# CD-ROM FOR THE LEARNING PACKAGE

This learning package includes a CD-ROM that contains various programs and examples. The CD-ROM will make it easier for you to work with this book. The examples printed here are included on the CD-ROM.

## 1.1 | Contemt of the CD-ROM

P Arduino™ developer environment (IDE)

P Example program code for the learning package

P

## 1.2 | GPL (General Public License)

You can exchange your own programs with other users online. The example programs are subject to the Open-Source license GPL (General Public License). Therefore, you are entitled to modify, publish and provide the programs to other users under the conditions of the GPL, provided that you subject your programs to the GPL as well.

## 1.3 | System Requirements

As of Windows 7/8/8.1, 32 and 64 bit or higher, Linux 32 and 64 bit, Mac OS X, CD-ROM-drive, Java

> **Note**
> This learning package includes VB.NET-programs that work only under Windows. The basic Arduino™-programs for these experiments work on other operating systems as well. Only the .NET-PC-programs need a Windows operating system with .NET Framework for experimentation.

### 1.4 | Updates and Support

Arduino™ is subject to continuous further development. Updates can be downloaded free of charge from the web-site **http://arduino.cc**

# CONTENT OF THE LEARNING PACKAGE

The learning package includes all parts that you need for the experiments. Please check the parts for completeness before starting the experiments!

**Note**
The Arduino™-UNO-micro controller PCB is not included in the delivery. Since this is a learning package targeted at the advanced Arduino™ programmer, an Arduino™ UNO or MEGA should already be present at the workplace. The boards are available at low costs at Conrad Electronic, in the Franzis Shop or at other online businesses.

**Parts List**
1 x breadboard Tiny, 1 x LCD 16 x 2 blue, 1 x pin strip 16 pins for soldering in, 1 x button, 1 x NTC 472, 1 x photo transistor, 1 x 10 kΩ, 1 x 2.2 kΩ, 1 x 330 Ω, 14 x jumpers

### 2.1 | Safety Information

The Arduino™ PCBs and the display are mainly secured against errors, so that it is hardly possible to damage the PC. The connections of the USB-socket are not insulated on the bottom of the PCB. If you place the PCB onto a metal conductor, there may be a higher current, which may damage the PC and the PCB.

Observe the following safety rules!

- Avoid metal objects under the PCB or insulate the entire bottom with a non-conductive protective board or insulating tape.
- Keep mains units, over voltage sources or live conductors with

more than 5 Volt (V) away from the experimenting PCB.

PIf possible, do not connect the PCB to the PC directly, but via a hub. This usually includes an additional effective protection circuit. If something happens anyway, the hub, and not the PC, will usually be damaged.

# THE PARTS AND THEIR FUNCTION

The parts of the learning packages are presented here and the respective functions are explained briefly. The following experiments will provide the practical experience with the circuit technology of the electronics.

## 3.1 | Breadboard

On the breadboard, you can set up your circuits without soldering. Our breadboard is made up of 17 columns and 5 rows. The columns with 5 contacts each are connected to each other in a line (from top down, see figure). The separating bridge in the middle of the breadboard marks that no connection to the other field of 17 columns and 5 rows is present. It has proven to be helpful to uncoil the connection wires of the parts diagonally first, to produce a kind of wedge at the wire ends. This makes it easier to plug the parts into the breadboard. If it is difficult to plug in the parts anyway, best use small high-precision mechanic flat nose pliers to push the part into the breadboard with a little more pressure.

## 3.2 | Jumpers

The learning package includes several pre-customised jumpers. They are used for connections between the breadboard and the Arduino™-PCB. The jumpers have a small pin on both sides that can be pushed into the experimenting board and the Arduino™-PCB easily. However, be careful anyway so that no pin will accidentally break off or bend!

## 3.3 | Buttons

A button has a similar function as a switch. You already know switches from switching light on and off in the apartment. When we push the rocker down, the light comes on. When we push it up, the light goes out again. A switch remains in its position. This is different for a button. When we push the button, the circuit is closed. It will only remain closed while we push the button. When we release it, it will open the circuit again and the light will go out. The button will automatically return to its original or resting position with its internal mechanics.

There are buttons that close the circuit when actuated, and such that open the circuit . Buttons that close a circuit are often called »N. O.« (normally open) and those that open the circuit »N. C.« (normally closed). The figure shows a button that is enclosed with the learning package. It closes the circuit when pushed and the current can then flow from contact 1 to 2. The two other contacts each are connected with each other.

## 3.4 | Resistors

Resistors are needed to limit current and to set working points, or as voltage dividers in electrical circuits. The unit for electrical resistance is Ohm (Ω). The prefix Kilo (k, thousand) or Mega (M, million) shortens the way of writing large resistances.

1 kΩ = 1000 Ω

10 kΩ = 10,000 Ω

100 kΩ = 100,000 Ω

1 MΩ = 1,000,000 Ω

10 MΩ = 10,000,000 Ω

In circuit diagrams, the symbol Ω is usually left out and 1 kΩ is shortened to 1 k. The value of the resistor is applied to the resistor in the form of a colour code. Usually, there are three coloured rings, and an additional fourth ring that indicates the accuracy of the resistor. Metal film resistors have a tolerance of only 1 %. This is indicated by a brown tolerance ring that is a little wider than the other four colour rings. This is to prevent mistakes for a normal value ring with the meaning »1«.

Resistors with a tolerance of +/--5 % are available at the values of the E24-series, with each decade containing 24 values at about equal distance to the adjacent value.

The resistors of the E24 standard series are as follows:

1.0 / 1.1 / 1.2 / 1.3 / 1.5 / 1.6 / 1.8 / 2.0 / 2.2 / 2.4 / 2.7 / 3.0 / 3.3 / 3.6 / 3.9 / 4.3 / 4.7 / 5.1 / 5.6 / 6.2 / 6.8 / 7.5 / 8.2 / 9.1

The colour code is read from the ring closer to the edge of the resistor. The first two rings represent two digits, the third ring the multiplier of the resistance value in Ohm. A fourth one indicates the tolerance.

A resistor with the colour rings yellow, violet, brown and gold has the value 470 Ω at a tolerance of 5 %. Try to identify the resistors of the learning package right away.

**Advice**
Entering the search term »Resistance code calculator« online will return many resistor colour code calculators, e.g. under http://www.ab-tools.com/de/software/resistancesrechner/ or http://www.dieelektronikerseite.de/Tools/resistancesrechner.htm.

There also is an old version as shown in the figure below. This »resistance gauge« or vitrometer permits quick determination of the resistance without a computer just by turning the colour wheels. This way, you will memorise the colour codes much more quickly than in the computer version.

### 3.5 | Temperature Sensor

To record temperatures, an NTC-temperature sensor is enclosed. The designation NTC means »negative temperature coefficient« and says that the resistance will drop when the temperature rises. This is a hot conductor. The NTC in the learning package has a resistance of 4.7 kΩ at 25 °C/298.15 K (Kelvin).

### 3.6 | Photo Transistor

To determine brightness, modern electronics often use photo transistors. The learning package includes a part that looks very similar to a white light emitting diode, except that it is a photo transistor. It not only looks different from the normal bipolar transistors but also has no base connection. The base, i.e. the input of a normal transistor that is responsible for the current control between the collector and emitter is the light falling into the housing in a photo transistor. The light hits the silicon there and makes a lower or higher current flow between the collector and emitter depending on the light strength.

### 3.7 | LC-Display

The main actor of this learning package is the blue-white LCD. The learning package uses an LCD with two rows 16 columns with 5 x 8 dots each. These displays can now also be purchased separately in any good electronics store or online store for a few Euro. They are available in green, blue, amber, yellow and a few special colours that are usually more expensive. In our case, a blue LCD is installed. The LCD controller installed is a KS0066/HD44780 that is produced by many manufacturers - more on this later.

Before you can use the LCD in the experiments, you need to solder the enclosed 16-pin pin strip into the contacts of the LCD. For this, plug the pin strip with the short contacts into the LCD from behind and solder only one contact on first. This way, you can align the pin strip cleanly at a 90°-angle. When the pin strip has been aligned, you can solder on the other pins. If you have no soldering gun yet, get a cost-efficient hand-held soldering gun with an output between 20 and 30 W and electric wire solder. This investment will pay off when dealing with Arduino™ and electronics in any case.

# FIRST FUNCTION TEST

Wire your first experiment as shown in the figure. Be careful to not bend or even break the pins of the jumpers.

At the end, check the circuit for accuracy again carefully to avoid damage to parts.

**Info**

If you are working with Arduino™ for the first time, you need to download the Arduino™ developer environment first. You can find it on the official Arduino™-websitehttp://www.arduino.cc.
Here, you can select your operating system and determine if you want to use the installer or the Zip-version. In the installer version, you install Arduino™ like a normal standard program. In the Zip-version, you need no installation. Unzip the Zip file and save it in a desired storage site on your computer. This has the benefit that you can save the Arduino™- e.g. on a USB drive and take it along anywhere.

**Attention**

Only save Arduino where you have all rights to read, write, etc.!

For the first function test, load the following program onto the Arduino™-board. The example programs can all be found on the enclosed CD in the Examples folder.

The program will make a text and a kind of counter appear on the LCD and is well suitable as the first function test to check that everything is working properly because it is very small and well-structured.

Example code: LCD

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 void setup()
009 {
010     // LED-Backlighting
011     analogWrite(9, 150);
012
013     // LCD output
014     lcd.begin(16, 2);
015     lcd.setCursor(0, 0);
016     lcd.print("**ARDUINO LCD**");
017     lcd.setCursor(0, 1);
```

```
018     lcd.print("CNT:");
019 }
020
021 void loop()
022 {
023     lcd.setCursor(5, 1);
024     lcd.print(millis() / 1000);
025 }
```

In the first line of the program, you can see that operation of the of the LCD requires integration of the Arduino™ Library with the name LiquidCrystal.h. It includes the more complex code that is needed to control the display. You can look in the Arduino™-folder, under Arduino\libraries\LiquidCrystal, and check out the LiquidCrystal.h and LiquidCrystal.cpp files to get an idea of the function of the Library. To open these files, it is recommended to use, e.g., the program Notepad++. You can download it free of charge from http://www.notepad-plus-plus.org.

You will see that this Library will do a lot of work for you that other programmers have already completed. In our Arduino™-program, we integrate only the header file with LiquidCrystal.h. Arduino™ will now automatically know all LCD functions.

In the next line, we inform Arduino™ which pins of the LCD are connected to the Arduino™-PCB.

```
001 LiquidCrystal lcd(11, 10, 2, 3, 4, 5)
```

The next command determines the brightness of our display backlighting. The LED of the LCD is connected to the Arduino™ digital/PWM-port D9. It can be used as a simple digital port or a PWM (pulse-width modulated port). In our tests, we use it as a PWM-port. Thus, we can set the brightness of the backlighting gradually. The value 150 already makes the LED shine sufficiently brightly. If the PWM-value is chosen lower, the LED will be darker. Try changing the value and observe what happens.

```
001 analogWrite(9, 150)
```

Initialisation is almost completed. Now you need to indicate how many columns and rows the LCD has: 16 columns/individual characters and 2 rows.

```
001 lcd.begin(16, 2)
```

The basic initialisation is now completed. Now we can use lcd.setCursor to determine the position of the cursor and thus the text to be output.

```
001 lcd.setCursor(0, 0)
```

The first parameter indicates the position within the row, i.e. 0 to 15 in this case. The second parameter indicates the row number, i.e. 0 or 1.

Now we can output the text in the specified position at the LCD with the commandlcd.print.

