

BEST-OF SERIES

RASPBERRY PI COMPILATION 2

THE MOST RELEVANT ARTICLES FROM ELEKTOR-MAGAZINE



155 pages for

€9.95

€7.50 for members



elektor
SELECT

LEARN > DESIGN > SHARE

DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN •
GN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN •
• SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DES
DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN
GN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN • DESIGN • SHARE • LEARN •

The Speaking Sonar Stick

**Raspberry Pi
+ ultrasonics
+ voice synth**

**= Aid
4 The Blind**

By **Bera Somnath** (India)

In good experimental fashion, this project is a valuable study into what can be achieved with 'commodity embedded technology' like a Raspberry Pi. This ultrasonic distance measuring "robot" will greet you, measure distances to objects continuously and read the values out loud. It can also be programmed to activate alarms when the measured distance crosses a user-programmed limit. Got a pair of U/S distance sensors and an RPi ready? Then go.

The actuation of the alarms is through GPIO activated relays which are energized depending on the 'danger level' set by the user or his/her personal aid. With some industrial design skills the project can be packaged to make a distance sensing walking stick for the blind and visually impaired for navigating their course. For instant tactile feedback to the blind user, the relay outputs may activate small vibration devices.

Why-the-Pi

For games Microsoft Windows is King, for networking Linux or Unix is King but for small scale automation where power consumption is a major factor, the Raspberry Pi microcomputer is one of the best options. Arduino can also do it to a certain extent but being a mere microprocessor, it is more suited to 'slave' functionality than 'master' like a Raspberry Pi, which by contrast can monitor, administer as well as execute functions at the same time, and at good speed.

For hardware we have...

... essentially, and easily identifiable in the schematic in **Figure 1**: a type HC-SR04 common-or-garden 4-pin ultrasonic distance sensor on K1, a Raspberry Pi Model A or B computer running the Raspbian Wheezy operating system, an optional HD 44780 compatible 2×16-character LCD panel, a handful of resistors, a Microchip MCP23008 I²C to parallel converter (IC1), and two FETs each driving a relay, Re1 and Re2 with their contacts bonded out to screw terminal blocks K2 and K3.

The circuit is powered with +5-V along with the RPi via its PWR IN USB micro connector. Most RPi connectivity with the circuit is through the GPIO pins, and all user control and interaction is through pushbutton S1.

Note that the FETs and the pushbutton are supplied from the +3.3-V rail furnished by the RPi.

For the sonar function, the RPi sends transmit pulses over the TRIG line (GPIO25) and 'listens' for a response on the ECHO line (GPIO8/SPIO_CEO). The

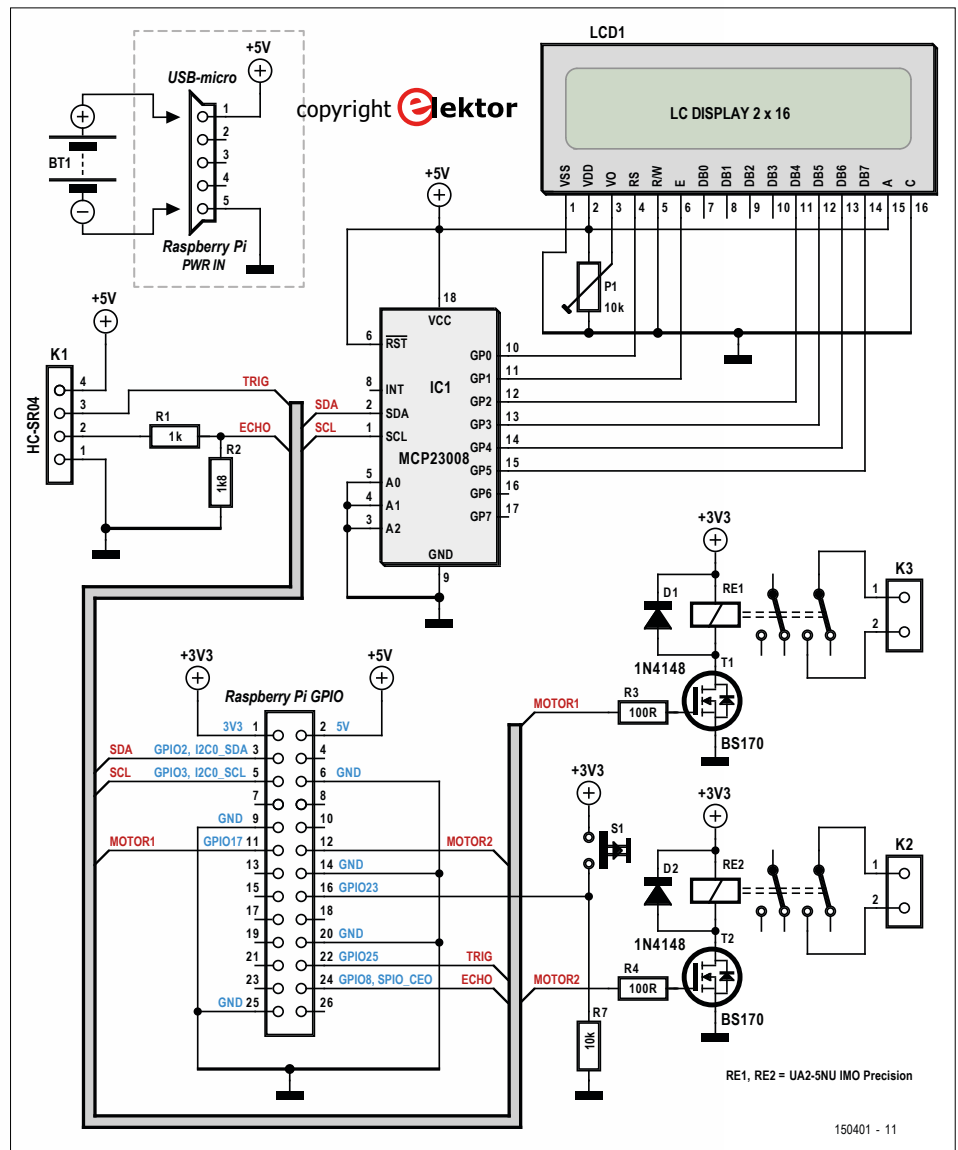


Figure 1. The Raspberry Pi computerette is so powerful and has so many GPIO lines it does not need many external components to be turned into a walking stick for the blind. The LCD shown here is optional, and intended for people aiding the person using the stick, or setting up the device, or just interested in seeing menus, values and stuff.

Features

- Ultrasonic-assisted walking cane for the blind and visually impaired
- Buzzer/vibrator warning outputs for object distance ≤ 1 m and > 2 m
- Ultrasonic transducer with 0.3 cm resolution
- Range values continuously read out loud, with s/w adjustable minimum-delta (typ. 2-5 cm)
- Raspberry Pi based
- LCD readout (optional)
- Freely downloadable RPi code

Component List

Resistors

R1 = 1k Ω
 R2 = 1.8k Ω
 R3,R4 = 100 Ω
 R7 = 10k Ω
 P1 = 10k Ω trimpot, horizontal

Semiconductors

IC1 = MCP23008-E/P, Microchip (Newark/Farnell # 1439387)
 D1, D2 = 1N4148
 T1,T2 = BS170

Miscellaneous

K1 = socket, 4-way, 90 degrees
 K2,K3 = 2-way PCB screw terminal block, 0.22" pitch
 S1 = tactile switch, 6x6 mm
 RE1,RE2 = relay, DPDT, 5V, IMO Precision Controls type UA2-SNU, (Newark/Farnell # 1094048)

Miscellaneous

LCD1 = LCD, alphanumeric, 2x16 characters (Elektor Store # 120061-74)
 Ultrasonic distance sensor (Elektor Store # 140194-91)
 ELPB-NG prototyping board (Elektor Store)

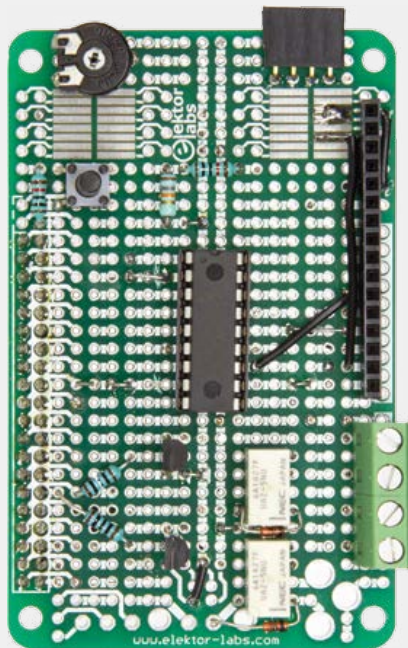


Figure 2. The project was built on the Elektor Labs ELPB-NG prototyping board.

time elapsing between a transmit and a receive burst of the 40-kHz (approx.) signal is used to compute the distance between the ultrasonic transducer device and the reflecting object, using sound velocity as a constant. In case another person comes in range with a similar ultrasonic sensor (or a car equipped with a parking sensor!) mutual interference may occur. The author is working on a solution to this problem. The type HC-SR04 ultrasonic ranging module has a detection range of 2 cm to 5 m approximately with an accuracy of about 0.3 cm and a 15-degree aperture angle.

A small loudspeaker is connected to the RPi's audio output, or earphones are used as an alternative. The LCD panel is optional.

Software

The software developed for the project is a free download you can find on the Elektor Magazine website [1]. We recommend opening and viewing the RPi code, some aspects of which will be discussed below. A piece of the code is shown in **Listing 1**, this is the ultrasonic echo routine.

The speech synth

To be able to adapt the project to your

language and/or preferred voice your need to install a voice synthesizer program. There are many voice synthesizer software available on Internet but out of these 'espeak' is the most versatile software, besides, it's free. Some other voice synthesizer "softwares" are Festival, Pico, and Cepstral [2].

eSpeak produces reasonable quality English speech. It uses a different synthesis method from other open source text-to-speech (TTS) engines, and has a unique sound. It's perhaps not as natural or "smooth", but in the author's opinion its articulation is clearer and easier to listen to for long periods.

```
$> sudo apt-get update
$> sudo apt-get upgrade
$> sudo apt-get install alsa-utils
```

Now edit the /etc/modules file to include the following line beside its other lines.

```
$> sudo nano /etc/modules
snd_bcm2835
```

Save nano (ctrl+o) and then exit (ctrl+x)

```
$> sudo reboot
```

The last command will reboot the Raspi. Now complete the installation by adding the following pieces of software.

```
$> sudo apt-get install mplayer
$> sudo apt-get install espeak
$> sudo apt-get install espeak-gui
```

To consult the espeak manual type:

```
$> man espeak or $> espeak -h
```

Before testing espeak, hook up an amplifier on the RPi audio socket as the normal audio output does not have enough power to drive more than an earphone. By running espeak-gui you can select from various "tonal" and volume options for the synthesizer's voice output. Here are a few examples:

```
$> espeak "Greetings, welcome to
Elektor" // male voice
$> espeak -ven+f3 "Hi, welcome to
Elektor" // female voice
$> espeak -ven+f3 -k9 -s150 -a200
"Fab, welcome to Elektor" // high
pitched, loud, female voice
```

The other bit of software needed for this project is rpi-gpio. Its present version can be checked at [3]; download the latest version and then install it, which is just a few clicks down the road.

Avoiding repetition

The Raspberry Pi will become long in the tooth when voicing the same values over and over again. That's why the software has an array where the last 20 distance values are stored.

When the current reading differs from the last one by more than the set value (here, we set 2 cm) RPi will speak up, else it will just remain quiet. Simply change the value of 'sv' at the beginning of the program to suit your requirement.

1-2-3, testing

Set the volume high if you connect the audio socket with an amplifier, else use earphones and then run the Python program in su mode, like so:

```
$> sudo python ultra3.py
```

The first initialization takes some time. You will hear your RPi's greeting message and then the program enters into a loop where it repeats the distance measu-

rement. GPIOs numbers 25 and 8 are connected to two 5-V relays to produce two signals on object detection — one at >200 cm (safe) and one at <100 cm (hazardous). To aid totally blind persons these can be connected with two small vibrating motors salvaged from discarded cellphones.

Finally, to put the program at the start insert the following line in the /etc/rc.local.

```
$> sudo nano /etc/rc.local
```

go to the end and before 'exit' add the following line

```
sudo python /home/pi/ultra4.py &
```

The program code for the Speaking Sonar Stick has many interesting aspects to it and Elektor readers are expressly invited to experiment with it, as well as extend, optimize or improve. And of course report back on your results through www.elektormagazine.com/labs, please.

Construction

To Elektor readers the construction of the electronic bits relevant to the Speaking Sonar Stick should be less challenging than the mechanical assembly let alone any attempt at "industrial/ergonomic design". However, some soldering must be done — you can't avoid that. Rather than designing a dedicated PCB, Elektor Labs used their "ELPB-NG" (that's Elektor Labs Prototyping Board New Generation) for the project. It's available from the Elektor Store. The assembled board is stacked onto the RPi computerette. **Figure 2** and the **Component List** basically tell the story as far as the electronics goes.

To be able to produce a proof of concept of the Sonar Walking Stick Elektor Labs used materials at hand. The result is shown in **Figure 3**, with the composing elements pictured separately in **Figure 3a** (Rpi, add-on board, LCD, no loud-speaker), **Figure 3b** (the battery pack) and **Figure 3c** (the U/S rangefinder). As further suggestions for progress towards a practical version, consider placing the vibrators in such a way the user is able to connect the location of the vibration to the distance measured, with the associated risk levels: <1 m (high risk) and >2 m (relatively low risk). In most

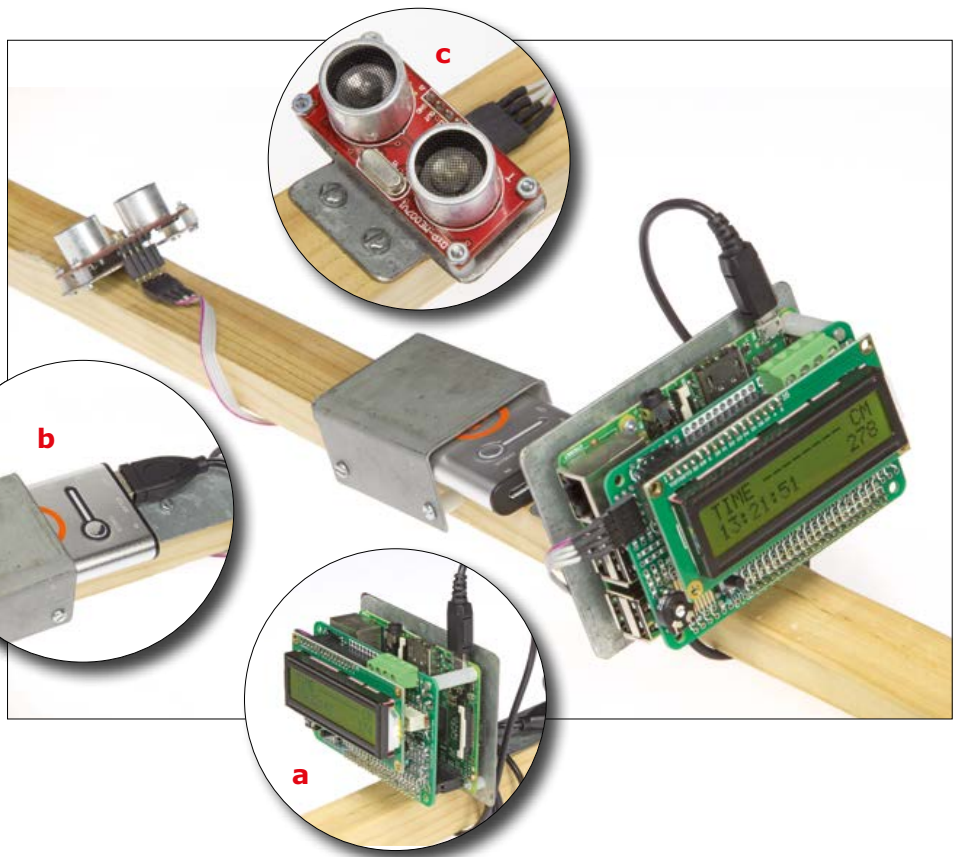


Figure 3. The Speaking Sonar Stick in an experimental construction on a wooden beam.

a. The RPi, the stacked-on extension board and the LCD form one unit for securing near the grip of the stick. **b.** Battery pack inserted in a metal holder secured roughly at the center of the stick. Here we used a Li-Ion powerpack with solar-assisted charging. **c.** Sonar transducer unit mounted at roughly 1/4th to 1/3rd stick height from the bottom. Its position is determined experimentally and should observe the user's wishes and physical abilities, as well as the opening angle of the U/S sensor.

How to make the script start automatically.

Raspberry Pi recipes, we love'm. Here's another one:

- Copy the 'rpi_distance_meter' folder to the home/pi/ directory on the Raspberry Pi, this can be done using a USB stick or using a program like winSCP.
- Open the terminal window and type 'cd rpi_distance_meter'.
- Type 'sudo chmod 755 launcher.sh'. This makes the launcher script an executable.
- Navigate back to the home directory with 'cd'.
- Type 'mkdir logs'. This creates a 'logs' folder.
- Type 'sudo crontab -e' This opens the crontab window. Crontab is a background process that lets you execute scripts at specific times.
- In this window, type '@reboot sh /home/pi/rpi_distance_meter/launcher.sh >/home/pi/logs/cronlog 2>&1'. This lets the script execute once upon startup.
- Reboot the RPi with 'sudo reboot' and see if it works. If it doesn't, see the log file in the 'logs' folder to see the errors you might have.

When the RPi boots the script starts automatically. It starts with a welcome message and then begins measuring and speaking the distance to the object in front of you. If the measurement value differs more than a preset amount from the previous value, the RPi speaks. To turn it off, press the button. The RPi will tell you that it's going to shut down.

Listing 1. RPi code for The Speaking Walking Stick (extract). Get the full code at [1]

```
#print "Ultrasonic Measurement"
GPIO.setup(GPIO_TRIGGER,GPIO.OUT)  # Trigger
GPIO.setup(GPIO_ECHO,GPIO.IN)      # Echo

while shutdown == 0:
    GPIO.output(GPIO_TRIGGER, False)
    time.sleep(0.1)
    # Send 10us pulse to trigger
    GPIO.output(GPIO_TRIGGER, True)
    time.sleep(0.00001)

    GPIO.output(GPIO_TRIGGER, False)
    start = time.time()
    while GPIO.input(GPIO_ECHO)==0:
        start = time.time()
    while GPIO.input(GPIO_ECHO)==1:
        stop = time.time()
    elapsed = stop-start
    #multiplied speed of sound (cm/s)
    distance = elapsed * 34300 / 2
    lcd.setCursor(0,0)
    lcd.message("TIME ----- CM")
    lcd.setCursor(0,1)
    buf = "%.0f" % distance
    if distance>99:
        lcd.message(datetime.now().strftime('%H:%M:%S')+buf)
    if distance<=99:
        if distance>9:
            lcd.message(datetime.now().strftime('%H:%M:%S')+buf)
        if distance<=9:
            lcd.message(datetime.now().strftime('%H:%M:%S')+buf)

    print " Distance : %.1f cm" % distance

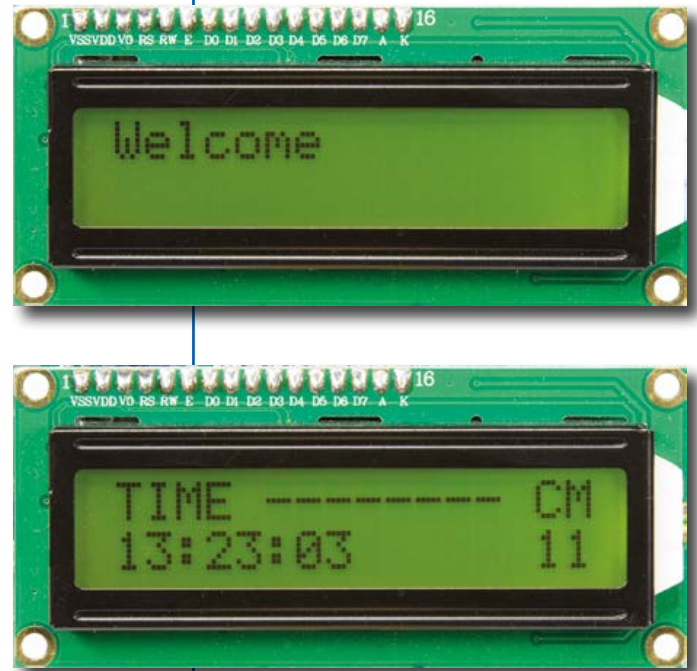
    if distance<=200 and distance>100:
        GPIO.output(motor1,1)
        GPIO.output(motor2,0)

    elif distance<=100:
        GPIO.output(motor1,0)
        GPIO.output(motor2,1)

    else:
        GPIO.output(motor1, 0)
        GPIO.output(motor2, 0)

    # if the difference between the previous distance and the current
    # distance is greater than a set value
    if abs(prevDistance-distance)>=sv:
        exitCode = subprocess.call(["espeak","-ven+f3","-a200",
            "-s120","%.0f" % distance])

    prevDistance = distance
```



cases the RPi and the ELBPB-NG board will be secured on the stick not far below the handle, and the battery pack (lead-acid, NiCd, Li-Ion) again a little lower. The sonar probe is at a position a little below the middle of the stick to securely identify low objects that can cause tripping.

The length of the stick is critical and should be adapted individually to the person using it. The earphones will produce the sonar readings only when there is a change in the reading exceeding the preset amount (setpoint, sv). Some earphones have built-in volume control. If not, the volume can be adjusted in the ultra3.py program by changing the -a220 value to something a little softer; note that -a200 is India Default i.e. full volume. ◀

(150401)

Web Link

- [1] Project software:
www.elektormagazine.com/150401
- [2] Cepstral voice synthesizer:
www.cepstral.com/raspberrypi
- [3] RPi-GPIO:
<https://sourceforge.net/projects/raspberrypi-gpio-python/>

Door Spy using Raspberry Pi

Communicate with a visitor using your smartphone

By **Luc Lemmens** (Elektor Labs)



An unexpected visitor can be a welcome surprise, but at other times you will have regretted answering that doorbell when and finding a less desirable guest on your doorstep. An intercom with a camera at the front door can be a solution, but that's of no use to you when you are not at home. With the project presented here you can, in principle, see from anywhere in the world who rang your doorbell, have a conversation with your visitor and even remotely open the door for them, provided you have your smartphone on you.

Using a Raspberry Pi (B+, 2 of 3, any of these will work), a Raspberry Pi camera and an RPi GSM HAT board designed by us, we will in this article build a doorbell that will take a picture of the visitor the instant they ring the doorbell. The picture is then sent in an MMS-message to your smartphone, so that you can see who is at your door. You can now call your door; the GSM module will answer automatically and after a bell signal you can converse with your visitor, using the built-in microphone and loudspeaker. There is even the possibility of sending an SMS message to the bell, that, with the correct password, can control an electric door opener. As an additional security message it also verifies the mobile phone number that sent the message to make sure that it was sent from your smartphone.

Although the system was devised, and is mainly intended, for those occasions when you are not at home, it can also be handy when — for whatever reason — you are not, or not easily able, to answer the front door. And, of course, you can also use it to decide whether you actually want to be at home for this particular visitor...

The project grew as a spin-off from the GSM breakout board [1], which uses an M95 GSM module from Quectel. This board already contained some features for this doorbell project, but the printed circuit board design was not really suitable to make it into a nice and tidy, compact module. Hence our new HAT board for the RPi, which also contains additional hardware to make the door spy complete. First let's take a closer look at the schematic.

Schematic

The greater part of this schematic has been taken directly from document 'M95 hardware design' from Quectel [2]. At the heart of the circuit (**Figure 1**) there is obviously the M95 GSM-module MOD1 with antenna and SIM-card holder SIM1. The resistors R15 through R17, the diode array D7 and the capacitors C19 through C22 serve as noise suppression and ESD protection for the SIM-card interface. The same is true for capacitors C6 through C17 and D3 through D6 for the microphone input and the audio amplifier output of the M95 respectively. The M95 is connected to the Raspberry Pi (K5) via its USART, all communications are done using AT commands. Voltage divider R6/R7 adapts the Tx signal to the 3.3 V rated digital input level of the RPi. It would have been the obvious

choice to operate the entire GSM board from a 3.3 V power supply voltage to prevent a difference in logic levels, but oddly enough we then run into a different problem. Although the power supply voltage of the M95 is specified to be in the range of 3.3 to 4.6 V, the module continually sends unsolicited warnings via the UART when the power supply

voltage approaches these limits within about 100 mV. This interferes with the decoding of those messages that we do want to receive from the M95, and furthermore, the M95 can then suddenly switch itself off. That is why the power supply level, using a MIC29302 (IC1), is set to 4.4 V, not a particularly common logic level. Nowhere in the documentation

does Quectel elucidate what the thinking behind this behavior is.

LED2 begins to flash as soon as the M95 is switched on. The label 'netlight' that Quectel uses to name this output would suggest to us that the module is then connected to a mobile network, but that is not the case. The M95 is in a standby mode Immediately after the power supply

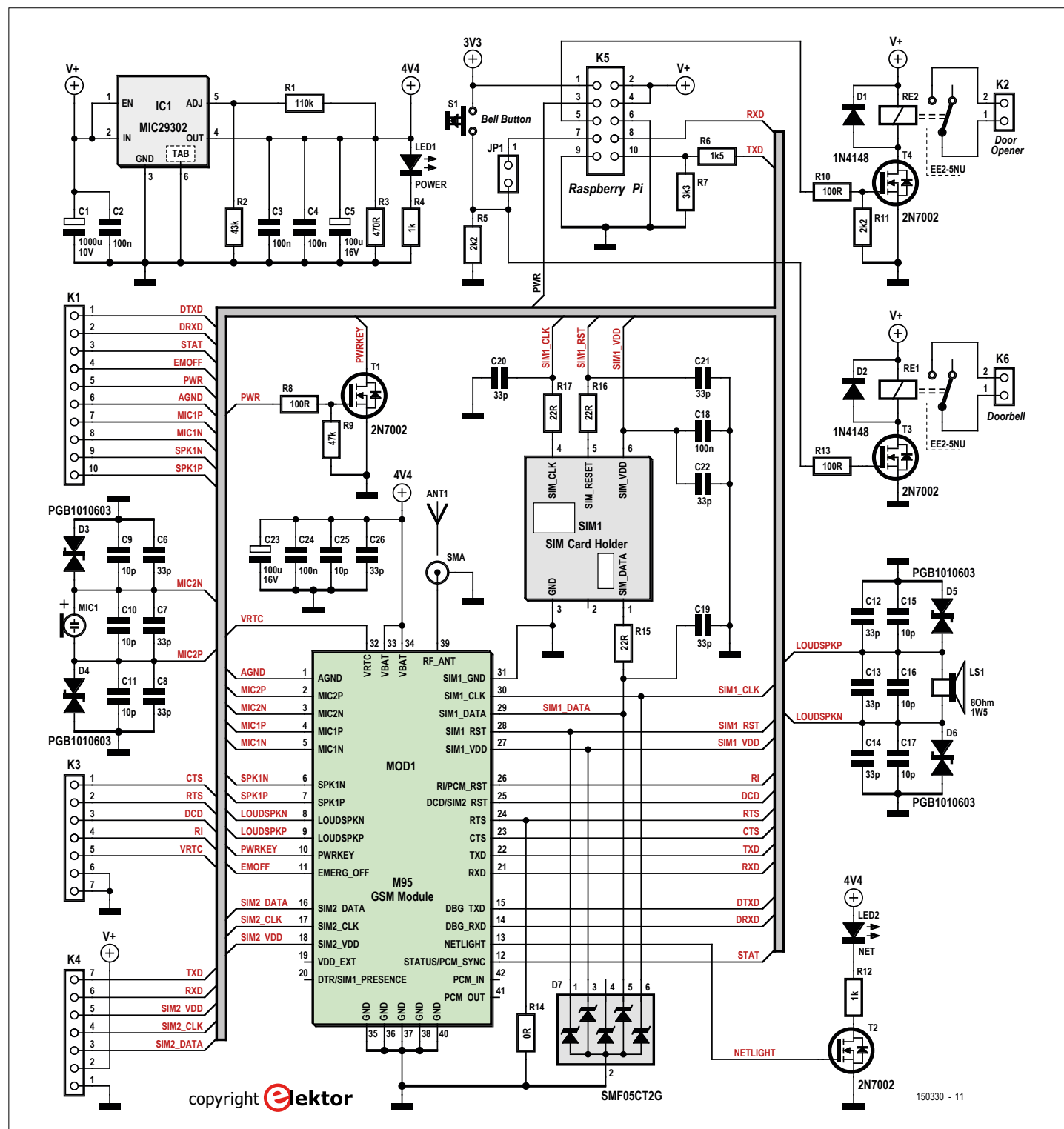


Figure 1. The major parts in this circuit are the M95 GSM module and the SIM card holder.

voltage is applied, a pulse on the 'pwr-key' line toggles the module on and off. Pushbutton S1 is the 'regular' doorbell pushbutton near the front door. In this case the button operates relay RE1, which can be used to activate the existing (AC) doorbell. JP1 (optionally can be replaced with a switch) determines whether or not the bell signal is sent to the RPi and the transmission of the picture commences. Sending an MMS message costs money, so when you are at home a normal doorbell is usually sufficient.

RE2 is intended for connecting an automatic door opener. This relay is switched

from the RPi via T4 when an SMS has been received that contains the correct password and has been sent from the correct mobile phone. To be frank, we ourselves are not all that fond of this feature. It requires knowledge in the area of (hardware) security to prevent someone from operating or bridging this relay and in this way gain access to your house. So using this feature is therefore entirely the responsibility of the user!

Printed circuit board

The circuit board that we have developed for the project can be seen in **Figure 2**.

Component List

Resistors

Default: SMD 0603, 1%, 0.1W)

R1 = 110kΩ
R2 = 43kΩ
R3 = 470Ω
R4,R12 = 1kΩ
R5,R11 = 2.2kΩ
R6 = 1.5kΩ
R7 = 3.3kΩ
R8,R10,R13 = 100Ω
R9 = 47kΩ
R14 = 0Ω
R15,R16,R17 = 22Ω

Capacitors

C1 = 1000µF 10V, radial SMD
C2,C3,C4,C18,C24 = 100nF, X7R, SMD 0805
C5,C23 = 100µF 16V, SMD 2312
C6,C7,C8,C12,C13,C14,C19,C20,C21,C22,C26 = 33pF, X7R, SMD 0805
C9,C10,C11,C15,C16,C17,C25 = 10pF, X7R, SMD 0805

Semiconductors

D1,D2 = 1N4148, SOD-323
D3,D4,D5,D6 = PGB1010603MR, SMD 0603
D7 = SMF05CT2G, SC-88
LED1,LED2 = LED, green, 50mcd, 2.1V @ 20mA, SMD 0805
T1,T2,T3,T4 = 2N7002, SOT-23
IC1 = MIC29302WU TR, TO-263

Miscellaneous

Mod1 = GSM module, Quectel type M95
SIM1 = microSIM socket (Molex 78723-1001)
ANT1 = SMA connector, PCB mount (Molex 73391-0070)
+ SMA antenna, right angled, (RF Solutions ANT-GHEL2R-SMA)
K1 = 10-way pinheader socket, 0.1" pitch (optional)
K2,K6 = 2-way pinheader socket, 0.1" pitch (optional)
K3,K4 = 7-way pinheader socket, 0.1" pitch (optional)
K5 = 10-way (2x5) pinheader socket with extended pins (Samtec ESQ-105-14-G-D)
S1 = 2-pin pinheader for doorbell connection (with make contact)
JP1 = 2-pin pinheader with jumper
LS1 = 2-pin pinheader for mini loudspeaker connection (8Ω, 1.5W)
Mic1 = 2-pin pinheader for electret microphone connection (ABM-715-RC)
Re1,Re2 = relay, SMT, DPDT, 2A, 5V_{DC} (Kemet EE2-5NU)
Raspberry Pi B+, 2 B or 3 B
Raspberry Pi Camera Board (V1.3)
PCB # 150400-1 from Elektor Store
or
PCB with SMDs preassembled: 150400-71 from Elektor Store

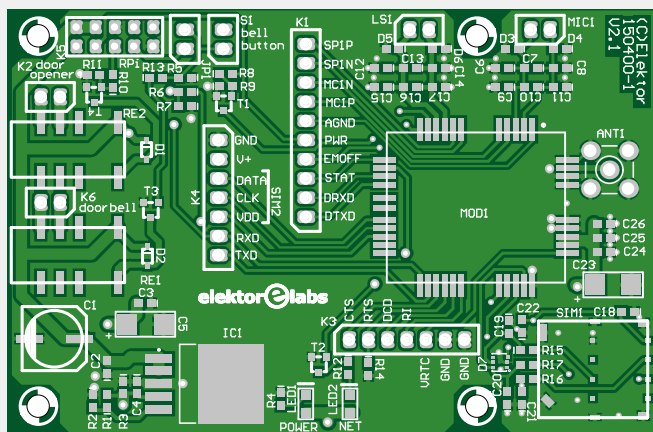


Figure 2. The printed circuit board for the door spy is designed as an RPi-HAT and contains mainly SMDs.

Since the GSM-module is only available in a SMD version, we decided to use SMD parts for all the components, with the exception of the antenna connection and the other connectors. This does, however, make the assembly more difficult for those who are less experienced in SMD soldering. But to make it easy for you, we also offer the PCB with all the SMDs soldered already (no. 150400-71). Real diehards can etch (or have etched) a circuit board themselves (layout can be downloaded from [4]) or a bare circuit board can be ordered from the Elektor Store and start from there. Note that various connectors are not absolutely necessary — this is mentioned in the component list. With K5 you need to insert the socket from the bottom of the circuit board and then solder it from the top side (see **Figure 3**). This connector is later plugged into the GPIO connector of the Raspberry Pi. Before soldering it in place, check whether the height of connector K5 needs to be adjusted so that the solder connections on the bottom stay far enough away from the USB connectors on the RPi.

Preparing the RPi

With the current version of Raspbian Jessie it is reasonably simple to make the Raspberry Pi suitable for our door spy. Download the most recent image from [3], at the time of this writing that is kernel version 4.4, May 2016. Use Win32 DiskImager (or an equivalent tool) to copy the .IMG to the SD card. Boot the RPi with this, open a terminal window and change the settings when starting up with the following command:

```
sudo raspi-config
```

First select option 6. Enable Camera to turn the RPi camera on. After this is done, go to 9. Advanced Options, from here choose A7 Serial and then select 'No' to disable access to the login shell via the serial port. After that select Finish and restart the RPi.

Note: in older versions of Raspbian the camera has to be switched on manually and the serial console has to be disabled in startup files, for additional help and further explanation refer to the numerous examples that can be found on the Internet.

This does then require a very important change to the Python script for the

door spy: on line 45 the '#'-sign in front of 'port.open' needs to be deleted! We have had a working prototype for some time, so when, for writing this article, we repeated the installation again 'from scratch', we did that with the most recent version of Raspbian and our script suddenly generated the error message 'Port already open' for this line 45. Apparently the new version makes the line for the 'open' command unnecessary.

Connecting the hardware

Now that we have the Raspberry Pi in order, we have to connect a few things before we can carry out our first test. It is easiest to start by connection the RPi camera; the flex cable can be routed along the bottom of our HAT board.

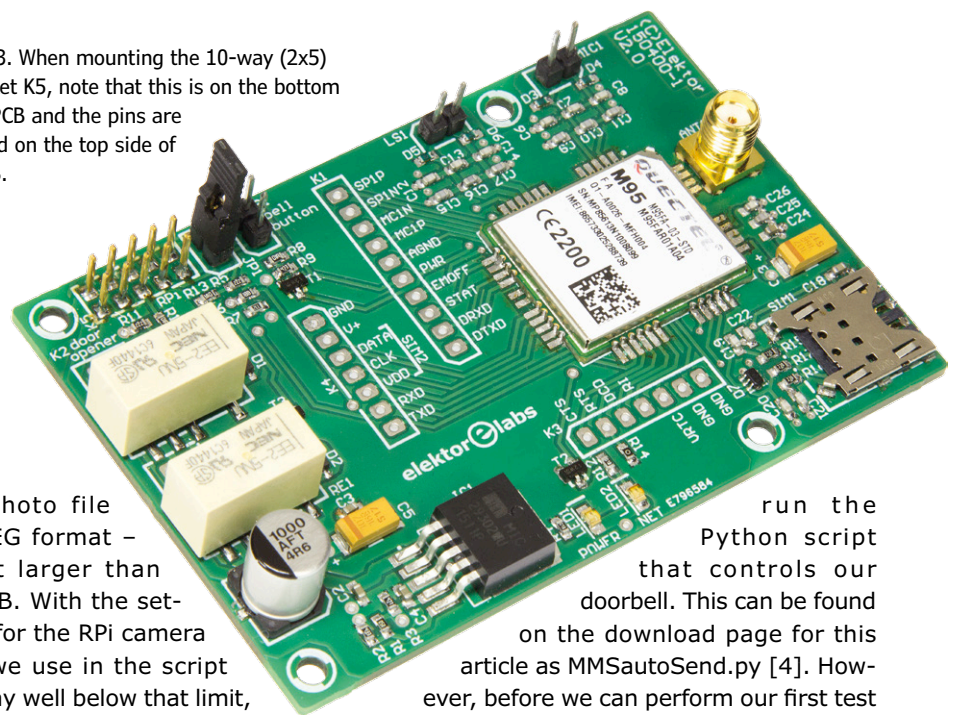
This board fits exactly on top of the RPi. Verify again that there is sufficient space between the underside of the board and the USB connections of the RPi. In particular look at the antenna connector ANT1 to make sure that it is far enough away from the metal shielding of the USB connectors. You can now assemble the boards neatly together at the correct separation using small bolts and standoffs (Figure 4).

Temporarily connect an arbitrary make pushbutton as 'doorbell' to the terminals of S1 and link jumper J1. The microphone and loudspeaker are attached to MIC1 and LS1 respectively. The antenna is connected to ANT1 and we now need a micro-SD card which will go into SIM1. But before you rush to the shop to purchase one of these first note the following...

SIM card and MMS

The M95 GSM-module in our application will only be able to do its job if there is a SIM card present with a sufficient prepay balance (or subscription, of course). Here in the lab we worked with two different prepaid SIM cards and a well-intentioned warning beforehand: perform careful comparative research before you buy a card. As mentioned earlier, the photo of the visitor is sent using MMS (Multimedia Messaging Service) and apparently this is used relatively little so that various providers deem it unnecessary to clearly specify the costs of this service. For as far as we could determine, most providers have a fixed tariff per MMS-message, provided the attachment – in our case

Figure 3. When mounting the 10-way (2x5) pinsocket K5, note that this is on the bottom of the PCB and the pins are soldered on the top side of the PCB.



the photo file in JPEG format – is not larger than 300 kB. With the settings for the RPi camera that we use in the script we stay well below that limit, also to reduce as much as possible the amount of time that is necessary to send and receive the JPEG. The latter is of course also very important: When the time between ringing the bell and reacting via the 'intercom' is too long there is a significant chance that the visitor has left already. For us in the lab it took some 20 to 30 seconds before the smartphone indicated that an MMS message was received. Add to that the time it takes for the recipient to act and you may therefore conclude that the system has little utility when the guests are impatient.

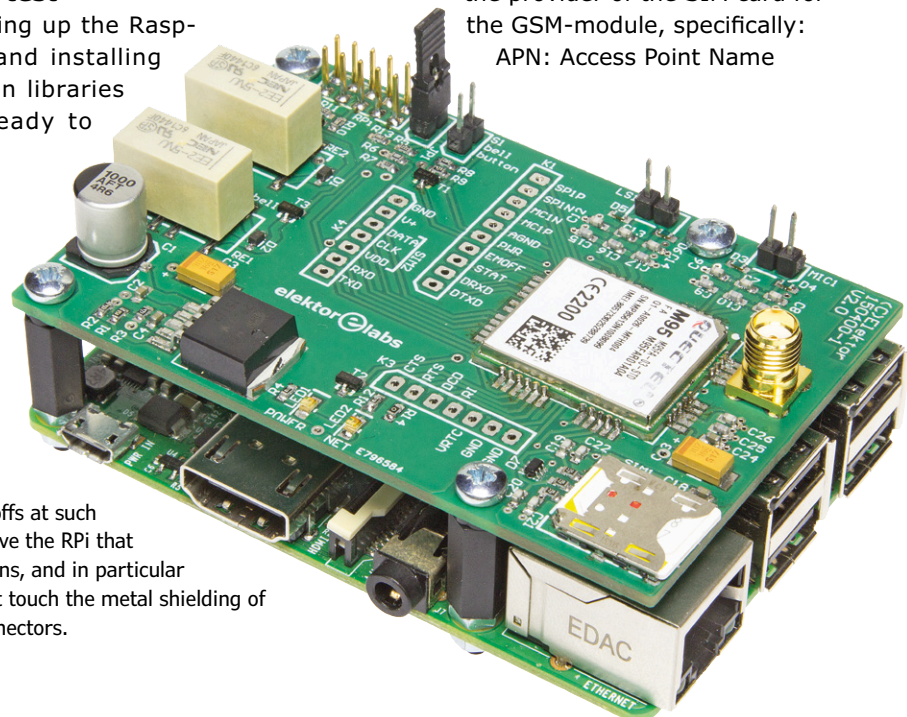
Python script for the doorbell and first test

After setting up the Raspberry Pi and installing the Python libraries we are ready to

run the Python script that controls our doorbell. This can be found on the download page for this article as MMSautoSend.py [4]. However, before we can perform our first test we first have to make some changes to this script. For this reason it will be convenient to connect a monitor, a keyboard and perhaps also a mouse while doing this testing. Once all the settings are correct the application will work without these peripherals. At the top of the script there is a block of constant definitions that needs to be set appropriately:

- ThisNumber: the mobile number of the SIM card for the M95
- ThatNumber: the number of the smartphone to which the MMS-message is to be sent
- Settings for using the MMS service of the provider of the SIM card for the GSM-module, specifically:
APN: Access Point Name

Figure 4. Mount the circuit board using standoffs at such a height above the RPi that the solder pins, and in particular ANT1, do not touch the metal shielding of the USB connectors.





You receive a snapshot of the visitor and you can then even talk to them

- MMSC: Multimedia Messaging Service Center
- MMSproxy: Multimedia Messaging Service proxy server
- MMSport: Multimedia Messaging Service port number

The correct settings can always be found on the website of the provider concerned. In the script are the MMS settings of the two Dutch providers that we used here in the lab.

After making the appropriate settings we are ready for the first test. Open the terminal window on the Raspberry Pi and start with the command:

```
sudo su
```

This is necessary to obtain the rights to be able to use the Python GPIO-library, or in other words, to be able to control the I/O-pins of the RPi.

The script can now be started using:

```
python MMSautoSend.py
```

The script itself is relatively straightforward. The RPi sends AT-commands to the GSM-module, where each time the M95 is given sufficient time to process the command with `time.sleep`. It begins with a range of settings, including parameters for the camera and volume settings for the microphone and loudspeaker, which incidentally, may have to be adjusted later, depending on the environment in which the doorbell will be installed.

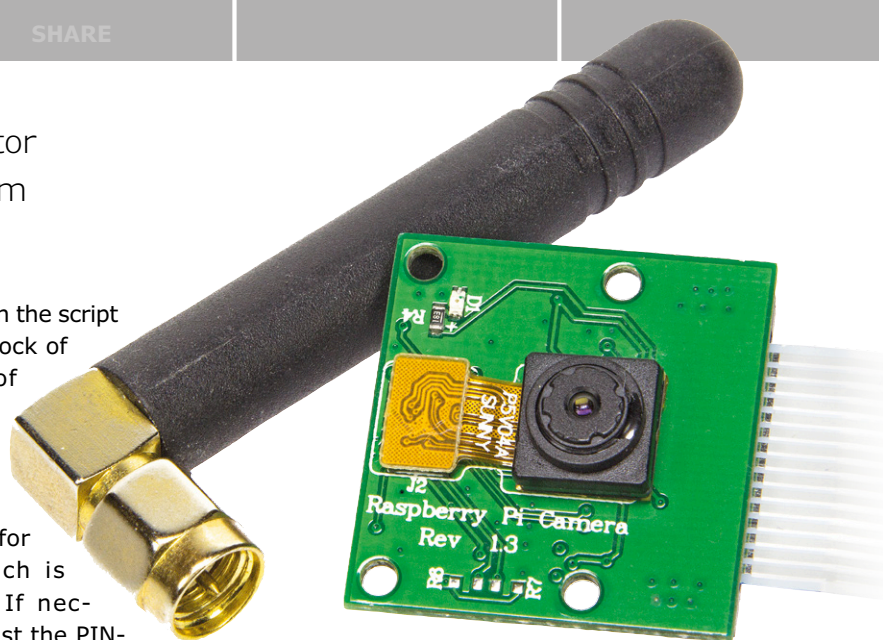
To keep things simple we have turned off the PIN-code protection of the SIM cards that we used. The easiest way to do this is to insert the SIM card temporarily into a mobile phone and then use the appropriate menu to disable the PIN-code

protection. In the script there is a block of four lines of comments in which the PIN-code can be entered for a card which is protected. If necessary, adjust the PIN-code according to the card that is used and remove the '#' at the beginning of the lines `port.write` and `time.sleep`.

After that the script ends up in an infinite loop, which can only be interrupted with CTRL+C to exit the script. But normally the RPi will continually check whether an SMS was received for operating the door opener OR checks whether the doorbell was operated or not.

In the event of an SMS, the script first checks where the message came from. If it came from the correct smartphone (ThatNumber) then it also checks that the contents of the message corresponds to the password that is set in the constant `Very_Secret` and if that is the case, relay RE2 will be activated for one second. Of course, this time can be changed in the script should that be necessary.

However, the receipt of an SMS message is usually preceded by another event: a visitor rings the doorbell. The Raspberry Pi then immediately takes a picture and prepares an MMS-message that will be sent to your smartphone. In addition to the photo, the message also contains a small amount of text, including the phone number of the doorbell, which will appear as a hyperlink on your smartphone. Depending on the person waiting



in front of your door, you can use this link to call the doorbell; the M95 will answer automatically, so that you can converse with the visitor. Once the conversation ends on the smartphone, the M95 will also 'hang up' and be ready for the next cycle of the bell ringing or the receipt of an SMS-message.

Building in

The mechanical workmanship is perhaps the biggest challenge for this project. The most ideal — and the safest — is to mount only the doorbell pushbutton, camera, microphone and speaker outside and keep the remainder of the electronics inside the house. In some cases you may be able to have the camera look outside through a window.

As already mentioned, in the script there are a few parameters to configure the camera, where specifically the orientation and brightness may have to be changed depending on the environment in which the camera will be installed. The resolution only affects the quality of the photo, keep this as low as possible so the size of the JPEG that is to be sent — and with that the time it takes to be transmitted — is within reasonable limits. While experimenting with these settings it will be best if you turn off the sending of MMS-messages in the script and only view the image file (`test1.jpg`) that is stored on the RPi to judge the quality. ◀

(150400)

Web Links

- [1] www.elektormagazine.com/labs/gsm-breakout-board-150330
- [2] www.quectel.com/UploadImage/Downlad/M95_Hardware_Design_V1.3.pdf
- [3] www.raspberrypi.org/downloads/raspbian/
- [4] www.elektormagazine.com/150400

Visitor Counter for Web Pages

With wireless LCD

By **Bert van Dam** (Netherlands)

In this article we will demonstrate how you can make a web page on a Raspberry Pi server, where the number of visitors is shown on the page itself. In addition, this information is also shown on a wireless Arduino with LCD.



Figure 1. Tablet and the XinoRF with LCD shield on top.

The hardware for this project comprises an XinoRF (an Arduino Uno with built-in radio module) and an LCD-shield. For the web server a Raspberry Pi is used with a 'Slice of Radio' module. The XinoRF and the Slice of Radio module are part of the 'Wireless Inventors Kit for Raspberry Pi' (abbreviated 'RasWIK'), which you can find in the Elektor STORE [1]. The Raspberry Pi has to be connected to your router using either a network cable or Wi-Fi and you need to know what the IP address of your Raspberry Pi is.

This project was tested with the SD card accompanying the Elektor book 'Raspberry Pi — Explore the RPi in 45 Electronics Projects' and also works with the SD card from the RasWIK set. Other SD cards don't have the necessary settings for radio traffic and/or Python web servers and it is preferable that you do not use those.

We assume in this article that you know how to operate the Raspberry Pi, how you can install software on it and how you issue commands. If this is not so, then it is perhaps a good idea to read the previously mentioned book or the new title 'The Internet of Things — In 35 projects with the Raspberry Pi and Arduino' first. The new title should be in our STORE as you read this.

Design

A simple Python internet server runs on the Raspberry Pi. The home page for this server (index.html) shows an image and then automatically starts a Python program on the Raspberry Pi. The result of this program is shown in the index.html page in an iframe (see **Listing 1**). This is a frame that is inside a web page. You can use it to show another web page from within a web page. In this project the second web page is generated by the Python program.

That the home page is able to run a program on the Raspberry Pi is because of the CGI, the Common Gateway Interface. As a security feature against unauthorized use, this only works on the Raspberry Pi with programs that are stored in a special directory with the name 'cgi-bin'. In addition, the owner of the Raspberry Pi has to give these programs execute rights beforehand. In this way unauthorized users are prevented from running any arbitrary program.

You may not be familiar with the HTML viewport tag at the head of the page (see **Listing 1**). This tag is used to scale the page so it fits properly on tablets and smartphones (see **Figure 1**). The image

on the page is 286 pixels wide, so we set the viewport slightly bigger, namely 300 pixels, so that we will have a small white border on the left and right of the image. You can also make the viewport exactly the same size, but then the user may wonder whether part of the page is missing because there is no white border to be seen.

The program started by the index.html page, visitor.py, first attempts to open a file that keeps track of the number of visitors. This file is stored on the Raspberry Pi server. If the file does not exist then the page has not yet been visited and the counter is initialized to 0. If the file exists then the value that is stored in it is retrieved. Subsequently this value is incremented by one and then stored back into the file. After this, the text "You are visitor number:" is added and sent via the wireless connection to the Arduino. Finally the program generates a web page containing this text and sends this back to the index.html page, where it is shown in an iframe. In **Listing 2** we only show the final part of the program, the complete source code is included with the download that accompanies this article [2].

You will see that the program 'prints' an HTML page. In reality, whatever a CGI-program prints does not go to the screen (or a printer), but is sent to the browser as a file. The browser will then show the file as a web page, in this case in the middle of the page index.html, at the position of the iframe.

The Arduino is running the program visitor.ino. This program runs continuously and is therefore not started by index.html. This program initializes the LCD

Listing 1.

```
<HTML>
<HEAD>
  <TITLE>Visitor</TITLE>
  <META NAME="viewport" content="width=300" CONTENT="initial-scale=1">
</HEAD>
<BODY>

<IMG SRC="welcome.jpg" WIDTH="286" HEIGHT="70"
ALIGN="BOTTOM" BORDER="0" NATURALSIZEFLAG="0">

<iframe name="myframe" src="cgi-bin/visitor.py" height="50"
  width="100%" frameborder="0"></iframe>

</BODY>
</HTML>
```

Listing 2.

```
# show the HTML page with the
  number of visits
print "Content-Type: text/html"
print ""
<HEAD>
<TITLE>Server Counter</TITLE>
</HEAD>
<html>
<body>
%s
</body>
</html>
"" % comment
```

and then waits for the data to arrive via the radio. This data is then displayed on the LCD and is neatly broken into multiple lines. When the Raspberry Pi sends the tilde character (~), the LCD screen is erased. This way we can ensure that the message will fit. The source code for this program is also in the download [2].

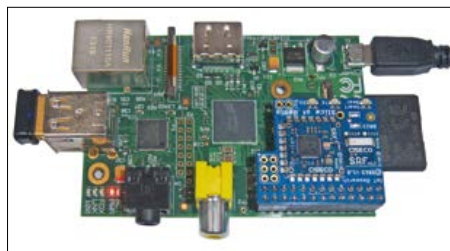


Figure 2. The Raspberry Pi with the Slice of Radio module.

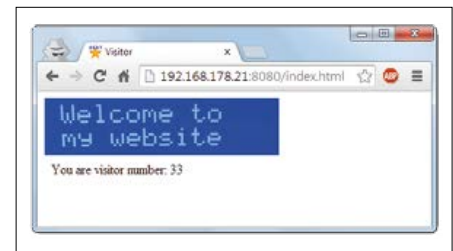


Figure 3. Web page index.html on a laptop.

Operation

Here follows a summary of the steps you need to follow:

1. Copy the program visitor.ino using the Arduino IDE from your PC to the XinoRF.
2. Make sure that the distance between the Arduino and the Raspberry Pi is at least 50 cm, because of the wireless connection. There is therefore no wired connection between the Arduino and Raspberry Pi!
3. Create a directory for this project on the Raspberry Pi and call this directory 'server'. Inside this directory create a sub-directory with the name 'cgi-bin'. Note: you *have* to use the name cgi-bin (without any capital letters and with the hyphen), otherwise you will not be able to run any programs remotely.
4. Copy into the server directory the following files (you can copy these from the download for this article): index.html, favicon.ico and welcome.jpg
5. Copy into the directory cgi-bin the following file: visitor.py. Note: do not open this file on your PC, not even just to read it. If you do this, then the file will be stored in a different way and it doesn't work anymore (you will get a message in the window in which the server is running that the file cannot be found, even though it is there).
6. Give the file execute permissions by executing the following instruction in the cgi-bin directory:
`chmod u+x visitor.py`
 If you forget to do this, you will still see the index.html page, but no counter value. An error message also appears in the window in which the server is running.
7. Go back to the sever directory and start the server using:
`python -m CGIHTTPServer 8080`
 This has to be done from the server directory. If you do not do that then you will not see the index.html page.
8. Now go to a web browser on your PC or tablet and enter the following address (don't forget to replace my IP address with that of your Raspberry Pi). If you do not know the IP address of your Raspberry Pi, then log into your router and go to the page that lists the connected devices. There you will see the Raspberry Pi, including its address.

<http://192.168.178.21:8080/index.html>

You will now see a page as is shown in **Figure 3**. Each time when you refresh the page the counter will increase by 1. What this looks like is shown in **Figure 1**.

You can find more projects like this in the newly published book 'IoT GET-U-GOING – In 35 projects with the Raspberry Pi and Arduino' by Maartje Jansee. ◀

(150551)

**Weblinks**

[1] www.elektor.nl/raswik

[2] www.elektormagazine.com/150551

Your own Media Player

For PC, Raspberry Pi, tablet and many more



Harry Baggen (Elektor Labs)

Every PC, laptop, smartphone and tablet these days comes complete with software preinstalled for playing photo, audio and video files. But this is usually not general-purpose or flexible enough; some formats are not supported, you will miss certain configuration settings or, for example, it cannot play files from across a network connection. What then, is the best alternative?

There is a whole bunch of media player software out there for all kinds of platforms, but when it comes to versatility and expansion capability one clearly stands out from all the others: Kodi, perhaps better known by its former name XBMC. This is an open-source media player which has been developed by the XBMC foundation, a group of enthusiastic programmers/users. The Kodi/XBMC was at one time (in 2004) developed as an independent media player for the first Xbox, hence the name Xbox Media Center (shortened to XBMC). Later on, versions were also developed for other operating systems, such as Windows, Linux, iOS and Android. Also various stand-alone versions have been made for smart-TVs, settop boxes and media players. Kodi now supports more than 75 languages.

One of the strong points of Kodi/XBMC is the option of adding so-called add-ons, which can be used to increase its functionality or to provide access to external media sources via the Internet. In addition to a large number of official add-ons there are also many 'gray' add-ons in circulation, which can be used to access all kinds of illegal audio and video sources. The XBMC foundation is increasingly

emphatic in distancing itself from these.

In this article I would like to say something about the many features that Kodi has to offer and on which devices you can use it. Many colleagues and acquaintances appear to be quite familiar with the name, but don't quite know how to get started with it themselves. It is actually quite easy and on YouTube there are by now hundreds of instructional videos to be found that can extend a helping hand.

Many versions

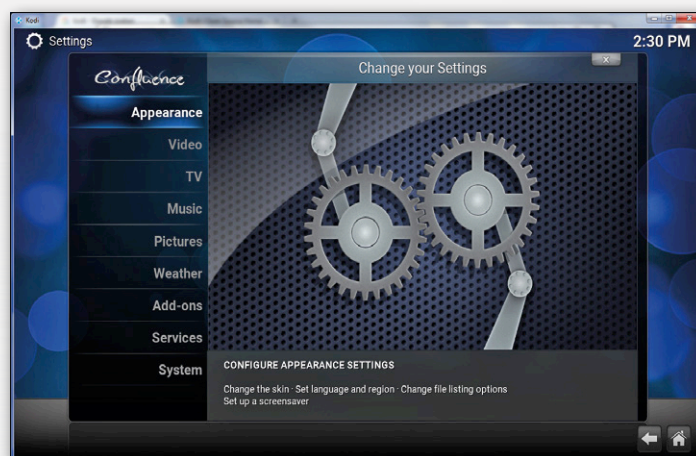
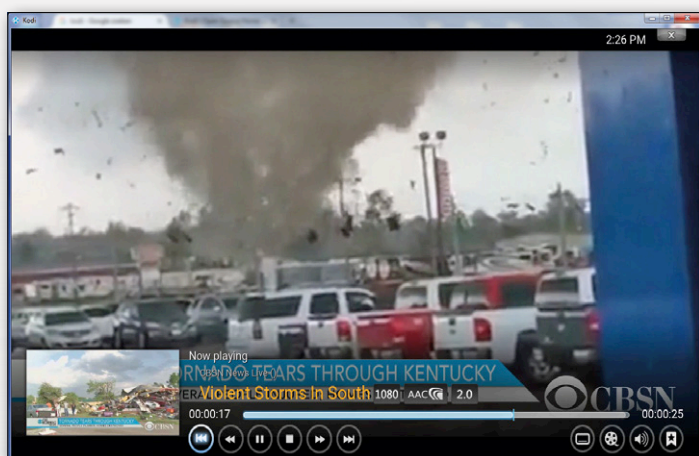
On the Download page of the official Kodi website [1] there are all the current versions available for Windows, Linux, Mac OS X, iOS, Android and a few specific hardware platforms such as Raspberry Pi and Amazon Fire TV. For most of these systems the user only needs to download the installation file and run that on their computer. For Android devices Kodi is also available from the Google Play Store. These days many more Android-based media-players are appearing on the market and Kodi is then quickly installed following this route.

An enormously popular hardware platform for building yourself a cheap media-player

is the Raspberry Pi. Probably a large number of all the RPi's that have been sold have been used for this purpose. Already starting from the very first version of the RPi special versions have been assembled for this. One of the most popular is OpenELEC, which stands for Open Embedded Linux Entertainment Center [2]. This is a so-called *Just Enough Operating System* (JeOS); you therefore do not need to install Linux or any other OS first, all you need is already built into the software. OpenELEC is not only available for the Raspberry Pi A, B, 2 and 3, but also for various other hardware platforms such as PCs with an X86-processor (ideal for a HTPC, no Windows required) and media boxes with a Freescale IMX6 CPU (Cubox). Although the installation of the OpenELEC software on the RPi is not hard, the procedure is a little different compared to the normal Kodi version. You just have to download a disk image (a literal copy of the memory contents) and this has to be copied to a USB-stick or (micro)SD card. This cannot be done by copying the image using Windows Explorer or similar. On a separate page [3] at OpenElec it is explained how you can do this from Linux, Windows and Mac OS X. Most Elek-



A large number of all the RPis that have been sold are used for Kodi



tor readers will be using Windows and in this case you need to use the program Win32Diskimager [4] to make a 1:1 copy of the image to the USB-stick or SD-card. After that, the stick or card only needs to be plugged into the RPi and the RPi can be turned on.

Usage

At first it takes a little bit of time to get used to the Kodi interface and the many configuration options. It is a good idea to first familiarize yourself with the user interface and configuration options using the Quick Start Guide in the Kodi wiki [5]. Under System/Settings there are a large number of tabs with configuration options, where the access level can also be switched between Basic, Standard, Advanced and Expert (start with Standard, only the most important settings are then displayed). You will not need to change most of the settings, but it can be nice to select under Appearance/International the desired language for the interface and under Videos/Subtitles the desired language for the subtitling. At System/Audio output you can select the desired audio output format. With an RPi this depends in whether its HDMI-output is connected to a surround receiver or directly to the TV.

In addition to playing videos and photos from your own media collection it is, of course, also interesting to active a num-

ber of add-ons which give access to a large number of Internet media sources, varying from Missed Broadcast to cooking and news programs from various countries. When you select Videos/Add-ons and subsequently Get more... (the same is true for Pictures and Music) there appears a list of available add-ons which are available in Kodi by default. You can then install the desired add-on, after which it appears in the Videos/Add-ons overview. This list can be extended with even more add-ons. An overview of all the official Kodi add-ons can be found at [6]. Here you will have to keep an eye on the Kodi version that you are using. Each version number of Kodi has its own name which is used to group the add-ons. The current version 16 is called 'Jarvis'. The installation of additional add-ons is

described in the Kodi wiki [7], but it is also recommended to search for this subject on YouTube.

Not all add-ons are approved by the Kodi developers, but there are nevertheless a large number of interesting unofficial add-ons in circulation (not to be confused with illegal). We name two sources which have collected a large number of add-ons and compiled those into handy zip files, so that the user has quite easy access to a large number of add-ons in one go. SuperRepo [8] has collected more than 2000 add-ons, while TVAddons [9] offers some 1200. There are many illegal ones among these, so take care which ones you install! ◀

(160031)

Web Links

- [1] <https://kodi.tv/>
- [2] <http://openelec.tv/>
- [3] http://wiki.openelec.tv/index.php/HOW-TO:Installing_OpenELEC/Creating_The_Install_Key#tab=DiskImage
- [4] <https://sourceforge.net/projects/win32diskimager/>
- [5] http://kodi.wiki/view/Quick_start_guide
- [6] http://kodi.wiki/view/Category:All_add-ons
- [7] <http://kodi.wiki/view/Add-ons>
- [8] <https://superrepo.org/>
- [9] <https://www.tvaddons.ag/>

Swiss Pi

A Swiss army knife for the Raspberry Pi

By **Ilse Joostens** and **Peter S'heeren** (Belgium)

The Raspberry Pi single board computer (SBC) was launched in the spring of 2012, and now — over four years later — it is still extremely popular. Not content to rest on their laurels, the Raspberry Pi Foundation has kept busy with upgrades and rolled out the third version last February. One thing that has certainly contributed to the popularity of the Raspberry Pi is that the 40-pin GPIO connector makes it very easy to extend the basic board with custom modules. We felt challenged.



However, the GPIO connector of the Raspberry Pi has some limitations compared to Arduino SBCs. For instance, the GPIO

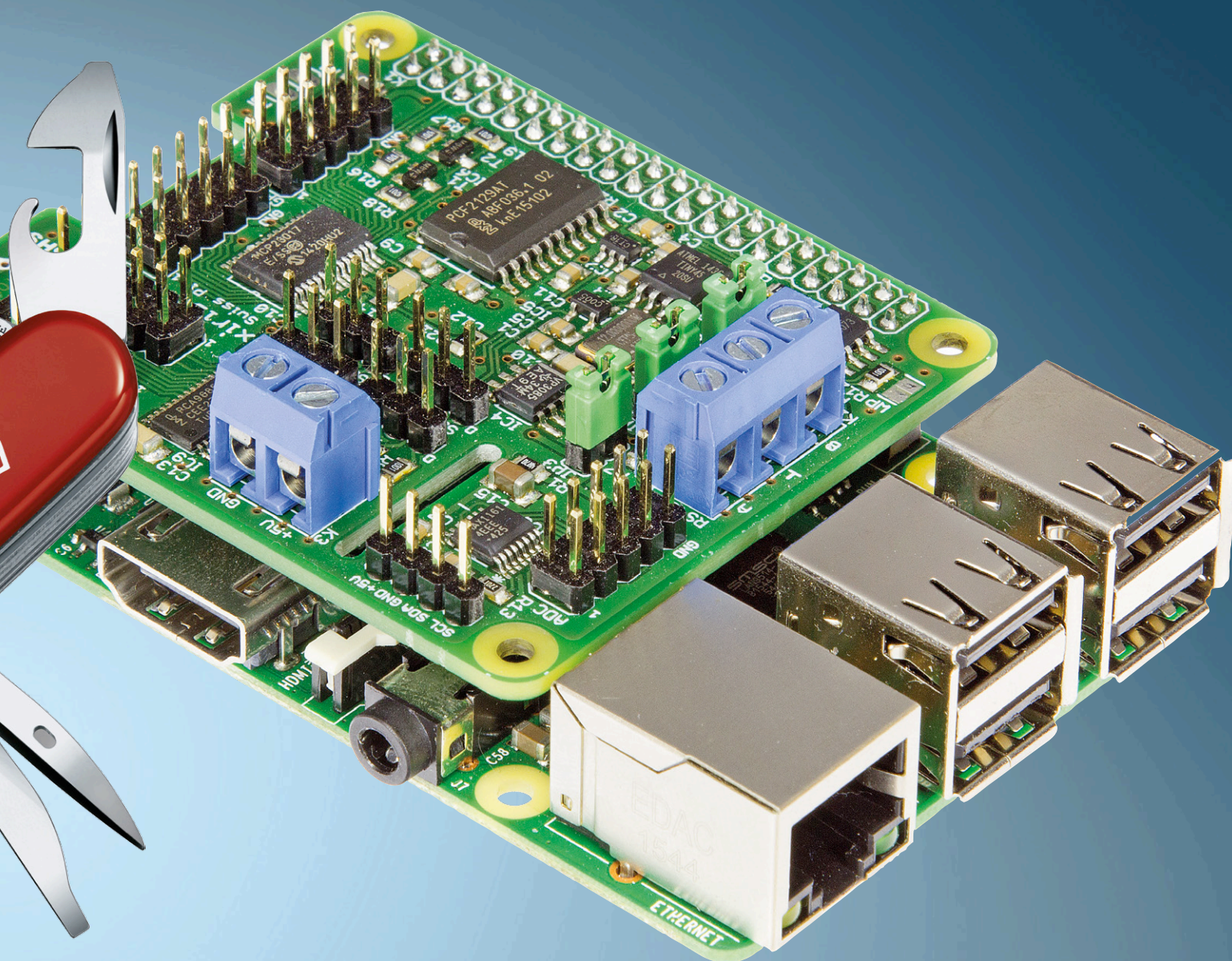
lines are connected directly to the CPU, which makes them fairly vulnerable, and there is hardly any hardware support

for generating PWM signals (only one PWM pin is available). There are also no analog (ADC) inputs. The Swiss Pi HAT described here eliminates these shortcomings and provides a number of extra features, including a real-time clock and an RS-485 interface.

The associated software, which can be downloaded for free from [1], runs under both Linux and Windows. A server program allows several clients to use different parts of Swiss Pi concurrently. The server can also be accessed using various scripting languages, including Python and PHP. Among other things, this allows the board to be controlled over the internet through a web interface. Finally, there is

Features

- 16 GPIO lines with configurable pull-up, 5-V signal levels
- 16 12-bit PWM channels, configurable 24–1526 Hz, 5-V signal levels
- 4 connectors for remote control servos (shared with PWM channels 0 to 3) and supply voltage connector
- 8 12-bit ADC inputs, range 0–4.096 V
- Half-duplex RS-485 interface compliant with TIA/EIA 485A standard, with ESD protection
- Precise real time clock with battery backup, 3 ppm accuracy
- I²C extension connector, 5-V signal levels
- ID EEPROM



a graphical user interface program that makes it easy to operate all the functions on the board interactively, which is very handy for testing.

Hardware

As befits a Swiss army knife, the Swiss Pi provides a broad assortment of handy tools, as can be seen from the schematic diagram in **Figure 1**. They are described individually below.

I/O expander

We chose the I²C MCP23017 I/O expander to control the 16 GPIO lines. The I/O lines can be configured as high-impedance inputs with pull-up resistors or as

outputs. This IC is often used together with the Raspberry Pi, so a lot of example programs and tutorials on this topic are available on the Internet.

PWM controller

Here we use the PCA9685, an I²C 16-channel LED controller, to generate the PWM signals. With a configurable PWM frequency range of 24 to 1526 Hz and 12-bit resolution, this IC can be used to drive LEDs (including RGB LEDs) as well as motors (with MOSFET bridge power stages) or remote control servos. For the latter purpose, PWM channels 0 to 3 are fed out to four three-pin connectors for direct connection to remote control ser-

vos. Due to the relatively high power consumption of remote control servos, they are not powered directly from the Raspberry Pi. An external power source must be connected to K3 for this purpose. If you want to use more than four servos, you will have to provide the additional connectors yourself.

In combination with an RC filter circuit as shown in **Figure 2**, the PCA9685 can also be used as a DAC, for example to generate an 0–10 V control signal for light dimmers.

ADC

The MAX11614EEE+ analog to digital converter IC used here has eight chan-

nels with 12-bit resolution and an internal 4.096 V voltage reference. The channels can be used in single-ended or differential mode, although the software included

with the Swiss Pi only uses the channels in single-ended mode. The dimensions of Raspberry Pi extension cards are standardized by the HAT specification, which

means that space on the board is limited. As a result, there are no additional protection diodes provided for the ADC inputs. If you use sensors that are powered from

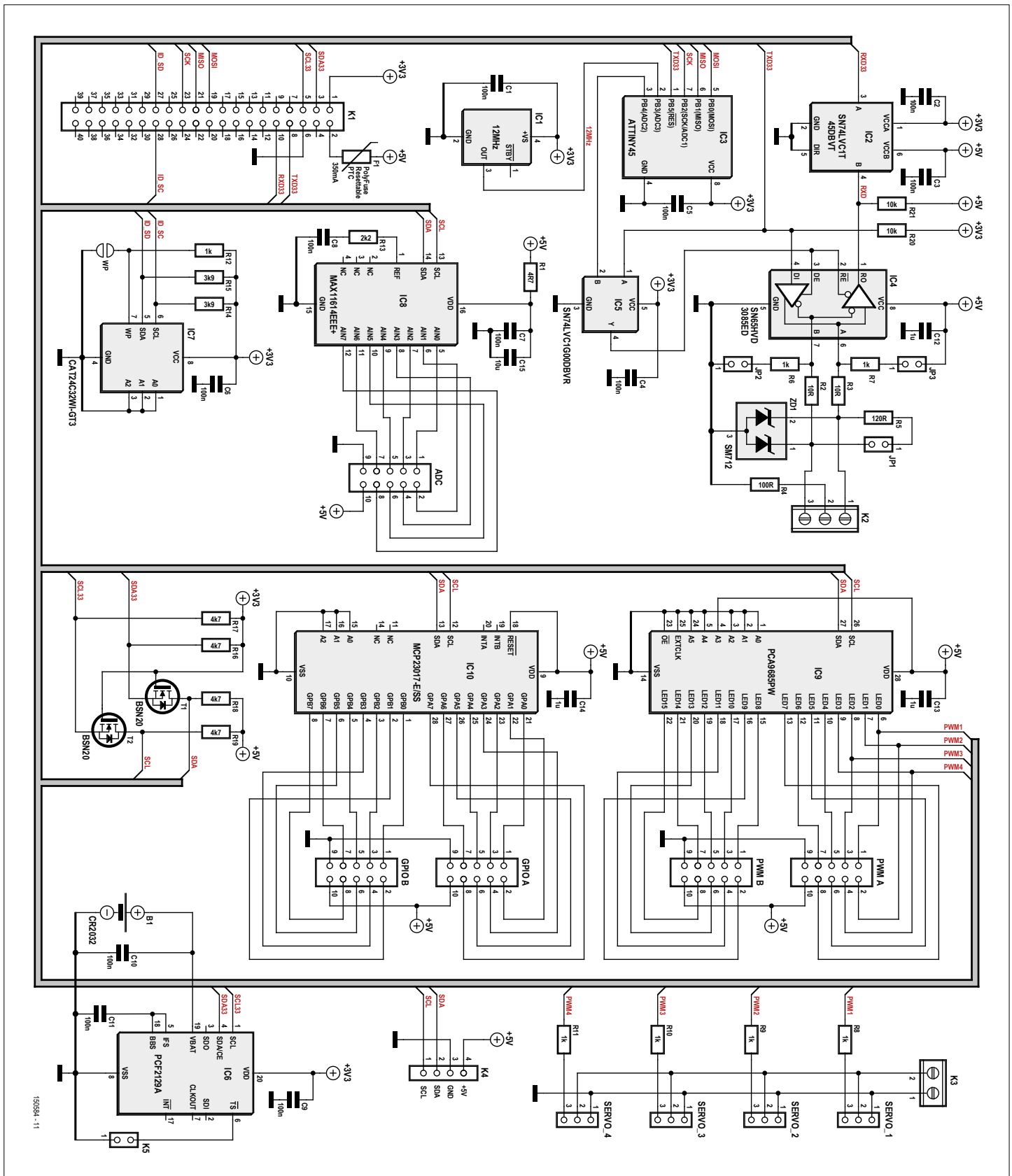


Figure 1. Schematic diagram of the Swiss Pi HAT.

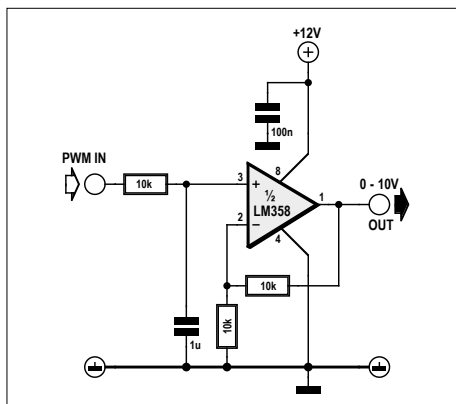


Figure 2. Converting a PWM signal into a clean analog voltage.

an external source or you use relatively long connecting cables, it is advisable to provide two additional Schottky diodes and a resistor for extra protection (see **Figure 3**).

RS-485

The RS-485 interface on the Swiss Pi enables communication with numerous Modbus devices, which opens up a world of potential applications.

We connected the RS-485 interface to the standard UART of the Raspberry Pi, rather than using a relatively expensive SPI to RS-485 converter IC. This has the additional advantage that you do not need a special driver. However, it has the disadvantage that it is not easy to switch the RS-485 transceiver (IC4) from transmit mode to receive mode with the RE/DE control line after transmitting a data word. The processor of the Raspberry Pi does not provide any hardware support for this, and using software with a GPIO pin for this control function is not particularly reliable because Linux is not a real-time operating system.

For that reason, we use a small microcontroller (an ATTiny45, IC3) and a NAND gate (IC5) to handle mode switching for the RS-485 transceiver. When the serial interface is quiescent and IC3 is inactive, both inputs of the NAND gate are high and the output is low, putting the RS-485 transceiver in receive mode. When the Raspberry Pi starts to send data, input A of the NAND gate is pulled low by the

falling edge of the TX signal (start bit). This causes the output to go high, putting the RS-485 transceiver directly in transmit mode. The TX signal is also applied to pin 1 of the ATTiny45, which pulls input B of the NAND gate low when it detects a falling edge on this input. As a result, the RS-485 transceiver remains in transmit mode even after the TX signal goes high again. After a configurable time interval the ATTiny45 sets input B of the

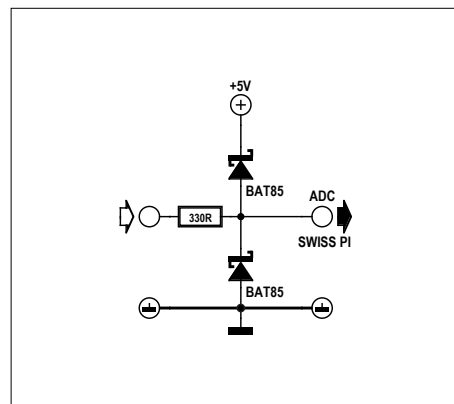
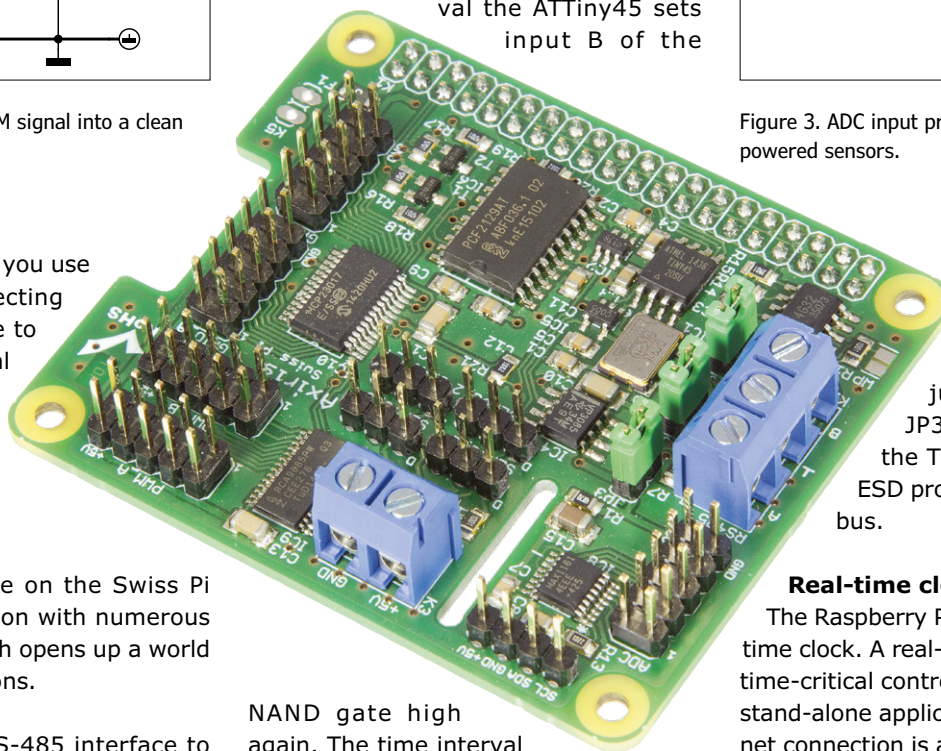


Figure 3. ADC input protection for externally powered sensors.



resistors and termination resistor for the RS-485 bus can be configured with jumpers JP1, JP2 and JP3. Finally, R2, R3 and the TVS diode ZD1 provide ESD protection for the RS-485 bus.

