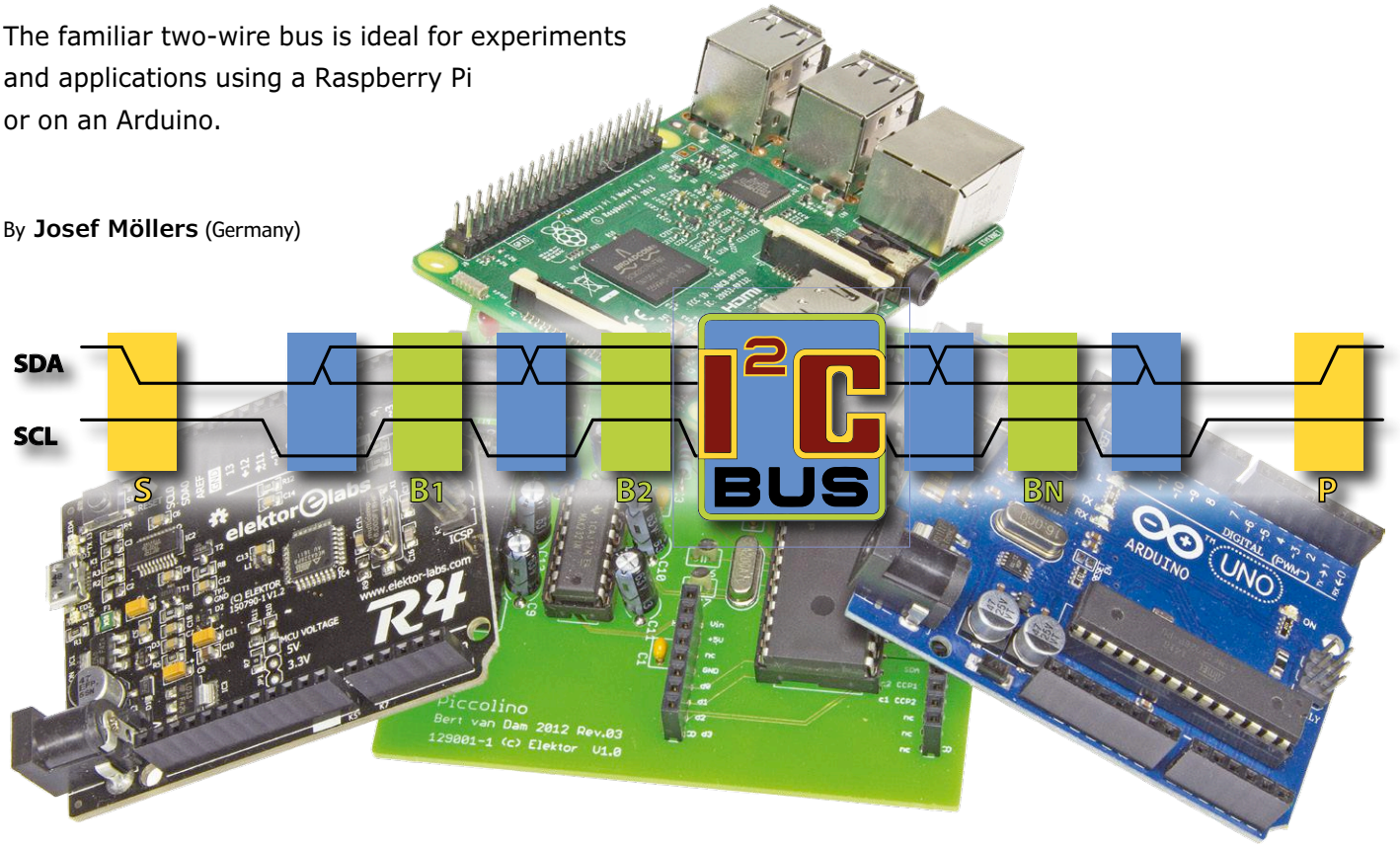
The I²C Bus

Part 2: using the bus with a microcontroller

The familiar two-wire bus is ideal for experiments and applications using a Raspberry Pi or on an Arduino.

By **Josef Möllers** (Germany)



Raspberry Pi, BeagleBone, Arduino, Genuino, ATmega, PIC, practically any PC: pretty much anything you can find sitting on a maker's work bench that can compute will have one or more I²C interfaces. Using the example of the LM75 temperature sensor we will look in this article at how the I²C interfaces of the Raspberry Pi, ATmega and Arduino can be used as bus masters, and, where possible, as bus slaves.