```
001 lcd.print("**ARDUINO LCD**”)
```

We can also see that we always need to write »lcd.« before the actual function of the LCD output. This specifies that we use the class lcd that we have integrated with #include <Liquid-Crystal.h>. Now Arduino™ knows where the call comes from and which class is responsible for it when »translating«, referred to as »compiling« by specialists.

If you have ever dealt with the programming language C++ before, you will recognise by the ending *.cpp, that these are C++-classes. Arduino™ is basically based on C++. This is a good way of programming own classes or Library and providing them to other Arduino™ users.

After this brief C++-excursion, let us return to our example. Up to now, we still remained in the function `Setup()`-which is gone through once at all times when starting the program and that is mostly used for start configuration. In it, we can pre-initialise variables before the actual program start and pre-configure the hardware.

The following `Loop()` function is an endless loop that is never ended. This is the Arduino™-main loop for our program at the same time. Here, we call the runtime in milliseconds at each run with the function `millis()` . Division by 1,000 will lead to the output in seconds. We will represent the program runtime in seconds on the LCD.

```
001 lcd.setCursor(5, 1)
```

```
002 lcd.print(millis() / 1000)
```

Since the function `millis()` is very interesting, we will try another experiment before dealing with the LC-display in more detail, since the function `millis()` can also be used to measure time of program runs, as shown in the following example .

Example code: TIME_DIFF

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 long time_diff, diff;
009
010 void setup()
011 {
012     // LED-Backlighting
013     analogWrite(9, 150);
014
015     // LCD output
016     lcd.begin(16, 2);
017     lcd.setCursor(0, 0);
018     lcd.print("**ARDUINO LCD**”);
019     lcd.setCursor(0, 1);
020     lcd.print("TIME DIFF: ");
021 }
022
023 void loop()
024 {
025     diff = millis();
026
027     // Myprogram start
028
029     lcd.setCursor(11, 1);
030     lcd.print(diff – time_diff);
031     delay(100);
032
033     // My program end
034     time_diff = diff;
035 }
```

The example code shows how to determine the program runtime . For this, we read the current counter status of `millis()` at every

new program run and subtract the last counter reading that was saved at the end of the program. The `delay(100)` at the end of the program simulates a longer program runtime. Your program code must be between // My program start and // My program end , to determine the throughput time of your program code.

The first experiments and thus also the function test have hereby been completed successfully. Leave the circuit set up as it is. You will need it and expand it in the following experiments. In the following chapters, you will learn a bit more about the LC-display and its properties.

# SETUP AND FUNCTION OF THE LC-DISPLAYS

LCDs are used in many electronic devices, such as entertainment electronics, measuring devices, mobile phones, digital clocks and calculators. Head-up-Displays and video projectors also use this technology. The following figure shows the LCD from the learning package. This is a standard-5x8-dot-matrix-display with 2 rows with 16 characters each.

An LCD generally consists of 2 individual glass panes and a special liquid in between. The special characteristic of this liquid is that it will turn the polarisation level of light. This effect is influenced by applying an electrical field. The two glass plates are therefore vaporised with a very thin metal layer each. To get polarised light, a polarisation film is stuck to the upper glass plate. This is called the polariser. Another such film must be applied to the bottom glass plate, with its polarisation level turned by 90°. This is the analyser.

In the resting condition, the liquid turns the polarisation level of the incoming light by 90°, so that it can pass the analyser unhindered. The LCD is thus transparent. Applying a specific voltage to the vaporised metal layer now will cause the crystals to turn in the liquid. This will turn the polarisation level of the light by, e.g., another 90°: The analyser blocks out the light; the LCD has become opaque.

## 5.1 | Polarisation of Displays

Polarisation in LC-displays does not mean polarity of the voltage supply, but the gas, liquid and filter structure of the display.

Most LCDs are TN-displays (Twisted-Nematic-displays). They contain a liquid that turns the polarisation level of light by 90°. STNs (Super-Twisted-Nematics) turn the polarisation level of light by at least 180°. This improves the display's contrast. However, this technique will lead to a certain colouration of the display. The most common colourations are called yellow-green and blue mode. A grey mode appears more blue than grey in practice. To compensate for the undesired colour effect, FSTN-technology uses another foil on the outside. The resulting light loss, however, makes this technology only sensible for lit displays. The different colours appear only in unlit or white-lit displays, however. Once the lighting is coloured (e.g. LED-lighting yellow-green), the respective display colour moves to the background. A blue-mode-LCD with yellow-green LED-lighting will always look yellow-green.

## 5.2 | Static Control, Multiplex Operation

Small displays with low display scope are usually controlled statically. Static displays have the best contrast and the maximum possible angle. TN-technology fully meets its purpose here (black-white display, cost-efficient). However, as the displays grow, more and more lines would be needed in static operation (e.g. 128 x 64 graphics = 8,192 segments = 8,192 lines). Since this number of lines would not fit on the display, nor a control-IC, Multiplex operation is chosen. The display is structured in rows and columns and a segment is located at each crossing point (128 + 64 = 192 lines). Here, row by row is scanned (64 x, i.e. multiplex rate1 : 64). Since only 1 row at a time is active, however, the contrast and also viewing angle suffer with increasing multiplex rate.

## 5.3 | Viewing Angle 6 O'Clock /12 O'Clock

Every LC-display has a preferred viewing direction. Viewed from this direction, the display has the best contrast. Most displays are produced for the 6 o'clock viewing angle (also: bottom view, BV). This angle corresponds to that of the calculator lying flat on the table. 12 o'clock displays (top view, TV) are best integrated into the front of a table unit. All displays can be read vertically from the front.

## 5.4 | Reflective, Transflective, Transmissive

Reflective (unlit) displays have a 100%-reflector on the rear. Lighting from the rear is not possible. Transflective displays have a partially permeable reflector on the rear. They can be

read with and without lighting. This makes them unlit, but a little dimmer than a reflective version. Nevertheless, it is probably the best compromise for lit LCDs. Transmissive displays have no reflector at all. They can only be read with lighting but are very bright. The LCD in the learning package is a transflective LCD.

## 5.5 | The Controller of the LC-Display

Dot-matrix-displays are produced by many manufacturers around the world (and particularly in Taiwan). In addition to displays from large providers like Data-Vision, there are also displays of which the manufacturer cannot be determined at all. Luckily, the function and connection of the displays are always the same. In this learning package, we will deal with displays that use a controller type HD44780 (or compatible), e.g. the KS0066.

The consistent behaviour of all displays is due to a controller chip that has become established as the standard and that is installed by all manufacturers. This is the HD44780 by Hitachi.

## 5.6 | This is How the Display is Controlled by the Display Controller

The following figures show how the display controller (KS0066) is connected to the display. These circuits do not need to be performed by you. They are already present on the LCD modules.

The controllers are partially differently connected to the displays and may also be switched differently depending on manufacturer. Thus, it is possible that a single-line 16-character display is made up of 2 x 8 characters. You need to check the data sheet for this. Larger displays also often use two controller chips that have a Chipselect- (CS) or two Enable lines. The reason for this is that a controller only has a character buffer of 80 characters. By connecting several display controllers, the character buffer will increase by another 80 characters each. The displays also have another connection and are relatively easy to distinguish from the stand-ard-LCD-modules. It can be assumed that a display without lighting has 14 pins and one with lighting has 16 pins.

## 5.7 | The Contrast Setting of the Display

As in other screens, we can set the contrast for LCD-modules as well. This is done, e.g., with a 10-kΩ-potentiometer, that is switched as a variable voltage distributor. Since the LCDs have

a very low scatter of the electrical properties by now, the technology used in the learning package can also be used with a single fixed resistor. For this, we use a 2.2-kΩ-resistor that is placed between the ground and the contrast connection Vee of the LCD to firmly set the contrast.

When using a potentiometer without ballast, the adjustable range that affects the contrast is very low, however. For a better spread of the contrast range, it is recommended to Switch a corresponding ballast between Vcc (+5 V) and one end of the potentiometer.

The voltage at the pin Vee should be adjustable between 0 and 1.5 V. This circuit is suitable for an ambient temperature of 0 to 40 °C. If the adjustable range is not optimal, (some LCDs deviate from this), we need to change the ballast. Practical values are in the range from 10 to 22 kΩ.

If we use the display outside of the normal temperature range (0 to 40 °C), it is recommended to perform the wiring as shown in the circuit diagram above. This circuit adjusts the contrast to the ambient conditions. The temperature will be measured with a temperature sensor NTC (Negative Temperature Coefficient Thermistor) that will shift the contrast voltage via the PNP-transistor. The LCD-modules can no longer be read properly at too-low temperatures below 0 °C. The contrast is temperature-dependent.

## 5.8 | The Character Set

The displays have a character set that is firmly integrated in the display controller. By sequencing the upper and lower 4 bits, the data byte for the corresponding ASCII-character will be formed. Example of the ASCII-character A: 01000001

Let us do an experiment with the LCD-character set. The following program code shows how you can write the characters from the character table onto the display. Special characters like the degree symbol or the Ohm symbol are not possible via a strong output. Since this is the expanded character set of the LCD , we need to do so using the character table.

```
001 lcd.write(B11110100)
```

Here, we will write the binary value in the display controller to output the Omega character. The B at the beginning of the

figure sequence marks that this is a sequence of numbers in binary annotation.

The upper and lower 4 bits for the Omega character are made up as follows:

```
001 upper       =       1111
002 lower       =       0100
```

Try outputting other characters as well, checking the , character table.

Upload

The experiment requires the LCD basic wiring that you set up in the function test.

Example code: CHARACTER SET

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 void setup()
009 {
010     // LED-Backlighting
011     analogWrite(9, 150);
012
013     // LCD-output
014     lcd.begin(16, 2);
015     lcd.setCursor(0, 0);
016     lcd.write(B11110100);
017
018 }
019
020 void loop()
021 {
022     // Nothing to do...
023 }
```

## 5.9 | Pin Assignment of the Common LCDs

Most displays without lighting have a pin assignment as that in the following table. If you use a different LCD than the one included in the learning package at a later time, we recommend

first looking at the associated data sheet to avoid damage to the LCD .

LCD-modules with lighting always require a little care. Some manufacturers do not apply the LED backlighting contacts to pins 15 and 16, but to pins 1 and 2. Again, check the manufacturer's data sheet before connecting the LCD.

> **Info**
> The LCD of the learning package has the LED-connections on pin 15 (+ = anode) and 16 (- = cathode).

If you have no data sheet at hand for the LCDs, e.g. if you have purchased the LCD on an electronics flea market, you need to track the tracks to find the backlighting connections. They are usually a little thicker than the other tracks. When you are sure which connections are responsible for lighting, you can use a multimeter set to diode testing to determine polarity. The passage duration must return a passage voltage between 2 and 4 V. Another option would be to identify the LED-pins and polarity when making the LED light up with a mains unit or a battery of approx. 5 V and a relatively high ballast (approx. 1–4.7 kΩ). The high ballast has a relatively small danger of destruction of the LCD.

# THE ARDUINO™ LIQUITCRYSTAL LIBRARY

As we have already learned in the function test, the Arduino™ LiquidCrystal Library has a number of functions specifically determined for output on the LCD. Now you will learn more about the LCD functions.

## 6.1 | LiquidCrystal

LiquidCrystal specifies with which Arduino™-pins the LCD is connected The LCD can be configured in 4- or 8-bit mode. To use it in 8-bit mode, you need to indicate eight instead of four data

pins (D0 to D7) and connect them to the Arduino™-PCB.

**Arduino™-Syntax**