Real-time clock

The Raspberry Pi does not have a real-time clock. A real-time clock is handy for time-critical control tasks, particularly in stand-alone applications where no Internet connection is available and therefore access to an NTP server is not possible. Here we opted for the PCF2129A, which has a built-in crystal. This IC is inexpensive and very precise, with internal temperature compensation and an accuracy of 3 ppm. A CR2032 button cell serves as a backup battery with a lifetime of about ten years.

Up to two time stamps can be generated via K5. For more information about this, see the PCF2129A data sheet

I²C level converter, polyfuse and ID EEPROM

With the exception of the real-time clock, all I²C devices are powered from +5 V. Two n-channel MOSFETs (T1 and T2) in combination with resistors R16–R19 are used to provide the necessary level conversion for the SDA and SCL lines. Polyfuse F1 limits the total current consumption of the circuit to 350 mA. This ensures that the Raspberry Pi still has enough current to work properly, even with a power source with relatively low capacity. That must be taken into account

NAND gate high again. The time interval must be chosen to ensure that this occurs just before the end of the last stop bit, after which the RS-485 transceiver goes back into receive mode.

The delay time of the ATTiny45 can be configured via the SPI interface of the Raspberry Pi and depends on the baud rate, the word length, the parity bits and the number of stop bits. We could not use I²C for this because the ATTiny45 has a limited universal serial interface (USI), which means that most of the I²C protocol has to be implemented in software. That is naturally not as fast as hardware, so clock stretching would be necessary to allow the ATTiny45 to track the I²C bus. A hardware error in the I²C Master of the Raspberry Pi prevents it from supporting this clock stretching function properly, so our only option was to use SPI instead.

Here the RS-485 transceiver is powered from +5 V, with IC2 providing the required level conversion to 3.3 V for the RX line to the Raspberry Pi. The bias

when you connect extensions to the GPIO pins and/or PWM channels.

Finally, there is an ID EEPROM in the form of IC7. It is necessary to conform to the requirements of the Raspberry Pi HAT specification. The idea is that the Linux kernel reads the ID EEPROM during

boot-up and then, based on this information, configures the GPIO lines of the Raspberry Pi correctly for the HAT concerned (auto configuration, device tree). As far as we know, support for this on the Raspberry Pi is still fairly limited at present.

PCB

We designed a compact PCB for the Swiss Pi. See **Figure 4** for the component side and **Figure 5** for the solder side.

Software

Swiss Server

The Swiss Server software provides the functions of the Swiss Pi module via the network ports and standard I/O according to its own communication protocol. The server can accept multiple client connections, allowing several clients to work with the Swiss Pi concurrently. This facilitates wide-ranging flexibility.

The communication protocol consists of commands and corresponding responses. A client sends a command, and the server returns a response once the command has been executed. The communication protocol is coded in ASCII format. The syntax is designed to make the communication protocol easy to use both manually and in software.

The server supports a large number of commands for working with the Swiss Pi module. The communication protocol is fully asynchronous, which means that a client does not have to wait for the response to a command before sending the next command. This allows clients to send several commands in direct succession, which dramatically increases communication efficiency. Each command can optionally be provided with an identification number, which is sent back with the response.

The server also has an option for exchanging data bytes directly between the RS-485 interface and a network port. If this option is disabled, clients can use commands to send data bytes over the RS-485 interface.

The Swiss Server works with both Linux and Windows. In the latter case the Swiss Pi module has to be connected to the computer through a USB port and an adapter board (AxiCat). More information about this will be provided in a future article on AxiCat.

RS-485 Control

This program talks with the ATtiny45 in order to configure the RS-485 functions using a registry file that can be accessed over the SPI interface. If you use the Swiss Server, you will probably

Component List

Resistors

Default: SMD 0805

R1 = 4.7Ω

R2,R3 = 10Ω high pulse loading

R4 = 100Ω ≥250mW

R5 = 120Ω ≥250mW

R6-R12 = 1kΩ

R13 = 2.2kΩ

R14,R15 = 3.9kΩ

R16-R19 = 4.7kΩ

R20,R21 = 10kΩ

Condensatoren

C1-C11 = 100nF, 0805

C12,C13,C14 = 1μF, 1206

C15 = 10μF, 1206

Semiconductors

T1, T2 = BSN20

IC1 = 12MHz oscillator block, SMD, type LFSPX0018037

IC2 = SN74LVC1T45DBVR

IC3 = ATtiny45-20SU, programmed

IC4 = SN65HVD3085EDG4

IC5 = SN74LVC1G00DBVR

IC6 = PCF2129AT

IC7 = BR24G32FJ-3GTE2, 32 Kbit I²C EEPROM

IC8 = MAX11614EEE+

IC9 = PCA9685PW,112

IC10 = MCP23017-E/SS

ZD1 = CDSOT23-SM712, 12V Transient Voltage Suppressor

Miscellaneous

F1 = 16V 0.35A Polyswitch type 1206L035/16YR

B1 = 2032 button cell holder

K1 = 40-way (2x20) socket, 0.1" pitch

K2 = 3-way PCB screw terminal block, RS-485-bus, 5mm pitch

K3 = 2-way PCB screw terminal block, 5mm pitch

K4 = 4-pin pinheader, 0.1" pitch

K5 = 2-pin pinheader, 0.1" pitch

ADC = 10-pin (2x5) pinheader, 0.1" pitch

PWM_A, PWM_B = 10-pin (2x5) pinheader, 0.1" pitch

GPIO_A, GPO_B = 10-pin (2x5) pinheader, 0.1" pitch

SERVO_1 - SERVO_4 = 3-pin pinheader, 0.1" pitch

JP1-JP3 = 2-pin pinheader, 0.1" pitch, with jumper

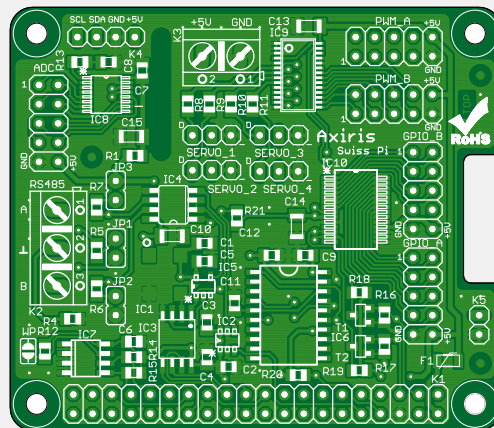


Figure 4. The component side of the compact PCB, which fits neatly on the Raspberry Pi board.

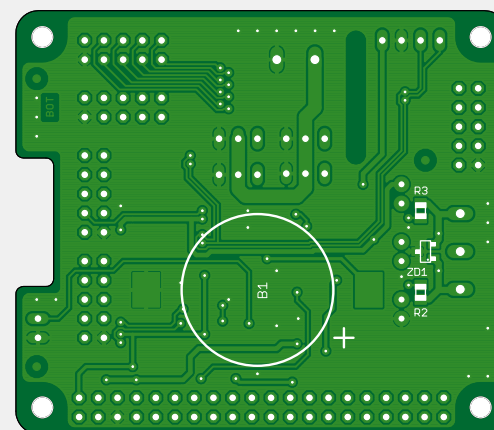
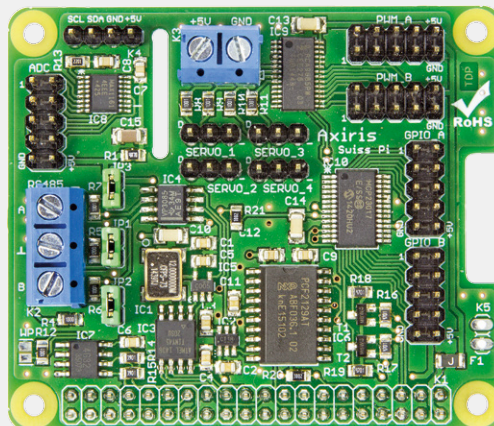


Figure 5. The solder side of the PCB.

not need this program very often because the Swiss Server provides the most frequently used RS-485 functions.

The program includes a detection function to determine the time parameters for transmit mode. This detection procedure can be executed on a naked Swiss Pi board; no specific hardware setup is required.

I/O Card Explorer

This program provides a graphical user interface to make working with the I²C devices on the Swiss Pi board easy and interactive, as shown in **Figure 7**. The program can communicate with the hardware in various ways. If the program is running on the Raspberry Pi, it can control the Swiss Pi directly. In that case the program has exclusive access to the Swiss Pi.

If the Swiss Pi is operating under control of the Swiss Server, you can configure the program to communicate with the hardware through the Swiss Server.

With the Swiss Server you can also run the program on a different computer and connect to the Swiss Server over a network. Different instances of the program and other clients can connect to the Swiss Server concurrently, allowing several parties to work with the Swiss Pi module at the same time.

Extensions

To make the Swiss Pi module more useful in the real world, it is desirable to provide a number of extensions in the future. Among other things, we are thinking about a relay/SSR card, a board with galvanically isolated digital inputs, a 4–20 mA current loop adapter, a 0–10 V controller for dimmers, a DC motor controller with bridge output, a RGB driver board for LED strips, and so on. The Swiss Pi can also be extended through the I²C connector with our eight-channel 1-wire mini host, enabling the use of 1-wire devices such as DS18B20 temperature sensors.

Finally, there is also a version of the Swiss Pi available for BeagleBone, dubbed “Swiss Cape”. In terms of hardware it is comparable to the Swiss Pi. The Swiss Cape uses the same software as the Swiss Pi.

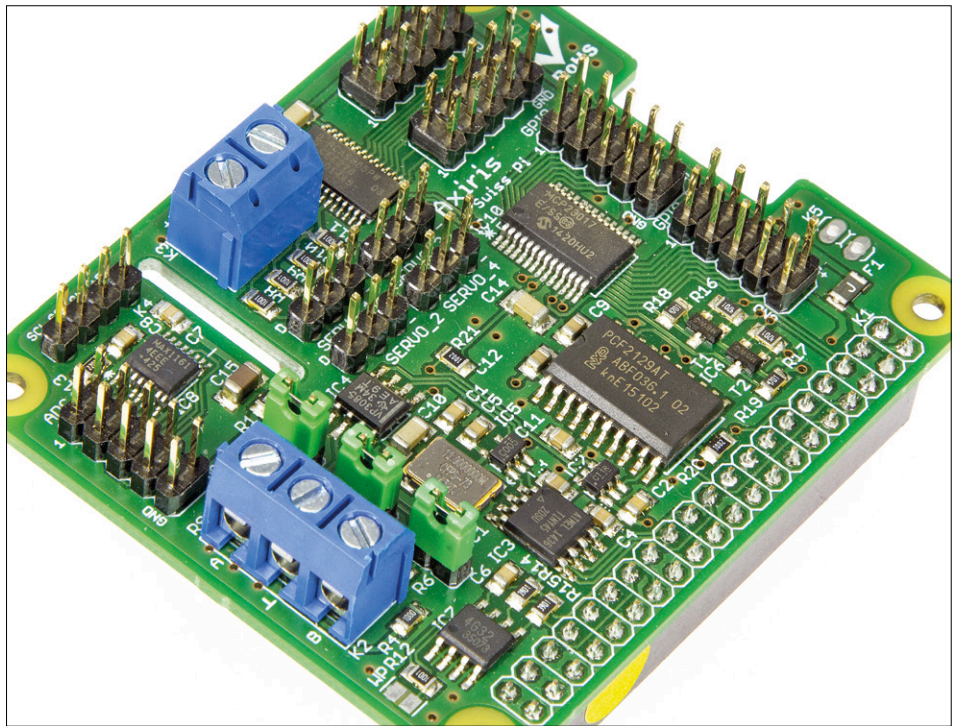


Figure 6. The fully assembled prototype module.

The layout files (Eagle) and the Gerber files are both available at [1]. Fully assembled Swiss Pi modules can be ordered from the Elektor Store [2].

(150584)

Web Links

- [1] www.elektormagazine.com/150584
- [2] www.elektor.com/swiss-pi

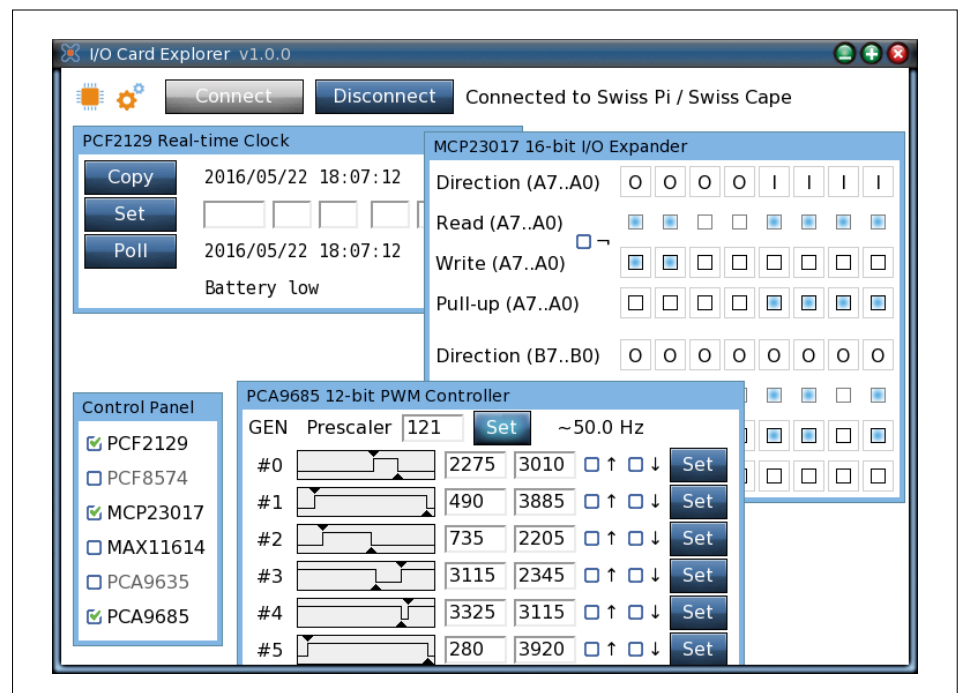
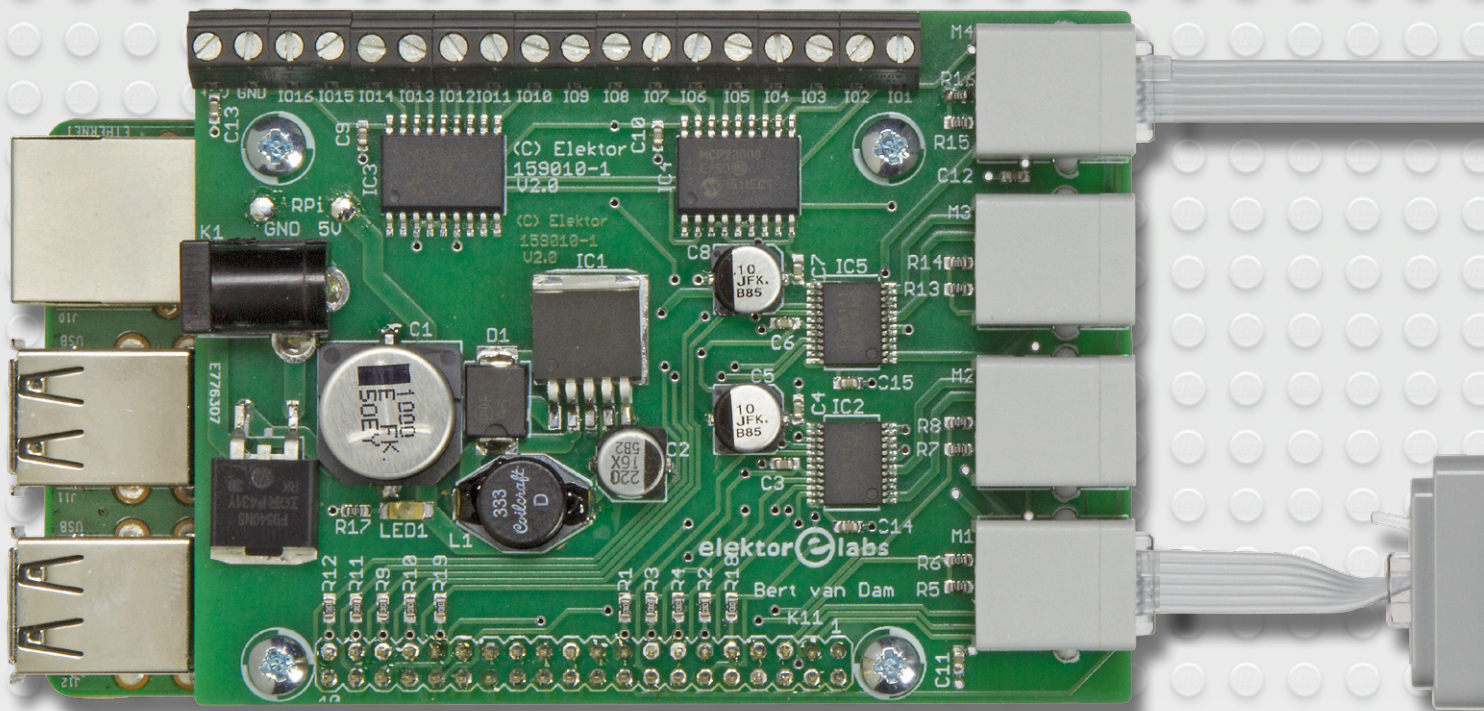


Figure 7. Screen shot of the I/O Card Explorer app for the project.

LEGO® Control Board for the Raspberry Pi



This Raspberry Pi HAT puts at your disposal 4 motor-control outputs for LEGO EV3 motors and 16 buffered I/O connections that can be used in combination with a powerful Raspberry Pi. Programming is possible in languages such as C and Python. As an application example we show how this board and a few LEGO parts can be used to build a 'useless box'.

The board described here combines the flexibility of LEGO with regards to

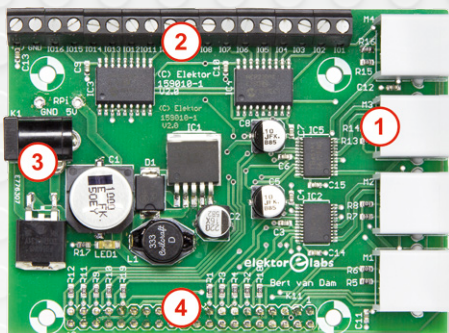


Figure 1. The connections to the LEGO control board.
1) 4 LEGO motor connections; 2) 16 I/O connections with screw terminals; 3) power supply socket, 9 V; 4) feedthrough of the Raspberry Pi I/O pins.

mechanical construction and the flexibility of the Raspberry Pi with regards to software, Internet communications and a wide range of potential sensors.

When you attach this board to a Raspberry Pi 2 or 3, you will have at your disposal 4 motor-control outputs for EV3 motors (or compatible types), which you can use to animate your LEGO models. There are also 16 buffered digital inputs and outputs present, to which you can connect sensors, LEDs, etc. You cannot, however, connect LEGO sensors (in that case you would be better served using a standard EV3 controller), but you can connect all sorts of other sensors such as infrared, ultrasonic, tilt and switches. You are limited only by your imagination. The board also contains a power supply

for the Raspberry Pi, you therefore only need to connect a 9-V (line) power supply to this board (recommended rating 5 A; positive to center pin). From this the motors, the board itself and the Raspberry Pi are powered. Mobile applications can use batteries, because the board uses an efficient buck converter.

Since the Raspberry Pi header is continued through the LEGO control board, you can also connect SPI and I²C devices and sensors. You can therefore combine all the projects and sensors from the book "Raspberry Pi – Explore the RPi in 45 Electronic Projects" with your LEGO models!

Through the Raspberry Pi itself you can communicate via the Internet, with your

With 4 motor-control outputs and 16 digital I/O channels

By Bert van Dam (Netherlands)



choice of a wired or Wi-Fi connection. If you connect a USB/4G dongle to your Raspberry Pi then you can even commu-

nicate with your model when it is away from your home.

With this board you can therefore make

models where you combine LEGO motors and various types of sensors with the power of the Raspberry Pi!

Table 1. Strong and weak points of the LEGO EV3 and Raspberry Pi.

LEGO — strong

- The mechanical construction, which allows you to do anything you like.
- The motors, although for serious applications you would need servos and hydraulics.

Raspberry Pi — strong

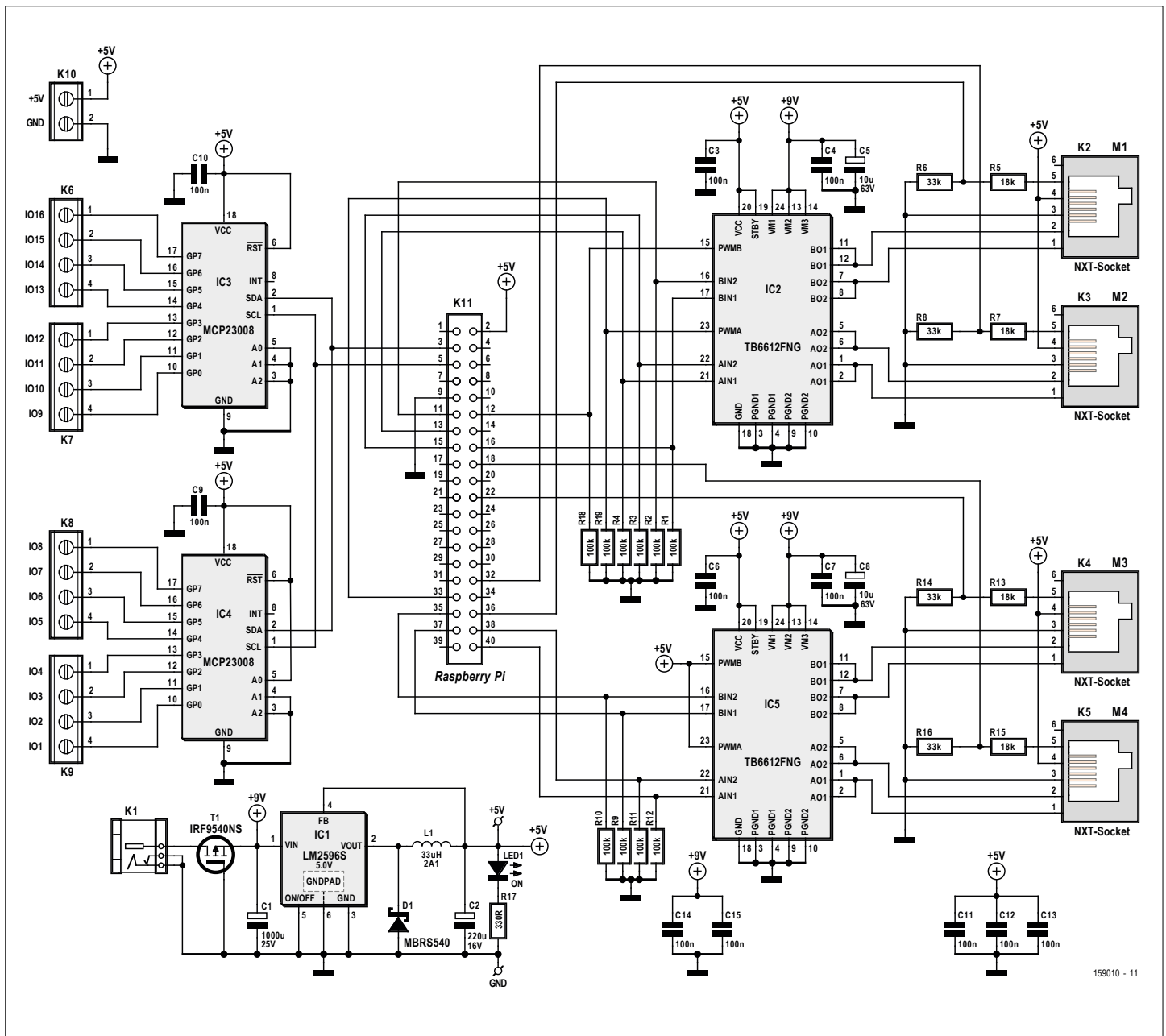
- Openness of the system; in combination was the easy-to-learn language Python much is possible. For the enthusiast C is an even more powerful option.
- Many connectors and via I/O-expanders offers the possibility of 'unlimited' expansion.

LEGO — weak

- The number of connections for sensors. A reasonably serious robot would need 4 and that is only the start.
- Video processing and all the other things for which LEGO have not come up with a sensor.

Raspberry Pi — weak

- Connection pins are mechanically weak and not electrically protected.
- Current consumption is on the high side.



159010 - 11

Figure 2. The circuit of the control board comprises 2 motor driver ICs, 2 I/O expanders and a step-down voltage regulator.

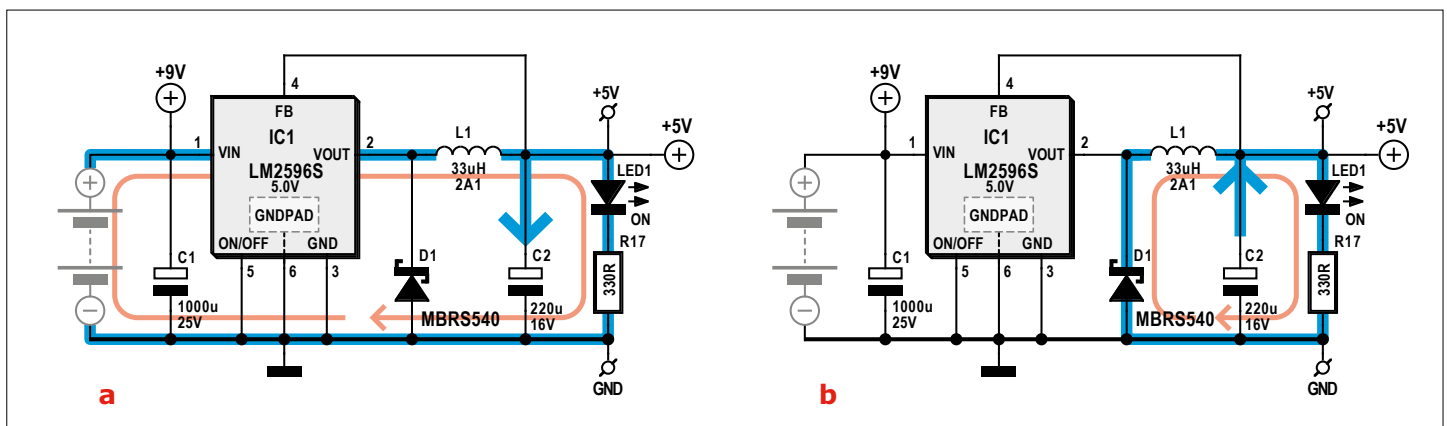


Figure 3. Buck converter: a) cranking up the flywheel b) coasting of the flywheel.

Design considerations

The LEGO control board is certainly not a copy of the EV3 controller, but it does combine the strong points of LEGO with those of the Raspberry Pi. **Table 1** lists the strong and weak points of each. The conclusion is that this board needed to have the following characteristics (and it therefore was given them):

- 4 LEGO EV3 motor control connections (for LEGO 45502, 45503 or compatible), with all 4 of provided with tacho feedback signals and 2 provided with PWM outputs;
- 16 buffered I/O pins (expandable thanks to the I²C connection);
- energy efficient power supply for the Raspberry Pi (using buck-converter);
- free Python 2 software library;
- feed-through of the Raspberry Pi header (you therefore can add further connections yourself, such as protected I/O or analog I/O).

Design result

The complete schematic for the LEGO control board is shown in **Figure 1**. The most relevant parts in this circuit are the two motor-driver ICs type TB6612FNG from Toshiba (which each can drive two motors with a maximum output current of 1.2 A) and two 8-bit I/O-expanders type MCP23008 from Microchip with a high-speed I²C interface and SPI interface. We now take a look at a few interesting details of the circuit.

Power supply

In order to enable everything to be powered a single source, the Raspberry Pi is powered from the control board. The board itself is powered from a 9-V DC power source. This power source can either be batteries or a substantial line power supply (recommend rating 5 A). To go from 9 to 5 V (the voltage required by the Raspberry Pi) an energy-efficient method needs to be used to prevent the batteries from draining too quickly. The choice therefore fell on a buck converter based on the step-down regulator LM2596S-5.0V, a so-called Simple Switcher Power Converter from Texas Instruments. This operates at a frequency of 150 kHz and can supply 3 A. The buck converter functions as follows (see **Figure 3**). The loop formed by the diode (D1), the inductor (L1) and the capacitor (C2) act like a kind of electronic

flywheel. The LM2596 begins by 'turning up' the voltage to 'full'. This gets the flywheel started. This does not happen immediately, because the capacitor first needs to be charged via the inductor. The inductor itself builds up a magnetic field. The diode does not conduct at this time since the current returns via the LM2596 itself.

When the voltage at the output threatens to become too high, the LM2596 switches its output off. The magnetic field in the inductor will now collapse, which causes the voltage across it to reverse and the diode will now conduct. This creates a current that circulates via the load and the diode (the path via the LM2596 is no longer possible because this is switched 'off'). The capacitor also discharges. When the voltage at the output threatens to become too low, the LM2596 switches on again.

The LM2596 therefore only switches on and off. The effect of this is that the efficiency is very high and the LM2596 therefore barely heats up. Because of this continuous on and off switching, the value of C1 is also important and for both C1 and C2 low-ESR types need to be used.

I/O

The MCP23008 I/O extenders IC3 and IC4 are controlled via I²C and ensure that the I/O connections of the Raspberry Pi do not need to be used directly. This design has several advantages:

- The I/O connections operate at 5 V
- The I/O pins have a greater current capability than the Raspberry Pi pins
- Should something nevertheless go wrong, the damage will generally be limited to the I/O extender chip

The maximum current that the MCP23008 can supply is 25 mA, with a maximum per IC of 125 mA. Note that one IC can dissipate a maximum power of no more than 700 mW. This applies to each MCP23008, so if you need a lot of current it would be best to spread the loads across both MCP23008s and not connect everything to a single IC.

Because the Raspberry Pi headers are continued through to the top of the control board you can add additional I²C components yourself, provided that they have different addresses. The LEGO control

board uses addresses 0x20 and 0x21. You can, for example, add further I/O-extendors, D/A converters for generating analog signals, or A/D converters to sense analog signals.

Motors

The two TB6621FNG ICs (IC2 and IC5) each contain two H-bridges to allow the control of four motors in total. The current consumption of the LEGO motors is approximately:

	EV3 small	EV3 large
free running	85 mA	80 mA
slowly forward	500 mA	1000 mA
stalled	780 mA	1800 mA

An EV3 motor can be stalled for only a short amount of time, because the protection built into the motor will switch it off. It is nevertheless always a good idea to prevent stalling the motor, because should the protection not work for some reason the motor could burn out.

The TB6621FNG can typically handle 1.2 A (per channel, so per motor) with a peak of 3 A; this is therefore more than sufficient.

If you intend to make a software library for the motor controllers, then take into account that the PWM pins have to be high in order for the TB6621FNG to operate. When you apply a PWM signal then it is always correct, independent of the direction of rotation.

Circuit board

In **Figure 4** you can see the circuit board that was designed for this LEGO control board. The dimensions and connector positions are obviously selected such that the board will fit trouble-free on a Raspberry Pi model 2 or 3 (using a 40-way I/O connector). Since most of the components are in SMD packages, you can also purchase a completely built version of the board from Elektor. Of course, it is also possible to do everything yourself, the circuit board layout is available from [1]. Note when assembling the board that the 40-way connector K11 is fitted on the bottom side of the circuit board and is soldered from the top side. The four NXT sockets for the motors are quite rare, you will have to search on the Internet for a suitable supplier.

Component List

Resistors

R1,R2,R3,R4,R9,R10,R11,R12,R18,R19 = 100k Ω 1%, 0.1W, case 0603
 R5,R7,R13,R15 = 18k Ω 1%, 0.1W, case 0603
 R6,R8,R14,R16 = 33k Ω 1%, 0.1W, case 0603
 R17 = 330 Ω , 0.1W, case 0603

Capacitors

C1 = 1000 μ F, 25V, aluminum, d 12.5mm, h 13.5mm, ELPP-CP-125-135
 C2 = 220 μ F, 16V, aluminum, d 6mm, h 7.7mm, ELPP-CP-063-066
 C3,C4,C6,C7,C9,C10,C11,C12,C13,C14,C15 = 100nF 10%, 16V, X7R, case 0603
 C5,C8 = 10 μ F, 63V, case D

Inductors

L1 = 33 μ H, 2.1A (e.g. Coilcraft DO3316)

Semiconductors

D1 = MBRS540, 40V 5A, ELPP-DO-214AB
 LED1 = LED, red, case 1206

T1 = IRF9540NSPBF, P-channel MOSFET, D2-PAK
 IC1 = LM2596S-5.0, DC/DC buck converter, TO-263-5
 IC2,IC5 = TB6612, motor driver, 24-SSOP (Digikey # TB6612FNGC8ELCT-ND)
 IC3,IC4 = MCP23008-E/SO, I/O expander, SOIC-18

Miscellaneous

K1 = DC adapter connector, 12V 3A, 1.95mm center pin
 K2,K3,K4,K5 = NXT socket
 K6,K7,K8,K9 = 4-way PCB screw terminal block, 3.81mm pitch
 K10 = 2-way PCB screw terminal block, 3.81mm pitch
 K11 = 40-pin (2x20) GPIO stacking header for RPi, extra tall
 PCB # 159010-1

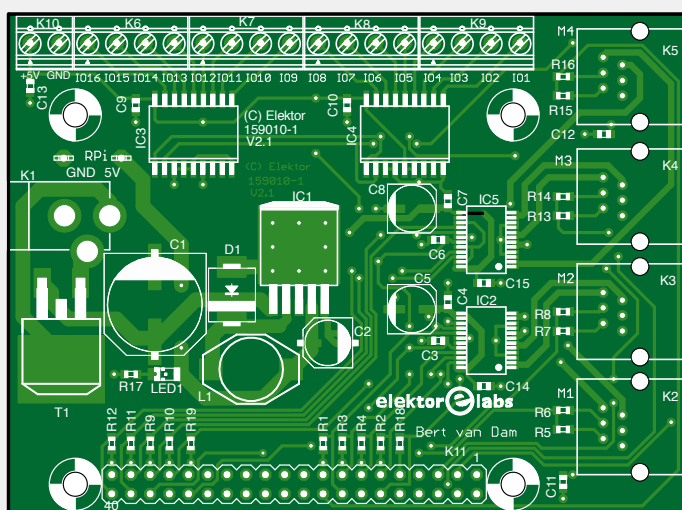


Figure 4. The layout of the circuit board. The 40-way connector K11 is mounted on the bottom of the circuit board.

workings; making the box we leave up to you. You can make this easily using cardboard or wood. With the download for this article you will find a short video of this project. Both the pictures accompanying this article as well as the video show only the mechanical part, without the box. This way you can get a clear view of its operation.

What do you need?

- The SD card from the book 'Raspberry Pi – Explore the RPi in 45 Electronic Projects';
- The free download that goes with this article (contains the library and the source code for the demo project);
- A LEGO motor (type 45502 'large' motor) with connecting cable and a few LEGO parts (you can build the demo project using the LEGO Mindstorms EV3 box).
- A Raspberry Pi model 2 or 3;
- The LEGO control board;
- Two resistors: 1 k Ω and 10 k Ω , 0.25W;
- An (easily operated) small in-line switch (see **Figure 4**).

The hardware

We use a 'large' LEGO motor type 45502 which can be found in, for example, the LEGO Mindstorms EV3 box and the parts that are shown in **Figure 5**. In the download you will find a short video where you can see how you assemble these parts into the motorized finger, which is shown on the right in the figure. You now attach the in-line switch on top of the assembly in such a way that the black beam can push the switch to its off position.

This switch is connected with two resistors (see **Figure 6**) to I/O pin 16. This could also be accomplished with only one resis-

Demo project: The Useless Box

The concept of this 'Useless Box' is a box with a switch on top. When you oper-

ate the switch, a 'finger' appears from the box which turns the switch off again. For this project we only build the inner

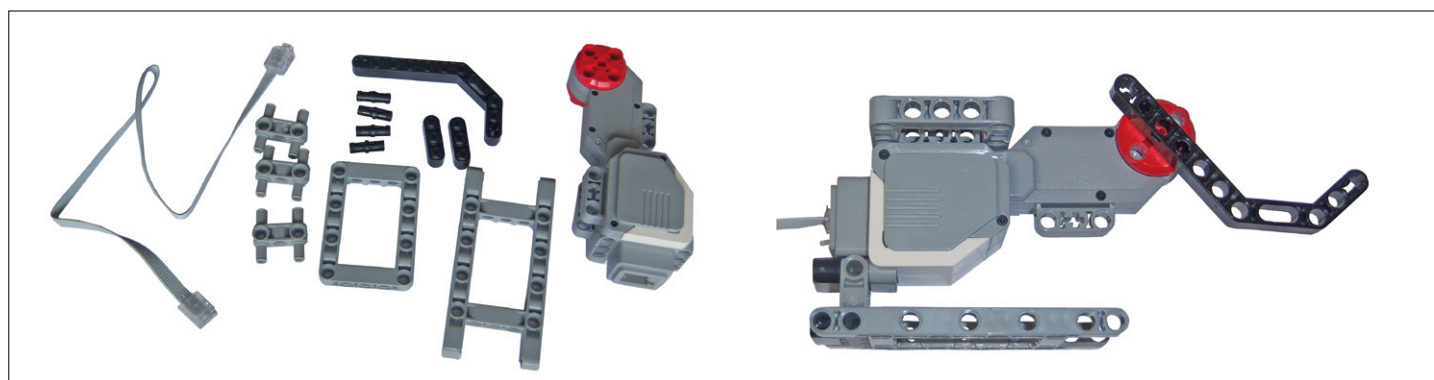


Figure 5. Parts and the assembled model of the motorized finger.

tor (the 10 kΩ one), but the additional resistor serves as protection. If you now inadvertently make the input into an output and set its level low then nothing bad will happen. Without the additional 1-kΩ resistor the I/O port could be damaged.

The software

The software is surprisingly simple. (Note: the library expects Python-2 programs!) You can find more information about the library and the instructions in the manual, which is a free download accompanying this project [1]. The very first thing we do is load the rpirobot library and execute it with the instruction 'execfile'. The motor outputs of the LEGO control board are now automatically configured correctly and the PWM threads are running. We then make pin 16 of the I/O on the control board an input, because this is where the switch is connected to.

```
execfile('rpirobot.lib')
```

```
# Switch on io 16, input
ioidir(16,1)
```

```
print "Useless Box Program running"
```

We use a try/except construct (see **Listing 1a**), so that the program waits until the switch is turned on by the user or the user has quit the program using Ctrl-C. Once this happens, the program waits half a second to give the user timer to remove their finger and then activates the LEGO finger with a PWM value of 60 (60% of the maximum speed). We wait in a loop because of the threads that are running. Since nothing happens in the loop this is not actually helpful, so we introduce a short sleep instruction. This instruction takes 0.000001 seconds (1 μs), and will

not noticeably slow down the program. As soon as the switch is off again, the program will turn on the brake in the motor. The next step (**Listing 1b**) is to return the LEGO finger at a slower speed (PWM 30, 30% of the maximum speed) to its rest position. For this case there is no switch at the end of the movement, so the program now uses the tacho. The program counts the tacho changes and once there are 100 (28% of a circle) the motor stops with the float instruction. The LEGO finger is now about one centimeter above the surface.

In the except-construct (**Listing 1c**) any errors or a Ctrl-C instruction from the user are caught. The program gives a brief abort instruction to the Raspberry Pi library and then terminates itself as well. Without this construct it is obviously still possible to exit the program using Ctrl-C, but then the library continues running and that is clearly not the intention.

The steps

1. Set the LEGO finger in the rest position, that means as far as possible from the switch (against the bottom surface for example).
2. Start the Raspberry Pi and wait until the booting process has completed.
3. Copy the test folder from the download to the Raspberry Pi (using WinSCP, for example).
4. Return to the test folder with the instruction 'cd test'.
5. Now issue the instruction 'python uselessbox.py'.
6. Wait until the text 'Useless Box Program running' appears on the Raspberry Pi and turn the switch on (and get your fingers out the way!). The LEGO finger will now return the switch to its off position.

7. Repeat step 6 as many times as you like and then stop the program using Ctrl-C.

This is only a simple application example. The LEGO control board can of course be used to realize much more complicated applications with its four motor control connections and 16 I/O pins. ◀

(150597)

Web Links

[1] www.elektor.com/150597

Listing 1a.

```
try:
    while 1:
        # wait for the user
        # to push switch on
        if ioread(16):
            # push switch off
            time.sleep(0.5)
            reverse(1)
            pwm(1,60)
            while ioread(16):
                time.sleep(0.000001)
            brake(1)
```

Listing 1b.

```
# retract lever
forward(1)
pwm(1,30)
counter1=0
lastpin1=tacho(1)
while counter1<100:
    time.sleep(0.000001)
    if tacho(1)!=lastpin1:
        counter1+=1
        lastpin1=tacho(1)
# float motor
float(1)
```

Listing 1c.

```
except:
    # user aborts program with
    Ctrl-C
    print "Program aborting"
    abort()
    print "Done"
```

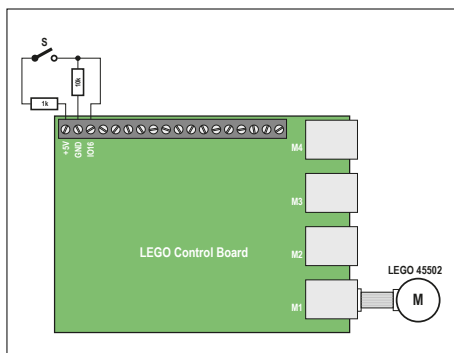


Figure 6. Connection diagram for the switch with the two resistors.

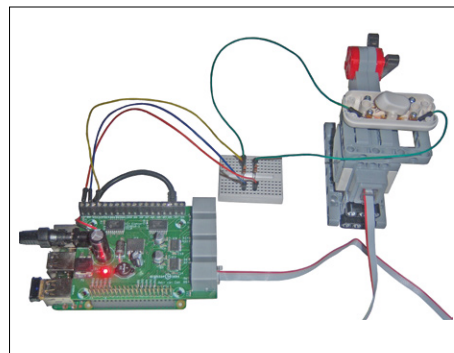
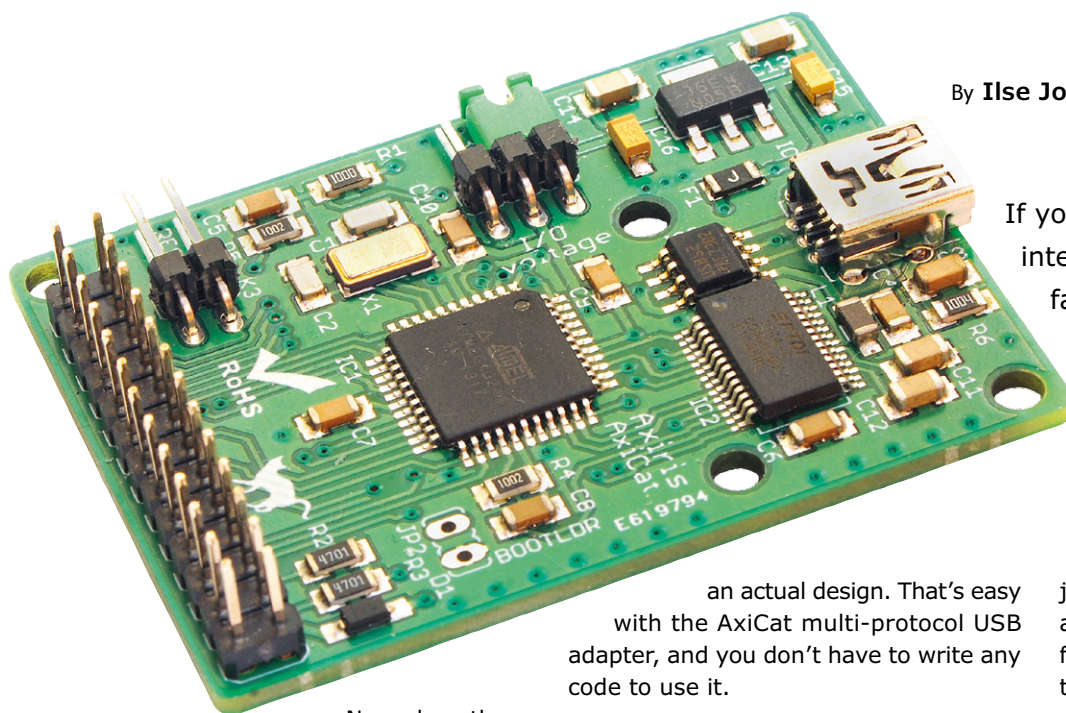


Figure 7. The complete model.

AxiCat

A versatile USB development tool



By **Ilse Joostens** and **Peter S'heeren** (Belgium)

If you are developing an app for an integrated circuit and you want a fast and easy way to link it to a Linux or Windows PC, we have the answer: AxiCat.

Nowadays there are thousands of useful ICs available to electronics hobbyists and professionals. A lot of them only come in SMD packages, which are not very convenient for hobbyists, but don't worry: many well-known (and less well-known) vendors who focus on the hobby sector offer a broad spectrum of breakout boards. Most of these ICs have an I²C or SPI interface, and it would be handy if you could quickly connect them to a Linux or Windows PC to check out their functions before you start using them in

an actual design. That's easy with the AxiCat multi-protocol USB adapter, and you don't have to write any code to use it.

Schematic and structure

The key component of the circuit, as shown in **Figure 1**, is an Atmel ATmega164A microcontroller (IC1). You can also mount other members of the same microcontroller family, such as the ATmega324A, 644A and 1284. The only difference is the amount of internal memory. The AxiCat normally comes with an ATmega164A or an ATmega324A, depending on availability.

The microcontroller is powered from 5 V or 3.3 V, depending on the position of

jumper JP1. For that reason we opted for a 12 MHz clock crystal instead of a higher frequency, to ensure that the microcontroller remains within its safe operating area with a 3.3 V supply voltage.

The USB connection is handled by IC2, an FT245RL from FTDI. Unlike the better known FT232RL, this device does not have a serial interface, but instead an internal FIFO with an 8-bit data bus. The advantage of this is that with Full Speed USB and a 12 MHz clock frequency, you can achieve a respectable data rate between the USB bus and the microcontroller (IC1).

Thanks to the underlying layers of the USB communication protocol between the driver in the host system and the FT245RL, the data arrives in the FIFO of the FT245RL without any bit errors. This means that the data read by the microcontroller is error-free, and the data output by the microcontroller also arrives in the host system free of errors, eliminating the need for error correction in the communication protocol. That also pays dividends in terms of speed. IC3, a USB6B1, protects the bus lines against ESD, making the board significantly more robust. The 3.3 V supply voltage is provided by IC4, an LM1117-3.3 low-drop voltage regulator. This voltage is available on GPIO connector K2, and along with the 5 V supply voltage from the USB port it can be used as desired to power the microcontroller and the I/O portion of IC2.

Features

- Based on a powerful ATmega164A microcontroller clocked at 12 MHz
- Compatible with Windows and Linux
- Selectable 3.3 V and 5 V logic levels
- 17 bidirectional I/O lines with configurable pull-up
- I²C bus with pull-up resistors, master and slave mode; maximum speed 400 kHz
- SPI master with four slave select lines; maximum speed 6 MHz
- Two independent serial ports
- 1-Wire master with strong pull-up option and built-in accelerators for enumeration
- Powered over the USB bus
- Asynchronous design – all interfaces can be used simultaneously
- Can be used as an in-system programmer for AVR microcontrollers
- Boot loader for easy firmware upgrade and loading your own software
- GPIO connector pinout compatible with Raspberry Pi A and B

Polyfuse F1, with a rated value of 350 mA, limits the total current consumption to prevent overloading of the supply voltage lines of the USB port. Specifically, this means you should keep the total current drawn from the GPIO connector (I/O lines, 5 V supply voltage and 3.3 V supply voltage) below 350 mA.

Diode D1 in the 5 V supply line to pins 2 and 4 of GPIO connector K2 is there to prevent extension cards that supply

power to the Raspberry Pi from also feeding power to the AxiCat and through the USB port to the computer.

GPIO connector K2 is a 26-pin header. This connector and its pinout (see **Figure 2**) were not chosen randomly; they largely correspond to the GPIO connector on the early Raspberry Pi models (A and B). The main advantage of this is that a large number of extension cards and HATs for the Raspberry Pi can also be

connected to the AxiCat. However, extension cards for the newer Raspberry Pi models with additional I/O lines on the 40-pin connector (A+, B+, 2 and 3) are not compatible.

Finally there is jumper/connector K3, which can be used to manually reset the AxiCat. If jumper JP2 is kept closed at the same time, the internal boot loader is activated. This allows you to send your own firmware to the AxiCat.

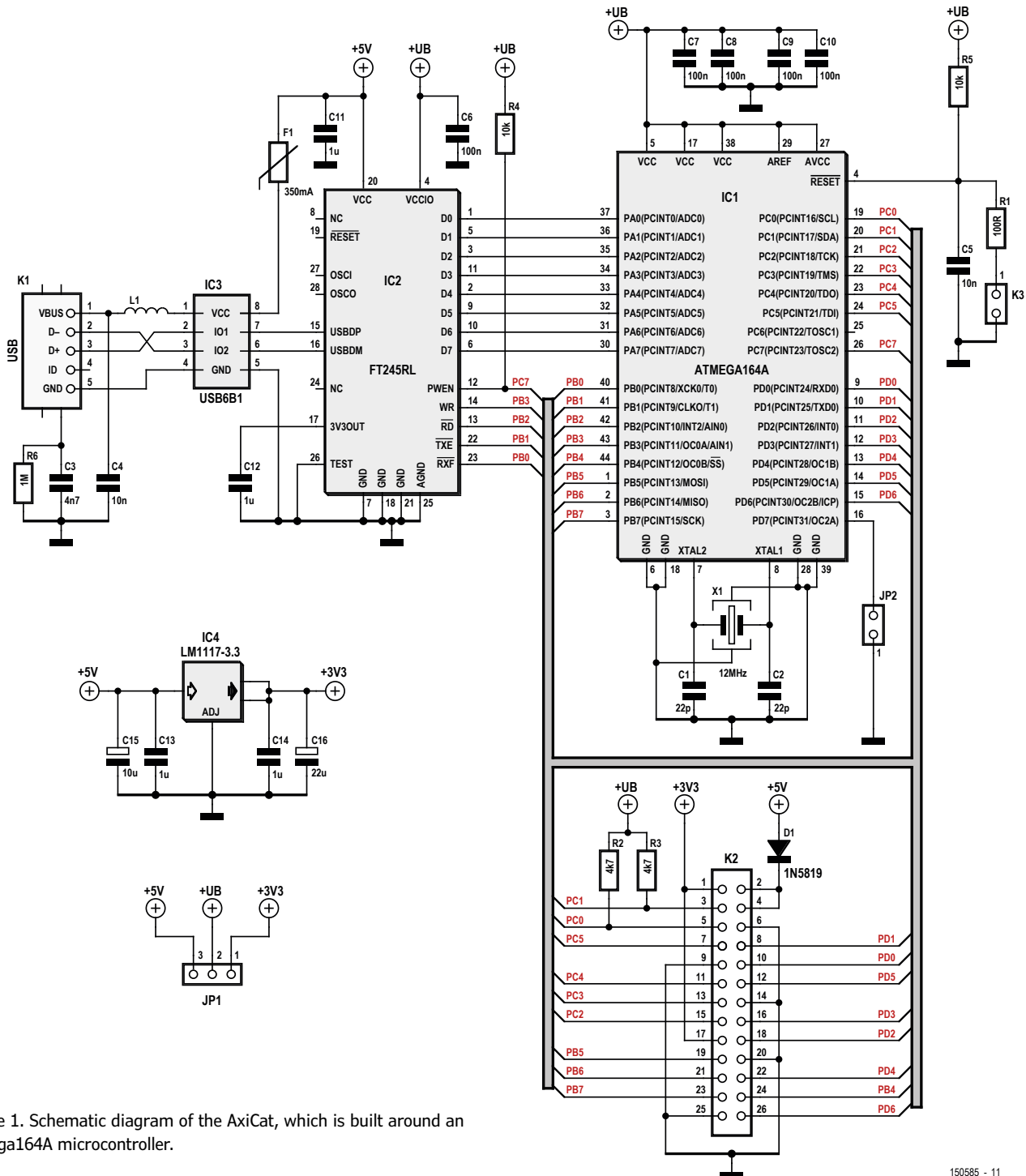


Figure 1. Schematic diagram of the AxiCat, which is built around an Atmega164A microcontroller.

150585 - 11

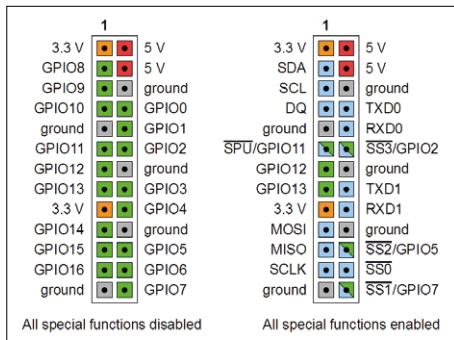


Figure 2. The header pinout is designed to be compatible with Raspberry Pi models A and B. This allows a large number of Raspberry Pi HATs to be connected directly to the AxiCat.

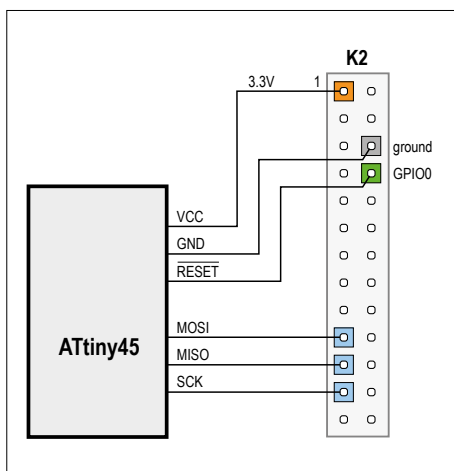


Figure 3. Using the AxiCat as a programmer connected to an ATtiny45.

Serial data protocol

The serial data protocol consists of byte packets which flow between the host and the AxiCat in full duplex mode. The host sends command packets, and the AxiCat sends response packets. The file *axi-cat.h*, included in the download package from [1], contains the definitions in the C language.

There are two types of commands: synchronous and asynchronous. They differ in how the AxiCat executes them after they are received. Synchronous commands are executed immediately. If a response package is defined for the command, the AxiCat returns it to the host immediately. Among other things, synchronous commands are used to read status data, modify settings, and access GPIO lines.

Some commands take a relatively long time to execute. These are commands

that involve bus transactions, such as an SPI transfer or a 1-Wire enumeration. To avoid blocking of the incoming flow of data packets, the AxiCat executes long-duration commands asynchronously, which means in the background. With asynchronous commands, the AxiCat waits until the command has been executed before it returns a response to the host.

Asynchronous commands are stored in working buffers, which are available for the following components: I²C master, I²C slave Tx, I²C slave Rx, SPI master, 1-Wire master, UART #0 Tx, UART #0 Rx, UART #1 Tx, and UART #1 Rx. Note that the UART working buffers do not retain any commands; they only send data bytes (Tx) or receive data bytes (Rx). The working buffers enable the AxiCat to perform bus transactions on several different interfaces in parallel. The effective size of the working buffers can be configured in the firmware and is primarily determined by how much RAM is available in the microcontroller.

There are also autonomous response packets, which are called unsolicited responses. For example, there is no command for reading received data bytes from a UART. This means that the AxiCat automatically sends a response packet when a UART receives data bytes.

Firmware

The main program of the AxiCat is *AxiCat Main*, which is what makes everything tick. The serial data protocol is implemented here. The main program is written in C; we used WinAVR 20100110 for this. WinAVR is a Windows version of GCC for AVR 8-bit microprocessors and AVR Libc. During the development of the AxiCat we frequently checked the generated assembler code, and in many cases we rewrote the source code to make the machine code more efficient and compact. That was necessary for two reasons: firstly because fast iteration of the main loop is crucial for the performance of the AxiCat, and secondly because the main program has to fit in the 16 KB of flash memory in the ATmega164A.

We opted to use interrupts as little as possible, because the GCC compiler often generates interrupt routines that push a lot of registers onto the stack and later retrieve them, which degrades the effi-

ciency of the interrupts. Instead of interrupt routines, we use non-blocking tasks integrated into the main loop. *TIMER1*, which acts as a millisecond counter, is the only hardware function that works with an interrupt routine.

We chose the size of the working buffers based on the available RAM and the code generated by the compiler. The main program uses working buffers with a size of 2^n bytes and a maximum size of 128 bytes, which results in very efficient and compact code for an AVR 8-bit microcontroller.

The main program supports USB remote wakeup, so the AxiCat can wake the host computer when it is in sleep mode. Of course, this is only possible if the operating system of the host also supports USB remote wakeup.

AxiCat Bootloader is executed when the AxiCat starts up. The boot loader first checks the state of jumper JP2. If the jumper is open, the boot loader jumps to the main program. If the jumper is closed, the boot loader remains active and you can use the boot loader program *aximcubldr* to download a new main program from the host computer to the AxiCat.

Software

The *AxiCat Server* program provides the functionality of the AxiCat via network ports and standard I/O in the form of a communication protocol. The server can accept multiple client connections, allowing several clients to work with the AxiCat concurrently. The server acts as a multiplexer between several clients and a single AxiCat, which creates a lot of flexibility.

The communication protocol consists of commands and corresponding responses. A client sends a command, and the server returns a response once the command has been executed. The communication protocol is coded in ASCII format. The syntax is designed to make the communication protocol easy to use both manually and in software.

The server supports a large number of commands for working with the AxiCat. Some examples are:

Configure GPIO pin 1 as an output, set the output low and read the state:

```
Command  iod 1 out
Response  iod ok
```



```

Command iow 1 0
Response iow ok
Command ior 1
Response ior 01 1 0 out

```

PCF2129A real time clock: activate I²C master, write register index 0, read registers 0–9:

```

Command ime
Response ime ok
Command imw 81 0
Response imw 081 00001 ack
Command imr 81 10
Response imr 081 08h 16h 04h 30h
          22h 16h 06h 00h 08h 16h nack

```

Activate SPI master, transfer four bytes and use slave select #2:

```

Command sme
Response sme ok
Command smt 2 0FFh 0FFh 10100b 5
Response smt 2 000 255 032 005

```

Many responses from the server contain numbers. The format of these numbers is fully configurable – you can choose from binary, decimal and hexadecimal, with or without leading zeros.

The communication protocol is fully asynchronous, which means that a client does not have to wait for the response to a command before sending the next command. This allows clients to send several commands in direct succession, which dramatically increases communication efficiency. Each command can optionally be provided with an identification number, which is sent back with the response. For each of the two UARTs, the server offers the option of transferring data bytes directly between the UART and a network port. This option allows you to use a terminal emulator program such as PuTTY to read and write data directly through the UART.

The *AxiCat Application Layer* module provides a C API on top of the serial data protocol. The C API supports the full functionality of the serial protocol, including asynchronous transfers, so that programmers can manage the various interfaces of the AxiCat optimally and operate them in parallel. The source code of the AxiCat server provides good insight into how to use the C API. This module is used in a variety of programs, including *AxiCat Server*, *AxiCat AVR ISP*, *I/O Card Explorer*, and *1-Wire Automation Server*.

AxiCat AVR ISP converts the AxiCat into an in-system programmer for 8-bit AVR microcontrollers. It can program flash memory, EEPROM and fuses, and it can read Intel hex files and raw files. *AxiCat AVR ISP* works with the SPI bus (but without slave select) and a reset line. You connect the SPI bus between the AxiCat and the microcontroller, and any desired GPIO pin to the reset pin of the microcontroller. An example is shown in **Figure 3**.

The *AxiCat Command Line* tool converts simple text commands into byte packets and sends them directly to the AxiCat. This tool is mainly useful for developing and debugging the main program for the AxiCat. You can use it to send invalid packets to the AxiCat, which is a good way to test the robustness of the main program. It is also a handy tool for anyone who wants to study the serial data protocol.

Summary

We mainly use the AxiCat to test our own extension boards for the Raspberry Pi (Swiss Pi and PiWire+) and to install firmware on these boards. We also use the AxiCat in all of our software development, because writing and debugging C code is a lot faster and easier on a PC than on a Raspberry Pi. At the end of the development process, we compile and test the software on a real Raspberry Pi. The AxiCat is available from the Elektor Store as a fully assembled module [2]. You can find a lot more background information and all necessary software at [3]. ◀

(150585-1)

Web Links

- [1] Software download:
www.elektormagazine.com/150585
- [2] AxiCat in the Elektor Store:
www.elektor.com/150585
- [3] More information and software:
www.axiris.eu/en/index.php/i-o-cards/axicat

Component List

Resistors

Default: SMD 1206
 R1 = 100Ω
 R2,R3 = 4.7kΩ
 R4,R5 = 10kΩ
 R6 = 1MΩ

Capacitors

Default: SMD 1206
 C1,C2 = 22pF
 C3 = 4.7nF
 C4,C5 = 10nF
 C6–C10 = 100nF
 C11–C14 = 1μF
 C15 = 10μF 10V, tantalum, case A
 C16 = 22μF 10V, tantalum, case A

Inductors

L1 = ferrite core, 3A (Farnell #1653393)

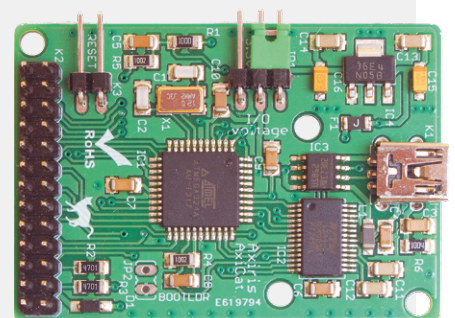
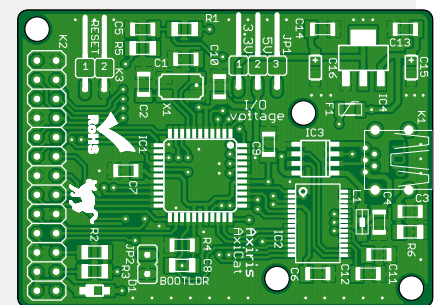
Semiconductors

D1 = 1N5819, SOD123
 IC1 = ATmega164A-AU, programmed*
 IC2 = FT245RL
 IC3 = USB6B1
 IC4 = LM1117IMP-3.3

Diversen:

F1 = polyfuse 350mA, 1206
 JP1 = 3-pin pinheader, right angled, 0.1" pitch with jumper
 K1 = mini-USB connector, type B
 K2 = 26-pin (2x13) pinheader, 0.1" pitch
 K3 = 2-pin pinheader, right angled, 0.1" pitch with jumper
 X1 = 12MHz quartz crystal

*see text



Internet Radio with Fluorescent Display (1)

Modern technology with a traditional look

A fine thing, a kitchen radio, with its CD player, clock and even a timer to make sure you don't burn your dinner, all in one compact unit. So, when the author's unit gave up the ghost a replacement had to be found. That display was a particularly neat feature: not too big, but with everything necessary on it. Perhaps it could be recycled into a new homebrew radio?

By **Michael Busser** (Germany)

A pretty good idea, at least by this engineer's standards. But, before turning a concept into reality, there was of course some (Internet-based) research to be done. Sadly, no sooner had I managed to bring together all the necessary information about the complexities of the display module and the perils of driving it, than an unfortunate accident turned my VFD (vacuum fluorescent display) into a mere FD, the V having been irreversibly dissipated into the atmosphere.

Fortunately similar displays are widely available on the surplus market, and so a 'new old' device can be had cheaply. I managed to obtain a type FV651G VFD (see **Figure 1**): excellent, if not quite as beautiful as the original.

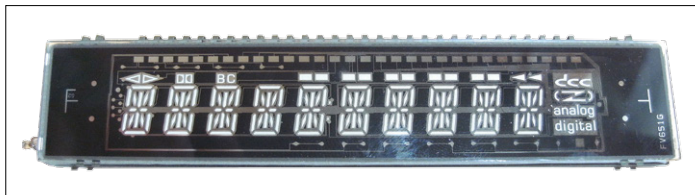


Figure 1. The FV651G VFD used in this project.

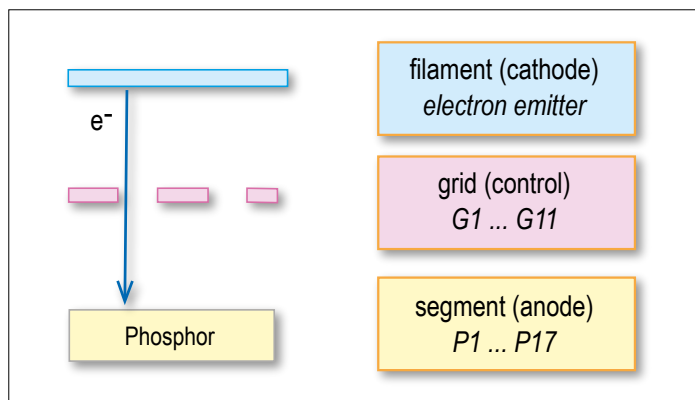


Figure 2. Structure of a VFD. The phosphor-coated anodes light up when they are hit by electrons.

How it works: the VFD

The principle of operation of each segment of a VFD is essentially the same as that of a (triode) tube or valve: see **Figure 2**. Details of its construction can be found at [1] and [2], but in simple terms it works as follows. As with all tubes there is a coated heater filament, which is the cathode of the device and which emits electrons. Each of the individual segments of the display (the anodes) has a phosphor coating: if electrons hit the phosphor it glows. Meanwhile, the voltage on the grid allows the flow of electrons to be controlled. A negative potential prevents the illumination of all the segments (in this case up to 17 of them) in a character position. With a positive potential on the grid the segments of the digits are allowed to be lit, but which segments do in fact light depends on their individual anode potentials. A negative anode potential repels the electrons from the segment, allowing them to be collected on the positive grid: the segment will not light. However, if the anode potential is sufficiently positive, the electrons will hit the phosphor and the segment will light. The grid connections G1 to G11 determine which character position is enabled, while the anode connections P1 to P17 control the individual segments within each character.

According to its datasheet the type FV651G display we use here requires an AC supply of 3.7 V for its heater, while the grid voltage is specified at 32 V. The latter of these requirements makes our driver circuit somewhat more complex. The display has eleven character positions, the one at the far right comprising special annunciators required for its original application. In each character position there are up to seventeen individually-controllable elements; the seventeenth segment, the colon, is only present in one position.

Corresponding segments are all connected together, which means that multiplexed operation is required. The display can conveniently be mapped in software to an array with eleven columns and seventeen rows.

Heater

In the defunct kitchen radio the heaters were powered from the mains transformer using a winding with a center tap. However, the plan is for the new radio design to run from an ordinary

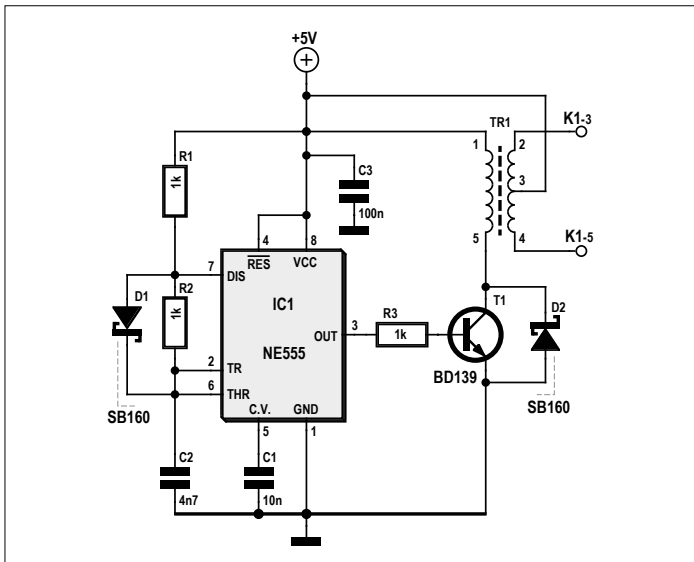


Figure 3. This auxiliary circuit by Kerry D Wong is used to supply the heater.

12 V mains adapter, and so we must look for an alternative approach. Note that it is not essential for the AC to have a frequency of exactly 50 Hz: **Figure 3** shows our solution using a good old NE555 timer and a custom-wound transformer.

A modification from the standard circuit is the addition of D1, which makes the output squarewave almost symmetrical. The transformer is wound on an N27 ferrite core with an external diameter of 16 mm. The primary side consists of 40 turns of enameled copper wire and the secondary side comprises two windings of 18 turns each. The result should be a voltage of just under 2 V_{RMS} across each secondary winding under load. The original circuit was designed by Kerry D Wong [3]. The drive voltage for the heater should not be centered around ground potential, as this can cause 'ghosting' (weak illumination of undriven segments): this is why the center tap of the transformer is tied to +5 V.

Anode voltage

The anode voltage is also generated from the 12 V output of the mains adapter, using a step-up converter: see **Figure 4**. The only change from the basic circuit given in the LM2577T datasheet is the value of the resistors.

According to the manufacturer the output voltage is given by the formula:

$$V_{OUT} = 1.23 \text{ V} \times (1 + R5 / R6)$$

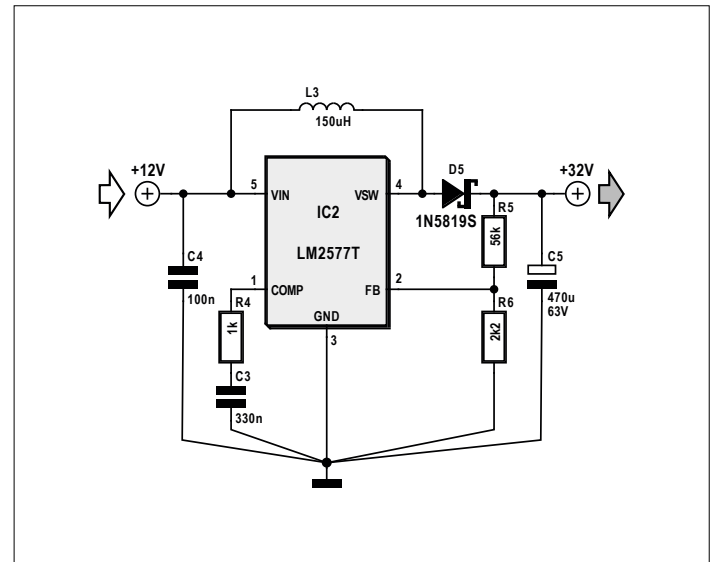


Figure 4. Auxiliary circuit to create the anode supply using a step-up converter.

With the values of R5 and R6 shown the anode voltage will be 32.5 V, as required.

Grid drive

In P1 to P17 and G1 to G11 we have a total of 28 display electrodes that have to be driven. The high anode voltage of 32 V means that we will have to use what is known as a 'high side driver' circuit. A suitable device for this application is the UDN2981. This IC contains eight drivers, and so we need a total of four chips. On the input side, these devices just need a regular 5 V drive signal.

The ATmega32 we are using in this project has only a limited number of I/O pins, and in order to avoid sacrificing almost all of them to driving electrodes an eight-bit latch is wired before each driver IC. The four latches are driven in parallel, and one latch is selected using its LE ('latch enable') signal. This reduces the number of outputs required to drive the display to twelve (eight data bits and four LE signals). The microcontroller can also check the status of a couple of user controls. The complete circuit diagram for the display driver is available as the document *Muxer_Schaltplan.pdf* in the zip archive *Muxer.zip* (see the download at [4]). The corresponding Eagle design files are also included in the archive.

ATmega firmware

From the point of view of the microcontroller the display is represented by a 32-bit value divided into four bytes, one for each latch. The main job of the microcontroller is to output these

Table 1. Meaning of the 32 bits

Byte	3								2								1								0							
Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Electrode	G11 to G01											P1 to P17																	unused			
Function	10	9	8	7	6	5	4	3	2	1	0			a	b	f	k	j	h	m	g	n	p	r	c	e	d					
	Character position (grid)											Segments (anode)																				

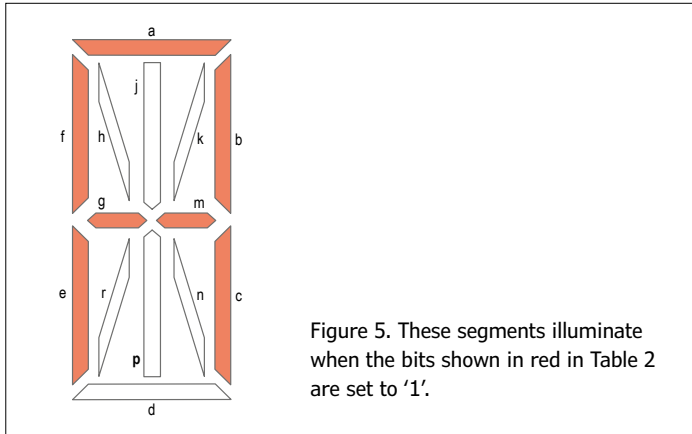


Figure 5. These segments illuminate when the bits shown in red in Table 2 are set to '1'.

Table 2. Driving the segments shown in Figure 5

Bit	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Electrode	P1 to P17																
Segments				a	b	f	k	j	h	m	g	n	p	r	c	e	d

four bytes to an eight-bit port at the multiplex frequency, and supply the right control signals to the latches. **Table 1** shows the meanings of the individual bits in the 32-bit word. The four least-significant bits are not used. The definitions can be found in the files *fv651.c* and *fv651.h* in the download.

The data structure *TDispControl* brings together the variables needed to drive the display:

```
typedef struct {
    uint32_t data;
    uint8_t grid;
    uint16_t anode;
} TDispControl;
```

The variable *data* contains the 32-bit pattern for the four latches; *grid* is a counter indicating the current display character position (from 0 to *MAX_DIGITS*). The variable *anode* contains the bit pattern of the 17 segments. The routine *flush_VFD()* shown in **Listing 1** transfers the contents of *data* to the four latches.

Listing 1. Processing the 32-bit word.

```
void flush_VFD() {
    uint8_t data;

    data = (ctrl.data & 0xFF);           //1st byte
    PORT_VFD_DATA = data;
    LE_VFD_0_ON;                         //impulse with nop =
    asm volatile („nop“);                //760ns
    LE_VFD_0_OFF;                       //measured with scope

    data = ((ctrl.data >> 8) & 0xFF);    //2nd byte
    PORT_VFD_DATA = data;
    LE_VFD_1_ON;                         //delay between 1st and 2nd impulse =
    asm volatile („nop“);                //2us
    LE_VFD_1_OFF;

    data = ((ctrl.data >> 16) & 0xFF);    //3rd byte
    PORT_VFD_DATA = data;
    LE_VFD_2_ON;                         //delay between 1st and 3rd impulse =
    asm volatile („nop“);                //4.7us
    LE_VFD_2_OFF;

    data = ((ctrl.data >> 24) & 0xFF);    //4th byte
    PORT_VFD_DATA = data;
    LE_VFD_3_ON;                         //delay between 1st and 4th impulse =
    asm volatile („nop“);                //7.5us
    LE_VFD_3_OFF;
}
```

Listing 2. ASCII table and character range tests.

```
uint16_t lookupVFDsegTable(uint8_t index) {
    static const flash uint16_t lookupSeg[SEGMENT_LOOKUP_TABLE_SIZE] = {
        14343, 4100, 12483, 12485, 6340, 10437, 10439, 12292, 14535, 14532, //0...9
        14534, 14535, 10243, 14343, 10435, 10434, 10375, 6342, 528, 536, //A...J
        1584, 2051, 7430, 6438, 14343, 14530, 14375, 14562, 10437, 8720, //K...T
        6151, 4388, 6190, 1320, 784, 8193, //U...Z
        103, 2247, 195, 4295, 83, 0, 0, 2130, 0, 0, //a...j
        0, 0, 0, 0, 103, 0, 0, 66, 0, 0, //k...t
        7, 36, 46, 0, 37, 73 //u...z
    };
    if (inRange(LUT_0, LUT_9, index)) {return(lookupSeg[index - LUT_0]);} //48...57
    if (inRange(LUT_A, LUT_Z, index)) {return(lookupSeg[(index - LUT_A) + LUT_A_OFFSET]);} //65...90
    if (inRange(LUT_a, LUT_a, index)) {return(lookupSeg[(index - LUT_a) + LUT_a_OFFSET]);} //97...12
    return (0);
}
```


Listing 3. Display firmware interface.

```

void init_VFD();                //initialize all data structures
void mux_VFD();                 //displays next digit, timer controlled

void clearVFDDisplay(uint8_t withSymbols); //clears display
    withSymbols = true           //deletes all symbols
    withSymbols = false         //lets symbols untouched

void clearVFDDisplayData(uint8_t index);    //deletes position „index“ in display
                                           //0 = most right digit

void setVFDDisplayData(uint8_t index, uint8_t data); //shows „data“ on digit „index“

void setVFDDisplayText(uint8_t index, const char *data); //shows string „*data“,
                                                         //starts @ position „index“

void setVFDdisplayInt(uint8_t index, uint8_t anzStellen, uint8_t wert);
                                           //shows number „wert“ with „anzStellen“
                                           //digits starting from righth

void clearRunText();

void setRunText( const char *data );

void tickRunText();

```

In sequence the four bytes are extracted from the 32-bit word and written to port A, which is connected to the latches' inputs. After each byte is written a pulse is generated on the appropriate latch enable signal so that the byte is stored in the latch. The length of this pulse, which is extended by the 'nop' instruction, is 760 ns. The execution time of the complete function is 9 µs at a clock rate of 16 MHz.