Raspberry Pi

The Raspberry Pi has two physical I²C buses, of which only one can normally be used directly. The Raspberry Pi board comes with pull-up resistors to the 3.3 V supply already fitted and permanently enabled. SDA and SCL are available on pin 3 (SDA) and pin 5 (SCL) of the expansion header, conveniently right next to the 3.3 V supply on pin 1, the 5 V supply on pins 2 and 4, and ground on pin 6. These pins belong to I²C bus number 1. It is therefore possible to make a very compact plug-in expansion board offering temperature sensing, real-time clock or position sensing functions.

Caution: the ports of the Raspberry Pi must **only be operated at 3.3 V** and connection to 5 V signals can damage the device. It is important, therefore, to check your circuit carefully before connecting it to the ports on the Raspberry Pi. In particular take care that no I²C slave contains pull-up resistors to the 5 V rail. If present, any such resistors should be removed: the slave will still work without them.

The Raspberry Pi provides a convenient environment for learning how to use new and unfamiliar I²C slave devices. **Figure 1** shows how a breadboard can be used to connect an LM75 to a Raspberry Pi.

Operation as bus master

Before the I²C bus on the Raspberry Pi can be used under Raspbian, it is necessary to install two extra drivers. To do this, launch `raspi-config` and select the `I2C` option under the `Advanced Options` menu item. This will enable the interface and load the necessary kernel module. Alternatively add the following lines to the file `/etc/modules` using a text editor:

```
i2c-dev
i2c-bcm2708
```

After restarting the system the two drivers (as well as any other extra drivers that are required) will be loaded and the device nodes `/dev/i2c-<n>` will be created. This can be confirmed using the following commands at the '$ ...' prompt.

```
$ lsmod | i2c_
i2c_dev          XXXX  0
i2c_bcm2708      YYYY  0
$ ls /dev/i2c-*
dev/i2c-1
```

In the above XXXX and YYYY stand for the size of the modules, while the two zeros indicate that no programs are currently using the modules. The commands will, incidentally, work even without root privileges.
Next install the `i2c-tools` package, which includes among other things code to detect I²C devices and buses:

```
sudo apt-get install i2c-tools
```

Raspbian already includes drivers from some I²C peripheral devices, including for the RV-8523 real-time clock (RTC). Unless configured otherwise, the Raspberry Pi drives the I²C bus in standard mode at 100 kbps.

For a first test you can use the `i2cdetect` tool from a terminal window to obtain an overview of the slave devices that the system recognizes on I²C bus number 1. The results might appear as shown in **Figure 2**, where the LM75 is responding to address 0x48.

The commands `i2cget`, `i2cset`, and `i2cdump` can be used to communicate with the LM75 without having to get involved in programming. In the example below 0x00 is the register number, which must always be specified.

```
pi@raspberrypi ~ $ i2cget -y 1 0x48 0x00 w
0xa010
```

The two bytes of the reply must be swapped over, giving 0x10a0. Of this result only the upper nine bits are valid: they are 0x021. The LM75 reports temperature in steps of 0.5 K, and so the temperature reading in this example is 16.5 °C.
The hardware of the Raspberry Pi in principle also supports operation as an I²C slave, but this is not supported by the Linux driver.

Programming in C and Python
Five functions are required to program the I²C bus **in C**.

- `open()` to access the I²C device node;
- `ioctl()` to set the I²C slave parameters;
- `read()` and `write()` for the actual communication with the slave; and
- `close()` to indicate when access to the I²C device node is no longer required.

For the following code, in addition to the other include files, the include file `linux/i2c-dev.h` is required for the definition of `I2C_SLAVE`.

```
# include <linux/i2c-dev.h>
```

The device node is accessed using the `open()` function.

```
fd = open("/dev/i2c-1", O_RDWR);
```

Next we set the slave address with an `ioctl()` call.

