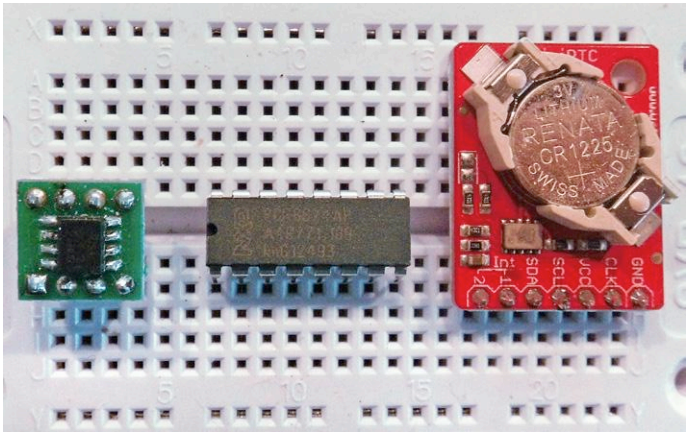
The I²C Bus

Part 3: components and troubleshooting



This last installment of our series picks out three I²C devices for closer examination: a temperature sensor, a port expander and a real-time clock. We will see how to read from and write to the registers on these devices and look at some handy software and hardware tools.

By Josef Möllers (Germany)

There exist many components and modules with an I²C interface, ranging from temperature, position and motion sensors to real-time clocks and LED and graphical displays. You need only enter 'I2C' into the search box of your favorite on-line supplier of electronic components to get an idea of the wide range of options available. From the pages and pages of results [1] we have selected a couple of devices to look at in more detail.

One advantage of I²C devices is that they do not require complicated wiring to connect to their host. A four-way ribbon cable is all that is need to carry both power and data. Our lead photograph shows the three devices we have selected for this article side-by-side on a breadboard. From left to right they are: a type LM75 temperature sensor on a breakout board; a PCF8574 port expander IC; and a real-time clock module using an RV-8523 RTC chip, which you can see just below and to the left of the coin cell.

LM75

The LM75 is the de facto standard temperature sensor with an I²C connection. Of the seven bits of its address only the upper four are fixed (at 1001); the other three bits can be set using external circuitry. Up to eight LM75s can therefore be connected to a single I²C bus, with addresses ranging from 0x48 to 0x4F. So if, for example, LM75s are to be used in a temperature monitoring application in a desktop PC, it is

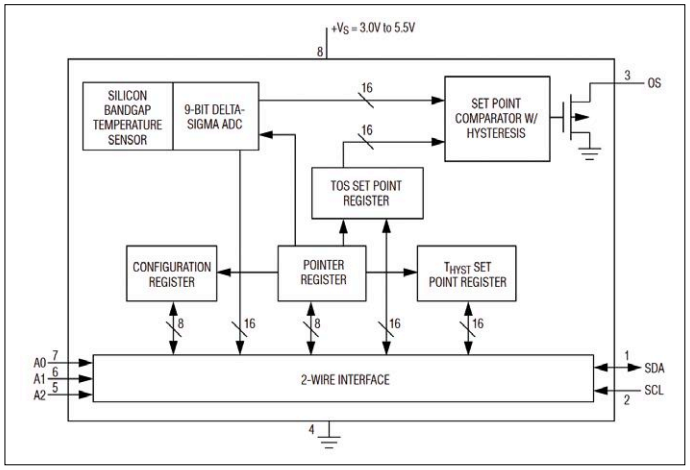


Figure 1. Innards of the type LM75 temperature sensor. (Source: Maxim)