ASCII table

We are left with the question of how the bit pattern corresponding to a character arrives in the variable *data*: this is taken care of in the function *mux_VFD()*. Suppose that we want to display the letter 'A' as shown in **Figure 5**. The segments we want to light are a, b, c, e, f, g and m. What is needed is a look-up table that allows us to convert from a code for a displayable character to a bit pattern. **Table 2** shows the bit pattern for this example, which corresponds to the decimal value 14534. Of course, we need a 16-bit value in the table for each character we might want to display, and generating these values can be done conveniently using a spreadsheet. The table can also be found in the download, and can easily be modified to suit any particular application.

The table of 16-bit values is stored in the microcontroller's memory, and the character code serves as an index into the table. For reasons of economy not all possible codes are included in the table and so there are gaps in the coverage against which the character code must be tested. The ASCII table and the code to test the character value are shown in **Listing 2**.

If this code is called with the ASCII value for 'A' (65 decimal), the second of the three 'if' statements will be triggered and the value to be returned will be read from the table at offset $65 - \text{LUT_A} + \text{LUT_A_OFFSET} = 10$.

Among the lower-case letters there is a number of zero entries in the table. An exercise for the reader: expand the table to that all the lower-case letters can be displayed.

What remains to be done is some bit-shifting so that the data bits read from the table occupy the right field in the 32-bit

word. The structure *TDispData* includes a member called *digits* with space for *MAX_CHARS* characters and a bit field called *symbols* that contains a bit for each annunciator that can be displayed outside the main display area.

Advertisement

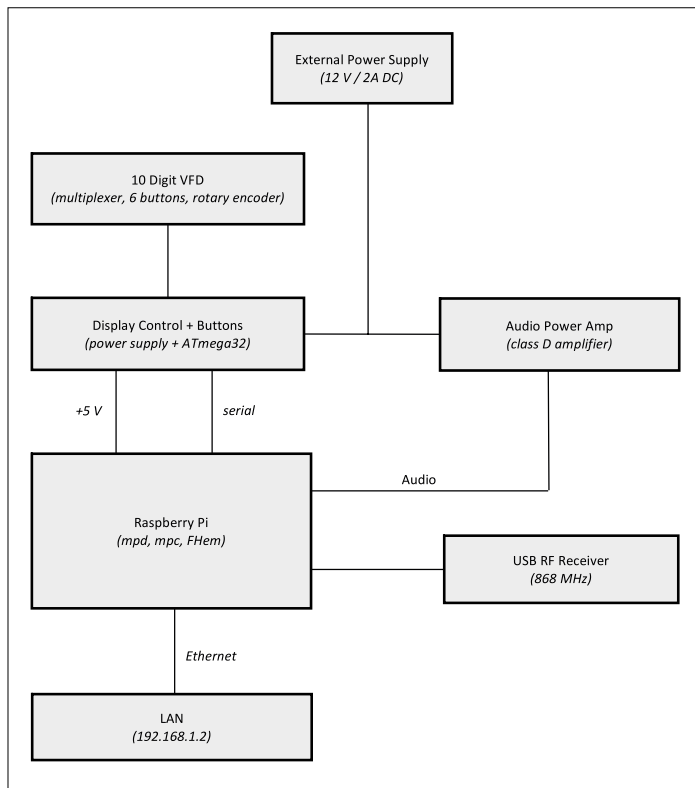


Figure 6. Block diagram of the complete kitchen radio.

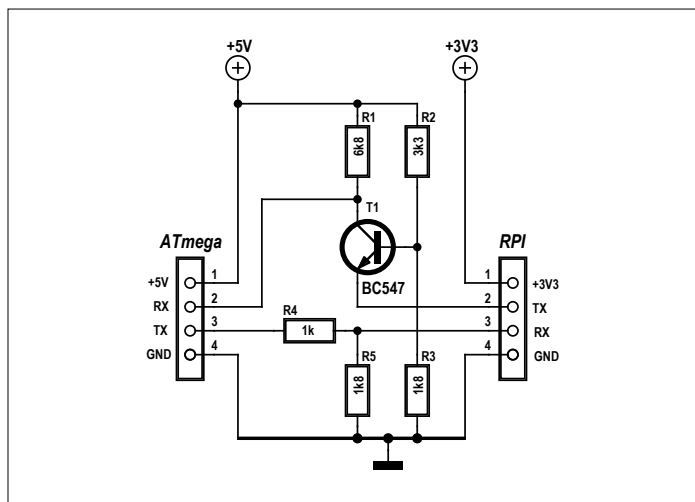


Figure 7. Level shifter for the serial interface, bridging between the 5 V world of the ATmega and the 3.3 V world of the Raspberry Pi.

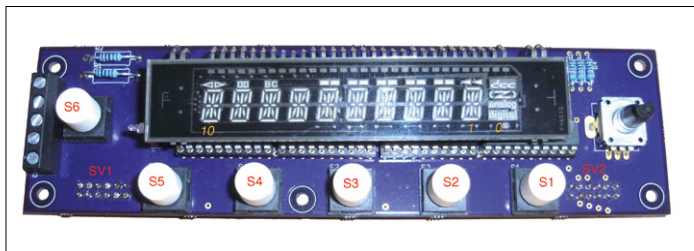


Figure 8. 'Muxer' board with display, drivers, latches, buttons and rotary encoder.

```
typedef struct {
    uint8_t digits[MAX_CHARS]; //one for each
                                //character position
    union TSymbolssymbols;     //annunciators on/off
} TDispData;
```

The function *mux_VFD()* is called every 5 ms under control of a timer. It reads the characters from *digits*, creates a bit pattern and writes the character to the display. This routine is rather more involved, and interested readers can find the commented code in the download.

The variable *display*, of type *TDispData*, together with the functions of **Listing 3**, provides the interface to the display module.

Character output

The following example command shows how to write the character 'A' to the fourth position of the display.

```
display.digits[4] = 65
```

The function call *setVFDDisplayData(4, 'A')* does the same thing, but is somewhat easier to read, and includes argument bounds checking. The functions to control the display are in the files *fv651.c* and *fv651.h* in the download. The complete source code is contained in the file *Software_1.zip* in the form of an Atmel Studio 6.2 project.

With this firmware we have complete control over the VFD, and our beautiful display (including some buttons for the user interface) is ready for the kitchen. Now we can turn our attention to the heart of the project: the Internet radio, based on a Raspberry Pi. That will be the main topic of the next part of this series.

Internet radio in outline

The block diagram in **Figure 6** gives an overview of the structure of the kitchen radio. The Raspberry Pi board is connected to the ATmega over a serial interface. Unfortunately, the Raspberry Pi runs at 3.3 V while the ATmega uses 5 V levels, and so we need a level-shifting circuit (**Figure 7**). In the transmit direction from the ATmega a simple voltage divider, formed by R4 and R5, suffices. From Raspberry Pi to ATmega the signal level needs to be increased. The solution here is very simple: the voltage divider formed by R2 and R3 holds the base of T1 at about 1.8 V, and the RX signal drives its emitter. If RX is high T1 turns off and its collector is pulled up to 5 V via resistor R1; if RX is low T1 turns on and a low level appears at its collector. Alternatively, the ATmega could be powered at 3.3 V: the level shifter could then be dispensed with, but the clock frequency of the microcontroller would have to be reduced to 12 MHz.

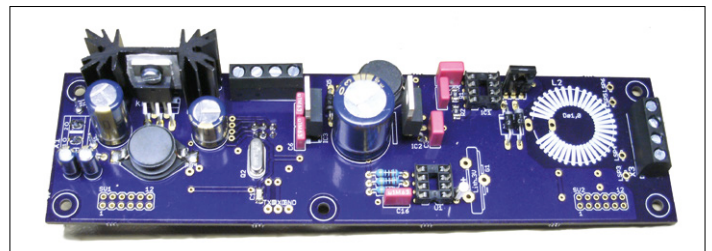


Figure 9. Partially-populated board with power supply and microcontroller.

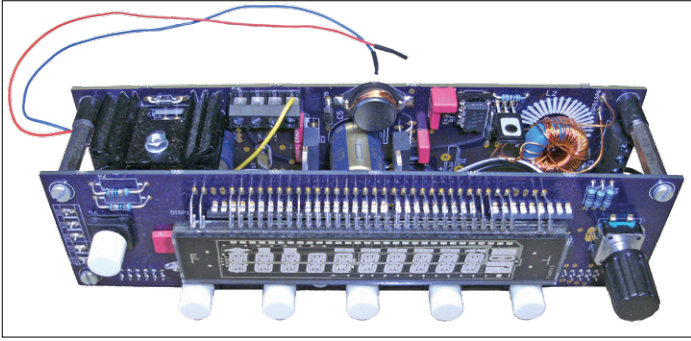


Figure 10. A two-board sandwich.

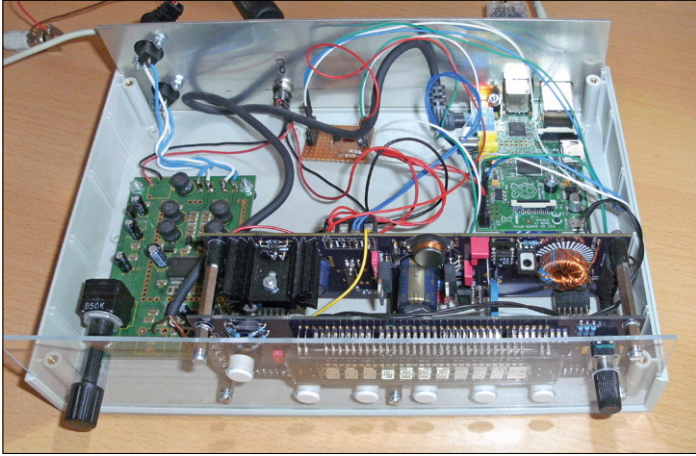


Figure 11. The assembled prototype in a clamshell enclosure, including the Raspberry Pi and the auxiliary circuits such as the level shifter.

Printed circuit boards and construction

The first printed circuit board designed by the author, called 'Muxer' (**Figure 8**), carries the display along with its drivers and latches, six buttons, and a rotary encoder with a push switch. The two headers SV1 and SV2 are fitted on the solder side. The drivers and latches are mounted under the display. The circuit diagram [4] also shows the port pins corresponding to the connections on SV1 and SV2.

The second board (**Figure 9**) contains the power supply and the ATmega32 microcontroller, and there is also a real-time

clock with battery back-up. The connections for the serial port used to connect the ATmega32 to the Raspberry Pi are brought out. After populating the boards and initial tests the two boards can be bolted together to form a compact module: see **Figure 10**. The module can then be mounted in a box along with the Raspberry Pi. **Figure 11** shows the prototype built into a clamshell enclosure. The board on the left converts the 12 V from the external power supply into a stable 5 V supply, which is also used to power the Raspberry Pi.

Coming up

Although the display module is a versatile unit with many applications, it is the Internet-connected kitchen radio that we will be focusing on in the next and final installment in this series. There we will describe the communication protocol and the software that will be running on the Raspberry Pi. ◀

(150720)

Internet Links

- [1] How vacuum fluorescent displays work: www.explainthatstuff.com/how-vacuum-fluorescent-displays-work.html
- [2] Guide to VFD operation: www.noritake-elec.com/display/vfd_operation.html
- [3] Heater power supply: www.kerrywong.com/2013/06/05/vfd-filament-driver-using-555/
- [4] Article downloads: www.elektormagazine.com/150720

Advertisement

About the Author

After training as a telephone engineer and studying electrical and communications engineering at technical college, Michael Busser spent 20 years at the German state telephone authority; later he worked on a range of projects at T-Systems, a German IT services and consulting company.

He then completed postgraduate study in vocational education at the technical university of Kaiserslautern in Germany, and for the last two years has been teaching electrical engineering and information technology at the BBS TGHS vocational college in Bad Kreuznach, Germany.

Feedback and questions are welcome via michael.busser@t-online.de.

Internet Radio with Fluorescent Display (2)

Raspberry Pi plus ATmega plus software

By **Michael Busser** (Germany)

The plan is to make a kitchen radio using a good, old-fashioned, tried-and-tested fluorescent display, but with the added twist of an Internet connection. In the first installment we looked at its functions, the hardware and how we drive the display. But, without software, the hardware is nothing: the ATmega in the display module needs firmware, and code is also needed to turn the Raspberry Pi into an Internet radio.

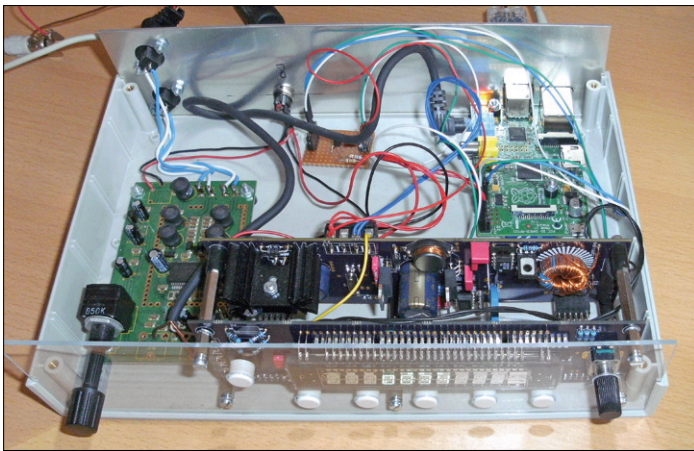


Figure 1. All the boards fitted into a clamshell enclosure and wired together.



Figure 2. Lid on, firmware in: the kitchen Internet radio in operation. The electronics is visible through the acrylic front panel.

If, with a little help from the circuit diagrams, board layouts and other documentation you have built the VFD module from the first installment of this series [1], you may have discovered that (even assuming you have put everything together correctly) not a lot happens when you power it up. Even with all the printed circuit boards installed in the box and wired together (**Figure 1**), and with the lid closed (**Figure 2**) the display will still not light up as shown in the photographs. The reason behind this is that the microcontroller in the display module needs firmware, so that it can understand the control commands sent to it. This firmware and, last but not least, the code for the Raspberry Pi, forms the topic of this second and final installment.

Communication protocol

If two computer systems want to exchange data packets with one another it is necessary to agree beforehand on how the packets are to be interpreted: a protocol has to be defined. The homebrew protocol used here was also designed for use in a home automation system using a two-wire bus, and so has a few features that we will not be needing in this case. Nevertheless, its structure follows the same lines as other similar protocols. The firmware for the ATmega in the display unit was written entirely within the AtmelStudio development environment (see screenshot in **Figure 3**).

Table 1 shows the general structure of a data packet. A message of type *TDataBuf* is passed to the transport layer. The

Table 1. Data packet structure.

STX <i>start of transmission</i>	Message Container	ETX <i>end of transmission</i>
0x02	data	0x03

Table 2. The components of a message.

STX	SRC	DST	MT	MODE	Payload (0 to 15 bytes)	CS	ETX
-----	-----	-----	----	------	-------------------------	----	-----

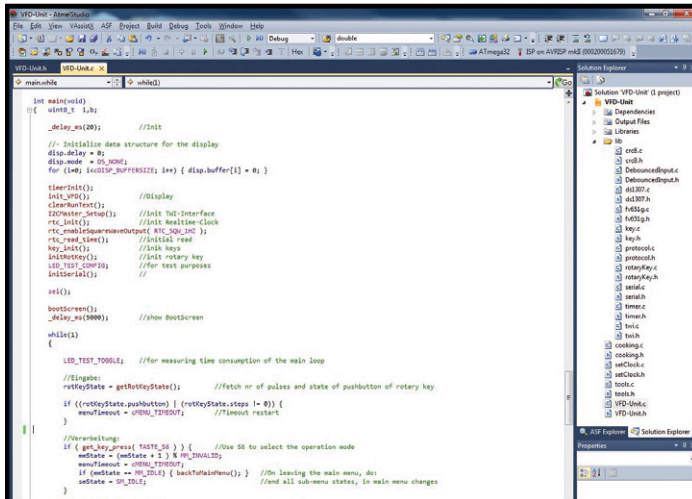


Figure 3. Screenshot of the AtmelStudio development environment showing the ATmega firmware for the display module.

transport layer starts by sending an STX character to mark the beginning of a transmission; an ETX character is sent at the end of each transmission. The STX and ETX characters should not appear within the message itself: if they do, they should be escaped by prefixing them with the data link escape (DLE, 0x10) character. The receiver removes any escape character from the data stream and appends the immediately following character to the partially-assembled message. If it is desired to transmit the escape character itself, it is necessary to send

Table 3. Message codes.

Field	Meaning
SRC	Source address: 0 to 255 except for the values 2 and 3.
DST	Destination address: 0 to 255 except for the values 2 and 3.
MT	Message type. The following are defined: 0x10 StatusReq 0x11 StatusResp 0x12 DisplayReq 0x13 DisplayResp 0x14 Key_Req 0x15 Key_Resp 0x16 Time_Req 0x17 Time_Resp Note that codes 0x02 and 0x03 may not be used; other codes are available for future expansion.
MODE	Bit Meaning 0...3: Length of payload in bytes: 0 to 15 4: 0/1 = checksum absent/ checksum present 5: 0/1 = acknowledgement with ACK not required/acknowledgement required 6: 0 (reserved) 7: always 1 Note: values 0x02 and 0x03 may not be used. To ensure this bit 7 is specified to always be 1.
CS	Message checksum.

two DLE characters. The following functions provide the software interface (see *serial.c* and *serial.h*):

```
void initSerial(void);
void sendMsg(TDataBuf msg);
uint8_t peekMsg(TDataBuf *msg);
```

The function *initSerial()* should be called once when the main program starts up. It initializes the UART and enables various interrupts. The baud rate can be adjusted by modifying the definition

```
#define UART_BAUD_RATE 9600UL
```

in *serial.h*. The other serial port parameters are eight data bits, no parity, and one stop bit ('8N1'). The function *sendMsg()* takes a message of type *TDataBuf*, copies it to the internal transmit buffer (if it is free) and then returns immediately to the caller. Transmission and reception proper all happen under interrupt control in the background. As soon as a valid message is received it is stored in one of the two receive buffers. A call to the function *peekMsg()* will then return a '1' and the valid message will be written to *msg*.

The escape mechanism is handled transparently within these functions. There is one transmit buffer, but two receive buffers that are used alternately: this allows one message to be processed while the next one is being received.

The code module is structured independently of the format of the messages being sent, which is defined in *protocol.c* and *protocol.h*. Two-wire bus systems such as RS-485 require an extra control signal which switches the hardware between transmit and receive modes. The default mode is 'receive', transmit mode being enabled only for the duration of a message transmission. This is the purpose of the two definitions

```
#define SENDER_OFF
#define SENDER_ON
```

in *serial.h*. The receiver runs continuously, and so will pick up any message that is sent: this can be used to detect a bus collision. The only function we need in *protocol.h* is *ownAdr()*, which returns our own address on the bus.

Table 2 shows the individual components of a packet and **Table 3** describes their meanings. **Listing 1** shows the structure of a message as a C 'struct'; the 'union' is a construct which describes how the *TDatagramm* structure and the *TBuf* byte array can overlay one another at the same memory addresses. This allows easy reading and writing of the message as a simple array of bytes, which makes the receive and transmit code more straightforward.

ATmega communication

The following message types are defined.

```
#define MT_STATUS_REQ 0x10
#define MT_STATUS_RESP MT_STATUS_REQ + 1
```



```
#define MT_DISPLAY_REQ 0x12
#define MT_DISPLAY_RESP MT_DISPLAY_REQ + 1

#define MT_KEY_REQ 0x14
#define MT_KEY_RESP MT_KEY_REQ + 1

#define MT_TIME_REQ 0x16
#define MT_TIME_RESP MT_TIME_REQ + 1
```

The *DISPLAY_REQ* message allows the Raspberry Pi to display something on the VFD. *KEY_REQ* informs the Pi when a button has been pressed on the display module. *TIME_REQ* lets the ATmega obtain the current time from the Pi, which in turn obtains the time over the Internet using NTP.

The ATmega can send and receive messages over its serial interface in a specified format using simple calls to the relevant functions in *protocol.c* (**Listing 2**). Received messages are processed in a loop in the main program (**Listing 3**), which continuously checks whether a new message is available and then decides how to handle it.

Let us look at an example of how these messages are used. Every 60 minutes the ATmega sends a message of type *MT_TIME_REQ* to the Raspberry Pi by calling the function *sendMsgTimeReq()*:

```
if (timerFlags.flags.bMin) {
    doNTP_Sync--;
    if (doNTP_Sync == 0) {
        doNTP_Sync = NTP_SYNC;
        sendMsgTimeReq(); // fetch
        current time from Raspberry Pi
    }
    timerClearMin();
}
```

The Raspberry Pi replies with a message of type *MT_TIME_RESP*, which is then processed in the receive loop in the main program.

The source code is too long to reproduce here in its entirety, but it can be found in the file *Software_2.zip* in the download available at [1].

Raspberry Pi software

We use version dated 09.09.2014 of the 'wheezy-raspbian' operating system on the Raspberry Pi. Detailed instructions for obtaining an image, copying it to an SD card, and carrying out basic config-

Listing 1. Message structure.

```
typedef struct {                //construction of a message
    uint8_t    src;             //source address (sender)
    uint8_t    dst;             //destination address (receiver)
    uint8_t    mt;              //message type
    TMode       mode;            //mode byte, describes structure of message
    TPayload    Payload;         //usage depends on application level
    uint8_t    cs;              //checksum, e.g. crc-8
} TDatagramm;

typedef union {
    TBuf        Buf;            //access via byte array
    TDatagramm  Datagramm;      //access via structure
} TDataBuf;
```

Listing 2. Example message function.

```
void sendMsgTimeReq(void) {
    TDataBuf msg;

    msgClear( &msg );
    msg.Datagramm.src = ownAdr();
    msg.Datagramm.dst = masterAdr();
    msg.Datagramm.mt = MT_TIME_REQ;
    msg.Datagramm.mode.bAckReq = NO_ACK_REQUIRED;
    msg.Datagramm.mode.bCRC = NO_CHECKSUM; //WITH_CHECKSUM;
    msg.Datagramm.mode.count = 0; //don't count header / trailer bytes
    msgPrepare( &msg );
    sendMsg( msg );
}
```

Listing 3. Processing a message.

```
if (peekMsg(&recMsg) == 1) { //if message received: evaluate
    switch (recMsg.Datagramm.mt) {
        case MT_STATUS_REQ:    break;
        case MT_STATUS_RESP:   break;
        case MT_DISPLAY_REQ: i=0;
            b=1;
            for (i=0; i < cDISP_BUFFERSIZE; i++) {
                if (recMsg.Datagramm.Payload[i] == 0) {b = 0;}
                if (b == 1) {
                    disp.buffer[i] = recMsg.Datagramm.Payload[i];
                } else {
                    disp.buffer[i] = 0;
                }
            }
            disp.delay          = cDISP_SHOWTEXT;
            disp.mode           = DS_TEXT;
            showText();
            break;
        case MT_DISPLAY_RESP:   break;
        case MT_KEY_REQ:        break;
        case MT_KEY_RESP:       break;
        case MT_TIME_REQ:       break;
        case MT_TIME_RESP:      setTimeOnMsgReq( &recMsg );    break;
    }
}
```

Listing 4. The *handleMsg()* function.

```

void handleMsg(int fd, TDataBuf *msg) {
    if (msg->Datagramm.mode.bCRC) {
        syslog( LOG_NOTICE, "checksum ignored");
    }
    if (msg->Datagramm.dst == masterAdr()) {
        switch (msg->Datagramm.mt) {
            case MT_STATUS_REQ:    onStatusReq(fd, msg);    break;
            case MT_DISPLAY_REQ:  onDisplayReq(fd, msg);    break;
            case MT_KEY_REQ:      onKeyReq(fd, msg);        break;
            case MT_TIME_REQ:     onTimeReq(fd, msg);       break;
            default:              syslog(LOG_ERR, "unknown message dropped");
                                dumpMsg(msg);
        }
    } else {
        syslog( LOG_NOTICE, "message not for me, ignoring it");
    }
}

```

Listing 5. The *onTimeReq()* function.

```

void onTimeReq(int fd, TDataBuf *msg) {
    TDataBuf reply;
    msgClear(&reply);
    reply.Datagramm.src = masterAdr();
    reply.Datagramm.dst = msg->Datagramm.src;
    getTimeResp(&reply);
    //dumpMsg(&reply);
    frame_Send(fd, &(reply.Buf[0]), reply.Datagramm.mode.count + MSG_
HEADER_LEN, reply.Datagramm.cs );
    syslog( LOG_NOTICE, "onTimeReq");
}

```

uration tasks can be found at [2].

At [3] there is a guide to configuring the Raspberry Pi as an Internet radio using *mpd* ('Music Player Daemon') and *mpc* ('Music Player Client'), and there are many other similar guides available. A search for 'Raspberry Pi mpd' will magically lead

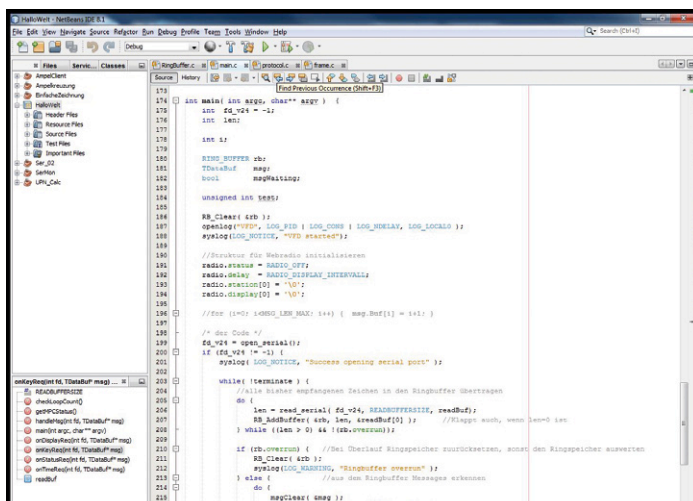


Figure 4. Screenshot of the NetBeans IDE. With the software shown we can turn the Raspberry Pi into an Internet radio.

you to everything you need to know. A development environment is required to get our first C program for the Raspberry Pi up and running (although in principle the command line can also be used). My environment of choice is NetBeans, which being written in Java also runs on my desktop computer. **Figure 4** shows a screenshot of the NetBeans IDE in action. A step-by-step guide to installing the NetBeans environment on the Raspberry Pi can be found at [4].

Once these hurdles are overcome, it is possible to write C programs for the Raspberry Pi conveniently under Windows, and then compile and debug them on the Pi itself.

Fortunately C code written for the ATmega will run without significant changes on the Raspberry Pi, as long as you avoid using any of the special functions and modules which it does not support.

The program running on the Raspberry Pi copies messages to a syslog server, which helps with debugging. The syslog server must be configured on the Raspberry Pi. The function *dumpMsg()* takes one message and outputs it to the syslog in human-readable form. Inside the function *main()* in the Raspberry Pi program we read all the characters that have arrived on the serial

port since we last looked and store them in a circular buffer:

```

do {
    len = read_serial(fd_v24, READBUFFERSIZE, readBuf);
    RB_AddBuffer(&rb, len, &readBuf[0]);
    //works even if len=0
} while ((len > 0) && !(rb.overrun));

```

The function *frame_checkFrame()* is called at regular intervals. It checks whether there is a complete message in the circular buffer. If so, it returns '1', and if not, it returns '0'. The parameters *startindex* and *length* give the starting offset and length of the message.

A call to the function *frame_get()* returns the corresponding message in *msg*. This is then processed by the function *handleMsg()* (see **Listing 4**). So if, for example, a message of type *MT_TIME_RESP* is received, the function *handleMsg()* will pass it on to the function *onTimeReq()* (see **Listing 5**). Here a reply to the request message is constructed and then immediately sent back to the sender using the function *frame_Send()*. The NTP client must be configured on the Raspberry Pi. System time is determined by a call to the function *localtime()* inside the function *getTimeResp()*. This function is also responsible for

assembling the reply message in the *TDataBuf* data structure.

Internet radio

The Internet radio functionality itself is implemented using the *mpc* music player client. This allows you to construct playlists from Internet radio stations: the details are set out at [5]. Also, the command *mpc help* is available from the command line. Our specially-written C program controls *mpc* by passing it the necessary options via a call to *system()*, in just the same way as it would be used from the command line. The excerpt from the function *onKeyReq()* shown in **Listing 6** is also worth a look. This function is called whenever a button on the display module is pressed.

If *mpc* is called without parameters then the first line of its output is the current station and the name of the currently-playing program. The function *getMPCStatus()* obtains this information at regular intervals and then sends the station name to the ATmega using an *MT_DISPLAY_REQ* message. The current station name then replaces the current date on the display.

Each radio stream must be stored in a playlist before it can be used by *mpd*. The playlists are stored in the directory */var/lib/mpd/playlists*.

Start-up

As it stands the program on the Raspberry Pi needs to be started by hand, as development is not yet complete. One slight niggle with using NetBeans is that it creates very long directory paths. If you follow the guide to installing NetBeans as the root user exactly, executable code ends up in the directory

```
/root/.netbeans/remote/192.168.1.24/mib2-Windows-
x86_64/D/Projekte/RaspberryPi/RPi/HalloWelt/dist/
Debug/GNU-Linux
```

The part of the path printed in bold changes according to the environment, including the IP address of the Windows PC,

Listing 6. Excerpt from the function *onKeyReq()*.

```
void onKeyReq(int fd, TDataBuf *msg) {
    uint8_t keys = msg->Datagramm.Payload[0];    // define variable / check button
    if (keys & KEY_S0) {syslog( LOG_NOTICE, "onKeyReq: S0");}
    if (keys & KEY_S1) {terminate = true; syslog( LOG_NOTICE, "onKeyReq: S1");}
    if (keys & KEY_S2) {syslog( LOG_NOTICE, "onKeyReq: S2");}
    if (keys & KEY_S3) {syslog( LOG_NOTICE, "onKeyReq: S3");}
    if (keys & KEY_S4) {                                //next station
        if (radio.status == RADIO_ON) {
            system("mpc next");}
        syslog( LOG_NOTICE, "onKeyReq: S4");
    }
    if (keys & KEY_S5) {                                //radio on / off
        if (radio.status == RADIO_OFF) {                //if radio off
            system("mpc play");                        //play last station
            radio.status = RADIO_ON;
        } else                                         //radio off
            system("mpc stop");
            radio.status = RADIO_OFF;
    }
    syslog(LOG_NOTICE, "onKeyReq: S5");
}
```

its hostname (here 'mib2') and the path to the project under Windows (here 'D:/Projekte/...').

However, it is easy to make this program execute automatically when the Raspberry Pi is booted up. One description of how to do this can be found at [6]. If the program is started from a shell, it remains dependent on this shell; if the shell is closed, the program will also terminate. A simple '&' character after the end of the command avoids this:

```
./hallowelt &
```

This makes the program execute in the background, with the shell generating a job number and a process ID. These are needed if you wish to use a command such as

```
fg %1
```

to bring the process back into the foreground.

The program is now ready to run on the Raspberry Pi. Extensions are possible and indeed ideas for extensions are welcomed: you can import the program directly into NetBeans from the file *HalloWelt.zip*. We hope you have fun experimenting! ◀

(160207)

Internet Links

- [1] Software download and first installment: <https://www.elektormagazine.com/150720>
- [2] Raspberry Pi operating system installation: <https://www.raspberrypi.org/documentation/installation/installing-images/>
- [3] Raspberry Pi as an Internet radio: <https://www.youtube.com/watch?v=jf3M1RVpQbM> (audio in German)
- [4] NetBeans for the Raspberry Pi: <http://bit.ly/2aBZ14A>
- [5] Internet radio tutorial: <https://www.youtube.com/watch?v=jf3M1RVpQbM> (audio in German)
- [6] Automatically run a program on boot: <http://pi.bek.no/autostartProgramOnBoot/>

IoT Gateway and Wireless Nodes

Part 1: The hardware



Sometimes you want to have specific functions in a home automation system that are not available with any commercial product. That's why the author decided to develop his own system, which features wireless communication between the various nodes and the central gateway. The gateway uses MQTT to send measurement data to an OpenHAB server, which processes and displays the data. His project has now been copied and extended by other users. In a series of two articles, the author describes the key elements of his system. More detailed information and the corresponding software are available for free on the Internet.

By **Hennie Spaninks** (Netherlands)

Home automation systems are enticing. There's something magical about having your lamps go on and off by themselves, a heating system that knows when you are at home, or receiving a message when someone is at your door. There are quite a few systems currently available, but most of them suffer from the following shortcomings:

- They often work in only one direction – you can send commands, but you never know whether they actually arrive. That's not a big issue when you're at home sitting next to a lamp and reading, but when you are on vacation in a foreign country it's nice to know that everything is working as it should.
- They often can only be operated using a specific app, and each vendor has their own app with protocols that do not work with other systems. As a result, you end up with a whole bunch of apps on your smartphone.

- The nice thing about home automation is that you can program them to send commands in response to sensor readings. However, because most commercially available systems are not compatible with each other, you have to use an external service (such as *IfThisThenThat*) to get a working system.
- Many wireless systems are based on Wi-Fi. In modern residential buildings made from reinforced concrete, the range of Wi-Fi signals is very limited.
- Security is not always optimal, so there is a distinct chance that your neighbors will be able to control your lamps.
- All in all, there are plenty of good reasons to roll up your sleeves and build your own home automation system.

Overview

For the end nodes, which means the sensors and actuators, you ideally want reasonably affordable devices which are also energy efficient. The central controller obviously has to support

standard network protocols, including TCP/IP. That is why the author opted for a two-tier system consisting of end nodes, a gateway and a central control unit. The overall system is shown schematically in **Figure 1**.

- For communication with the end nodes, the system uses wireless duplex links operating at a relatively low frequency. In Europe the available frequency bands for this purpose are 433 MHz and 868 MHz. The 433 MHz band is fairly crowded, so we opted for 868 MHz. Transceivers for this band which can be controlled over an SPI bus are available from HopeRF. We chose the version with the highest transmit power: the RFM69HW. This module has integrated hardware encryption, so security is not an issue.
- The RFM69 does not have a TCP/IP stack, so fixed-size data blocks of 66 bytes are exchanged over the radio link. A gateway is necessary to convert the data from the wireless network to TCP/IP and vice versa.
- To distribute the data from the sources (sensor nodes) to the recipients (for example, a smartphone), we opted for a different standard protocol which runs over TCP/IP: MQTT. It is a messaging protocol designed to send short messages to several recipients in a simple manner. Because MQTT is a standard protocol, data streams from our home automation system can also be used in other MQTT-based systems. MQTT requires a server to act as the message center, which is called the broker. A recipient (such as a smartphone) can subscribe to a data stream, after which it receives messages for that stream through a push service. To implement our MQTT broker we use Mosquitto [1], an open source MQTT broker implementation which is available for several platforms (including Raspberry Pi).
- Our home automation system should preferably be compatible with devices from other vendors. This means that it must also be able to communicate using other protocols. For this we opted for a system based on OpenHAB [2]. This open source software can convert messages in various protocols to a universal message stream. For example, you can use it to control Philips Hue lamps.
- OpenHAB has a standard app (for Android and iOS) which communicates with your own OpenHAB system over the Internet using a secure connection. That means you don't have to worry about configuring firewalls or encryption for the link between your smartphone and your home automation system. OpenHAB can be accessed on the local area network through a Web browser.
- OpenHAB has an underlying database which makes it easy to make charts of measured values. Creating rules and scripts to automate tasks is easy in OpenHAB. OpenHAB also has external interfaces for sending alerts via email or the OpenHAB app.
- We use readily available and relatively low-cost components: Arduino as the controller for the wireless nodes, and a Raspberry Pi as the platform for Mosquitto and OpenHAB.

Protocol The protocol for the end nodes

Various functions are defined for the end nodes. They include sensors for temperature and humidity, switches, PIR motion detectors for input, and relays and LC displays for output. End

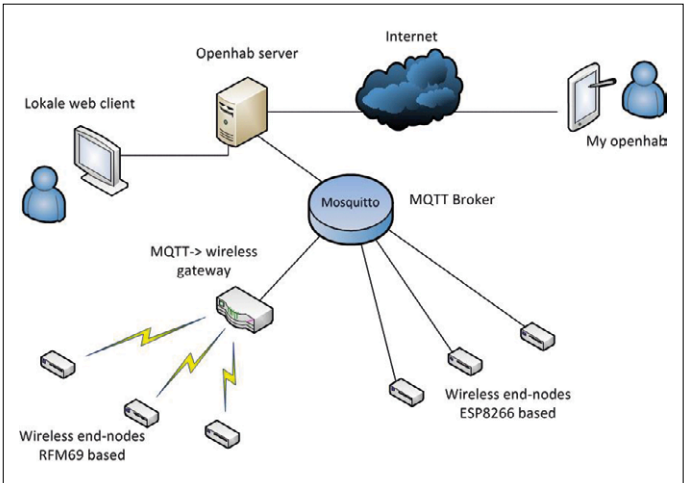


Figure 1. The architecture of the home automation system.

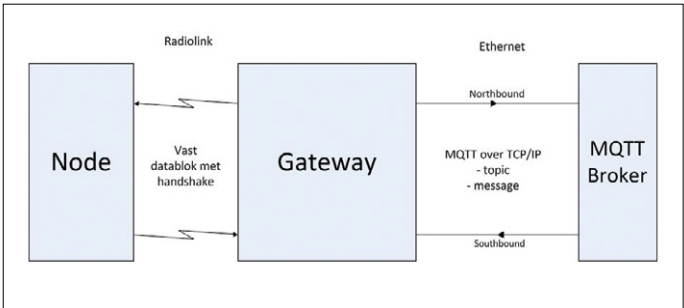


Figure 2. The various data streams passing through the gateway.

Table 1. Device identifiers.	
DevID range	Function
0-15	System devices
16-31	Binary output (relay, lamp)
32-39	Integer output (dimmer, PWM)
40-47	Binary input (switch, PIR module)
48-63	Decimal input (temperature, humidity)
64-71	Integer input (keypad, switch)
72	String transfer (LCD display, RFID reader)
73-90	Reserved for future use
90-99	Error messages

nodes can autonomously transmit measurement data at regular intervals. This measurement data can also be sent on request in response to read commands.

The data flow through the gateway is shown in **Figure 2**. End nodes can be programmed for various functions. To ensure that the gateway handles communications with each end node correctly, there is a fixed allocation of functions to devices. Based on the device identifier (DevID), the gateway knows how to handle the incoming data. **Table 1** shows an overview of the DevIDs recognized by the gateway.

Devices 0 to 15 provide system functions. They are described in **Table 2**.

The error messages generated by the gateway are listed in **Table 3**.

Table 2. System functions.

DevID	Name	RW	Function
00	Uptime	R	Minutes since node start
01	TxInterval	RW	Transmit interval in seconds (0 = no periodic transmission)
02	RSSI	R	Radio signal field strength
03	Version	R	Software version of the end node
04	Voltage	R	Battery voltage
05	ACK	RW	Flag for acknowledging sent commands
06	Toggle	RW	Flag for toggle function of switch on end node
07	Timer	RW	Flag for timer function of pushbutton on end node
08	Btnpress	RW	Flag for sending button press message
09	TXreply	R	Number of repeats necessary on the radio link

Table 3. Error messages.

Error ID	Name	Description
90	Link error	The radio link is interrupted.
91	Syntax error	There is a syntax error in the MQTT message.
92	Invalid device	The addressed DevID is not present in the end node.
99	Wakeup	Message sent when the node starts up.

Table 4. MQTT examples.

Topic	Message	Description
home/rfm_gw/sb/node02/dev16	ON	Switches on the LED at node 02.
home/rfm_gw/sb/node02/dev16	READ	Queries the status of the LED at node 02.
home/rfm_gw/sb/node03/dev01	300	Sets the transmit interval of node 03 to 5 minutes.
home/rfm_gw/sb/node03/dev01	0	Disables periodic transmission from node 03.
home/rfm_gw/sb/node18/dev48	READ	Reads the temperature from node 18.
home/rfm_gw/sb/node05/dev02	READ	Reads the radio signal field strength at node 05.
home/rfm_gw/sb/node05/dev03	READ	Reads the software version of node 05.

MQTT messages

The MQTT protocol works on the basis of subscribing to topics (see [3]). After a recipient has subscribed to a topic, the broker ensures that all messages related to that topic are sent to the recipient.

The format of the MQTT topic subscription is

`home/rfm_gw/direction/nodeID/deviceID`

Here `direction` specifies the direction of the data flow: “nb”

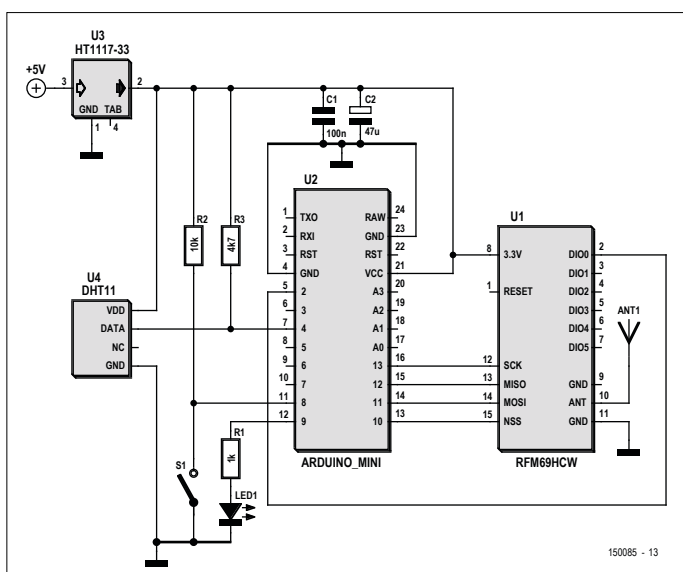


Figure 3. Schematic diagram of an end node.

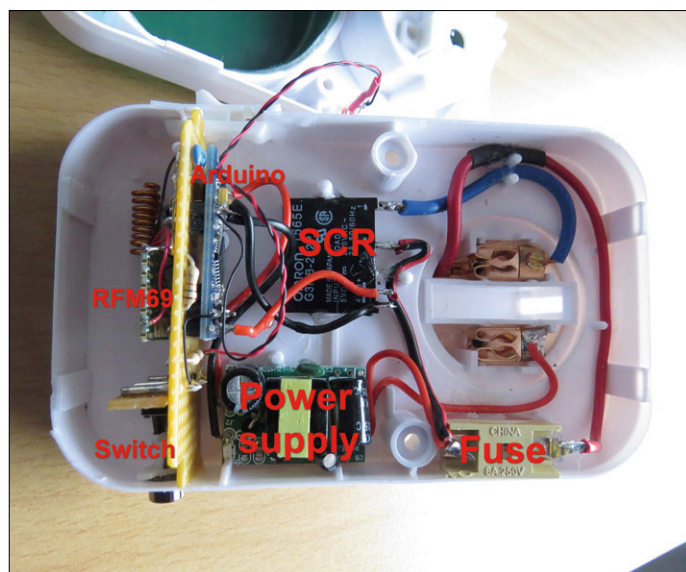


Figure 4. A DIG node in a plug-in timer housing.

(northbound) means from the node to OpenHAB (sensor data), while “sb” (southbound) means from OpenHAB to the node (commands).

- **nodeID** is the node identifier. Each node has a unique ID (address), which is determined when the code is compiled. The gateway always has node ID 01.
- **deviceID** is the function identifier, as previously described.

The MQTT message (payload) depends on the device and the direction.

- **Southbound:** The message contains commands for the node (ON, OFF, READ) or settings for the node, such as the number of seconds for the transmit interval.
- **Northbound:** The message contains the value of the parameter belonging to the DevID.

The gateway subscribes to all southbound messages for all nodes in the network by means of the following wildcard topic:

`home/rfm_gw/sb/#`

Of course, OpenHAB needs to receive all messages from the nodes, so it subscribes to the appropriate topics:

`home/rfm_gw/nb/#`

Table 4 shows a number of examples of MQTT messages.

End nodes

The end nodes are based on Arduino boards. The following conditions and constraints are taken into account in the design:

- The RFM69 module has an operating voltage of 3.3 V, and the maximum permissible voltage on its inputs is 3.9 V. To keep things simple, we use an Arduino that also operates from 3.3 V. We opted for the Arduino Pro Mini, which operates from 3.3 V and can be connected to the RFM69 without level conversion. You can also use the Arduino Buono R3, which can be switched to 3.3 V.
- The RFM69 module briefly draws a significant amount current (130 mA) when transmitting, so the supply needs to have sufficient capacity.
- The Arduino and the RFM69 communicate with each other over the SPI bus. The standard Arduino pins are used for that purpose.
- The RFM69 may not be operated without an antenna. A piece of wire 8.6 cm long gives excellent results in most cases.

The schematic diagram of one of the end nodes is shown in **Figure 3**. This node is equipped with a DHT11 temperature and humidity sensor connected to pin 4 of the Arduino. A pushbutton is connected to digital I/O pin 8, and an LED is connected to pin 9.

The RFM69 module is connected to the Arduino over the SPI bus (SCK, MOSI, MISO and NSS). The 3.3 V supply voltage is provided by an AMS1117 voltage regulator, with enough decoupling capacitors to suppress current spikes. The software loaded into the Arduino determines the function of the end node. The following end node devices have been developed so far:

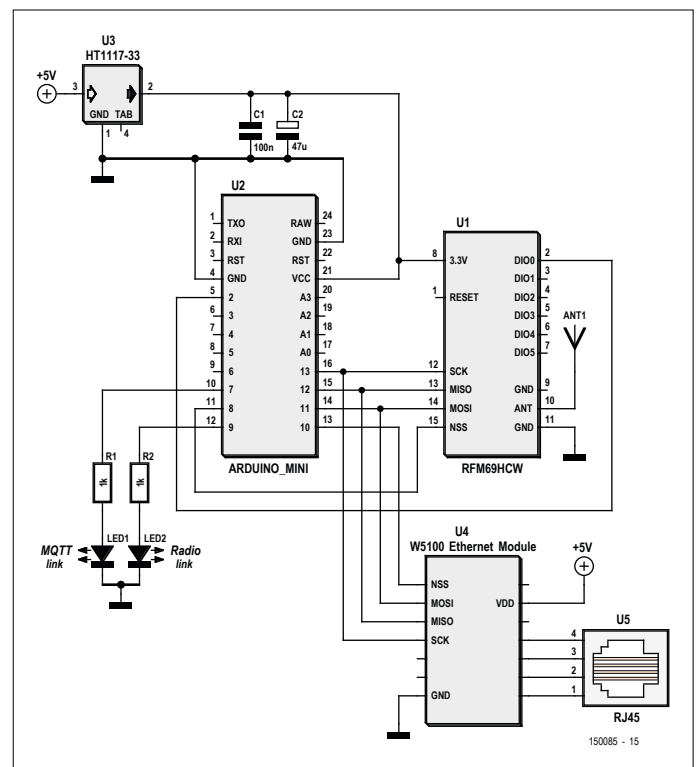


Figure 5. Schematic diagram of the gateway.

- DHT is a node with sensors for temperature and humidity. This node has a digital output and a pushbutton.
- DIG is a simplified version of the DHT node. It only has a pushbutton and a digital output.
- RFID is a node with an RFID reader. When an RFID device is detected, the node sends the RFID identifier to OpenHAB.
- LCD is a node with a liquid crystal display. Text strings can be sent from OpenHAB to the display.
- RC is a node equipped with a 433 MHz transmitter module. It can be used to operate switches in the Dutch *KlikAanKlikUit* home automation system [4].

The software can easily be adapted to add or modify functions. The construction of a node is strongly dependent on its function. For example, the case of an inexpensive mechanical timer switch can be used to house a DIG node. An example of this is shown in **Figure 4**.

The gateway

The gateway consists of an Arduino, an RFM69 module and an W5100 Ethernet module. **Figure 5** shows the schematic diagram. The gateway also uses an Arduino operating at 3.3 V. On the gateway the SPI bus is shared by the RFM69 and W5100 modules. The SCK, MOSI and MISO signals are connected to both devices in parallel. Different Slave Select (SS) signals are used to select the individual devices on the bus: pin 8 for the RFM69 and pin 10 for the Ethernet module.

An AMS1117 also provides the 3.3 V supply voltage for the gateway. Two LEDs are provided to indicate the status of the gateway.

A separate module is used for the Ethernet connection. If you use an Arduino Buono board, you can use a Wiznet Ethernet

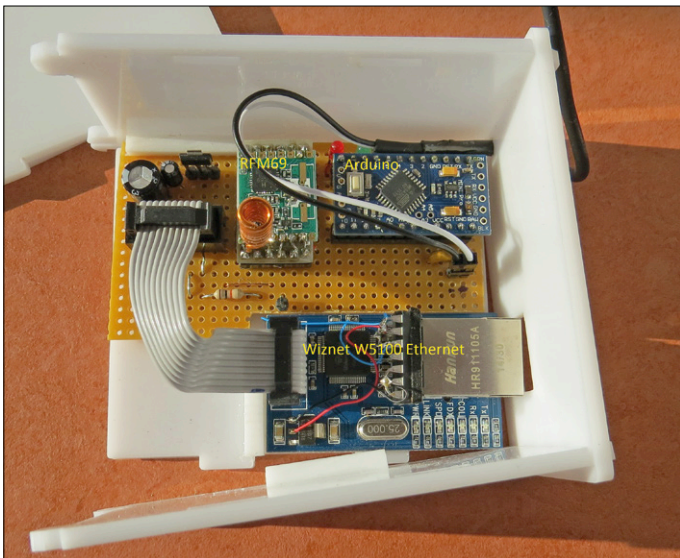


Figure 6. The author's fully assembled gateway.

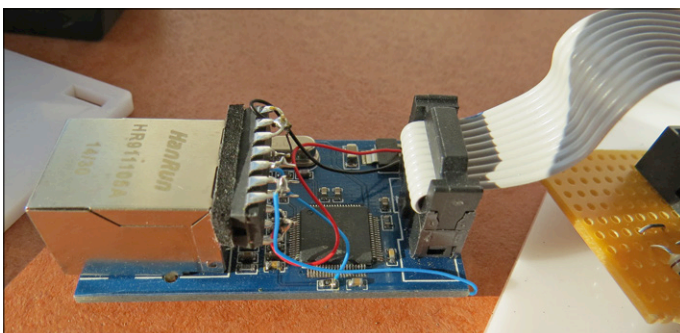


Figure 7. The 4011 is attached to the housing of the Ethernet connector and generates the SEN signal for the W5100 from the CS signal.

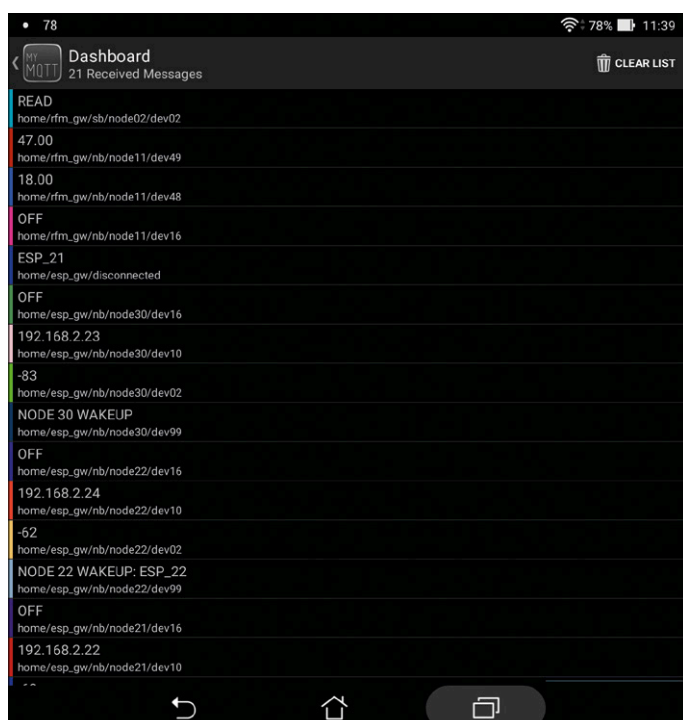


Figure 8. This screenshot shows some examples of MQTT messages.

shield instead. Only one gateway is needed for the entire system. For this reason, the author built the gateway on a prototyping board (**Figure 6**).

There are two things you have to pay attention to when building the gateway, as described below.

W5100 bug

The hardware of the W5100 is not directly suitable for sharing an SPI bus with other devices. When an SPI bus device is not using the bus, it is supposed to release the signal lines so that they can be used by other devices. The original W5100 is not designed to do this. The author solved this problem by using an inverter to generate the SEN control signal from the CS signal (see [5] for more details). The necessary inverter is integrated in newer versions of the W5100 module. The module which the author bought on eBay did not include this feature, so the inverter had to be added externally. As can be seen in **Figure 7**, a type 4011 IC was used for this. There you can also see that it is easy to connect the SEN signal line via a pull-up resistor on the circuit board. Remember to connect the unused inputs of the 4011 to ground.

Short-circuit via ICSP header

If you use a standard-format Arduino with the W5100 shield, the Arduino board is connected to the 5 V supply voltage of the Ethernet shield through the ICSP header. That pulls up the 3.3 V supply voltage line on the Arduino, which can result in damage to the RFM69. This can be avoided by cutting off the VCC pin of the ICSP header or bending it aside so that it does not make contact.

To be continued

In the following article we will describe the software for the gateway and the nodes, as well as the configuration of Mosquitto and OpenHAB. If you can't wait, you can already download the software at [6]. You must have Mosquitto available in order to test the gateway. It's also handy to have an MQTT client for testing. The MyMQTT app is a good choice for Android, and on a Windows system you can use Chrome Lens or MQTT.FX. **Figure 8** gives an impression of the messages you can expect to see.

This project has now been built by several other people. Extensions, issues and experience are discussed and shared on the forum site [7]. ◀

(150085)

Web Links

- [1] <https://mosquitto.org>
- [2] www.openhab.org
- [3] www.elektormagazine.com/tags/journey-into-the-cloud
- [4] www.klikaanklikuit.nl (Dutch!)
- [5] <http://john.crouchley.com/blog/archives/662>
- [6] <https://github.com/computourist/RFM69-MQTT-client>
- [7] <http://homeautomation.proboards.com/board/2/openhab-rfm69-based-arduino>

IoT Gateway and Wireless Nodes (2)

Part 2: the software



The author has looked in vain for a suitable commercial home automation system. Since he couldn't find anything to his liking, he decided to develop his own system. Various 'end nodes' (sensors and actuators) communicate via an 868 MHz radio link with a central gateway. The gateway uses MQTT to send the measurement values to an OpenHAB server, which processes them into a form suitable for display [1]. The construction of the hardware was extensively covered in the previous article [2]. This article concerns itself with the software that makes everything tick.

By **Hennie Spaninks** (The Netherlands)

As we mentioned in the previous part, we used the software on the following platforms. The Arduino sketches were written by the author himself, who naturally made use of the various Open-Source Libraries available.

End node

The Arduino sketch for the end node: this code controls the radio link, reads the sensors and controls the outputs.

Gateway

The Arduino sketch for the gateway: this code controls the radio link and decodes the messages. An MQTT client is used to exchange messages with the broker.

Home automation server (Raspberry Pi)

The Raspberry Pi is used to run the MQTT broker (Mosquitto) as well as the OpenHAB home automation controller.

A standard Arduino development environment was used to compile the Arduino code, extended with a number of libraries, which are described later on. It's best to use the most powerful version of the Raspberry Pi (RPI 3), although the Raspberry Pi 2 would probably cope as well. We use Raspbian for the OS.

End node

The code for the various end nodes can be found at [3]. The core of the code is the same for all of the end nodes. For this description we'll focus on the DHT end node [4]. This node has a digital input (a push-button) on input 8, a digital output (relay/led) at pin 9 and a humidity/temperature sensor at pin 4. When the push-button has been pressed, the node can carry out one of two local functions:

- Start a timer and set the output high. Reset the output when the timer has finished. The timer period is set via device 7 (see Table 2 in part 1). A value of '0' disables the timer function.
- Toggle the output whenever the push-button is pressed.

This enables you to manually turn the output on and off locally. You have to set device 6 to 'ON' to enable this function.

Completely separately from these two functions, a status message is transmitted via device 40 whenever the level changes at the digital input (pin 8). The values of the humidity/temperature sensor can be read via device 48/49. Regular transmissions are also possible.

The code makes use of three libraries:

- RFM69 library by Felix Ruso [5].
- SPI library to control the SPI bus.
- DHT library to read the DHT11 sensor.

The code begins with a number of configuration parameters (from line 60 onwards):

```
#define NODEID 2          // unique node ID within
                           the closed network
#define GATEWAYID 1       // node ID of the Gateway
                           is always 1
#define NETWORKID 100     // network ID of the
                           network
#define ENCRYPTKEY "xxxxxxxxxxxxxxxx" // 16-char
                           encryption key; same as on Gateway!
// #define DEBUG          //
                           uncomment for debugging
#define VERSION "DHT V2.2" // this
                           value can be queried as device 3
```

The NODEID gives the end node a unique address within the radio network. This ID is also used in the MQTT topic to address the required node. The nodeID of the gateway is always '1'. The radio link from the end node is always made with the gateway. There is therefore never any traffic directly between individual end nodes.

The NETWORKID is used to create closed networks. Only those nodes that are within the same network can then communicate with each other. We're using a closed network in this project, which means that the network ID has to be identical in the gateway and all of the end nodes.

The ENCRYPTKEY is the key used to encrypt the radio traffic. This key consists of 16 characters and has to be identical for all nodes in the network.

The DEBUG parameter is a switch to enable the debugging mode. When this parameter has not been defined (by setting this line as a comment by preceding it with '//'), no debug information will be generated. When the '/' is removed, DEBUG becomes active and status messages will be transmitted via the serial port.

Several other parameters are required for the RFM69 module to function correctly:

```
#define FREQUENCY RF69_868MHZ
#define IS_RFM69HW // uncomment only for RFM69HW!
#define ACK_TIME 50 // max # of ms to wait for an
                    ack
```

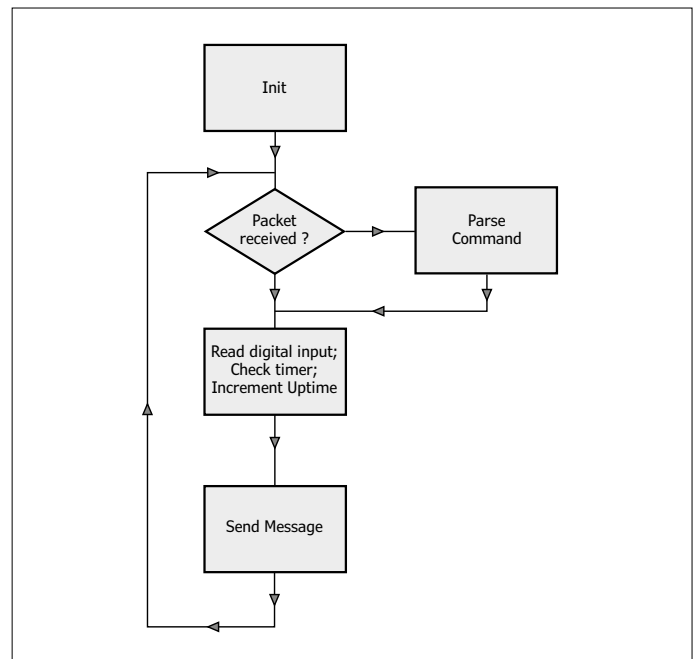


Figure 1. Program flow of the end node.

FREQUENCY is the frequency used, which in Europe is either 433 MHz or 868 MHz. IS_RFM69HW states that the High Power version is used. ACK_TIME is the maximum time that the module waits for an acknowledge after the transmission of a message.

From line 78 onwards are several definitions for the DHT sensor, the pin connections used, and for some variables. The comments in the source code give more details of the function of these variables. Following line 135 is the initialization of the program (setup()), where the output pin is configured and the radio module is initialized.

The program flow is shown in the flowchart in **Figure 1**.

First of all, (in line 167) we check if the RFM69 module has received a radio packet. If so, the function parseCmd is called. A certain send flag is set, depending on the deviceID and the R/W command in the received data block. Next, we check if the push-button status has changed or if the timer period has completed. After that, the uptime counter is incremented, if required. When the interval for periodic transmissions has expired, the send flags of these parameters are set.

And finally in line 235, the function sendMsg is called. In here, a number of radio packets are prepared depending on which send flags are active. The send flag is then reset and the function transmits the packet by calling txRadio().

If the temperature or humidity is required (device 48 or 49), the value is obtained from the DHT module in line 478 or 485.

Gateway

The gateway has the following functions:

- Receive an MQTT message, decode it, create a radio packet and transmit it.
- Receive a radio message, decode it, create an MQTT message and publish it.
- Receive, decode and answer requests to the gateway itself, such as uptime, version information, etc.

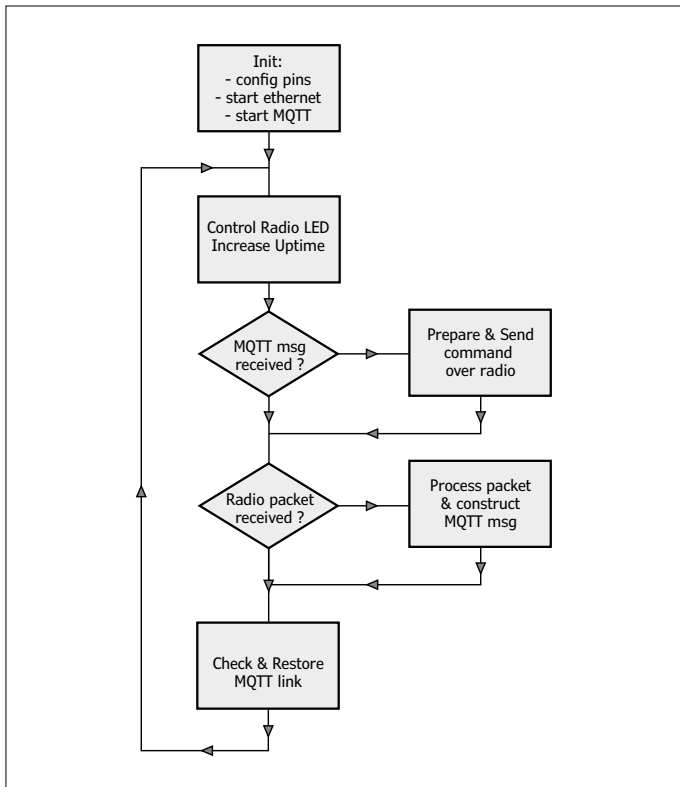


Figure 2. Program flow of the gateway.

The gateway has a fixed Ethernet connection and obtains its network information via DHCP. The gateway has two leds. The radio led is lit when the radio connection is active. The MQTT led is lit when there is a connection to Mosquitto.

Since the gateway program takes up a lot of memory, the debug function has been split into two parts: one part to debug the radio connection and another part to debug the connection to the MQTT broker. Only one of these can be active, otherwise the program won't compile.

The description of the program is for the latest version (2.4 as we go to press), which can be found at [6].

The gateway sketch makes use of the following libraries:

- *RFM69*
- *SPI*
- *Ethernet*
- *PubSubClient* [7]

There are several differences in the gateway sketch, compared to the standard settings:

- The SPI bus is used by two devices in the gateway. Both devices use '10' for the default device address. We selected devicecode 8 for the RFM module and 10 for the Ethernet module. This configuration is selected in line 80 of the gateway code.
- The RFM module uses interrupts to indicate when a radio packet has been received. As long as the Ethernet driver (W5100.h) is active on the SPI bus, these interrupts won't be processed correctly, which causes the gateway to hang. We make use of a solution suggested by Martin Harazinov

and make some changes to the file w5100.h to prevent this from happening (see [8]). In the file w5100.h (which can be found in `../Documents/Arduino/libraries/Ethernet/src/utility`) the lines from line 342 have `cli()` and `sei()` statements added to them.

```

#else
inline static void initSS() { DDRB |= _BV(2); };
inline static void setSS() { cli(); PORTB &= ~_BV(2); };
inline static void resetSS() { PORTB |= _BV(2); sei(); };
#endif

```

This stops any interrupts from occurring while the Ethernet traffic goes over the SPI bus.

The code in the gateway sketch begins with a declaration section that looks very similar to the one for the end node. All the parameters for the radio link are set here. The following parameters for the Ethernet are also set here:

- *Ip*, for the IP address of the gateway. This is a fallback, since the gateway uses DHCP and therefore gets a dynamically allocated address.
- *Mac*, for the MAC address of the Ethernet connection. Note that this must be unique within an IP subnet, so be careful when you have other Arduinos in the same network.
- *Mqtt_server*, the address of the Mosquitto server (we'll look at this later).

The program flow of the gateway is shown in the flowchart in **Figure 2**. The initialization starts at line 135. The output pins for both leds are configured, the Ethernet connection is started and then the connection to the MQTT broker is started. The MQTT client is initialized in line 129. A subscription is requested that's defined in the function `mqtt_subs` (line 382). The topic of the subscription is `home/rfm_gw/sb/#` (line 121), which causes all SouthBound messages to be received. When MQTT messages are received, the variable `mqttToSend` is set to TRUE. The main program starts with a test to see if the radio led has to be turned off, and the uptime of the gateway is incremented.

In line 223 the variable `mqttToSend` is used to check if there are any messages that need to be sent over the radio network. This is done by the function `sendMsg` (line 248). If necessary, the last message is retransmitted every half second until the reception of the packet has been acknowledged by the end node. This is repeated a maximum of five times.

A check is then made in line 225 to see if a radio packet has been received from an end node. If that's the case, the function `processPacket` (line 288) is called. The MQTT topic from the gateway is always NorthBound and is made up from the `nodeID` and the `deviceID` and then stored in the string `buff_topic` in line 303. Depending on the `deviceID`, the type of data in the radio packet is determined (integer, float, status or string), as is the required format of the MQTT message. The string `buff_mess` will eventually contain the MQTT message. Finally, a check is made in line 227 to see if the MQTT link is

still active. If this isn't the case, the link is restarted every two seconds in line 230. The MQTT led is turned on in line 235 if the connection has been made.

OpenHAB

The architecture of OpenHAB is shown in **Figure 3**. Openhab has been designed around an event bus. This bus is used to exchange events between the input and output interfaces and the rest of OpenHAB. This configuration permits the use of several IO interfaces ('bindings') at the same time. You can make specific bindings for various protocols. At this point in time bindings are available for smart meters, Sonos, Philips Hue lamps and also for the Nest thermostat and Tesla electric vehicles, amongst others. The page at [9] has a complete overview of the available bindings.

Installation

For the installation we'll use a Raspberry Pi 3 with a 16 GB SD card. We use version 2 of OpenHAB. The installation of OpenHAB is easiest if you use OpenHABian. This is an image with a minimal linux system in combination with installation scripts. This image can be downloaded from [10]. A suitable tool (WinDiskImager [11], for example) can be used to write this image to an SD card. This is put in the Raspberry Pi, and the installation will start immediately after booting. Be aware that this will take quite some time.

As part of the installation process *Samba* is also installed. This makes the configuration directories of OpenHAB accessible via the network, so you can make simple configuration changes. Once the installation is complete, we can install Mosquitto using an installation tool:

```
pi@openHABianPi: sudo openhabian-config
```

If all is well, you should see the window as shown in **Figure 4**. Option 22 will install Mosquitto. We must ensure that the Raspberry Pi always has the same IP address. This is done by modifying the file */etc/network/interfaces*. You should set address and gateway to the required values for your configuration, for example:

```
auto eth0
iface eth0 inet static
    address 192.168.2.7
    netmask 255.255.255.0
    gateway 192.168.2.254
```

The new settings will become active after a reboot.

Configuration

To get OpenHAB to work, we need to configure the MQTT binding. The file that needs to be modified is on the RPi at */etc/openhab2/services/mqtt.cfg*. The broker URL is the only parameter that needs to be changed, as shown here:

```
# Define your MQTT broker connections here for use
# in the MQTT Binding or MQTT
# Persistence bundles. Replace <broker> with an ID
# you choose.
```

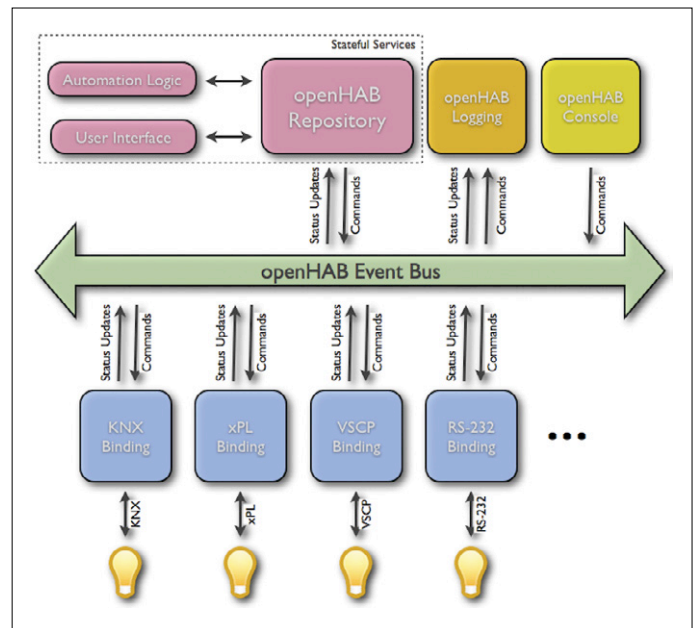


Figure 3. OpenHAB architecture (source: OpenHAB.org).

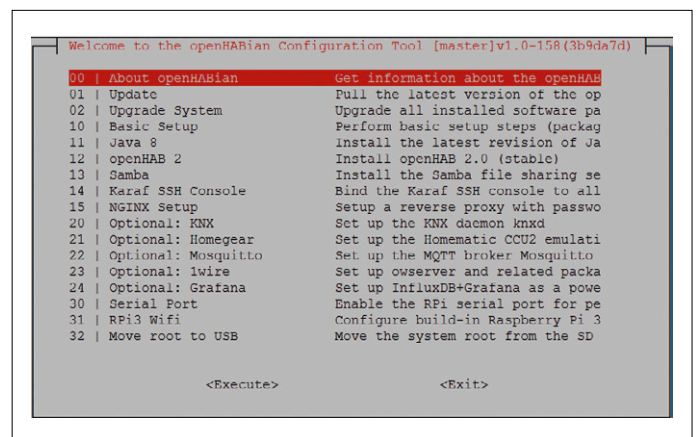


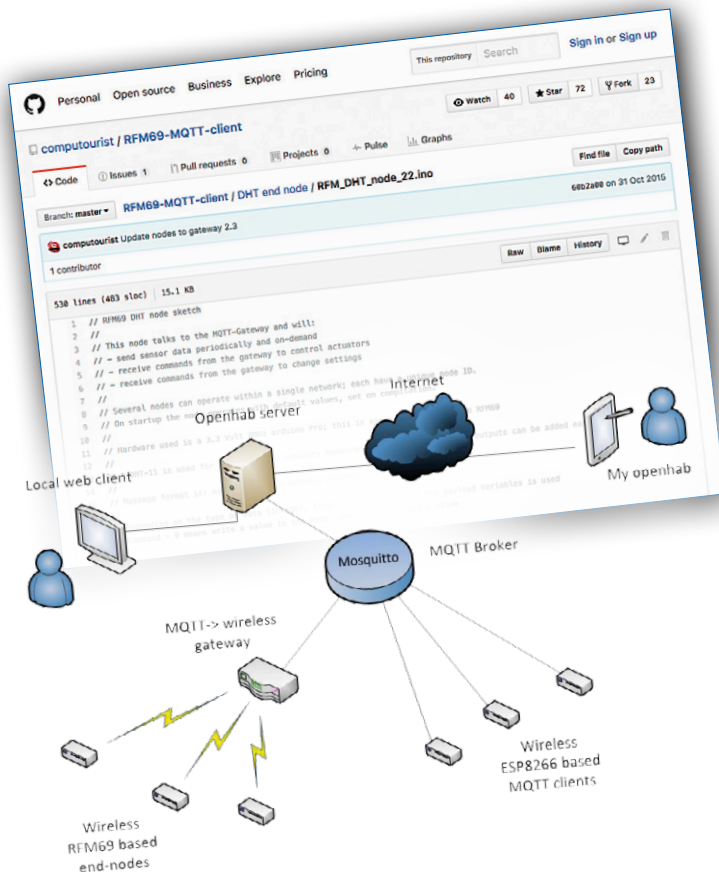
Figure 4. OpenHABian configuration.

```
# URL to the MQTT broker, e.g. tcp://localhost:1883
# or ssl://localhost:8883
mosquitto.url=tcp://localhost:1883
```

This way we point at localhost for the Mosquitto server. Note that 'mosquitto' is the name (ID) of the broker in this configuration. In the item configuration (see further down in the text) we refer to this ID to establish the communications. Port 1883 is used by default for MQTT traffic.

Openhab uses three configuration files to describe the system:

- *Items* in the directory */etc/openhab2/items*; this contains the definitions of the parameters used, and their connection to the IO.
- *Sitemap* in the directory */etc/openhab2/sitemaps*; this describes the layout of the user interface.
- *Rules* in the directory */etc/openhab2/rules*; this defines which



```
Text item=HUM2
Text item=RSSI2 }
```

The first line states the name of this configuration (home), so we can load the correct sitemap when we access OpenHAB with a browser or an App. This is followed with a definition to help display the output of node 2, and then a frame where the temperature, humidity and field strength are shown. And finally, we create a file called *home.rules* in */etc/openhab2/rules* with the following lines:

```
//
// refresh rules
//

rule "refresh RSSI2" // periodically refresh
value for signal strength
when
    Time cron "0 0/1 * * * ?"
then
    sendCommand(getRSSI2, "READ")
end
```

The cron command is used to transmit a 'READ' MQTT message every minute to device 2 on node 2, to obtain the field strength.

It's now time to try out a few things. Open a browser and navigate to:

<http://192.168.xx.xx:8080/basicui/app?sitemap=home>

You should of course put in the IP address that you're using. The result will be shown in a window like that in **Figure 5**. It's possible that not all of the values are available yet (a '-' will be shown in that case), but after a while the first refresh from the node will occur, when that will be resolved.

Debugging

What to do when things don't appear to work properly.

Is MQTT working?

When the MQTT connection between the gateway and the broker is working, the MQTT led will be lit. This means that messages from the gateway are received by Mosquitto. The next step is to install an MQTT test client (MQTT.fx [13] or MQTTLens [14] from the Chrome web store). MQTT.FX has a *broker status* tab, which can be used to request information about Mosquitto. When you subscribe to the topic *home/rfm_gw/#*, you should see some messages coming through. If all is well, that

operations need to be carried out (scripting).

As an example we'll show the configuration for a DHT end node with nodeID 2, which has a node configuration where the temperature, humidity and output state are periodically transmitted. We create a file called *home.items* in the item directory (*/etc/openhab2/items*), with the contents shown in **Listing 1**. The OpenHAB wiki [12] explains how the configuration of an item should be created. We then create a file called *home.sitemap* in the sitemap directory (*/etc/openhab2/sitemaps*), with the contents:

```
sitemap home label="MyHome"

Frame label="Controls" {
    Switch item=OUT2 label="Myhome lamp"
    icon="light" }

Frame label="Sensoren" {
    Text item=TEMP2
```

Listing 1. OpenHAB configuration for a temperature/humidity sensor end node.

```
Number RSSI2 "RSSI [%d dBm]" {mqtt="<[mosquitto:home/rfm_gw/nb/node02/dev02:state:default]"}
String getRSSI2 "get RSSI" {mqtt=">[mosquitto:home/rfm_gw/sb/node02/dev02:command:*.default]"}
Number TEMP2 "Temperatuur [%1f °C]" {mqtt="<[mosquitto:home/rfm_gw/nb/node02/dev48:state:default]"}
Number HUM2 "Vochtigheid [%1f %]" {mqtt="<[mosquitto:home/rfm_gw/nb/node02/dev49:state:default]"}
Switch OUT2 "zolder node" <PushBtn> {mqtt=">[mosquitto:home/rfm_gw/sb/node02/dev16:command:ON:ON],>[mosquitto:home/rfm_gw/sb/node02/dev16:command:OFF:OFF]"}

```

will also include messages from the end node. If that isn't the case, it's time to compile the gateway or the node with the DEBUG option, and to connect a PC to the serial port. Start PUTTY [15], select the required COM port and set its speed to 115200 bps. The output of the node/gateway will indicate if there is a connection and what values are sent. If you use the MQTT client to send commands to the node (for example, send *READ* to */home/rfm_gw/node02/dev03*), you should be able to find out where the data stream fails.

Is OpenHAB working?

If the MQTT connection is working satisfactorily, the next logical step is to look at OpenHAB. A good way of doing this is to inspect the log files. The command

```
> tail -f /var/log/openhab2/events.log
```

will show the last ten lines of the event log. Any updates received from a node or a user interface should be visible here. You should also have a look at */var/log/openhab2/openhab.log*. This is where the OpenHAB system events are logged.

And next...

For the first enhancement you may well be looking at accessing the system via the Internet. This is something you can do via *myopenhab*. Openhab has an app that communicates with your own OpenHAB system via a secure gateway over the Internet. More details can be found at [1].

Openhab also has the facility to show graphs of the variation of measurements over time. This requires that a local database is set up first, along with the selection of measurements that should be stored. This can be done locally, although external



Figure 5. Browser window.

databases are also possible. This is described in more detail in [16].