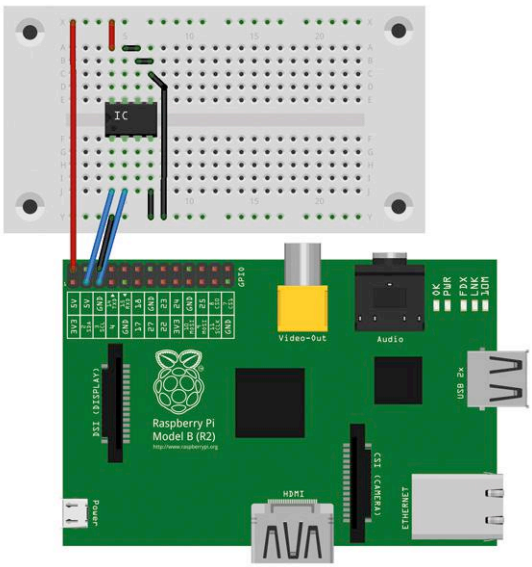


Figure 1. Connecting an LM75 to a Raspberry Pi using a breadboard.

```
ioctl(fd, I2C_SLAVE, 0x48);
```

And now we can write to and read from the device.

```
unsigned char buf[2];
float T;
buf[0] = 0;
write(fd, buf, 1);    /* write register number 0 */
read(fd, buf, 2);     /* read temperature register */
T = ((buf[0]<< 8) | buf[1]) / 256.0;
```

Finally we close the device node: Geräteknoten wieder:

```
close(fd);
```

Of course it is a good idea to check the return values from the function calls properly in order to detect possible errors, such

```
pi@raspberrypi ~ $ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- 48 -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Figure 2. The i2cdetect tool has found an LM75 at address 0x48.

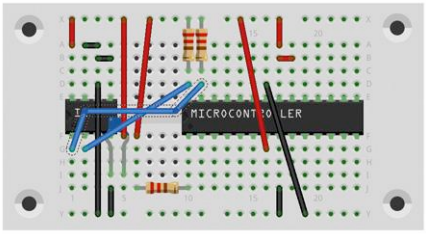


Figure 3. The ATmega88 and the sensor on a breadboard.

as whether the user running the code has permission to open the device node `/dev/i2c-1`, whether a slave exists at address 0x48, and whether the data transfers were successful.

Before programming the I²C bus **in Python**, it is necessary to install (or to have already installed) the `python-smbus` Raspbian package.

```
sudo apt-get install python-smbus
```

Then the temperature can be read from the LM75 as follows.

```
#! /usr/bin/python

import smbus

import time

bus = smbus.SMBus(1)

address = 0x48

w = bus.read_word_data(address, 0)

print format(w, '04x')
```

Listing 1. Main loop for reading from the LM75.

```
# include <i2cmaster.h>

# define LM75 (0x48 << 1)
                              // see datasheet
int
main(void)
{
unsigned char val[2];

i2c_init();
                              // initialisation I2C

i2c_start(LM75 | I2C_WRITE);
                              // addressing to write
i2c_write(0x00);
                              // temperature register
i2c_rep_start(LM75 | I2C_READ);
                              // addressing to read
val[0] = i2c_read();
                              // degrees Celsius
val[1] = i2c_read();
                              // tenth part bit
i2c_stop();
                              // ready
for(;;);
}
```

The "0" in the call to the `read_word_data` method supplies the register number, which here is the index of the temperature register. In the SMBus protocol, which sits on top of the I²C protocol, a register number must always be set at the start of a communication. This causes problems in the case of the PCF8574 port expansion chip, for example, which has no address register.
The returned value has the same problem as with the `i2cget` command: the two bytes in the word `w` must be swapped over.

ATmega

The Atmel ATmega microcontroller series have an integrated I²C controller which supports both standard mode (at 100 kHz) and fast mode (at 400 kHz). It can be operated as a master, as a slave or as a combination of the two. The ATmega324PB and the ATmega328PB devices include two I²C buses. In the following examples we will be using the ATmega88, which can be plugged into a breadboard along with the sensor as shown in **Figure 3**.
Here the best approach is to use the `i2cmaster` library by Peter Fleury [1]. Note, however, that this library does not automatically enable the internal pull-up resistors! It is therefore necessary to take care of this ourselves: Figure 3 shows the resistors towards the top of the breadboard, running to the 5 V rail. The library configures the I²C bus controller in the ATmega to run in standard mode (100 kHz). The LM75 sensor can be accessed using code like that shown in **Listing 1**.
Using `lcdlibrary`, by the same author, we can construct a digital thermometer with an external liquid crystal display. This can be connected using an I²C interface board (which will usually be based on a PCF8574), or alternatively an LCD with built-in I²C interface can be used.
The I²C bus controller in the ATmega devices does not have to be operated in master mode, controlling a slave. It can also be run in slave mode, controlled by an external master. A composite mode of operation is also available, where, for example, the ATmega might at one moment be communicating with the LM75 as a bus master, and then at the next moment be acting as a slave to a Raspberry Pi master. One situation where an ATmega might be used as a slave is where the ATmega is reading data in real time over its port pins, doing some processing, and then supplying the results upon request to a Raspberry Pi. This kind of set-up is more complex, and entails tight coupling with the rest of the code running on the ATmega. There do exist, however, the rudiments of a library implementing operation in slave mode [2].
The description below follows that given in Atmel's datasheets, which contain tables of the various states of the I²C controller. For example, in the datasheet for the ATmega48/88/168, the relevant information can be found in section 22.7.

It is sensible to make the software implementing I²C slave mode run under interrupts. The bus controller hardware can trigger an interrupt under the following conditions.

- after transmitting a ‘start condition’ or ‘repeated start condition’;
- after transmitting the address and read/write bit;
- after transmitting a data byte;
- when the ATmega has lost an address arbitration (when a collision occurs while transmitting the ‘start condition’, the address byte or the read/write bit);
- when the ATmega has detected a ‘start condition’ and has been addressed as a slave;
- when the ATmega has received a data byte;
- when the ATmega has detected a ‘stop condition’, or a ‘repeated start condition’ where it has (again) been addressed as a slave; or
- when an invalid bus transaction has been detected.

The first four of these situations are only relevant to master mode, and so we are only interested in the last four. The I²C peripheral unit must be initialized with the desired slave address. It can then be enabled and it will start to run.