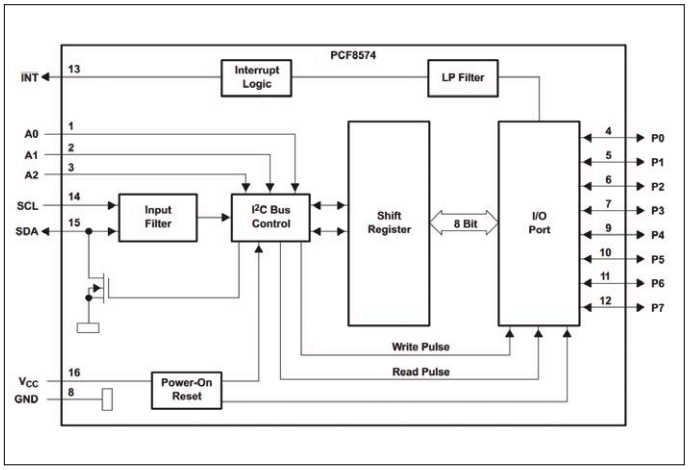


Figure 2. Internal circuit of the PCF8574 port expander. (Source: Texas Instruments)

possible to measure the temperature at up to eight different places within the case.
Internally the LM75 has four registers, which are addressed using two bits (see **Figure 1**):

- 00H: a 16-bit temperature register, which can only be read from;
- 01H: an 8-bit configuration register;
- 02H: a 16-bit hysteresis register;
- 03H: a 16-bit threshold register.

At power up the temperature register is selected by default, but even if that is the only register you wish to access it is always a good idea to write the register's address before reading it. When writing to a register the address must be given: the first byte after the write command is always interpreted by the LM75 as a register number.
After the register number come the data. In the case of a 16-bit register the more significant byte is transferred first, followed by the less significant byte.
In contrast to some other I²C devices the LM75 does not automatically increment the register number after each access: the register pointer remains fixed. If there is only one bus master and only the temperature reading is of interest, it is therefore unnecessary to reset the register pointer to zero before each access. All you need to do in the read operation is simply transfer the two data bytes representing the temperature.
The accuracy of the LM75 does leave a little to be desired. According to the datasheet the reading can be in error by up to 2 °C. However, there are alternative devices, such as the TMP275, which are more accurate and in general protocol- and register-compatible with the LM75.
The LM75 is also rather sensitive to interference on its power supply lines. To avoid collecting garbage instead of temperature readings, it is wise to stick to the 'one 100 nF capacitor per package' rule of thumb.

PCF8574

The PCF8574 is a 'remote 8-bit I/O expander', a parallel I/O chip controlled over an I²C bus interface: see **Figure 2**. It comes in two variants, which differ only in their I²C slave address. In the case of the PCF8574 the upper four bits of the address are 0100, while in the case of the PCF8574A they are 0111. This means that it is possible to connect up to sixteen PCF8574-series devices to a single I²C bus.
Internally there is just one register, which is directly connected to the port. When a bit pattern is written to the register, the port pins change state; if a bit is set to '1', then the pin can also be used as an input. When the register is read the device returns the logic levels on the external I/O pins. **Figure 3** shows how to connect a PCF8574 with an LED wired to port pin P0 to a Raspberry Pi. If a '1' is written to the PCF8574:

```
i2cset -y 1 0x40 0x01
```

the LED will light. If a '0' is written:

```
i2cset -y 1 0x40 0x00
```

the LED will extinguish. Strictly speaking we are here setting the register number to `0x01` or `0x00`, since, according to the manual and online guides the `i2cset` command expects a register number after the device address. However, the port expander interprets the number as a data value representing the desired bit pattern on its outputs.
It is sometimes desirable to use a device like this to provide a degree of isolation between a computer and a peripheral: if an output pin should accidentally be shorted to 12 V, for example, then it is only the PCF8574 that is likely to suffer any harm.
The PCF8574 is also used on simple LCD interface boards, allowing an I²C bus to drive the common one- or two-line LCD panels that employ the HD44780 controller IC. The LCD is operated in four-bit mode, and three further pins on the