The node described here measures the temperature and humidity, and can turn on a digital output. There are many more possibilities when you adapt the electronics in the node, and the software. On Github [3] they show how to connect an LCD and an RFID reader [17]. A node is also described that can control the Dutch range of "klik-aan-klik-uit" switches via OpenHAB. An interesting variation to this project is one based on the ESP8266. This node uses WiFi to create the connection with the MQTT broker and Openhab. The design for this is described at [18].

It's worth having a look on the forum [19] for other applications, where various users have presented designs for the control of PIR detectors, roller shutters, level sensors and more. ◀

(160318)

Web Links

- [1] www.myopenhab.org
- [2] www.elektormagazine.com/150085
- [3] <https://github.com/computourist/RFM69-MQTT-client>
- [4] https://github.com/computourist/RFM69-MQTT-client/blob/master/DHT%20end%20node/RFM_DHT_node_22.ino
- [5] <https://github.com/LowPowerLab>
- [6] https://github.com/computourist/RFM69-MQTT-client/blob/master/Gateway_2.4/RFM_MQTT_GW_24.ino
- [7] <https://github.com/knolleary/pubsubclient>
- [8] <http://harizanov.com/2012/04/rfm12b-and-arduino-ethernet-with-wiznet5100-chip>
- [9] <http://docs.openhab.org/addons/bindings.html>
- [10] <http://docs.openhab.org/installation/openhabian.html>
- [11] <https://sourceforge.net/projects/win32diskimager>
- [12] <https://github.com/openhab/openhab1-addons/wiki/Explanation-of-items>
- [13] <http://mqttfx.org>
- [14] <https://github.com/sandro-k/MQTTLensChromeApp>
- [15] www.putty.org
- [16] <http://docs.openhab.org/configuration/persistence.html>
- [17] <https://github.com/computourist/RFM69-MQTT-client>
- [18] <https://github.com/computourist/ESP8266-MQTT-client>
- [19] <http://homeautomation.proboards.com/board/2/openhab-rfm69-based-arduino>



welcome in your **ONLINE STORE**

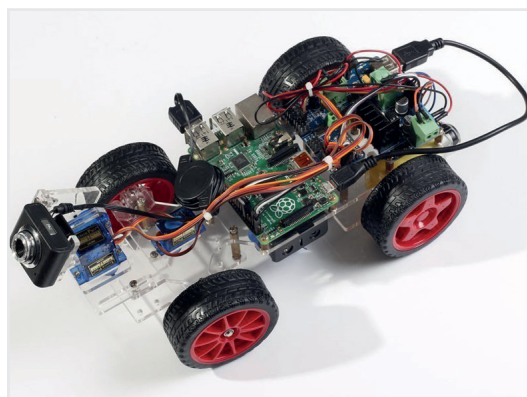
EDITOR'S CHOICE



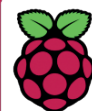
The Sunfounder Smart Video Car Kit for Raspberry Pi is very well done. As the name of the kit suggests, a (colour) video camera is included. Also, as the use of 'for' implies, the Raspberry Pi is not included, but a Wi-Fi dongle is. The 'Smart' refers to the clever software the RPi can execute. The parts fit perfectly, the manual is great and the software just works. The car is remotely controlled over Wi-Fi and out of the box (after fully assembling it first, of course) it has the functions 'move forward', 'move backward', 'go left', and 'go right' plus 'pan' and 'tilt' for the camera.

The remote control is a computer (Windows, Linux, Mac, other) with Python 2.7 installed and a web browser (and, of course, a Wi-Fi connection). There is also an Android app.

Clemens Valens
(Elektor Labs)



Read the full review at www.elektor.com/smart-video-car-review

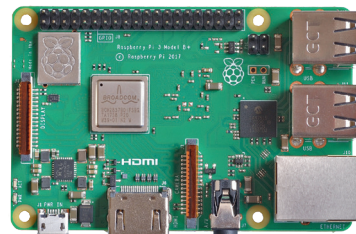


Raspberry Pi

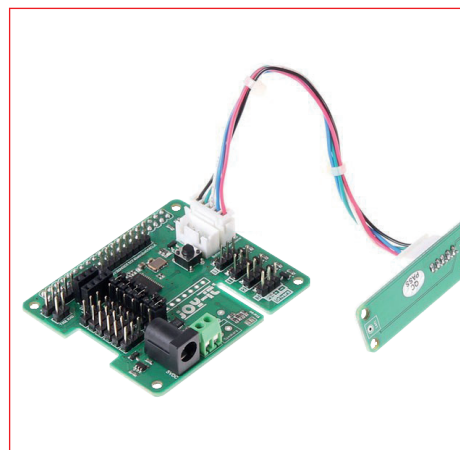
APPROVED RESELLER

RPi Bestsellers

1. Raspberry Pi 3 (model B+) www.elektor.com/rpi3b-plus
2. Raspberry Pi 3 Basic to Advanced Projects www.elektor.com/rpi-projects-book
3. Raspberry Pi 3B+ Starter Kit www.elektor.com/rpi-starter-kit
4. pi-top 2 (DIY Laptop Kit) www.elektor.com/pi-top-v2
5. Talking Pi www.elektor.com/talking-pi
6. Official 7" Touch Display for RPi www.elektor.com/7-inch-rpi-display
7. Raspberry Pi Zero WH www.elektor.com/rpi-zero-wh



Talking Pi

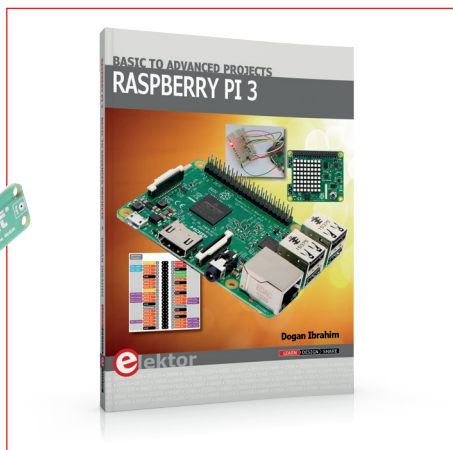


Talking Pi is an intelligent, universal open source speech control assistant for the Raspberry Pi. The extension module is compatible with Google Home / AIY project. Simply use voice commands to control home lighting, switch your power outlets on and off or activate your coffee machine when you open the front door! There are hardly limits to your imagination.

 member price: £27.95 • €31.46 • US \$37

www.elektor.com/talking-pi

Raspberry Pi 3 Basic to Advanced Projects



This book is about the Raspberry Pi 3 computer and its use in various control and monitoring applications. The nice feature of this book is that it covers many Raspberry Pi 3 based hardware projects using the latest hardware modules such as the Sense HAT, Swiss Pi, MotoPi, Camera module, and many other state of the art analog and digital sensors. All the projects in the book are based on the Python programming language and they have been fully tested.

 member price: £27.95 • €31.46 • US \$37

www.elektor.com/rpi-projects-book

Raspberry Pi Desktop Case



The Raspberry Pi Desktop is a desktop computer kit based on Raspberry Pi 2/3. It includes a case and an expansion board that can turn a Raspberry Pi into a real desktop PC. It provides an intelligent and safe power controller, a real-time clock, and a high capacity Solid State Drive (SSD) expansion card socket for additional storage. Connect the new RPi Desktop to a display through an HDMI interface and enjoy an authentic PC experience!

 member price: £47.95 • €53.96 • US \$64

www.elektor.com/rpi-desktop-case



Raspberry Pi 3B+ Starter Kit

RPi 3B+ included!

This special Raspberry Pi 3 B+ Starter Kit includes everything to get started right away with the world's most popular mini computer.

Kit contents:

- Raspberry Pi 3 (Model B+) with pre-mounted heat sinks
- Official Micro-USB Power Supply (5.1 V, 2.5 A) for Raspberry Pi 3 B(+) – black
- Official Case for Raspberry Pi 3 B(+), 2 and B+ (grey-black)
- High Speed HDMI Cable (1 m)
- Patch Cable Cat.5e (2 m)
- microSD Card (16 GB, Class 10) with SD Adapter (pre-installed with NOOBS)



MEMBER PRICE: £71.95 • €80.96 • US \$95

www.elektor.com/rpi-starter-kit

pi-top 2 (DIY Laptop Kit)

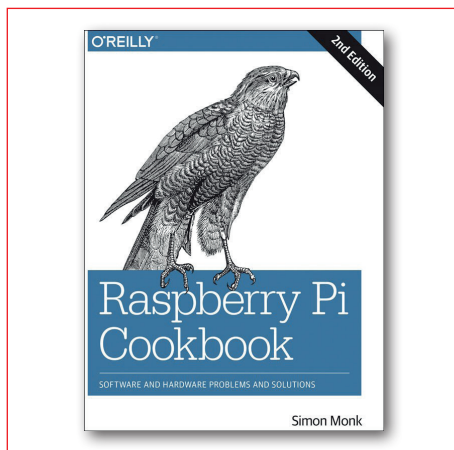


The new pi-top 2 turns a Raspberry Pi into a modular laptop. Learn to code, create your own devices, and take your knowledge to the next level. Inspired by makers, educators and inventors who love the original pi-top, this new version comes as a killer, new generation of pi-top laptops: 1920 x 1080 screen resolution, keyboard & trackpad, battery (up to 8h) and Polaris operating system. The included inventor's kit comprises several components including pi-topPROTO+.

member price: £237.95 • €269.96 • US \$316

www.elektor.com/pi-top-v2

Raspberry Pi Cookbook



The second edition of this popular cookbook provides more than 240 hands-on recipes for running this tiny low-cost computer with Linux, programming it with Python, and hooking up sensors, motors, and other hardware—including Arduino and the Internet of Things. Prolific hacker and author Simon Monk also teaches basic principles to help you use new technologies with Raspberry Pi as its ecosystem continues to develop.

member price: £25.95 • €28.76 • US \$34

www.elektor.com/rpi-cookbook

Motor Control Projects with Arduino and RPi



This book is aimed at students, hobbyists, and anyone else interested in developing microcontroller based projects using the Arduino Uno or the Raspberry Pi Zero W. The (tested) projects in the book cover the standard DC motors, stepper motors, servo motors, and mobile robots. Full program listings of all the projects as well as the detailed program descriptions are given in the book. Users should be able to use the projects as they are presented, or modify them to suit to their own needs.

member price: £26.95 • €29.66 • US \$35

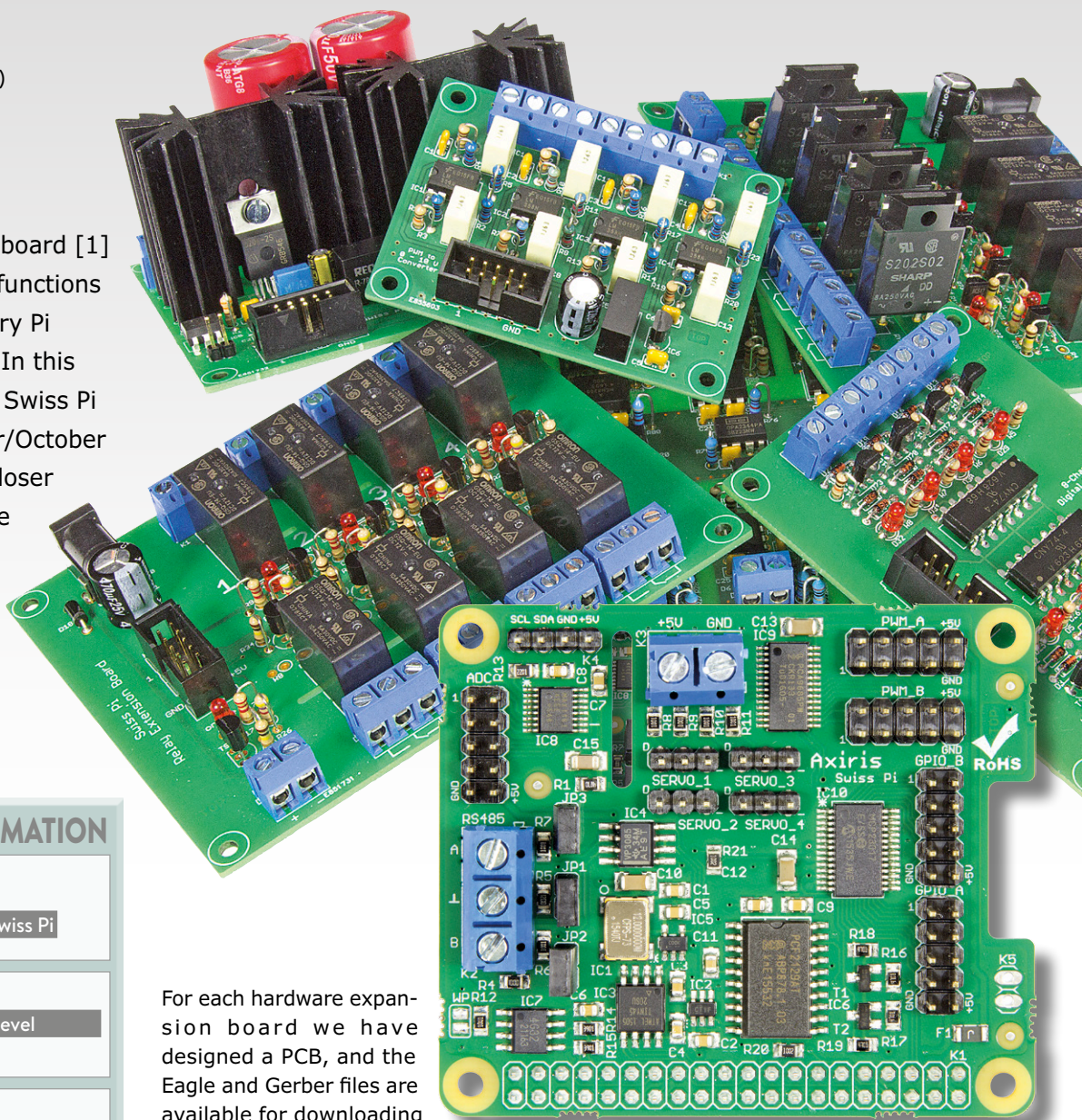
www.elektor.com/motor-control

Swiss Pi Extensions

Hardware extensions

By **Peter S'heeren**
and **Ilse Joostens** (Belgium)

The Swiss Pi expansion board [1] provides a lot of useful functions for the popular Raspberry Pi single-board computer. In this second follow-up to the Swiss Pi article in the September/October 2016 issue, we take a closer look at several hardware extensions developed specifically for the Swiss Pi board.



PROJECT INFORMATION



Microcontrollers

Raspberry Pi Swiss Pi



entry level

→ intermediate level

expert level



0.5 to 1.5 hours per board



Soldering station
with suitable solder tip



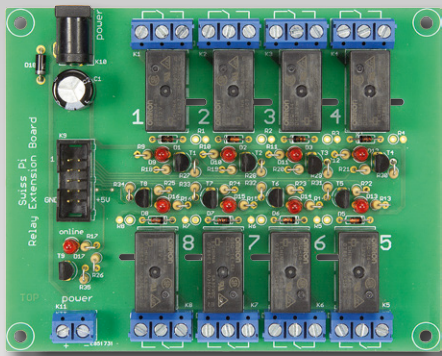
€25/£20/\$30 to
€50/£40/\$60 per board

For each hardware expansion board we have designed a PCB, and the Eagle and Gerber files are available for downloading [2]. That way you can have the boards made by a PCB manufacturer of your choice. We have avoided the use of SMD components as much as possible, to make it easy to build the boards. If you want to go a step further and design something yourself, you can draw on these circuits for suitable inspiration. Naturally, these boards are not limited to use with the Swiss Pi; they can be used in a wide variety of environments.

There is nothing to stop you from using them with your own microcontroller circuits or with an Arduino board.

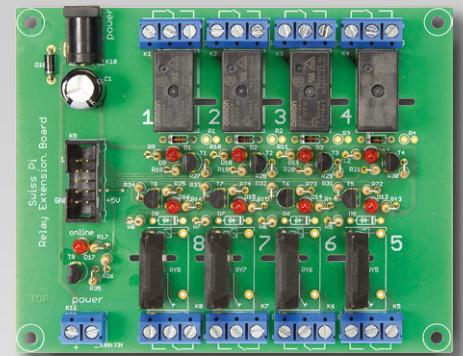
Eight-channel relay board

The eight-channel relay board (**Figure 1**) uses relays from the Omron G5Q series. Sharp S202S02 solid-state relays can also be mounted on the board in place of conventional relays. Unfortunately those



only solid-state relay that fits within the footprint of a G5Q relay. The S202S02 only has an LED at the input, so a series resistor is necessary.

Despite their compact dimensions, the G5Q relays fulfill all requirements with regard to isolation distances and clearances for use with AC line voltage. By contrast, the pins of the solid-state relays are closer together, so we provided slots in the PCB to fulfill the clearance distance requirements despite this limitation. Although the G5Q relays can switch a maximum of 10 A, the actual current has to be limited to about 5 A due to the width of the PCB tracks.



relays are no longer being made, but for now they are still readily available on eBay. There is no other direct alternative because the S202S02 is by far the

The relays are driven by standard NPN transistors, and there is a status LED for each relay. There is also a power indicator LED on the relay board, which is lit

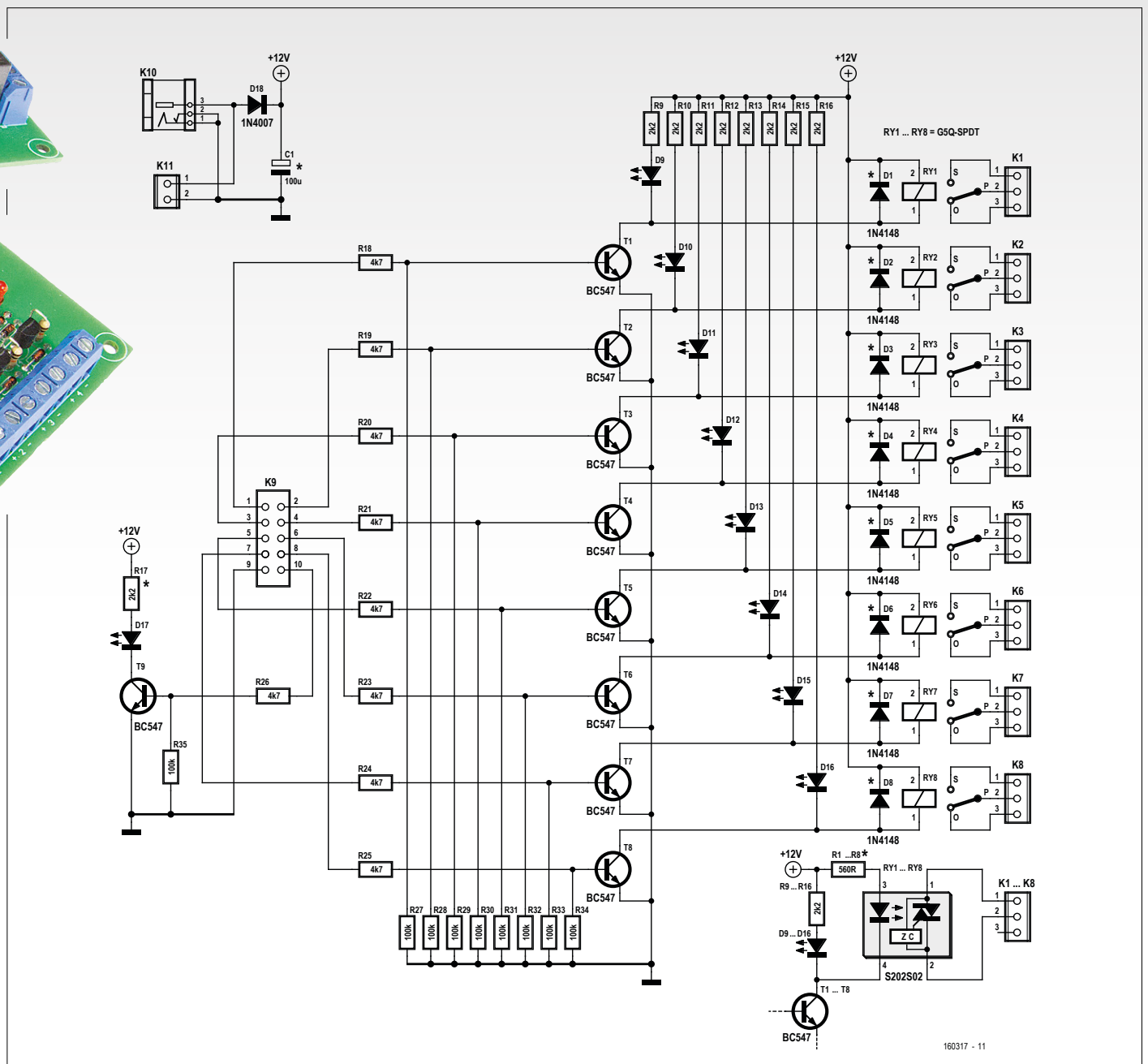


Figure 1. The eight-channel relay board is suitable for both conventional G5Q-SPDT relays and S202S02 solid-state relays.



COMPONENT LIST FOR RELAY BOARD

Resistors

R1–R8 = 560Ω*
 R9–R17 = 2.2kΩ
 R18–R26 = 4.7kΩ
 R27–R35 = 100kΩ

Capacitors

C1 = 100µF/16V

Semiconductors

D1–D8 = 1N4148**
 D9–D17 = LED, 3mm, red
 D18 = 1N4007
 T1–T9 = BC547B

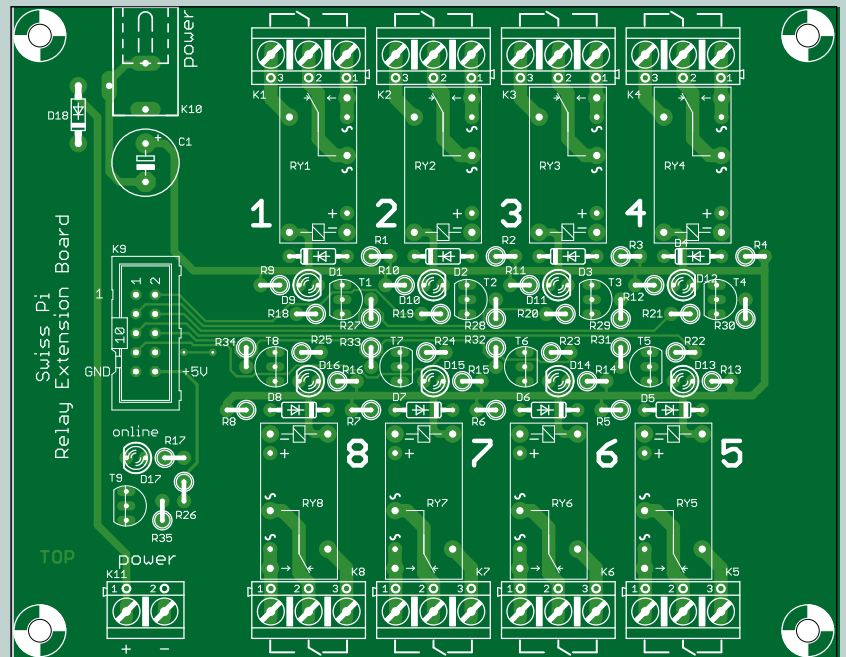
Miscellaneous

RY1–RY8 = G5Q-14-EU 12DC or S202S02***
 K1–K8 = 3-way PCB screw terminal block, 0.2" pitch
 K9 = 10-pin boxheader
 K10 = DC power connector, coaxial
 K11 = 2-way PCB screw terminal block, 0.2" pitch

*Mount when using an S202S02 solid state relay

**Mount when using a G5Q relay

***Mount per channel as desired; for the G5Q relays a different supply voltage can be chosen if necessary. In that case it may be necessary to alter the values of R1–R17 and C1.



when the supply voltage is present on the Swiss Pi board.

If desired, you can also use G5Q relays with a different supply voltage rating,

such as 24 V. However, in that case a number of resistor values will have to be changed.

Eight-channel digital input board

This extension provides eight galvanically isolated inputs with an input volt-

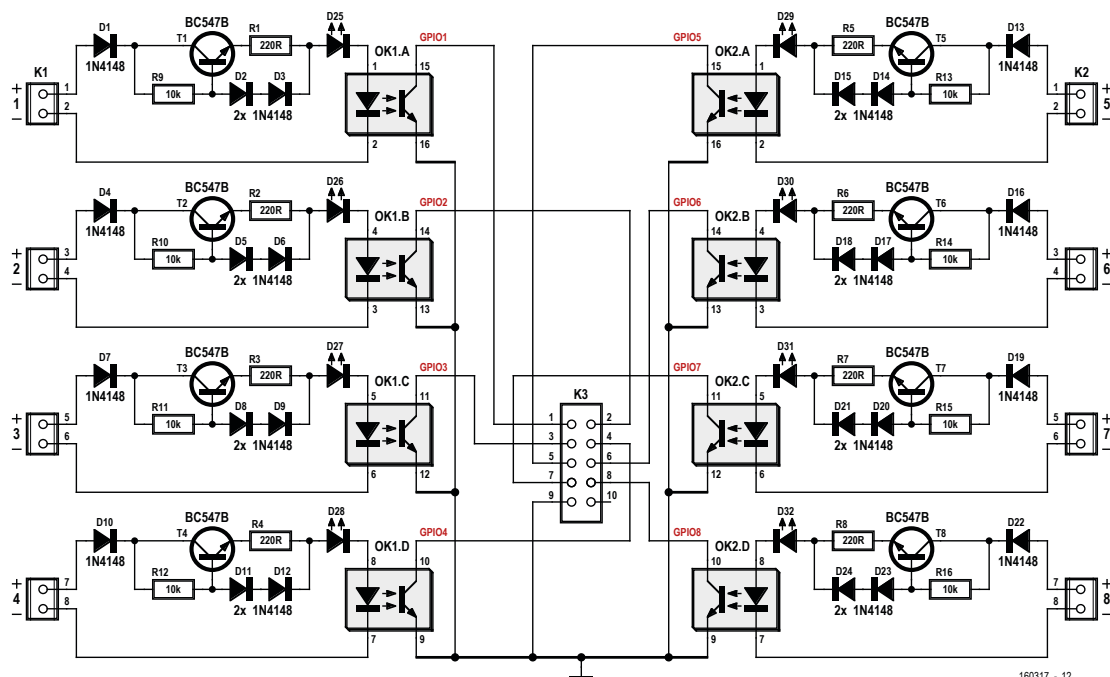


Figure 2. Eight galvanically isolated digital inputs are implemented with optocouplers.

age range of 5 to 45 V (see **Figure 2**). Galvanic isolation is provided by optocouplers. On the input side the current sources built around the BC547 transistors keep the current through the LEDs of the optocouplers and the indicator LEDs more or less constant, regardless of the input voltage. This current is equal to the forward voltage drop of two 1N4148 diodes minus the base-emitter junction voltage of the transistor, divided by 220 (the resistance value of R1–R8). The current through the 1N4148 diodes ranges from a few microamps to several milliamps over the input voltage range of 5 to 45 V. If you look at the characteristic curve of the 1N4148, you can see that this is precisely the region where the forward voltage is most strongly dependent on the forward current. As a result, the current supplied by the current sources is not truly constant over the entire input voltage range. In practice it ranges from 3 to 6 mA, depending on the input voltage.

If the voltage over the current source is reversed, the collector-emitter voltage of the transistor will be negative. However, the transistor will continue to operate as a low-grade NPN transistor with very low gain and a low reverse breakdown voltage. If the magnitude of the negative collector-emitter voltage rises above 6 V, the

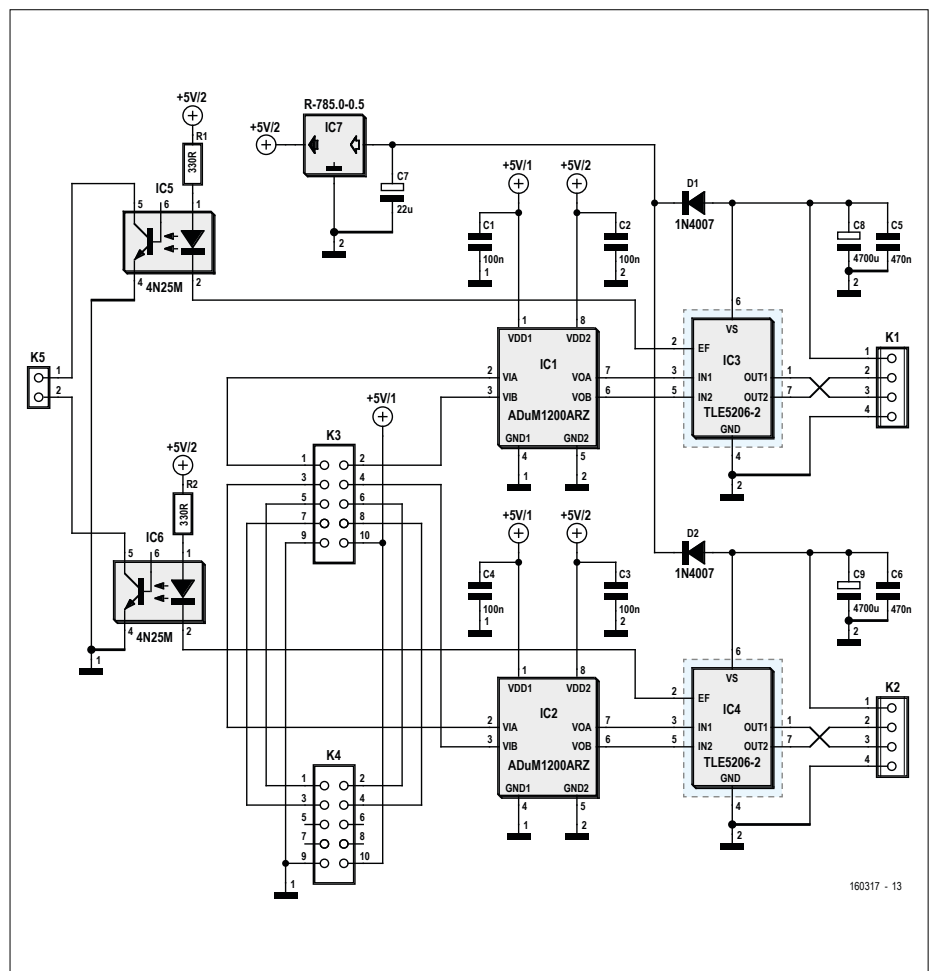


Figure 3. The galvanically isolated motor controller boards can be daisy-chained to drive up to eight DC motors (two per board).



COMPONENT LIST FOR DIGITAL INPUT BOARD

resistors

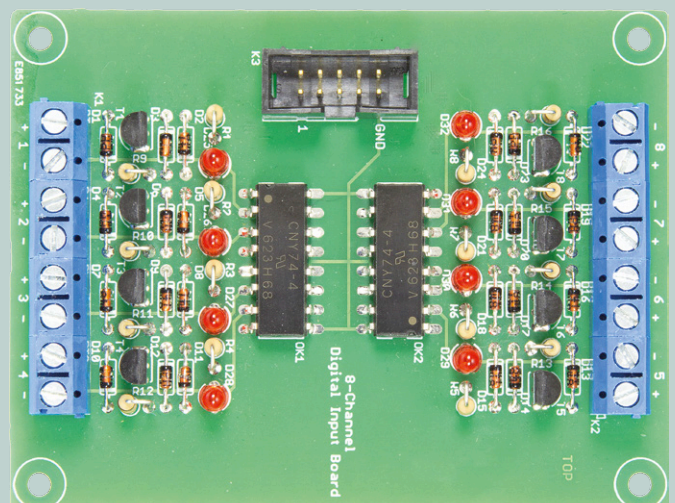
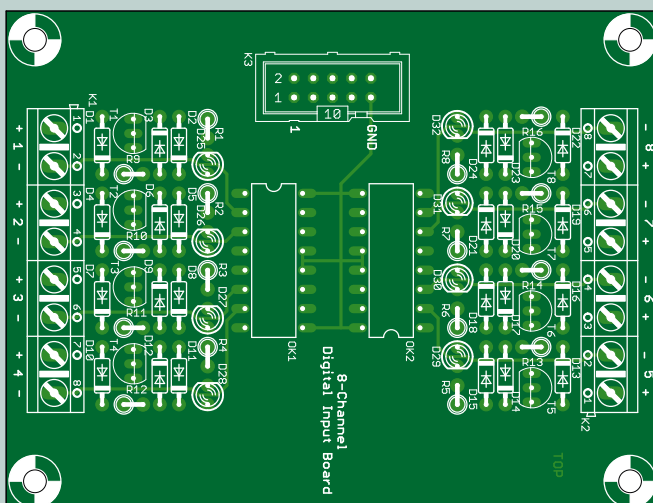
R1–R8 = 220Ω
R9–R16 = 10kΩ

Semiconductors

D1–D24 = 1N4148
D25–D32 = LED, 3mm, red
T1–T8 = BC547B
OK1, OK2 = CNY74-4

Miscellaneous

K1, K2 = 8-way PCB screw terminal block, 0.2" pitch
K3 = 10-pin boxheader



transistor will start conducting and the current will only be limited by the 220 Ω resistors and the leakage currents of the indicator and optocoupler LEDs. To avoid potential damage to the LEDs, particularly with relatively high input voltages, an additional 1N4148 diode is included to provide reverse-polarity protection. In the prototype we used red LEDs as indicators, but you can also use other colors. Bear in mind that this will have some effect on the input voltage range, especially the minimum input voltage. On the Swiss Pi side the optocouplers have open-collector outputs. This means that the GPIO pins must be configured as inputs with pull-up resistors. When a voltage is applied to an input on the extension board, the corresponding GPIO input is pulled low (inverse logic).

DC motor controller

The DC motor controller board (**Figure 3**) can drive two DC motors independently, with a maximum operating voltage of 40 V and a maximum current of 5 A. The output stage of this board consists of two TLE5206-2 MOSFET H-bridge ICs. These ICs are especially easy to control and are protected against overcurrent and shorted outputs. The drain-source resistance $R_{DS(on)}$ is 0.2 Ω , so the power dissipation remains within reasonable limits. A drawback of this IC is that the switching time ("source on" time) can be as much as 15 μs , which is fairly slow. However, that is not a significant problem because the PWM controller of the Swiss Pi has a maximum upper frequency of about 1.5 kHz. This also means that the drive frequency is

audible on the DC motor, especially at low PWM values.

To prevent noise and harmonics on the motor supply voltage rails from reaching the Swiss Pi and the Raspberry Pi, galvanic isolation is provided between the H-bridges and the Swiss Pi. For the PWM signals this is done using type ADUM1200ARZ digital isolators. These devices have two channels and much wider bandwidth than optocouplers. Unfortunately, they are not available in a through-hole version, but the 8-pin SOIC package is relatively easy to solder by hand. Conventional optocouplers are used for the optional feedback of error flag signals from the H-bridges.

The dual motor controller boards can



COMPONENT LIST FOR DC MOTOR CONTROLLER

Resistors

R1,R2 = 330 Ω

Capacitors

C1-C4 = 100nF

C5,C6 = 470nF/63V*

C7 = 22 μ F/50V*

C8,C9 = 4700 μ F/50V*

Semiconductors

D1,D2 = 1N4007

IC1,IC2 = ADUM1200ARZ

IC3,IC4 = TLE5206-2

IC5,IC6 = 4N25

IC7 = DC/DC converter SIP3, 5V/1A, 40Vin
(e.g. Würth 173010542)*

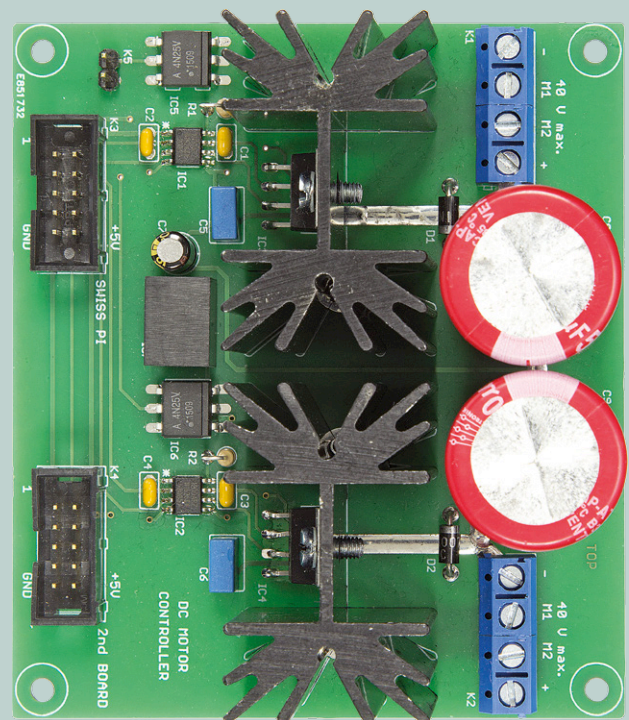
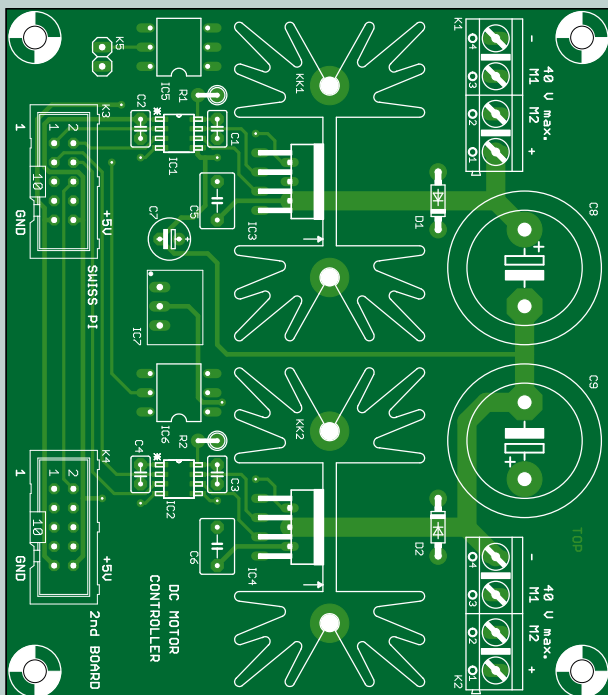
Miscellaneous

K1,K2 = 4-way PCB screw terminal block, 0.2" pitch

K3,K4 = 10-pin boxheader

KK1,KK2 = heatsink, e.g. SK 129 38,1 STS from Fischer Elektronik (38.1 x 42 x 25 mm)

*These component values can be adjusted if the motor controller is used with voltages lower than 40 V. IC7 does not need to supply 1 A, but all available DC/DC converters with a relatively high input voltage rating (over 28 V) are 1 A types.



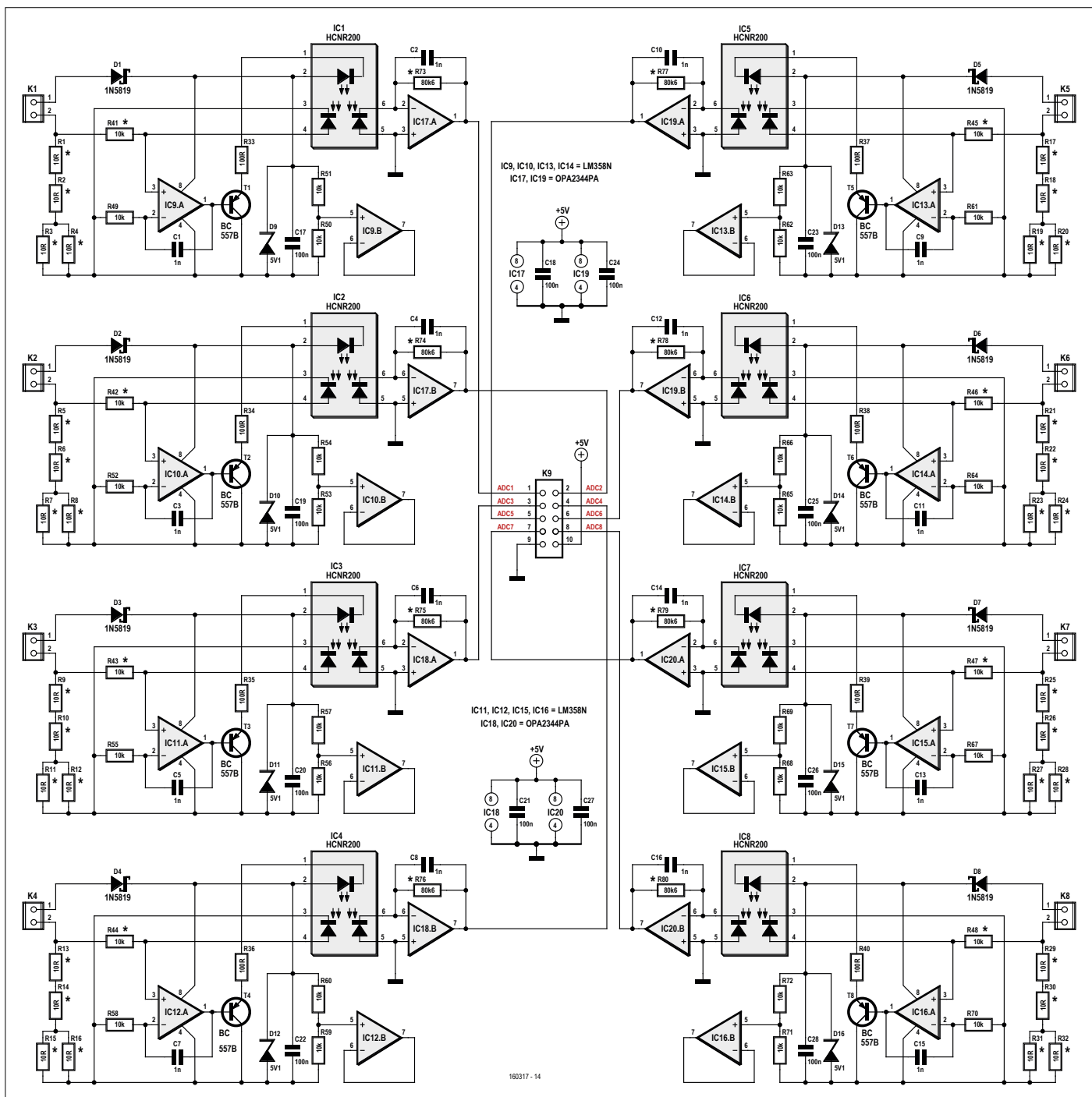


Figure 4. The schematic of the current loop interface board looks fairly large, but it actually consists of eight identical circuits.

be daisy-chained to enable a total of eight DC motors to be controlled with the Swiss Pi.

Finally, a DC/DC converter provides the 5 V supply voltage for the logic circuitry on the H-bridge side. If you want to use relatively high supply voltages, it is essential to choose a DC/DC converter that can withstand a 40 V input voltage.

Current loop interface

Current loop interfaces are still widely used in the industrial sector for read-

ing sensors, despite the advent of digital alternatives. Current loops are simple and robust, and they have a long working range. Consequently, there are a lot of sensors available which use this technology.

The interface described here (see **Figure 4**) is built around HCNR200 linear optocouplers from Avago. The internal structure of these devices consists of two matched photodiodes and an LED. At the transmit end, one photodiode is used to

provide feedback from the power emitted by the LED. The overall transmission characteristic is virtually linear because the current through the photodiode at the receive end is nearly the same as the current through the photodiode at the transmit end.

For this circuit we used a fairly direct copy of the typical application circuit on the Avago data sheet, with only a few minor changes. For instance, we replaced the LM158 opamps at the transmit end



COMPONENT LIST FOR CURRENT LOOP INTERFACE

Resistors

R1–R32 = 10Ω*
 R33–R40 = 100Ω
 R41–R48 = 10kΩ*
 R49–R72 = 10kΩ
 R73–R80 = 80.6kΩ*

Capacitors

C1–C16 = 1nF
 C17–C28 = 100nF

Semiconductors

D1–D8 = 1N5819
 D9–D16 = Zener diode 5.1V, 400mW
 IC1–IC8 = HCNR200
 IC9–IC16 = LM358N
 IC17–IC20 = OPA2344PA (LM6142BIN/
 NOPB)

Miscellaneous

K1–K8 = 2-way PCB screw terminal block, 0.2"
 pitch
 K9 = 10-pin boxheader

*1% metal film or better

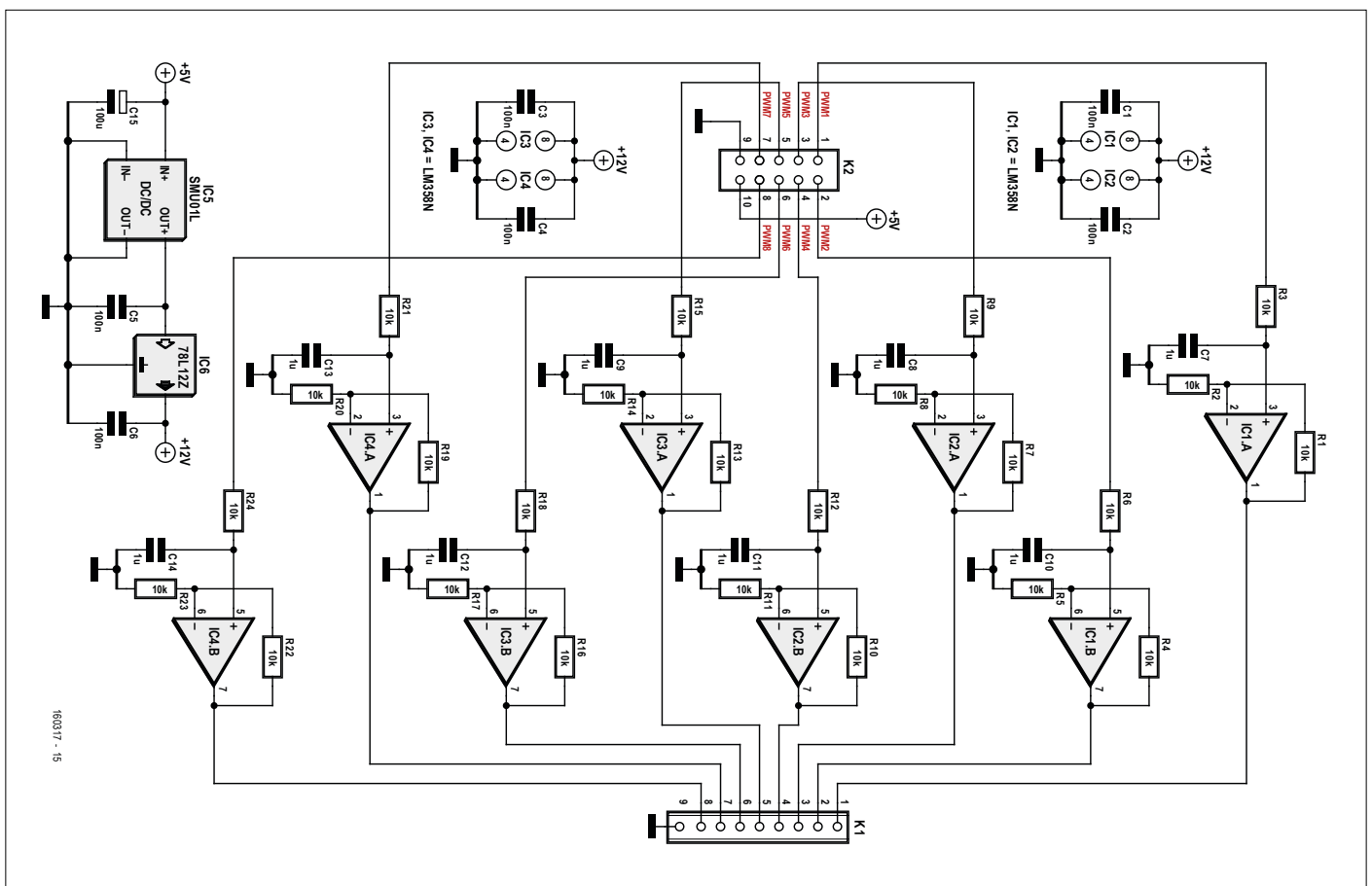
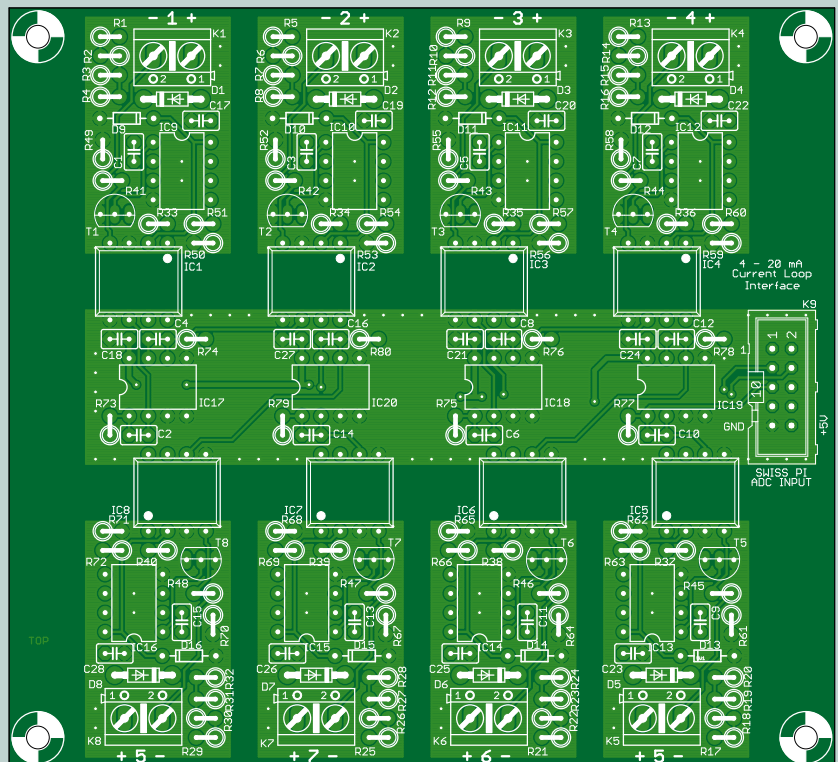
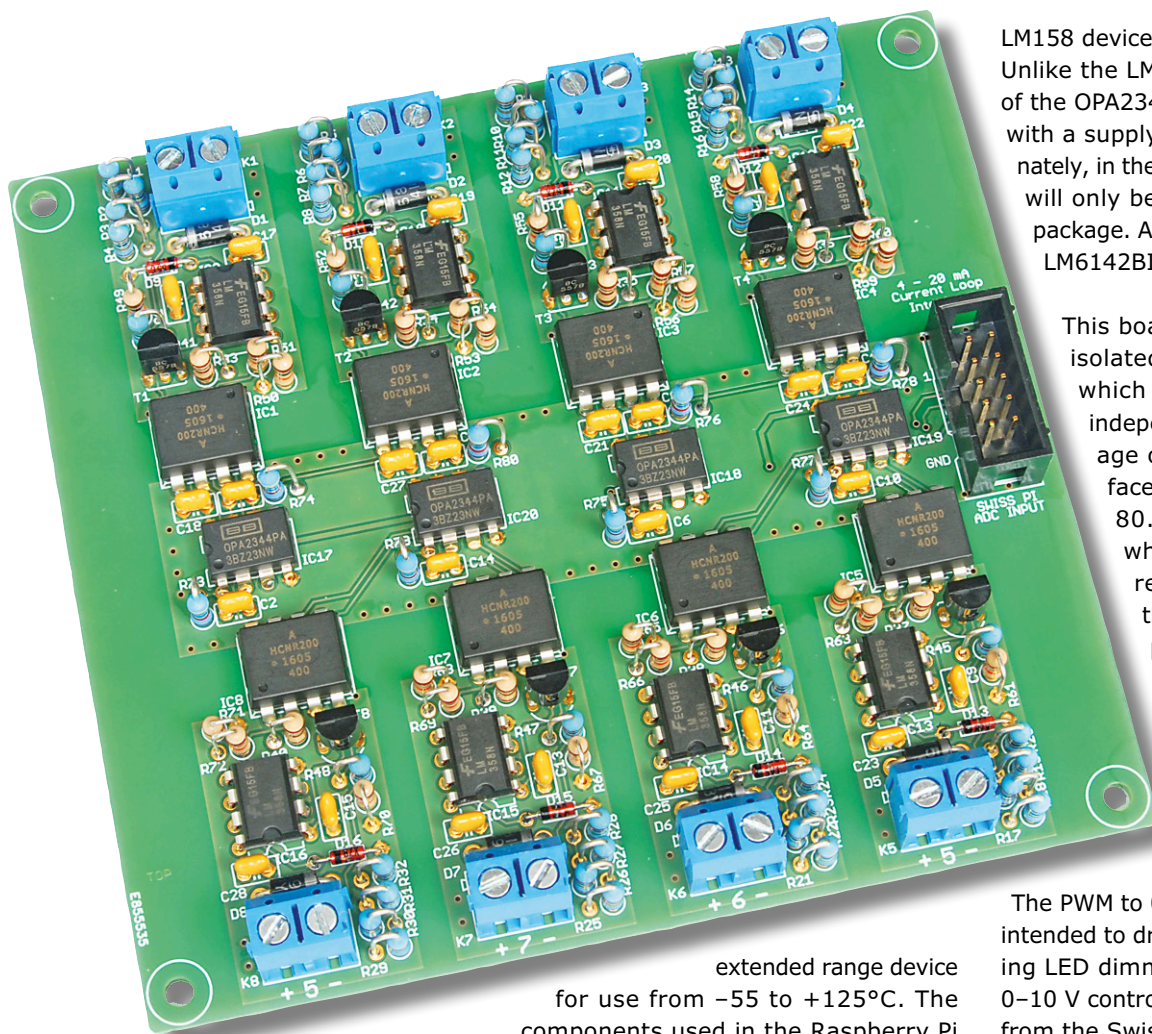


Figure 5. The PWM to voltage converter board generates 0 to 10 V voltage outputs from PWM signals.



LM158 devices by OPA2344PA opamps. Unlike the LMx58 opamps, the output of the OPA2344PA can easily reach 4 V with a supply voltage of 5 V. Unfortunately, in the near future these opamps will only be available in the SOIC-8 package. A possible alternative is the LM6142BIN/NOPB.

This board provides galvanically isolated current loop interfaces which can be used completely independently. The output voltage of the current loop interface is equal to $I_{in} \times (25 \Omega \times 80.6 \text{ k}\Omega) / (10 \text{ k}\Omega + 25 \Omega)$, which is $I_{in} \times 201$. The current range of 4 to 20 mA thus corresponds to an output voltage range of 0.804 to 4.02 V, which is within the A/D conversion range of the Swiss Pi.

PWM to voltage converter

The PWM to 0–10 V converter board is intended to drive light dimmers (including LED dimmers), most of which use 0–10 V control signals. The PWM signal from the Swiss Pi is converted to a DC voltage by a RC filter, and this voltage is then buffered and doubled by the opamps (see **Figure 5**).

The current consumption of the opamps

by LM358 devices. They cost a lot less and are more readily available than the LM158. The main difference is in the rated temperature range of the opamps: the LM158 is an

extended range device for use from –55 to +125°C. The components used in the Raspberry Pi and the Swiss Pi are intended for commercial use with a temperature range of 0 to 70°C. In that situation there is not much reason to use the LM158. At the receive end we replaced the



COMPONENT LIST FOR PWM TO VOLTAGE CONVERTER

Resistors

R1–R24 = 10kΩ

Capacitors

C1–C6 = 100nF

C7–C14 = 1μF

C15 = 100μF

Semiconductors

IC1–IC4 = LM358N

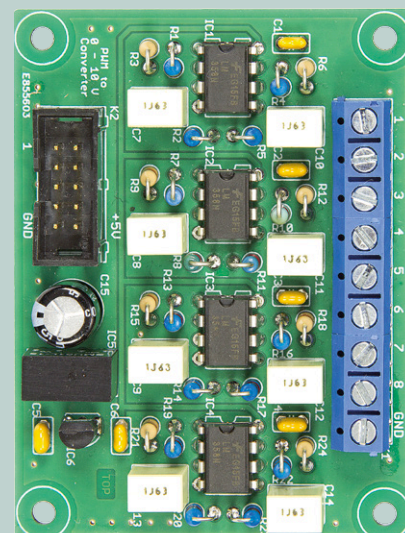
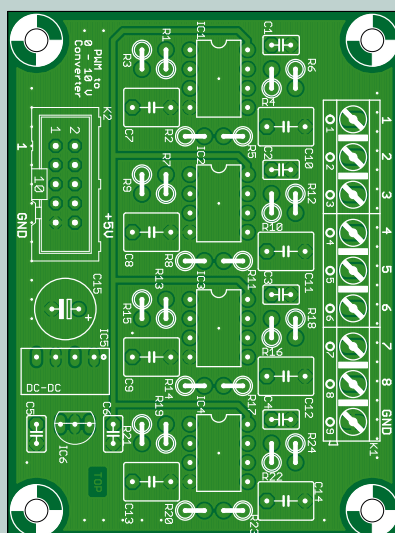
IC5 = SMU01L-15

IC6 = 78L12Z

Miscellaneous

K1 = 9-way PCB screw terminal block, 0.2" pitch

K2 = 10-pin boxheader



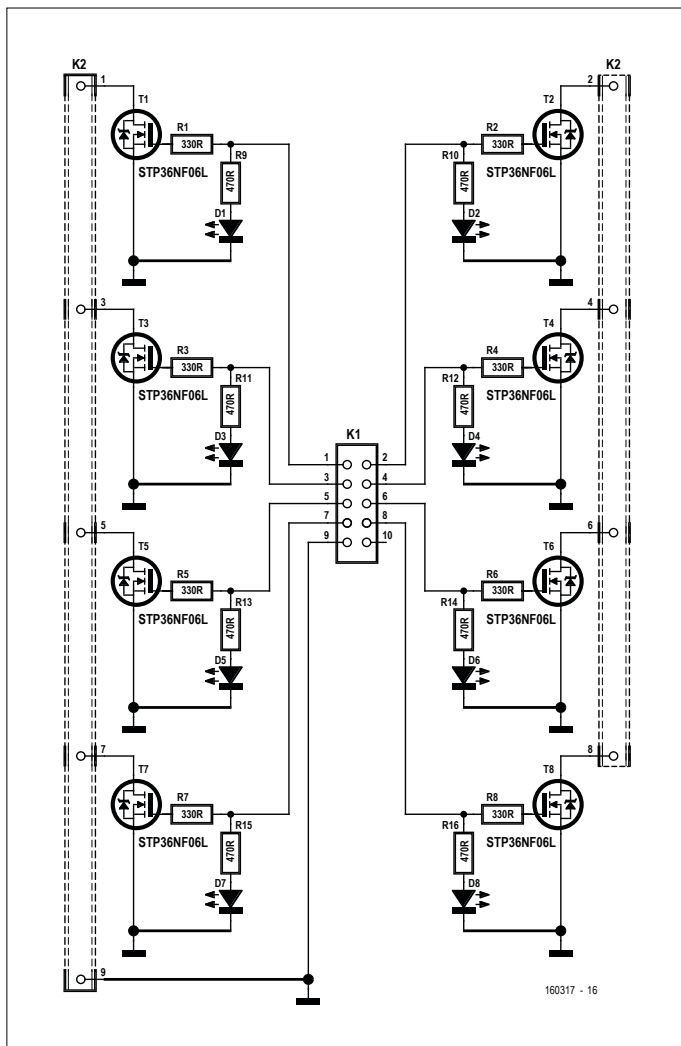


Figure 6. The MOSFET board is populated with STP36NF06L devices, which can handle 30 A and a maximum voltage of 60 V.

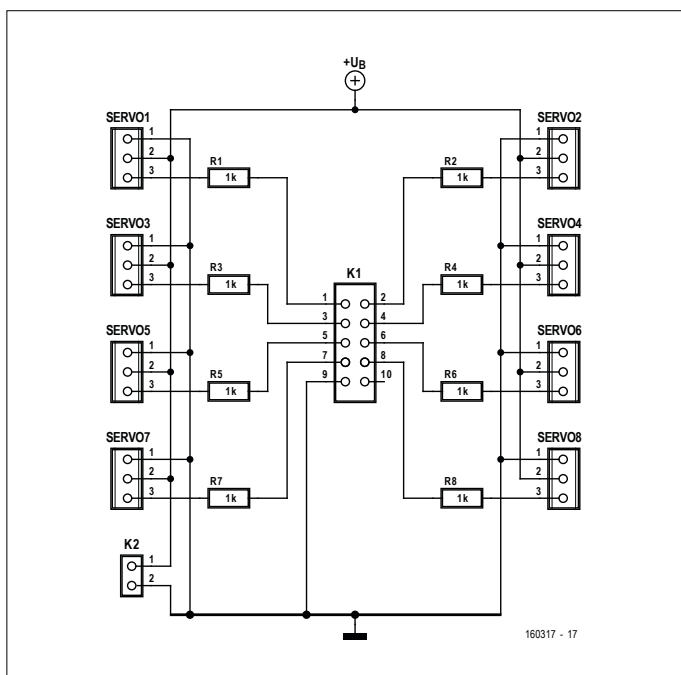


Figure 7. With this simple servo motor board you can control up to eight servo motors from the Swiss Pi.



COMPONENT LIST FOR MOSFET BOARD

Resistors

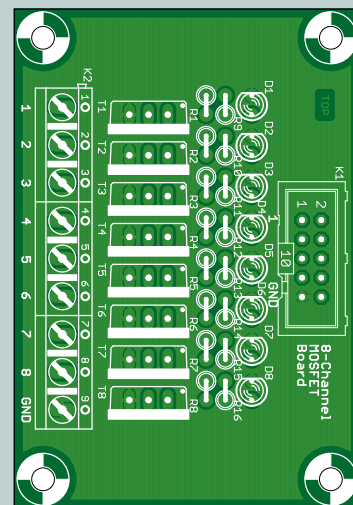
R1–R8 = 330Ω
R9–R16 = 470Ω

Semiconductors

D1–D8 = LED, 3mm, red
T1–T8 = STP36NF06L

Miscellaneous

K1 = 10-pin boxheader
K2 = 9-way PCB screw
terminal block, 0.2" pitch



is low and the load on the 0–10 V signals is usually low, so a simple DC/DC converter is used to provide the higher supply voltage for the opamps. It converts the 5 V supply voltage from the Swiss Pi to 15 V, which is then reduced to a regulated 12 V by a 78L12. That eliminates the need for an external power source.

You should bear in mind that the output voltage range may not be exactly 0 to 10 V — in practice it extends from 0 V to twice the supply voltage of the PWM IC on the Swiss Pi board.

MOSFET board

The MOSFET board (**Figure 6**) can be used to switch DC loads. The normal GPIO lines of the Swiss Pi or the PWM channels can be used for this purpose. Driving RGB LED strips is a typical application.



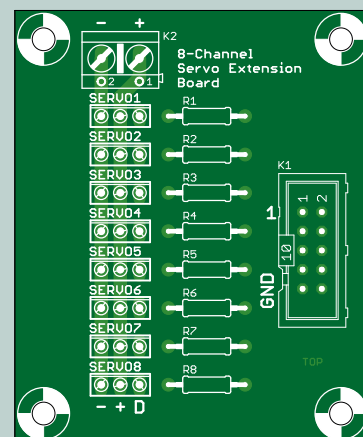
COMPONENT LIST FOR SERVO MOTOR BOARD

Resistors

R1–R8 = 1kΩ

Miscellaneous

K1 = 10-pin boxheader
K2 = 2-way PCB screw
terminal block, 0.2" pitch
K3–K10 = 3-pin header,
0.1" pitch



For the MOSFETs we chose type STP36NF06L devices. That is an N-channel MOSFET which is suitable for voltages up to 60 V with a rated current of 30 A ($R_{DS(on)} = 0.032 \Omega$ at $V_{GS} 10 \text{ V} / 15 \text{ A}$). The STP36NF06L is a logic-level MOSFET, which means it can be driven fully on even with a gate voltage less than 5 V. With a V_{GS} of 5 V the on resistance $R_{DS(on)}$ is 0.045Ω , which in practice means that you can switch about 5 A per channel without needing a heat sink for the MOSFET. Each channel also has an indicator LED.

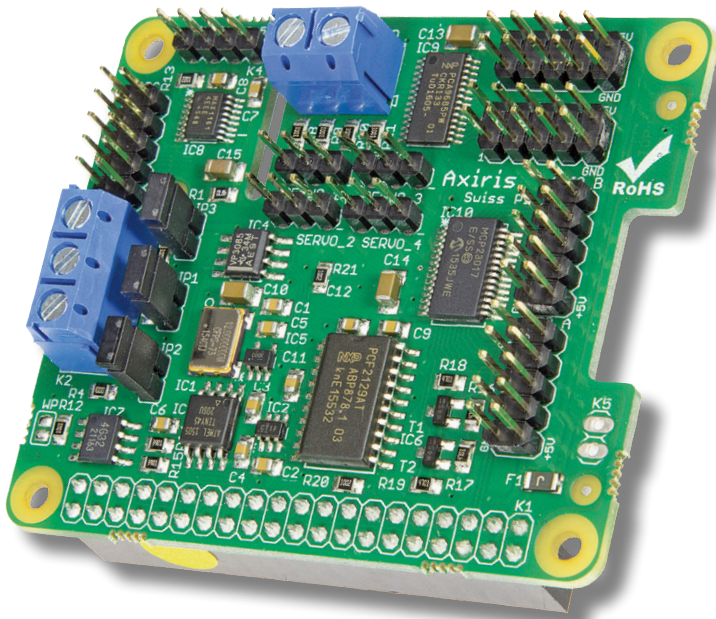
Servo motor extension

With the servo motor extension board (**Figure 7**) you can connect more servo motors to the Swiss Pi than the four supported directly. With two of these boards you can control a maximum of sixteen servo motors.

The resistors protect the Swiss Pi in the event that the three-pin plug of a servo motor is accidentally connected the wrong way.

In combination with the previous article and the many software examples described there [3], you now have an arsenal of tools to tackle a wide range of applications with the Swiss Pi / Raspberry Pi duo. ◀

(160317)



Web Links

- [1] www.elektormagazine.com/150584
- [2] www.elektormagazine.com/160317
- [3] www.elektormagazine.com/160237



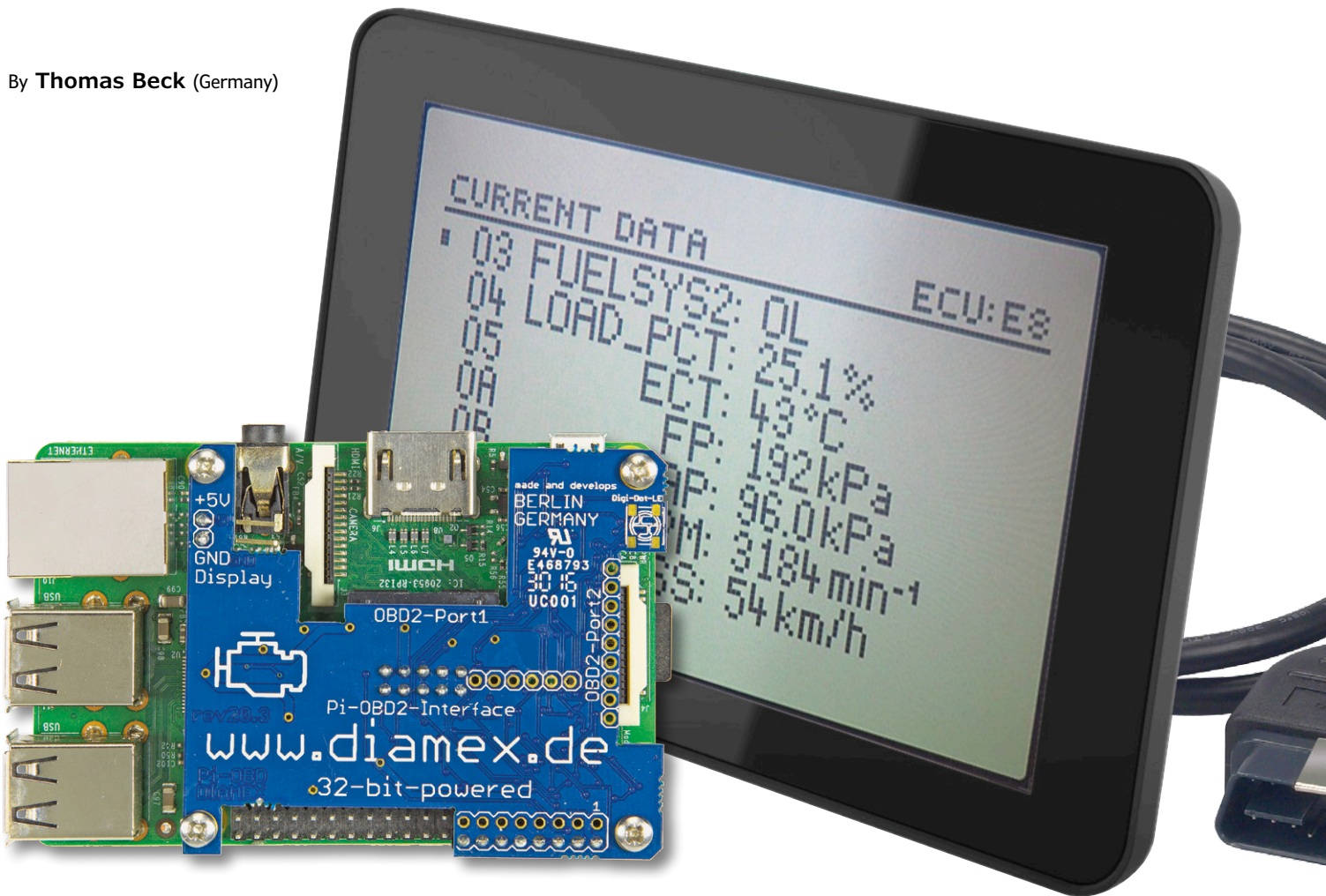
IN THE STORE

- SKU 17713 Swiss Pi (150584-91))
- SKU 17631 Raspberry Pi 3

OBD2 Handheld using a Raspberry Pi

With new diagnostic software

By **Thomas Beck** (Germany)



The Pi-OBD-HAT is an off the shelf add-on board for the Raspberry Pi which converts the tiny computer into a dedicated OBD2 diagnostic tool. One disadvantage was the need of a terminal program running on a PC for control. A much more convenient and neater solution is provided by the HHGui OBD2 diagnostic software.

The GUI software HHGui is a variant of the HHemu program I wrote for the OBD2 handheld emulator [1] which is a sort of Swiss army knife tool for developers working with the OBD2 analyzer 'NG/DIAMEX Handheld Open' (HHOpen). This unit is a handheld OBD2 diagnostic tool

published in September 2009 in *Elektor Magazine* [2]. The OBD2 analyzer NG employs the DIAMEX DXM OBD2 module [3].

The development process

The software development process is a

little unconventional and shows how firmware originally written for a microcontroller can be ported to a Raspberry Pi or PC, ideally without making any changes to the firmware source code.

HHemu was originally conceived as a pure development tool to test new ver-

sions of the AVR microcontroller firmware for the OBD2 analyzer on a PC without the need to plug it into an OBD-equipped vehicle. In addition to the firmware HHEmu contains an OBD2 emulator which supplies simulated OBD2 control unit data to the firmware via a simulated DXM OBD2 module.

The next step in the development process HHEmu was provided with a serial interface to a PC. Together with a Bluetooth expansion board published in April 2010 [4] this gave the OBD2 analyzer NG a great deal of added flexibility. The firmware running on a PC within HHEmus can control a real NG analyzer and its DXM module using a serial interface to the Bluetooth adapter with the Bluetooth extension. This makes it possible to test new firmware versions on a real vehicle without first flashing the firmware into the OBD2 analyzer. This was a real milestone in the development process! In the meantime HHEmu has become the standard diagnostic software for DXM-based diagnostic interfaces. In addition the OBD2 emulator in HHEmu

can now act as a data generator to test OBD2 diagnostics tools produced by other manufacturers or for the development of new OBD2 software. As well as the improved test possibilities for firmware development there are also other benefits of porting to a PC. With each future firmware update the diagnostic software automatically receives the functionally identical update.

Now HHEmu has been ported to the Raspberry Pi. After all the changes necessary for operation with the Pi-OBd hat and the removal of any OBD2 analyzer-NG specific menus the first version of HHGui is ready for publication.

Operational modes

We can say, because of its development path that HHGui is essentially OBD2 diagnostic software with a user interface that mimics the look of the OBD2 analyzer NG as well as internally running its firmware adapted to the PI-OBd module. The current range of the firmware functions available are given in the **Table**.

HHGui presents the OBD2 user menus and data on a simulated LCD display scaled to the larger graphic display of the RPi. This is a compromise so that in

The OBD2 functionality of HHGui

HHGui supports the OBD2 services contained in ISO15031-5. A more detailed description can be found on the project page [5], for more on the sub-functions refer to [6], [7] or [8]. The range of supported services will depend on the vehicle type and its OBD2 control unit. HHGui will interrogate the respective control unit and display the supported sub-functions.

- Support for up to eight OBD2 ECUs (Electronic Control Units) according to ISO 15765-4 [9]
- OBD2-Service 0x01: Read Live or Current Data; support also for Parameter Identifier (PIDs): 0x00...0x60, 0x70, 0x80, 0x8D sub-functions
- OBD2-Service 0x02: Read Freeze Frame Data, supports PIDs as for Service 0x01
- OBD2-Service 0x03: Read confirmed Diagnostic Trouble Codes (DTC)
- OBD2-Service 0x04: Clear DTCs, stored values, MIL status, OBD2 monitor and other data
- OBD2-Service 0x07: Read Pending DTCs
- OBD2-Service 0x09: Read Vehicle Information; support also for (InfoTypes): 0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0B
- OBD2-Service 0x0A: Read permanent DTCs

addition the official 7" touchscreen recommended for the RPi it also supports displays as small as 320x240 pixels. A modification of the Pi-OBd module is necessary as described in the first project update of the Labs-project [5].

The second level of functionality implemented in HHGui is the OBD2 simulator which can be used to represent the Pi-OBd module and up to eight configurable OBD2 controllers. This feature allows you to test the full range of software functions without the need for a Pi-OBd module or vehicle. Plugging into a real vehicle will also not give you access to all the possible OBD2 functions and subfunctions because many of these are exclusively related to the type of engine fitted (i.e. petrol (gas) or diesel powered vehicles). Another bonus is that this simulator is completely harmless, there's no chance of accidentally deleting any of the vehicle's OBD2 data. The range of OBD functions provided by the simulator is identical to the range of functions in the firmware. Commands to control the OBD2 simulator entered via the keyboard are described in the Labs HHEmu project [1].

What's possible using OBD2 diagnostics?

So returning to the main functions of the OBD2 diagnostic software, what can we expect to achieve with the services available via the OBD2 interface?

When you take a close look at the ISO 15031-5 [6] specification (or the identical SAE J1979 [7]) it becomes clear that via

the OBD2 port we can read emission-relevant information and trouble codes and partially delete them. This information is mainly generated by the vehicle's ECU. Vehicles with automatic transmission can also produce OBD2 trouble codes from the Power-train Control Module. Less often the vehicle will be fitted with additional control systems or even multiple ECUs in the case of hybrid vehicles. General settings such as automatic door locking or inhibiting the warning gong sound for seat belt reminder cannot be made via the OBD2 connector. With the exception of the Malfunction Indicator Lamp (MIL) none of the other fault indicator lamps on the instrument cluster or the service warning lamp or service interval can be reset. These actions can only be performed using the manufacturer's diagnostic equipment which is usually specific to the vehicle model.

That may seem a little disappointing at first. The advantage of this manufacturer standardized OBD2 diagnostics is that now after 20 years since its introduction in the US (and since the year 2000 for petrol-powered vehicles in Europe), it works with almost every vehicle. Since 2008 the proliferation of different protocols for transferring OBD2 data has been drastically reduced so that now only the CAN protocol can be used.

Take a closer look at all the available OBD2 sub-functions you will find an incredible amount of sensor information, counter values and other measurements (altogether more than 100) that's sure



to make you want to explore further. Apart from the sensor information you would expect to have access to from the Engine Control Unit you can also read data originating from other equipment on the vehicle such as the brake control module.

In addition to the raw OBD2 data some values undergo processing before they are displayed on the instrument panel. Some examples are:

- value smoothing (outside air temperature, fuel gauge and generally any instrument with a pointer driven by a stepper motor);
- values with a plateau function with a limited range of interest (coolant temperature);
- values with added offsets, to provide a margin for error and ensure that displayed values will not allow the driver to unknowingly exceed any legal restriction or simply to comply with the manufacturers recommended limits (speedometer, engine RPM).

Data supplied from the OBD2 interface can also be raw sensor values without any processing so we can read the actual speed rather than the value displayed on the speedometer.

There is also data that you might be interested in reading which the manufacturer does not display such as oil temperature or turbocharger boost pressure. Engine load values may also be of use. The average user might also be interested to know the state of readiness of the vehicle for its next inspection and test (the annual MOT in the UK, or vehicle safety inspection in the US). To get this information it's necessary to run all the OBD2 test programs (OBD2 monitors) at least once after fault conditions have been erased. This condition can be checked in HHGui when in Inspection/Maintenance readiness menu the status of all the OBD2 monitors is displayed as '... monitoring ready: YES'.

In addition to this easily understandable OBD data we also have more esoteric information such as lambda probe read-

ings more of interest to the specialist. A list of all the available Parameter IDs (PIDs) is too long to include here. The latest full-version of the official specification is available at considerable expense, as an alternative it's worthwhile taking a look at the Wikipedia entry [8].

Now we turn to a subject which is of more interest to developers.

HHGui: Structure and Function

Functional Principle

The firmware is executed in a thread — all inputs and outputs of the firmware are processed and responded to in additional threads. All of the microcontroller registers and interrupts need to be simulated.