```
001 LiquidCrystal lcd(rs, enable, d4, d5, d6, d7)
002 LiquidCrystal lcd(rs, rw, enable, d4, d5, d6, d7)
003 LiquidCrystal lcd(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7)
004 LiquidCrystal lcd(rs, rw, enable, d0, d1, d2, d3, d4,
                                                    d5, d6, d7)
```

Our learning package uses the following configuration:

```
001 // RS, E, D4, D5, D6, D7
002 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
```

RS= Arduino™-Pin D11

D4= Arduino™-Pin D2

D1= Arduino™-Pin D5

## 6.2 | .begin()

`.begin()` initialises the LCD with the indicating rows and columns. Our LCD has 2 rows and 16 columns. Parametrisation therefore needs to be as follows:

**Arduino™-Syntax**

```
001 lcd.begin(16, 2)
```

## 6.3 | .clear()

`.clear()` deletes the displayed characters and positions the cursor in the upper left corner.

**Arduino™-Syntax**

```
001 lcd.clear()
```

## 6.4 | .home()

`.home()` positions the cursor in the upper left corner. No characters are deleted.

**Arduino™-Syntax**

```
001 lcd.home()
```

## 6.5 | .setCursor()

`.setCursor()` sets the cursor to the specified position. As so often in informatics, this count starts at zero. The upper left position, i.e. the first character in row 1, is as follows:

**Arduino™-Syntax**

```
001 lcd.setCursor(0, 0)
```

The first parameter is the character position, the second parameter is the row.

## 6.6 | .write()

`.write()` writes a single character onto the LCD. This can be used to output special characters from the character table as well, or to indicate the ASCII-code for the character.

Arduino™-Syntax

```
001 lcd.write(64)
```

The character @ has the decimal digit 64 in ASCII code.

Individual ASCII-characters are marked with an apostrophe. We can also write as follows:

Arduino™-Syntax

```
001 lcd.write('@')
```

## 6.7 | .print()

With `.print()`, we can output entire character sequences, called Strings. It is also possible to output variables this way. For

this, there is a number of formatting parameters (BASE), that are indicated as second parameter.

Arduino™-Syntax

```
001 lcd.print(data, BASE)
002
003 lcd.print("Arduino")  // only a text is output
004
005 int variable1 = 100
006 lcd.print(variable1) // the value of "Variable 1"
                                                    is output
007
008 lcd.print(40 + 2)    // the total of 40 + 2
                                                  is output
009
010 lcd.print(3.1415, 2) // only 3.14 is output
011
012 lcd.print(42, BIN)   // 42 is output in binary
```

### 6.8 | .cursor()

`.cursor()` switches on the cursor. If the cursor has been off, it is visible again now.

Arduino™-Syntax

```
001 lcd.cursor()
```

### 6.9 | .noCursor()

`.noCursor()` switches off  the cursor (invisible).

Arduino™-Syntax

```
001 lcd.noCursor()
```

### 6.10 | .blink()

`.blink()` switches on the cursor and makes it flash.

Arduino™-Syntax

```
001 lcd.blink()
```

### 6.11 | .noBlink()

`.noBlink()` switches off the cursor and ends flashing.

Arduino™-Syntax

```
001 lcd.noBlink()
```

### 6.12 | .noDisplay()

`.noDisplay()` switches off the display. The character and cursor position are saved.

Arduino™-Syntax

```
001 lcd.noDisplay()
```

### 6.13 | .display()

`.display()` switches on the LCD again after a .noDisplay(). The last values are restored.

Arduino™-Syntax

```
001 lcd.display()
```

### 6.14 | .scrollDisplayLeft()

`.scrollDisplayLeft()` scrolls the screen content to the left by one character every time it is called.

Arduino™-Syntax

```
001 lcd.scrollDisplayLeft()
```

### 6.15 | .scrollDisplayRight()

`.scrollDisplayRight()` scrolls the screen content to the right by one character every time it is called.

Arduino™-Syntax

```
001 lcd.scrollDisplayRight()
```

## 6.16 | .autoscroll()

`.autoscroll()` automatically scrolls the display content from the right to the left. When the end of the character string is reached, the scroll direction is automatically switched. It is pushed on 1 x (scrolled) at each call.

Arduino™-Syntax

```
001 lcd.autoscroll()
```

## 6.17 | .noAutoscroll()

`.noAutoscroll()` ends the `.autoscroll()`-function.

Arduino™-Syntax

```
001 lcd.noAutoscroll()
```

## 6.18 | .leftToRight()

`.leftToRight()` specifies the output direction of the characters. They are written from the left to the right.

Arduino™-Syntax

```
001 lcd.leftToRight()
```

## 6.19 | .rightToLeft()

`.rightToLeft()` specifies the output direction of the characters. They are written from the right to the left.

Arduino™-Syntax

```
001 lcd.rightToLeft()
```

### 6.20 | .createChar()

`.createChar()` creates a dedicated character. For this, we need to create an array with eight data fields, by defining our character. `lcd.createChar` gives our character a serial number with the first parameter. The Second parameter hands over the name of the array. Up to eight own characters, which are called with 0 to 7 can be created.

Arduino™-Syntax

```
001 byte myChar[8] = {
002   B00000,
003   B10001,
004   B00000,
005   B00000,
006   B10001,
007   B01110,
008   B00000,
009 }
010
011 void setup()
012 {
013   lcd.createChar(0, myChar)
014   lcd.begin(16, 2)
015   lcd.write(byte(0));
016 }
```

# LCD FUNCTIONS

The following example summarises the LCD functions explained above in a larger example. Look at the program code and change a few of the parameters just described to fully understand the function.

**Upload**
The experiment requires the LCD basic writing that you set up in the function test.

**Example code: FUNCTIONS**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
```

```
007
008 #define Backlight   9
009
010 int i;
011
012 void setup()
013 {
014     analogWrite(Backlight, 200);
015
016     lcd.begin(16, 2);
017     lcd.setCursor(0, 0);
018     lcd.print("ARDUINO LCD”);
019     delay(1000);
020     lcd.clear();
021 }
022
023 void loop()
024 {
025     // Cursor flashing and position change
026     lcd.clear();
027     lcd.setCursor(0, 0);
028     lcd.print("blink/setCursor");
029     delay(1000);
030     lcd.clear();
031
032     lcd.setCursor(0, 0);
033     lcd.blink();
034     delay(1500);
035
036     lcd.setCursor(15, 0);
037     delay(1500);
038
039     lcd.setCursor(0, 1);
040     delay(1500);
041
042     lcd.setCursor(15, 1);
043     delay(1500);
044
045
046     // Cursor on/off
047     lcd.noBlink();
048     lcd.clear();
049     lcd.setCursor(0, 0);
050     lcd.print("cursor on/off");
051     delay(1000);
052     lcd.clear();
053     lcd.home();
054     lcd.cursor();
```

```
055
056     char txt[6] = {"HALLO"};
057      for(i = 0; i < 5; i++)
058     {
059         lcd.print(txt[i]);
060         delay(500);
061     }
062
063     lcd.noCursor();
064     delay(2000);
065
066
067     // Scroll LCD
068     lcd.clear();
069     lcd.noBlink();
070     lcd.setCursor(0, 0);
071     lcd.print("scroll LCD");
072     delay(1000);
073     lcd.setCursor(0, 0);
074
075     for(i = 0; i < 16; i++)
076     {
077         lcd.scrollDisplayLeft();
078         lcd.setCursor(0, 0);
079         lcd.print("FRANZIS ARDUINO IS MEGA GREAT!");
080         delay(350);
081     }
082
083     delay(1500);
084
085     for(i = 0; i < 16; i++)
086     {
087         lcd.scrolldisplayRight();
088         lcd.setCursor(0, 0);
089         lcd.print("FRANZIS ARDUINO IS MEGA GREAT!");
090         delay(350);
091     }
092
093     delay(1500);
094 }
```

What is new is that we indicate the pin for the backlighting with `#define backlight`. This is a preprocessor command that will replace all names occurring in the source code with the designation Backlight by the value 9. This way, you can perform changes to parameters very quickly without having to search the entire source code.

**Advice**
For more on the subject of preprocessors, see:
http://www.mikrocontroller.net/articles/C-Pr%C3%A4prozessor

The following program point offers an option for outputting characters individually as if on an old typewriter:

```
001     char txt[6] = {"HELLO"};
002      for(i = 0; i < 5; i++)
003     {
004         lcd.print(txt[i]);
005         delay(500);
006     }
```

Here, an array with 6 characters is set up and preassigned with the string »HELLO« . We always need to set up the array bigger by 1 because an invisible string termination (\0) is added automatically.

The `For()`-loop outputs every single character from the array with a brief pause. For the entire thing to look more like a typewriter, the cursor is switched on.

# CREATING OWN CHARACTERS

Creating own characters as has already been described just now using `.createChar()` is often needed when using dot-matrix LCDs, since many characters needed in practice are not included in the character table of the LCD. There is the option of creating own characters dot by dot for this and displaying them then. If you need, e.g., a smiley, you can define it via an array and send it to the LCD.

There is the option of filing up to eight own characters in the RAM (memory) of the LCD. The array for our special character must be 8 bytes large and is best written as shown in the example code. The character can, e.g., be designed on a checked drawing pad. You can see that it is made up of 8 rows at 5 values each, which

reflect our 5 x 8 dots in the LCD. Where we set a 1 in the binary code, a white dot will appear later. With `lcd.write(byte(0))`, we write the character onto the LCD.

The example makes the entire thing even clearer. Try to produce a battery symbol or a thermometer.

**Upload**

The experiment requires the LCD basic writing that you set up in the function test.

**Example code: OWN CHARACTERS**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 byte myChar[8] = {
009   B00000,
010   B10001,
011   B00000,
012   B00000,
013   B10001,
014   B01110,
015   B00000,
016 };
017
018 void setup()
019 {
020     // LED-Backlighting
021     analogWrite(9, 150);
022
023     lcd.createChar(0, myChar);
024     lcd.begin(16, 2);
025     lcd.write(byte(0));
026 }
027
028 void loop()
029 {
030
031     // Nothing to do...
032
033 }
```

# DIMMING BACKLIGHT

The following experiment shows how we can automatically set the LCD-lighting brighter or darker. By changing the PWM-value at pin D9, the brightness of the backlighting is adjusted gradually. If the PWM-value is chosen higher, the LED will be brighter. A lower value will dim the lighting. By changing the PWM-value, we will change the plus-pause ratio between activation and deactivation duration of the 5-V-signal at D9. The following figure illustrates this.

Transfer the program and observe the LED. This already looks almost as if the display had been brought to life.

**Upload**

The experiment requires the LCD basic writing that you set up in the function test.

**Example code: LCD_LED**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define Backlight   9
009
010 byte i = 0;
011 byte flag = 0;
012 unsigned long previousMillis = 0;
013 const long interval = 10;
014
015 void setup()
016 {
017     analogWrite(Backlight, 0);
018
019     lcd.begin(16, 2);
020     lcd.setCursor(0, 0);
021     lcd.print("**ARDUINO LCD**”);
022 }
023
024 void loop()
025 {
026     unsigned long currentMillis = millis();
027     if(currentMillis – previousMillis >= interval)
028     {
```

```
029         if(flag==0)i++;
030         if(flag==1)i--;
031
032         if(i==255)flag=1;
033         else if(i==0)flag=0;
034
035         analogWrite(Backlight, i);
036
037         previousMillis = currentMillis;
038     }
039 }
```

The experiment /down counter with a limit can be put into practice. It is important that the variable named `flag` receives a defined starting value of 0. When starting the program, the variable `i` is counted up to 255 . If we wanted to start at full brightness, we would have to initialise the variable `flag` with 1 and the variable `i` with 255.