```
TWAR = (I2C_Slave_Addr << 1);
TWCR = _BM(TWEA) | _BM(TWEN) |
_BM(TWIE);
```

There is no need to set the bit rate for slave mode, as the communication speed is determined by the master. When an interrupt occurs, the first step is to determine its cause. To do this it is necessary to read the `TWSR` status register and examine its top five bits.

Listing 2 shows a fragment of the interrupt service routine (ISR) for receiving and transmitting data as a slave, and for detecting a ‘stop condition’.

The descriptions and tables in the ATmega datasheets are very comprehensive, including explanations of the status codes, actions required in software, and the resulting behavior of the hardware.

Arduino

Many Arduinos are based on Atmel ATmega-series microcontrollers, and so the above discussion applies equally to them. However, a popular and very convenient library called `Wire.h` is also available for the Arduino.

Working with the Arduino is just as straightforward as working with the Raspberry Pi (see **Figure 4**). Depending on the exact model of Arduino (or clone), you may find that the processor

Listing 2. Interrupt service routine to deal with address matching and detection of ‘stop condition’. The complete and extensively commented code is available on the project web page [5].

```
ISR(TWI_vect)
{
    /*
     * These variables need to be preserved across interrupts
     */
    static unsigned char i2c_idx,       /* Index into twi_msg[] */
                         i2c_tosend;    /* Number of bytes to send */

    switch (TWSR & 0xf8)
    {
    ...
    /*
     * RECEIVE Code
     * See Table 19-4.  Status Codes for Slave Receiver Mode
     * [Page 229]
     */
    case 0x60:
        /*
         * Own SLA+W has been received; ACK has been returned
         * TWDR: No TWDR action
         * STA=X STO=0 TWINT=1 TWEA=1
         * Data byte will be received and ACK will be returned
         */
        TWCR = (TWCR & ~_BM(TWSTO)) | (_BM(TWINT) | _BM(TWEA));
        i2c_idx = 0;
        break;
    …
    case 0xA0:
        /*
         * A STOP condition or repeated START condition has been
         * received while still addressed as slave
         * TWDR: No action
         * TWA=0 STO=0 TWINT=1 TWEA=1
         * Switched to the not addressed Slave mode;
         * own SLA will be recognized;
         * GCA will be recognized if TWGCE = "1"
         */
        TWCR = (TWCR & ~(_BM(TWSTA) | _BM(TWSTO))) | (_BM(TWINT) | _BM(TWEA));
        break;
    …
    case 0xA8:
        /* Own SLA+R has been received; ACK has been returned
         * TWDR: Load data byte
         */
        /*
         * The address (register number) has been received,
         * Start sending payload
         */
      TWDR = 0x42;
      break;
    }
}
```

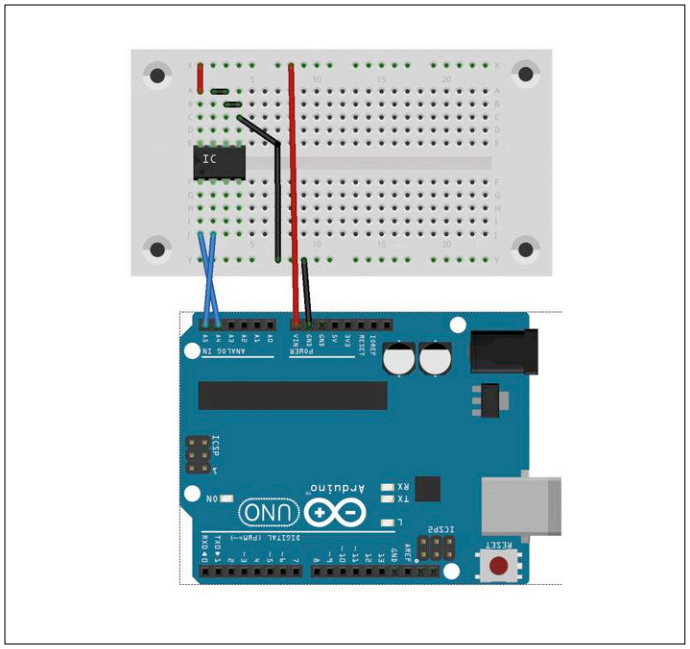


Figure 4. Connecting to an Arduino is practically the same as connecting to a Raspberry Pi.

runs on 3.3 V and that its ports cannot tolerate 5 V signals. So check and measure before connecting your circuit and if necessary remove any pull-up resistors on the slave.

The `Wire` library can be found directly within the Arduino IDE. As in the case of Peter Fleury's `i2cmaster` library the library and I²C interface must be initialized at the start of your code.