Actual implementation

HHGui consists of six threads. **Figure 1** is a simplified block diagram showing the interaction between the threads. The source files use different names as a result of the program development history. The Pi-OBd module thread is called

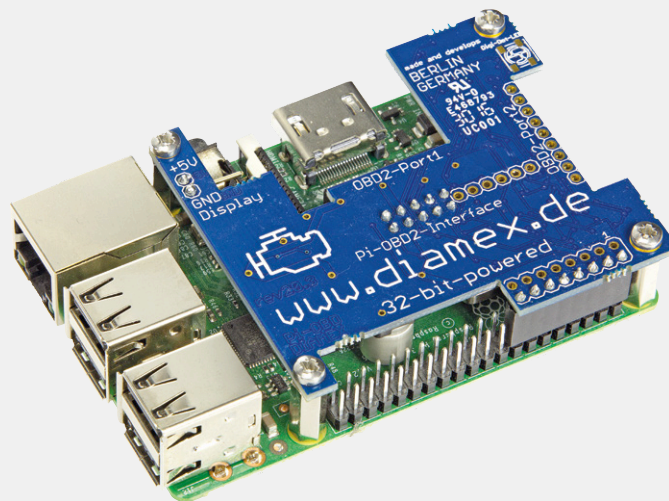
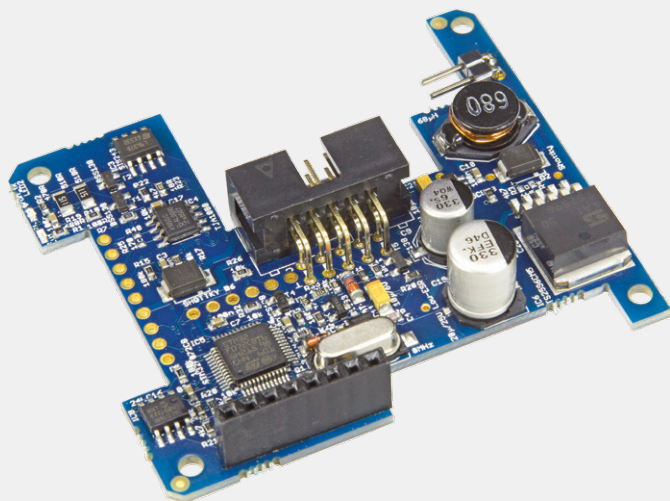
PI-OBd: The Hardware

The PI-OBd-HAT [10] from Diamex turns a Raspberry Pi into an OBD2 diagnostics adapter. Fitted with the 7" touch screen you have a handy and powerful stand-alone diagnostic tool for vehicles with an OBD2 interface. The RPi and its display are powered directly by the vehicle's own 12 V supply. Any of the Raspberry Pi variants can be used, but the RPi 2

communication takes place using the TxD and RxD pins at 115,200 Baud with 8N1.

Plugging it all together

Plug the Pi-OBd hat into the Raspberry Pi headers. The 8-way header socket plugs onto eight of the I/O pins along the edge



Model B or RPi 3 Model B are preferred owing to their higher processor speed. The PI-OBd hat uses just eight pins of the RPi headers to provide power, connections for serial data interface and reset signal to activate the PI-OBd bootloader. Serial

of the RPi board. The hat is held in place using the threaded spacers and screws supplied.

The Raspberry Pi is powered by the vehicle battery via the OBD2 plug. No other external power source is required unless

DxmThread and in the Firmware for den OBD2 analyzer NG HhopenThread. The shorter names are used in the following description.

The NG-Analyzer firmware is executed in the HhopenThread. This thread communicates with other threads in various ways. Display data for the LcdThread is passed via the SPDR register; RGB values for the LED back light are passed to LcdThread via registers OCR1A/B/C. The UDR0 register is used to pass AT commands, OBD requests and responses to the DxmThread. A firmware reboot is made via the WDTCR register or from MainThread of HHGui. Key presses from the MainThread are received in the PINA register. HhopenThread is controlled via the Interrupt service routines for CounterThread and DxmThread. Details are given in the descriptions of the other five threads.

The MainThread takes care of evaluating command line parameters, initializing the serial interface, generation of other threads and at the end of the program

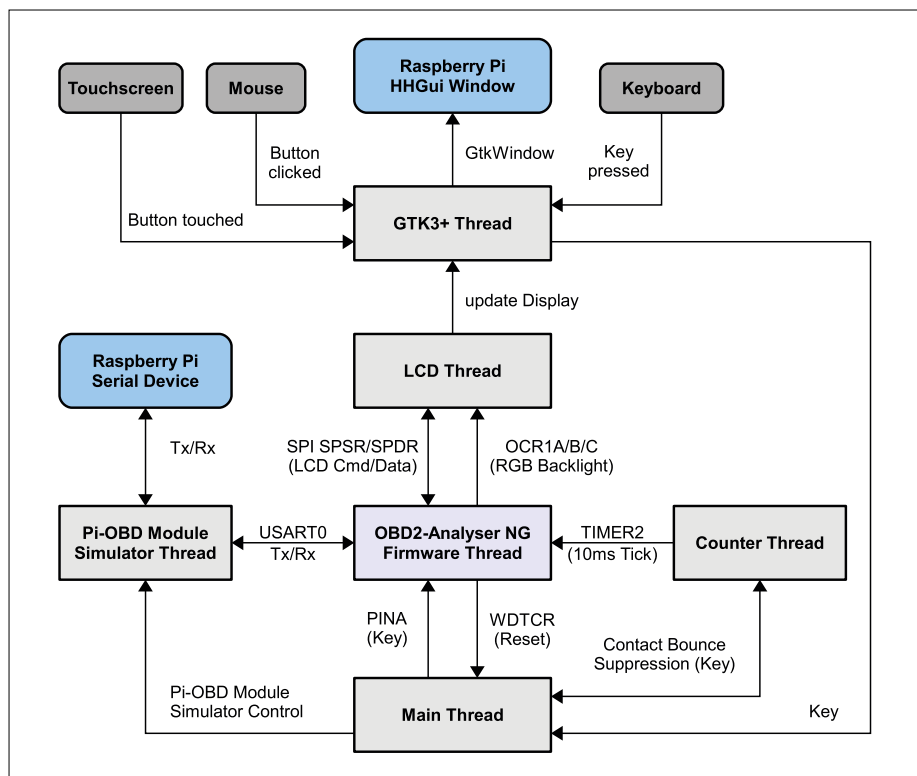


Figure 1. HHGui: Block diagram.

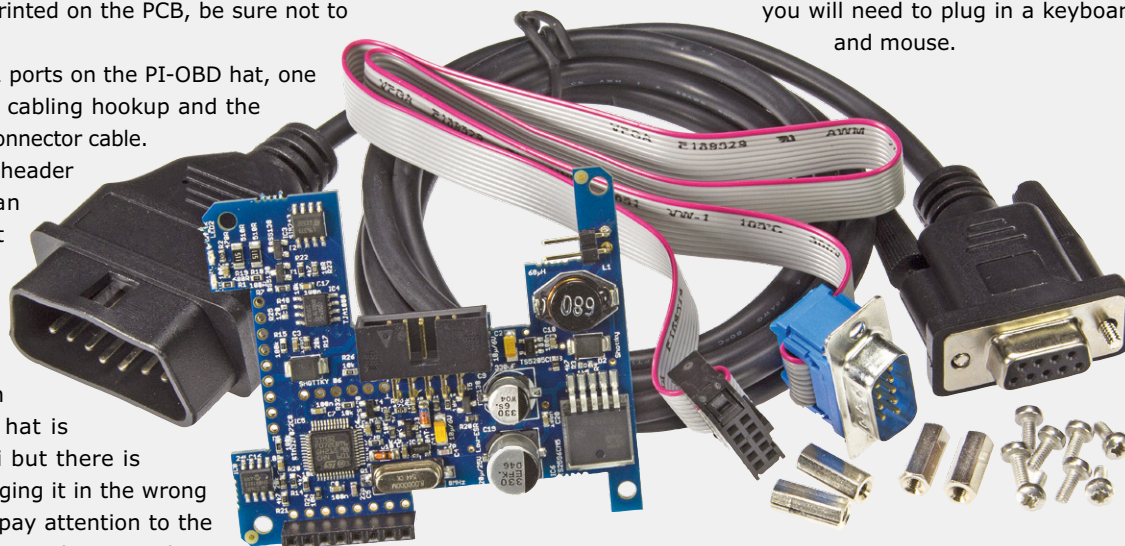
you are testing or configuring the unit away from the vehicle when it will then be powered via its USB connection. Power for the 7" Raspberry Pi display is provided by the two-pin header on the hat. The photo is from under the RPi with the OBD hat fitted and the display PCB above. A description of the header pin connections is printed on the PCB, be sure not to get them mixed up!

There are two OBD2 ports on the PI-OBD hat, one is suitable for 'free' cabling hookup and the other with a sub-D connector cable.

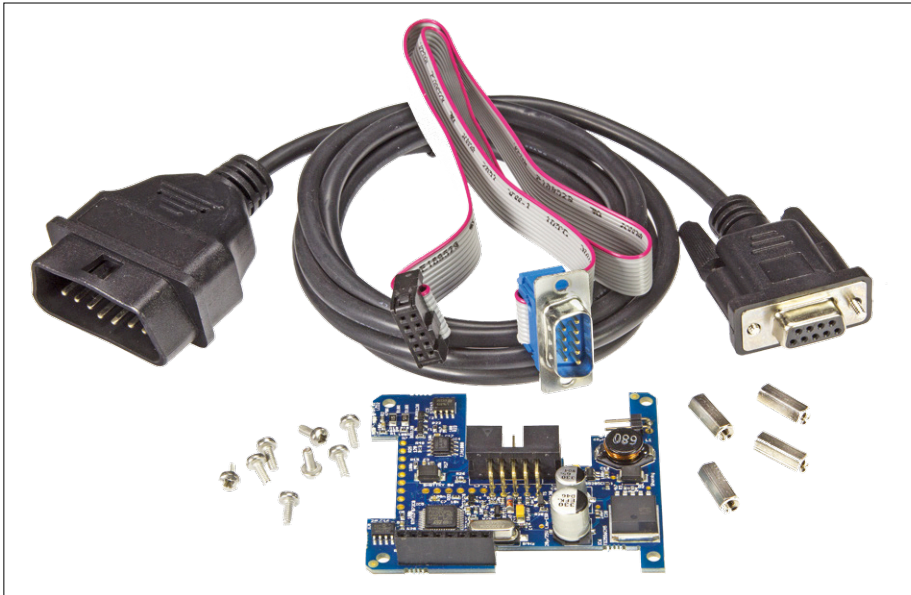
A single-row 9-way header plug (or socket) can be used to connect to Port 2 along the board edge. The port 2 connector has the advantage it can be plugged in and out whilst the hat is attached to the RPi but there is also the risk of plugging it in the wrong way round. Always pay attention to the printed identification on the PCB! There can be no confusion on port 1 because the connector on the board has a detent position and the 9-pin sub-D connector at the other end of the cable can only be connected one way round. The flat band connector on this port must be inserted before the hat is assembled onto the RPi, it can't be plugged in or withdrawn without first releasing the hat.

Fire it up

To begin using the PI-OBD HAT you will need some basic knowledge of the Raspberry Pi environment, the use of Linux and the software installation. For the purposes of configuration and operation, unless you are using a touch screen with the RPi, you will need to plug in a keyboard and mouse.



The first exercise now is to enable the RPi's serial interface but a slightly different procedure is necessary for each different RPi model and space limitations here makes it necessary to provide this information via this link [10]. The unit will be ready for use after a functional test using a terminal program.



returns these resources. The thread also evaluates key press information sent by GtkThread in an endless loop. It is necessary to differentiate between the keys used to control the OBD2 simulator and the Up/Down/ESC/OK keys used for the HhopenThread. For these keys used by the HhopenThreads it is necessary for the firmware to ignore multiple contacts produced by contact bounce. The key press must be present for 40 ms in the PINA register in HhopenThread to be recognized. This 40-ms window is measured using the CounterThread. Keys used only for the OBD2 emulator produce changes in settings in the DxmThread.

The loop that evaluates pressed keys terminates when a watchdog reset is triggered from HhopenThread (by holding down the ESC key) or by terminating HHGui using the Q key.

LcdThread evaluates the commands and display data received via the simulated SPI interface from the HhopenThread and translates them into commands for the

GTK3+ graphics library. The LED backlight RGB values in the OCR1A/B/C registers define the displayed colors together with the contrast setting (via the SPI interface). The LCD in the OBD2 analyzer NG uses an ST7565R display controller so LcdThread contains an ST7565R simulator for the ST7565R commands used by the Firmware.

The GtkThread is necessary because the GTK3+ main loop (function *gtk_main()* in the GTK3+ library) blocks the rest of the gtk code. GTK3+ drawing functions should only be called from inside GtkThreads or in context of the GTK3+ main loop. When LcdThread requests a display update, it installs a callback-function, which will be called later from the GTK3+ main loop. The GtkThread evaluates all kinds of input events and generates the Up/Down/ESC/OK button presses for the HhopenThread or key input events to control the OBD2 emulator.

The CounterThread is primarily used to simulate the interrupt generated by

Timer2 which occurs every 10 ms. In response the `TIMER2_COMP_vect()` Interrupt Service Routine (ISR) is called every 10 ms. Apart from that it is used in the MainThread in contact debouncing as described earlier.

In the OBD2 software mode the DxmThread acts as an interface adapter and just transfers the data unchanged. The received data in the UDR0 register (USART0-I/O data register) which may be AT commands for the Pi-OBd module or OBD2 requests, are sent to the real PI-OBd module using the configured serial interface. After each byte is read the firmware is notified that the byte has been sent via a call of the `USART_UDRE_vect()` ISR indicating that next byte can be written to the register. The reply from the Pi-OBd module is received through the serial interface and passed to the firmware via register UDR0. The reply (OK, Pi-OBd module prompt character >, error or OBD2 data) is handled bitwise in the same way by writing to the UDR0 register and a call to the `USART0_RX_vect()` ISR tells the firmware that the register has been read.

In OBD2 emulator mode the DxmThread emulates the Pi-OBd module and one or more OBD2 controllers. There is no connection to the real serial interface. ◀

(160204)

FROM THE STORE

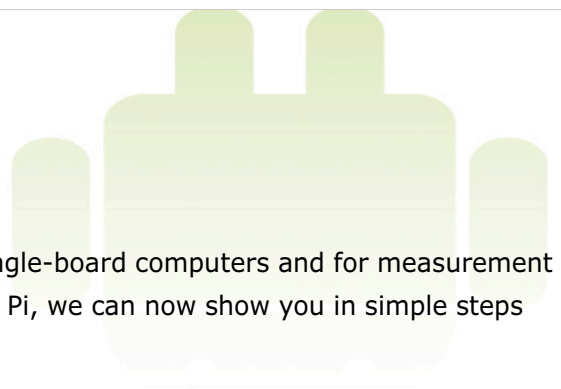
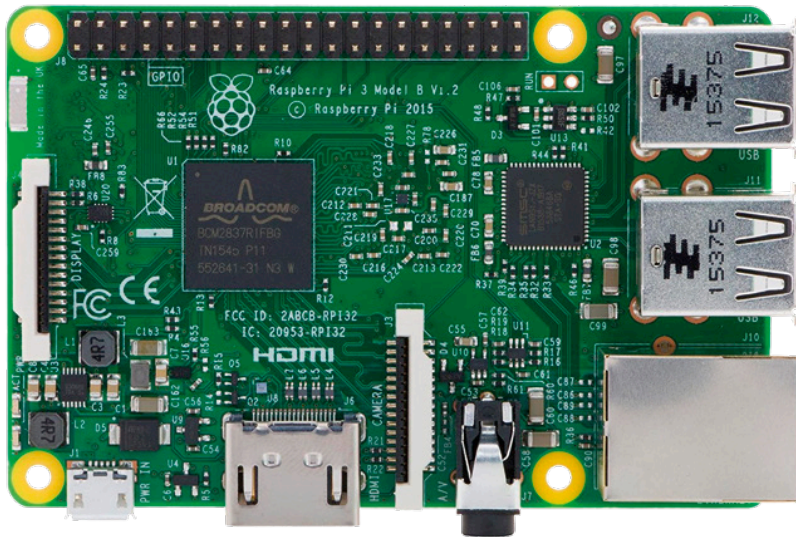
- SKU 17944
PI-OBd-HAT Module (built and tested)
- SKU 17415
Official 7"- Raspberry Pi touchscreen
- SKU 17631
Raspberry Pi 3

Web Links

- [1] HHEmu: www.elektormagazine.com/labs/firmware-update-and-emulator-for-obd2-analyser-ng-wireless-obd2
- [2] OBD2-Analyser NG: www.elektormagazine.com/magazine/elektor-200909/19167
- [3] DXM-Modul (German!): www.diamex.de/dxshop/DIAMEX-DXM-OBd2-Modul
- [4] Bluetooth for OBd-2: www.elektormagazine.com/magazine/elektor-201004/19297
- [5] HHGui: www.elektormagazine.com/labs/obd2-for-raspberry-pi
- [6] www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=66368
- [7] http://standards.sae.org/j1979_201408/
- [8] https://en.wikipedia.org/wiki/OBD-II_PIDs
- [9] www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=67245
- [10] Pi-OBd-HAT: www.elektor.com/pi-obd-hat-obd2-module-for-raspberry-pi

Android on your Rpi (1)

Using GPIO Pins for measurement and control functions



Google has released a new version of Android — especially for single-board computers and for measurement and control applications. Having tested Android on the Raspberry Pi, we can now show you in simple steps how to get a first 'Hello World' project to work.

By **Tam Hanna** (Slovakia)

Microsoft's rapid advance into Raspberry Pi territory has certainly caused Google some headache. After desktop computers, tablets and smartphones, the Internet of Things (IoT) looks like becoming the next battleground.

Pay attention to quality!

If you are ordering memory cards from China make sure you carry out detailed quality tests. Many manufacturers offload batches of inferior quality products that may well report the full complement of memory to your microcontroller. In actual use, however, your data may be lost.

With the Redmond brigade now well advanced along the road towards platform independence, convergence is the new keyword. Google simply cannot afford to ignore the IoT arena, as otherwise Microsoft could exploit this and open a second front on which to attack the smartphone market.

However, Google's *Brillo* platform and language for the Internet of Things announced about a year ago made little impact. Maybe the name was the problem. Whatever the reason, for its second attempt, Google knew it must go direct for the jugular and named its new product simply 'Android Things'.

Simple access to hardware

Let's begin with the obvious. Despite all the kerfuffle — comparisons with Microsoft's 'Windows 10 IOT Core' are purely

coincidental — Android Things is by and large a normal version of Android, albeit one that contains some extensions for interacting with hardware. **Figure 1** shows the structure of the operating system.

The *Things Support Library* incorporated with this consists of two modules. First up is the *Peripheral I/O API*, which at the time of writing provides access to the following four types of peripheral (whether OneWire will be retrofitted is currently unclear):

- GPIO with PWM
- I²C
- SPI
- UART

The second component is the *User Driver API*. We're talking here about a program-

ming interface that developers can use to make information on their own proprietary sensors available to the rest of the operating system. Further information on the modules included in the *Things Support Library* can be found at [1]. Let's venture now into some first attempts using practical hardware.

Preconfigured images

At press time Google (doubtless having learnt from its flop with *Brillo*) already offered ready-built Images for the Intel Edison, NXP Pico and Raspberry Pi 3 platforms. In this first article we shall focus on the (by far most widely selling) Raspberry Pi 3.

The first stage involves downloading the Image that awaits you at [2] and then burning it in the normal way onto a memory card (in this connection don't miss the **text panel** *Pay attention to quality*). For the steps that follow the author used an 8 GB capacity card. Using cards of greater capacity should not be a problem; on the other hand the Image size of 4.6 GB means that it will not work with cards any smaller than this.

The next step is to connect the screen and a network cable to your router on the RPi, and start it up by connecting the power supply. Note that a keyboard and mouse are not required, because Android in itself registers only one desktop, without relevant interaction possibilities. We do not want to get involved with debugging over WLAN, because the higher latency of wireless connections can lead to delays while you are tracking down for errors. Nevertheless, if you do want to try this, you will need to connect the RPi to the wireless network following the instructions given in [3].

The first time you start up Android Things on a Raspberry Pi 3 it will take a couple of minutes. At the very beginning, a message appears about Ethernet being missing; you can safely ignore this if the network cable is already connected. With this work done, the operating system presents the start screen shown in **Figure 2**. The IP address — 10.42.0.44 in this case — will be needed shortly.

Development environment

For space reasons, we have to assume at this point that you already have a working installation of Android Studio on your development PC. If this is not the case, please refer to [4] for help

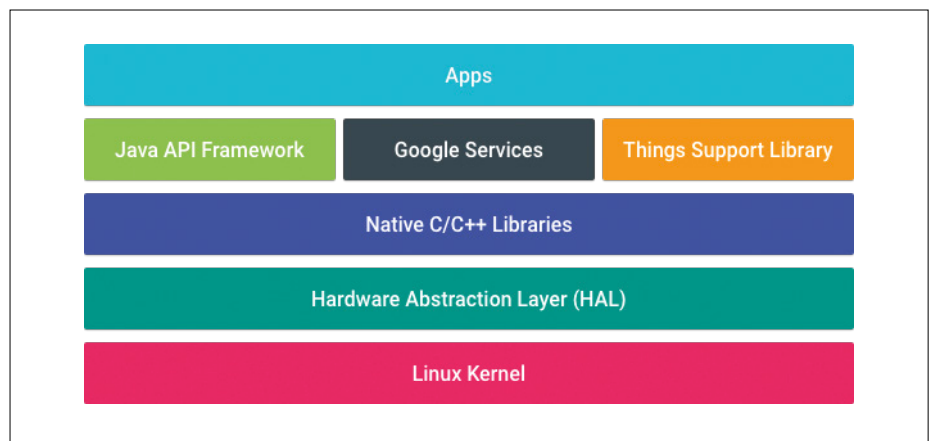


Figure 1. Android Things is a variant of Android (image credit: Google).

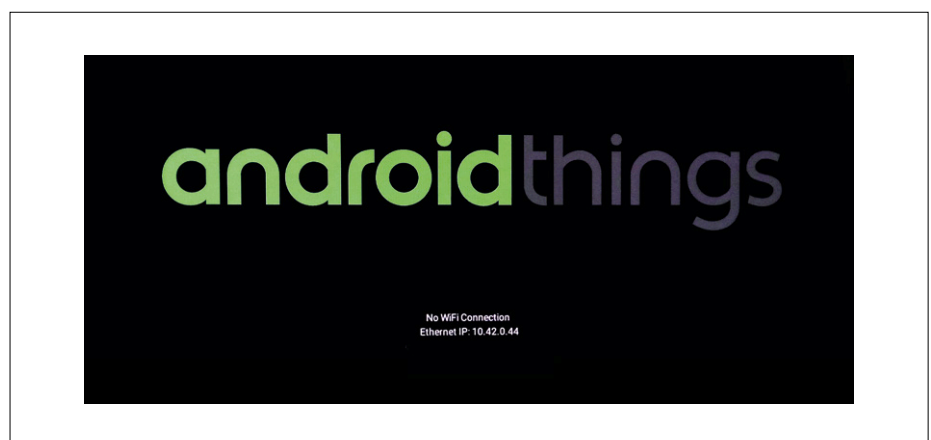


Figure 2. Our own 'Thing' is ready to put into use — remember to make a note of the IP address.

and advice. The author used an AMD eight-kernel workstation with Ubuntu 14.04 to perform the following stages (it's very similar under Windows and Mac OS).

In the next step, change to the root directory of the *Android Debug Bridge* (ADB) on the PC. Then connect the development computer to the RPi by typing the console commands shown in bold as follows:

```
tamhan@TAMHAN14:~/Android/Sdk/
platform-tools$ ./adb connect
10.42.0.44
```

connected to 10.42.0.44:5555

```
tamhan@TAMHAN14:~/Android/Sdk/
platform-tools$ ./adb devices
```

```
List of devices attached
10.42.0.44:5555 device
```

```
tamhan@TAMHAN14:~/Android/Sdk/
platform-tools$
```

Inputting adb devices is not absolutely necessary here — we are showing you the command because it allows you to identify all devices connected to the Android Debug Bridge. Also, remember that the connection between the ADB and the RPi can be lost, for example when your PC falls asleep.

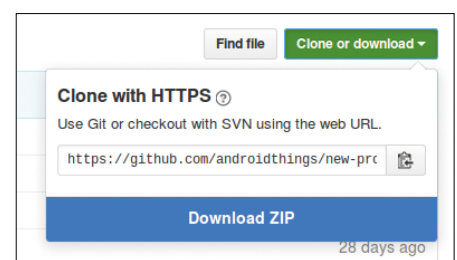


Figure 3. Using GitHub has always been a challenge and remains so.

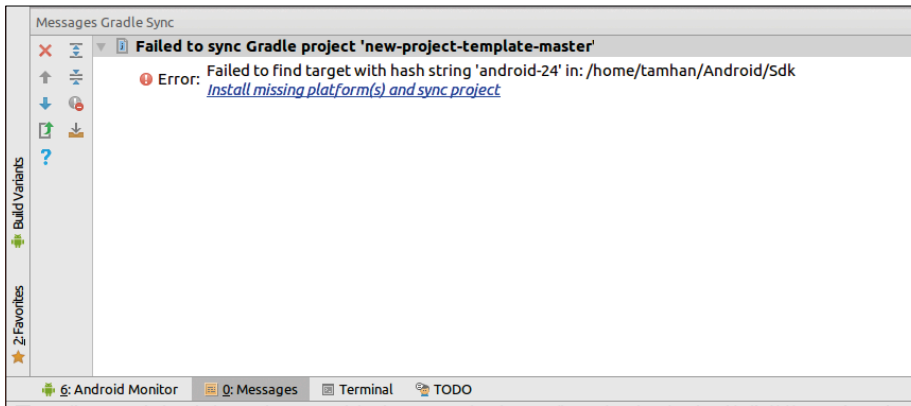


Figure 4. A compatible SDK is missing here.

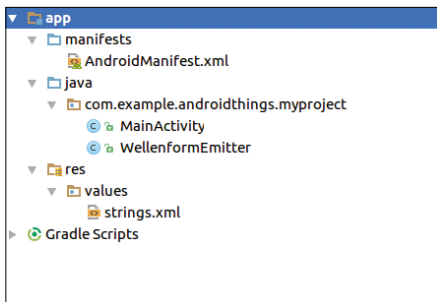


Figure 5. This application gets by without XML markup.

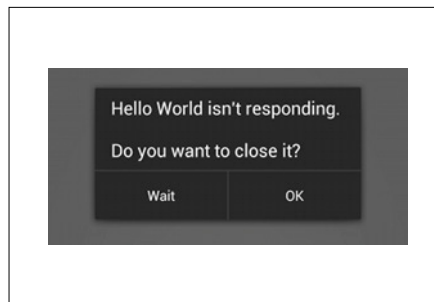


Figure 7. Anyone who blocks the GUI Thread will be punished (image credit: Google).

Android Things is still at an early stage of development and there is no template available yet in the project generator of Android Studio. In its absence you need to call up the existing GitHub Repository [5] and download a sample

project skeleton by clicking on the Button shown in **Figure 3**.

If you are already running a project in Android Studio now, close it by clicking *File → Close Project*. The IDE will then display the Welcome dialog. At this

point, you must extract the Archive that you downloaded in the previous step and move this into a convenient location in the file system. Then click *Open existing Android Studio Project* and cross over into the directory that contains this project. After you have clicked OK, the IDE will begin to deploy the project. As part of the synchronization of the Gradle project, the IDE will attempt to download any missing components automatically. If any problems arise, you will see an error message along the lines of **Figure 4**; clicking the hyperlink usually resolves the problem.

Do not be surprised if Android Studio complains more than once to report it has found obsolete versions. Android Things requires you to use the absolutely latest version of several components of the operating system, which must be installed in several steps.

After launching the IDE successfully, you should click *Build → Make Project* once more in order to launch a complete compilation. It is important to note that an Internet connection is required when you make the first compilation; afterwards you will also have a realistic chance of being able to work offline with your project skeleton.

Alles anders! Different strokes...

The MainActivity code serving as starting point is different from normal Android applications, because Google has added some extra logic calls. These are necessary because Android Things is able to work either with or without a monitor screen. In the latter case, thanks to the Log Calls, you do at least receive information in the debugger console.

Also note that the *res* directory is blank, as shown in **Figure 5**. There is no XML Markup supporting this activity. This affects the code as shown in **Listing 1** — the call to load of resources from XML normally available is missing here.

Another thing to concern us is the construction of the Manifest file, shown in abbreviated form as **Listing 2**. Two modifications are of interest here. Firstly, the Library section ensures that the Libraries mentioned above are inte-

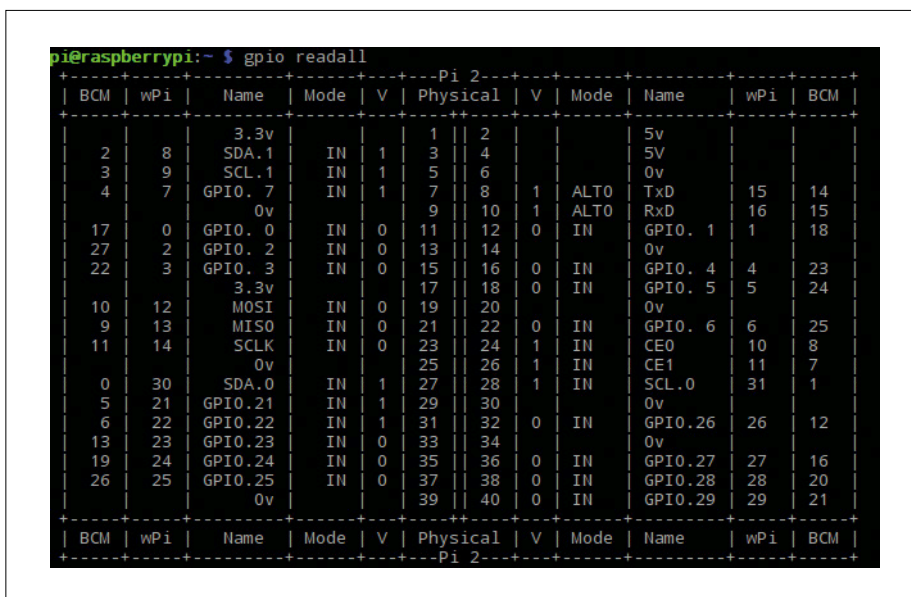


Figure 6. The table indicates the Pins of the Raspberry Pi.

grated into the project. Secondly, MainActivity has an additional *Intent Filter* written into it, which characterizes it as an entry point for Android Things devices.

This is relevant because Android Things has to make do without a program start. RPi is basically a 'one trick pony' that simply processes its host application.

Input and output

For our first demo let's turn to the GPIO Engine. There's an old saying stating that the real-time capability of an operating system decreases linearly (or even logarithmically!) with its complexity. Java-based systems are particularly disagreeable in this respect, with the *Garbage Collector* causing grief from time to time. At this point you must be utterly and completely forewarned that the Raspberry Pi is a 3.3 V platform — connecting 5 V subsystems will result in certain disaster!

With that said we can now turn our attention to the application code in MainActivity (**Listing 3**). First we declare two additional Member Variables. The Peripheral API is implemented in the form of a System Service. A Service is generally available across the entire operating system; applications that need to make use of it obtain a Referral and then interact with it. The Gpio Class is responsible for the actual switching on and off of the Pins.

The Function onCreate (Listing 3) is used to procure the Service. Even if Google is not 100 % orientated towards the Arduino where GPIO Functions are concerned, it's perfectly clear what's going on here. The BCM Strings can be assigned to individual Pins on the basis of the table in **Figure 6**. This table is obtained by entering the command GPIO Readall on a Raspberry Pi (loaded with the standard operating system Raspbian, not Android).

Start the music!

In order to determine the reliability or real-time efficiency of an operating system, it is a good idea to implement a periodically changing output on the GPIO Pin as quickly as possible. If you examine the spectrum on a modulation domain analyzer, you can draw some

Listing 1. Framework of the MainActivity — starting point of the program without user interface.

```
public class MainActivity extends Activity {
    private static final String TAG = MainActivity.class.
        getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy");
    }
}
```

conclusions. In theory, there should be nothing wrong with implementing periodic on and off-switching of the Pin directly in the OnCreated Function, which is called up at the start of the program. However, if you try this, you will be confronted with the error message shown in **Figure 7**.

The reason for this admittedly rather strange behavior lies in a peculiarity of the Android operating system: the

program's user interface is managed in a dedicated Thread called a GUI Thread. If you block this, the operating system punishes you without leniency by 'shooting down' your application. An endless loop would be a classic 'obstruction' that would certainly earn you no mercy.

To avoid this problem we can outsource our Routine in a further Thread. The simplest way of generating a

Listing 2. Manifest file with Library integration.

```
<manifest . . .>
    <application
        - - ->
        <uses-library android:name="com.google.android.Things"/>

        <activity android:name=".MainActivity">

            . . .

            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.
IOT_LAUNCHER"/>
                <category android:name="android.intent.category.
DEFAULT"/>
            </intent-filter>

        </activity>
    </application>
</manifest>
```

Listing 3. The Gpio Class responsible for switching Pins on and off.

```

public class MainActivity extends Activity {

    private static final String TAG = MainActivity.class.
getSimpleName();
    PeripheralManagerService service;
    Gpio myGPIO;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d(TAG, "onCreate");

        PeripheralManagerService service = new
PeripheralManagerService();
        try{
            myGPIO = service.openGpio("BCM6");
            myGPIO.setDirection(Gpio.DIRECTION_OUT_INITIALLY_LOW);
            myGPIO.setValue(true);
        }
        catch (Exception e){Log.d(TAG, "Fehler:" + e.getMessage());}
    }
}

```

Thread is to create a *Runnable*. So your first step is to create a new Class employing the following structure:

```

public class WaveformEmitter
implements Runnable {
    @Override
    public void run() {

    }
}

```

Listing 4. The endless loop of signal generation is performed in a Thread of its own — a Runnable serves as Container.

```

public class WaveformEmitter implements Runnable {
    Gpio myGpio;
    public WaveformEmitter(Gpio _which){
        myGpio=_which;
    }
    @Override
    public void run() {
        try {
            while(1==1){
                myGpio.setValue(true);
                myGpio.setValue(false);
                myGpio.setValue(true);
                myGpio.setValue(false);
                myGpio.setValue(true);
                myGpio.setValue(false);
            }
        }
        catch (Exception e){}
    }
}

```

Runnables are basically a kind of container in which we 'pack' the logic that executes the new Thread. In addition to the run Method described here, you can of course implement various Members in order to provide the information necessary for executing the Thread.

In its fully completed state the resulting Class now looks like **Listing 4**.

In addition to a Constructor responsible for accepting the GPIO Instance, we also have now populated the Method run(), which is what produces the actual waveform output. We generate a characteristic waveform from three rectangles: what's interesting in this instance is that the duration of the execution of the while loop is shown separately.

The question now arises how we activate the Runnable in OnCreated. A classic mistake made by novices is to invoke run() directly — doing this executes the code in the context of the activating Method (that means in the GUI Thread again). The correct way to activate a second Thread is to create a new Thread Class, with an Object indicated as *Payload* for the Parameter (**Listing 5**). The Thread is initiated using the start() Method.

We are now ready to toss the program in the RPi's direction.

Thanks to the ADB, which acts as an abstraction layer, it's sufficient to merely click on Run — the RPi behaves like a phone connected to the PC via USB. Since the MainActivity does not contain a user interface, a plain white screen appears if a monitor is connected, letting us know that our activity is cheerfully getting on with its work.

Debriefing

The next step is to connect your RPi to a modulation domain analyzer, so you can admire the screenshot shown in Figure 8 (the author provides an English language video at [6] giving further information on the function and benefits of using a modulation domain analyzer).

The most conspicuous feature, apart from the occasional jitter, is the formation of two prominent peaks. The region around 2.073 kHz shows the two 'linear sweeps' during the transition through the while loop, leading to the somewhat lower frequency of 'only' 2.062 kHz.

It's obvious that, in this situation, Android simply cannot match the pace of classic Unix; the interposed Java VM is taking its toll and slowing things down. You can read more on the real-time behavior of Android in various papers on the Internet (for instance in [7]).

The verdict

The program just created is a classic example of overkill in the embedded domain: a shrewd programmer could generate a stable waveform without any real-time operating system, using just a few lines of code and an IC.

Mind you, Bit banging and all that — let's be honest — is definitely not the intended application use of Android Things. On the other hand, the platform will always play its strengths when it comes to realizing demanding applications. In our next issue we'll show you how to interpret data from sensors and display the result in a diagram. Until then I wish everybody good coding! ◀

(160361)

Listing 5. Start of our signal generation Thread in MainActivity.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    . . .

    WaveformEmitter myEmitter=new WaveFormEmitter(myGPIO);
    new Thread( myEmitter ).start();
}
```

Web Links

- [1] <https://developer.android.com/things/sdk/index.html>
- [2] <https://developer.android.com/things/preview/download.html>
- [3] <https://developer.android.com/things/hardware/raspberrypi.html>
- [4] <https://developer.android.com/studio/install.html>
- [5] <https://github.com/androidthings/new-project-template>
- [6] www.youtube.com/watch?v=IBLEFVUVGyU
- [7] www.utdallas.edu/~cxl137330/courses/fall14/RTS/papers/4a.pdf

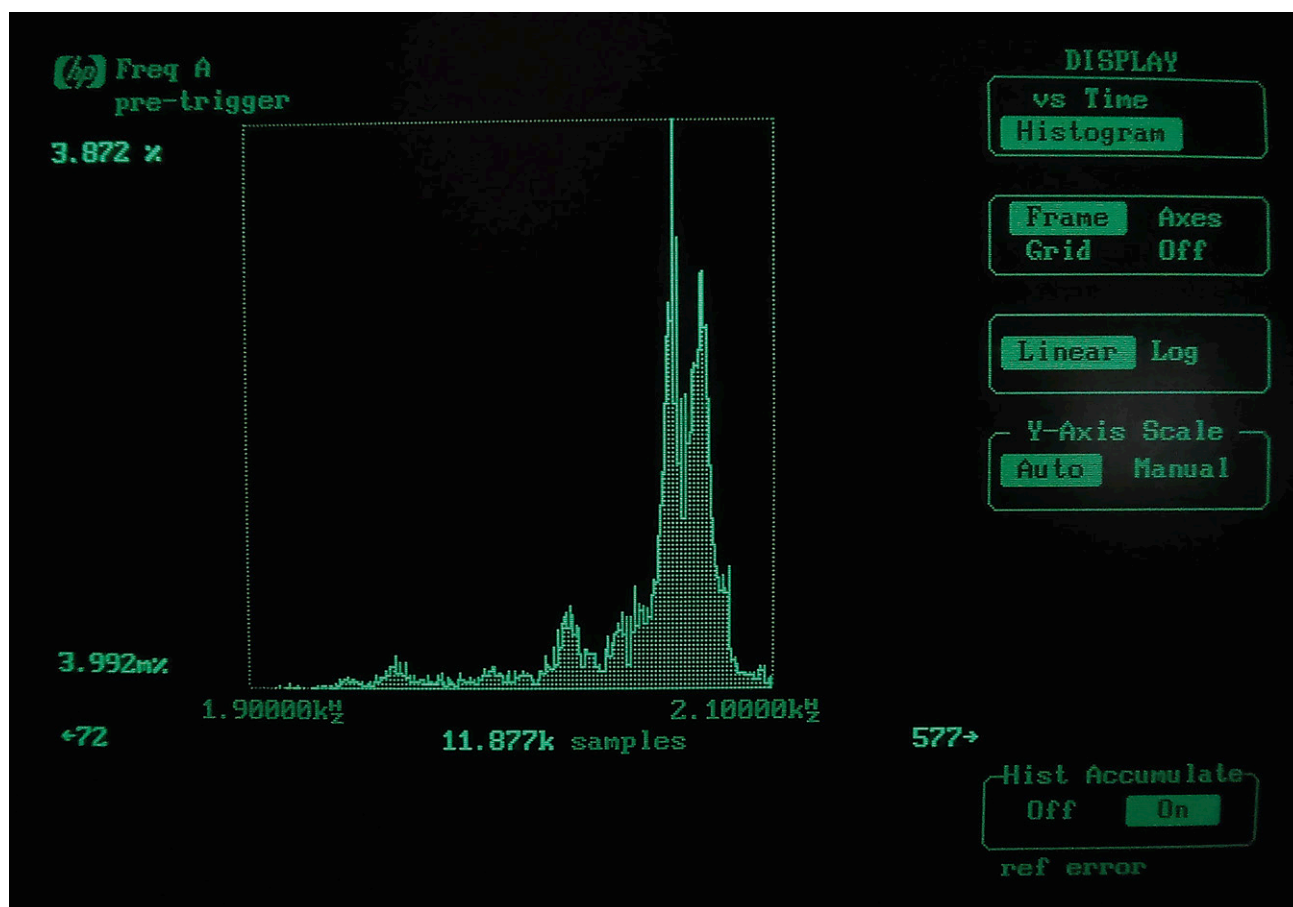
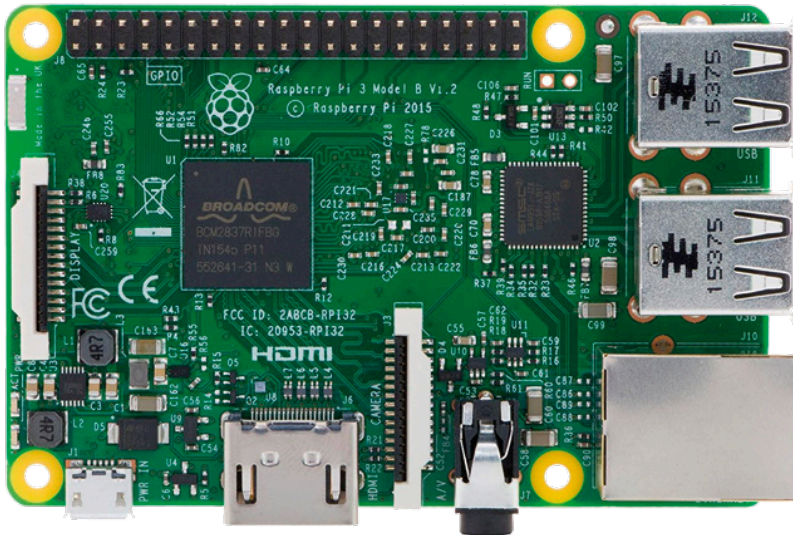


Figure 8. Arduinos generate significantly 'prettier' histograms in this application.

Android on your RPi (2)

Controlling a display using SPI



In our previous edition we got our first Android program running. In it we switched an output between High and Low in an endless loop in order to investigate its behavior in real time [1]. If we now wish to control peripherals, operating Pins 'manually' does not make sense — instead we use the Hardware Bus Interface of the Controller. In this article we show you how to control a small OLED display using SPI.

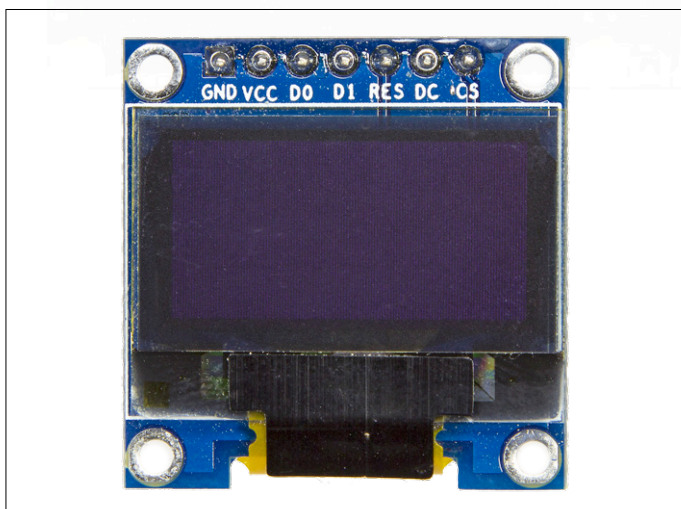


Figure 1. in terms of both current consumption and size this display is significantly smaller than an HD monitor.

By **Tam Hanna** (Slovakia)

If you plan to control peripherals with Android Things you are currently spoiled for choice between SPI and I²C. In the steps that follow we will use one of the widely available SPI OLED displays such as the part shown in **Figure 1**, which at time of writing is available from aliexpress.com for around \$2.50 with free shipping (and a long wait) from China.

Circuitry

From a hardware perspective a Raspberry Pi with Android behaves just like a regular RPi. The circuit shown in **Figure 2** is very similar to the one used in the series *Windows on the Raspberry Pi* [2].

Android Things identifies SPI buses (just as it does GPIO Pins) by using Strings. Our first exercise consists of identifying those available on the Raspberry Pi:

```

@Override
protected void onCreate(Bundle savedInstanceState) {

    . . .

    PeripheralManagerService manager = new
        PeripheralManagerService();
    List<String> deviceList = manager.getSpiBusList();

    if (deviceList.isEmpty()) { } else {
        Log.i(TAG, "List of available devices: " +
            deviceList);
    }
}

```

The program generates a list of all buses available in the Debugger Console. At the time of writing the author's Raspberry Pi 3 indicated two endpoints:

```

I/MainActivity: List of available devices: [SPI0.0,
    SPI0.1]

```

At this stage Gradle and Android Studio produce a minor trap for the unwary: if you plan to perform data extraction using a Breakpoint, you must be sure to start the program by clicking on the Debug Symbol. If you click on the normal Run arrow, the Debugger will not be embedded with the program.

When creating a fresh demo project our first task is to initialize the SPI Bus (it goes without saying that you can still download the code again from the Elektor website [3]). For this we use a new Method named `configureAndFindDevice`. First up we expand the Main Activity with a global GPIO Object, which controls the Reset connection on the display. We then apply an SPI device that encapsulates the actual communication between the display and our Pi.

```

public class MainActivity extends Activity {

    SpiDevice myDevice;

    Gpio myPinReset;
}

```

The following step covers the — more or less classic — means of initializing the SPI Bus within `configureAndFindDevice` (**Listing 1**). Invoking `setFrequency` specifies the communication frequency — in the author's tests both 4 MHz and 8 MHz worked fine without any problems.

Attentive readers who bothered to look at the listing may possibly wonder what `Thread.Sleep` in the GUI Thread is searching for. The answer is that a delay of 10 ms is of no consequence and accordingly does not justify the effort of generating a separate Thread.

To invoke the Method we use `onCreate`, so that it takes place at the start of the Program (**Listing 2**).

Next we initialize an additional GPIO Pin to facilitate control of the Data Control line. We also invoke the `initDisplay` Function, which is responsible for defining the various parameters of the

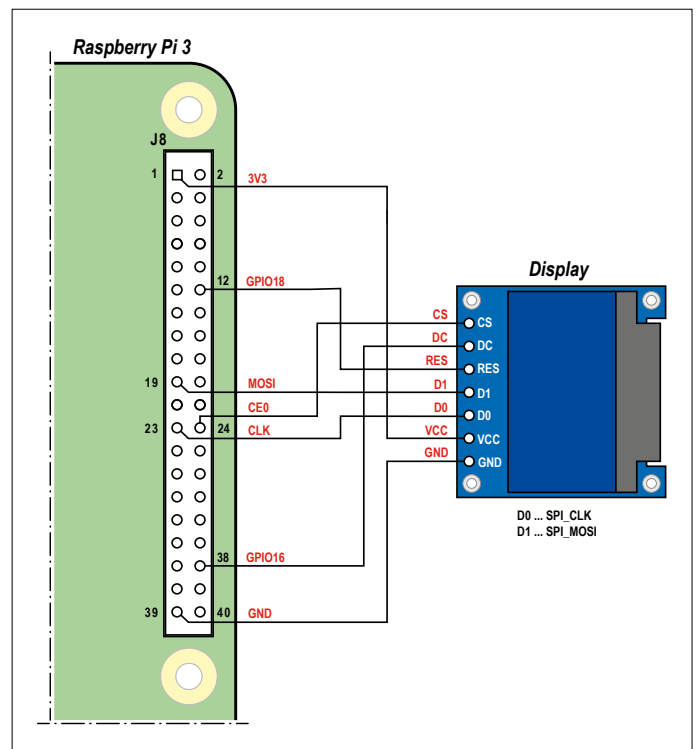


Figure 2. A handful of wires link the Raspberry Pi and the display module.

displays (the content of this Method follows in the next section). For the sake of total completeness we should note that the SPI device needs to be released when the Activity is ended — this is handled using the code in `onDestroy`, see **Listing 3**. If these `close` commands are missing, during Debugging

Listing 1. Initializing the SPI interface.

```

void configureAndFindDevice(){

    PeripheralManagerService manager = new
        PeripheralManagerService();

    try {

        myPinReset = manager.openGpio("BCM18");
        myPinReset.setDirection(Gpio.
            DIRECTION_OUT_INITIALLY_LOW);
        myPinReset.setValue(false);

        myDevice = manager.openSpiDevice("SPI0.0");
        myDevice.setMode(SpiDevice.MODE0);
        myDevice.setFrequency(4000000);
        myDevice.setBitsPerWord(8);
        myDevice.setBitJustification(false);
        Thread.sleep(10);
        myPinReset.setValue(true);
    }

    catch (Exception e) { . . . }
}

```


Listing 2. Initializing the pins uses the method *onCreate*, invoked when the program starts.

```
@Override
protected void onCreate(Bundle savedInstanceState) {

    . . .

    configureAndFindDevice();

    try {

        PeripheralManagerService manager = new
            PeripheralManagerService();
        myPinDataCommand = manager.
            openGpio("BCM16");
        myPinDataCommand.setDirection(Gpio.
            DIRECTION_OUT_INITIALLY_LOW);

    }catch(Exception x){}

    initDisplay();

}
```

you may occasionally meet Exceptions of the type `android.os.ServiceSpecificException`: BCM18 is already in use. Note too that it is also permissible to remove Activity Classes of the Android OS without interrupting an ongoing operation — whether you do this is up to your own personal fancy, which frequently causes strife in the world of Android.

Listing 3. Tidying up after the program ends.

```
@Override

protected void onDestroy() {

    super.onDestroy();

    Log.d(TAG, "onDestroy");

    try {
        if(myDevice!=null) myDevice.close();
        if(myPinDataCommand!=null) myPinDataCommand.
            close();
        if(myPinReset!=null) myPinReset.close();
        myDevice = null;

    } catch (IOException e) {

        Log.w(TAG, "Unable to close SPI device", e);

    }

}
```

Let initialization begin...

Having successfully configured the SPI Framework we can move forward to data transfer, in order to deploy the display controller into the correct mode and bring data onto the monitor screen. The data sheet for the SSD1306 is available widely but when using an IC for the first time it is also a good idea to check out the ready-to-use Libraries at Adafruit and in the documentation of various other manufacturers. These will show you how the chip is used in practice.

You can see the code for `initDisplay` in **Listing 4**.

Next we invoke `Thread.sleep` once more. This ensures that the Controller has sufficient time to start up properly following arousal from Reset status. After this we send three Commands; these look after activating the charge pump and switching on the display. Before invoking the command `sendData` (responsible for transmitting data for the display), we also need to make sure that the display Buffer contains some more or less random information.

The actual declaration of the Command Constants follows a somewhat strange procedure. It is interesting to note that before each individual Constant, a 'Cast' Byte is required. If this is missing, the Java compiler would otherwise set up a Tree:

```
private byte[] CMD_DISPLAY_ON = { (byte)0xAF };
```

Next, we can turn to the two methods that send information to the display controller. First a brief explanatory note: the SD1306 works as an ordinary SPI device, both when accepting data and when collecting commands. The level at the digital DC input determines the mode. If the DC signal is high during the transmission of data, the data is moved into the display

Listing 4. Displaying lines on the screen.

```
void initDisplay()

{

    try {

        Thread.sleep(100);
        SendCommand(CMD_CHARGE_PUMP_ON);
        SendCommand(CMD_MEM_ADDR_MODE);
        SendCommand(CMD_DISPLAY_ON);

        for(int i=0;i<myDisplayBuffer.length;i++){
            myDisplayBuffer[i]=22;
        }

        sendData();

    }

    catch(Exception x){Log.e("Elektor",x.
        getMessage());}

}
```

Buffer — if it is low, the information is instead written into the Command memory. This leads to the following code:

```
void SendCommand(byte[] _what){
    try {
        myPinDataCommand.setValue(false);
        myDevice.write(_what, _what.length);
    }

    catch (Exception x){
        Log.e("Elektor",x.getMessage());
    }
}
```

The version for transferring display data differs only minimally from `SendCommand`. Instead of accepting data from the Function Parameter, the Function serves as a Member of an Array (`myDisplayBuffer`).

```
void sendData(){
    try {
        myPinDataCommand.setValue(true);
        myDevice.write(myDisplayBuffer,
            myDisplayBuffer.length);
    }

    catch (Exception x){
        Log.e("Elektor",x.getMessage());
    }
}
```

And with this, the first version of our program is ready to put into use. Try it out now — if you have implemented it correctly, a series of horizontal lines will appear on the screen (**Figure 3**).

Now we'll communicate for real

We don't need a dedicated Real Time operating system to display line patterns; even on an AVR a display memory of just 1 KB can be used successfully for this task. But you can see how the Raspberry Pi shows its superior capabilities when the operating system is displaying complex information.

A genuinely interesting exercise would be to show GUI control elements (for instance a diagram) on the screen as a Bitmap. To do this let's embed *MPAndroidChart* in our program — *MPAndroidChart* is a high performance diagrams Library that also plays a role in the development of classic Android applications. The first step is to open the file *build.gradle* covering the entire project and enhance it using the following passage:

```
allprojects {
    repositories {
        jcenter()
        maven { url "https://jitpack.io" }
    }
}
```

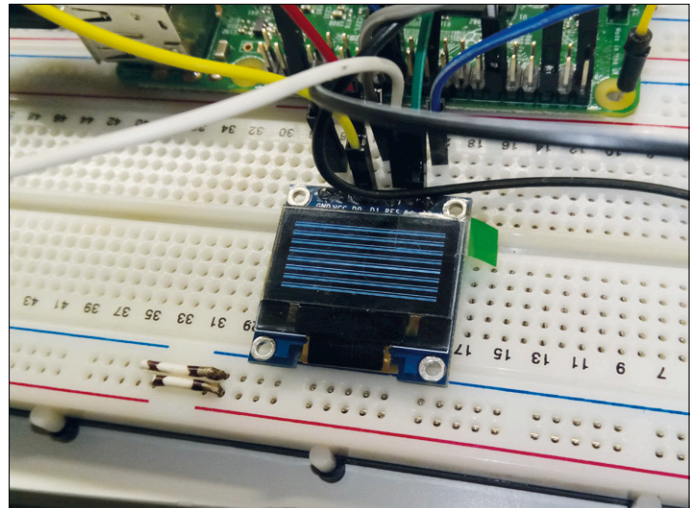


Figure 3. The test pattern confirms that everything is functioning perfectly. On the other hand a completely black OLED indicates it's 'dead'.

Next we expand the file *build.gradle* (*Module:app*) relating to the Application as follows:

```
dependencies {
    provided 'com.google.android.
        things:androidthings:0.1-devpreview'
    compile 'com.github.PhilJay:MPAndroidChart:v3.0.1'
}
```

After saving both files Android Studio superimposes a yellow banner, on which you now click on *Sync Now*. The IDE now actualizes the project in its entirety — one of the key strengths of Gradle is its ability to download Libraries and other necessary support materials from the Internet actually *during the compilation process*.

In the next step right-click on the folder for this project and select the option *New → File → Android Resource File*. In the dialog that now appears select the settings indicated in **Figure 4** so as to create a Layout File.

At this stage the Layout Subsystem implemented in Android would normally cause us grief, because it would dynamically

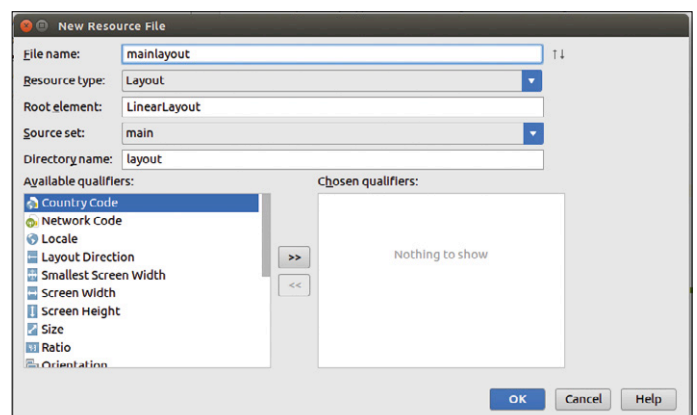


Figure 4. These Parameters produce a new Layout file.

adapt the size of the control elements shown. However, here we need to have a Widget having the size 128 × 64 pixels, which the following adjustments to the code will define:

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android=http://schemas.android.com/
    apk/res/android
    android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

    <com.github.mikephil.charting.charts.LineChart
        android:id="@+id/chart1"
        android:layout_width="128px"
        android:layout_height="64px"/>

</LinearLayout>
```

The circuitous detour via the main screen of the Raspberry Pi employed here is not absolutely necessary but it simplifies our work, as we can then view the results of this exercise on a studio monitor that we might also wish to connect.

In the next step we turn to the Chart Library, which of course requires a bit of help before displaying diagrams. This now looks as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.mainlayout);
}
```

The first difference from a typical Android Things application is the invocation of `setContentView` — this takes care of loading the information found in the *XML Layout*.

Next we create an Array that delivers more or less random display data:

```
ArrayList<Entry> values = new ArrayList<Entry>();

for (int i = 0; i < 20; i++) {
    float val = (float) (Math.random() * 25) + 3;
    values.add(new Entry(i, val));
}
```

Following this we define the parameter settings: *MPAndroidChart* is provided for handling large amounts of data, which is why the Initialization turns out to be relatively complex (and why we show only a few extracts):

```
LineChart mChart = (LineChart) findViewById(R.
    id.chart1);
mChart.getDescription().setEnabled(false);
LineDataSet s1=new LineDataSet(values,"");

. . .

ArrayList<ILineDataSet> dataSets = new
    ArrayList<ILineDataSet>();
dataSets.add(s1);
```

```
LineData data = new LineData(dataSets);
mChart.setData(data);
```

The primary difficulty, which leads to the enormous length of the code, is the elimination of unnecessary drawing commands. The display resolution of 128 × 64 Pixels is so small that the developers of *MPAndroidChart* probably never even tested this. Our next exercise consists merely of converting the result into a Bitmap that can be sent to the display. Here it is crucially important to discard the color information, because as you know, our screen is only monochrome (black and white).

On account of the comparatively bizarre drawing technique of *MPAndroidChart*, we need to slow down the conversion process. This can be accomplished using a Handler. The bare bones of the code used in the Handler are as follows:

```
final Handler handler = new Handler();
handler.postDelayed(new Runnable() {

    @Override
    public void run() {

        //Code ist hier

    }
}, 700);
```

`postDelayed` accepts an Object that needs to prepare a `run()` Method, in which we have the code whose execution needs to be delayed. This is very extensive, so we'll begin with the conversion of the Widget into an Instance of the Bitmap Class:

```
public void run() {
    LineChart mChart = (LineChart) findViewById(R.
        id.chart1);

    Bitmap returnedBitmap = Bitmap.createBitmap(128,
        64,Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(returnedBitmap);

    Drawable bgDrawable =mChart.getBackground();
    if (bgDrawable!=null)
        bgDrawable.draw(canvas);
    else
        canvas.drawColor(Color.WHITE);

    mChart.draw(canvas)
```

At this stage `returnedBitmap` contains a full-color version of the diagram rendered by *MPAndroidChart*. Our next task consists of converting this Bitmap into a Frame Buffer. The first challenge is to construct a bank of monitor screen Memory that is split into a group of 'pages'.

The simplest way of solving problems of this kind is to pick clean the Memory cells, step by step, using *for* Loops. In our case three Loops are necessary to do this:

```
byte counter=0;
byte store=0;
int counter2=0;
```

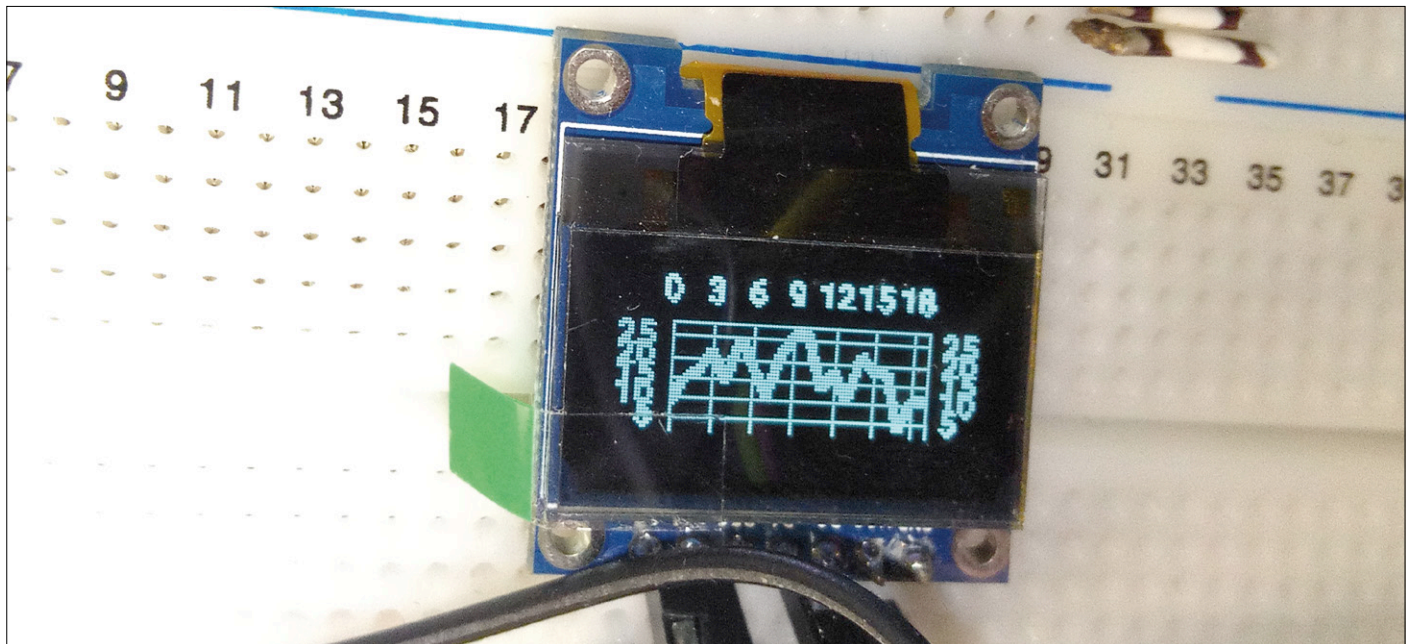



Figure 5. Thanks to Android we can achieve an attractive line graph with modest effort.

```
for(int ctr=0;ctr<8;ctr++)
    for(int x=0;x<returnedBitmap.getWidth();x++)
    {
        for (int y = ctr*8; y < (ctr+1)*8; y++)
        {
```

Next comes the thankless task of grouping every eight Pixels into a Byte. We start off by procuring the color information of the respective Pixels, so that we can then convert the color components, using simple comparison techniques, into 'black' or 'white':

```
int myVal=returnedBitmap.getPixel(x,y);
if(Color.red(myVal)!=255 || Color.
    green(myVal)!=255 || Color.blue(myVal)!=255)
    { //Set
        store=(byte)(store | ((byte)(1 <<
            (counter))));
    }
```

The actual outputting takes place in increments of eight. Using Bit operators, `store` is filled progressively with data — and as soon as an eight-Bit Word is 'complete', it is shifted into `myDisplayBuffer`.

```
        counter++;
        if(counter==8) {
            myDisplayBuffer[counter2]=store;
            store=0;
            counter=0;
            //Byte raus
            counter2++;
        }
    }
}
```

Last but not least, we direct a Command to the Display Controllers to establish a defined condition. `sendData` then passes the data via the SPI Bus to the display:

```
SendCommand(CMD_RESETCOLADDR);
SendCommand(CMD_RESETPAGEADDR);

sendData();

}
```

The reward for your efforts is the diagram shown in **Figure 5**.

And finally

We've reached the end of our experiments with Android Things. It's a platform that certainly won't appeal to all users, but anyone prepared to get to grips with it will be enriched with an immense range of new capabilities. With classic eight-bit machines many tasks are impossible — or virtually impossible — to bring to reality. Whereas putting this kind of high-performance 32-bit process computer on the job really saves you time and effort. And with this in mind we wish you plenty of success! ◀

(160369)

Web Links

- [1] www.elektormagazine.com/160361
- [2] www.elektormagazine.com/150520
- [3] www.elektormagazine.com/160369

Small but Perfectly Formed: the Raspberry Pi Zero W

First steps for the newest arrival in the Pi family

The mini version of the Raspberry Pi with the suffix Zero has been with us since the end of 2015.

February 2017 saw the advent of the Zero W (W for 'wireless') model, enhanced with WLAN and Bluetooth capabilities.

In this article, we examine how to get started with the new board started, what it does, and the advantages of adding wireless technology to this tiny single-board computer. As a practical example, we'll hook up a temperature sensor and distribute the data by WLAN using the MQTT protocol.

By **Markus Ulsass** (Germany)

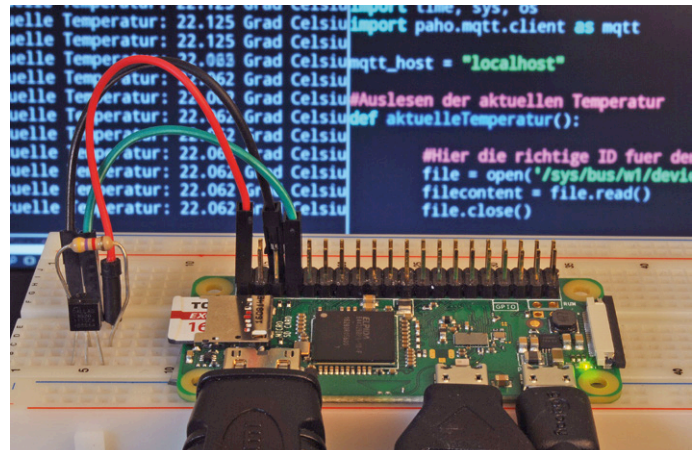
At approximately €25 / £10 / \$20 (prices vary from country to country) the cost of the new Raspberry Pi Zero W may be double that of its basic predecessor but the higher price is fully compensated by its cordless WLAN and Bluetooth capabilities, which are indispensable for so many projects. Low-cost robotic, smart home and Internet of Things (IoT) projects become feasible, thanks to the compact form factor and very low power consumption of the Zero W. The single-core Broadcom SoC BCM2835 chip is clocked at 1 GHz (512 MB RAM). This makes the Zero W not as speedy as its bigger brother Raspberry Pi 3 but nevertheless more than adequate for most applications.

The British shop Pimoroni [1] sells to customers worldwide; their prices are keen even when you include the postage cost and delivery is snappy at the time of writing.

The Zero W is equipped with Mini HDMI and Micro USB On The Go (OTG) ports; the single board device is supplied with a bare (unpopulated) 2x20-pin (HAT-compatible) GPIO interface. This needs a 40-way header to complete and for that reason, if you don't already have something suitable in your store cupboard, it's worth ordering the corresponding accessory parts for your Pi at the same time (dealers offer these as an adapter kit).

Installing the operating system and what comes next

Having got hold of a Zero W, together with all the necessary adapters and a power supply (see the shopping list panel for the necessary accessories), we now need to install the operating system on a micro SD card with at least 8 GB of memory.



To do this we download the current file Image of *Raspbian Jessie with Pixel* from [2]. For transferring the Image onto an SD card we need a card burner tool such as *Etcher*, which can be downloaded at [3] (**Figure 1**). Incidentally, a key advantage of this program is that it is available for all operating system platforms and the downloaded Image does not have to be unpacked. Having installed and started the program, we next select the appropriate Image and transfer it then to the chosen SD card.

After this we insert the SD card into the Zero W, we use an adapter to connect the monitor (using HDMI) together with a keyboard and a mouse (using USB) and finally connect up the power supply. Take special care here: the connector for this is the right-hand one of the two Micro USB sockets, assuming you have the board horizontal in front of you, with the SD card on the left-hand side (**Figure 2**). The Raspberry Pi Zero W should now start up in the Pixel desktop of the Raspbian operating systems. And with that, the task of installing an oven-ready Linux operating system on this super-small computer is complete.

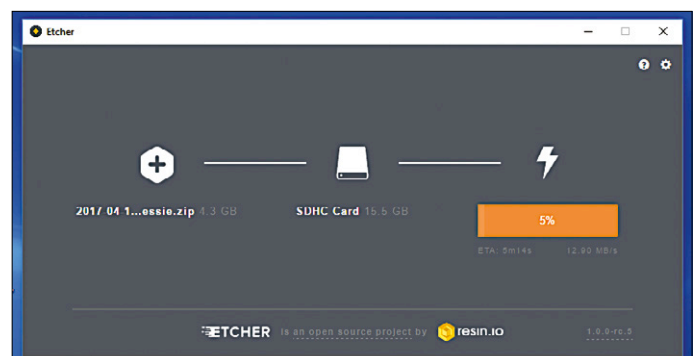


Figure 1. Etcher is an SD Card burning tool for loading the Raspbian image.

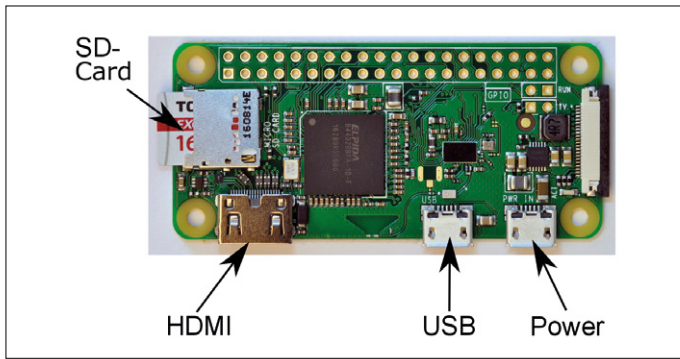


Figure 2. Connections for the Raspberry Pi Zero W, with the SD card at far left.

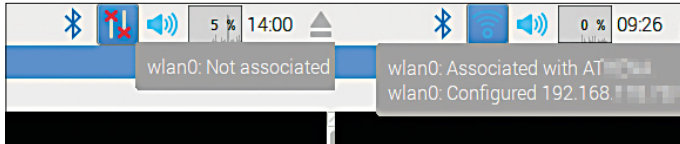


Figure 3. Before (left) and after (right). The Zero W should link up to the WLAN following configuration. Frequently you'll see additional information when you move the mouse over the symbols.

Fine tuning

If you now open a Terminal (**CTRL+T** or click on the Monitor icon in the upper navigation bar), you'll see that the default language, regional settings and keyboard layout assume that you are located in Britain. If you need to change these, type `sudo raspi-config` into the Terminal. This opens the *Raspberry Pi Software Configuration Tool*, in which you use the cursor to select (only if necessary) **4 Localisation Options** and press **Enter**. If, for example, you are in Germany, just select **I1 Change Locale de_DE.UTF-8 UTF-8** with the space bar and use the **Tab** to select **<OK>**. In the following window set `de_DE.UTF-8` as the *Default locale for the system environment* and confirm once more using **<OK>**. You can now restart the Pi Zero W using `sudo reboot`.

Your system's default language is now German but the key-

board layout is still incorrect. Call up `sudo raspi-config` once more, select **4 Localisation Options** and this time select **I3 Change Keyboard Layout**. There you may choose *Generische PC-Tastatur mit 105 Tasten (Intl)*, in the next window *Andere*, then *Deutsch* and confirm with **<OK>**. Move the cursor to *Deutsch – Deutsch (T3)* and confirm once more using the **Enter** key. In the next window confirm again with **<OK>** or **<Nein>** as appropriate and finally return to the main menu. When you leave this using **<Finish>**, you enter a **z** on the Console and should find it set up for the German keyboard layout. Obviously you can follow a similar process for any other languages and regional settings that you wish to specify — the above information is given with Elektor Magazine's wide distribution in Europe in mind. As the Zero W has a default password of *raspberrypi* for the user *pi*, it's vital that you change these settings. This is done using `sudo raspi-config` and the entry **1 Change User Password**. Provide a new password and make a written note of this somewhere.

In order to identify your Zero W subsequently on the network unambiguously by name, you should change this (the default is *raspberrypi*). To do this you edit the relevant entry in two text files, using the *nano* text editor. First enter `sudo nano /etc/hostname` and in the text file replace the entry *raspberrypi* with *zerow*. Save this with **CTRL+O**, **Enter** and finalize with **CTRL+X**. Next you open the second file with `sudo nano /etc/hosts` and search for the last entry with *127.0.0.1*. Here too you again replace *raspberrypi* with *zerow*. Now save it again using **CTRL+O**, **Enter** and finalize with **CTRL+X**.

External connections

In order to be able to link our Zero W to a local network using Wi-Fi, we now have to enter the data for the WLAN router (SSID, password). To do this click on the symbol with den two vertical lines and red crosses at upper right on the navigation bar (**Figure 3**). Choose the correct access point, enter the password and the Raspberry Pi Zero W should connect to the WLAN.

With a fresh Image it makes sense update the package

Listing 1. Flashing LED Python script.

```
#!/usr/bin/python
#LED_Blink.py
import RPi.GPIO as GPIO #link GPIO library
import time #library required for Sleep

LED = 14

GPIO.setmode(GPIO.BCM) #use BCM-GPIO labels
GPIO.setwarnings(False) #disable warnings
GPIO.setup(LED, GPIO.OUT) #use LED pin as output
PAUSEON = 1.0 #On time
PAUSEOFF = 1.0 #Off time

while True:
    GPIO.output(LED, GPIO.HIGH) #LED on
    time.sleep(PAUSEON) #On time
    GPIO.output(LED, GPIO.LOW) #LED off
    time.sleep(PAUSEOFF) #Off time
```



SHOPPING LIST

- Raspberry Pi Zero W
- Micro SD card (min. 8 GB)
- HDMI to mini HDMI adapter
- Micro USB OTG adapter (e.g. Delock 65296)
- Micro USB power supply (min. 1 A)
- 2 pcs 20-pin header strips (0.1" pitch)
- Breadboard
- Jumper cables
- LED (red)
- Resistor 470 Ω
- Resistor 4.7 kΩ
- Temperature sensor DS18B20

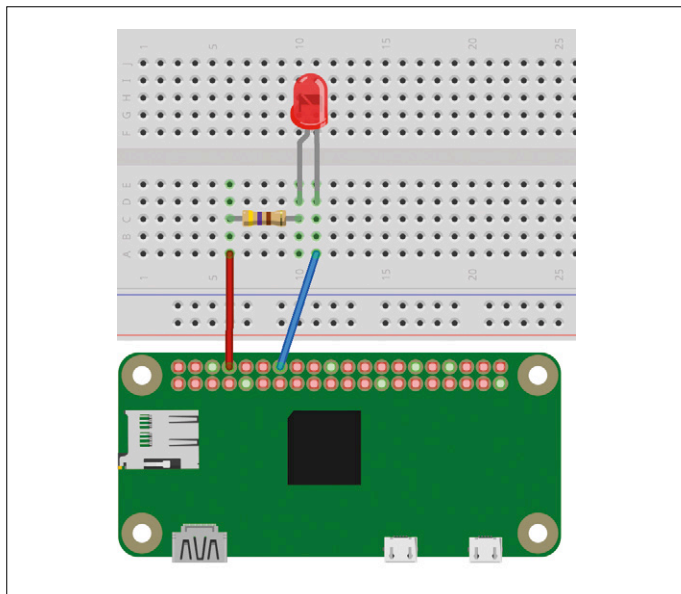


Figure 4. Fritzing sketch for hooking up the LED to the Zero W.

sources and the system to the most recent version. We do this with `sudo apt update && sudo apt upgrade`, again in the Terminal. Confirm the question whether to proceed with Y and wait a couple of minutes until all of the downloads are complete.

'Hello World' ersatz with an LED

With these initial preparations finished, we will skip typing 'Hallo World' and instead get started by connecting some simple hardware. First we need to solder the 40-pin header to the Zero W board, so that we can use some jumper wires and breadboard for controlling an LED.

Before connecting the resistor and the LED we need to shut down the Zero W completely, to avoid any short circuits or other harmful effects that might occur when hooking up hardware direct to the GPIO connections. We take care of this using `sudo shutdown -h now` on the Console or by clicking on the raspberry symbol on the navigation bar, selecting *Shutdown* in the drop-down menu and choosing *Shut Down* in the window that follows.

Forgoing the urge to type 'Hello World', we employ GPIO14 for control purposes along with a 470-Ω resistor and a red LED. You can find the Pinout for the Zero W at [4] for example. We connect GPIO14 to the resistor, attach this to the anode of the LED and link its cathode to one of the Ground Pins on the Pi Zero W. **Figure 4** shows the Fritzing sketch and **Figure 5** demonstrates the assembly as it actually appears.

For a program to make the LED flash we use a small Python script (`LED_Blink.py`). It is printed out in **Listing 1** but you can also download it from the Elektor website at [5].

We can generate the code using once more the Editor *nano* or alternatively using the somewhat more convenient *Geany* (*Menu* → *Development* → *Geany*), which is included in the Raspbian operating system by default. Note that in Python you make indentations (as in listings for instance) using the space bar, otherwise you'll see the message `IndentationError`. After saving the program we call it up on com-

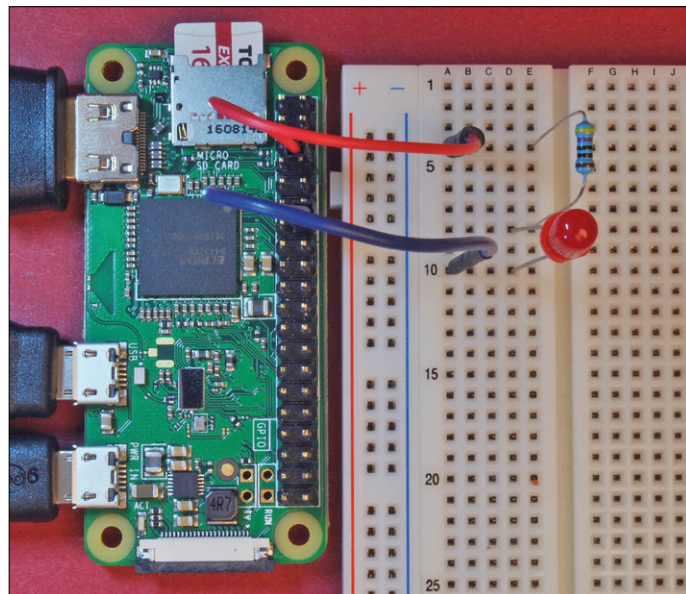


Figure 5. Zero W cabled to the LED and resistor on the breadboard.

mand line in Terminal with `python LED_Blink.py`. The LED should now flash once a second.

We close the program with `CTRL+C`.

Temperature measurement with the DS18B20

A popular and simple-to-connect digital temperature sensor is the DS18B20 from Maxim Integrated. First we shut down the system in order to link the circuitry safely to the breadboard and the Zero W.

The left-hand Pin (with the part number facing upwards) of the DS18B20 is linked to the 3.3 V power supply of the Zero W, the right-hand one to a Ground Pin. A jumper wire goes from the center (data) Pin to GPIO4. Lastly we provide a pull-up resistor between the data Pin and the supply voltage to supply 'parasitic' power to the sensor (see the Fritzing schematic in **Figure 6**).

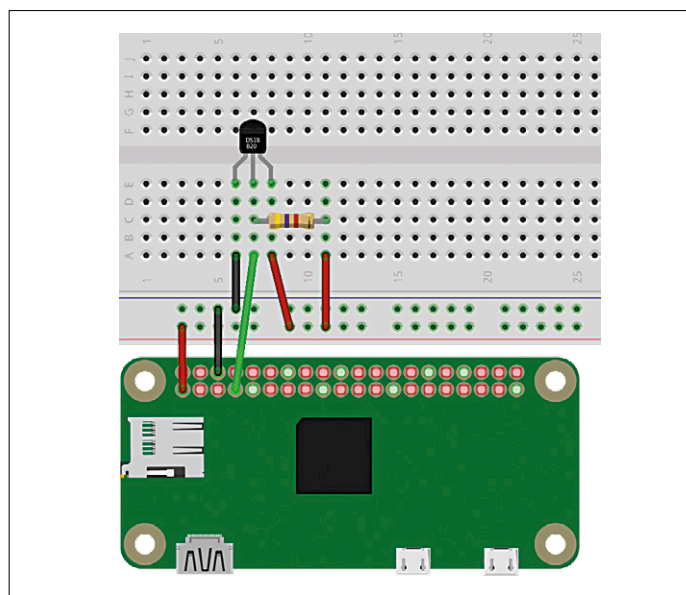
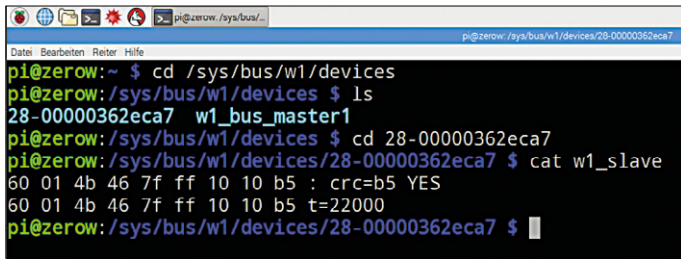


Figure 6. Wire connections for the DS18B20 temperature sensor.