When the counter value `i` of 255 or 0 is reached, the variable `flag` will always be set from 0 to 1 or from 1 to 0 and the counter direction will change (i++ increases the counter status, incremented by 1, i-- reduces the counter status, decremented by 1). In the `Loop()`-function, we do not use a break this time, but determine the time via the function `millis()`. Only when a specified difference that we have declared in the variable `previousMillis` has expired will the brightness be changed by one level. This way, the program will continue to run at full speed outside of the If()-query. Only if the PWM-value is changed will the throughput time change a little, since some functions that require a certain processing time will be called . Here, you can also try to determine the different throughput times with the `Millis()`-function that you have already learned about.

# DOT-MATRIX-LCD CLOCK

In many applications, a clock is needed for program control - either a simple timer, a control to comply with a precise schedule, or an operating hours counter. The applications that need a clock are diverse.

The experiment shows how to program a very simple clock yourself. The program runs in the `Loop()`-function, is finite and counts up at a cycle of 10-ms. If the counter reading cnt = 100, the time

is output. This is done every second. Our user LED L will flash every second as well. We monitor the function of the program to ensure that it is still running and to see if an error occurred in programming. However, note that the clock does not have the precision of an actual quartz clock since the cycle and the deviation of the micro controller quartz at 16 MHz is much higher than in a clock quartz in the Kilohertz range (clock quartz = 32.768 kHz). Deviations of more than one minute per day are not rare. The accuracy also strongly depends on ambience temperature. If it fluctuates strongly over time, the clock will also have a higher time error. We can correct the time deviation with `delay ()`. Alternatively, we can also use `delayMicroseconds()` to correct the deviation even better. For this, configure a digital pin as the output and toggle it at every program run , change the condition once at every run. This signal can be trimmed precisely to a throughput time of 10 ms with an oscilloscope. You can determine the deviation across an extended period by comparing the time to a different, precise clock, e.g. a DCF-clock for a while (1 to 2 days), calculating the difference and then correcting the deviation with `delayMicroseconds()`.

These variables are then used to set the clock:

```
001 Second = 12
002 Minute = 0
003 Hour = 0
```

**Advice**
For more on the subject of clock quartz,
see: http://de.wikipedia.org/wiki/Uhrenquarz

**Upload**
The experiment requires the LCD basic writing that you set up in the function test.

**Example code: RTC**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
```

```
008 #define Backlight   9
009
010 int cnt, Second, Minute, Hour=0;
011 int LED=13;
012
013 void setup()
014 {
015       pinMode(LED, OUTPUT);
016       analogWrite(Backlight, 200);
017       lcd.begin(16, 2);
018
019       // Time specification
020       Second = 12;
021       Minute = 0;
022       Hour = 0;
023 }
024
025 void loop()
026 {
027
028   cnt++;
029   if(cnt == 50)digitalWrite(LED, LOW);
030
031   if(cnt == 100)
032   {
033       digitalWrite(LED, HIGH);
034
035       lcd.setCursor(3, 0);
036
037     if(Hour < 10) lcd.print("0");
038     lcd.print(Hour);
039     lcd.print(":");
040
041     if(Minute < 10) lcd.print("0");
042     lcd.print(Minute);
043     lcd.print(":");
044
045     if(Second < 10) lcd.print("0");
046     lcd.print(Second);
047
048     Second++;
049     if(Second == 60)
050     {
051         Second = 0;
052         Minute++;
053         if(Minute == 60)
054         {
055             Minute = 0;
```

```
056             Hour++;
057             if(Hour == 24)
058             {
059                 Hour = 0;
060             }
061         }
062     }
063     cnt = 0;
064   }
065
066   delay(10);
067 }
```

If we were to output the counter readings one to one on the display, the counter readings below 10 would look strange because the leading zero would not be displayed. For the clock to have the familiar »00:00:00«- format, we need to check if the value is less than 10 before the output.

```
001 if(Second < 10) lcd.print("0")
```

If the value is less than 10, a simple output would only display, e.g. »12:1:8«. However, we check if the value is less and add a »0« manually if necessary to fill the tens.

# CAPACITY METER

Building your own meters with the simplest of media is always interesting and exciting. Arduino™ permits programming a capacity meter for small capacitors in the range of 1 nF to approx. 100 µF for our hobby lab at very low costs and effort. This is how our capacity meter with auto range function works:

At commencement of the measurement, the variable `C_time` is initialised with zero. Port D12 is configured as an output and then immediately switched to LOW (GND), to discharge the connected capacitor (test piece) before the actual measurement.

After a brief end charging pause of 1 second, port D12 will be configured as input and the internal pull-up-resistance will be activated. The pull-up-resistance will now charge the capacitor to be tested until the port D12 recognises HIGH. The threshold from when onwards the digital port recognises HIGH is at approx. 3.5 V at an operating voltage of 5 V. This level therefore

depends on the operating voltage and is indicated in the data sheet for the micro controller at Vcc x 0.7.

```
001 HIGH = 5V x 0,7
```

Before a HIGH level is recognised in the digital port, some time will pass. We measure it within the Do-while-loop using the variable `C_time`. `C_time` is approximately proportional to the capacity of the capacitor, i.e. when `C_time` is very large, the capacity to be measured is very large as well.

To get the proper measured value, we need to convert the variable (`C_time` • `Factor`). The value (factor) must be determined experimentally with »a few calibration capacitors«, since recognition of a HIGH level will deviate slightly from controller to controller in spite of the information Vcc x 0.7 in the data sheet, and the oscillator frequency of 16 MHz is not 100 % the same in every board (quartz tolerances). Finally, the measured value will be divided into Nanofarad (nF) and Mikrofarad (µF) with a simple If()-query and output on the LCD before a new measurement starts.

## 11.1 | Setup of the capacitor meter

**The capacitor is plugged in at pin D12 and GND. Observe that you additionally discharge the capacitor when plugging it into the Arduino™-PCB by touching the two wires of the capacitor together. Even if the program discharges the capacitor before starting the measurement, it is possible that the capacitor you chose has previously been operated at a higher voltage than 5 V. This may damage your Arduino™-board.**

## 11.2 | Calibrating your Capacitor Meter

Get a few new capacitors – always use capacitors of which you know precisely what capacity they have. You may have them measured by an electronics specialist in the lab. New capacitors usually have the printed-on values +/- 20 %, depending on type.

**In this measurement, enter factor 1 (`c_time = 1.0`). Plug one of these capacitors into the meter and read the value after transmission at the terminal. Then divide the capacity value of the plugged-in capacitor by the measured result on the LCD , and enter the result as a factor, e.g. 1 µF / 19.55 µF = 0.0511.**

Example code: CAPA

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define Backlight   9
009
010 int messPort=12;
011 float c_time=0.0;
012 float kapazitaet=0.0;
013
014 void setup()
015 {
016     analogWrite(Backlight,200);
017
018     lcd.begin(16, 2);
019     lcd.setCursor(0,0);
020     lcd.print("C-METER");
021 }
022
023 void loop()
024 {
025       // Discharging
026       pinMode(messPort,OUTPUT);
027       digitalWrite(messPort,LOW);
028       c_time=0.0;
029       delay(1000);
030
031       // Charging
032       pinMode(messPort,INPUT);
033       digitalWrite(messPort,HIGH);
034
035       // Measuring
036       do
037       {
038         c_time++;
039       }while(!digitalRead(messPort));
040
041       // Converting
042       kapazitaet=(c_time*0.06162)*10.0;
043
044       // Range
045       if(kapazitaet<999)
```

```
046       {
047           lcd.setCursor(0,1);
048           lcd.print(kapazitaet);
049           lcd.print("nF    ");
050       }
051       else
052       {
053           lcd.setCursor(0,1);
054           lcd.print(kapazitaet/1000);
055           lcd.print("uF    ");
056       }
057
058       delay(1000);
059 }
```

# RANDOM NUMBERS – THE LOTTERY RESULTS GENERATOR

When writing measuring, control, regulating or playing programs, it is often of benefit to generate random numbers, e.g. when lights are to go on and off at different times in a house to program a presence simulator. For this purpose, the Arduino™-random()-function can be used. This permits a simple lottery number generator that will draw 6 out of 49 for you. You will no longer have to think about which numbers you are supposed to take when you complete your lottery slip.

For the setup of the lottery number generator, you need a button and an aerial. The button is debounced in the software. The button and switch tend not to close the contact 100 % at once but to trigger several times after being pushed. This is comparable to tossing a ball to the floor. It will bounce a few times before finally resting on the ground. This happens much faster in a button, but the Arduino™-micro controller is so fast that it will still record these millisecond »hops«. To avoid this, the button is debounced by being queried twice in sequence with a break of 50 ms, which is enough for a debouncing routine in practice . Only if the second evaluation still recognises a LOW at input D7 will the instruction between the brackets be executed.

At commencement of the program, we switch the port D7 to INPUT

and activate the internal pull-up-resistance, by ing `digitalWrite()` to write a 1 for HIGH on the input. Now a voltage of approx. 5 V is pending at the input in the resting condition. Now you can pull the input against GND (ground) with the button. This pull-up-resistance is integrated in the microcontroller and has a value of approx. 20 to 50 kΩ. Regarding function, it is the same as if applying an external resistor from input D7 to +5 V. In the resting condition, the program will therefore always recognise a HIGH at input D7 and a LOW when the button is pushed. Therefore, the button query is applied with a question mark in the program. This is called the NOT-operator in C-programming. Since it is known that an If()-query checks for TRUE , everything else will be interpreted as FALSE . If the If()-query has been performed without this operator, the condition would always be TRUE. It would then already be performed before the button had even been pushed. The NOT-operator inverts the status of the button. 1 turns into 0 and 0 turns into 1 and the If()-query is now only TRUE if the button has actually been pushed.

Example code: LOTTERY

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define Backlight   9
009 #define button      7
010
011 int i, zahl=0;
012 int Anz = 6;
013
014 void setup()
015 {
016     analogWrite(Backlight, 200);
017
018     pinMode(button, INPUT);
019     digitalWrite(button, HIGH);
020
021     lcd.begin(16, 2);
022     lcd.setCursor(0, 0);
023     lcd.print("LOTTO 6 of 49");
024     delay(1000);
025     lcd.clear();
026
027     randomSeed(analogRead(0)*5);
```

