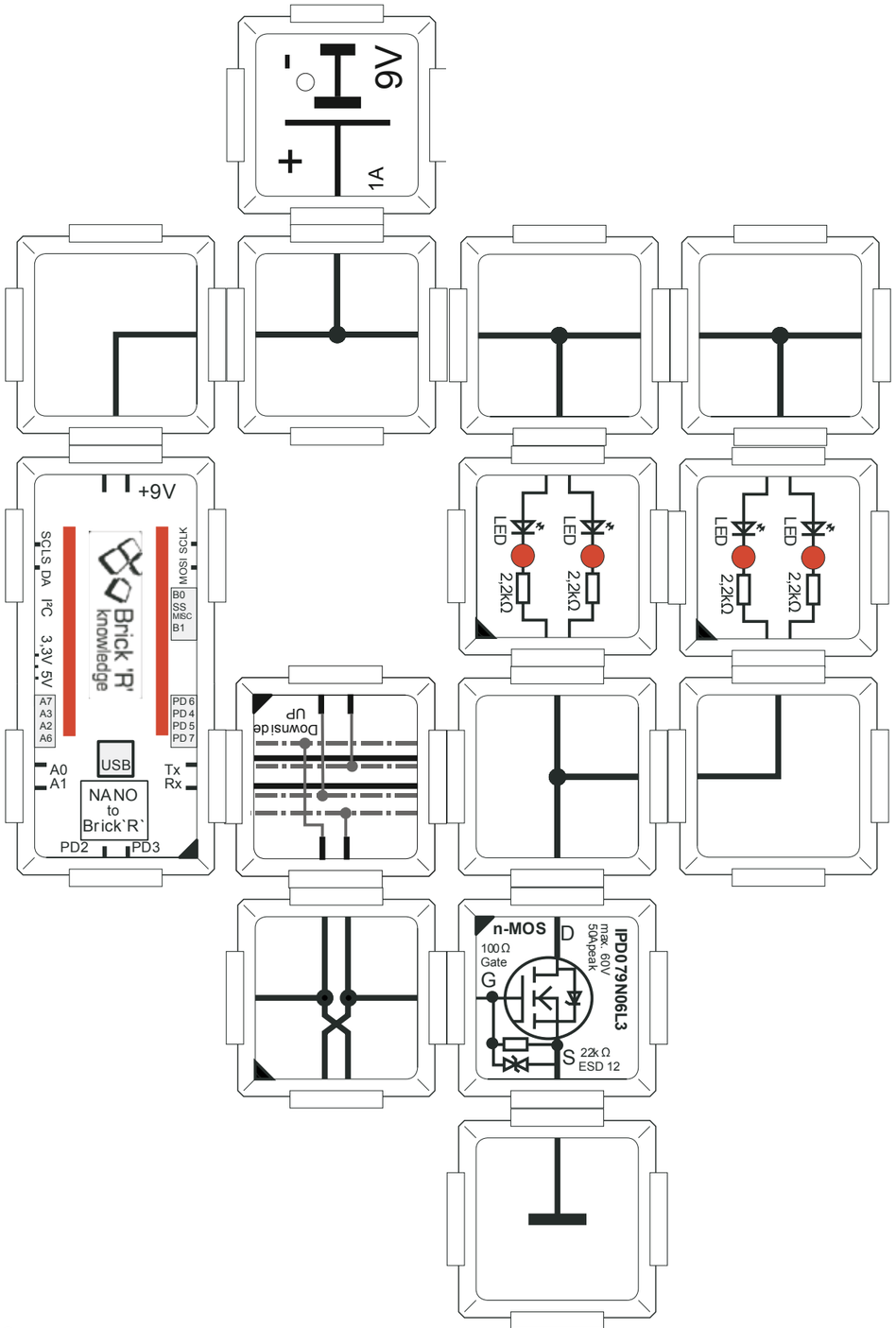



```
// EN_40 switching of loads 1

#define PORTLAST 7 // use PD 7

void setup() {
  pinMode(PORTLAST,OUTPUT); // output PD 7
}

void loop() { // Schleife
  digitalWrite(PORTLAST,HIGH); // set to high
  delay(1000); // 1 second delay
  digitalWrite(PORTLAST,LOW); // set to low
  delay(1000); // 1 second delay
}
```

What happens? The LEDs should turn on and off together in an interval of 1 second.