```
pi@zerow:~$ cd /sys/bus/w1/devices
pi@zerow:/sys/bus/w1/devices$ ls
28-00000362eca7 w1_bus_master1
pi@zerow:/sys/bus/w1/devices$ cd 28-00000362eca7
pi@zerow:/sys/bus/w1/devices/28-00000362eca7$ cat w1_slave
60 01 4b 46 7f ff 10 10 b5 : crc=b5 YES
60 01 4b 46 7f ff 10 10 b5 t=22000
pi@zerow:/sys/bus/w1/devices/28-00000362eca7$
```

Figure 7. Testing the sensor after connection.

After making these connections and a quick check whether the wiring is correct, we can restart our Zero W. But wait: before we can use the one-wire temperature sensor, we must first integrate (include) it with the system. We do this using *sudo raspi-config* under *P7 1-Wire*, after which we answer *Yes* to the question *Would you like the one-wire interface to be enabled?* We then close the configuration program and must reboot the system.

It's time now to test whether the sensor has been embedded into the system correctly and is connected (available). For this we use the Terminal to switch into the *cd /sys/bus/w1/devices* Folder and list its contents with *ls*. This should make an alphanumeric combination appear that represents our sensor.

We now navigate to *cd <Sensor-ID>* (in this set-up we're using *cd 28-00000362eca7* in the relevant subfolder) and read out the sensor's current data using *cat w1_slave*, which is displayed in two lines (**Figure 7**). Here *t=22000* indicates a temperature of 22 degrees Celsius (centigrade). With this now done, the sensor is configured correctly and fully operational.

Sharing our temperature sensor with the rest of the world using MQTT

There will be times when we wish to forward sensor values

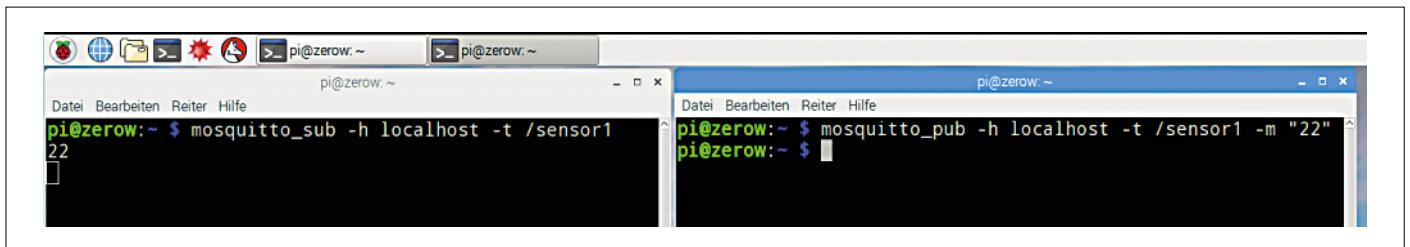
captured with our Raspberry Pi Zero W to other recipients. For this we employ the streamlined MQTT protocol – a data protocol for machine-to-machine communication. MQTT uses a so-called publish/subscribe (pub/sub) system. With this for instance sensors can 'publish' data on specific topics and users known as Clients can sign up ('subscribe') for these topics and receive the data. The topics are memorized to the data file using included slashes (an example would be *Apartment/Kitchen/Temperature*) and even wildcards are possible. Instead of files, however, we specify different sensors or actuators.

Our own temperature sensor might transmit the temperature values it captures under the topic */home/outdoors/temperature/sensor1*, for which a Client (for example a home automation system like openHAB, fhem or Node-Red) has subscribed.

For MQTT we require an intermediary – called the *Broker* in the MQTT jargon – that organizes the data flow between sender and receiver. A popular Broker is Mosquitto, which we install on our Zero W using *sudo apt install mosquitto mosquitto-clients*.

To check that everything has been installed properly we can test both transmitting and receiving data on the same device. For this we first start up the Broker with *sudo systemctl start mosquitto* on the Console. In order to activate it automatically the next time we boot up we also enter *sudo systemctl enable mosquitto*.

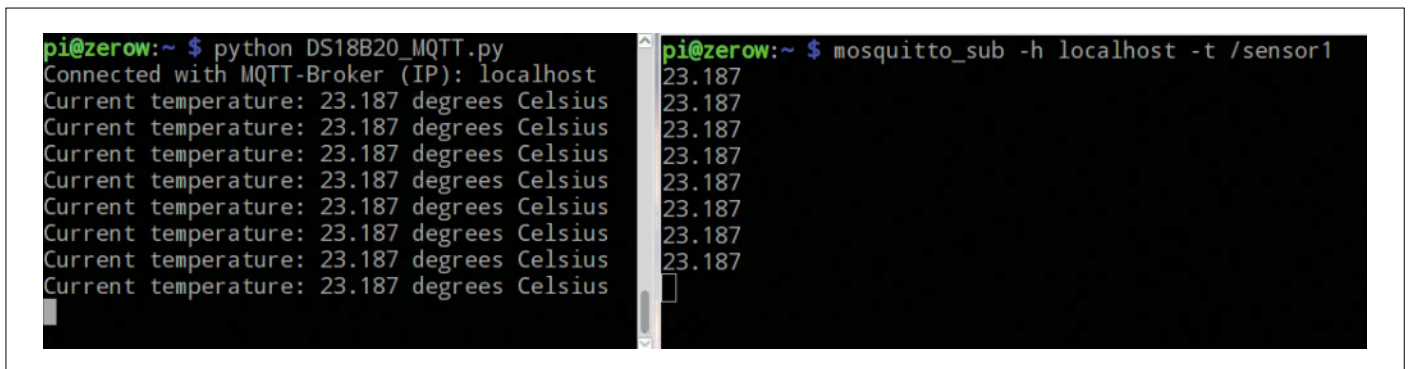
Next we'll do some simulating on the Console with *mosquitto_sub -h localhost -t /sensor1* (in the examples that follow we will save ourselves some writing effort and abbreviate the Topic shown above): first an information subscriber, who will later receive data (*mosquitto_sub*). First of all nothing happens, because the system is of course waiting for some data. We open a second Terminal window and enter *mosquitto_pub -h localhost -t /sensor1 -m "22"* (**Figure 8**). Doing this is the same as if we were transmitting



```
pi@zerow:~$ mosquitto_sub -h localhost -t /sensor1
22

pi@zerow:~$ mosquitto_pub -h localhost -t /sensor1 -m "22"
```

Figure 8. Testing the MQTT function.



```
pi@zerow:~$ python DS18B20_MQTT.py
Connected with MQTT-Broker (IP): localhost
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius
Current temperature: 23.187 degrees Celsius

pi@zerow:~$ mosquitto_sub -h localhost -t /sensor1
23.187
23.187
23.187
23.187
23.187
23.187
23.187
23.187
```

Figure 9. Testing MQTT data transfer with our temperature sensors: local output (on the left), received data using MQTT (on the right). [Celsius = centigrade]

a temperature value (mosquitto_pub) or respectively Publishing it to remain in the MQTT vocabulary. Following this a 22 should appear in the first window. Our system for publishing the value from our temperature sensor is now up and running, and in a final step we can automate it even further.

Transmitting temperatures by MQTT

We employ a Python script for automating data transfer by MQTT. For this we use a Library that we install using *sudo pip install paho-mqtt*.

Our script (**Listing 2**) first imports the required Modules and defines the IP address of the computer on which Mosquitto Broker is running ('localhost' for the local broker, actual IP address for a remote MQTT broker).

The current temperature value is then read out in the Function `currentTemperature()`. For the sensor ID given in the sample code you need to enter the ID of the sensor used of course. The date read out is converted into a suitable value and displayed as the value returned by the Function.

Following this a connection is made to the local (or remote) MQTT broker (mqtt_host). In the *while* loop the temperature value (temperature) is then polled every two seconds, displayed on the command line and transmitted to the MQTT broker under the topic /sensor1.

After we have created the program with an Editor or downloaded it (according to your preference), we open a Terminal and initiate it with *python DS18B20_MQTT.py*. The command line should now report a connection between the MQTT broker and the local Zero W, followed by a display of the current temperature value (**Figure 9**). Now we generate a second Terminal screen and Subscribe ourselves again by MQTT-Client to the Topic /sensor1 on the local computer: *mosquitto_sub -h localhost -t /sensor1*.

And that's still not everything...

With the Zero W the Raspberry Pi Foundation has delivered a truly fantastic board, right on time for the fifth birthday of the original Raspberry Pi. Who would have thought, five years ago, that a Linux nanocomputer like this could be possible, in such a tiny form factor and for such an affordable price?

The USB port, blocked on predecessor versions by the keyboard or the WLAN module, is now available for additional functionality. Even the current consumption is very modest and is already almost in the same league as self-powered Internet of Things or Smart Home devices. A comparison

Listing 2. Python script for transmitting the temperature by MQTT.

```
#!/usr/bin/python
#DS18B20_MQTT.py
#Import of required module
import time, sys, os
import paho.mqtt.client as mqtt

mqtt_host = "localhost"

#read out current temperature
def currentTemperature():

    #enter correct sensor ID here
    file = open('/sys/bus/w1/devices/28-00000362eca7/w1_slave')
    filecontent = file.read()
    file.close()

    #convert temperature value
    temperaturestring = filecontent.split("\n")[1].split(" ")[9]
    temperaturevalue = float(temperaturestring[2:]) / 1000
    return(temperaturevalue)

def on_connect(client, userdata, flags, rc):

    print("Connected with MQTT Broker (IP): " + mqtt_host)

client = mqtt.Client()
client.on_connect = on_connect

client.connect(mqtt_host, 1883, 60)

client.loop_start()

while True:
    time.sleep(2)

    temperature = currentTemperature()
    print("Current temperature: " + str(temperature) + " degrees Celsius")
    client.publish("/sensor1", temperature)
```

chart of the current drawn by various types of Raspberry Pi can be found at [6].

With such amazing benchmark parameters it's not surprising that the board can be found only in limited quantities at present. That was also the case with the first Raspberry Pi and therefore we can only hope that the availability of Zero W boards improves soon. ◀

(160451)

Web Links

- [1] <https://shop.pimoroni.com/>
- [2] www.raspberrypi.org/downloads/raspbian/
- [3] <https://etcher.io/>
- [4] <https://pinout.xyz/>
- [5] www.elektormagazine.com/160451
- [6] https://blog.adafruit.com/2017/03/03/how-much-power-does-pi-zero-w-use-piday-raspberrypi-raspberry_pi/

The I²C Bus

Part 1: The protocol

By **Josef Möllers** (Germany)

The inter-IC bus, or I²C bus for short, is a two-wire bus designed for connecting simple peripherals that do not require rapid transfer of large amounts of data. In this three-part series we will look at the essential features of the bus and see how it can be used in practice with a Raspberry Pi, Arduino or other system.



This short three-part series will not describe the I²C bus (also known, for example in the Atmel world, as the two-wire interface or TWI) down to the last detail, but instead aims to show how easy it is to build the bus into a design and how straightforward it is to use. In this first installment we will examine the data protocol; in the second installment we will test out the I²C bus on a range of systems from an Arduino and a Raspberry Pi to a full-fat PC; and then in the final part we will look at some typical I²C slave devices and give some ideas to show how it is possible to track down data transfer errors in a system.

The I²C bus was developed in the 1980s by Philips. If you would like to fully understand the I²C bus at the hardware level, rather than just as a black box under the control of ready-made software libraries, you will want to download

the official specification from NXP [1] and commit to some serious reading.

A huge range of resources on the subject of the I²C bus is available on the Internet. One good example is [2], by the Hamburg-based consultancy Telos, which covers practically all aspects of communication on the bus.

A little light physics

The usual scenario is where a programmable microcontroller is used to control one or more peripheral components with fixed functions, such as a port expander, a real-time clock or an EEPROM. The microcontroller is the master, providing the bus clock (see below), while the peripheral devices are slaves. But it is also possible to connect two microcontrollers together using an I²C bus, with one of them taking on the

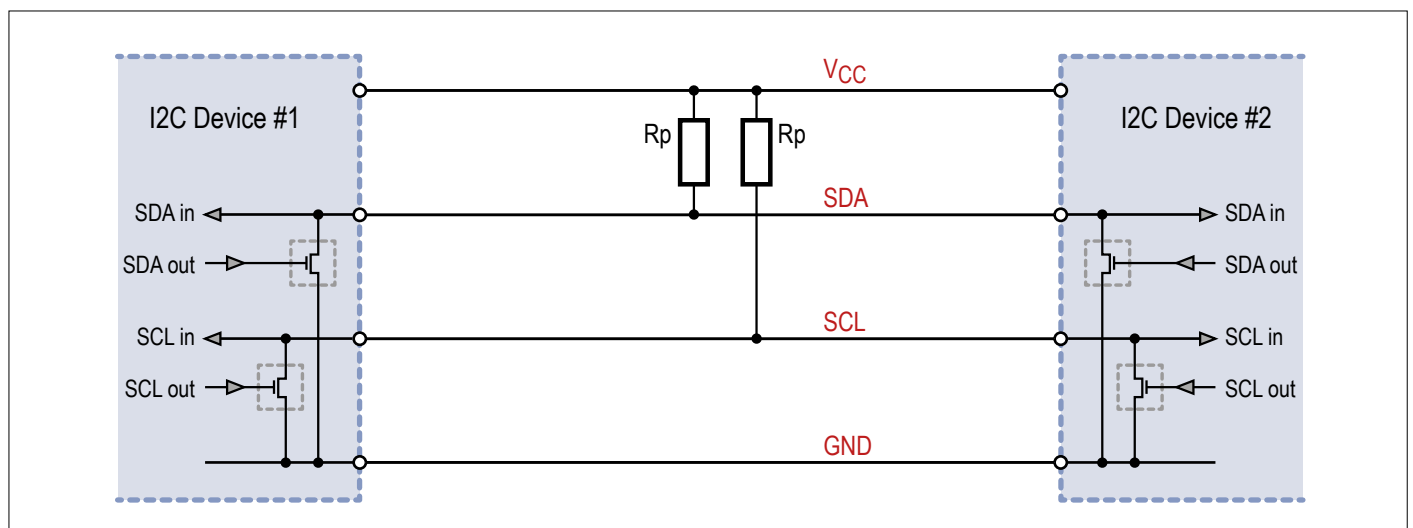


Figure 1. The I²C bus is based on a wired-AND connection.

role of the bus master. In the case where there are several slave devices, they are all connected to the same bus connections. An I²C bus can also have multiple master devices. The two wires that make up the I²C bus carry a clock signal (SCL) and a data signal (SDA). This allows the use of a simple four-way ribbon cable both to control a peripheral device and to provide it with power. To minimize crosstalk it is a good idea to route the ground connection between the two signal wires.

The signals on both wires operate with positive logic: a high voltage on SDA corresponds to a logic '1' and a clock on the SCL line consists of a brief positive-going pulse.

In the quiescent state a pair of pull-up resistors pull the two signals to a high level. The drivers in I²C devices have an open collector (or an open drain), which means that each of the two wires forms a bus-wide 'wired AND' connection (see **Figure 1**). The voltage level on one of the signal wires is only high when all the connected outputs are at a high level; if just one of the outputs is pulling low, then the whole wire will be at a low level. Each participant on the bus must therefore monitor the signal lines: if it wants to place a logic '1' on the bus it must make sure that no other participant is outputting a logic '0' at the same time. The value of the pull-up resistors is not especially critical, and should be in the region of a few kilo-ohms. The Raspberry Pi has these resistors already built in (1.8 k Ω pulling up to the 3.3 V supply). In the case of some microcontrollers such as the ATmega series there are pull-up resistors that can be enabled in software, but unfortunately they cannot be used for this purpose as their value (from 20 k Ω to 50 k Ω) is too high.

According to the specifications a high logic level should be at least $0.7 V_{DD}$ and a low logic level should be at most $0.3 V_{DD}$. In practice, however, 5 V devices work with pull-up resistors to a 3.3-V supply, as in the case of the Raspberry Pi. However, out-of-tolerance signal levels can sometimes be the cause of problems, and in such cases level shifters will be required.

I²C slave modules with built-in pull-up resistors are also available. Normally, however, these should be removed or disabled, as if more than one slave has such resistors fitted they will be in parallel on the bus and the resulting overall pull-up resistance will be too low. An I²C bus should have just one pair of pull-up resistors, most conveniently located at or near the master device.

At this point we run into a problem: some devices that operate from a 3.3 V supply cannot tolerate higher voltages on their input pins. If the pull-up resistors are tied to a 5 V supply, these devices might give up the ghost. On the other hand, for the bus to operate reliably, we would like to stick to the specified ranges of voltage levels. One solution that allows 5-V and 3.3-V devices to share a common bus is to use a bidirectional level shifter, as described in [3]. The circuit uses two MOSFETs and some additional pull-up resistors to link the 5-V and 3.3-V sections of an I²C bus: see **Figure 2**.

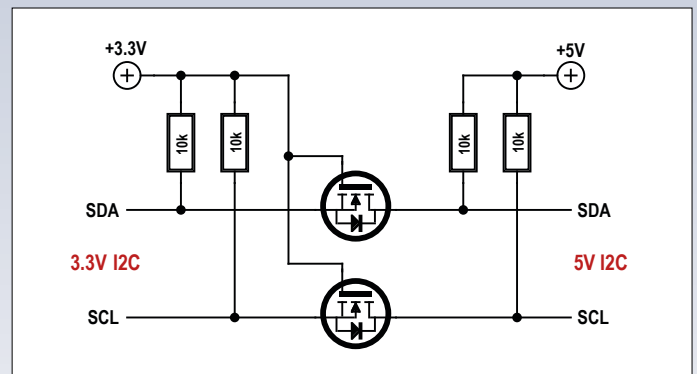


Figure 2. MOSFETs can be used to divide a bus into a 5 V section and a 3.3 V section.

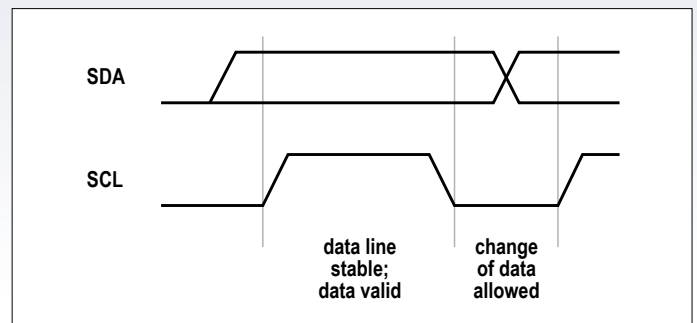


Figure 3. Timing of one data bit.

The protocol

The I²C bus is a pure master-slave bus: that means that communication is always initiated by the master, which addresses a slave and then exchanges data with it. A slave is not able to initiate communication by itself, as is possible for example in the case of a SCSI bus (using 'asynchronous event notification' and 'reselection'). As already noted above, a microcontroller can be programmed to take on the role of a master or a slave, and indeed can in theory change its role on the fly. However, in general, each bus participant has a fixed role as either a master or a slave.

The master is always responsible for setting the basic overall communication speed. Four speeds are officially supported:

- 100 kbps (standard mode or 'Sm', the default)
- 400 kbps (fast mode or 'Fm')
- 1 Mbps (fast mode plus or 'Fm+')
- 3.4 Mbps (high speed or 'Hs')

Data transfer is possible either from master to slave or from slave to master, but the clock signal, on the SCL line, is always generated by the master. Data bits are transferred synchronously to the clock, as shown in **Figure 3**: in order to send a bit, the transmitter must place the required logic level on the SDA line, using positive logic. The master then releases the SCL line, which is then pulled high, and then shortly afterwards pulls it back low again. One bit has now been transferred and the transmitter can now proceed to the next bit. The SDA signal may not change state while SCL is high during a transmission.

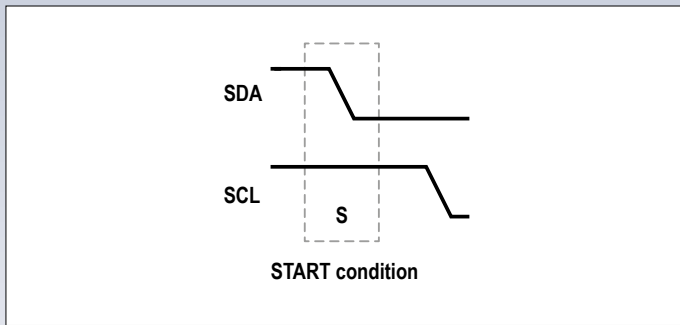


Figure 4. Start condition.

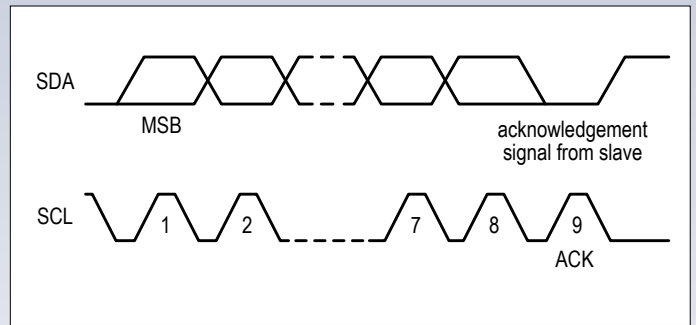


Figure 5. Sending the address bits.

1. To begin a transmission, the master pulls first the data line SDA and then the clock line SCL low (see **Figure 4**). This sequence is called a 'start condition'.
2. After the start condition the master first sends the slave address (see **Figure 5**), with most significant bit first. This is followed by the read/write bit: 0 here means that data will be transferred from master to slave and 1 means that data will be transferred from slave to master. The slave address is usually seven bits long (although the I²C standard also provides for ten-bit addressing, this feature is rarely used) and is hard-wired, either wholly or partly, by the manufacturer of the slave device. It is often possible to use external circuitry on a slave device to configure some of its address bits, which allows several instances of the same device to be connected to a single bus using different addresses for each. The slave address will be given in the device data-sheet, or can be determined empirically using a program such as [i2cdetect](#). The slave address plus the read/write bit makes a total of eight bits, or one byte.
3. After the address byte the master generates the SCL clock signal to transfer data, one byte at a time and with each byte transmitted most-significant bit first (see **Figure 6**). If the read/write bit in the address byte was 1, then the master expects the slave to place its data bits on the SDA line at the appropriate moment, before it generates the clock signal on the SCL line. If the read/write bit was a 0, however, the master itself places the data bits on the SDA line as well as generating the clock pulses. Each byte, including the byte comprising the address and the read/write bit, must be acknowledged by the receiver during the ninth bit time. Here a low level indicates a positive acknowledgement (ACK), while a high level is a negative acknowledgement (NACK). A NACK does not necessarily mean that something has gone wrong with the transmission: it is also used simply to indicate that the byte in question was the last one in the message. In the case of the address byte, a NACK means that no slave recognized the combination of address and read/write bit as valid.
4. At the end of a transmission the master releases first the SCL clock line and then the SDA data line: the pull-up resistors then pull the two signals to a high level. This sequence (see **Figure 7**) is called a 'stop condition'. To create the

stop condition it may be necessary for the master to first pull the SDA line low.

The start condition and the stop condition are exceptions to the rule that SDA is not allowed to change while SCL is high. There is no limit to the number of bytes that can be transferred in a single transaction. Higher-level protocols, such as the Power Management Bus (PMBus) and System Management Bus (SMBus), which are based on the I²C bus, specify upper limits to the message length or define record formats that, for example, include a length field within the message. In any case 'unlimited' messages are not supported by all hardware implementations or by all software libraries, as in many instances the number of bytes to be received must be specified before a transaction can be started: one example of this is the 'Wire' library for the Arduino. The protocols also define timeout values (for example for clock stretching: see below) and data integrity checks ('Packet Error Checking', or PEC).

Each transaction, between a start condition and a stop condition, is always unidirectional: the master specifies the direction of data transfer for the whole transaction with the read/write bit.

Only a small number of devices have a sufficiently simple internal structure that they can be controlled using a single message. (One example is the PCF8574, which we will look at in the last part of this series.) Mostly I²C devices require the master to use a write transaction to set a parameter in the slave, such as a register pointer or a memory address, before a read transaction can be carried out. If the master wishes to send data to the slave, this can be done after the parameter within a single write transaction; however, if the master wishes to read from the slave, it has to start a new read transaction to do so. To avoid the situation where a second master interrupts this sequence and sends a different parameter to the same slave, the first master can omit the stop condition at the end of the parameter-setting message: this is called a 'repeated start condition'. The sequence is as follows:

```
start condition - address+write - register number -
(repeated) start condition - address+read - data -
... - stop condition
```

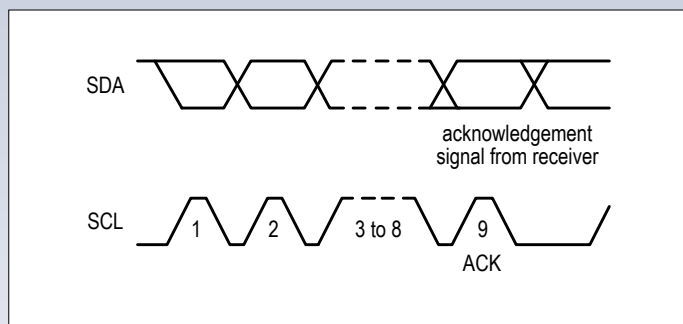



Figure 6. Sending the data bits.

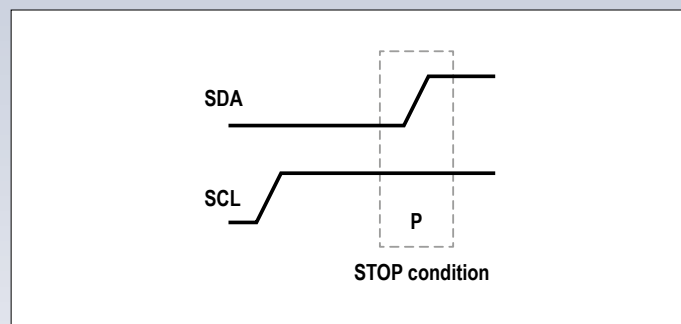


Figure 7. Stop condition.

Some peripheral devices such as memories or the RV-8523 real-time clock module automatically increment the register pointer or memory address after each byte is accessed; others, such as the LM75 temperature sensor, do not. In the former case it is possible to transfer the complete set of registers or the complete memory contents in one transaction. In the latter case, however, many separate transactions are needed; on the other hand, if the requirement is to read the same register again and again (which might be the case with the temperature register of the LM75) then there is no need to repeat the message to set the register pointer. Details on this point for a particular device will be found in its datasheet.

The I²C protocol does not provide for interrupts. Of course a slave can use a separate signal to trigger an interrupt on the master device, which can then in turn interrogate the slave over its I²C interface, but this does not form part of the standard I²C protocol itself.

Clock stretching and arbitration

When a slave is asked to send data to the master, it will sometimes need a little time before it is ready. During this period the slave must find some way to stop or delay the master's clock signal. Since the bus signals are in a wired-AND arrangement, one way for the slave to do this is for it to hold the SCL line low. The master can detect this situation, as it is able to see that the SCL signal does not go high when it releases it. This mechanism is called 'clock stretching', although perhaps more accurately it might be called 'pause stretching' as it is the interval between clock pulses that is lengthened, not the length of the pulse itself.

As mentioned above, it is possible to have more than one master on a single I²C bus. The bus is considered 'busy' in the period between a start condition and a stop condition, and no other master is permitted to generate a start condition during this period. A (would-be) bus master must therefore continuously monitor the bus to ensure that it only tries to send a start condition when the bus is free. If two masters should happen to send a start condition simultaneously, then it will in general be the case that either the slave addresses or at least the read/write bits will differ. That means that there will come a point where one master is trying to send a logic '1' while the other is trying to send

a logic '0'. Because of the wired-AND arrangement, the master trying to send a logic '0' will see its logic '0' on the bus, but the master trying to send a logic '1' will not see its logic '1', and when it detects this it must immediately stop transmitting. The 'winning' master (the one that was trying to send logic '0') does not notice that anything has gone awry and continues its transmission.

The process described in the preceding paragraph is sometimes called (including in the official I²C documentation) 'arbitration'. However, this is not really the correct term as there is no separate phase with priorities and timeouts: all that happens is that any master that wants to initiate a transaction waits for the bus to be free, starts transmitting, and watches what happens on the bus. There is nothing more to 'arbitration' as explained in NXP's documentation.

A master is allowed to terminate a transaction early at any time by sending a stop condition. A slave acting as a receiver can also terminate a transaction using a NACK, but a slave acting as a transmitter cannot. In fact, if a master is acting as a receiver and, as a result of a program bug or for some other reason either acknowledges a byte with a NACK or sends a stop condition, the transaction is not terminated. The slave may well stop sending data, but the master can carry on regardless, happily reading in whatever random data it happens to see on the SDA line.

Coming up

In the next installment in this series we will look at how various devices, from microcontrollers such as ATmegs and Arduinos to Raspberry Pis and PCs, can be used as I²C masters and slaves. We will include software examples in C and in Python to show how to address a device, send commands to it and read data from it. ◀

(160095)

Web Links

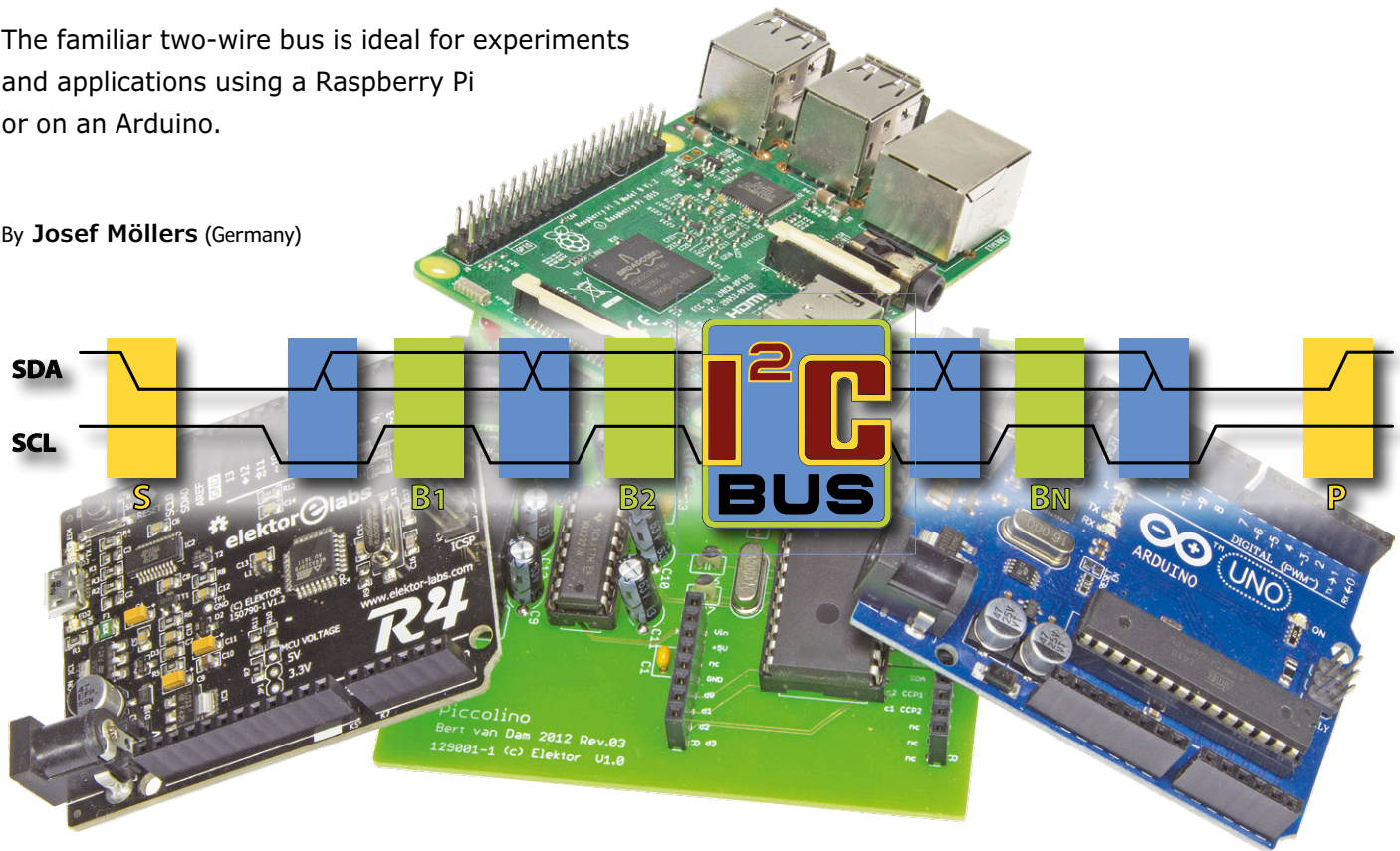
- [1] www.nxp.com/documents/user_manual/UM10204.pdf
- [2] www.i2c-bus.org
- [3] <http://playground.arduino.cc/Main/I2Cbi-directionalLevelShifter>

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.

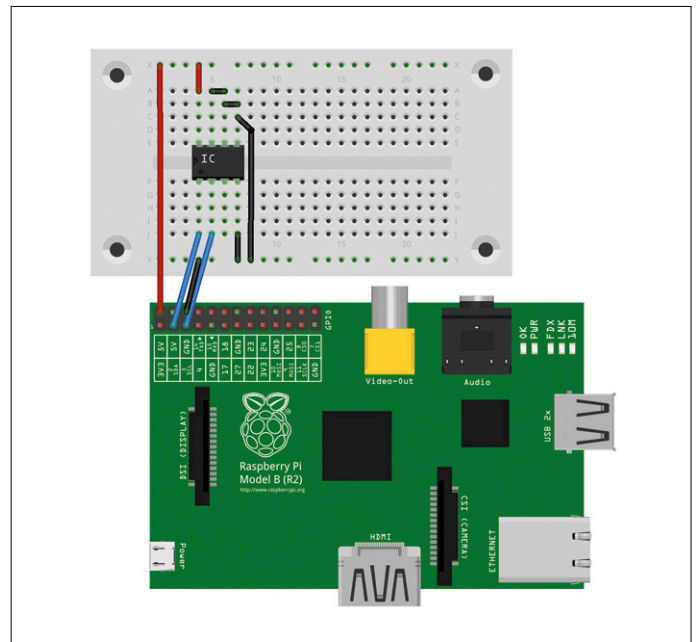


Figure 1. Connecting an LM75 to a Raspberry Pi using a breadboard.

```
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

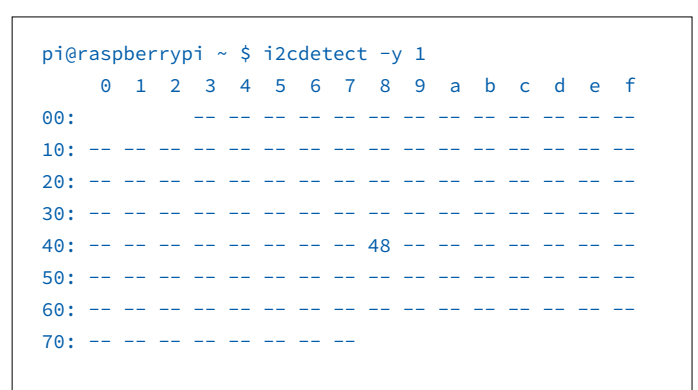


Figure 2. The `i2cdetect` tool has found an LM75 at address 0x48.

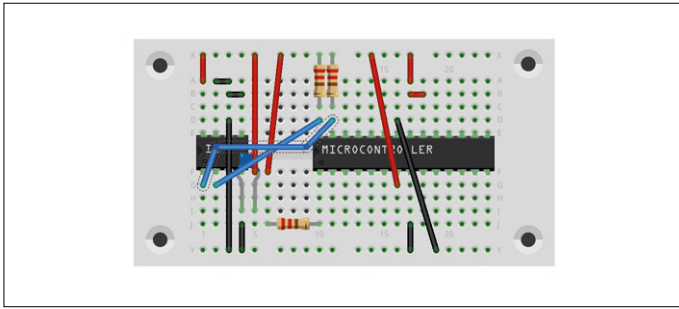


Figure 3. The ATmega88 and the sensor on a breadboard.

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
```

Listing 1. Main loop for reading from the LM75.

```
# include <i2cmaster.h>

# define LM75 (0x48 << 1)

int
main(void)
{
    unsigned char val[2];

    i2c_init();

    i2c_start(LM75 | I2C_WRITE);

    i2c_write(0x00);

    i2c_rep_start(LM75 | I2C_READ);

    val[0] = i2c_read();

    val[1] = i2c_read();

    i2c_stop();

    for(;;);
}
```

```
bus = smbus.SMBus(1)
address = 0x48
w = bus.read_word_data(address, 0)
print format(w, '04x')
```

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;
    }
}
```

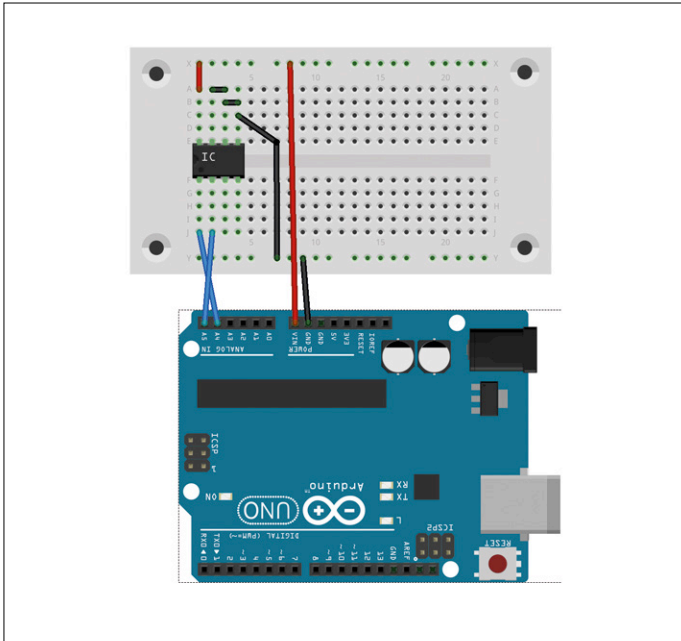


Figure 4. Connecting to an Arduino is practically the same as connecting to a Raspberry Pi.

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 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-- > 0) {
    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. ◀

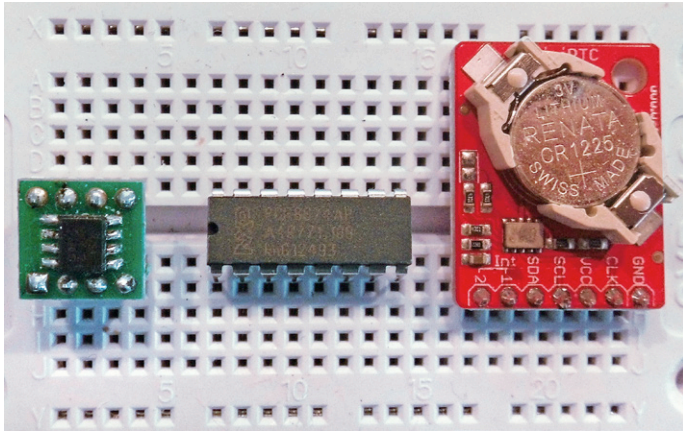
(160418)

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

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

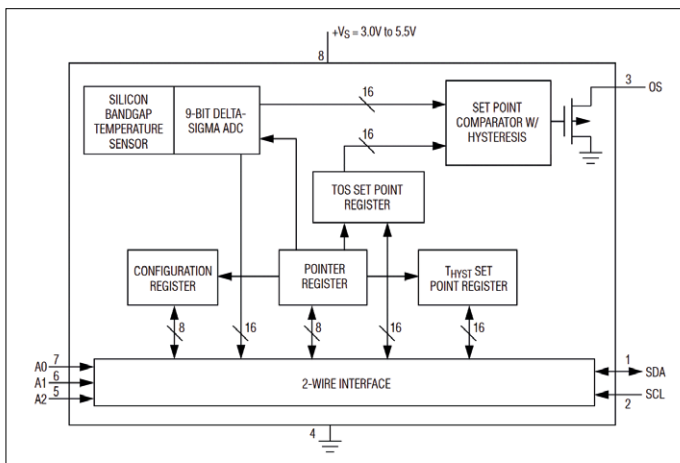


Figure 1. Innards of the type LM75 temperature sensor. (Source: Maxim)

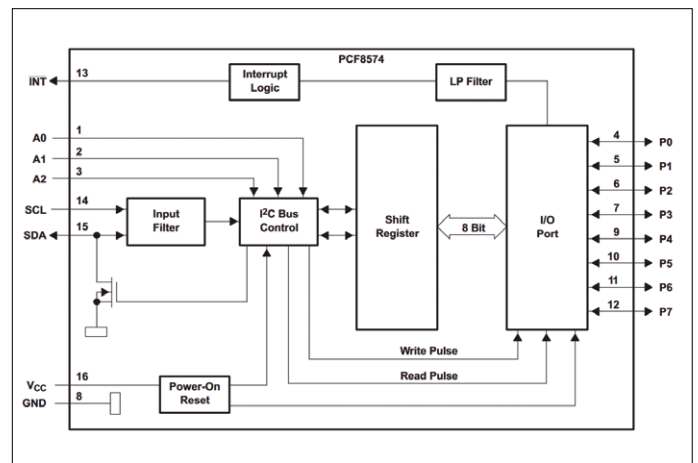


Figure 2. Internal circuit of the PCF8574 port expander. (Source: Texas Instruments)

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

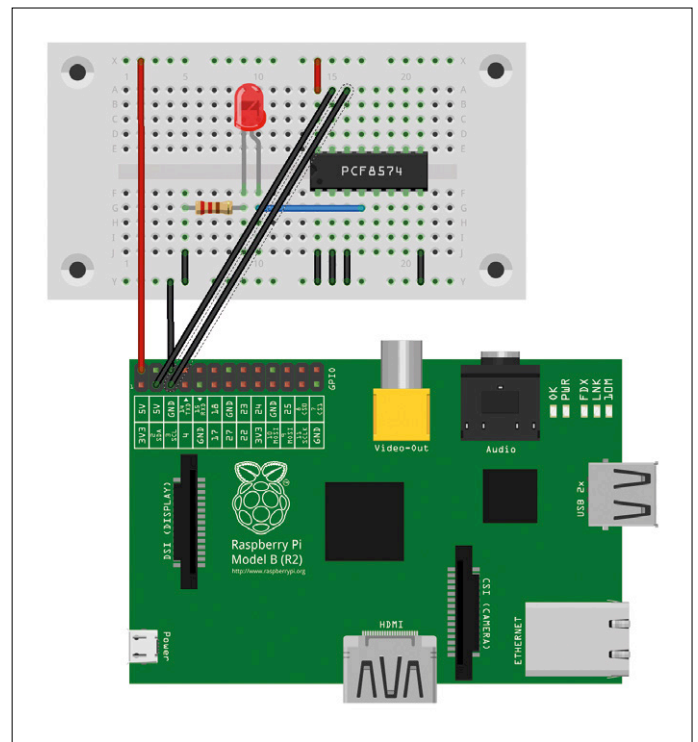


Figure 3. The PCF8574 port expander connected to a Raspberry Pi.

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

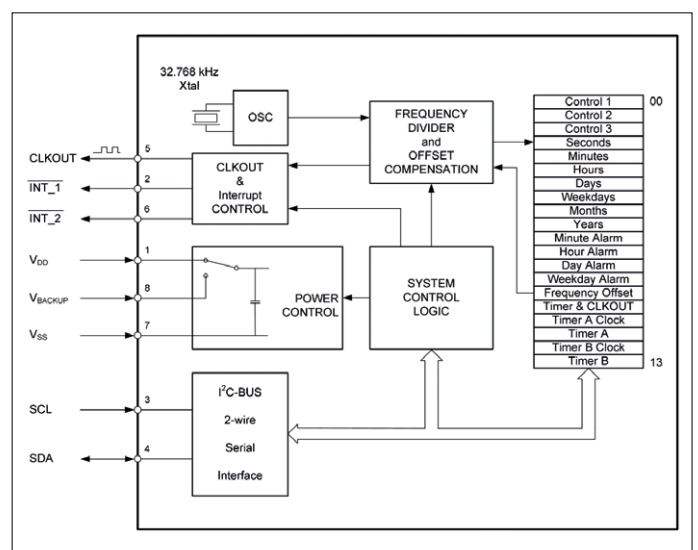


Figure 4. This real-time clock IC has twenty addressable control, time and alarm registers. (Source: Micro Crystal)

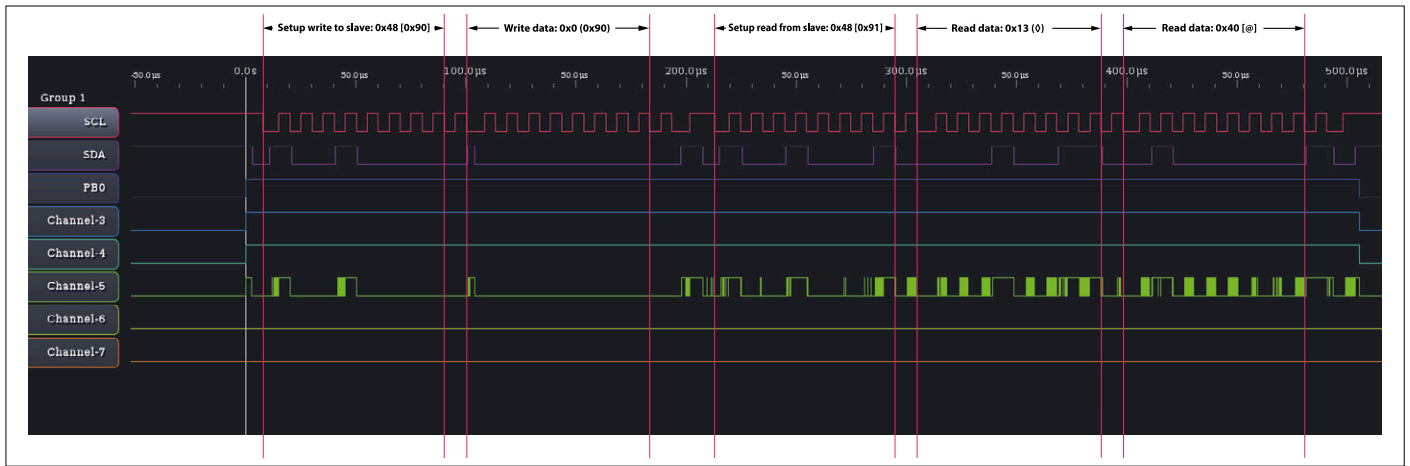


Figure 5. The Open Logic Sniffer looks on while an ATmega88 reads temperature data from an LM75.

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

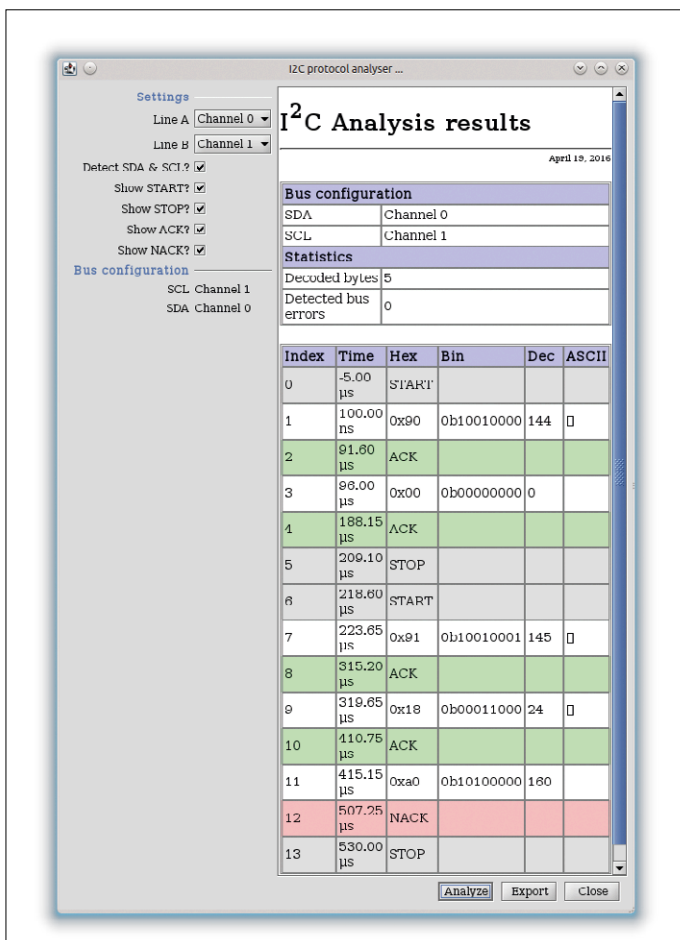


Figure 6. The OLS Java client presents the bus data in tabular form.

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].

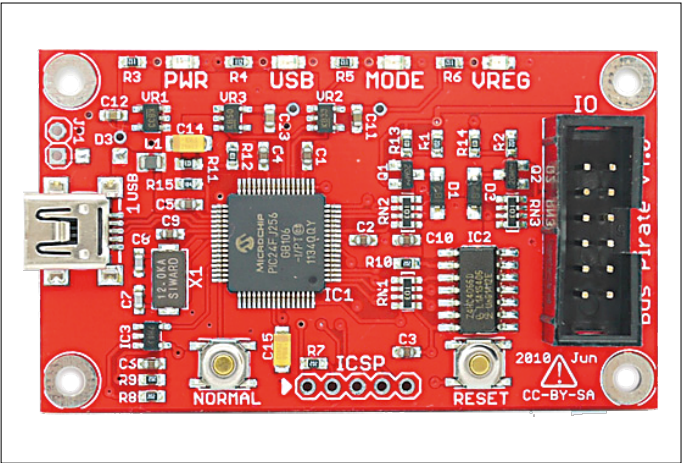


Figure 7. The tiny Bus Pirate is a universal bus interface for a PC. This is the new version 4. (Source: Dangerous Prototypes)

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)		
(1)>4		
Set speed:		
1. ~5KHz		
2. ~50KHz		
3. ~100KHz		
4. ~400KHz		
(1)>3		
I2C READY		
I2C>(2)		
Sniffer		
Any key to exit		
[0x90+0x00+] [0x91+0x13+0x40-]		
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 Bernhardt-Grisson, Norbert Bandzius and Thomas Schlüssler for proof-reading the original German article and for their feedback.

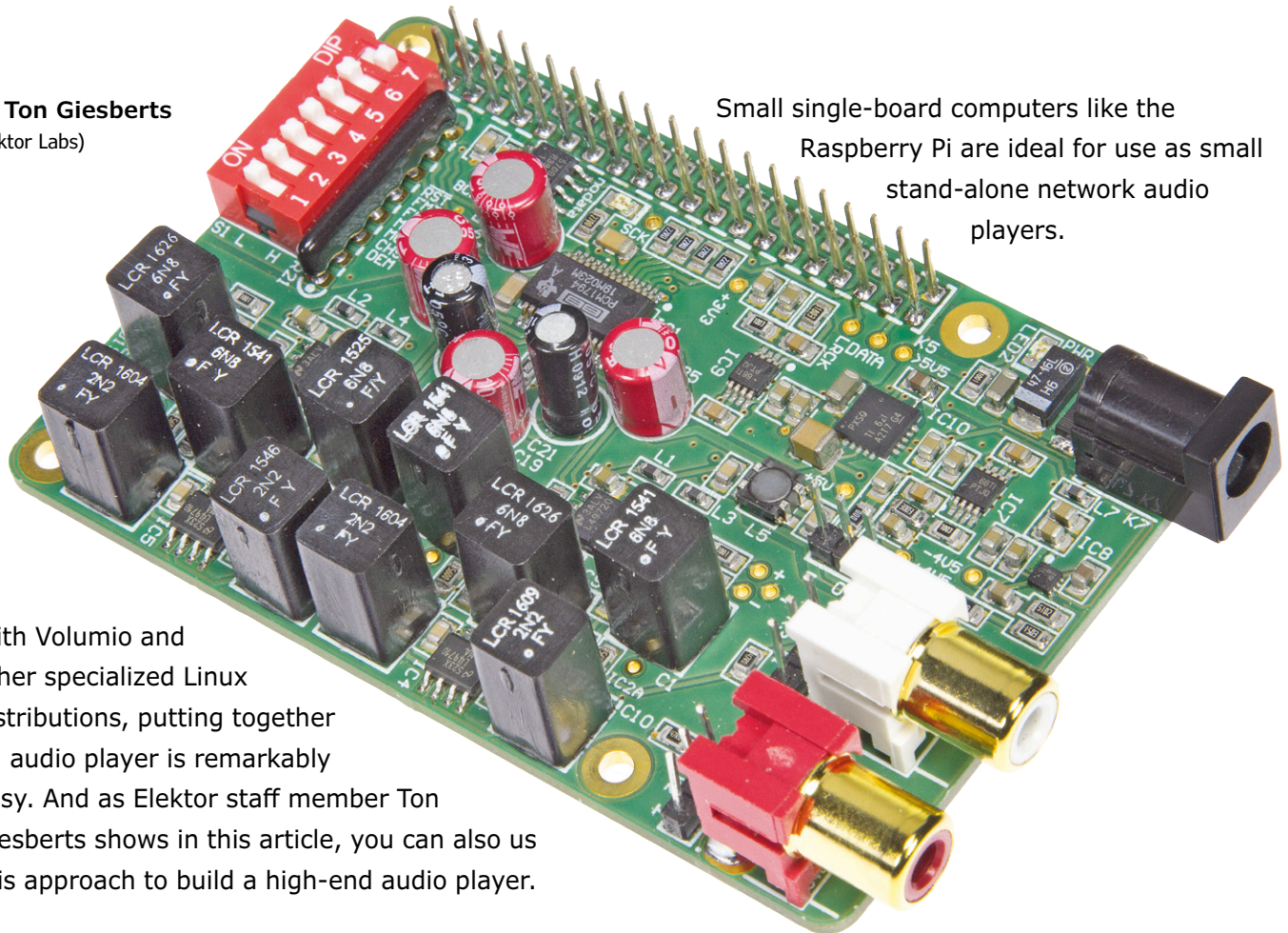
Audio DAC for Raspberry Pi

A network audio player with Volumio

By **Ton Giesberts**
(Elektor Labs)

Small single-board computers like the Raspberry Pi are ideal for use as small stand-alone network audio players.

With Volumio and other specialized Linux distributions, putting together an audio player is remarkably easy. And as Elektor staff member Ton Giesberts shows in this article, you can also use this approach to build a high-end audio player.



If you are looking for a small stand-alone network audio player, preferably with a touchscreen user interface, you don't have a lot of options. Usually you end up with a largish amplifier from one of the big brand names, with a corresponding price tag. And most of them do not have a touchscreen. If what you want is something compact and portable, these devices do not fit the profile. That means your only real option is to make your own. Apps based on the Raspberry Pi are a good starting point, and DACs with fairly good specs are now available for the RPi platform. However, at Elektor we are not satisfied with "fairly good" – if we are going to develop a

project for this, we might as well design a good high-end DAC for the RPi using components with top-notch specs.

The project at a glance

Digital to analog converter (DAC) ICs from Burr Brown (now part of Texas Instruments) have always been (and still are) about the best you can buy, and we have used them previously in various high-end DAC projects. For this project we opted for one of the top of the line devices from Burr Brown: the PCM1794A. This is a 24-bit DAC which can handle sampling rates up to 200 kHz and has an integrated 8× oversampling filter. This IC comes in a 28-pin SSOP

package, and it has outstanding specs in all respects, including a dynamic range of 127 dB ($2 V_{RMS}$, stereo) to 132 dB ($9 V_{RMS}$, mono), a THD of 0.0004%, and differential current outputs ($7.8 mA_{pp}$). The PCM1794A accepts all known data formats: standard, left justified, and I²S (which is important for our project). The digital portion of the IC operates at 3.3 V, but the digital inputs are also compatible with 5 V signals. For a complete overview of the specifications, see the device data sheet [1].

In order to convert the differential current outputs into single-ended voltages, we need a current to voltage converter for each channel, followed by a combined

filter and gain stage that converts the differential voltage output into a single-ended voltage output.

Based on this knowledge, we can draw the block diagram shown in **Figure 1**. From that we can derive the following shopping list: on the hardware side a Raspberry Pi 3 (version 2 is also usable) and the previously mentioned audio DAC, along with a Waveshare 3.5" touchscreen display for the Raspberry Pi to provide the user interface, and on the software side Raspbian for the RPi, Volumio or something similar (such as Mood Player), and the driver for the Waveshare display.

Power supply considerations

The digital to analog converter IC1 (see the detailed schematic diagram in **Figure 2**) needs two different supply voltages: +3.3 V for the digital portion and +5 V for the analog

converters and analog output filters, we use an on-board inverting voltage regulator to derive it from the +5 V rail. If you take a closer look at the circuit diagram, you can see that the analog supply voltage is actually +5.2 V instead of +5 V. We intentionally chose this somewhat higher voltage, which is still within the safe range for the DAC IC, to ensure a maximum undistorted signal level of 1 V from the output filter. You often see an output voltage of 2 V, but for that you have to use rail-to-rail opamps or a higher supply voltage, and most rail-to-rail opamps cannot match the outstanding specifications of special high-end audio opamps.

We chose a TPS7A4700 ultra low-noise, low-dropout voltage regulator to provide the +5.2 V main supply voltage. It can supply a maximum current of 1 A with a voltage drop of just 307 mV. The output

PROJECT INFORMATION

audio

network player DAC

Raspberry Pi touchscreen

entry level

→ intermediate level

expert level

5 hours approx.

Raspberry Pi 2 or 3, computer with Putty (Windows) or Linux, two AC adapters

€110 / €95 / \$120 approx.

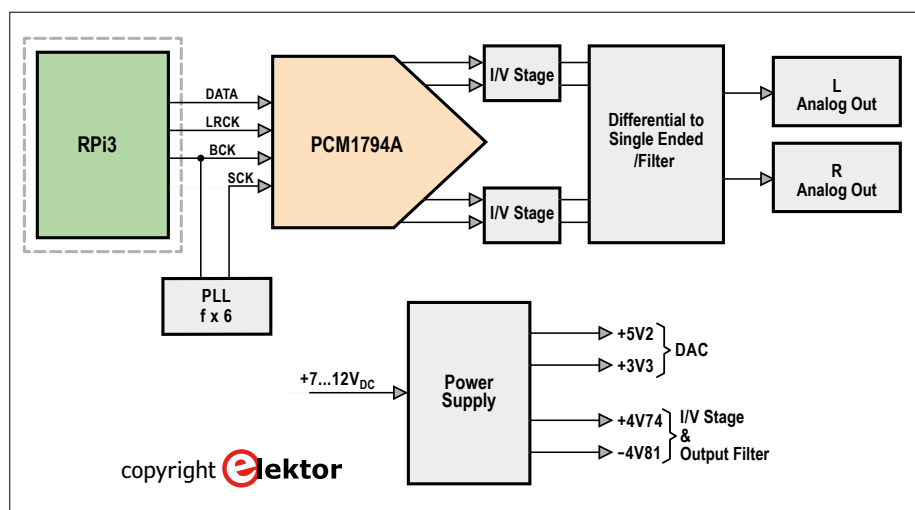


Figure 1. Block diagram of the audio DAC.

portion. In theory we could use the supply voltages available on the GPIO expansion connector of the Raspberry Pi board, but they are so noisy that there would be nothing left of our high-end aspirations. That means we have to provide an independent source of +3.3 V and +5 V. Of course, it would also be possible to power the Raspberry Pi from the DAC board, but that would have the same detrimental effect on the ultimate signal quality. A drawback of this totally separated approach is that we will need two AC adapters to power the overall device. To avoid the need for yet a third AC adapter to provide the negative supply voltage for the current to voltage

voltage can be set by grounding specific pins which are connected to internal resistors. Each pin has an associated voltage, and the output voltage is equal to the sum of the voltages of the individual pins. The minimum output voltage is the same as the reference voltage of 1.4 V. To obtain an output voltage of +5.2 V, we tie pins 6, 10 and 11 to ground, so the output voltage is given by the formula:

$$V_{out} = 1.4V + 3.2V + 0.4V + 0.2V = 5.2V$$

Other voltages

The positive supply voltage for the analog output portion is set to +4.74 V, which is

derived from the +5.2 V supply voltage by a TPS7A4901 ultra low-noise linear voltage regulator (IC7). The exact value of the output voltage is set by the voltage divider R28/R29:

$$V_{out} = \left(\frac{R28}{R29} + 1 \right) \times 1.185V$$

For the +3.3 V supply voltage we use the same type of linear voltage regulator (IC9), with the output voltage set by the voltage divider R32/R33.

The negative supply voltage is provided by an LM27761 low-noise inverting voltage regulator (IC8). It uses switched capacitors to invert the input voltage, followed by a low-noise linear regulator which generates a stable output voltage from the inverted input voltage. This output voltage is set to -4.81 V by the voltage divider R30/R31:

$$V_{out} = - \left(\frac{R30}{R31} + 1 \right) \times 1.22V$$

We intentionally chose these “odd” voltages for the output portion (instead of the more common ±4.5 V) to ensure an undistorted maximum output signal level of 1 V (0 dB).

The power connector (K7) is protected against reverse polarity by the Schottky diode D1. The voltage drop over this diode is just 0.3 V. The current

capability will also be implemented in the planned update to Volumio 2.

The D/A converter

The DAC device (IC1) has an integrated 8x digital oversampling filter and can handle sampling frequencies from 10 kHz to 200 kHz. The I²S signals are connected through four 22 Ω resistors (R22 to R25) to block RF noise, since

the master clock runs at nearly 74 MHz with a sampling frequency of 192 kHz. The various hardware select lines (MONO, CHSL, DEM, MUTE, FMT0, FMT1 and RESET) are routed to a 7-pole DIP switch and a set of seven 47-kΩ pull-up resistors. This makes changing the settings very easy. The ZERO output on pin 13 drives LED1, which indicates that no audio data is present.

For supply voltage decoupling we use aluminum-polymer capacitors (C19, C23, C25, C27) due to their low equivalent series resistance (ESR) of 40 mΩ at 100 kHz. However, they have the disadvantage of a relatively high leakage current of 100 μA (max.). Since it is not clear whether this leakage current affects the internal input current, we added a pair of normal electrolytic capacitors



COMPONENT LIST

Resistors

Default: 1%, 0.125W, 0805
 R1,R2,R3,R4,R7,R10,R13,R16 = 360Ω
 R5,R8,R11,R14,R34 = 1kΩ
 R6,R9,R12,R15 = 560Ω
 R17,R18 = 47kΩ
 R19 = 270Ω
 R20 = 10kΩ
 R21 = 47kΩ SIP 7-resistor array, 125mW, 2%
 R22,R23,R24,R25 = 22Ω
 R26,R27 = 0Ω
 R28 = 300kΩ
 R29,R33 = 100kΩ
 R30 = 150kΩ
 R31 = 51kΩ
 R32 = 180kΩ

Capacitors

C1,C2,C5,C6,C9,C14 = 6.8nF, 63V, 1%, polystyrene, EXFS/HR 6800PF ±1%, LCR Components (alternative 5mm pitch or 0805)
 C3,C4,C7,C8,C12,C13,C17,C18,C29,C34,C35 = 100nF, 50V, 10%, X7R, SMD 0805
 C10,C11,C15,C16 = 2.2nF, 63V, 1%, polystyrene, EXFS/HR 2200PF ±1%, LCR Components (alternative 5mm pitch or 0805)
 C19,C23,C25,C27 = 10μF, 35V, 0.04Ω, diam. 6.3mm max., pitch 2/2.5mm, 870055673001 (WCAP-PTHR series), Würth Elektronik
 C20,C21 = 10μF, 63V, 1.06Ω, diam. 6.3mm max., pitch 2/2.5mm, UPM1J100MDD, Nichicon
 C22,C24,C26 = 47nF, 50V, 10%, X7R, SMD 1206
 C28 = 10nF, 50V, 10%, X7R, SMD 0603
 C30,C31,C32,C33,C36,C37,C40,C41,C43,C44,C46,C47,C48,C49,C52,C53 = 4.7μF, 25V, 10%, X5R, SMD 0805
 C38,C39,C50,C51 = 10nF, 50V, 10%, X7R, SMD 0805
 C42 = 2.2μF, 10V, 10%, X7R, SMD 0805
 C45,C55 = 1μF, 16V, 10%, X7R, SMD 0805
 C54 = 47μF, 16V, 20%, X5R, SMD 1210
 C56,C57 = 10μF, 25V, 10%, X5R, SMD 1206
 C58 = 47μF, 16V, 10%, tantalum, SMD-C (2312), TR3C476K016C0350, Vishay

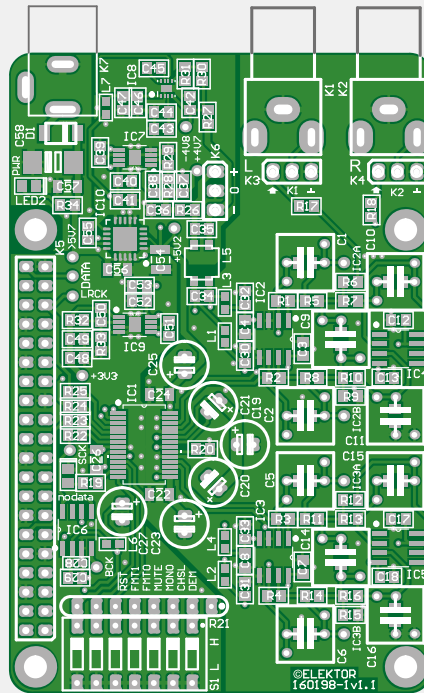


Figure 3. The component side of the PCB.

Inductors and Transformers

L1,L2,L3,L4,L6,L7 = 600Ω at 100MHz, 0.15Ω, 1.3A, SMD 0603, BLM18KG601SN1D, Murata
 L5 = 2x0.05Ω, 230Ω at 100MHz, 2.6A, SMD, ACM4520-231-2P-T, TDK

Semiconductors

D1 = PMEG6030EP, 60V, 3A, SMD SOD-128
 LED1,LED2 = green, low power, SMD 0805
 IC1 = PCM1794ADB, SMD SSOP-28
 IC2,IC3 = LME49720MA/NOPB, SMD SOIC-8
 IC4,IC5 = LME49710MA, SMD SOIC-8
 IC6 = ICS570BILF, SMD, SOIC-8
 IC7,IC9 = TPS7A4901DGNT, SMD MSOP-8
 IC8 = LM27761DSGT, SMD WSON-8
 IC10 = TPS7A4700RGWT, SMD VQFN-20

Miscellaneous

K1 = RCA audio connector, white, PCB mount, right angle, gold plated, PJRAN1X1U02AUX, Switchcraft

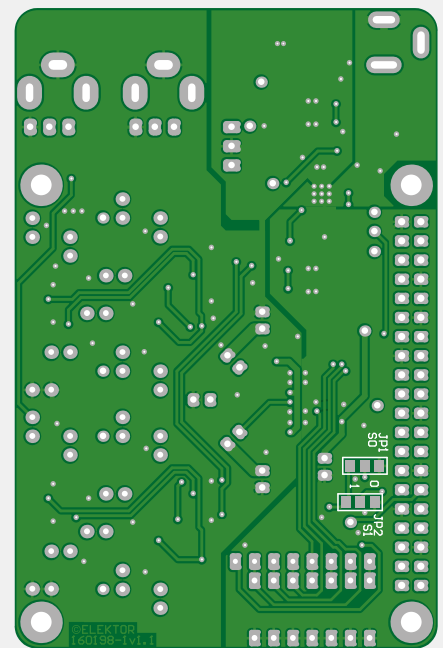


Figure 4. The bottom side of the PCB. Jumpers JP1 and JP2 can normally be omitted.

K2 = RCA audio connector, red, PCB mount, right angle, gold plated, PJRAN1X1U03AUX, Switchcraft
 K3,K4,K6 = 3-pin pinheader, vertical, 0.1" pitch
 K3,K4 = jumper, 0.1" pitch
 K5 = 40-pin GPIO stacking header, 2x20-way female, extra tall
 K7 = DC power connector (jack), 3A, 1.95mm, Lumberg NEB 21 R
 S1 = 7-position DIP switch
 4x M2.5 17mm male/female threaded standoff
 4x M2.5 14mm male/female threaded standoff
 8x M2.5 nut
 PCB 160198-1 v1.1

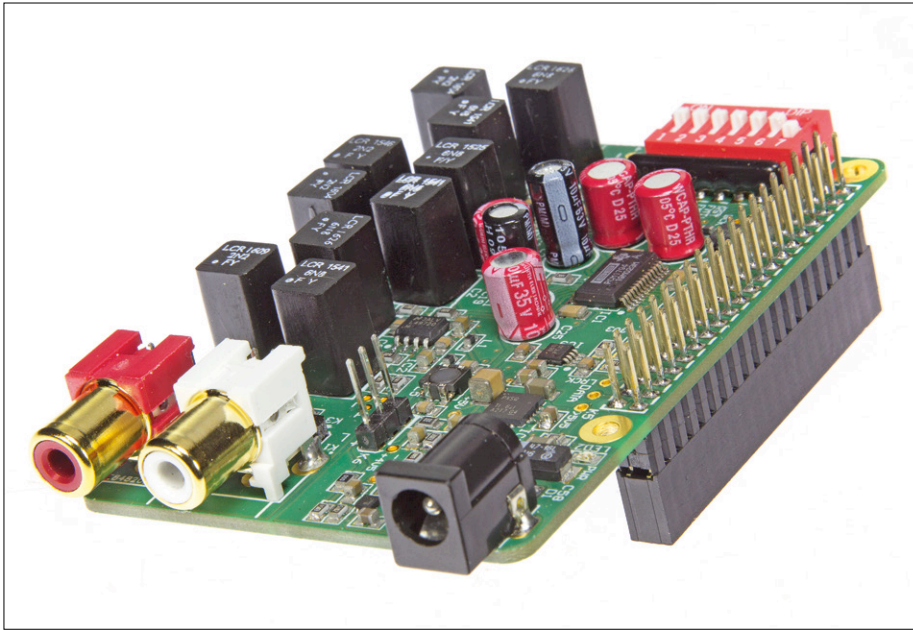


Figure 5. The GPIO connector is mounted on the bottom side.

Table 1: S1 settings		
S1-1	L	De-emphasis disabled for 44.1 kHz
S1-2	L	Digital filter with steep falling edge
S1-3	L	Mono off
S1-4	L	Mono off
S1-5, S1-6	L	I ² S mode
S1-7	H	Reset off

Table 2: Current consumption at different sampling frequencies	
Sampling frequency	Current consumption (K7)
No data	120 mA
32 kHz	127.5 mA
44.1 kHz	131.7 mA
48 kHz	133 mA
96 kHz	149.5 mA
192 kHz	182.1 mA
(Measured with 8 V lab power supply; both outputs terminated with 10 kΩ)	

Table 3: Distortion				
THD+N				
Sampling frequency		48 kHz	96 kHz	192 kHz
1 kHz	BW = 22 kHz	0.0008%	0.0009%	0.0013%
	BW = 80 kHz	0.0028%	0.0012%	0.0014%
7 kHz	BW = 22 kHz	0.00095%	0.0011%	0.0013%
	BW = 80 kHz	0.003%	0.0014%	0.0016%
IMD				
50 Hz:7 kHz = 4:1		0.0014%	0.002%	0.0036%

(ESR 1.06 Ω at 100 kHz, leakage current 4 μ A). If you want to experiment with different capacitor types, there is plenty of room on the PCB.

Output filter

Although the sampling artifacts are largely suppressed by the 8x digital oversampling filter, there are still undesirable high frequency components present in the output signal of the DAC. We use a reconstruction filter to suppress these components. The filter in question is a third-order Butterworth filter, and the current to voltage converters (IC2/IC3) are part of this filter. The corner frequency is 64.5 kHz. With a sampling frequency of 44.1 kHz, the 8x frequency component at 352.8 kHz is attenuated by more than 40 dB, and with a sampling frequency of 192 kHz the attenuation is more than 80 dB. The chosen corner frequency is a compromise between adequate attenuation at low sampling frequencies and the audio bandwidth of the output signal at higher sampling frequencies. Here the filter bandwidth is more than three times the normal audible range.

The DAC IC has differential current outputs, which allows an external filter to be used. To convert the output current into a single-ended voltage, we need a current to voltage converter, which in this case is built around the dual opamps of IC2 and IC3. The differential output voltages of IC2A/IC2B or IC3A/IC3B are fed to IC4 or IC5, respectively, which convert them into single-ended voltages. They are also configured as second-order filter stages. In combination with the current to voltage converters, the overall filter characteristic is equivalent to a third-order Butterworth filter.

On the PCB the single-ended output signals are available on connectors K1 and K2 via headers K3 and K4. Jumpers allow the signals from K3 and K4 to be fed directly to K1 and K2. It is also possible to connect a stereo potentiometer to the headers to provide a simple volume control.

The DAC IC does not have integrated volume control capability. Although it is possible to use Volumio for software volume control, that comes at the expense of the resolution and therefore affects the sound quality. For us that is out of the question. We therefore decided to add hardware volume control in the form of a separate board, which also

allows the volume to be adjusted under remote control. The volume control board will be described in the next issue of *Elektor*.

As a final detail, the balanced analog supply voltages for the filter and current to voltage stages are decoupled from the digital supply voltage by a common-mode choke and four RF chokes (for each channel individually).

The board

For the DAC board we only need a single external supply voltage in the range of 7 to 8 V, since the Raspberry Pi board has its own 5 V supply. An external supply voltage of 9 to 12 V is possible, but not recommended due to the higher power dissipation in IC10.

The top and bottom PCB layouts for the DAC board are shown in **Figure 3** and **Figure 4**. The board has the same dimensions as the Raspberry Pi. The power connector and the two signal outputs are on the same end of the board, with the seven-pole DIP switch S1 on the opposite end (adjacent to the WiFi antenna on the RPi 3 board). The 40-pin GPIO stacking header K5 is mounted on the bottom of the board, as can be seen from the photos of the assembled board. The pins of connectors K1–K4, K6 and K7 protruding on the bottom of the board must be trimmed as short as possible to avoid shorting on the metal shells of the Ethernet and USB connectors on the Raspberry Pi board. It is probably a good idea to place a bit of insulation between the two boards. In our prototype we used M2.5 metal male/female threaded standoffs with a length of 17 mm (0.675") to attach the DAC board to the Raspberry Pi. The solder pads for the capacitors in the filter stages are laid out to allow three different types of capacitor to be mounted for C1, C2, C5, C6, C9–C11, and C14–C16. SMD 0805 components can be used here, but it is also possible to mount conventional through-hole capacitors with a lead pitch of 5 mm. For best results, we recommend using 1% polystyrene capacitors.