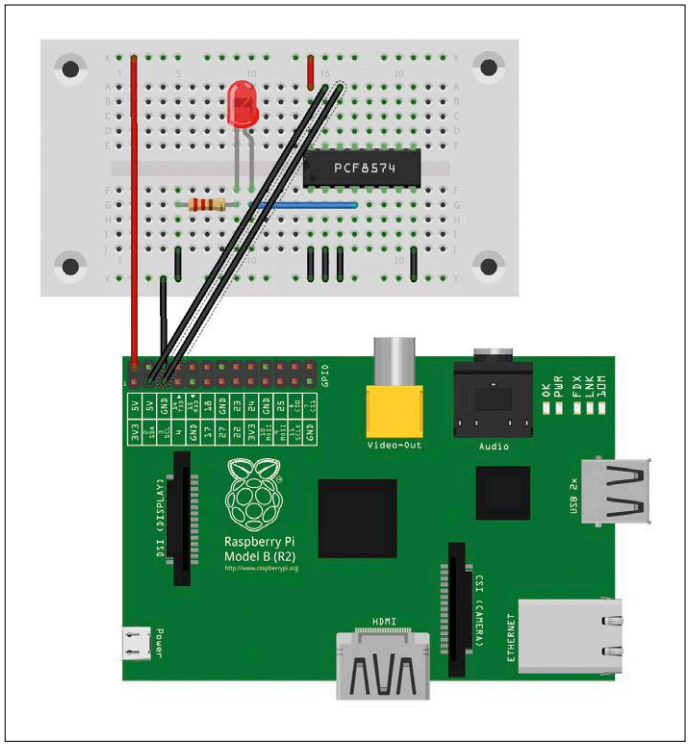


Figure 3. The PCF8574 port expander connected to a Raspberry Pi.

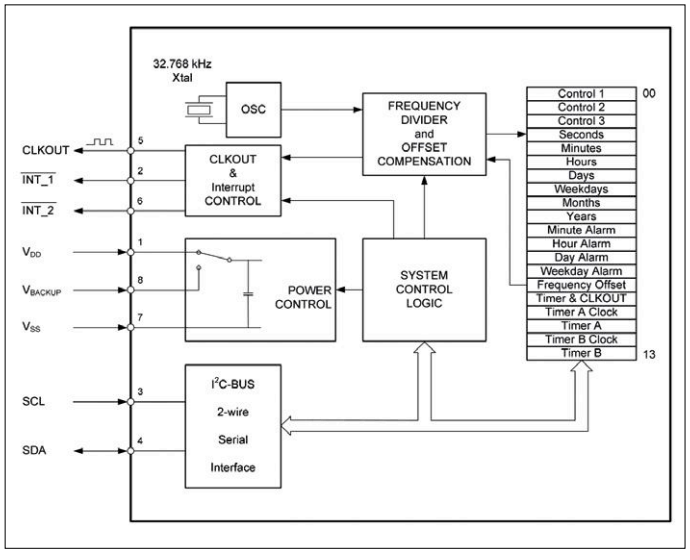


Figure 4. This real-time clock IC has twenty addressable control, time and alarm registers. (Source: Micro Crystal)