```
028
029 }
030
031 void loop()
032 {
033     lcd.clear();
034     lcd.setCursor(0, 0);
035     lcd.print("PUSH BUTTON");
036
037     while(digitalRead(button));
038     {
039         if(!digitalRead(button))
040         {
041             delay(50);
042             if(!digitalRead(button));
043             {
044                 lcd.clear();
045                 lcd.setCursor(0, 0);
046                 lcd.print("YOUR LOTTERY NUMBERS");
047
048                 lcd.setCursor(0, 1);
049                 for(i=0;i<Anz;i++)
050                 {
051                    zahl=random(49);
052                    zahl++;
053                    lcd.print(zahl);
054                    lcd.print(" ");
055                    delay(500);
056                 }
057
058                 delay(5000);
059             }
060         }
061     }
062 }
```

When starting the program, `randomSeed()` generates a value as starting point for the `Random()`-function. When changing the value of `randomSeed()`, different random numbers will be generated in each case. If the value of `randomSeed()` always were the same when the program starts, the same random number series would be generated every time, which would not be helpful for playing lottery.

Our aerial is used here. To produce different values with the function `randomSeed()`, we use an ADC-input connected to a jumper and remaining open on the other side. This acts like an aerial and produces a higher noise at the analogue input and thus a different value for `randomSeed()` in each case. This works best if

you put your hand next to the aerial or if the aerial is placed near electrical devices. The result with different number series can only be seen when you push the reset button on the Arduino™-PCB and then have the numbers output. You can enter a fixed number instead of analogRead() once to see that the same number sequences will appear every time after the reset.

# BAR CHART DISPLAY

Bar chart displays are often used in measuring technology. They are also called bar displays. They display a visual/trend measured value. When setting electronic circuits, a bar chart display makes things much easier because the trend towards max. or min. can be read more easily than in digital numeric value displays. We know this display and progress bar in computer programs as well, e.g. when installing a program. Here, the bar shows how far the installation has already progressed. Generally, the bar chart display shown in the experiment is an analogue display on a digital basis. Electromechanical bar chart displays were already used in early electronics. Our bar chart display, however, uses a modern LCD and a microcontroller.

In the simplest case, we would be able to display a full character for each display step (5 x 8 dots). We can only implement a very general display of 0 to 16 then. It would be nice if you could use the individual five columns of every single character. Since we can generate eight own characters, it is simple to program a bar chart display with 5 x 16 = 80 characters/conditions.

**Upload**

The experiment requires the LCD basic writing that you set up in the function test.

**Example code: BARGRAPH**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define LCD_LENGHT 16.0
009
010 int value;
011 byte flag = 0;
012
013 byte MyChar0[8] = {
```

```
B00000,
B00000,
B00000,
B00000,
B00000,
B00000,
B00000,
B00000,
B00000,
};

byte MyChar1[8] = {
B10000,
B10000,
B10000,
B10000,
B10000,
B10000,
B10000,
B10000,
B10000,
};

byte MyChar2[8] = {
B11000,
B11000,
B11000,
B11000,
B11000,
B11000,
B11000,
B11000,
B11000,
};

byte MyChar3[8] = {
B11100,
B11100,
B11100,
B11100,
B11100,
B11100,
B11100,
B11100,
B11100,
};

byte MyChar4[8] = {
B11110,
B11110,
B11110,
B11110,
```

```
B11110,
B11110,
B11110,
B11110,
};

byte MyChar5[8] = {
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
};

void draw_bargraph(byte percent)
{
  byte i, c1, c2;

  lcd.setCursor(0, 0);
  lcd.print(percent);
  lcd.print("%  ");

  lcd.setCursor(0, 1);

  percent = map(percent, 0, 100, 0, 80);

  c1 = percent / 5;
  c2 = percent % 5;

  for(i = 0; i < c1; ++i)
  {
    lcd.write(byte(5));
    lcd.write(c2);
  }

  for(i = 0; i < 16 – (c1 + (c2 ? 1 : 0)); ++i)
  {
    lcd.write(byte(0));
  }
}

void setup()
{
    analogWrite(9,200);

```

```
110     lcd.createChar(0, MyChar0);
111     lcd.createChar(1, MyChar1);
112     lcd.createChar(2, MyChar2);
113     lcd.createChar(3, MyChar3);
114     lcd.createChar(4, MyChar4);
115     lcd.createChar(5, MyChar5);
116
117     lcd.begin(16, 2);
118 }
119
120 void loop()
121 {
122     double percent;
123
124     if(flag == 0)value++;
125     if(flag == 1)value--;
126     if(value > 1024)flag=1;
127     else if(value == 0)flag=0;
128
129     percent = value / 1024.0 * 100.0;
130     draw_bargraph(percent);
131     delay(10);
132 }
```

The individual segments of the bar chart display are specified with the arrays `MyChar0` to `MyChar5`. The program code already shows how the segments fill with ones bottom-up array by array. In the `Setup()`-function, the characters are created with `lcd.createChar()`.

In the `Loop()`-function, an Up/Down counter that we know from the preceding experiments is used again. However, this time, it counts from 0 to 1,024. This way the counter could easily be replaced by an Arduino™-analogue input that covers a value range of 0 to 1,023. The counter value is converted to percent and handed over to the function `draw_bargraph()`.

The `draw_bargraph()`-function is exciting. Here, the bar chart display is put together and displayed. Every time the function is called, the cursor will be set to position (0, 0). This is the starting position to re-draw the display. In the first row, we output the percentage value of the variable `percent` and write a percentage sign behind it. The percentage sign will be followed by two spaces to clear the display from tens or hundreds offset. The digital value output in percent is thus complete.

Then we use `setCursor(0, 1)` to place the cursor in the lower, i.e. the second row of the LCD. To divide the percentage value, which goes from 0 to 100 % into the 80 individual areas (pixels), we

use the `Map()`-function. This scales the input value `percent`, from 0 to 100 to the output value from 0 to 80. The variable `percent` then holds a value between 0 and 80, depending on the input value `percent`.

Now we determine the value of `percent` by dividing the number of boxes to be filled entirely by 5, and write the result into the variable `c1`. The modulo operation % determines the rest or the partial filling. We will write this value into the variable `c2`. Now we know how many boxes are to be filled completely and how many are filled only partially.

In the following `For()`-loop, we will count up until all filled boxes are reached, and write a full box onto the LCD in every loop passage. Since every subsequent output at the LCD automatically moves the characters by one, we will have a bar with full boxes at the end of the loop. This point of the program also shows that the `For()`-loop uses no braces. In the `For()`-loop, the compiler only takes the subsequent row into the loop. In this case, this would be the call of `lcd.write(byte(5))`. After completing this first box loop, the box with the partial filling is written onto the LCD. This leads to a bar that is built pixel by pixel on the LCD.

Example
The percentage value 43 is to be displayed. We divide the number 43 by 5, which makes 8.6. Since the variable c1 is declared as byte, only an 8 will be saved in it. Modulo 5 from 43 is 3, which means three partial strokes.

If the value reduces again, we need to delete the superfluous characters from the LCD. A new operation, which is called conditional expression or ternary selection operator, is added .

Generally, the entire thing can be viewed like an If-Else-instruction, but it is only an abbreviated C annotation.

The syntax for the conditional instruction would be: Condition ? Expression1 : Expression2

This seems familiar to you, doesn't it? It generally is no different from:

```
001 if(condition)
002 {
003     // Expression1
004 }
005 else
006 {
007     // Expression2
008 }
009
```

```
010 for(i = 0; i < 16 – (c1 + (c2 ? 1 : 0)); ++i)
011 lcd.write(byte(0))
```

The loop deletes the superfluous characters from the LCD by determining how long the area that is not used is and overwriting it with spaces.

A little know-how is needed for the bar chart display, but once it has been understood, it can be used easily in many applications.

# LIGHT METER – THE PHOTOMETER

A photometer is a meter to determine the light density or light strength. It is used, e.g. by photographers as lighting meter or in astronomy to determine the brightness of stars.

In chemistry, it is used to determine concentrations. In the last experiment, we programmed a bar chart display; this time, instead of the up/down counter, we will hand over a true physical value and program a simple photometer.

The circuit diagram shows how the photo transistor is connected to the analogue input (ADC) of the Arduino™-board. The photo transistor is not easy to tell from a regular LED. It has a clear, transparent housing, but that is true for some LEDs as well. If you are not sure if you have an LED or a photo transistor, you can check the behaviour with a multimeter set to measuring resistances. For this, connect the collector (C – flattened side of the housing) to the pulse line and the emitter (E) to the minus line of the meter. If you darken a photo transistor, the resistance value will change enormously. In an LED, the effect will be tiny. Multimeters with an auto range function are ideal here because the measured value can be between a few Kiloohm and several Megaohm.

Example code: PHOTOMETER

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
```

```
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define LCD_LENGHT 16.0
009
010 const int numReadings = 10;
011 unsigned int readings[numReadings];
012 int index = 0;
013 unsigned int total = 0;
014 int average = 0;
015
016 byte MyChar0[8] = {
017 B00000,
018 B00000,
019 B00000,
020 B00000,
021 B00000,
022 B00000,
023 B00000,
024 B00000,
025 };
026
027 byte MyChar1[8] = {
028 B10000,
029 B10000,
030 B10000,
031 B10000,
032 B10000,
033 B10000,
034 B10000,
035 B10000,
036 };
037
038 byte MyChar2[8] = {
039 B11000,
040 B11000,
041 B11000,
042 B11000,
043 B11000,
044 B11000,
045 B11000,
046 B11000,
047 };
048
049 byte MyChar3[8] = {
050 B11100,
051 B11100,
052 B11100,
053 B11100,
```

```
B11100,
B11100,
B11100,
B11100,
};

byte MyChar4[8] = {
B11110,
B11110,
B11110,
B11110,
B11110,
B11110,
B11110,
B11110,
};

byte MyChar5[8] = {
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
B11111,
};

int adc_AVG(byte channel)
{
    total= total – readings[index];
    readings[index] = analogRead(channel);
    total= total + readings[index];
    index = index + 1;

    if (index >= numReadings)index = 0;

    average = total / numReadings;
    return average / 1024.0 * 100.0;
}

void draw_bargraph(byte percent)
{
    byte i, c1, c2;

    lcd.setCursor(0, 0);
    lcd.print("Brightness: ");
    lcd.print(percent);
```

```
102     lcd.print("%  ");
103
104     lcd.setCursor(0, 1);
105
106     percent = map(percent, 0, 100, 0, 80);
107
108     c1 = percent / 5;
109     c2 = percent % 5;
110
111     for(i = 0; i < c1; ++i)
112     lcd.write(byte(5));
113
114     lcd.write(c2);
115
116     for(i = 0; i < 16 – (c1 + (c2 ? 1 : 0)); ++i)
117     lcd.write(byte(0));
118 }
119
120 void setup()
121 {
122     analogWrite(9,200);
123
124     lcd.createChar(0, MyChar0);
125     lcd.createChar(1, MyChar1);
126     lcd.createChar(2, MyChar2);
127     lcd.createChar(3, MyChar3);
128     lcd.createChar(4, MyChar4);
129     lcd.createChar(5, MyChar5);
130
131     lcd.begin(16, 2);
132 }
133
134 void loop()
135 {
136     int raw_adc = adc_AVG(0);
137     draw_bargraph(100 – raw_adc);
138     delay(20);
139 }
```

When you have set up the circuit and transferred the example code to the Arduino™-board, a value near 100 % will be displayed on the LCD in bright environments, and the bar of the bar chart will fill the lower row of the LCD almost completely.

If you darken the photo transistor now, the value will reduce to near zero. Look at the circuit more precisely. The photo transistor is connected to the 10-kΩ-resistance at the collector, which is connected to +5 V, and to GND (ground) at the emitter. The analogue input  0 of the Arduino™-PCB is connected

to the node point of the collector and the resistor. If the photo transistor is now exposed to light, it will become conductive, and the voltage drop between collector and emitter will reduce. We are measuring a very low voltage. When the photo transistor is darkened, barely any current will flow, the photo transistor will lock and the collector-emitter voltage will increase. Now we measure almost the entire 5 V.

Between the two extremes, the photo transistor will be very dynamic and react even to the smallest light fluctuations. Since the display would work precisely inverted this way - very bright would be a low value and dark a very high one - we need to adjust the measured value. For this, we subtract the measured value from 100 %, to get the desired result. We invert our analogue measured value.

To keep the bar chart display from »twitching“ too much at smallest light changes, the bar charge display function had an average formation added. It smoothens out the analogue measured values and calculates a sliding average, called AVG for Average, from it .

For this, the current measured value is added to an array at each run and, depending on how high the counter reading in this function is at the moment, divided by it. This will continually return the current average. The number of measuring series for average formation is specified in the variable `numReadings-`. The higher the number of the values to be averaged, the more precise, but also the more idle the display. You can experiment with the values here. Values between 8 and 64 have proven to be sensible. At the end of the AVG-function, the input value (0 to 1,023) is then calculated for a percentage for the bar chart function.

This kind of photometer can also be reprogrammed to automatically activate and deactivate lighting or, as the next experiment shows, be turned into an alarm system.

# ALARM SYSTEM

The photometer can also be used as an alarm system that will react to the smallest light changes. At the beginning of the alarm system program, the current light intensity at the analogue input A0 that is used as the reference point for the measurement will be determined . If the voltage value in the continuous measurement increases or reduces due to a light change (e.g. a person walking past), and if this exceeds or undercuts the specified threshold, the alarm will trigger.

Since brightness in a room will change over the course of the day, a new reference value (current voltage of the photometer) will be determined automatically every 10 seconds to be used as

the new reference point for continuous measurements.

We therefore compare a fixed light value that is measured anew every 10 seconds to the light value of the continuous measurement. The defined threshold will only cause the alarm to be tripped if the current light value +/- exceeds or undercuts the threshold.