Pin 1 of each IC is marked with a small white dot, but with IC7 this is difficult to see due to capacitor C38. There was not enough room to put the numbers of the 0805 resistors and the 0805 and 1206 capacitors next to the components concerned, so we put them inside the component outlines. That means they

are hidden on the board. If you assemble the board manually using a hot-air soldering iron or a reflow oven, it's a good idea to use an enlarged copy of the component layout to help you put all the components in the right place. With many of the 0805 components, replacement is difficult or virtually impossible after the conventional through-hole components

have been soldered in place. The latter should therefore be kept for last.

Jumpers JP1 and JP2 are located on the bottom of the PCB. They can normally be left open ($SCK = 6 \times BCK$). The audio output connectors (Cinch/RCA connectors) are Switchcraft components with two terminals closer together than in the standard 10 mm version. This means

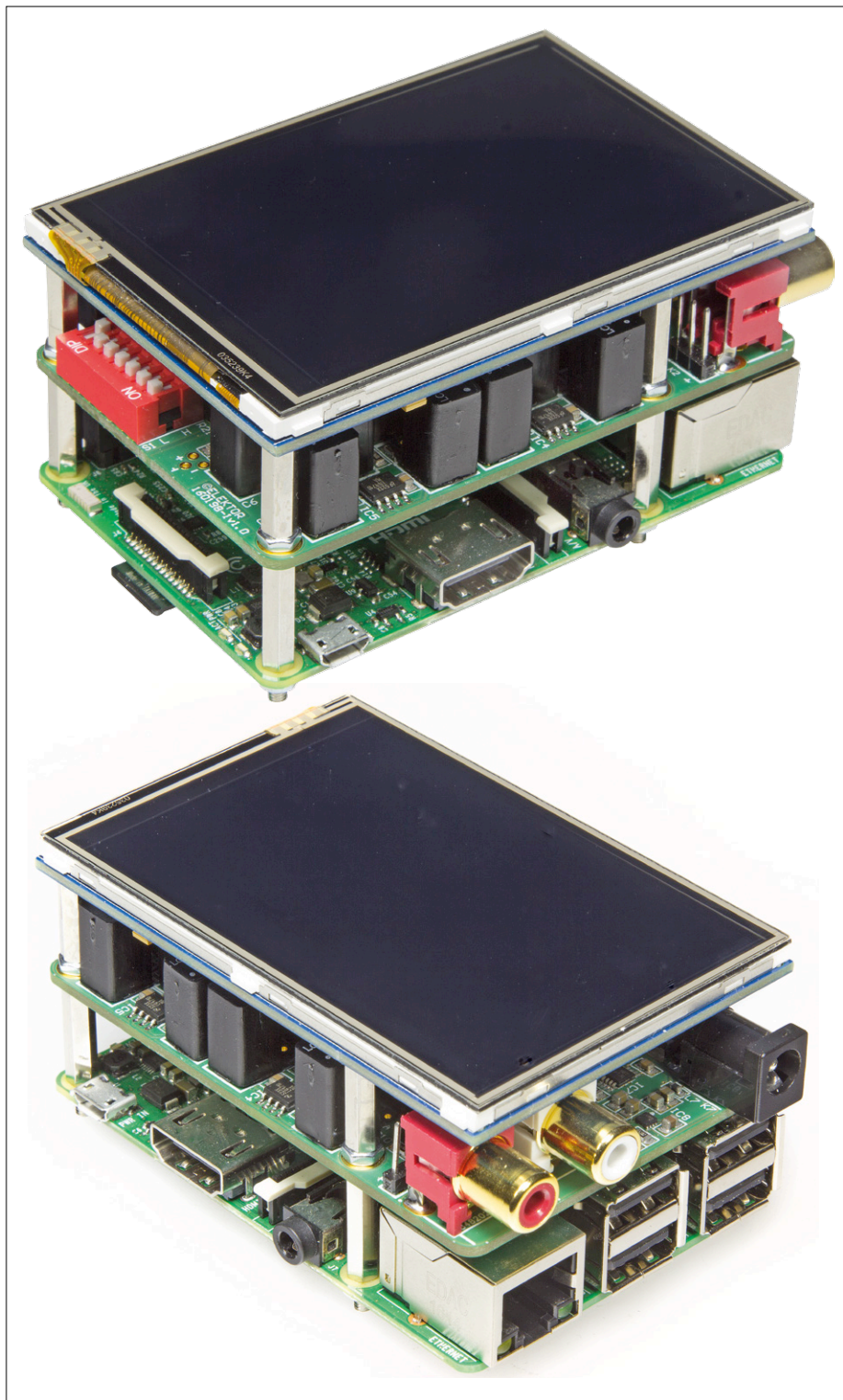


Figure 6: The complete network audio player.

Volumio

Volumio is a free open source Linux distribution specifically tailored to music playback. Volumio runs on a wide variety of devices, including small inexpensive computers such as the Raspberry Pi. After you install Volumio, the device turns onto a headless audiophile music player. Here “headless” means that you have to use another device, such as a smartphone, computer or tablet, to operate the audio player.

For that you can use the Volumio user interface, which is a web app that runs on any device with a browser and makes playing music files easy and intuitive. The web app and Volumio communicate with each other over the local area network. On the Elektor Labs page for this project [2] we provide detailed instructions for installing Volumio on your Raspberry Pi (please note that we used version 1.55) so that you can **also** use the touch screen for this purpose. ×

Figure 7 shows the Volumio user interface on the display (left) and on a PC monitor (right).

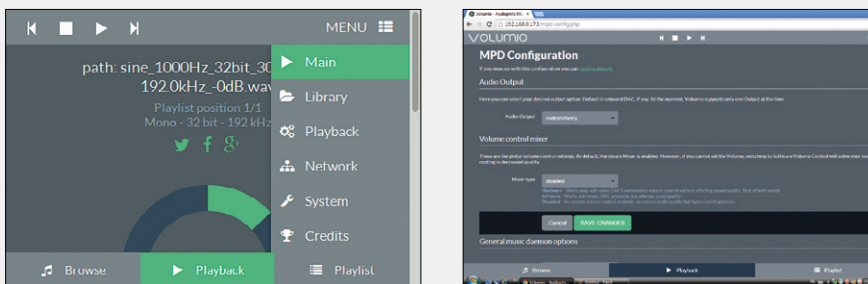


Figure 7. Volumio can be operated using the touchscreen (left) or a PC (right).



Web links

- [1] PCM1794A data sheet: www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=pcm1794a&fileType=pdf
- [2] Elektor Labs project page: www.elektormagazine.com/labs/audio-dac-for-rpi-networked-audio-player-using-volumio



FROM THE STORE

→ 160198-1:
Bare PCB

→ 160198-91:
Assembled module with touchscreen; headers included separately (to be soldered by user))

→ SKU 17631:
Raspberry Pi 3 (model B)

that if you want to use a different type, it probably will not fit.

The Waveshare 3.5 inch touchscreen display for the Raspberry Pi must be mounted 16 mm (0.625”) above the DAC board. For this we used four metal male/female threaded standoffs with a length of 14 mm (0.55”) together with an M2.5 nut under each standoff for the extra distance. This means you should not press the connector down as far as possible. In case of doubt, it is better to use a second stacking header and mount the display a bit higher. The distance between the 26-pin connector of the display and the DAC board is about 2 mm.

Figures 5 and 6 show what the complete network audio player looks like.

Measurements

It goes without saying that we subjected our audio DAC to extensive testing. The key results are summarized in **Tables 2 and 3**. All measurements were made with the S1 settings listed in **Table 1**. The current consumption increases at higher sampling frequencies, as can be seen from **Table 2**.

The measured figures for total harmonic distortion plus noise (THD+N) and intermodulation distortion (IMD) are listed in **Table 3**.

Several plots made with our Audio Precision analyzer are also available on the Elektor Labs page for this project [2]. ◀

(160198-1)

Volume Control for RPi Audio DAC

loud and quiet
in high quality

By **Ton Giesberts** (Elektor Labs)

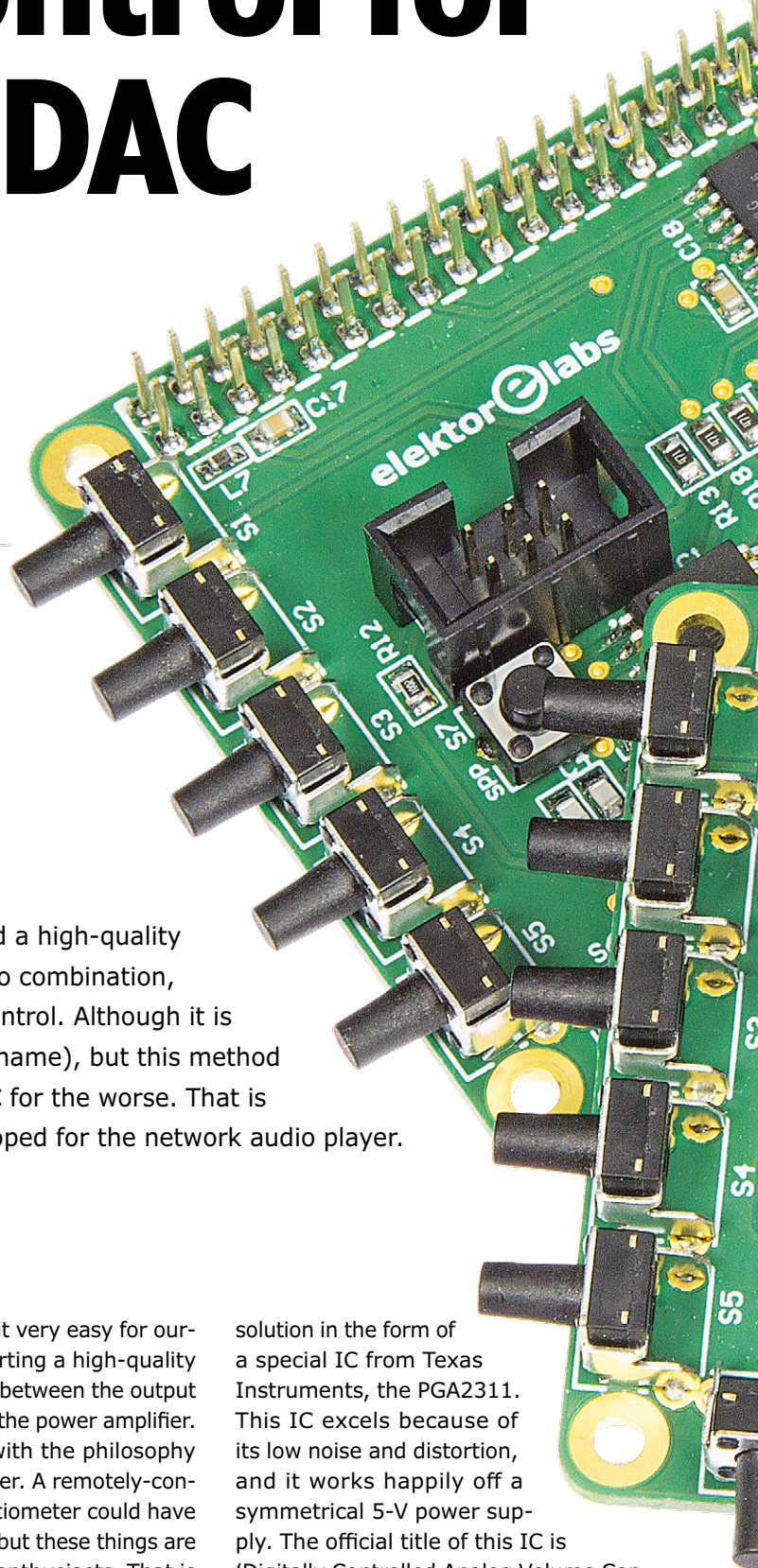
To the delight of many, the previous issue of Elektor had a high-quality network audio player based on the Raspberry Pi/Volumio combination, which however was not provided with a 'real' volume control. Although it is possible to control the volume via Volumio (what's in a name), but this method seriously affects the high-end characteristics of the DAC for the worse. That is why we now describe a volume control especially developed for the network audio player.

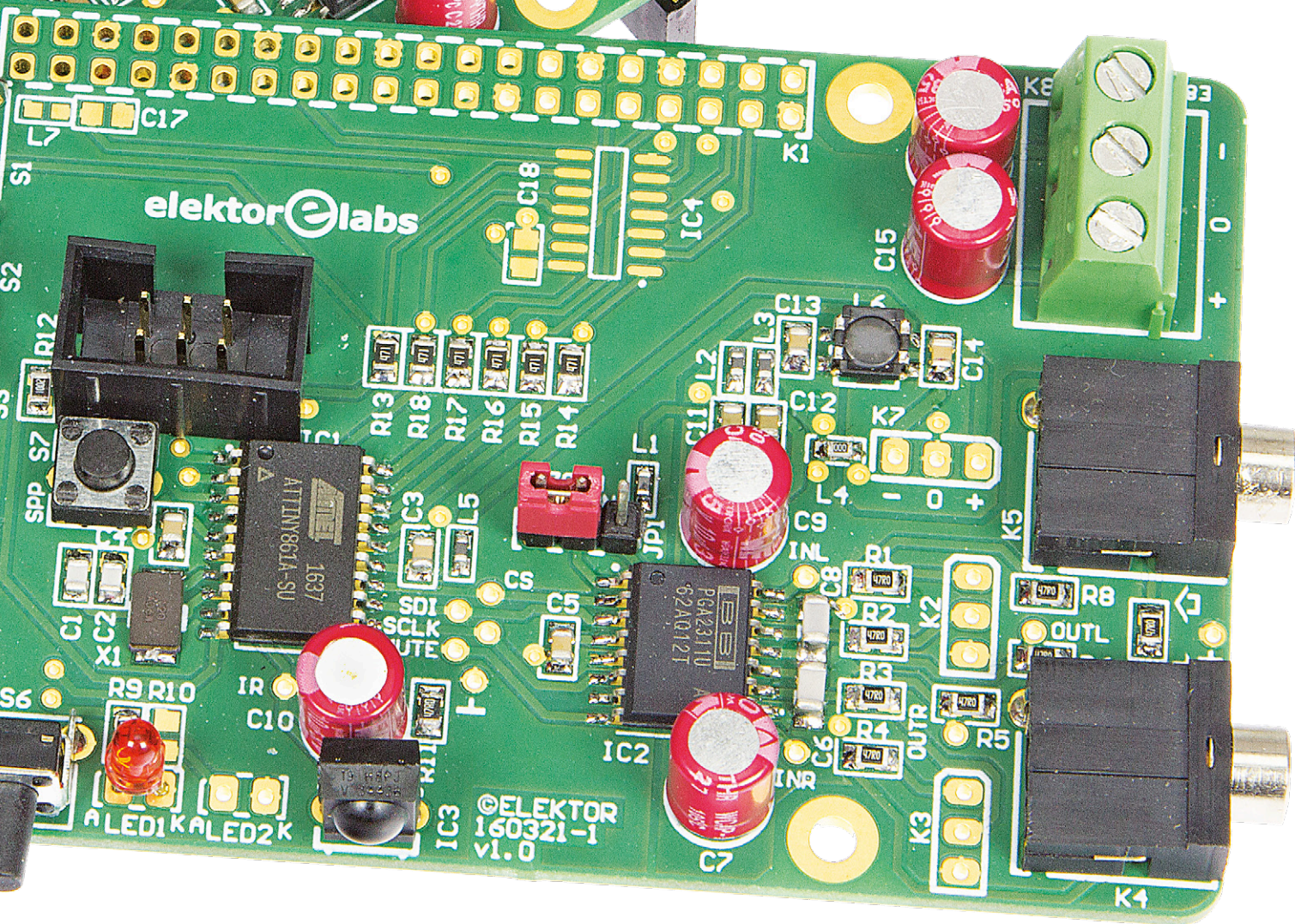
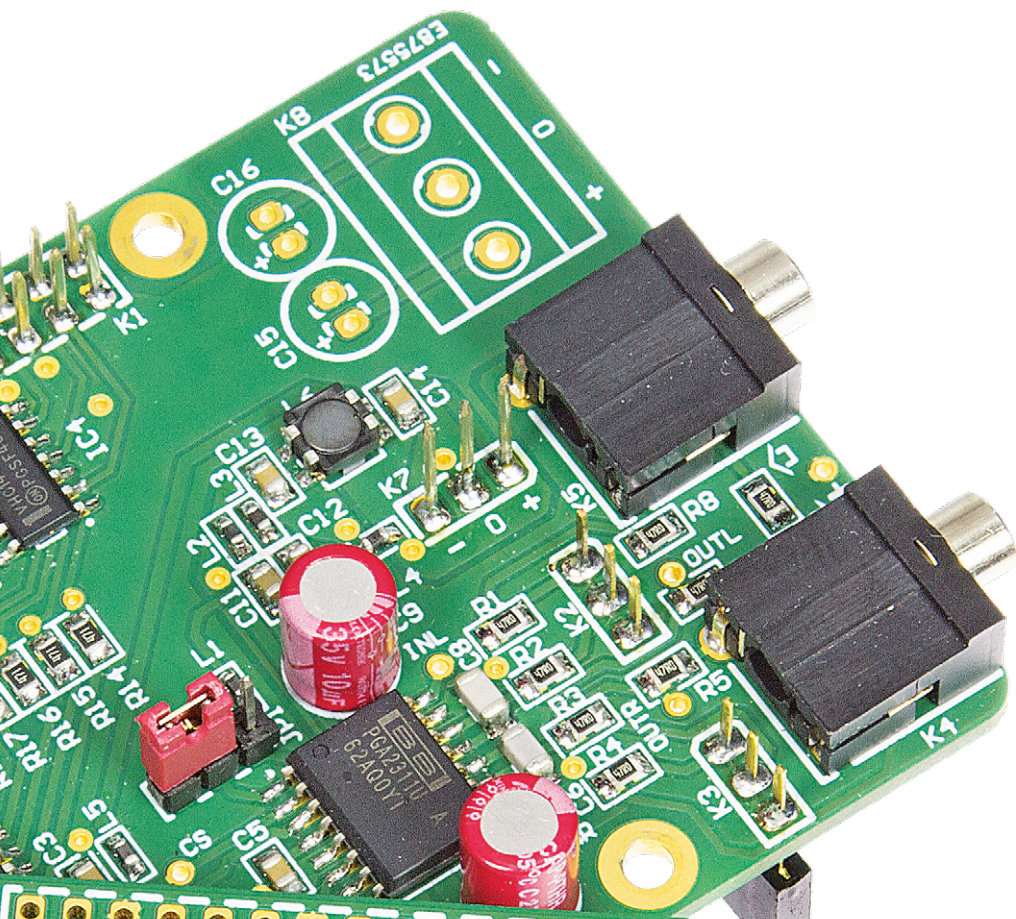
Characteristics

- High-end volume control
- Suitable for Raspberry Pi + audio DAC
- Compact sandwich construction
- Uses an RC5 remote control
- Also suitable as stand-alone volume control

We could have made it very easy for ourselves by simply inserting a high-quality stereo potentiometer between the output of the audio DAC and the power amplifier. But that doesn't fit with the philosophy behind the audio player. A remotely-controllable motor potentiometer could have been a solution here, but these things are scarcely available to enthusiasts. That is why we chose a high-quality, solid-state

solution in the form of a special IC from Texas Instruments, the PGA2311. This IC excels because of its low noise and distortion, and it works happily off a symmetrical 5-V power supply. The official title of this IC is 'Digitally Controlled Analog Volume Control'; the control of this IC takes place





PROJECT INFORMATION



volume control high-end

RC5 remote control

audio-DAC stand-alone



entry level

→ intermediate level

expert level



5 hours approx.



Soldering equipment,
programming adapter



€100 / £95 / \$125 approx.

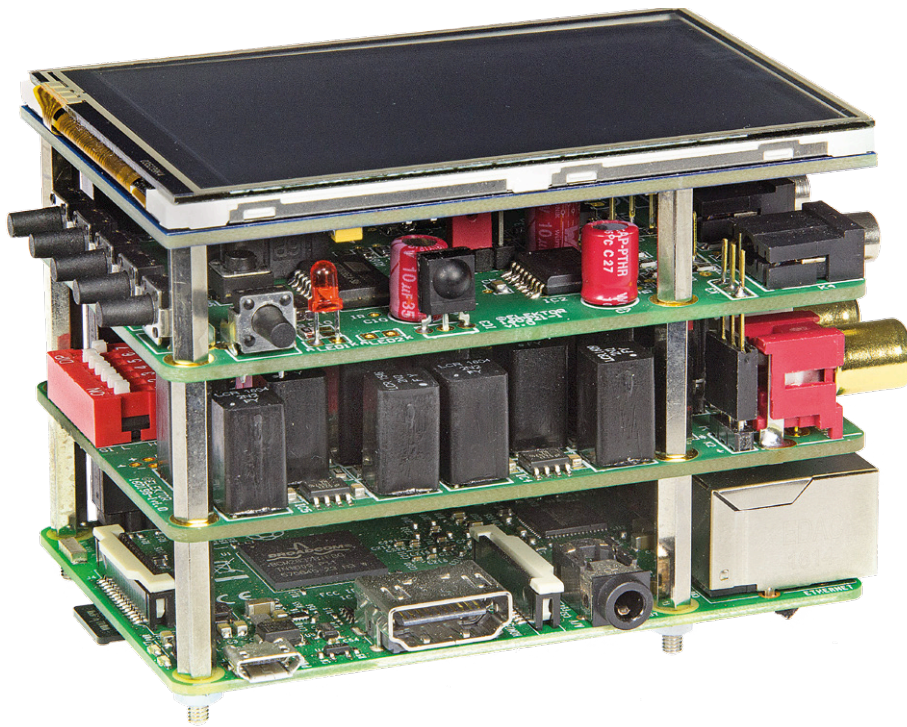


Figure 1. The 'Big DAC' in full glory.

across a serial interface connection. Additional features of this IC include a zero-crossing detector and a mute function. For the control we use a microcontroller from Atmel's ATtiny series.

General design

Because this volume control was designed, in the first instance, to be used with the audio-DAC from the previous edition, its circuit board has been designed such that it can simply be plugged into the DAC circuit board. This way a thick sandwich of Raspberry Pi, audio-DAC, volume control and display is created (disrespectfully called a 'Big DAC', after a well-known product

from the fast-food chain with the Scottish name), as can be clearly seen in **Figure 1**.

With the aid of half a dozen pushbuttons (a seventh button is used to store the default volume setting), Volumio and/or another application that is running on the RPi can be controlled. In addition, there is (obviously) provision for using an IR-remote control that uses the standard RC5 protocol.

And now then the good news for all those readers who are searching for an 'independent' usable volume control (that is, without RPi and audio-DAC): that is perfectly possible with the design described here! You can even omit a few parts...

The schematic

We've drawn the complete schematic for the volume control in **Figure 2**. You may have expected us to describe the circuit as a block diagram first, but that is not necessary — to the untrained eye the schematic looks complicated, but appearances can be deceiving.

We attempt to deal with the description systematically, and start with the beginning: the power supply (after all, without a power supply even the most artistically assembled circuit board is useless).

The power supply

If the volume control is used as an

'add-on' to the audio-DAC then a separate power supply is not necessary. For the volume controller, the required (analog) +4.74 V and -4.81 V (abbreviated in the schematic to +4V7_A and -4V8_A) are supplied by the DAC board; the same is true for the +4.7 V for the microcontroller (+4V7_D in the schematic). These power supply voltages enter via connector K7, which is soldered to the bottom side of the circuit board (refer later on) and connects with K6 on the DAC board. The +3.3 V, which is only required for the inverters (IC4) that drive the GPIO pins of the expansion connector, is generated by the Raspberry Pi (pin 1 of the expansion connector).

The power supply voltage for the PGA2311 is filtered by the common-mode filter coil L6 and inductors L2, L3 and L4. Filter inductor L5 filters the digital power supply voltage for the microcontroller (IC1) while L7 assumes this task for the 3.3-V power supply. Inductor L1 ensures sufficient separation between the digital and analog power supply voltages for the PGA2311. Inductor L4, finally, has a very important job: it prevents the creation of an RF ground-loop, because both the power supply ground (K7) and the ground connections of K2 and K3 are already connected together on the DAC board.

When used as an add-on to the DAC board, connector K8 and capacitors C15 and C16 do not need to be fitted.

On the other hand, when using the volume control for other applications, the components IC4, C17, C18, L7, K2, K3 and K7 do not need to be mounted. Additionally, in this case it is also best to replace L4 with a 0-Ω resistor. Note: for stand-alone use, a separate, symmetric 5-V power supply is required!

The PGA2311

For those who are interested, in **Figure 3** we have drawn the internals of the PGA2311. It is immediately clear that we have two (because it is stereo) amplifiers, the amplification of which are set by a set of (serial) control inputs. The control signals ($\overline{CS}$, SCLK, SDI and $\overline{MUTE}$) come from microcontroller IC1.

A brief remark about the connection of ZCEN (Zero Crossing Enable). The PGA2311 has a zero-crossing detector; the idea behind this is (when this function is enabled) that the change in volume only takes effect after the next positive-going zero-crossing of the audio

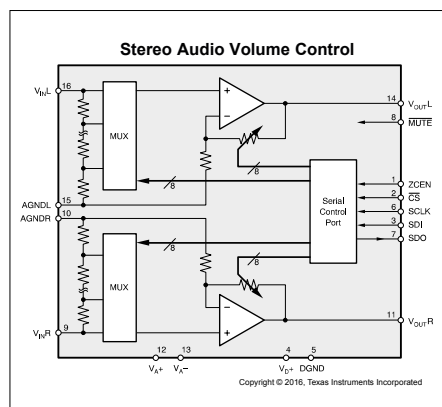


Figure 3. Amplifiers with variable gain are used for the volume control.

input signal. In this way spurious audio artifacts as a consequence of the change in volume can be minimized. On the circuit board ZCEN can be connected via a jumper to either +4.7 V or ground,

so that you have the choice whether to enable or disable this functionality. For a more detailed description of the inner workings of the PGA2311 we refer you to the datasheet [1]. Between the

microcontroller and the PGA2311 we provided test points on the circuit board for the serial control signals (CS, SDI, SCLK and MUTE). The input and outputs go via resistors

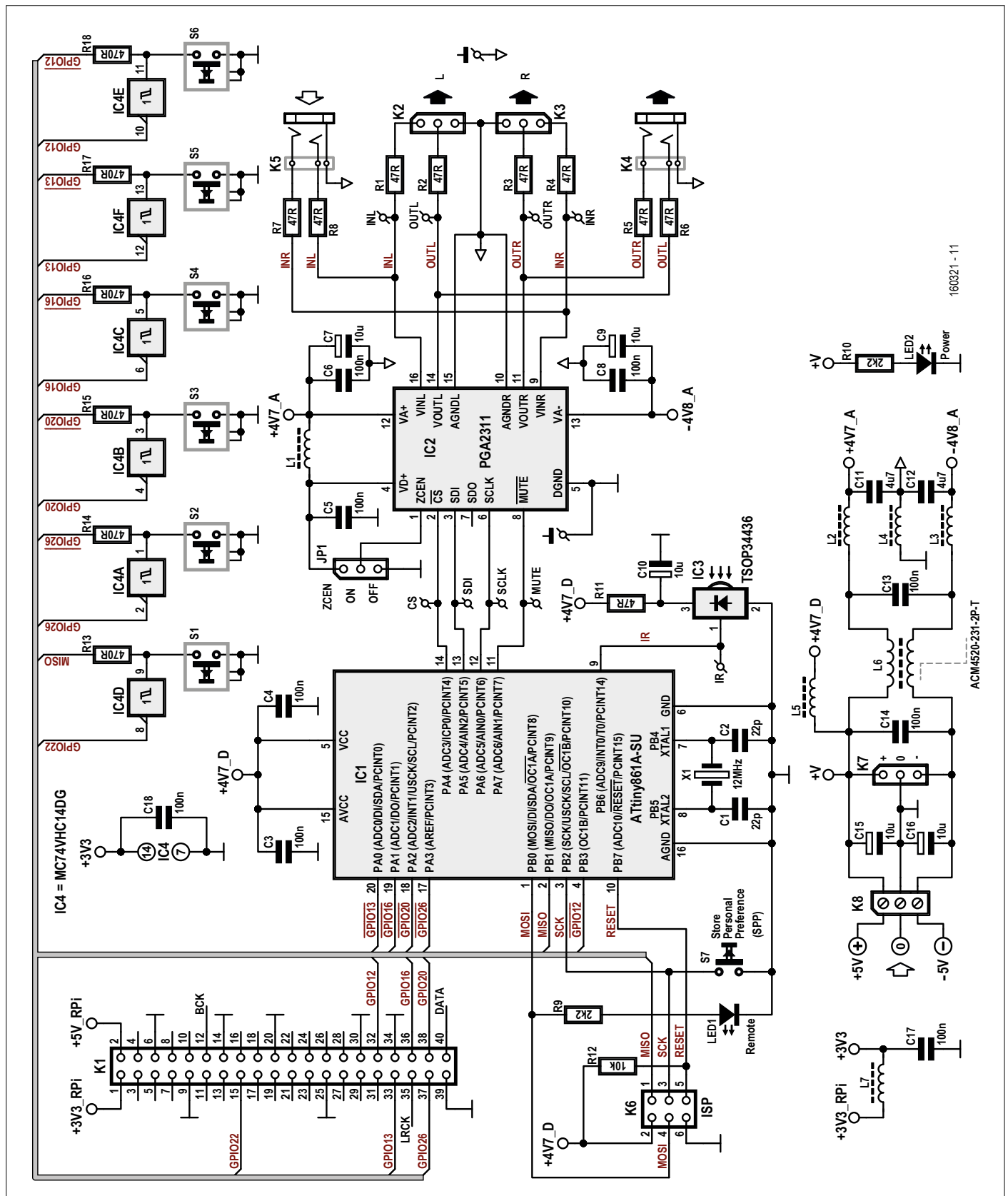


Figure 2. Unsurprisingly, at the heart of the circuit sits a microcontroller.

of 47 Ω (R1 through R8) to two different connectors. A pair of so-called stack-through-connectors (K2 and K3) connect the inputs and outputs of the DAC and the volume control (in the same way as is done for the power supply).

The inputs and outputs of IC2 also go to two 3.5-mm audio connectors (K4 and K5) on the circuit board. This means that the unprocessed DAC signal is available using a 3.5-mm jack (K5) or via the cinch connectors on the DAC board. The out-

put signal from the volume control is available on K4.

When the board is used as an 'independent' volume control, K5 is the input for the PGA231 and K4 is the output. The 47- Ω resistors protect the outputs from



COMPONENT LIST

Resistors

R1,R2,R3,R4,R5,R6,R7,R8,R11 = 47 Ω , 125mW, 1%, SMD 0805
R9,R10 = 2.2k Ω , 100mW, 5%, SMD 0805
R12 = 10k Ω , 100mW, 5%, SMD 0805
R13,R14,R15,R16,R17,R18 = 470 Ω , 100mW, 5%, SMD 0805

Capacitors

C1,C2 = 22pF, 50V, 5%, COG/NP0, SMD 0805
C3,C4,C5,C13,C14,C17,C18 = 100nF, 50V, 10%, X7R, SMD 0805
C6,C8 = 100nF, 25V, 5%, COG/NP0, SMD 1206
C7,C9,C10,C15,C16 = 10 μ F, 35 V, 0.04 Ω , 6.3mm max. diam., pitch 2mm or 2.5mm, Würth Elektronik 870055673001 (WCAP-PTHR Series)
C11,C12 = 4.7 μ F 25V, 20%, X5R, SMD 0805

Inductors

L1,L2,L3,L4,L5,L7 = 600 Ω @ 100MHz, 0.15 Ω , 1.3A, SMD 0603, Murata BLM18KG601SN1D
L6 = ACM4520-231-2P-T (TDK), 2 x 0.05 Ω , 230 Ω @ 100MHz, 2.6 A, SMD

Semiconductors

LED1 = red, T-1, 3mm
LED2 = green, low-power, SMD 0805
IC1 = ATtiny861A-SU, SOIC-20, programmed with firmware # 160321-11 (or 160321-12)
IC2 = PGA2311UA, SOIC-16
IC3 = TSOP34436, 3-pin epoxy package
IC4 = M74VHC14DG, SMD SOIC-14 (ON Semiconductor only)

Miscellaneous

K1 = 40-way (2x20) GPIO stacking socket (female), extra long
K2,K3,K7 = 3-way bus strip, vertical, 0.1" pitch (mounted on the bottom side of the board), with long stack-through pins
K4,K5 = 3-way stereo jack, 3.5mm, PCB mount, e.g. Lumberg KLBR 4
K6 = 6-way (2x3) boxheader, vertical, 0.1" pitch
K8 = 3-way PCB screw terminal block, 0.2" pitch, 630V
JP1 = 3-pin pinheader, vertical, 0.1" pitch
LED1 = 2-pin pinheader, vertical, 0.1" pitch
JP1 = jumper, 0.1" pitch

S1,S2,S3,S4,S5,S6 = pushbutton, right angle, SPST, FSMRA4JH TE Connectivity
S7 = pushbutton, SPST, FSM4JRT TE Connectivity
X1 = 12MHz miniature quartz crystal, 5x3.2mm, SMD, Cload 18pF
4 pcs 17mm M2.5 spacer (05.12.173 Ettinger)
4 pcs 6mm M2.5 bolt
4 pcs M2.5 nut
PCB 160321-1 v1.1

Optional

To support the LC display (instead of 4 bolts):
4 pcs 14mm M2.5 spacer (Ettinger 05.12.143)

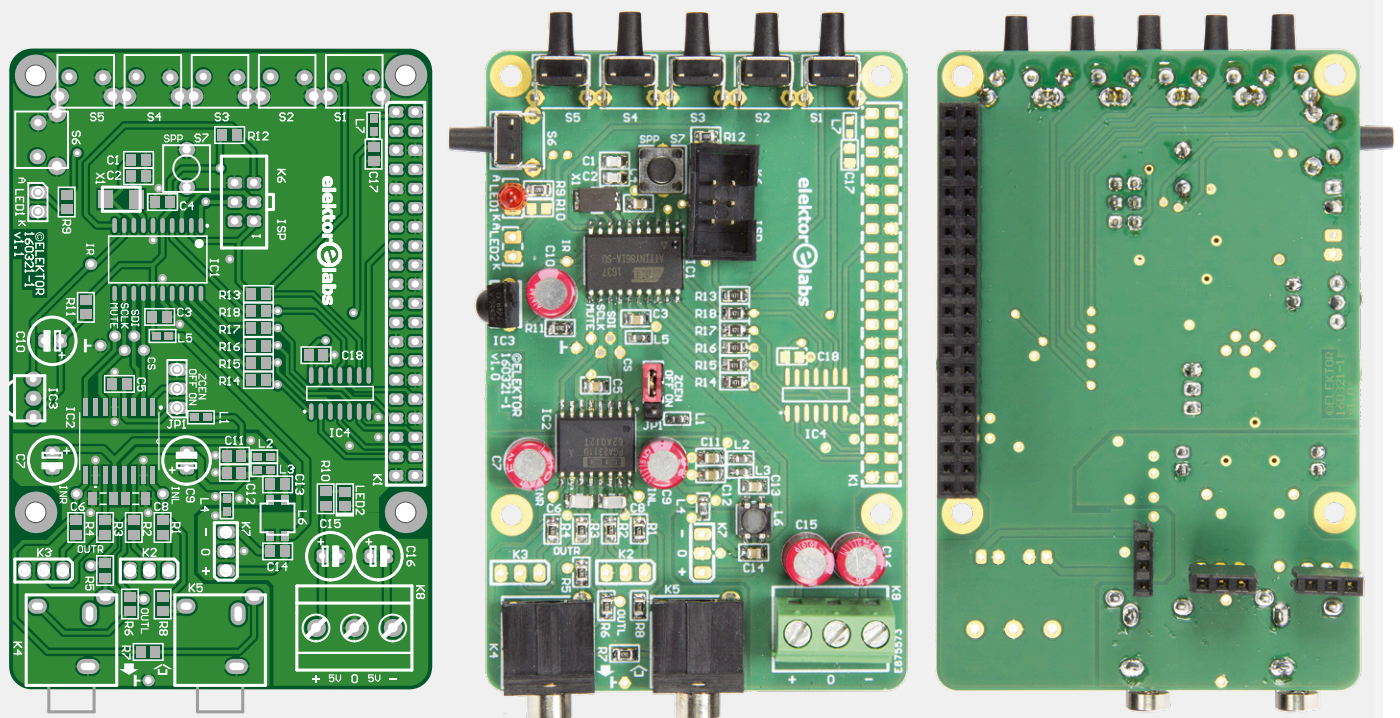


Figure 4. The circuit board for the volume control fits perfectly on the circuit board for the audio DAC.

capacitive loads and create a separation between the various connectors.

Construction

For the volume control we designed a circuit board with dimensions that are identical to those of the Raspberry Pi and the audio DAC (**Figure 4**). The board can be sandwiched with the RPi and the audio DAC; after that the display can be plugged into the entire assembly. **Figure 5** gives an impression of the complete assembly.

Note: for connectors K2, K3 and K7 (the so called stack-through connectors), parts with extra long pins are used; these have to be placed some distance away from the circuit board, otherwise they will not fit into the corresponding headers of the DAC board. And, naturally, you have to make sure that the connector bodies are fitted nice and parallel to the circuit board. Mutatis mutandis this is also true for the expansion connector K1, which is also fitted on the bottom side of the circuit board.

The circuit boards are firmly attached together using male/female standoffs; the same applies to the display.

The microcontroller

In order to get the PGA2311 to do what it needs to do, we need a microcontroller that sends the appropriate data to the serial interface of the volume controller. Here we use an ATtiny861A — a 20-pin IC with just enough I/O for the serial interface, seven pushbuttons, the IR receiver (for the remote control) and an LED. The latter (LED1) is a through-hole version that, if desired (using a PCB header on the board and a couple of wires), can potentially be mounted somewhere else. All the ISP I/O lines are also in use (except Reset); when you are programming the microcontroller (via connector K6) you have to make sure that your fingers are away from the pushbuttons! For the IR receiver we used a read-made solution in the form of the TSOP34436 (IC3); the output of which is also available as a test point. The IR protocol is RC5, for which there is an extremely handy command in Bascom (which is what we used when writing the firmware):

```
Getrc5(Address, Command)
```

Two version of the firmware are available,



Figure 5. Here you can clearly see how the sandwich is put together.



High-end volume control on a small surface area

depending on the version of Volumio that you are using. Both versions are part of the free download [3]. We will describe both versions below.

Firmware 160321-12

This version of the firmware was developed with Volumio 2 in mind [2]. It is provided with a plug-in for the GPIO buttons (**Figure 6**). Nearly all GPIOs can be linked to one for the following functions: play/pause, volume+/-, previous, next and shutdown. Because our DAC does not have its own volume control, four functions are used to operate Volumio; however for the benefit of other DACs that have a volume control we have also connected the functions volume+ and volume-. For these functions nearly any of the GPIO can be freely selected – provided that they are not already used for I²S or the 3.5" touch-sensitive display.

The GPIOs are provided with pull-downs

by default. Our Volumio version (2.041 from 12.12.2016) does not (yet?) support the 3.5" LCD. In **Table 1** we summarize the functions of all the pins of the RPi expansion connector for you.

We use the RC5 commands from **Table 2**.

For the GPIOs logic '1' (High) corresponds to 3.3 V. We therefore need to use an MC74VHC14DG (IC4, hex Schmitt-trigger inverter — note: from ON Semiconductor) as a level shifter. The inputs of this IC can withstand voltages up to 7 V — perfect for our purpose. The corresponding I/Os of the microcontroller have to be High in the inactive state. Consequently the two I/Os that are configured as the inputs for the volume control (PA0/PA1), the pullups have to be turned on; the outputs that are configured for the four functions (PA2/PA3/PB1/PB3) have to be High in the inactive state.

Table 1: Raspberry Pi Expansion Connector Pin Function				
K1	RPI (v3)	160321-12	I ² S	3,5" LCD
1	3,3 V	3,3 V		3,3 V
2	5 V			5 V
3	GPIO2 (SDA1)			NC
4	5 V			5 V
5	GPIO3 (SCL1)			NC
6	GND	GND	GND	GND
7	GPIO4 (GPIO_GCLK)			NC
8	GPIO14 (TXD0)			NC
9	GND	GND	GND	GND
10	GPIO15 (RXD0)			NC
11	GPIO17 (GPIO_GEN0)			TP_IRQ
12	GPIO18 (GPIO_GEN1)		BCK	NC
13	GPIO27 (GPIO_GEN2)			NC
14	GND	GND	GND	GND
15	GPIO22 (GPIO_GEN3)	S1/Play/Pause		NC
16	GPIO23 (GPIO_GEN4)			NC
17	3,3 V			3,3 V
18	GPIO24 (GPIO_GEN5)			LCD_RS
19	GPIO10 (SPI_MOSI)			LCD_SI/TP_SCK
20	GND	GND	GND	GND
21	GPIO9 (SPI_MISO)			TP_SO
22	GPIO25 (GPIO_GEN6)			RST
23	GPIO11 (SPI_SCLK)			LCD_SCK/TP_SCK
24	GPIO8 (SPI_CE0_N)			LCD_CS
25	GND	GND	GND	GND
26	GPIO7 (SPI_CE1_N)			TP_CS
27	ID_SD			
28	ID_SC			
29	GPIO5			
30	GND	GND	GND	
31	GPIO6			
32	GPIO12	S6/Shutdown		
33	GPIO13	S5/Volume up		
34	GND	GND	GND	
35	GPIO19		LRCK	
36	GPIO16	S4/Volume down		
37	GPIO26	S2/Previous track		
38	GPIO20	S3/Next track		
39	GND	GND	GND	
40	GPIO21		DATA	

Table 2: RC5 commands	
Play/pause	RC5 command 1 (Digit entry)
Previous track	RC5 command 2 (Digit entry)
Next track	RC5 command 3 (Digit entry)
Shutdown	RC5 command 12 (Standby)
Mute/de-mute	RC5 command 13
Personal preference settings	RC5 command 14
Increase sound volume	RC5 command 16
Decrease sound volume	RC5 command 17
Shift sound balance to the right	RC5 command 26
Shift sound balance to the left	RC5 command 27

The six pushbuttons (S1 through S6) are connected via resistors of 470 Ω to the I/Os of the microcontroller. When one of these buttons is pushed the resistors prevent a short-circuit at the output of the μC, while at the same time the corresponding inverter is activated by making its input Low. The value of the resistors is low enough to make the inputs of the μC logic low.

When pushbutton S7 is pressed the volume level that is currently set will be stored as a personal preference. This button is not with the other six near the edge of the circuit board because it is likely it will be used only sporadically. LED1 flashes briefly four times when S7 is pushed, to indicate the present setting of the volume level is being written to EEPROM. After switching on this setting is also used as the initial value.

When a new microcontroller is used for the first time, a default volume level of -20 dB is set. This can be ‘surprisingly’ loud — you are warned!

LED1 indicates that an RC5 command has been received — but that does not necessarily have to be one of the commands actually supported by the volume control. The IR receiver (a Vishay type suitable for TC5) is connected to PB6 because the Getrc5 command uses TIMER0 and the TIMER0 interrupt. If you are going to develop your own software for another IR protocol, then do make sure you use the correct IR receiver. In any case, the connections for the receiver are fairly common.

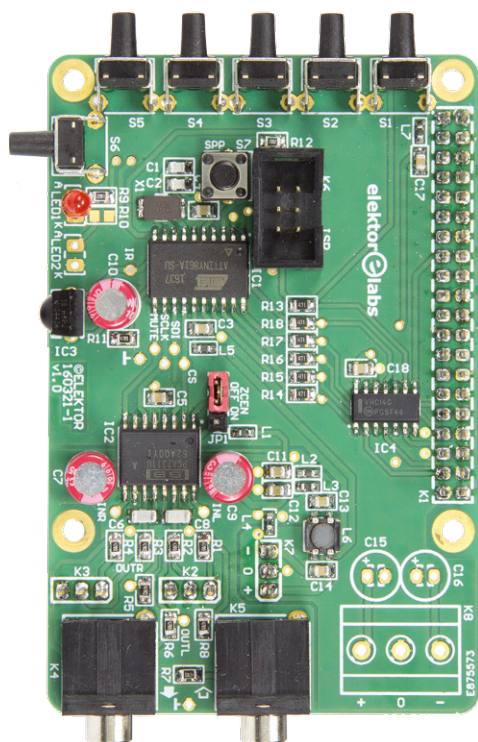
While we were developing the software we noticed something strange: when the ‘divide clock by 8’ is turned off, the Getrc5 statement doesn’t work very well. This means that the ‘ckdev8’ fuse has to be enabled when programming the fuses in the microcontroller (see **Figure 7**)! For this reason we have increased the crystal frequency for the microcontroller to 12 MHz (X1 in the schematic).

Firmware 160321-11

This version of the firmware was initially developed for use with Volumio 1.55 and our Raspberry Pi Audio DAC (ref Elektor July & August 2017). However, this version can also be used with Volumio 2 — or for entirely other applications. Note: when you use the volume control as a stand-alone device, you need a separate symmetrical 5-V power supply! In order to prevent ground loops we strongly suggest using an isolated power supply;

additionally L4 has to be replaced with a 0-Ω resistor (0603) or a wire link. If K4 and K5 are fitted and the appropriate cables are used, the volume control can be inserted into any analog audio signal path (V_{max} amounts to 2 V). In this way you can provide any amplifier with a remote volume control.

It is also possible to use connectors K2 and K3, instead of the 3.5-mm audio connectors K4 and K5. In that case circuit board headers have to fitted on the board (or the connecting wires can be soldered directly to the circuit board). Pushbuttons S1 through S6 are used only for changing the volume settings; these functions are, together with the RC5 commands you can find summarized in **Table 3**.



Just as with the other firmware version, here S7 is also used to store the current volume setting as a personal preference in EEPROM.

When the mute function (IC2 pin 8) is activated (via software or hardware), the internal preamplifiers in the PGA2311 are disconnected from the outputs and VOUTL and VOUTR are connected to ground via 10-kΩ resistors. This can increase the noise in a connected amplifier. Also when the mute function is activated an internal offset calibration is carried out.

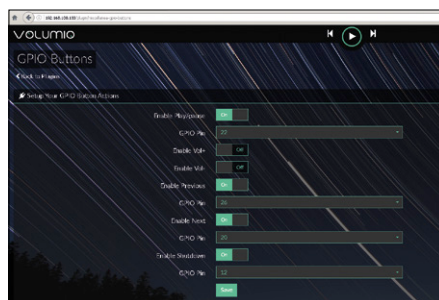


Figure 6. The Volumio GPIO plugin.

Volumio Mk. 2

At the time of writing, the small 3.5" touch screen is not (yet) supported in Volumio 2. When the plugin for supporting the original Raspberry Pi display is

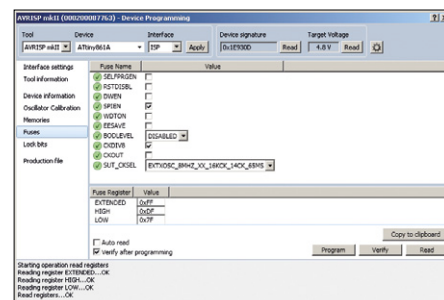


Figure 7. Must-know: the programming of the fuses.

installed, a standard monitor can be connected to the HDMI connected of the RPi. Another hint related to the installation of this plugin: do not connect a keyboard to the RPi. If you are using a dongle for

Table 3: Pushbuttons and RC5 commands (firmware 160321-11)

S1	personal preference	RC5 command 14
S2	balance to the left	RC5 command 27
S3	balance to the right	RC5 command 26
S4	volume down	RC5 command 17
S5	volume up	RC5 command 16
S6	mute	RC5 command 13

Table 4: Selected measurement results

Power supply current	
LED1 off	11 mA
LED1 on	12 mA
Maximum output voltage (1 kHz, THD = 0.1%)	
0 dB gain	2.3 V
20 dB gain	2.6 V
-20 dB gain, (3.9 V in)	0.39 V
THD+N	
1 kHz, B = 22 kHz	0.0003% (0 dB, 2 V out)
1 kHz, B = 80 kHz	0.00042% (0 dB, 2 V out)
20 kHz, B = 80 kHz	0.0022% (0 dB, 2 V out)
1 kHz, B = 22 kHz	0.0016% (20 dB, 2 V out)
1 kHz, B = 80 kHz	0.0025% (20 dB, 2 V out)
20 kHz, B = 80 kHz	0.0035% (20 dB, 2 V out)
IMD (50 Hz : 7 kHz = 4:1)	
0 dB gain, 2 V in	0.0007%
20 dB gain, 200 mV in	0.0038%
DIM (3.15 kHz square wave + 15 kHz sine)	
0 dB gain, 2 V in	0.0009%
20 dB gain, 200 mV in	0.0008%
Bandwidth (-3 dB)	
0 dB, output triangle	1.1 MHz
20 dB	690 kHz
Crosstalk	
1 kHz	< -100 dB
20 kHz	< -90 dB

the mouse/keyboard combo it is better to remove that too. It is recommended to start the installation from within the WebUI.

Measurements

Just as with the audio DAC for the RPi we have performed a few measurements on our volume control; the most important results are summarized in **Table 4**.

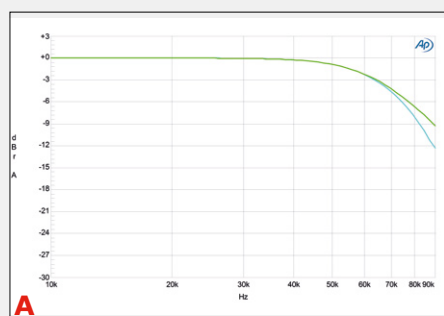
We have also made a few plots with our Audio Precision Analyzer, both for the audio DAC as well as the volume control. These are further explained in the sidebar. ◀

(160321)

Web Links

- [1] Datasheet PGA2311: www.ti.com/lit/ds/symlink/pga2311.pdf
- [2] Volumio: <https://volumio.org>
- [3] Download: www.elektormagazine.com/160321

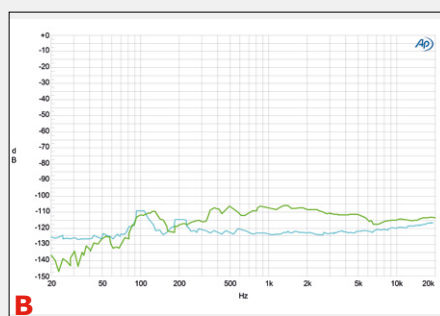
Measurements on the Big DAC



One of the showpieces of our former audio lab in Holland is a full blown Audio Precision Analyzer — which fortunately has survived the recent move to Aachen intact. Naturally we have tested the ‘Big DAC’ using our analyzer; below you can see some of the results.

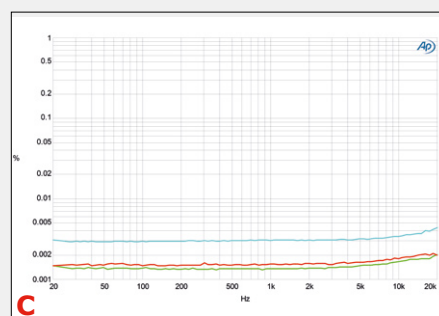
Audio-DAC

Plot A shows the amplitude as a function of the frequency with a stepped sine signal (70 frequencies, full scale) at a sampling frequency of 192 kHz. The top (green) curve was measured with a steep output filter; the bottom (blue) curve with is less steep filter. The corner frequency amounts to 63.5 kHz (64.5 kHz in theory). At 90 kHz the amplitude is -9.2 dB (green) and -12.3 dB (blue) respectively. At 45 kHz and 51 kHz both curves are equal (-0.5 dB and -1 dB respectively).



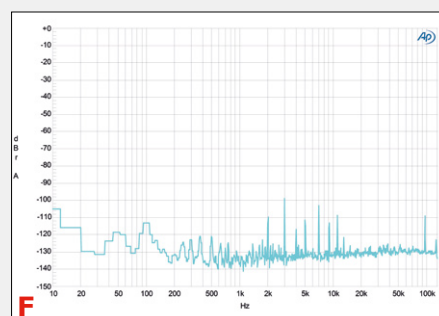
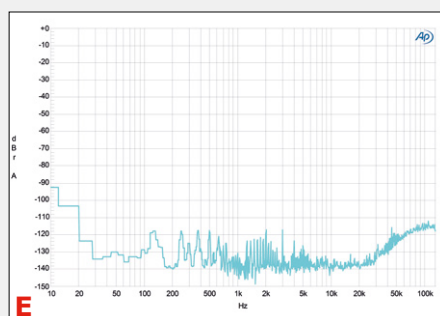
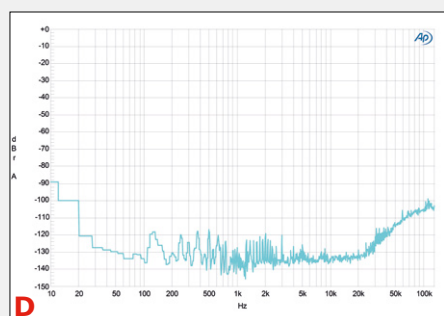
Plot B gives an impression of the channel separation as a function of frequency. The top (green) curve is the crosstalk from the left channel to the right channel. In this direction the crosstalk is slightly larger than the other way around. This is because the length of the signal traces on the circuit board to the output connectors are not the same. The trace for the right channel is longer, with the consequence of more crosstalk from left to right. Nevertheless, this is a good result for such a compact design.

Plot C shows the total harmonic distortion plus noise (THD+N) as a function of frequency at sample frequencies of 48, 96 and 192 kHz. In all cases the bandwidth is 80 kHz. Incidentally, at a bandwidth of 22 kHz the THD+N is much lower when the sampling frequency is 48 kHz.



Plot D shows the FFT analysis of a 1 kHz full-scale sine at a sampling frequency of 32 kHz. Remarkable is the increase of the noise just above the audio bandwidth. This could be noise shaping. At a bandwidth of 22 kHz the THD+N amounts to 0.0007% (at a bandwidth of 80 kHz that is 0.012%). A large number of noise spikes are visible, mainly caused by the RPi, which is just below the DAC. The second and third harmonics of the sample frequency are just visible.

Plot E shows the same as Plot D, but then at a sampling frequency of 44.1 kHz. The high-frequency noise is here less when compared to the previous plot. The second and third harmonics of the sample frequency are a little bit more pronounced here; nevertheless the harmonic distortion is much lower.





FROM THE STORE

→ 160321-1

Bare printed circuit board

→ 160321-91

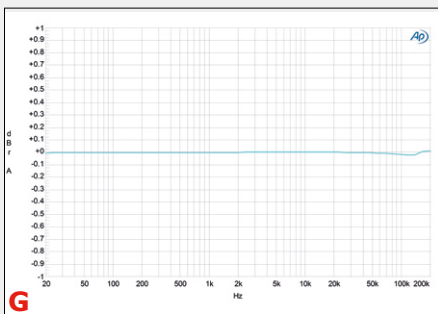
Assembled module
(headers supplied separately –
DIY soldering)

→ 160198-91

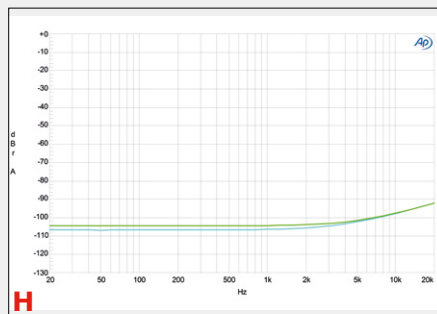
Assembled Audio DAC module
(headers supplied separately –
DIY soldering)

→ SKU 17631

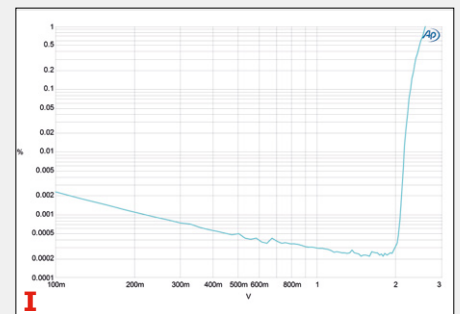
Raspberry Pi 3 (model B)



G



H



I

tion (without noise) amounts here to only 0.00019%.

Plot F finally shows the FFT analysis of a 16-bit full-scale sine of 1 kHz at a sampling frequency of 32 kHz, 'on the fly' converted to 24-bit/96-kHz (as is possible in Volumio 1.55). Remarkable is the absence of the high-frequency noise that is visible in Plot D.

Volume control

Plot G again shows the amplitude as a function of frequency at a gain of 0 dB and an output level of 2 V. The maximum frequency of the generator in the Audio Precision Analyzer goes up to 200 kHz. The small 'bump' around 100 kHz is not significant (note the vertical scale!).

Plot H shows the crosstalk as a function of frequency. The difference between the two curves amounts to about 2 dB — the circuit therefore behaves almost completely symmetrically.

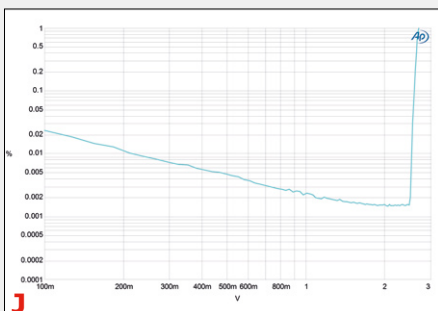
Plot I makes the THD+N visible as a function of output level at a gain of 0 dB. The bandwidth is here reduced to 22 kHz to better show where the circuit begins to clip. At this low power supply voltage the maximum undistorted output voltage is about 1.9 V. Above this level the THD increases quickly (0.1% at 2.3 V and 1% at 2.6 V).

Plot J is the same as Plot I, but now at a gain of 20 dB. The maximum undistorted output voltage is now surprisingly much higher, at 2.48 V (THD = 0.0015%) the circuit begins to clip. Apparently at lower

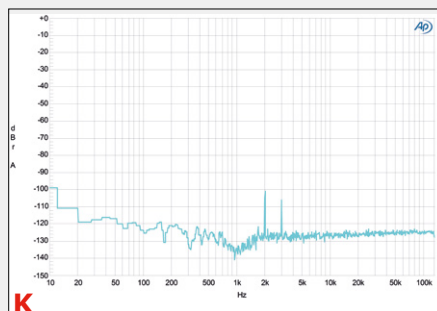
gain this occurs earlier in the input stage because the input voltage is then higher.

Plot K shows the FFT of a 1-kHz signal at a gain of 20 dB and an output voltage of 2 V. Here only the second and third harmonics are visible, corresponding to a THD (without noise) of 0.001%. (THD+N at 22 kHz bandwidth is 0.0016%, and 0.0025% at 80 kHz bandwidth).

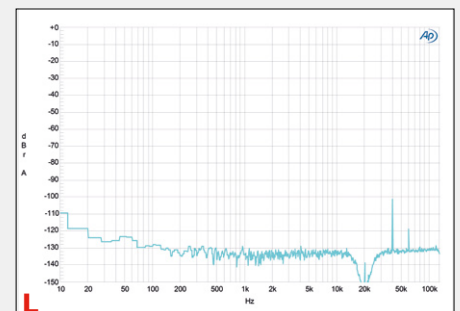
Plot L finally is the same as Plot K, but now for a 20-kHz signal. Here too the second and third harmonics are visible. THD = 0.00086%; THD+N at 80 kHz bandwidth is 0.0035%.



J



K



L

and Oranges

ite

Raspberry Pi; see also **Figure 3**.

In the following experiments we will use the Orange Pi Lite. Weighing in at some 28.8 grams it is lighter than the Raspberry Pi 3, and it also costs less, at around twelve dollars. Nevertheless it offers two USB ports and a Wi-Fi antenna. The Orange Pi One, on the other hand, has just one USB port instead of two, and Ethernet instead of Wi-Fi, and comes in even cheaper at around nine dollars.

Configuration

We have to say one thing for the Raspberry Pi Foundation: if you buy their processor boards you will find that support in terms of the software and the ecosystem is excellent. Also, a Raspberry Pi can be powered from any ordinary micro USB charger, which makes it very easy to get started.

The Orange Pi Lite also has a micro USB port but, amusingly enough, cannot be powered from it because of the interface components used: you can find out more about this at [2]. Instead, power at 5 V has to be applied at the power jack or at pins 2, 4 and 6 of the header (pins 2 and 4 to 5 V, pin 6 to ground). This inconvenience is present across the Orange Pi family: if you need to be able to power your computer over the micro USB port, you will have to avoid Shenzhen Xunlong's products.

The next problem is software. The operating system images that can be found on the manufacturer's website at [3] are little more than window dressing. The links sometimes fail to work, and any file you do manage to download will likely be poorly supported.

The Armbian distribution has established itself more or less as a standard among the Orange Pi user community: downloads are available at [4]. Note that the Armbian distribution builds are different for the different members of the Orange Pi family. With our Orange Pi Lite we will be using a Debian-based distribution, but the Ubuntu-based version should work just as well. The differences between the versions are in general only small but they do manifest themselves from time to time and that means that you need to take extra care when testing.

Download the installation image [4a] and write it onto a sufficiently

Android on board

The internal flash memory of the Orange Pi PC Plus contains a Chinese version of Android that is reportedly quite usable.

ciently high-capacity SD card. Connect the processor board to an HDMI display, mouse and keyboard, and then apply power. Don't be surprised if it takes a minute or so before you see anything happen: the Orange Pi boards generally take a long time to start up and present an image on the screen.

It runs Linux!

Armbian images are classical computer operating systems: when first started up they present the user with a login prompt. The credentials to use at this point vary from image to image: on the Orange Pi Lite you can log in with the user name 'root' and password '1234'.

After the first login the operating system will prompt you to change your password for security reasons and to run an installation wizard. The user name has to be supplied, but the other fields can be left blank by simply repeatedly pressing 'Enter'. After that, you can reboot the system and you will be presented again with Armbian's graphical user interface (see **Figure 4**). Armbian is Debian, a Linux distribution with a long history which also serves as the basis for the Raspbian distribution used on the Raspberry Pi. It follows, therefore, that at the application level the Raspberry Pi and the Orange Pi are more or less compatible. The first small difference comes in how external hardware is controlled. Where Raspberry Pi uses the WiringPi library, the people of Shenzhen Xunlong bring you WiringOP. The library must be downloaded and compiled before it can be used, which (assuming all goes well) can be done using the following sequence of commands:

```
cd ~
mkdir opigpiolib
git clone https://github.com/zhaolei/WiringOP.git -b
```

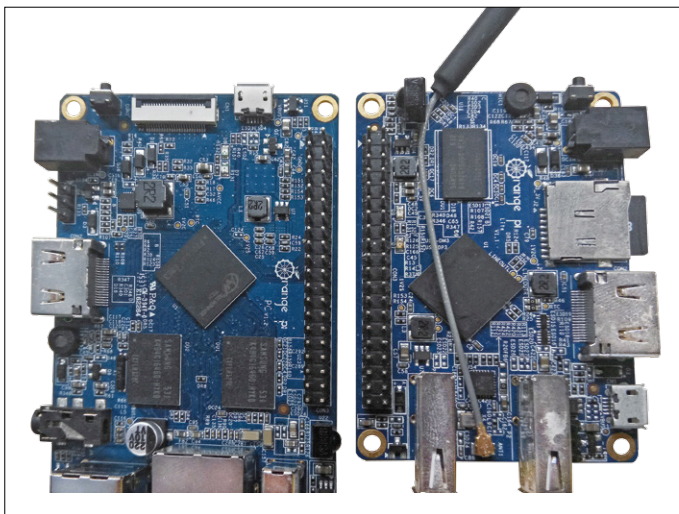


Figure 3. Watch out! The orientation of the headers on the two boards is different.

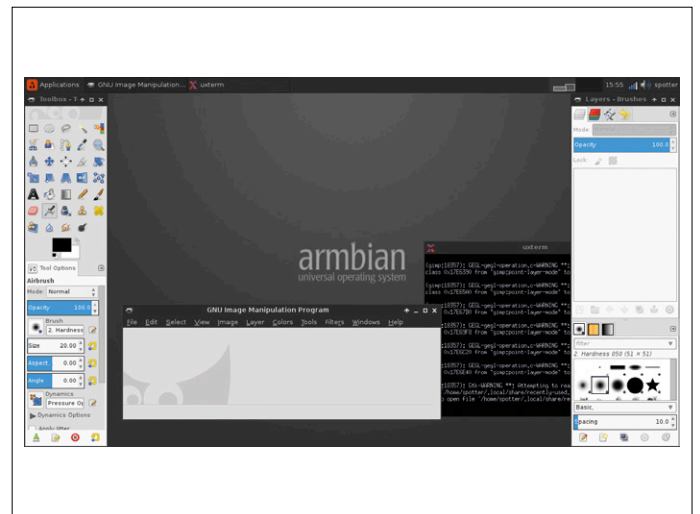


Figure 4. Just like the Linux we know and love.

Orange Pi										
BCM	wPi	Name	Mode	V	Physical	V	Mode	Name	wPi	BCM
		3.3V			1	2		5v		
12	8	SDA.0	ALT5	0	3	4		5v		
11	9	SCL.0	ALT5	0	5	6		0v		
6	7	GPIO.7	ALT3	0	7	8	0	ALT3	TxD3	15
		0v			9	10	0	ALT3	RxD3	16
1	0	RxD2	ALT3	0	11	12	0	ALT3	GPIO.1	1
0	2	TxD2	ALT3	0	13	14		0v		110
3	3	CTS2	IN	0	15	16	0	ALT3	GPIO.4	4
		3.3v			17	18	0	ALT3	GPIO.5	5
64	12	MOSI	IN	0	19	20		0v		68
65	13	MISO	IN	0	21	22	0	ALT3	RTS2	6
66	14	SCLK	IN	0	23	24	0	ALT4	CE0	10
		0v			25	26	0	ALT3	GPIO.11	11
19	30	SDA.1	ALT4	0	27	28	0	ALT4	SCL.1	31
7	21	GPIO.21	ALT3	0	29	30		0v		18
8	22	GPIO.22	OUT	0	31	32	0	ALT3	RTS1	26
9	23	GPIO.23	OUT	1	33	34		0v		200
10	24	GPIO.24	OUT	0	35	36	0	ALT3	CTS1	27
20	25	GPIO.25	OUT	0	37	38	0	ALT3	TxD1	28
		0v			39	40	0	ALT3	RxD1	29
										199
BCM	wPi	Name	Mode	V	Physical	V	Mode	Name	wPi	BCM

Figure 5. The GPIO options on the Orange Pi are rather complicated.

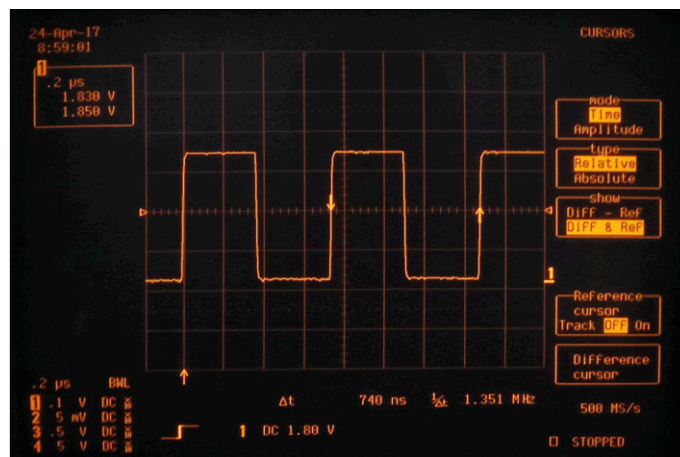


Figure 6. The square wave looks usable.

```
h3
cd WiringOP
chmod +x ./build
sudo ./build
```

In the next step we look at a small example program that generates a square wave on a GPIO pin. To do this, create a file called 'main.c' with the following contents:

```
#include <wiringPi.h>
int main (void) {
    wiringPiSetup ();
    pinMode (0, OUTPUT);
    for (;;) {
        digitalWrite (0, HIGH);
        digitalWrite (0, LOW);
        digitalWrite (0, HIGH);
        digitalWrite (0, LOW);
    }
    return 0;
}
```

The process of compiling this code is largely identical to the familiar process on the Raspberry Pi:

```
$ gcc -Wall main.c -lwiringPi
$ sudo ./a.out
```

A sensitive flower

Orange Pi boards are extremely sensitive to a supply voltage that is too low: this can lead to crashes when graphics mode is enabled and read failures on the SD card. If you have problems booting up your new computer, one of the first things to check is the voltage drop across the supply cables. Note that the board can draw currents of up to 800 mA!

On some versions of the Orange Pi you will see compilation errors that stem from the fact that the 'pthread' library is not present:

```
$ gcc -Wall main.c -lwiringPi
//usr/local/lib/libwiringPi.so: undefined reference
to 'pthread_join'
//usr/local/lib/libwiringPi.so: undefined reference
to 'pthread_create'
//usr/local/lib/libwiringPi.so: undefined reference
to 'pthread_cancel'
collect2: error: ld returned 1 exit status
```

This in turn results from misconfiguration of the gcc compiler, which is not automatically linking in the pthread library. To solve this it is simply necessary to add another option to the compiler command line as follows:

```
$ gcc -Wall main.c -lwiringPi -pthread
$
```

It should be possible to obtain the layout of the individual pins on the GPIO header using the command 'gpio readall'. For simplicity we give an example result from this command run on a 40-pin Orange Pi PC Plus in **Figure 5**.

We can now team up the processor board with a duo of instruments: a modulation domain analyzer and a digital storage oscilloscope. **Figure 6** and **Figure 7** show the results you might get.

Enabling the UART

In our next example we will show how to read an RFID card. The Seeed RFID transponder module, which operates at 125 kHz, is ideal for this task as it only requires a small amount of configuration. It is available from Mouser (order code 713-113020002) for around twelve dollars including antenna. **Figure 8** shows the electronics required to connect up this module: whereas the Raspberry Pi 3 is to some extent 5 V tolerant, this is not the case for the Orange Pi and a (not especially sophisticated) level shifting circuit is required.



Figure 7. The stability of the waveform, too, leaves not much to carp about.

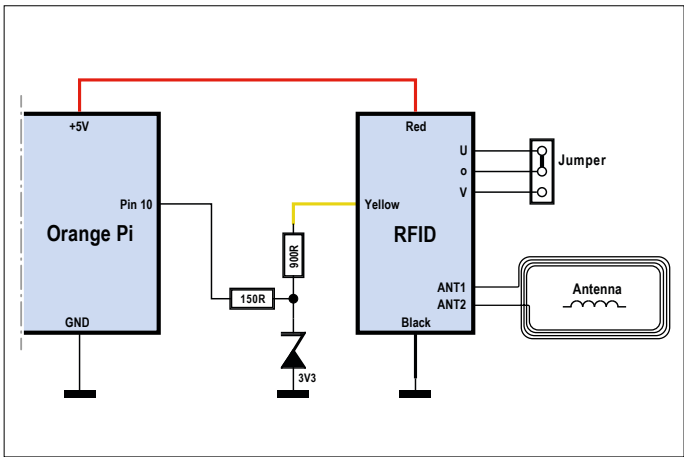


Figure 8. A voltage level shifter is essential.

Allwinner supplies the hardware configurations for its SoCs in the form of a binary file which is loaded into the device by the kernel as part of the boot process to set up the voltage regulators, clock sources and various other subsystems. The UART used in our program example is not normally accessible, as can be seen from a look at the /dev directory where only one UART (ttyS0) is present:

```
$ ls -l
. . .

crw--w---- 1 root tty      4,   9 Apr 12 06:07 tty9
crw--w---- 1 root tty    251,  0 Apr 12 06:08 ttyS0
crw----- 1 root root    248,  0 Apr 12 06:07 tv
```

To get around this problem the first step is to modify the configuration file that is loaded into the hardware during the boot process. The scripts used for the various boards can be found on the SD card in the directory '/boot/bin', which under Arm-bian looks like this:

```
$ ls
aw-som-a20.bin      bananapirolcd7.bin  lime-a10-lcd.bin  nanopi2r1.bin      orangepilite.bin   pcduino2.bin
bananapi.bin        beelinkx2.bin       lime-a10.bin      nanopi2r2.bin      orangepione.bin    pcduino3.bin
bananapilcd7.bin    cubieboard.bin      lime-a33.bin      nanopi2r3.bin      orangepipc.bin      pcduino3nano.bin
bananapi1plus.bin   cubieboard2.bin     lime.bin          nanopi2r4.bin      orangepipcplus.bin
bananapi1pluslcd7.bin cubieboard2dual.bin lime2-emmc.bin    olinux-som-a13.bin orangepiplus.bin
bananapi2plus.bin   cubietruck.bin      lime2.bin         orange2i.bin        orangepiplus2e.bin
bananapiro.bin      lamobo-r1.bin        micro.bin         orange2i2.bin       orangepizero.bin
```

Meanwhile, the file that is actually used to initialize the board lies in the directory immediately above this one under the name 'script.bin'. However, this is in fact a symbolic link, and the `realpath` command can tell us the real file to which it corresponds:

```
$ pwd
/boot

$ realpath script.bin
/boot/bin/orangepipcplus.bin
```

These configuration files can exist in two different forms: as well as the binary version that the processor itself understands there is a '.fex' format, which is human-readable. We can convert the binary file (the one for our processor is, as we discovered above, called '/boot/bin/orangepipcplus.bin') to readable form using the following command:

```
$ sudo bin2fex orangepipcplus.bin orangepipcplus.fex
[sudo] password for spotter: . . .
fexc-bin: orangepipcplus.bin: version: 1.2
fexc-bin: orangepipcplus.bin: size: 36392 (84
sections), header value: 36392
```

The structure of a .fex file is reminiscent of that of a .ini file, and is documented in detail at [5]. Of interest to us at the

moment is how the UART that is connected to the RFID reader is configured. Edit the block so that it contains the following information:

```
[uart3]
uart_used = 1
```



```

uart_port = 3
uart_type = 4
uart_tx = port:PA13<3><1><default><default>
uart_rx = port:PA14<3><1><default><default>
uart_rts = port:PA15<3><1><default><default>
uart_cts = port:PA16<3><1><default><default>

```

like this:

```

[uart2]
uart_used = 0
uart_port = 2
. . .

```

For comparison, a block that describes a disabled UART looks

However, it is not sufficient simply to modify the .fex file and

reboot, as this will not change the behavior of the machine. We first have to convert the edited .fex file back into a .bin file, using the command `fex2bin`.

```

$ sudo fex2bin orangepipcplus.
fex orangepipcplus.bin

```

And now at this point we reboot the machine and admire the fruits of our handiwork: the new serial port appears in the /dev directory

```

crw--w---- 1 root tty
251,  0 Apr 24 13:34 ttyS0
crw-rw---- 1 root dialout
251,  3 Apr 24 13:34 ttyS3
crw----- 1 root root
248,  0 Apr 24 13:34 tv

```

Enabling and configuring a UART is just one of the possibilities opened up by the ability to edit a .fex file. Another example would be changing the processor clock frequency to gain more processing speed or to save power, and more information on this topic can be found in the relevant forums. But meanwhile, let's crack on with reading some RFID data.

Scanning an RFID card

All we need to make a fully-functional card reader is a program to read serial data delivered from the UART. Its first job is to open the connection to the UART: see the first part of **Listing 1** [6].

The next step is to read in the incoming data. In view of the ever-increasing importance of RFID technologies the author feels a brief digression is justified. There are two basic types of reader. The first group, to which our module belongs, reads all the information it can find on the card and sends this as a serial data stream to whatever it is connected to. Readers in group number two implement a Wiegand interface. This involves a protocol that communicates using pulses on one of two data lines.

Listing 1. Reading an RFID card over a UART.

```

#include <stdio.h>
#include <string.h>
#include <errno.h>

#include <wiringPi.h>
#include <wiringSerial.h>

int main ()
{
    int fd ;
    int count ;
    unsigned int nextTime ;

    if ((fd = serialOpen ("/dev/ttyS3", 9600)) < 0)
    {
        fprintf (stderr, "Unable to open serial device: %s\n", strerror (errno)) ;
        return 1 ;
    }

    if (wiringPiSetup () == -1)
    {
        fprintf (stdout, "Unable to start wiringPi: %s\n", strerror (errno)) ;
        return 1 ;
    }

    nextTime = millis () + 300 ;

    for (count = 0 ; count < 256 ; )
    {
        delay (3) ;

        while (serialDataAvail (fd))
        {
            printf (" -> %3d", serialGetchar (fd)) ;
            fflush (stdout) ;
        }

        printf ("\n") ;
        return 0 ;
    }
}

```

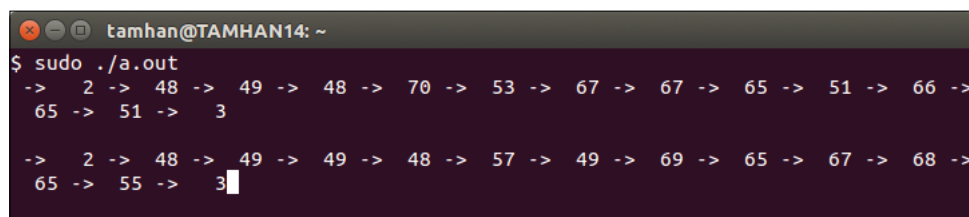


Figure 9. Two different RFID cards duly recognized: press 'Enter' after each scan to split up the lines of data.

Risk of collision!

When several programs try to access the same serial bus at the same time, collisions can occur. Experience shows that the operating system does not detect this and the individual programs only receive fragments of the data they are expecting.