```
#include <Wire.h>
void setup() {
 Wire.begin();
}
```

The call to `Wire.begin()` enables the internal pull-up resistors. The value of these resistors is relatively high, which can cause problems: if so, add two external resistors with a value of 10 kΩ to 20 kΩ in parallel. Alternatively, disable the internal resistors altogether and just rely on external pull-ups (which should then be in the region of 4.7 kΩ). This can be done with the following two lines of code after the call to `Wire.begin()`.

```
digitalWrite(SDA, 0);
digitalWrite(SCL, 0);
```

The I²C bus on a PC

Practically every PC has its own I²C interface, and most have several such interfaces. There are I²C slaves in displays (DDC [3]) and DRAM DIMMs (SPD [4]). Internal temperature sensors are also often connected over I²C. Unfortunately there is scant to non-existent manufacturer information on the devices used and on whether and how the buses can be accessed externally: sometimes a bus will be used only within a particular module. If the PC is running Linux, it is easy to run some experiments by loading the `i2c-dev` module as follows.

```
$ sudo modprobe i2c-dev
```

You can then look at what device nodes are present in `/dev`.

```
$ ls /dev/i2c*
/dev/i2c-0  /dev/i2c-1  /dev/i2c-2  /
dev/i2c-3 /dev/i2c-4 /dev/i2c-5
```

Up to this point you do not need root privileges. Under Debian and its derivatives (which includes Raspbian and Ubuntu) you can install `i2c-tools`.

```
apt-get install i2c-tools
```

Sometimes there are slaves on only one of the buses, as in the following example.

```
root@bounty:~# i2cdetect -y 5
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- 37 -- -- 3a -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- 58 -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

It is not always clear what peripheral devices are present. In the example below the monitor's EDID PROM is at address 0x50.

```
root@bounty:~# i2cdump -y 5 0x50
No size specified (using byte-data access)
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 00 ff ff ff ff ff ff 00 1a b3 d4 07 ec 22 02 00    ........?????"?.
10: 0a 16 01 03 80 34 20 78 2a ef 95 a3 54 4c 9b 26    ?????4 x*???TL?&
20: 0f 50 54 a5 4b 00 81 80 81 00 81 0f 95 00 95 0f    ?PT?K.???.???.??
30: a9 40 b3 00 01 01 28 3c 80 a0 70 b0 23 40 30 20    ?@?.??(<??p?#@0
40: 36 00 06 44 21 00 00 1a 00 00 00 fd 00 38 4c 1e    6.?D!..?...?.8L?
50: 52 10 00 0a 20 20 20 20 20 20 00 00 00 fc 00 42    R?.?      ...?.B
60: 32 34 57 2d 35 20 45 43 4f 0a 20 20 00 00 00 ff    24W-5 ECO?  ....
70: 00 59 56 32 45 31 34 30 30 31 32 0a 20 20 00 8e    .YV2E140012?  .?
```

If your PC has a VGA connector, you can try connecting an I²C slave to pin 12 (SDA) and pin 15 (SCL). Pins 6 (SCL) and 7 (SDA) of a DVI connector and pins 15 (SCL) and 16 (SDA) of an HDMI connector are also good candidates for experimentation. Unfortunately in some cases these buses are under the sole control of the graphics card and its firmware.