```
001 cnt++;
002 if(cnt > 2000)
003 {
004     cnt = 0;
005     value = analogRead(PHOTOTRANSITOR)
006 }
```

If the variable `cnt` is above 2,000, a new value will be read from the analogue input A0. An inserted pause that influences the program run speed will only increment the variable `cnt` by 1 every 5 ms. This leads to a value of 5 ms x 2,000 = 10,000 ms = 10 seconds.

```
001 Threshold = 25
002 if(analogRead(PHOTOTRANSITOR) > (value + Threshold) ||
            analogRead(PHOTOTRANSITOR) < (value – Threshold))
```

Here, the value of the variable named `Threshold` is set. It is responsible for trigger sensitivity and should not be set too high or too low, since the alarm system will either barely react at all or trigger too many false alarms otherwise. The evaluation of the continuous measured value that is recorded every 5 ms will take place in this program row.

To test this, run your hand over the phototransistor at a distance of approx. 50 cm in a normally lit room. the alarm will trip. This can even be done very quickly. The detector will record you at once if darkening by your hand exceeds 5 ms.

The alarm system could also be placed in a refrigerator and add a counter variable to the alarm message, to record the opening cycles of the refrigerator. In the refrigerator experiment, you will find that the LCD will change its contrast and will also turn very idle. Just try it out.

**Upload**

We use the same setup as for the photometer light meter!

**Example code: ALARM**

```
001  // Integrating LCD-Library
```

```
#include <LiquidCrystal.h>

// Specifying LCD pins
// RS, E, D4, D5, D6, D7
LiquidCrystal lcd(11, 10, 2, 3, 4, 5);

#define Backlight   9

int PHOTOTRANSITOR = 0;
int cnt = 0;
int value, Threshold;

void setup()
{
    analogWrite(Backlight, 200);
    lcd.begin(16, 2);
    lcd.setCursor(1, 0);
    lcd.print("ALARM SYSTEM!!!");
    value = analogRead(PHOTOTRANSITOR);
}

void loop()
{

  Threshold = 25;

  cnt++;
  if(cnt > 2000)
  {
    cnt = 0;
    value=analogRead(PHOTOTRANSITOR);
  }


  if(analogRead(PHOTOTRANSITOR) > (value + Threshold) ||
        analogRead(PHOTOTRANSITOR) < (value – Threshold))
  {
      lcd.setCursor(1, 1);
      lcd.print("<<< ALARM >>>”);
      delay(2000);
      value=analogRead(PHOTOTRANSITOR);
  }
  else
  {
      lcd.setCursor(0, 1);
      lcd.print("               ");
  }

```

```
049   delay(5);
050
051 }
```

# DIGITAL VOLTMETER WITH BAR CHART DISPLAY AND USB INTERFACE

With what you have learned so far, you can now program a digital Voltmeter with analogue bar chart display. The bar chart display will be valuable for setting work. You can see much more precisely where, e.g., the maximum or minimum is, on an analogue display than on a digital display with pure numeric output. As a special feature, we supplement the program with a serial output that will send your measured data to the PC via the USB interface. Here, we use the USB- interface already present on the Arduino™-UNO-board that we are already using for programming.

Resistors R1 and R2 are not needed in this experiment and are not enclosed with the learning package! They will, however, be further explained in this chapter. If you need them, you can purchase them in a specialist electronics store later.

You can use the circuit to measure very precise voltages between 0 and 5 V using the analogue input A0 without the two resistors R1/R2. However, ensure that you do not connect any higher voltages to the connection, since the Arduino™-board would be damaged by this. You may already measure the voltage of one or two Mignon cells (AA) or Micro cells (AAA) extremely precisely with the circuit. The example is very similar to that of the photometer, but with a few different details. This time, we also use the serial interface (UART = Universal Asynchronous Receiver Transmitter) of the Arduino™-micro controller. The measured data are sent through the serial interface of the micro controller (UART) to the UART-to-USB-converter on the Arduino™-PCB , which passes it on to the PC. The serial connections D0/RX and D1/TX are already firmly connected to the converter and no further

wiring work is required for this. On the PC-side, a virtual Comport is produced when installing the Arduino™-PCB. This is already used for programming. Now, we can also simply use it to transfer data to the PC. For this, we only need to initialise the UART interface in the program. This is configured with `Serial.begin()`. The parameter `19200` between the brackets represents the transfer speed. Initialisation only needs to be executed once at program start in the `Setup()`-function.

```
001 Serial.begin(19200)
```

Baud is the unit for the symbol rate in message and telecommunications technology. 19200 Baud means , that 19,200 symbols per second will be transferred. The symbol rate can contain different numbers of bits depending on coding and must be set equally on the transmitter and receiver sides to permit transmission.

The following lines are now used to send the measured result of the ADC (0 to 1013) to the PC directly without prior conversion. Conversion to Volt takes place in the PC-program, since we only need to send two individual bytes to the PC this way, which are much easier to evaluate than a string (ASCII-character string).

Now the buffer of the UART-interface will be emptied with `flush`.

```
001 Serial.flush()
```

Now we will take apart the analogue measured value, which ranges from 0 to 1023, into a high and a low byte. We will get the high byte by dividing the measured value by 256.

```
001 highbyte = adc_raw / 256
```

We will get the low byte with the modulo operation 256.

```
001 lowbyte = adc_raw % 256
```

Then we will send first the high byte and then the low byte to the PC.

```
001 Serial.write(highbyte)
002 Serial.write(lowbyte)
```

To check that the values have been transferred correctly, we can see a checksum at the end of the transfer that is made up of a fixed number and the XOR-formation of this, as well as the two bytes.

```
001 crc = 170^highbyte^lowbyte
002 Serial.write(crc)
```

**Info**
The program also works without PC-program and can be used as a stand-alone volt meter.

Example code: VOLTMETER

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define LCD_LENGHT  16.0
009 #define ADC_CHANNEL 0
010
011 const int numReadings = 20;
012 unsigned int readings[numReadings];
013 int index = 0;
014 unsigned int total = 0;
015
016 byte MyChar0[8] = {
017 B00000,
018 B00000,
019 B00000,
020 B00000,
021 B00000,
022 B00000,
023 B00000,
024 B00000,
025 };
026
027 byte MyChar1[8] = {
028 B10000,
029 B10000,
030 B10000,
```

```
031 B10000,
032 B10000,
033 B10000,
034 B10000,
035 B10000,
036 };
037
038 byte MyChar2[8] = {
039 B11000,
040 B11000,
041 B11000,
042 B11000,
043 B11000,
044 B11000,
045 B11000,
046 B11000,
047 };
048
049 byte MyChar3[8] = {
050 B11100,
051 B11100,
052 B11100,
053 B11100,
054 B11100,
055 B11100,
056 B11100,
057 B11100,
058 };
059
060 byte MyChar4[8] = {
061 B11110,
062 B11110,
063 B11110,
064 B11110,
065 B11110,
066 B11110,
067 B11110,
068 B11110,
069 };
070
071 byte MyChar5[8] = {
072 B11111,
073 B11111,
074 B11111,
075 B11111,
076 B11111,
077 B11111,
078 B11111,
```

```
079 B11111,
080 };
081
082 int adc_AVG(byte channel)
083 {
084     total= total – readings[index];
085     readings[index] = analogRead(channel);
086     total= total + readings[index];
087     index = index + 1;
088     if (index >= numReadings)index = 0;
089     return total / numReadings;
090 }
091
092 void draw_bargraph(byte percent)
093 {
094   byte i, c1, c2;
095
096   lcd.setCursor(0, 1);
097
098   percent = map(percent, 0, 100, 0, 80);
099
100   c1 = percent / 5;
101   c2 = percent % 5;
102
103   for(i = 0; i < c1; ++i)
104   lcd.write(byte(5));
105
106   lcd.write(c2);
107
108   for(i = 0; i < 16 – (c1 + (c2 ? 1 : 0)); ++i)
109   lcd.write(byte(0));
110 }
111
112 void setup()
113 {
114     analogWrite(9, 200);
115
116     lcd.createChar(0, MyChar0);
117     lcd.createChar(1, MyChar1);
118     lcd.createChar(2, MyChar2);
119     lcd.createChar(3, MyChar3);
120     lcd.createChar(4, MyChar4);
121     lcd.createChar(5, MyChar5);
122     lcd.begin(16, 2);
123
124     Serial.begin(19200);
125 }
126
```

```
127 void loop()
128 {
129     double percent;
130     float voltage;
131     byte highbyte, lowbyte, crc;
132     int adc_raw = adc_AVG(ADC_CHANNEL);
133
134     voltage = (5.0 / 1024.0) * adc_raw;
135
136     lcd.setCursor(0, 0);
137     lcd.print(voltage, 2);
138     lcd.print(" V   ");
139
140     percent = voltage / 5.0 * 100.0;
141     draw_bargraph(percent);
142
143     Serial.flush();
144     highbyte=adc_raw/256;
145     lowbyte=adc_raw%256;
146     Serial.write(highbyte);
147     Serial.write(lowbyte);
148     crc=170^highbyte^lowbyte;
149     Serial.write(crc);
150
151     delay(20);
152 }
```

To start the PC-program, you only need to execute the EXE file in the folder ...\VOLTMETER\vb.net\bin\Release. Then choose the Comport, which is identical to the one you have already set in the Arduino™-IDE for the program transfer. If you click Connect now, the voltage in the PC-program will be displayed.