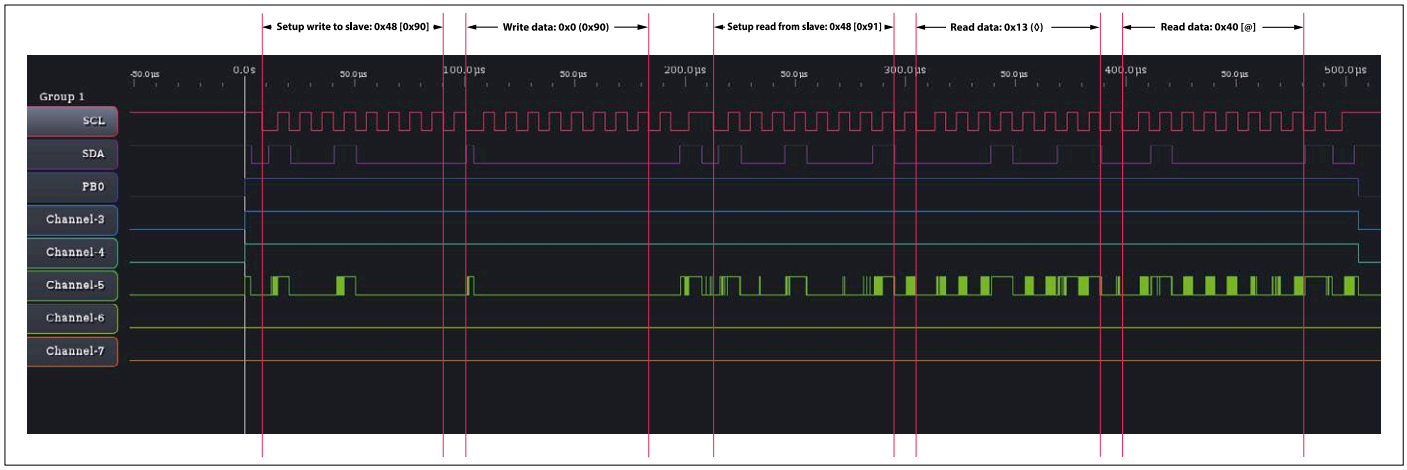


Figure 5. The Open Logic Sniffer looks on while an ATmega88 reads temperature data from an LM75.

PCF8574 are connected to the LCD panel's E, RS and R/W signals. The one remaining port bit is sometimes used to control the LCD backlight. Some of these boards include pull-up resistors on the bus lines to 5 V. These should normally be removed, or not fitted in the first place.

RV-8523

The RV-8523 is a real-time clock (RTC) with 20 registers each eight bits wide. The registers, numbered from 00H to 13H inclusive, are listed on the right in **Figure 4**. The RTC has an internal supply voltage monitor and can switch itself automatically over to battery power. As the lead photograph shows, the device is available in module form complete with battery holder.

After the register number (from `0x00` to `0x13`) has been sent, the register can be accessed. In contrast to the LM75 this device automatically increments the register pointer, wrapping round from `0x13` to `0x00`. It is therefore possible to read from or write to all twenty registers in a single operation.

Suppose for example that we wish to read just the date and time. We set the register pointer to 3 and read seven bytes. Using the Arduino Wire library the code might look like the following.

```
Wire.beginTransmission(0x68);
 Wire.write(byte(0x03));    // set register number to
3
 Wire.endTransmission();
 Wire.requestFrom(0x68, 7); // read time and date
 seconds = Wire.read();
 tenseconds = (seconds >> 4) & 0x07; seconds &= 0x0f;
 minutes = Wire.read();
 tenminutes = (minutes >> 4) & 0x07; minutes &= 0x0f;
 ...
```

The resulting values are BCD-encoded, and so conversion to binary may be required.

Besides the clock itself, the RV-8523 also has an alarm function that can produce an interrupt at a specified point in time. The only wrinkle is that although the INT_1 output goes low at the appointed hour, it does not automatically go high again: it is necessary to reset the alarm interrupt explicitly with a write to AF in control register 2.

Some operating systems, including Raspbian, already have a driver for this device built in (`rtc_pcf8523`). In such cases there is no need for any programming if you are only interested in the current date and time, as the `hwclock` command will talk to the RTC and read or set the clock. An `rc` script run at boot time can be used to run this command to set the system clock automatically, and at power down the updated system time (which may have been adjusted either manually or over the