Note that this is not an official solution and that things may change in the future. In any case it is a good idea to measure the effective pull-up resistance with a multimeter after initialization is complete.

Caution: when the internal pull-up resistors are disabled in this way there will be a brief period during which they are enabled. This can cause problems if a 3.3 V slave with two external pull-ups to 3.3 V is connected to a 5 V Arduino. In this case it is essential to use a level-shifting circuit.
After initialization, data can be sent to the I²C slave in the `loop()` function. The following example sets the address pointer in the LM75 to point to the temperature register.

```
void loop() {
 Wire.beginTransmission(0x48);
 Wire.write(byte(0x00));
 Wire.endTransmission();
```

The following fragment reads from the LM75.

```
Wire.requestFrom(0x48, 2);
 c1 = Wire.read();
 c2 = Wire.read();
}
```

As you can see, the `Wire` library requires that you first specify the number of bytes expected from the slave (in this case, 2). The first call reads in the two bytes, and then the subsequent calls allow you to access the received data.
The Arduino `Wire` library can also be configured for operation in slave mode. Again, the library must first be initialized. In this case the call to `Wire.begin()` must be given a parameter which is the desired slave address: it is the presence of this parameter that selects slave mode. It is also necessary to set up an event handler which will be called whenever the Arduino is addressed as a slave. In the example below the event handler is called when the Arduino is to receive data.

```
#include <Wire.h>
void setup() {
 Wire.begin(0x48);
 Wire.onReceive(receive);
}
```

Although not normally necessary for a slave device, the call to the `Wire.begin()` method again enables the built-in pull-up resistors. They can be disabled if necessary as described above.

Caution: note again that although a 5 V Arduino can be configured as a slave to a 3.3 V master, there will be a brief pulse to 5 V on the signal lines which may damage the 3.3 V master. So, for example, if you want to operate an Arduino as a slave to a Raspberry Pi, you must make certain that the Arduino is only electrically connected to the Raspberry Pi after it has completed initialization.
When a data byte is received, the event handler (which was configured to be `receive` in the above example code) will be called. Again the data bytes have already been read in, and the number of bytes received is passed as a parameter to the handler.

```
void receive(int n) {
 while (n--) {
  uint8_t c = Wire.read();
  // Verarbeite c
 }
}
```

If you wish to transfer data to the master then you should call the method `onRequest()` instead of `onReceive()`. Again, the name of the event handler function that will produce the data to be sent is passed as a parameter. Since at the moment of calling the handler function the number of bytes to be sent is not known, no parameter is passed to it. The data bytes are sent using **a single** call to `Wire.write()`.

```
#include <Wire.h>
void setup() {
 Wire.begin(0x48);
 Wire.onRequest(transmit);
}
void loop() {
 while (1) delay(1000);
}
void transmit() {
 uint8_t msg[N];
 // Erzeuge msg[] Inhalt
 Wire.write(msg, N);
}
```

The two event handler set-up methods can be called together in the same sketch.

```
#include <Wire.h>
void setup() {
 Wire.begin(0x48);
 Wire.onReceive(receive);
 Wire.onRequest(transmit);
}
```

Such a configuration would allow you to emulate an LM75 accurately, using a 1-wire temperature sensor such as the DS18B20 instead of the LM75, or to emulate a real-time clock peripheral receiving time over DCF77 or GPS.

Coming up in part three

The next installment in this short series will look at some popular I²C peripheral devices: besides the LM75 temperature sensor we will also examine the PCF8574 port expander and the RV-8523 real-time clock. Finally we will close with some thoughts on troubleshooting using tools within the means of the average hobbyist. ⏮

(160418)

Web Links

[1] http://homepage.hispeed.ch/peterfleury/avr-software.html
[2] www.jtronics.de/avr-projekte/library-i2c-twi-slave.html (in German, English machine translation available)
[3] https://en.wikipedia.org/wiki/Display_Data_Channel
[4] https://en.wikipedia.org/wiki/Serial_presence_detect
[5] www.elektormagazine.com/160148