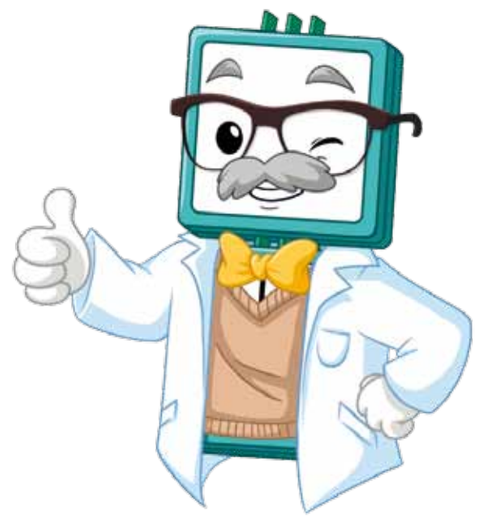
12.5 Switching of loads 2

Also MOSFETs can be used to switch high loads. Here the gate does not need a protection resistor as MOSFETs are controlled by voltage and no current should flow into the gate.




```
// EN_41 switching of loads 2

#define PORTLAST 7 // use PD 7

void setup() {
  pinMode(PORTLAST,OUTPUT); // output PD 7
}

void loop() { // Schleife
  digitalWrite(PORTLAST,HIGH); // set to high
  delay(1000); // 1 second delay
  digitalWrite(PORTLAST,LOW); // set to low
  delay(1000); // 1 second delay
}
```

What happens? The LEDs should turn on and off together in an interval of 1 second.



13. Appendix

13.1 Listing - seven segment library with an example

Here the complete code including the common library for the 7-segment-display. The library code is printed in black, the specific example in blue. Just copy the black part into the code mentioned with „...“ in the listings. The complete code example can also be found for download on the homepage.

```
//  EN_15 SevenSegementDisplay as I2C Brick
#include <Wire.h>

// standard built in: 8574T
#define i2cseg7x2alsb1 (0x40>>1) // 7 bit
#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)

// optional built in: 8574AT
#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)

// setup of the 7 segment assignments to the
// bits, 0x80 is for the DOT
// ************************
//          01
//       20    02
//          40
//       10    04
//          08
//                80
// ************************

// conversion table ASCII -> 7 segment
// OFFSET Asciicode 32..5F corresponds to
// Space up to 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
};

// convert ASCII Code to7-Segment table index
unsigned int get_7seg(unsigned char asciicode)
{
    // convert 0..255 to
    // 7 segment table index
    // only digits an capital letters
    // 20..5F
    // the rest will be mapped to the above
    asciicode = asciicode & 0x7f;     // mask 7 bit only
    if (asciicode < 0x20) return (0); // no special characters
    if (asciicode >= 0x60) asciicode = asciicode - 0x20; // map lower case letters
    return((~siebensegtable[asciicode-0x20])&0xff);      // return table index
}

// Display a single ASCCII character, which will be
// handed over as a character. Output over the 7 segment Brick
// In addition handover the segment addess as parameter.
void display_seg1x(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C address begin
  Wire.write(get_7seg(ch1)); // take table index and write it
  Wire.endTransmission();    // end I2C
}

// Output without conversion, if own characters shall be used.
// Parameter is the binary code
void display_seg1xbin(unsigned char i2cbaseadr, unsigned char ch1)
{
  Wire.beginTransmission(i2cbaseadr); // I2C address begin
  Wire.write(ch1); // write binary code direct to port
  Wire.endTransmission(); // end I2C
}


// Start only needed once
void setup() {
  Wire.begin(); // initialize I2C library
}

void loop() {
   // write all potential addresses of display 8574T,
   // to see which address is occupied
  display_seg1x(i2cseg7x2amsb1,'4'); // write own address
  display_seg1x(i2cseg7x2alsb1,'0'); //
  display_seg1x(i2cseg7x2bmsb1,'4'); // they are always COUPLES
  display_seg1x(i2cseg7x2blsb1,'4'); // two commands for one Brick
  display_seg1x(i2cseg7x2cmsb1,'4');
  display_seg1x(i2cseg7x2clsb1,'8'); // from 40 up to 4C
  display_seg1x(i2cseg7x2dmsb1,'4');
  display_seg1x(i2cseg7x2dlsb1,'C');
  // if 8574AT exists, write also to that
  display_seg1x(i2cseg7x2amsb2,'7'); // write own address
  display_seg1x(i2cseg7x2alsb2,'0');
  display_seg1x(i2cseg7x2bmsb2,'7'); // here
  display_seg1x(i2cseg7x2blsb2,'4'); // from 70 up to 7C
  display_seg1x(i2cseg7x2cmsb2,'7');
  display_seg1x(i2cseg7x2clsb2,'8');
  display_seg1x(i2cseg7x2dmsb2,'7');
  display_seg1x(i2cseg7x2dlsb2,'C');
}
```