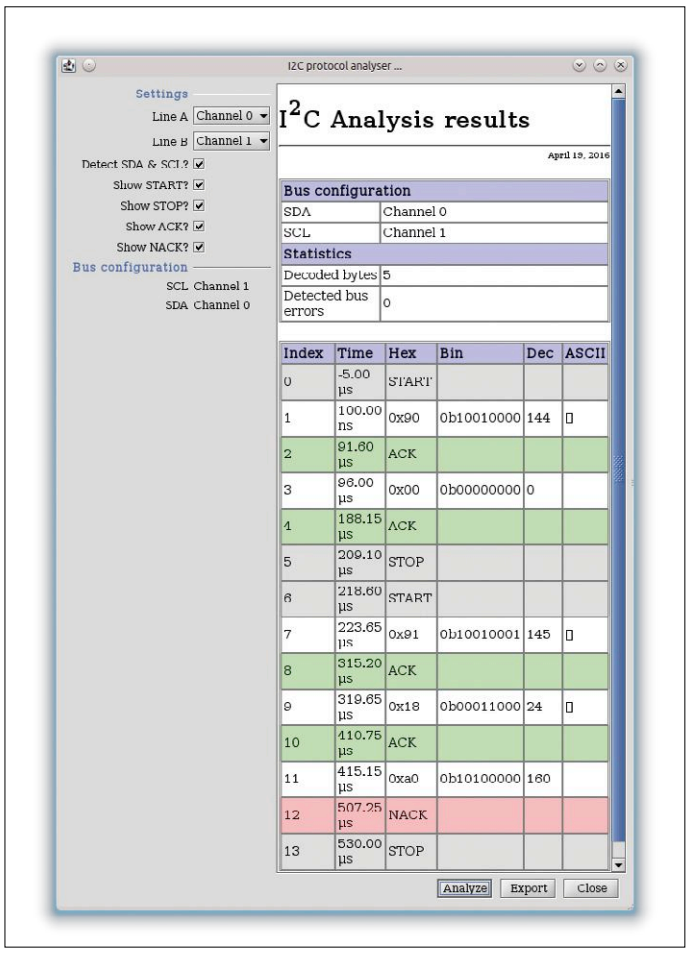


Figure 6. The OLS Java client presents the bus data in tabular form.

network) can be written back to the RTC. This arrangement allows a Raspberry Pi, even without a network connection, to maintain its clock across a power failure with minimal additional hardware. However, if you wish to use the alarm feature of the RTC, then you will need to get involved in some programming. Once the RTC has been set using a Raspberry Pi, it can then be connected to an ATmega or Arduino. The back-up battery on the module ensures that the clock continues to keep time. Then it is just a matter of a few lines of code to read the time into the ATmega or Arduino.

On the trail of the lonesome bug

Not everything works first time. A multimeter is a useful tool to start trying to track down a fault, but if the problem lies at the protocol level it is usually not good enough.

In the quiescent state SDA and SCL are high. So the first thing to check, with a normal **multimeter**, is that on power up (and ideally knowing that the bus is idle) the voltages on SCL and SDA are high enough. As we have mentioned previously, the standard specifies that a high level should be at least 0.7 V_{CC} (and so with V_{CC} = 5 V a minimum of 3.5 V). However, many 5 V devices will work perfectly well with pull-up resistors to 3.3 V: the high level in 3.3 V logic is usually well over 3 V. Sometimes the problem can simply be that one or both of the pull-up resistors are missing.

If the voltages are correct, then a more sophisticated approach is called for. **Logic analyzers** are available at prices to suit a wide range of pockets. Options range from the tiny ScanaQuad [2] to the more grown-up Red Pitaya [3]; my tool of choice is the Open Logic Sniffer [4] from Dangerous Prototypes, along with the OLS Java client from ols.lxtreme.nl [5]. All of these tools let you get to the bottom of what is going on on your I²C bus.