The VB.NET-program is enclosed as source code and can be used as a basis for your own experiments. For this, you need to download the free Visual-Basic-Express-Version by Microsoft. You can find it athttps://www.visualstudio.com/downloads/download-visual-studio-vs. When you have opened the program with Visual Basic, you will see the source code and the designer before you. The designer displays the visual control elements such as buttons, texts fields, etc. Now have a look at the source code of the Voltmeter program by switching to source code view.

```
001 Imports System.IO.Ports.SerialPort
002 Imports System.Text.Encoding
```

First, we will import the functions needed for the serial

interface and coding. Without importing the Encoding Library, we will, e.g., be unable to evaluate any values exceeding 128, since we cannot switch the interface to UTF-8 format then. Our measured result would be wrong then!

```
001 Dim input_data(10) As Byte
```

`input_data(10)` is used to set up an array that can take up up to 10 bytes. The received bytes from the serial interface will be filed in this later.

```
001 Private Sub Form1_Load(ByVal sender As System.Object,
            ByVal e As System.EventArgs) Handles MyBase.Load
```

In the function `Form1_load()`, present Comports will be found now. They are listed in the combo box. The `Form1_Load()`-function will automatically be called first when the program starts – similar to the `Setup()`-function of our Arduino™-program.

```
001 Private Sub Button_Connect_Click(ByVal sender As System.
                Object, ByVal e As System.EventArgs) Handles
                Button_Connect.Click
```

In the function `Button_Connect_Click()`, you will configure and open the serial interface at the same time. This function is called when clicking the Connect button.

```
001 SerialPort1.PortName = ComboBox_Comport.Text
002 SerialPort1.BaudRate = 19200
003 SerialPort1.Encoding = System.Text.Encoding.UTF8
004 SerialPort1.Open()
```

Here, we specify the Comport, the Baud rate and the Encoding, and open the interface with `SerialPort1.Open()`, which will then be ready to transfer and receive.

```
001 Private Sub Button_Disconnect_Click(ByVal sender As
        System.Object, ByVal e As System.EventArgs) Handles
        Button_Disconnect.Click
```

The function `Button_Disconnect_Click()` closes the interface again. It

is then available for other programs again, such as the Arduino™-IDE. If you want to transfer any program changes or other programs to your Arduino™-board, you need to terminate the program first or click Disconnect to release the interface again.

```
001 Private Sub SerialPort1_DataReceived(sender As Object
       , e As System.IO.Ports.SerialDataReceivedEventArgs)
       Handles SerialPort1.DataReceived
002
003         Dim cnt As Byte
004         Dim in_bytes As Byte
005         Dim HighByte As Byte
006         Dim LowByte As Byte
007         Dim crc As Byte
008         Dim crc_ok As Byte
009         Dim data_Word As Integer
010         Dim voltage As Single
011
012         Try
013
014             ' This is where the data are received
015             If SerialPort1.IsOpen Then
016
017                 Control.CheckForIllegalCrossThreadCalls
                                                  = False
018
019                 ' How many Bytes are in the buffer
020                 in_bytes = SerialPort1.BytesToRead
021
022                 ' Collect all bytes
023                 For cnt = 1 To (in_bytes)
024                     input_data(cnt) = SerialPort1.ReadByte
025                 Next
026
027                 ' Empty buffer
028                 SerialPort1.DiscardInBuffer()
029
030                 HighByte = input_data(1)
031                 LowByte = input_data(2)
032                 crc = input_data(3)
033
034                 ' Checksum
035                 crc_ok = 170 Xor input_data(1) Xor input_
                                                  data(2)
036
037                 If crc = crc_ok Then
038
039                     ' High and Low Byte are
```

```
                                                assembled again
040                     data_Word = ((HighByte * 256) + LowByte)
041                     voltage = data_Word * (5.0 / 1024.0)
042
043                     ' RAW value conversion and display as
                                                    voltage
044                     Label1.Text = Format(voltage, "0.00 V")
045
046                 End If
047
048             End If
049
050         Catch ex As Exception
051         End Try
052
053     End Sub
```

In the function `SerialPort1_DataReceived()`, the most interesting part of the VB.NET-program will now follow. This function will be called every time data are received from the serial interface. Here, we read the bytes that our Arduino™-program sends, and process them right away. Before reading and processing, we always check if the connection is opened first and then check how many bytes are available in the reception buffer. Then we read the bytes into the `input_data()`-arrays and assign the received values to the variables `HighByte`, `LowByte` and `crc`. Last, we will calculate the checksum by applying the same procedure as in our Arduino™-program. If the calculated and the received checksum are the same, no transmission error has occurred. Now we need to perform measured value calculation. To get a certain formatting, we use the `Format()`-function of VB.NET and output the formatted value on the `Label1`.

## 16.1 | Expansion of the Measuring Range

If you want to measure higher voltages, you need a pre-voltage divider, consisting of the resistors R1 and R2. You can use them to expand the input voltage range as desired. However, observe that the resolution will also reduce with an increasing input voltage range.

In our experiment, which is meant for an input voltage of 5 V, we have a resolution of 0.00488 V or 4.88 mV per conversion step. Our digital value of the analogue input can dissolve 1,024 steps, since it has a digital resolution of 10 bits.

Conversion steps (Steps): $1{,}024 = 2^{10}$
Resolution per Digit = $U_{ADC}$ / Steps

5 V / 1,024 = 0.00488 V = 4.88 mV

If we were to raise the input voltage range to 30 V, the resolution would deteriorate five-fold ((30 V – 5 V) / 5 V = 5). We would »only« be working with a resolution of 0.0244 V = 24.4 mV anymore.

Now we can determine the voltage divider for a measuring range up to 30 V. We already know that we want to expand the input range from 5 V to 30 V, which corresponds to factor 5. The measuring input for voltage measurements must not be too low-Ohmic and should be at least 100 kΩ. Modern voltage meters in contrast, have an input resistance of 10 MΩ to put as little stress as possible on the voltage source and to avoid falsifying the measuring result as far as possible.

Let us assume an input resistance of approx. 100 kΩ and define the resistor R1 at 100 kΩ. In a serial circuit, the ratio of the voltages is identical to the ratio of the resistors to each other. The resistor R2 only needs to be 1/5 as large as the resistance of R1. We take 100 kΩ / 5 and will receive the value 20 kΩ for R2.

Let us now determine the current that flows in the circuit. In a serial circuit, the current through the resistors is the same and the voltages will be divided. The current through the resistors is calculated as follows:

$I = U / R_{Total}$
30 V / 120 kΩ = 0.25 mA

The following voltage drop would result at R2 then:

U = R2 x I
20 kΩ x 0.25 mA = 5 V

At an applied voltage of 15 V, the value at the ADC would be:

15 V / 120 kΩ = 0.125 mA
20 kΩ x 0.125 mA = 2.5 V

Let us calculate the power loss across the resistors at maximum input voltage now:

P = U x I
30 V x 0.25 mA = 7.5 mW

You can now calculate your own matching voltage divider and adjust the measured value in the program by multiplying it by your voltage divider factor. The measuring error can be determined by experimenting with a precise voltage meter and is included in calculation of the multiplication factor. Observe the resistance values of the E-series and do not try to include the mathematically determined values in a circuit by using unusual E-series or even trying to switch resistors in series or in parallel to get precisely the calculated resistance.

This calculation also does not consider the input resistance of the ADC. In the Arduino™-UNO ATmega328, it is at approx. 100 kΩ.

Therefore, we should not go any higher than 100 kΩ with the input voltage divider to still get useful measuring results. For more precise results, you may perform the calculation for a loaded voltage divider and include the resistance of the ADC in R2.

Also consider that the tolerance adds up in a serial circuit. If you are using two resistors with 5 % tolerance each for the circuit shown, the total tolerance is already at 10 %.

The values of the E12-series have turned out to be ideal. They are available in every electronic shop and are part of the standard series. When building meters, however, the tolerances of the components should be kept as low as possible for the most precise measuring results.

**Advice**
The following link will take your to an online tool with which you can calculate voltage dividers: http://www.peacesoftware.de/einigewerte/spannungsteiler.html

# TEMPERATURE DISPLAY IN DEGREES CELSIUS AND FAHRENHEIT

This experiment shows how you can use a cost-efficient temperature resistor like the NTC (Negative Temperature Coefficient Thermistor) used here to program a simple LCD-thermometer.

An NTC is a resistor that changes its resistance depending on its temperature. The NTC is called a hot conductor. This means that its resistance reduces when the temperature increases.

The circuit diagram shows the setup in more detail. This is another variable voltage divider, consisting of a 10-kΩ-fixed resistor and the variable NTC-resistor. As the temperature drops, the resistance of the NTC increases and so does the voltage at the analogue input A0.

**Example code: LCD THERMO**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
```

```
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define Backlight   9
009 #define ADC_NTC     0
010
011 float temp_celsius, temp_fahrenheit;
012 int ADC_raw;
013
014 float Grad_to_Fahrenheit(float grad)
015 {
016     return (9.0 / 5.0) * grad + 32;
017 }
018
019 void setup()
020 {
021     analogWrite(Backlight, 200);
022     lcd.begin(16, 2);
023     lcd.setCursor(0, 0);
024     lcd.print("THERMO – ARDUINO");
025     Serial.begin(9600);
026     delay(2000);
027     lcd.clear();
028 }
029
030 void loop()
031 {
032     ADC_raw = analogRead(ADC_NTC);
033     temp_celsius = (580.0 – ADC_raw) / 10;
034     temp_fahrenheit = Grad_to_Fahrenheit(temp_celsius);
035
036     lcd.setCursor(0, 0);
037     lcd.print(temp_celsius, 1);
038     lcd.write(223);
039     lcd.print("C  ");
040
041     lcd.setCursor(0, 1);
042     lcd.print(temp_fahrenheit, 1);
043     lcd.write(223);
044     lcd.print("F  ");
045
046     Serial.print("Temperature = ");
047     Serial.print(temp_celsius);
048     Serial.print(" °C");
049
050     Serial.print(" | ");
```

```
051     Serial.print(temp_fahrenheit);
052     Serial.println(" °F");
053
054     delay(1000);
055 }
```

The resistance curve of the NTC is not precisely linear and needs to be adjusted by a calculation.

```
001 temp_celsius = (580.0 – ADC_raw) / 10
```

To receive the output not only in degrees Celsius, the value will be converted to Fahrenheit and displayed on the LCD.

```
001 float Grad_to_Fahrenheit(float grad)
002 {
003     return (9.0 / 5.0) * grad + 32;
004 }
```

As you can see, we can write calculations at once after the command return and do not need to hand it over to a variable first. This function calculates the value in degrees Fahrenheit, as used in the US system based on the degrees Celsius.

In this program, the serial interface is also used and the same value is output on the LCD and additionally through an ASCII-string via the UART-interface. When opening the Arduino™-internal terminal program and setting its interface to 9,600 Baud, you will receive the measured values in plain text in the terminal program.

# TEMPERATURE PLOTTER WITH USB-INTERFACE

Let us expand the thermometer with a VB.NET-program and change the program code so that the plain text output as in the predecessor program will be replaced by sending only the

temperature value to the PC. The circuit remains the same, but the program is changed as follows:

```
001 Serial.flush()
002 highbyte=ADC_raw/256
003 lowbyte=ADC_raw%256
004 Serial.write(highbyte)
005 Serial.write(lowbyte)
006 crc=170^highbyte^lowbyte
007 Serial.write(crc)
```

As you can see, we transfer the temperature value like the measured value of the voltage in the digital USB-Voltmeter. You can find the complete program on the enclosed CD-ROM. It is called »TEMP_PLOT“. The program code does not need to be listed beyond this, because it corresponds to the predecessor program.

However, some things have changed in theVB.NET-program. It has been expanded by a graphical output. For this, a control element with the name »AutoRedraw« has been programmed that draws a continuous line in the X and Y directions, depending on the input variable, and serves as a temperature plotter. This program is enclosed as source code and can be used for your own experiments. A detailed description of the drawing functions would, however, exceed the scope of this learning package. Relevant websites such as www.vb-paradise.de and VB.NET-text books can be used to learn more about VB.NET-programming.

# WEBSYNCHRONOUS CLOCK

We have already programmed a clock in this learning package. Since it is not very precise and subject to a very large deviation in the course of the operating time, we now program a web-synchronous clock, knowing about the serial transmission between PC and Arduino™. It is websynchronous, because the Windows time is by default automatically reconciled with an online time server in the background. In this VB.NET-program, we will now send the time of the PC to Arduino™ and output it on the LCD. The VB.NET-program is enclosed as an executable EXE file and as source code.

**Upload**

The experiment requires the LCD basic writing that you set up in the function test.

**Example code: PC Time**