13.2 Listing OLED library with example

Here the complete code including the OLED library. The library code is printed in black, the specific example in blue. Just use the black parts for code example with „...“.

```
#include <Wire.h>         // I2C library
#include <avr/pgmspace.h> // access to ROM

// ------------------------OLED ---------------------------------------------
// GLO066-D-M2005 -- SSD 1306 driver
// 011110sr  s=sa r=rw bei ssd1306 sa = adressbit to set optional
// if necessary adapt here the address  to 78 or 7A acording to switch
#define i2coledssd (0x7A>>1)  // default is 7A


// ------------------------OLED ---------------------------------------------
// GLO066-D-M2005 -- SSD 1306 driver
// 011110sr  s=sa r=rw bei ssd1306 sa = adressbit to set optional
// if necessary adapt here the address  to 78 or 7A acording to switch
#define i2coledssd (0x7A>>1)  // default is 7A

//
// ******************************************************************
// 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, 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, 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, 0x00, 0xE8, 0xB0,
  0x00, 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  -  Ω       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,      /*  Ω       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 !!! has to be dividable by 32 for
longs.
unsigned char lcdbuffer[LCDSIZE/8];


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

```
  Wire.beginTransmission(i2cbaseadr); // I2C address begin
  Wire.write(0x80); // 1000 0000 co=1 DC =0  command oder parameter for last command
  Wire.write(cmdvalue);
  Wire.endTransmission(); // end I2C
}


// ALL PIXEL ON or DATA FROM RAM
//------------------------------
void i2c_oled_entire_onoff(unsigned char i2cbaseadr, unsigned char onoff)
{
  if (onoff == 1) { // ALL PIXEL ON
    i2c_oled_write_command(i2cbaseadr,0xA5);
  } else { // DATA FROM RAM
    i2c_oled_write_command(i2cbaseadr,0xA4);
  }
}

// switch display on/off
//----------------------
void i2c_oled_display_onoff(unsigned char i2cbaseadr, unsigned char onoff)
{
  if (onoff == 1) { // switch on display
    i2c_oled_write_command(i2cbaseadr,0xAF);
  } else { // switch off display
    i2c_oled_write_command(i2cbaseadr,0xAE);
  }
}

// control display brightness
//---------------------------
void i2c_oled_setbrightness(unsigned char i2cbaseadr, unsigned char wert)
{
  i2c_oled_write_command(i2cbaseadr,0x81); // cmd for brightness
  i2c_oled_write_command(i2cbaseadr,wert); // 2. value is brightness
}

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

// INIT 64x48 display:
//--------------------
void i2c_oled_initall(unsigned char i2cbaseadr)
{
  // i2c_oled_display_onff(i2cbaseadr,0);  // FIRST TIME switch off
  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);  //  FIRST TIME switch off
  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 from top to bottom c0 bottom to top
  //
  i2c_oled_write_command(i2cbaseadr,0xda); // common pads hardware: alternative
  //
  i2c_oled_write_command(i2cbaseadr,0x12); // 12: OK, 32: the same, 02: data garbage
```