Figure 5 shows what happens when an ATmega88 reads the temperature register of an LM75. At the beginning (time 0.0 s) you can see the start condition, and at the end (500 µs) the stop condition. At 200 µs there is a repeated start condition. From the trigger point to around 200 µs the LM75 is being addressed in write mode and the value `0x00` is being written to its register pointer. After the repeated start condition the LM75 is being addressed again, this time in read mode. The two bytes `0x13` and `0x40` are read from the temperature register into the ATmega88. Here the trigger is obtained from PB0, which is being set to 1 at the start of the transaction and cleared back to 0 at the end of the transaction. Alternatively it would be possible to trigger on the falling edge of SDA and obtain similar traces. The OLS Java client analyses the traces in order to parse the I²C communication, and displays the bytes being transferred in tabular form (see **Figure 6**). The repeated start condition is erroneously displayed as a stop condition followed by a start condition, despite the fact that it is clear from the traces that there is no stop condition at 200.0 µs.

The Dangerous Prototypes **Bus Pirate** [6] shown in **Figure 7** is a tool for analyzing data transfer using serial protocols such as I²C, SPI and UART. Dangerous Prototypes sells the Bus Pirate and Open Logic Sniffer themselves, but also make the hardware design and software available so that you can build both devices yourself. According to the manufacturer, version 4 of the Bus Pirate is 'designed for the future', but does not work quite as reliably as version 3.6, which is available from distributors including Watterott Electronic [7].

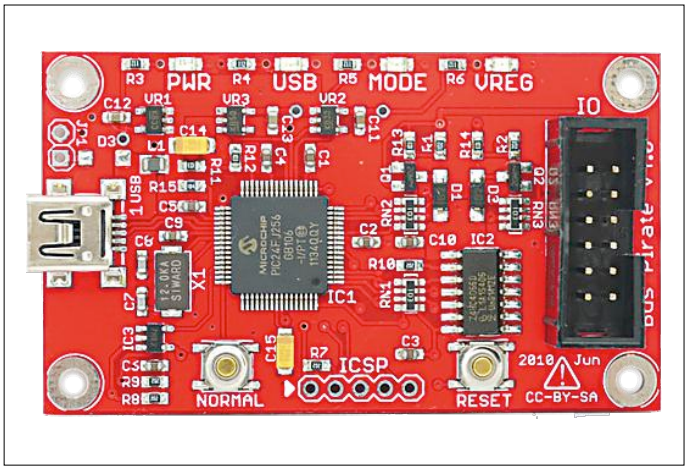


Figure 7. The tiny Bus Pirate is a universal bus interface for a PC. This is the new version 4. (Source: Dangerous Prototypes)

Table 1 illustrates I²C addressing and data transfer. As you can see, the LM75 acknowledges its address in both read and write transactions with an ACK. It also acknowledges the register number (`0x00`), as it is possible (despite the fact that the temperature register is read-only) that further bytes might follow. The ATmega88 acknowledges the first of the two bytes it reads from the temperature register with an ACK, as it is expecting further data; it responds to the second byte with a

Table 1. Addressing and data transfer using an LM75.

HiZ>**m** 1. HiZ 2. 1-WIRE 3. UART 4. I2C 5. SPI 6. 2WIRE 7. 3WIRE 8. LCD 9. DIO x. exit(without change)	Display menu...
(1)>**4** Set speed: 1. ~5KHz 2. ~50KHz 3. ~100KHz 4. ~400KHz	Choose I²C bus...
(1)>**3** I2C READY	Set bus speed to 100 kHz...
I2C>**(2)** Sniffer Any key to exit [0x90+0x00+][0x91+0x13+0x40-]	Start the bus sniffer

Bold type indicates user input;
'[' indicates a start condition;
numbers give the data bytes being transferred;
'+' indicates an ACK;
'–' indicates a NACK;
']' indicates a stop condition.

Table 2. Reading an LM75 sensor using the Bus Pirate as a bus master.