```
001 // Integrating LCD-Library
002 #include <LiquidCrystal.h>
003
004 // Specifying LCD pins
005 // RS, E, D4, D5, D6, D7
006 LiquidCrystal lcd(11, 10, 2, 3, 4, 5);
007
008 #define Backlight   9
009
010 byte Hour, Minute, Second;
011
012 void setup()
013 {
014     Serial.begin(9600);
015
016     analogWrite(Backlight, 200);
017
018     lcd.begin(16, 2);
019     lcd.clear();
020     lcd.setCursor(0, 0);
021     lcd.print("ARDUINO PC-CLOCK”);
022
023     Hour= 0;
024     Minute = 0;
025     Second= 0;
026 }
027
028 void loop()
029 {
030
031     if(Serial.available()>3)
032     {
033         Hour = Serial.read();
034         Minute = Serial.read();
035         Second = Serial.read();
036
037         lcd.setCursor(0, 1);
038         lcd.print("NOW: ");
039
040         if(Hour < 24)
041         {
042             if(Hour < 10) lcd.print("0");
043             lcd.print(Hour);
```

```
044             lcd.print(":");
045         }
046         if(Minute < 60)
047         {
048             if(Minute < 10) lcd.print("0");
049             lcd.print(Minute);
050             lcd.print(":");
051         }
052         if(Second < 60)
053         {
054             if(Second < 10) lcd.print("0");
055             lcd.print(Second);
056         }
057     }
058
059     Serial.flush();
060     delay(100);
061 }
```

The basic structure of the program corresponds to that of the regular clock (RTC) that we have already learned about at the beginning of the learning package.

This time, we receive no data in the VB.NET-program, but send data from the PC to the Arduino™. The data correspond to hours, minutes and seconds in the form of bytes. We read them with the Arduino™ when `Serial.available()` is larger than three. This characterises that three bytes are pending in the reception buffer. We use `Serial.read()` to systematically read in the three arriving bytes and assign them to the Arduino™-variables for hour, minute and second. That is  it on the side of Arduino™. Let us look at the transmission function in the VB.NET program. For this, we use a timer set to 500 ms.

```
001 Private Sub Timer1_Tick(sender As System.Object, e As
                    System.EventArgs) Handles Timer1.Tick
002
003         If SerialPort1.IsOpen Then
004
005             SerialPort1.Write(Chr(Now.Hour) & Chr(Now.
                                Minute) & Chr(Now.Second))
006             Label1.Text = Now.Hour & ":" & Now.Minute &
                                      ":" & Now.Second
007
008         End If
009 End Sub
```

The timer could also be set to 1,000 ms. It is also possible that the second output will jerk on the LCD now and then because the data overlap. It is better to set the update time a little faster or just to half, to make the output appear more fluent.

In the VB.NET-timer function, a check is performed before the actual output for safety purposes to see whether the serial connection is opened. If this is the case, `SerialPort1.Write()` will be used to transfer the time data. For this, we use the function `Now()`. It contains the time and date and may be instructed with the parameters `Hour`, `Minute` and `Second` to output only the desired places. To convert the values for the transmission to bytes, each value will be converted with `Char()` to one byte.

Our PC clock shows the precise time on the LCD now .

# ANNEX

## 20.1 | Electrical Units

We distinguish between voltage, current, resistance and the units in which the values are measured (e.g. Volt or Ampere). Every unit has an abbreviation that is used in the formulas. The abbreviations permit a short and well-structured annotation. Instead of current equal 1 Ampere we write only I = 1 A.

These abbreviations are used in all formulas in his book.

| Value | Abbreviation | Unit | Abbreviation |
|---|---|---|---|
| Voltage | U | Volt | V |
| Current | I | Ampere | A |
| Resistance | R | Ohm | Ω |
| Power | P | Watt | W |
| Frequency | f | Hertz | Hz |
| Time | t | Second | s |
| Wavelength | Λ (Lambda) | Metres | m |
| Inductiveness | L | Henry | H |
| Capacity | C | Farad | F |
| Area | A | Square me- | m² |

| tres |
|---|

## 20.2 | ASCII-Table

| Character | decimal | hexadecimal | binary | Description |
|---|---|---|---|---|
| Character | decimal | hexadecimal | binary | Description |
| NUL | 000 | 000 | 00000000 | Null Character |
| SOH | 001 | 001 | 00000001 | Start of Header |
| STX | 002 | 002 | 00000010 | Start of Text |
| ETX | 003 | 003 | 00000011 | End of Text |
| EOT | 004 | 004 | 00000100 | End of Transmission |
| ENQ | 005 | 005 | 00000101 | Enquiry |
| ACK | 006 | 006 | 00000110 | Acknowledgment |
| BEL | 007 | 007 | 00000111 | Bell |
| BS | 008 | 008 | 00001000 | Backspace |
| HAT | 009 | 009 | 00001001 | Horizontal TAB |
| LF | 010 | 00A | 00001010 | Line Feed |
| VT | 011 | 00B | 00001011 | Vertical TAB |
| FF | 012 | 00C | 00001100 | Form Feed |
| CR | 013 | 00D | 00001101 | Carriage Return |
| SO | 014 | 00E | 00001110 | Shift out |
| SI | 015 | 00F | 00001111 | Shift in |
| DLE | 016 | 010 | 00010000 | Data Link Escape |
| DC1 | 017 | 011 | 00010001 | Device Control 1 |
| DC2 | 018 | 012 | 00010010 | Device Control 2 |
| DC3 | 019 | 013 | 00010011 | Device Control 3 |
| DC4 | 020 | 014 | 00010100 | Device Control 4 |
| NAK | 021 | 015 | 00010101 | Negative Acknowledgment |
| SYN | 022 | 016 | 00010110 | Synchronous Idle |
| ETB | 023 | 017 | 00010111 | End of Transmission Block |
| CAN | 024 | 018 | 00011000 | Cancel |

| Character | decimal | hexadecimal | binary | Description |
|---|---|---|---|---|
| EM | 025 | 019 | 00011001 | End of Medium |
| SUB | 026 | 01A | 00011010 | Substitute |
| ESC | 027 | 01B | 00011011 | Escape |
| FS | 028 | 01C | 00011100 | File Separator |
| GS | 029 | 01D | 00011101 | Group Separator |
| RS | 030 | 01E | 00011110 | Request to Send, Record Separator |
| US | 031 | 01F | 00011111 | Unit Separator |
| SP | 032 | 020 | 00100000 | Space |
| ! | 033 | 021 | 00100001 | Exclamation Mark |
| « | 034 | 022 | 00100010 | Double Quote |
| # | 035 | 023 | 00100011 | Number Sign |
| $ | 036 | 024 | 00100100 | Dollar Sign |
| % | 037 | 025 | 00100101 | Percent |
| & | 038 | 026 | 00100110 | Ampersand |
| ‘ | 039 | 027 | 00100111 | Single Quote |
| ( | 040 | 028 | 00101000 | Left Opening Parenthesis |
| ) | 041 | 029 | 00101001 | Right Closing Parenthesis |
| * | 042 | 02A | 00101010 | Asterisk |
| + | 043 | 02B | 00101011 | Plus |
| , | 044 | 02C | 00101100 | Comma |
| - | 045 | 02D | 00101101 | Minus or Dash |
| . | 046 | 02E | 00101110 | Dot |
| / | 047 | 02F | 00101111 | Forward Slash |
| 0 | 048 | 030 | 00110000 | |
| 1 | 049 | 031 | 00110001 | |
| 2 | 050 | 032 | 00110010 | |
| 3 | 051 | 033 | 00110011 | |
| 4 | 052 | 034 | 00110100 | |
| 5 | 053 | 035 | 00110101 | |
| 6 | 054 | 036 | 00110110 | |

| Character | decimal | hexadecimal | binary | Description |
|---|---|---|---|---|
| 7 | 055 | 037 | 00110111 | |
| 8 | 056 | 038 | 00111000 | |
| 9 | 057 | 039 | 00111001 | |
| : | 058 | 03A | 00111010 | Colon |
| ; | 059 | 03B | 00111011 | Semi-Colon |
| < | 060 | 03C | 00111100 | Less Than |
| = | 061 | 03D | 00111101 | Equal |
| > | 062 | 03E | 00111110 | Greater Than |
| ? | 063 | 03F | 00111111 | Question Mark |
| @ | 064 | 040 | 01000000 | AT Symbol |
| A | 065 | 041 | 01000001 | |
| B | 066 | 042 | 01000010 | |
| C | 067 | 043 | 01000011 | |
| D | 068 | 044 | 01000100 | |
| E | 069 | 045 | 01000101 | |
| F | 070 | 046 | 01000110 | |
| G | 071 | 047 | 01000111 | |
| H | 072 | 048 | 01001000 | |
| I | 073 | 049 | 01001001 | |
| J | 074 | 04A | 01001010 | |
| K | 075 | 04B | 01001011 | |
| L | 076 | 04C | 01001100 | |
| M | 077 | 04D | 01001101 | |
| N | 078 | 04E | 01001110 | |
| O | 079 | 04F | 01001111 | |
| P | 080 | 050 | 01010000 | |
| Q | 081 | 051 | 01010001 | |
| R | 082 | 052 | 01010010 | |

| Character | decimal | hexadecimal | binary | Description |
|---|---|---|---|---|
| S | 083 | 053 | 01010011 | |
| T | 084 | 054 | 01010100 | |
| U | 085 | 055 | 01010101 | |
| V | 086 | 056 | 01010110 | |
| W | 087 | 057 | 01010111 | |
| X | 088 | 058 | 01011000 | |
| Y | 089 | 059 | 01011001 | |
| Z | 090 | 05A | 01011010 | |
| [ | 091 | 05B | 01011011 | Left opening Bracket |
| \ | 092 | 05C | 01011100 | Back Slash |
| ] | 093 | 05D | 01011101 | Right closing Bracket |
| ^ | 094 | 05E | 01011110 | Caret |
| _ | 095 | 05F | 01011111 | Underscore |
| ` | 096 | 060 | 01100000 | |
| a | 097 | 061 | 01100001 | |
| b | 098 | 062 | 01100010 | |
| c | 099 | 063 | 01100011 | |
| d | 100 | 064 | 01100100 | |
| e | 101 | 065 | 01100101 | |
| f | 102 | 066 | 01100110 | |
| g | 103 | 067 | 01100111 | |
| h | 104 | 068 | 01101000 | |
| i | 105 | 069 | 01101001 | |
| j | 106 | 06A | 01101010 | |
| k | 107 | 06B | 01101011 | |
| l | 108 | 06C | 01101100 | |
| m | 109 | 06D | 01101101 | |
| n | 110 | 06E | 01101110 | |

| Character | decimal | hexadecimal | binary | Description |
|---|---|---|---|---|
| o | 111 | 06F | 01101111 | |
| p | 112 | 070 | 01110000 | |
| q | 113 | 071 | 01110001 | |
| r | 114 | 072 | 01110010 | |
| s | 115 | 073 | 01110011 | |
| t | 116 | 074 | 01110100 | |
| u | 117 | 075 | 01110101 | |
| v | 118 | 076 | 01110110 | |
| w | 119 | 077 | 01110111 | |
| x | 120 | 078 | 01111000 | |
| y | 121 | 079 | 01111001 | |
| z | 122 | 07A | 01111010 | |
| { | 123 | 07B | 01111011 | Left opening Brace |
| \| | 124 | 07C | 01111100 | Vertical Bar |
| } | 125 | 07D | 01111101 | Right closing Brace |
| ~ | 126 | 07E | 01111110 | Tilde |
| DEL | 127 | 07F | 01111111 | Delete |

# PROCUREMENT SOURCES

**Conrad Electronic SE**
**Klaus-Conrad-Straße 1**
**92240 Hirschau**
**www.conrad.de**

**Electronic Assembly GmbH**
**Zeppelinstraße 19**
**82205 Gilching bei München**