```
  //
  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 more flat, stronger , 11//22 less
strong
  //
  i2c_oled_write_command(i2cbaseadr,0xDB);   // COM deselect level
  i2c_oled_write_command(i2cbaseadr,0x40);   // 0.83*VCC according to datasheet from
legendary
  //
  i2c_oled_write_command(i2cbaseadr,0xA4);    // sisplay on; all pixel on
  //
  i2c_oled_write_command(i2cbaseadr,0xA6);   // set normal display a6=normal
a7=inverse
  //
  i2c_oled_write_command(i2cbaseadr,0xAF);   // enable display
  //
}


// lines 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
  //
  // THEN send data; switch over to data mode
  //

  //  at Arduino apparently max 32 bytes data
  // wirelib max 32 bytes buffer size !!
  // therefore dissassemby necessary !!

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

  // number of bytes must be dividable by 8 !!
  int j=0;
  int k=0;

// MAX LIMIT therefore two loops with 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();
  }
  //
}

// ATTENTION INTERLOCKED display...
void disp_lcd_frombuffer() {
    // 132 fix at the position !!
    // 64yx48
    // ATTENTION col 1..64-seg95..seg32  row 1..48 com32..com55
    // interlocked com 0..23 to 48..2 and com 32..55 to row 47..1
    // offsets in 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);

}

// clear buffer with color
//------------------------
// 0 oder ff
#define COLOR_BLACK 0
#define COLOR_WHITE 1      // special case

void disp_buffer_clear(unsigned short data) {
  unsigned char  *ptr = (unsigned char*)&lcdbuffer[0];
  unsigned char datal = 0;
  if (data >0) datal = 0xffffffff; // all on then at clear
  int i = 0;
  for (i = 0; i<LCDSIZE/8; i++) {
    *ptr++ = datal;
  }
}
```

```
// x,y
// at the same time organize row col
// 132 x 64 pixel at the same time 8 pixel in y-direction starting at 0 on a byte
// x horizontal y vertical

// 64 x 48 pixel !!

// set pixel on position x,y with color
//-------------------------------------
void disp_setpixel(int x, int y, unsigned short col1) {
  // COL = 0 dark = 1 bright
  // 1 aet pixel  (little endian)
  // col1 = 0 bright = 1 dark
  // table 8 bytes per line
  // entry 0 is the right pixelgroup
  // y=0 is bottom left, but physicaly from top to bottom (row 7 bit 7 -> y=0)

  unsigned char  *dest;  // destination pointer 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
  // SPECIAL x-axis 0..63 but y-axis dependant from even- or odd- line-number
  if ((y & 1) == 0) {
    // even line-number
    ymod = y & 0x7; // bit position 0..7
    yoff = y >> 3; // offset of the rows lines 0..7 (y>0 !!) bit 0..2 absent


  } else {
    // odd-numbered lines

  }
 #endif

  // calculate row col and byte pos.
  // y = 63 - y;  // thereby y=0 turned y=0 top left !!
  ymod = y & 0x7; // bit position 0..7
  yoff = y >> 3;  // offset of the rows lines 0..7 (y>0 !!) bit 0..2 absent

  dest = &lcdbuffer[yoff * LCDWIDTH + x]; // THEREBY Byte defined
  // now interchange bit 0..7 as efficient as possible
  if (col1 == 0) { // clear bit
   *dest &= ~(1<<ymod);
  } else { // set bit
   *dest |= (1<<ymod);
  }
  //
}

// set single character on position x,y with color
```



```
//-------------------------------------------------
unsigned short disp_setchar(int x, int y,  unsigned char chidx1, unsigned short color)
{
  // returns width of the letter as result (necessary for proportional letters)
  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; // number of lines
  int i = 0, v1 = 0, v2 = 0;
  int b1;
  //