I2C>**[0x90** I2C START BIT WRITE: 0x90 ACK	Send start condition, address 0x48 and write bit (0): LM75 responds with ACK
I2C>**0x00** WRITE: 0x00 ACK	Send byte 0x00: LM75 responds with ACK
I2C>**]** I2C STOP BIT	Send stop condition
I2C>**[0x91 r:2** I2C START BIT WRITE: 0x91 ACK READ: 0x14 ACK 0x20	Send another start condition, followed by address 0x48 and read bit (1): LM75 acknowledges these with ACK. Then read two bytes: the Bus Pirate acknowledges the first received byte with ACK. On receiving the second byte it waits until it has determined that no further bytes are to be read (that is, when it is told to send the stop condition).
I2C>**]** NACK I2C STOP BIT	Send another stop condition. Since the previously received byte is now known to be the last of the transaction, first acknowledge it with a NACK.
The above sequence can be done in a single line as follows.	
I2C>**[0x90 0x00][0x91 r:2]** I2C START BIT WRITE: 0x90 ACK WRITE: 0x00 ACK I2C STOP BIT I2C START BIT WRITE: 0x91 ACK READ: 0x14 ACK 0xA0 NACK I2C STOP BIT	

NACK, as this is the last byte of the transfer. The ATmega88 then ends the bus transaction. The Bus Pirate is not able to determine which device has sent a NACK or ACK.
The Bus Pirate can also supply (10 kΩ) pull up resistors. In version 3 of the hardware pin 5 of the I/O header (VPU) must be connected to the desired voltage. This voltage passes through an analog switch that can be turned on to enable all the pull-up resistors on the Bus Pirate. This is unnecessary, and, if a 5 V supply is used, possibly even dangerous, when the device is used in conjunction with a Raspberry Pi. In version 4 of the Bus Pirate hardware the pull-up supply is configured in software instead. Whether or not pull-ups are enabled, and to what voltage they are connected, can be determined by simply making a measurement on pin 7 (SCL) or pin 8 (SDA) on the bus connector.
The pull-up resistors are turned on and off using the following commands.

```
I2C>P
Pull-up resistors ON
I2C>p
Pull-up resistors OFF
```

Once the bus is configured it is possible to use the Bus Pirate to search for connected slaves in a similar way to the `i2cdetect` command in Raspbian. Write and read addresses are displayed separately, as follows.

```
I2C>(1)

Searching 7bit I2C address space.

Found devices at:

0x90(0x48 W) 0x91(0x48 R)
```

This is a convenient alternative way to find the I²C address of an unknown slave if you do not have a Raspberry Pi to hand. The Bus Pirate can also be used as a Master and the individual phases of the I²C protocol can be stepped through manually. **Table 2** illustrates how to use this function to read temperature data from an LM75.

Wrapping up

The I²C bus provides a very simple way to connect peripherals to a processor, as long as the required data volumes are low and interrupts are not needed. You only need two wires! Moreover, it is a multi-drop bus, allowing multiple slaves to be connected using the same two wires. Data transfer over distances of up to a meter or so are feasible. Thanks to the availability of free libraries such as that written by Peter Fleury, the associated programming is not too arduous and many of the potential pitfalls are easily avoided.
The instruments required for tracking down problems are either readily available (such as a multimeter), or not too expensive. There is a world of sensors out there waiting to be connected to the microcontroller board on your bench! ⏮

(160373)

Web links

[1] For example: http://rn-wissen.de/wiki/index.php/I2C_Chip-%C3%9Cbersicht
[2] www.elektor.com/scanaquad-sq100
[3] www.elektor.com/stemlab-125-10-starter-kit
[4] http://dangerousprototypes.com/docs/Open_Bench_Logic_Sniffer
[5] https://lxtreme.nl/projects/ols/
[6] http://dangerousprototypes.com/docs/Bus_Pirate
[7] www.watterott.com/en/Bus-Pirate

Thank you!

I am grateful to my (former) hardware colleagues Franz Otte and Michael Kleineberg for hints and help on hardware matters, and to my (former) colleagues Reinhard Bernhardi-Grisson, Norbert Bandzius and Thomas Schlüssler for proof-reading the original German article and for their feedback.