  // width <=8 pixel then one byte
  // width >=8 pixel then two bytes (max width = 16 pixel)
  if (w <= 8) { // 1 byte max
    for (i = 0; i<14; i++) { // process all lines
      v1 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i);
      for (b1 = 0; b1<w; b1++) {    // run through every bits 1 byte per font
        if (v1 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + b1, y + i, color);
        }
      } // run through every bit
    }
  } else { // draw 2 bytes
    for (i = 0; i<14; i++) { // process all lines
      v1 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i * 2);     // run
through 1 byte per font
      v2 = pgm_read_byte_near(fontArial14h_data_tablep+offset + i * 2 + 1); // run
through 1 byte per fon
      // run through all bits
      for (b1 = 0; b1<8; b1++) {    // first half
        if (v1 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + b1, y + i, color);
        }
      } // run through all bits
      for (b1 = 0; b1<w; b1++) {    // rest of the bits
        if (v2 & (1 << (7 - b1))) { // big endian bits
          disp_setpixel(x + 8 + b1, y + i, color);
        }
      } // run through all bits
    }
  }
  return(w);
}

// set complete string on position x,y with color
//-----------------------------------------------
int disp_print_xy_lcd(int x, int y, unsigned char *text, unsigned short color, int
chset) { // 0=14 1=27
  // returns x position of last character
  int x1;
  x1 = x;
```

```
  if (text == 0) return(x);

    while (*text != 0) {
      x1 = x1 + disp_setchar(x1, y,  *text++, color); // set one character and add
letter width to x position
    }

  return(x1);
}

// draw line with color
//---------------------
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);
}

// draw rectangular with color
//----------------------------
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);
```



```
}

// draw filled rectangular with color with optionaler border
//-----------------------------------------------------------
void disp_filledrect_lcd(int x1, int y1, int x2, int y2, unsigned short col) {
  // disp_setpixel(x+b1,y+i,color);
  // quick fill
  //
  // x1,y1 and x2,y2 swap eventually
  unsigned short  *dest;    // destination pointer lcdbuffer with 8 bytes per row !!
(little endian coded)

  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;
  // ATTENTION calculate 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 ------------------------------------------
```

```
// the setup function runs once when you press reset or power the board
void setup() {
  Wire.begin();                 // initialize I2C
  i2c_oled_initall(i2coledssd); // initialize OLED
}


void loop() {  // start loop
  // 64x48 pixel OLED
  static int phase=0; // use oahse as scroll offset
  disp_buffer_clear(COLOR_BLACK); // clear virtal buffer
  disp_line_lcd (0,24,63,24, COLOR_WHITE); // x0,y0,x1,y1 to middle
  for (int i=0; i<63;i++) { // for each 64 rows
    int y=0; // sine curve calculation
    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); //0,0 is left upper adjust for -y
  } // all columns
  disp_lcd_frombuffer(); // now update display
  phase++; // next phase
  if (phase>=360)phase=0; // keep within 0..359 deg
}
```

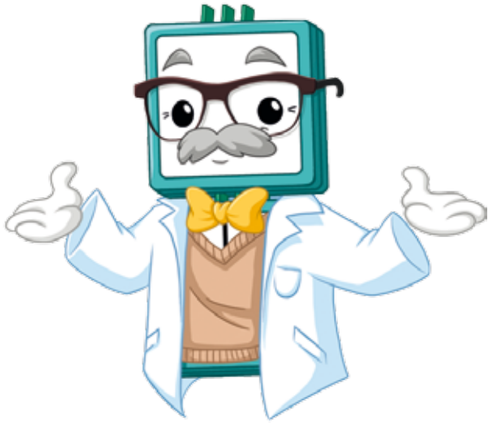
14. Future

The BrickRKnowledge system was developed and invented by Dipl. Ing. Rolf-Dieter Klein. He uses the ham radio callsign DM7RDK and educational callsign DN5RDK, member of DARC and ARRL as well as ACM and IEEE. The development was done originaly for the ham radio education for young people and for license eduction as well for industrial developments.

The author has a long term educational history, for example in tv series for public television (BR,BR-alpha,NDR) originally inventor of the NDR-Klein-computer, which was used for education in the pioneer times of microcomputing. The author wrote many book on this topic and is also holder of several industrial patents.

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