Despite being limited to data lengths of a mere 26 bits or 31 bits, the Wiegand protocol shows no sign of going away and so remains relevant in embedded control applications. Most RF readers, in fact, are available in both Wiegand and UART editions, with the UART edition being the more sophisticated. For simplicity our program gathers data one byte at a time (see Listing 1) and displays them in the terminal window.

We are now ready for a first test. Connect the card reader to the computer, compile the program in the usual way, and bring a card near to the red reader coil. The display in the terminal window should look something like **Figure 9**.

Going further

As we have seen, the Orange Pi is a classical embedded controller board that runs the Linux operating system. That means that it supports a range of protocols such as I²C and SPI in much the same way as the Raspberry Pi and its friends. Anyone who is willing to get to grips with the Linux I²C and SPI APIs (or who is willing to use the WiringOP library) will find it

easy to connect a diverse range of peripheral devices, including displays and sensors in all shapes and sizes.

Conclusion

When the first Orange Pi computer saw the light of day, many had their doubts regarding the expected longevity of the manufacturer. Two years later, and Shenzhen Xunlong is still going strong and has brought many new models to market. Also, practically all of the devices are still readily available. Using an Orange Pi is particularly beneficial in mass-produced systems: a cost reduction to the tune of 50 % or even more is not something to be sniffed at. In small quantities there is the question of development costs: if you are not an expert Linux user, you will probably be better off with the support of the larger Raspberry Pi community. ◀

(160456)

Web Links

- [1] www.orangepi.org/
- [2] <https://www.youtube.com/watch?v=4zICevDOhr8>
- [3] www.orangepi.org/downloadresources/
- [4] www.armbian.com/download/
- [4a] <https://dl.armbian.com/orangepilite/>
- [5] http://linux-sunxi.org/Fex_Guide
- [6] www.elektormagazine.com/160456

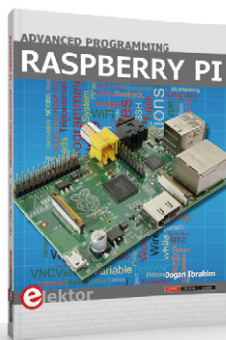
Advertisement

LEARN > DESIGN > SHARE

Raspberry Pi @ Elektor

It's very easy to connect the Raspberry Pi 3 to a computer or a television set. You can program this high-performance computer yourself, use it to play videos, play games with it, and use it to develop your own highly innovative concepts.

www.elektor.com/rpi3



Raspberry Pi Advanced Programming
www.elektor.com/rpi-book



Raspberry Pi 3 Starter Kit (Deluxe)
www.elektor.com/rpi-deluxe



pi-top DIY Laptop Kit
www.elektor.com/pi-top



More Raspberry Pi at www.elektor.com/rpi

Raspberry Pi Internet Radio

Tune in and chill out

By **Roy Aarts** (Elektor Labs)

With this project you can listen to any online radio station, without forking out on a dedicated Internet radio.

Features

- Plays streaming audio from Internet radio stations
- Uses built-in audio jack
- Operates stand-alone
- Add your own favorite radio stations
- No soldering required

Internet radios are available widely and in many shapes and sizes, but they can get quite expensive. With this project you can make one yourself using the by now ubiquitous Raspberry Pi 2. Through the headphone jack it plays audio decoded from a radio stream on the Internet. To make it you need a Raspberry Pi, a 2.8-inch touchscreen, an SD card, and optionally a Wi-Fi dongle. The user interface is made in python using pygame. The code is easy to understand and can be adapted to your own taste.

Installation

First we need to install Raspbian on the Raspberry Pi. Download Raspbian Jessie from [1] and use Win32DiskImager, free download from [2], to burn the *image* on an empty SD card with a capacity of at least 4 GB. When done, insert the SD card in the Raspberry Pi SD card slot, attach a monitor, mouse, keyboard and a USB Wi-Fi dongle (or a hard-wired Internet connection) and power it up. When the Raspberry Pi has finished booting we can set the network like in the top right corner in **Figure 1**. After connecting to the network, write down the IP address of the Raspberry Pi. This is useful for configuring it later. To get the IP address, hover the mouse over the icon used to connect to the network.

Now we have to install the touchscreen libraries. If you haven't already attached the touchscreen, now is the time. But not before having shut down the Raspberry Pi (top left corner, select Shutdown)! Attach the touchscreen and power the RPi on again. We will install the necessary drivers via the computer. Download PuTTY from [3], start the program and enter the IP address of your RPi in the bar that says "Host address



(or IP address)". If a notification pops up, click "Yes". Now a terminal window should open, asking you to log on. Log on with username "pi" and password "raspberrypi". Now type

```
sudo wget -N https://github.com/watterott/RPi-Display/raw/master/rpi-display.sh
```

and press Enter. This will download the necessary files from github. When the prompt returns, type

```
sudo /bin/bash rpi-display.sh 270
```

This will install the github files in the system. After some time, some questions appear. Answer them as follows:

- | | | |
|--|-----|----------|
| • Activate the console on the TFT display? | y/n | Y |
| • Install fbcp (Framebuffer Copy)? | y/n | N |
| • Install xinput-calibrator? | y/n | Y |
| • Install tslib (touchscreen library)? | y/n | Y |
| • Calibrate touchscreen now? | y/n | N |
| • Reboot the system now? | y/n | Y |

The Raspberry Pi will reboot and the connection with the computer will be lost. If everything goes right, you should see the user interface on the touchscreen when the Raspberry Pi boots up.

Calibration & music player

To configure the touchscreen, reconnect to the RPi using PuTTY. Log in and type


```
sudo TSLIB_FBDEVICE=/dev/fb1 TSLIB_TSDEVICE=/dev/
input/touchscreen ts_calibrate
```

On the touchscreen you'll see the calibration program (see **Figure 2**). After completing the steps shown on the screen the touchscreen is calibrated.

Now we can install the music player client ('mpc') and daemon. To do this, type the following commands:

```
sudo apt-get update
sudo apt-get install mpc mpc
```

User interface

Now we need to install the user interface. For this we use winSCP (free download from [4]) and the download provided on the Elektor magazine website for this project [5]. We have to copy the files from the download onto the SD card in the Raspberry Pi. So we close PuTTY and start winSCP. We log in with the same credentials as used in PuTTY.

If a notification pops up, click "Yes". From the download we copy the folder 'pi-radio' onto the Raspberry Pi. Then we copy the files 'launcher.sh' and 'shutdown.sh' to the Desktop folder of the Raspberry Pi. These two files launch the radio and shut down the RPi. Now we can close winSCP and open PuTTY again.

The two files we just copied onto the Pi's desktop do not work as clickable icons yet. To fix this, in PuTTY, type:

```
sudo chmod +x /home/pi/Desktop/launcher.sh
```

and

```
sudo chmod +x /home/pi/Desktop/shutDown.sh
```

Radio stations

The radio is now installed, but there are no radio stations to be heard yet. To add a radio station, we need to have its streaming link. This is the address from where the audio is streamed. A practical website listing these links is SomaFM [6]. This website lists many free stations of different genres and provides streaming links we can use. To retrieve a usable link from SomaFM, we click on the station we want and click on "Direct stream links". The link we need is beside "Direct server:".

To add a station, we open PuTTY and enter mpc add followed by the station url. For example:

```
mpc add http://ice1.somafm.com/bagel-128-mp3
```

Links ending with '.m3u' will not work. There are many other stations available from other websites that can be added, you're not limited to SomaFM. A url can be tested by entering it in the web browser on your computer. If it's a valid link, a music player will open, playing the radio station.

To list the added radio stations, we can use the command mpc playlist. This lists all added radio stations. To remove a radio station, type mpc del followed by the number in the playlist in PuTTY, for example: mpc del 2 removes the station in the second position.

All set to use

The radio player can be started now by double tapping the icon on the touchscreen and pressing 'Execute'. The interface (**Figure 3**) has been kept simple so that it can be operated with your fingers rather than a stylus (**Figure 2**). These are the available commands:

- 'Play' starts the mpc on the current playlist.
- 'Pause' stops the mpc on the current playlist.
- 'Volume down' reduces volume.
- 'Volume up' increases volume.
- 'Mute' sets the volume to zero.
- 'Refresh' stops the player and starts it again, refreshing the screen.
- 'Exit' closes the radio player interface but keeps the mpc playing.

To conveniently shut down the Raspberry Pi, double tap the shutDown icon and press 'Execute'. Happy listening! ◀

(160043)

Web Links

- [1] www.raspberrypi.org/downloads/raspbian
- [2] <https://sourceforge.net/projects/win32diskimager>
- [3] www.chiark.greenend.org.uk/~sgtatham/putty/download.html
- [4] <https://winscp.net/eng/download.php>
- [5] www.elektormagazine.com/160043
- [6] <http://somafm.com>

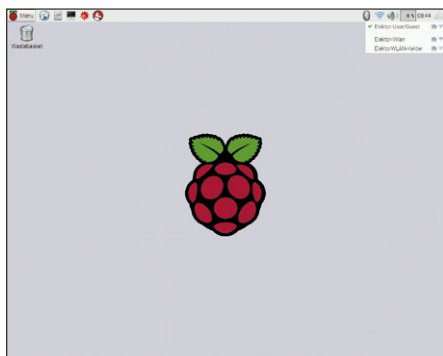


Figure 1. Here we can configure the RPi network settings.

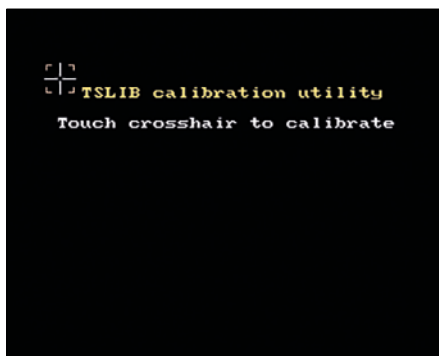
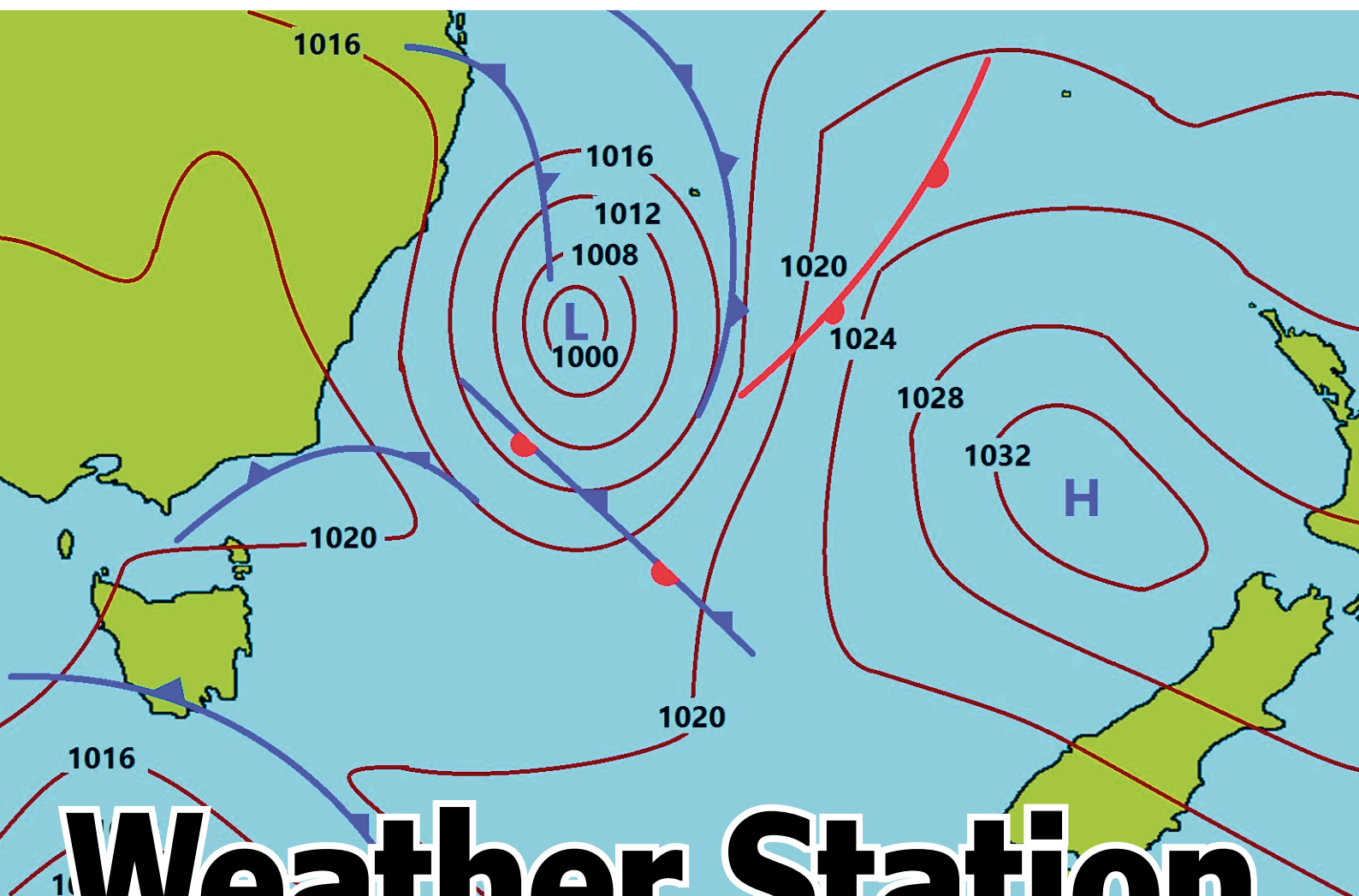


Figure 2. Calibration the touchscreen before operation...



Figure 3. The interface can be operated by finger — no stylus needed.



Weather Station

With no moving parts

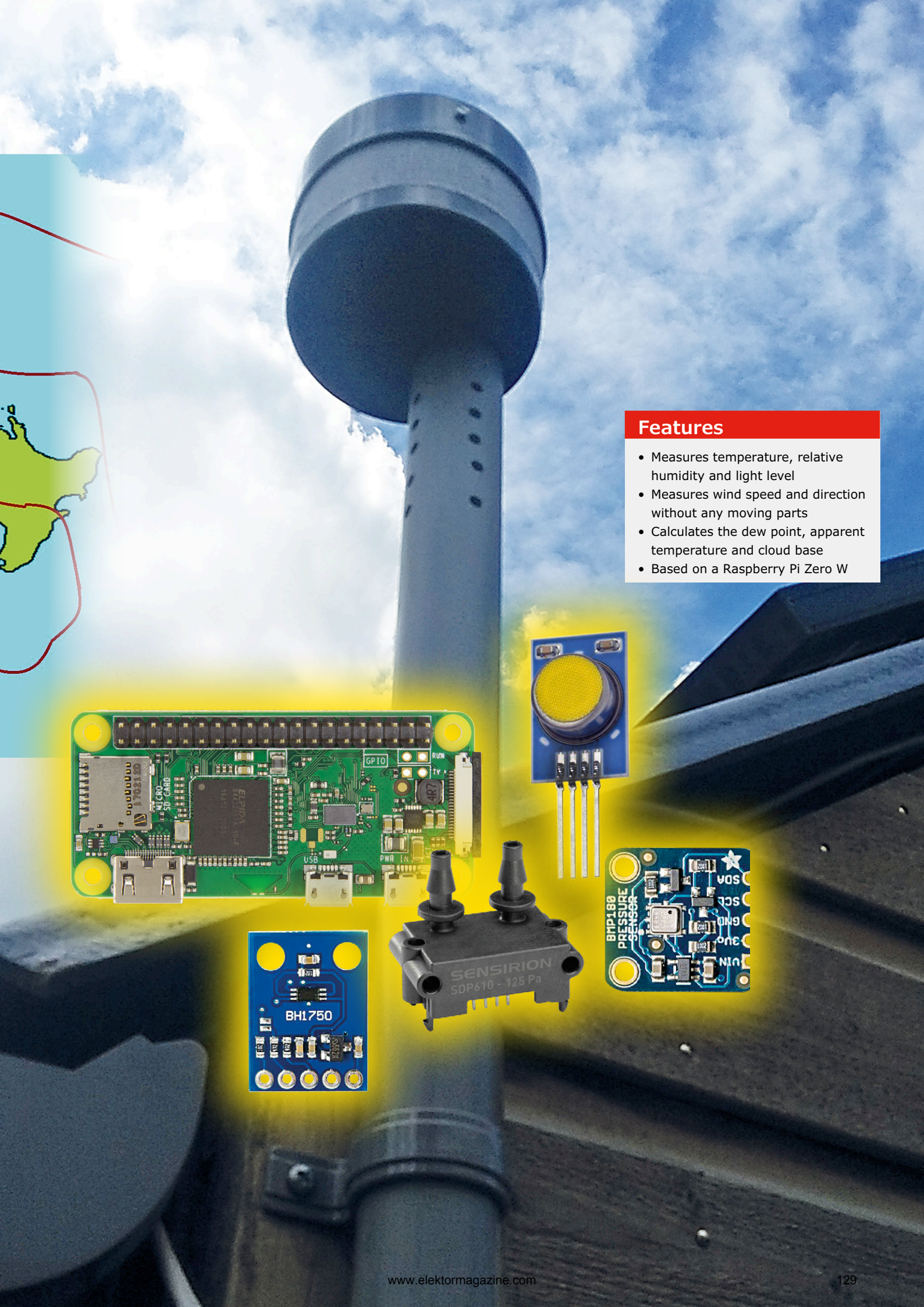
By **Zeno Otten** (Netherlands)

Many aspects of our daily lives are determined by the weather: whether or not you take an umbrella with you, whether you walk to work or take the car, whether you close the window or leave it open, whether you stay in bed or get up, and so on. Even though a glance out the window can tell you a lot about the weather at the moment, you need data to help you estimate what it will be later – and that's where the weather station described here comes in handy.

If cost is the only consideration, building your own weather station nowadays is hardly worth the effort. Commercial instruments made in Asia are so inexpensive (and in many cases remarkably accurate, despite the low cost) that an

electronics hobbyist cannot compete with them. However, the ready-made devices have some shortcomings that justify a DIY project, and in any case building something yourself is a lot more fun than buying it.

For example, most instruments use an anemometer – usually a rotor with cups to catch the wind – to measure wind speed and a wind vane to measure wind direction. These are mechanical devices that wear out over the course of time



Features

- Measures temperature, relative humidity and light level
- Measures wind speed and direction without any moving parts
- Calculates the dew point, apparent temperature and cloud base
- Based on a Raspberry Pi Zero W

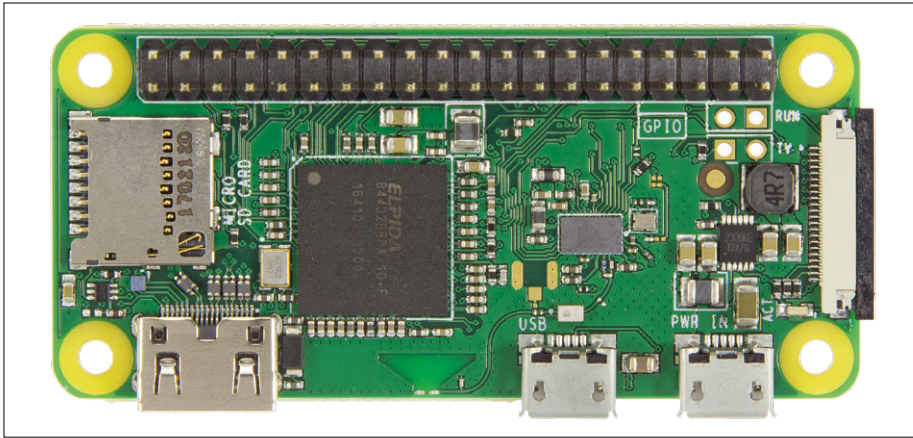


Figure 1. The Raspberry Pi Zero W.

(and depending on the quality, that can be a fairly short time) and need to be replaced. That's when the problems arise, because these mechanical components are usually not available separately. Another thing is that communication

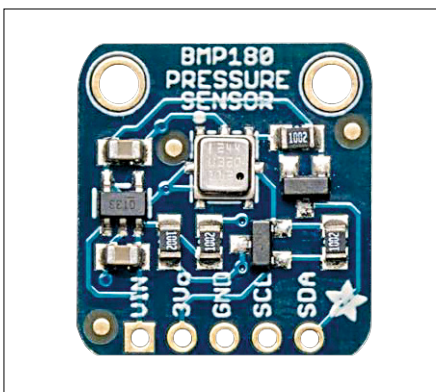


Figure 2. The BMP180 barometric air pressure sensor.



Figure 3. The HYT221 temperature and relative humidity sensor.

between the sensors mounted outdoors and the base station indoors is almost always wireless, in many cases using a transmitter/receiver pair operating at 433 MHz with a non-standard protocol. That makes it virtually impossible to make any changes or extensions yourself. Finally, most weather stations do not show any derived quantities, such as the dew point or the apparent temperature.

The challenge

For the author, all this was sufficient reason to tackle the challenge of building a weather station. The basic criteria for this were:

- A design with no moving parts
- Measurement of air pressure, relative humidity, temperature, light level, wind speed and wind direction
- Calculation of derived quantities: dew point, apparent temperature and cloud base
- All sensors connected over the I²C bus
- All sensors powered from 3.3 V
- Measurement data stored in a database
- Data presentation via a Web interface
- Communication over LAN or the Internet via WLAN
- Support for expansion (for example, with a rain gauge)
- Understandable and extensible software

The approach

A quick review of the above list of basic criteria makes the choice of the basic platform for the project fairly easy: the Raspberry Pi Zero W is perfect for this

task. This compact single-board computer has sufficient I/O and processing power, and it has everything on board necessary for easy connection to the LAN or the Internet, either wired or wireless. When you also consider that this little marvel is smaller than a credit card (**Figure 1**) and only costs about 10 euros/pounds/dollars, the choice is absolutely clear. The RPi Zero W was described in detail in the September/October 2017 issue of Elektor Magazine [1], and for anyone who does not have previous experience with the Raspberry Pi or the RPi Zero, an excellent starter kit is available in the Elektor Store [12].

The various sensors are connected to the RPi Zero via the I²C bus, along with the lines for supply voltage and ground. Originally developed by Philips, the I²C bus is very popular for communication between microcontrollers, ICs or even entire modules. The I²C bus uses just two connection lines: SDA (serial data) for the data and SCL (serial clock) for the clock signal. Of course, additional lines are necessary for the supply voltage and ground, making a total of four lines. I²C employs a master/slave protocol in which the master (in this case the RPi) addresses a slave device (one of the sensors) and asks it to send the available data, such as a temperature reading. Each slave (sensor) has a unique address, which in many cases can be set (or partially set) by the user.

The bus uses pull-up resistors, so the SDA and SCL lines are at the high level in the quiescent state. These pull-up resistors have values in the range of 4.7 kΩ to 10 kΩ. In many cases these resistors are integrated into the sensors, so you have to be careful to ensure that the net resistance is within the stated range.

The sensors

With the right sensors, it's easy to measure air pressure, temperature, relative humidity and light level. It's a bit more difficult with wind speed and direction, but we can deal with this later.

Air pressure

For air pressure measurement, the author used a BMP180 [2] — a barometric sensor that can measure the absolute air pressure over a range of 300 to 1100 hPa (mbar). As you know if you have ever climbed a mountain, the air pressure depends on the altitude, so this sensor can also be used as an altimeter.

The sensor is mounted on a breakout board (**Figure 2**) and supplies air pressure data with a resolution of 12 bits, corresponding to an air pressure increment of 0.03 hPa. The supply voltage is 3.3 V, and the pull-up resistors for the I²C bus are mounted on the board. The sensor is connected to the RPi via the SCA and SCL lines.

Note: The BMP180 may be difficult to obtain. You can use the BME280 as an alternative, although some changes to the weather station firmware may be necessary. The BME280 is available in the Elektor Store [3].

Temperature and relative humidity

A robust HYT221 sensor [4] is used for this purpose (**Figure 3**). This sensor is ideal for outdoor use and can withstand condensation. The measuring range for relative humidity is 0 to 100% with an accuracy of ±1.8%, and the measuring range for temperature is -40 to +120°C with an accuracy of ±0.2°C.

This sensor also has an I²C interface with integrated pull-up resistors. The rated supply voltage range is 2.7 to 5.5 V.

Light level

Light level measurement can be used to detect darkness, overcast conditions or sunshine and convert them to numerical values. In the prototype the author used a BH1750 sensor [5], which features a large measuring range of 1 to 65,535 lux. This sensor is also mounted on a breakout board (**Figure 4**) with an I²C port, including pull-up resistors. The rated supply voltage range is 3.3 to 5 V. Typical light level values are 0.001 to 0.02 lux (night), 500 to 25,000 lux (overcast), and 50,000 to 100,000 lux (sunlight).

Wind speed and direction

The working principle for measuring these quantities with two differential pressure sensors is briefly described in the inset.

Table 1: Wind speeds	
Gale	>17 m/s
High wind	14–17 m/s
Strong breeze	10–14 m/s
Fresh breeze	8–10 m/s
Light to moderate breeze	1.5–8 m/s
Light air	0.3–1.5 m/s
Calm	0.0–0.2 m/s

The measuring range is the most important consideration for choosing the sensors. The wind speeds that can be expected in the Netherlands are listed in **Table 1**. If we assume an 'average' storm with a wind speed of 20 m/s, the formula mentioned in the inset gives a differential pressure of 260 Pa with an air density (ρ) of 1.3 kg/m³. This means that in theory you need a sensor with at least this measuring range. Fortunately, storms with such high wind speeds are not all that common here, so the author chose the Sensirion SDP600/610-125Pa [6], which has a range of 0 to 125 Pa (**Figure 6**). It has better resolution with smaller differential pressures, corre-



Figure 4. The BH1750 light level sensor.

Intermezzo: "The wind blows where it wishes"

Most wind gauges for household use consist of an anemometer (a rotor with cups to catch the wind) and a wind vane, which are moving parts. However, in the present era with the focus on software and microcontrollers, many electronics hobbyists have a strong aversion to messing about with mechanical construction, so we have to find something that suits them. If you think about what keeps an airplane in the air, the solution is fairly obvious. An airplane wing is more curved on the top than on the bottom. As a result, the air flowing over the top surface of the wing has to travel further than the air flowing over the bottom surface, and that produces a pressure difference — the air pressure above the wing is lower than the air pressure below the wing. The net result is an upward force, or lift, that keeps the airplane in the air. This phenomenon was first described in the 18th century by the physicist Daniel Bernoulli.

Incidentally, the pitot tubes [11] used to measure aircraft speed work on the same principle. The pressure of a moving fluid (which is air in the case of aircraft, but the same thing applies to other fluids) is measured simultaneously parallel and perpendicular to the direction of motion. From this pressure difference, the speed v of the fluid (in m/s) can be calculated using the following equation formulated by Bernoulli:

$$v = \sqrt{\frac{2\Delta p}{\rho}}$$

Here Δp is the pressure difference between the measurements perpendicular and parallel to the direction of motion, and ρ is the density of the fluid (which in the case of air is 1.3 kg/m³).

For a static wind gauge (in other words, without any moving parts) you can use a vertically oriented cylinder, with the air flowing around it as shown schematically in **Figure 5**. Two differential pressure sensors are fitted in this cylinder, giving you four measuring points (ram pressure points) P1 – P4. The differential pressures between P1/P2 and between P3/P4 are measured.

The wind direction is determined by calculating how often the highest pressure is measured at each of these points during a given period and then averaging the result to obtain the actual wind direction.

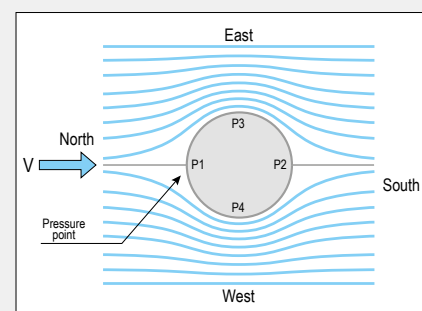


Figure 5. Air flow around a cylinder.



Figure 6. The SDP600/610-125Pa differential pressure sensor.

sponding to lower wind speeds. The sensor operates with a supply voltage of 3.3 V, and the data is again read out over the I²C bus. However, there is

a bit of a problem here. With most I²C sensors the address can easily be configured externally. However, with this differential pressure sensor the address is set to a fixed value of 0x40 in the factory. You need two of these sensors, which means you would have to use two different I²C buses. That's not very convenient.

Fortunately, the I²C address of the sensor is stored in an internal EEPROM. On request of the author, the manufacturer provided the information necessary to overwrite this address with a different value (at the cost of voiding the warranty). In this case the address of the second sensor was set to 0x21. This information, which you can also request from the manufacturer if necessary, is incorporated in a routine forming part of the weather station software (*weerstation.py*) included in the free download for

this project [7]. This routine only needs to be run once, with just one of the two sensors connected.

The Sensirion differential pressure sensor does not have integrated pull-up resistors for the I²C bus, so they were added separately.

Derived quantities

The measured quantities mentioned above – temperature, relative humidity, wind speed and direction, and light level – are nothing special; virtually any ready-made weather station can also show them. Things get a lot more interesting when you calculate some quantities that are less ordinary.

Dew point

The dew point is the temperature at which the air is saturated with water vapour. The dew point is an important parameter in meteorology – it is used to calculate the cloud base, the chance of fog, and the chance of ice formation (which is important for aviation). A high dew point value also means that there is a greater chance of thunderstorms. The dew point can be calculated from the measured relative humidity and the measured air temperature, as described in [8] and other sources.

Apparent temperature

There are two kinds of temperature: the actual temperature, which is what you measure with a thermometer or similar device, and the apparent temperature, which is what you feel. There can be a considerable difference between the two. For example, a measured temperature of -3°C can feel like -10°C (or even colder) when there is a strong wind and high relative humidity.

As you may have guessed, the apparent temperature is calculated from the measured wind speed and the measured relative humidity. There are various ways to do this; the author opted for the method used by the meteorological service of the Australian government [9].

Cloud base

Finally there is the cloud base, which is also a significant parameter: the lower the cloud base, the higher the chance of precipitation (as every farmer used to know). As described in [10], the cloud base can be calculated from the air temperature and the dew point.

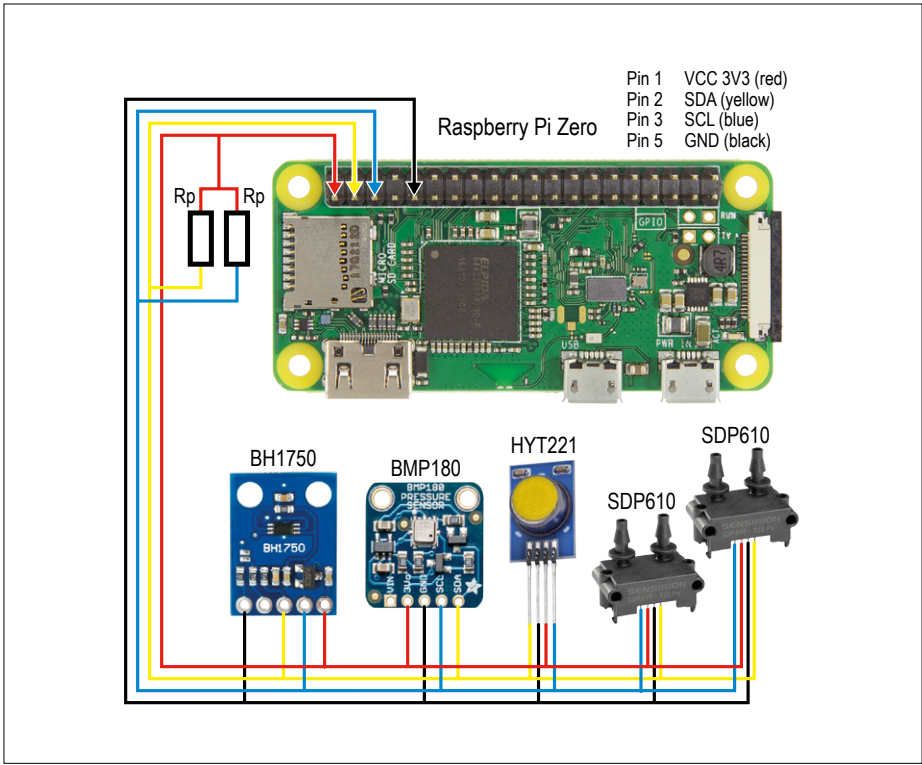


Figure 7. The sensors are connected to the RPi Zero W via a 4-wire bus.

Table 2: I ² C components and addresses		
Component	Address (hex)	Function
BMP180	0x77	Barometric air pressure sensor
HYT221	0x28	Temperature and relative humidity sensor
SDP01	0x40	Differential pressure sensor 1
SDP02	0x21	Differential pressure sensor 2
BH1750	0x23	Light level sensor

The electronics

The electronics of the weather station are fairly simple, as long as you regard the RPi and the various sensors as black boxes. The layout shown in **Figure 7** consists of the RPi Zero W, the sensors, and the four-wire bus (two wires for I²C, two for the supply voltage and ground) that links everything together.

The power source for the RPi is not shown; the author used a USB charger/adaptor connected to the power connector of the RPi board (at the bottom right on the photo in Figure 1). The resistors R_p shown in Figure 7 are the pull-up resistors for the Sensirion differential pressure sensors.

Figure 8 shows the author's prototype fitted in a small plastic enclosure. The USB power cable for the RPi exits through a strain relief, as does the four-wire UTP cable for the wind gauge. In the prototype this cable is 5 m long, which does not cause any problems for data transmission over the bus, and it is also protected by a strain relief. This allows the wind gauge to be mounted well above the roof, with the rest of the electronics located indoors. The addresses used for the devices on the I²C bus are listed in **Table 2**.

Mechanical construction

Building the wind gauge requires a bit of tinkering. For this the author used a piece of PVC pipe with a diameter of 10 cm. He drilled four holes in the pipe at right angles to each other (for the four compass directions) and glued a piece of plastic tubing (of the sort used for aquarium lines, for example) into each of these holes. These tubes lead to ports P1 to P4 of the two differential pressure sensors.

Figure 9 shows the wind gauge in its natural element. To protect the electronic components, the top and bottom of the wind gauge must be properly sealed to prevent water entry. Screening should also be placed in front of the tube openings to prevent insects from using them as hiding places. If you like tinkering, this is your golden opportunity.

The firmware

The firmware for the weather station consists of two parts. The first part is the *weerstation.py* software. It is written in Python, which is a natural choice because the Python 2.6 programming environment is a standard part of the Jessie Linux distribution for the RPi Zero.

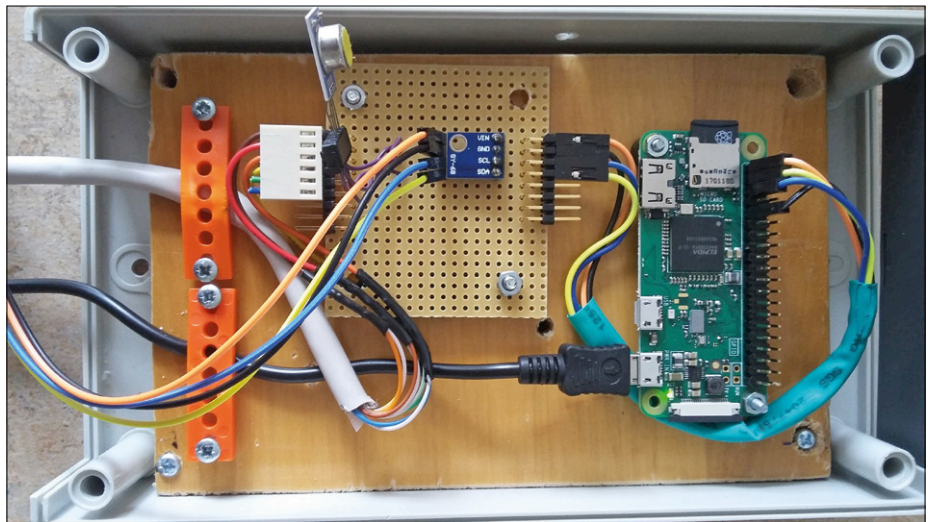


Figure 8. The author's prototype.



Figure 9. The wind gauge in its natural element, well above the roof.

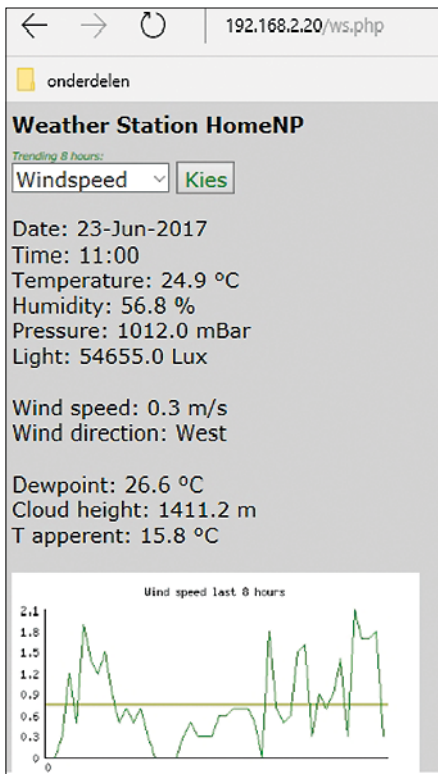


Figure 10. Visualisation of the measured and derived quantities on the web page.

This software reads the sensors every 10 minutes, performs the necessary calculations, and stores the results in a MySQL database.

The second part is a combination of HTML and PHP script. This script runs on the Apache webserver of the RPi Zero. It reads the data from the database and configures it for presentation on a web page (see **Figure 10** for an example). There's no point in showing a complete listing of the firmware here, since it can be downloaded with a couple of mouse clicks. The code is clearly structured and generally self-explanatory. For your interest, the following libraries are used in the *weerstation.py* software:

```
import MySQLdb as mdb
import sys
import time
import os
import smbus
from ctypes import c_short
import math
```

The web page *ws.php* (see the download) is published by the Apache webserver installed on the RPi Zero. The RPi Zero is connected to the author's LAN, so the

data from the weather station can be viewed on every device within range of the network. If desired, the web page can also be published on the Internet via the LAN router.

Conclusion

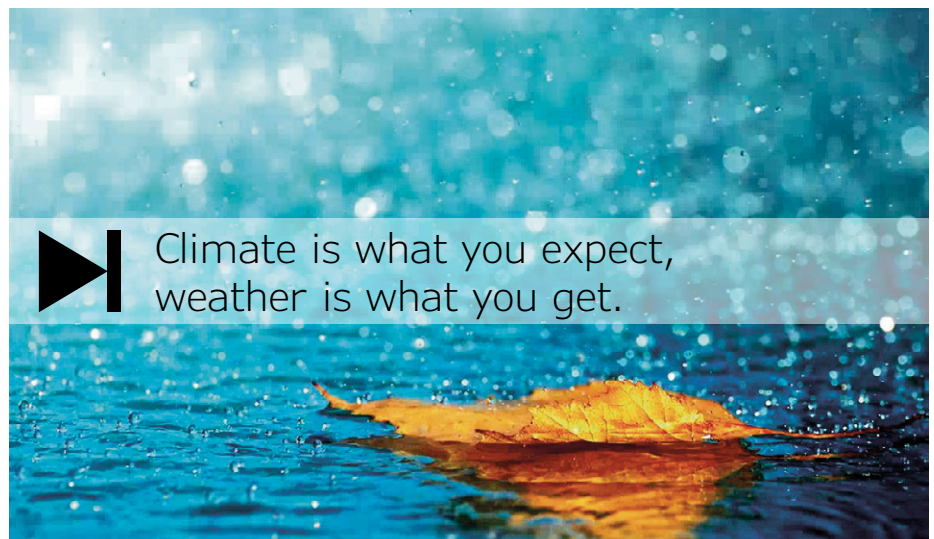
This article is not intended as a detailed DIY project, but instead as a stimulus for interested readers to build their own weather stations. As always, we welcome your comments and suggestions, and we would like to hear from readers who have built weather stations. ◀

(160566-I)

FROM THE STORE

→ **SKU18328**

Raspberry Pi Zero W Starter Kit



Web Links

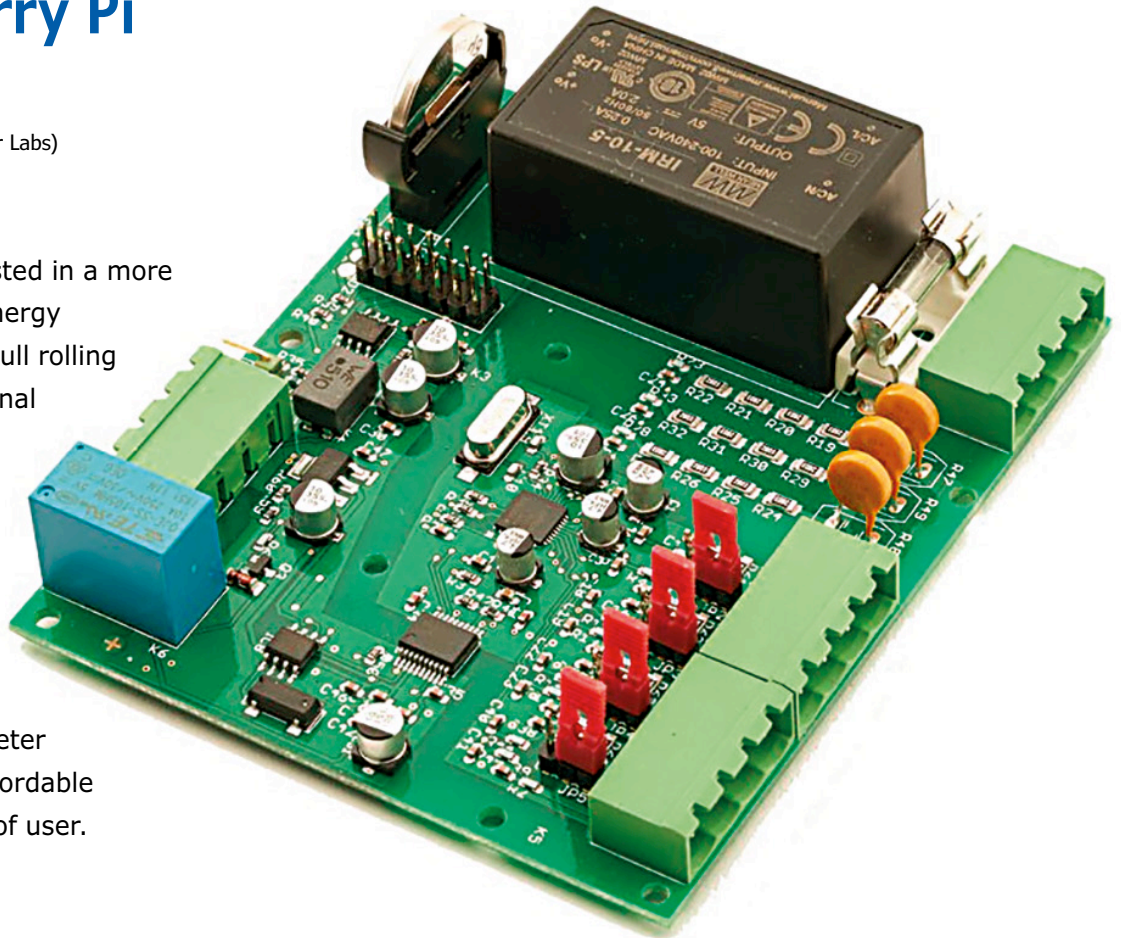
- [1] www.elektormagazine.com/160451
- [2] <https://learn.sparkfun.com/tutorials/bmp180-barometric-pressure-sensor-hookup->
- [3] www.elektor.com/bme280-mouser-intel-i2c-version-160109-91
- [4] www.farnell.com/datasheets/1643979.pdf
- [5] <http://domoticx.com/esp8266-wifi-lichtintensiteit-sensor-bh1750-gy-302-nodemcu/>
- [6] www.sensirion.com/en/environmental-sensors/humidity-sensors/digital-differential-pressure-sensors-without-zero-point-drift/
- [7] www.elektormagazine.com/160566
- [8] https://en.wikipedia.org/wiki/Dew_point
- [9] www.bom.gov.au/info/thermal_stress/?cid=003bl08
- [10] https://en.wikipedia.org/wiki/Cloud_base
- [11] https://en.wikipedia.org/wiki/Pitot_tube
- [12] www.elektor.com/raspberry-pi-zero-w-starter-kit

SmartPi Reviewed

Smart energy meter extension for Raspberry Pi

By **Clemens Valens** (Elektor Labs)

Many people are interested in a more detailed view of their energy consumption than the dull rolling numbers on the traditional domestic watt-hour meter. Likewise, energy suppliers and network operators need better insight in “consumption patterns”. The SmartPi “bidirectional” smart meter is an interesting and affordable solution for both types of user.



After a successful crowdfunding campaign back in 2016, the SmartPi was recently updated to revision 2.0. Compared to the previous version the device has become more practical with all the commonly needed connectors at the same side and a built-in power supply. All connectors now have screw terminals, improving flexibility, and an RS-485 interface is available for connecting to Modbus devices. The RPi's Real-time Clock (RTC) now boasts a back-up battery, and a relay port has been added. Finally, the new enclosure has a DIN-rail-compatible clip.

Specifications

The SmartPi measures current up to 100 A on four channels (three Phases plus Neutral), voltage (up to 390 V_{rms}) on three channels, and frequency; it calculates several types of power (active, reactive, apparent) and energy consumption. Besides measuring power consumption, it also measures energy and power generation, which is why the manufacturer rightly calls it a bidirectional smart meter.

Two types of current sensors can be used with the SmartPi: 50 mA and 1 A secondary-current devices, allowing currents of up to 700 A to be measured (this requires changing jumper settings inside the SmartPi).

The SmartPi add-on board can be bought separately, but also as a configurable kit or even fully assembled. The assembled

version (reviewed here) is built on a Raspberry Pi 3 model B. The software, written in Go, is open source and available from GitHub [1]. It includes all the necessary drivers, a web server with webpage, and a REST interface for integrating the SmartPi into self-made applications and apps.

Where is the manual?

As is — unfortunately — so common these days, the manual that ships with the SmartPi is very succinct, more or less forcing you to go online. Unfortunately, things are not much better here. Note that the website is in German, but, at least for the SmartPi, webpages in English exist too. To get there you must either type in the URL yourself [2] or click the video (!) link in the shop. Actually, the Kickstarter page [3] is a much better source for information, nor forgetting the forum [4]. Following the installation link does not take you to a page explaining how to set up and install the device in your home, but instead provides detailed instructions on installing the SmartPi SD card image on a Raspberry Pi. How to hook it up to your domestic AC network is left as an exercise for the user. The ‘Use’ link leads to a page with a connection drawing, but nothing is said on how to wire the device to the network. The page even suggests leaving the installation to a qualified electrician (**Figure 1**). Sound advice of course, but we would have



Figure 1. The SmartPi installed by a qualified technician (I am — really).



Figure 2. The pinout of the SmartPi voltage and current connectors.

liked to see a bit more detail here. As another point of concern, the page has not been updated and still refers to the SmartPi 1.0 that required an external power supply.

Connecting the SmartPi to a data network

The SmartPi exposes all RPi connectors, including the Ethernet connector. After cabling the SmartPi to my router, it showed up in its DHCP client list; point a browser to port 1080 at the address shown by the router and the SmartPi user interface appears. Unfortunately my router is located far away from the electrical switchboard, so Wi-Fi would be handy. How to do this or even if this is possible is not mentioned anywhere. The RPi3

has on-board Wi-Fi, so I decided to try to configure it over SSH. The default username is 'pi' and the default password is either 'smart4pi' or 'raspberry'. In my case it was 'smart4pi'.

SmartPi & Wi-Fi

Connect to the SmartPi over SSH with PuTTY (or some other SSH client) and edit the file `wpa_supplicant.conf` using the command:

```
sudo nano /etc/wpa_supplicant/wpa_supplicant.conf
```

Add the following at the end of the file (replacing `mySSID` & `myPassPhrase` by your SSID & your pass phrase):

```
network={
    ssid="mySSID"
    psk="myPassPhrase"
    key_mgmt=WPA-PSK
}
```

Press Ctrl-O followed by the Enter key to save the file, press Ctrl-X to quit the editor. Disconnect the Ethernet cable and reboot the SmartPi. Now it's accessible over Wi-Fi. Why on earth didn't they document this?

Change the SmartPi password

Now that we are busy setting up the RPi, we might as well change the password. To do so is highly recommended by the SmartPi people, but, again, how to do it isn't explained. Here's the clue. Launch the SSH client (again) and connect to the SmartPi, then launch the Raspberry Pi configuration tool with:

```
sudo raspi-config
```

Select `Change User Password` and enter a new password.

Connecting AC power to the SmartPi

Once more, the documentation is not very helpful. Luckily the SmartPi's inputs correspond to the phases (P) and neutral (N) printed on a label stuck at the backside of the device (but not on mine, **Figure 2**).

The evaluation SmartPi came with three pre-wired current sensors. I live in France. My home has a single-phase electrical network, divided into two groups, hence I put a current sensor on each group. The third sensor was clamped on the wire going to the water heater (in the second group). Note that the sensors have an orientation, indicated with arrows. According to the manufacturer they should point in the direction of the power consumer (the building, the motor, etc.), but I found that I had to reverse them to get positive readings.

I connected the three voltage inputs together and wired them to a free fuse (I happened to have one).

Once you have managed to get positive currents and powers on all (three) channels, you can start trying to interpret the data.

Dashboard

What follows applies to software version 0.6.4.

The web interface (by default at port 1080, but it's configurable) showed three currents, three voltages and three frequencies (**Figure 3**).

The web interface has a settings page with a 'Measurement' tab where you can invert the current directions, select the

FROM THE STORE

- **SKU 18165:** SmartPi 2.0 add-on board including three 100 A current sensors
- **SKU 18166:** Case for SmartPi 2.0
- **SKU 18167:** Current sensor, 100 A

type of current sensor, and select the voltage channels to be monitored. Here you can also set the target line frequency and voltage. I had to reboot the SmartPi every time I changed a setting. Deactivating the Neutral current channel was not taken into account for some reason.

It seems that when the currents are small the measurements go haywire and the Cos Phi and frequency numbers start jumping around (I observed values from 0 to more than 50 GHz, see **Figure 4**). It should be easy to make the software handle this a little better.

When the SmartPi has collected some days' worth of data, it is time to look at its graphs and try to understand just what they show. It's also possible to export the data as a CSV file (you can set the decimal point character on the Expert tab of the settings page, but not the field separator...) and process it yourself in a spreadsheet.

The graphs are nice and scale automatically, which sometimes is nice, but sometimes it isn't, especially when there has been a large peak in the recent past. The graphs can display current, voltage, power, and energy production and consumption — just click on the buttons to activate or deactivate what you want to see. Hovering the mouse over the graph shows a cursor displaying the values and timestamp at the position of the mouse. Histograms are used for power consumption and generation over the last week.

Options and interfaces

As mentioned at the beginning the SmartPi exposes a REST API, allowing other applications to talk to it. Browsing the Settings section on the dashboard reveals more communication and interfacing options that show the intentions of the developers. Here it is possible to configure an UMTS connection, an FTP server and MQTT client support. Searching through the SmartPi repository at GitHub shows that MQTT is supposed to be working, but nothing about FTP which appears to be intended for automatic uploading of CSV files to an FTP server somewhere. Future extension will certainly also include the RS-485 port (Modbus) and the relay-switched contact.

A smart meter platform for makers

Having spent many hours fiddling with the SmartPi, its hardware and its software, what remains is a kind of mixed feeling. It is a product with a lot of potential that you would like to be a success, but for now (at the time of writing) it is hindered by unripe software and incomplete documentation. The SmartPi is (a Raspberry Pi extension board) for makers.

The software supporting it is full of good intentions, but lacks precision and finesse. New updates are released regularly though — check the forum [4] for the latest version.

The documentation is wildly insufficient. Users without RPi or Linux experience will be lost quickly. This is a pity as it should not be very difficult to add a more complete setup (Wi-Fi configuration, password management, etc.) to the configuration menu, add some Help pages, and add a few paragraphs with clear pictures to the manual. Doing so would make the SmartPi interesting for a much wider audience. On the other hand, if you are looking for a platform on which to build a smart energy consumption and/or production metering system, the SmartPi definitely is a good start. ◀

(160485)

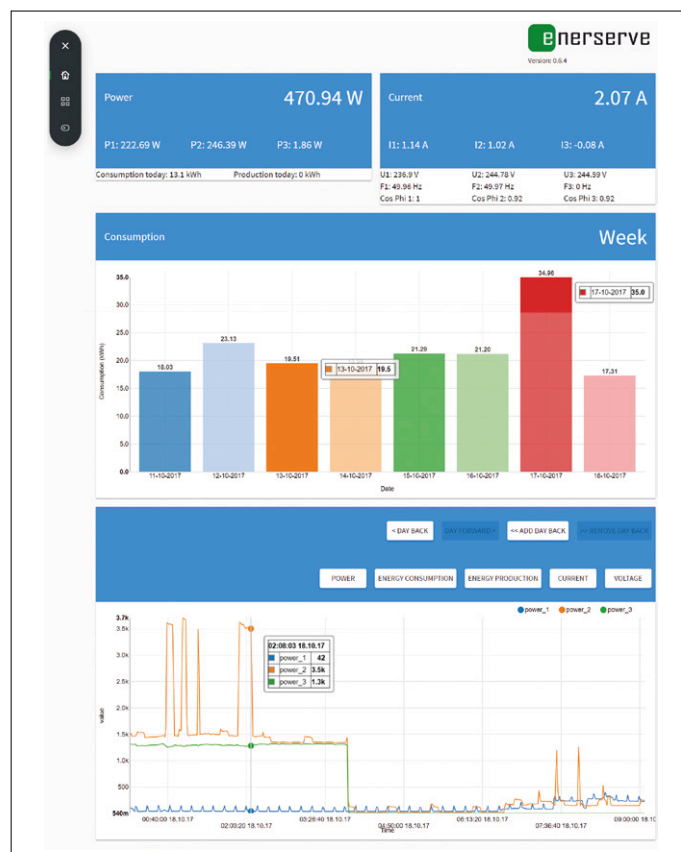


Figure 3. The dashboard shows graphs of past and current consumption (pun intended).

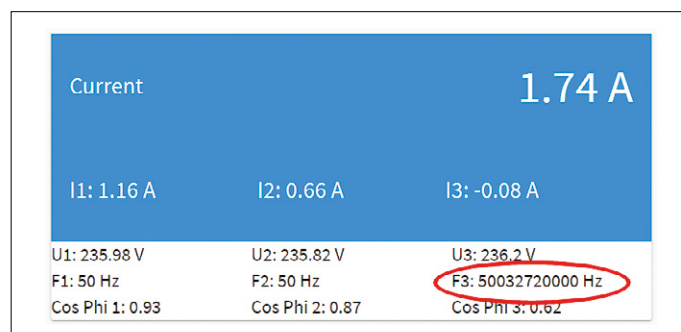


Figure 4. That can't be right. A line frequency of 50 GHz?

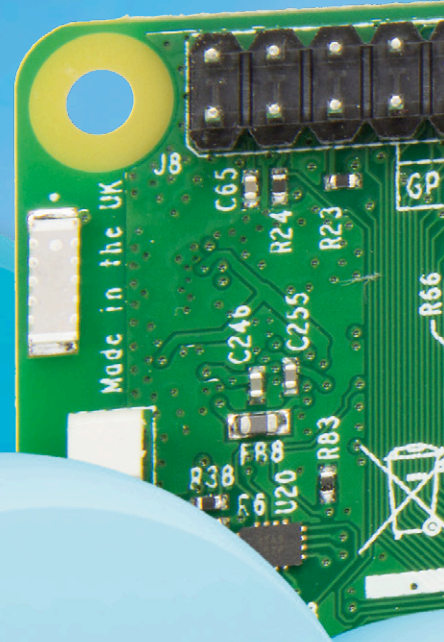
Web Links

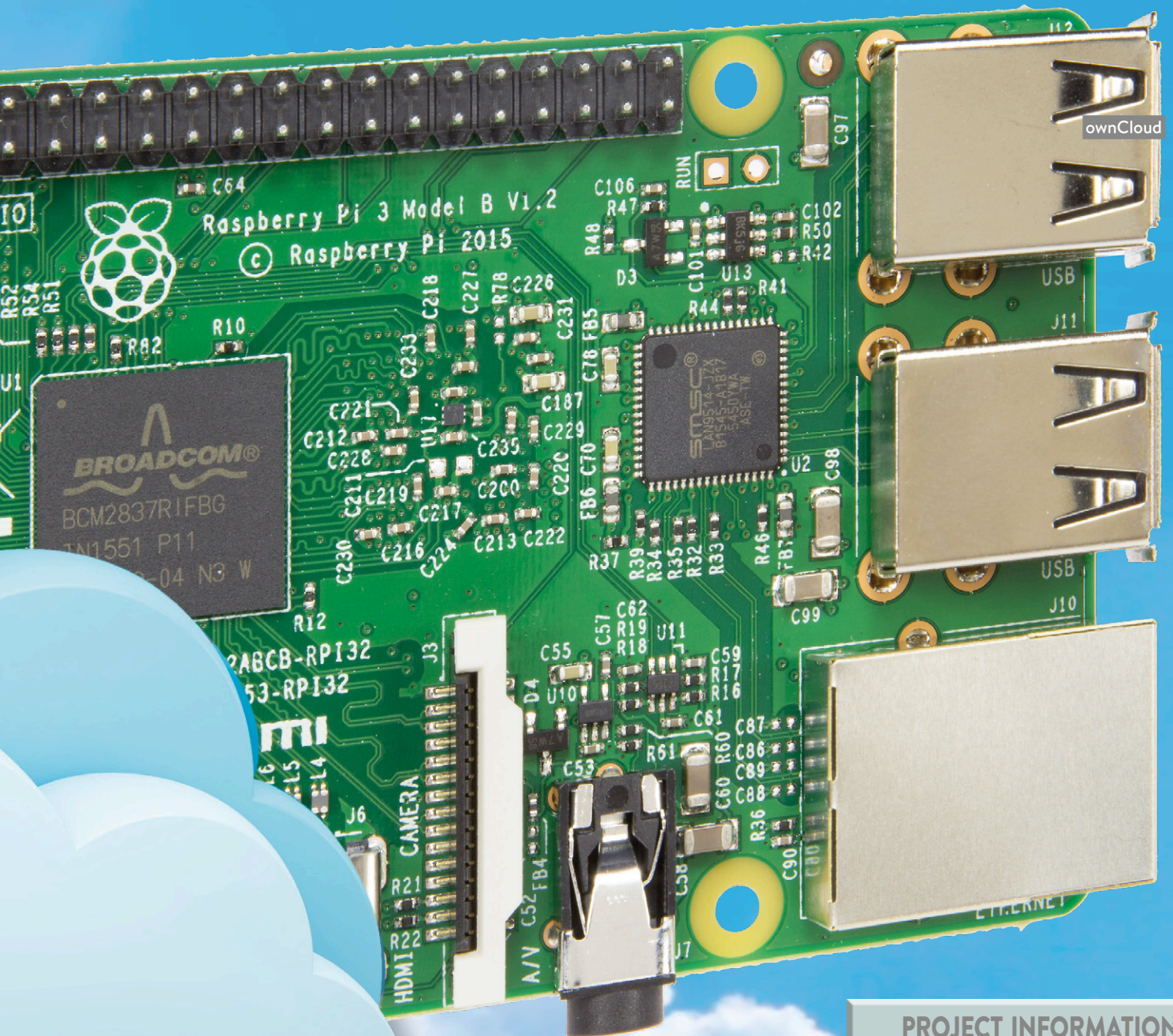
- [1] SmartPi on GitHub: <https://github.com/nDenerserve/SmartPi>
- [2] Homepage: <http://www.emanager.eu/en/products/smartpi>
- [3] On Kickstarter: <https://www.kickstarter.com/projects/1240982104/smartpi-turn-your-raspberry-pi-into-a-smartmeter>
- [4] Forum: <https://forum.enerserve.eu/>
- [5] www.elektormagazine.com/160485

A Cloud in the Shape of a Raspberry

install a cloud in your living room

By **Sébastien Guerreiro de Brito** (France)





Today the cloud is a popular solution to save and share files as well as allow access to your files from any computer anywhere in the world. Because many people are wary of the reliability and security of cloud storage, this article shows you how to create your personal cloud and its power supply board.

PROJECT INFORMATION



Raspberry Pi

Cloud ownCloud

Apache2 HTTPS RPi



entry level

intermediate level

→ expert level



4 hours approx.



Computer,
SMD soldering iron



€150 / £135 / \$185

Characteristics

- Accessible from anywhere in the world
- Share your files, photos and applications
- Use the cloud

How do you create a Cloud?

A cloud is made partly of hardware and partly of software. On the hardware side, I chose the Raspberry Pi platform equipped with a hard disk to be able to store a large number of files. Because hard disks consume a lot of power, I decided to design a power supply capable of correctly powering the Raspberry Pi – hard disk assembly (see the inset and **Figures 1 and 2**).

For the software part, I chose the solution 'ownCloud'. Offered as an alternative to services like DropBox, ownCloud is a

free software to create file storage and distribution service platforms. It offers the following characteristics:

- it's free, with open source code and no limitations;
- administration is done via a web page;
- the administrator can manage the users by their names or their group;
- sending an internet link is sufficient to allow sharing;
- files are encrypted during transfer;
- there is a client to synchronise selected local files under Windows or Linux.
- ...

It's really pretty good and doesn't seem to differ that much from other well-known free systems. On the other hand, it has numerous advantages:

- you know exactly where your files are stored;
- the amount of storage may be substantial, even bigger than existing solutions;
- you may shut down the system whenever you wish.
- Etc.

Preparing the Raspberry Pi

To start, download the Raspbian image on the official Raspberry Pi site [1] and then install it. For this article, we used [2017-11-29-raspbian-stretch-lite](#). The 'Lite' version of the image is sufficient as we are not using a screen with graphical interface. Follow the installation instructions from the official site, or refer to any of the numerous tutorials available on the internet.

Once the image is burned to the microSD card (which is easy with 'Etcher', <https://etcher.io/>), place it in the reader slot of the Raspberry Pi. Connect a screen and keyboard, then the power.

Log in to the Raspberry Pi. I should remind you that the default login is 'pi' and the default password is 'raspberrypi'. The start the configuration of the board by typing the command:

```
sudo raspi-config
```

From the configuration menu, set the keyboard according to your language (localisation). I would advise to activate the SSH service. This will be useful for remote interventions on the board. For

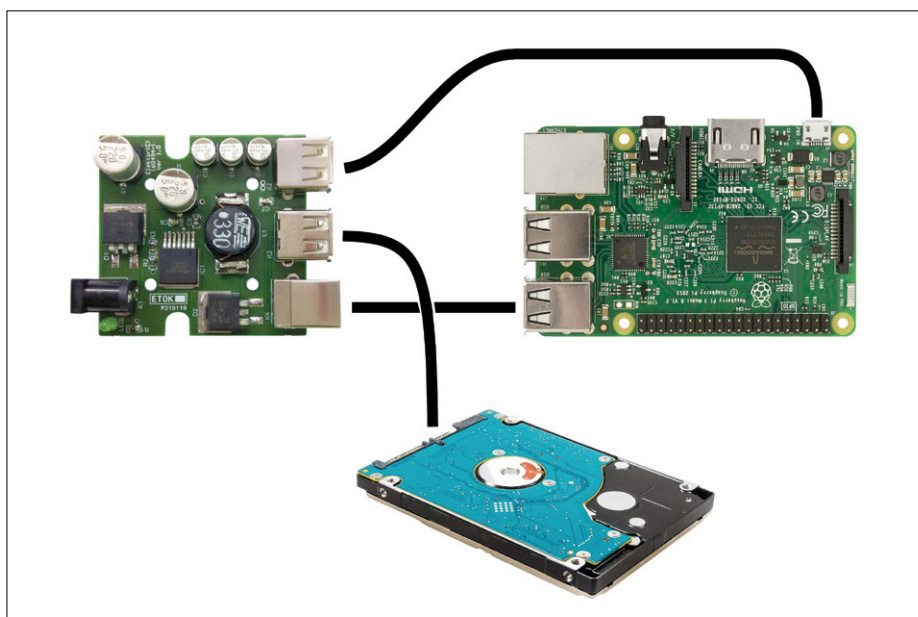


Figure 1. Here's how to connect a SATA hard disk powered only by USB.

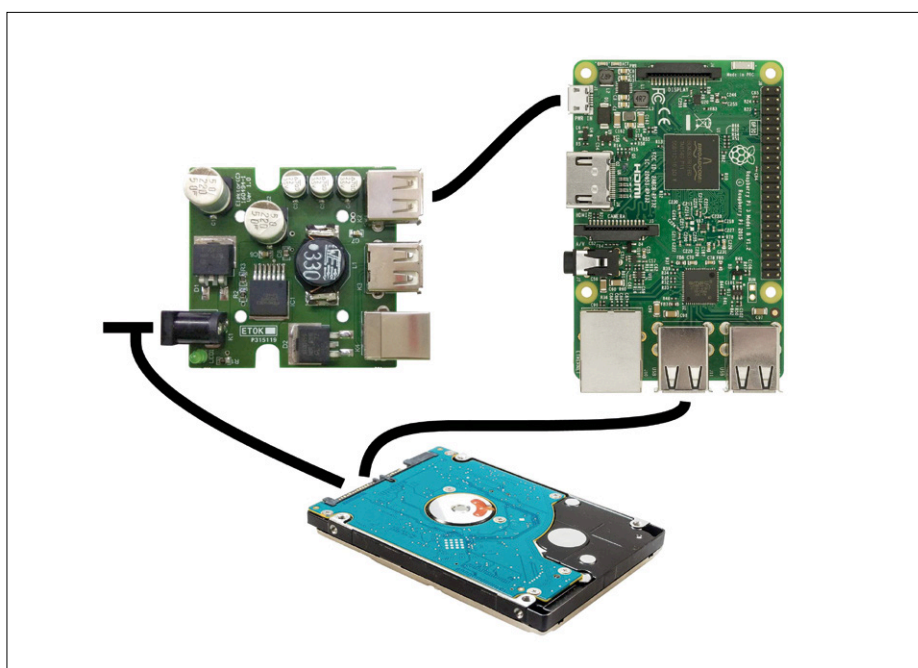


Figure 2. An IDE hard disk for which the USB Converter has its own power can be connected as well.

It's also a good idea to give a fixed IP address to the Raspberry Pi:

```
sudo nano /etc/network/interfaces
```

Next, add the lines:

```
auto eth0
iface eth0 inet static
address 192.168.yyy.xxx  <=
    Desired IP address
netmask 255.255.255.0
gateway 192.168.yyy.zzz  <= IP
    address of your router
```

Press Ctrl-O then enter to save the file, Ctrl-X to quit the editor. You could instead, if preferred, configure your router to always provide the same address to the IP au Raspberry Pi.

Configuration of the storage hard disk

Let's start by creating the folder on which the hard disk will be mounted:

```
sudo mkdir /mnt/usb
```

Then change the rights of this folder and mount the disk which, in our case, is called sda1. Its file system is `ext4`:

```
sudo chmod -R a+w /mnt/usb
sudo mount /dev/sda1 /mnt/usb/
```

Now we pass to the automatic mounting of the external hard disk on startup of the Raspberry Pi. For this we have to find out the PARTUUID of the disk:

```
sudo blkid
```

Look for your hard disk in the list and note its PARTUUID, then open the file `fstab`:

```
sudo nano /etc/fstab
```

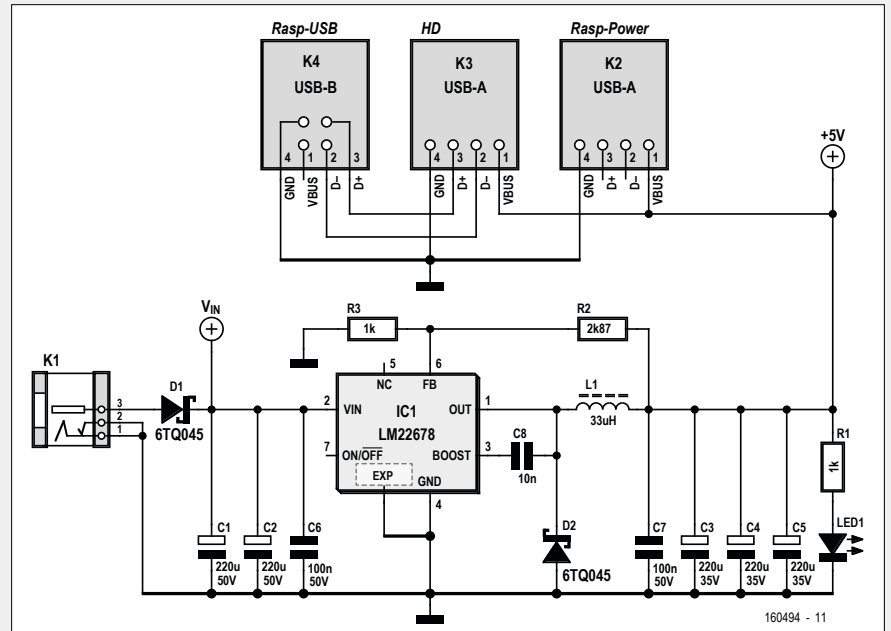
Add this line to the end of the file:

```
PARTUUID=6f20736b-01 /mnt/usb ext4
defaults 0 2
```

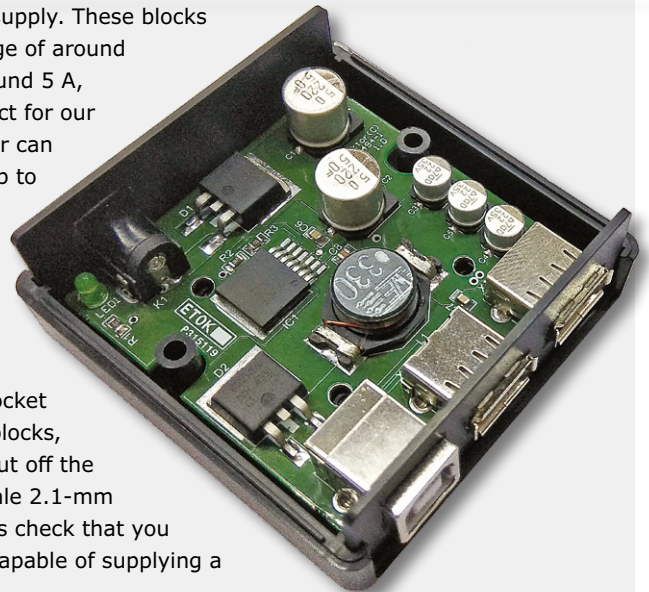
Here '6f20736b-01' corresponds to my hard disk, replace this value with the identifier of your hard disk. Press Ctrl-O then Enter to save the file, Ctrl-X to quit the editor. Take care not to make

The power supply board

The circuit diagram of the power supply is based on an LM22678 switching regulator. It is capable of supplying a current of 5 A maximum. The more hungry SATA hard disks can consume up to 2 A at 5 V. If you consider that the Raspberry Pi nano-computer consumes a maximum of 1 A at 5 V, our regulator has definitely got what it takes.



To power our cloud, the idea is to use an old ‘brick’ laptop computer power supply. These blocks generally deliver a voltage of around 19 V and can supply around 5 A, which makes them perfect for our application. Our regulator can support input voltages up to 42 V, so this is ideal. To be compatible with the maximum number of laptop power supply blocks, our power supply card is equipped with a 2.1-mm DC connector socket (barrel jack). For some blocks, it may be necessary to cut off the original plug and fit a male 2.1-mm DC connector. But always check that you choose a laptop supply capable of supplying a current of 5 A.



Diode D1 protects the supply against wrong polarity of the input voltage.

Regulator IC1 exists in two versions of output voltage: adjustable or fixed at 5 V. I used the adjustable version, because that's what I had on hand :-). The output voltage is set by the ratio of the values of the resistors R2 and R3.

$$V_{out} = 1.285 \times (R2/R3 + 1)$$

With $R_2 = 2.87 \text{ k}\Omega$ and $R_1 = 1 \text{ k}\Omega$, the value of V_{out} is 4.97 V. If a fixed 5-V regulator is used, don't fit R_3 and make $R_2 = 0 \text{ }\Omega$ (or a link).

The two large electrolytic capacitors at the input serve to provide the extra current needed by the SATA hard disk at startup.

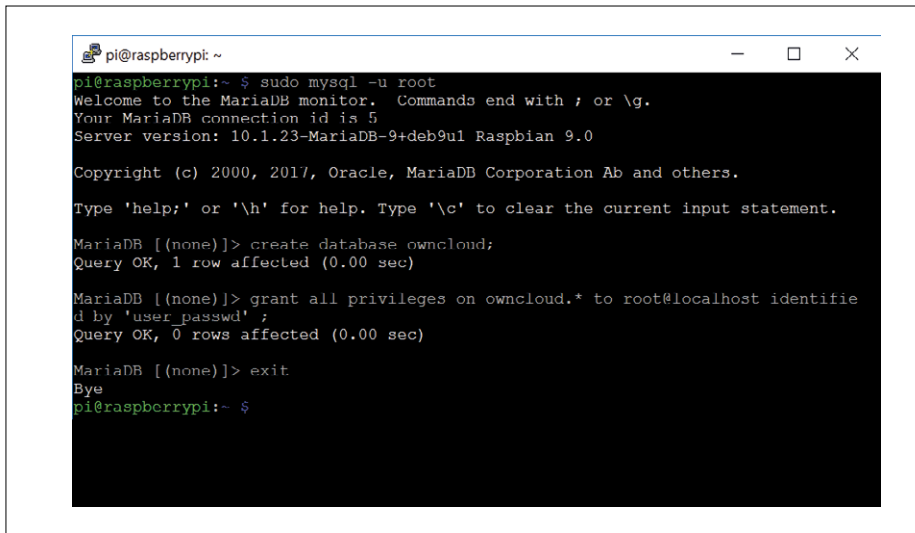


Figure 3. Preparation of the data base for user 'root' with password 'user_passwd'.



any typos, because the results can be disastrous, you would be obliged to reinstall the image on the SD card, etc. Restart the Raspberry Pi with the command:

```
sudo reboot
```

If all goes well, you can read (after connection, of course) the contents of /mnt/usb with the command:

```
ls /mnt/usb
```

Install ownCloud

The ownCloud application needs PHP version 5.6 or more recent and the web server Apache2. So we install Apache2, MySQL, PHP 7.0 plus a host of supplementary modules:

```
sudo -i
```

Now we are in 'Super User' mode which relieves us of the obligation to precede every command with 'sudo'.

```
apt-get update
```

The lines below constitute a single command:

```
apt install -y apache2 mariadb-
server libapache2-mod-php7.0 \
php7.0-gd php7.0-json php7.0-
mysql php7.0-curl \
php7.0-intl php7.0-mcrypt php-
imagick \
php7.0-zip php7.0-xml
php7.0-mbstring
```

Next we download the ownCloud package, we unpack it and copy it into our directory. Note, the version number of the package may be different from that used here (10.0.4), change it if necessary:

```
wget https://download.owncloud.
org/community/owncloud-
10.0.4.tar.bz2
tar -xjf owncloud-10.0.4.tar.bz2
sudo cp -r owncloud /mnt/usb
```

Create a database

For ownCloud to function, it is necessary to create a database:

```
mysql -u root
create database owncloud;
grant all privileges on
owncloud.* to <user>@localhost
identified by '<user_passwd>';
exit;
```

The name <user> is the name we will give to the database user and <user_passwd> will be the password (**fig. 3**).

Configure the web server

Apache2 needs to know some details on the location of ownCloud. We place this information in a configuration file:

```
nano /etc/apache2/conf-available/
owncloud.conf
```

To continue we enter:

```
Alias /owncloud "/mnt/usb/
owncloud/"
<Directory /mnt/usb/owncloud/>
Options +FollowSymlinks
AllowOverride All
<IfModule mod_dav.c>
Dav off
</IfModule>
<IfVersion < 2.3>
order allow,deny
```

Figure 4. Successful installation, you now have access to your cloud.

```
allow from all
</IfVersion>
<IfVersion >= 2.3>
Require all granted
</IfVersion>
SetEnv HOME /mnt/usb/owncloud
SetEnv HTTP_HOME /mnt/usb/owncloud
</Directory>
```

Press Ctrl-O then Enter to save the file, Ctrl-X to quit the editor. We finish the configuration with some supplementary commands:

```
ln -s /etc/apache2/conf-available/
owncloud.conf /etc/apache2/
conf-enabled/owncloud.conf
a2enmod rewrite
a2enmod headers
service apache2 restart
chmod a+w /mnt/usb/owncloud/config
```

First connection to ownCloud

To grant the Raspberry Pi web server (Apache2) access to the cloud files, we have to modify the rights. The default

user and group for Apache2 are 'www-data'. So we execute the commands:

```
chown www-data:www-data -R /mnt/
usb/owncloud
chmod 770 -R /mnt/usb/owncloud
```

Using your preferred browser, go to this address (replace « yyy.xxx » with the corresponding values for your network):

```
http://192.168.yyy.xxx/owncloud
```

You will then be invited to create an administrator account to manage the application (**Figure 4**). Enter the path where the data are stored (/mnt/usb/owncloud/data in our example) then select the type of database installed (mysql). To finish, enter the username that you have chosen (<user>), the password (<user_passwd>), the name of the database ('owncloud' in our case) and finally, for the address of the database, use 'localhost'.

And there you go! You can now use your own cloud (**Figure 5**), either from your browser, or using an ownCloud client

which you can download from ownCloud's website.

Connect via HTTPS

When you first connect, you will notice that ownCloud recommends a secure connection using HTTPS. To reconfigure our system, enter the following commands:

```
sudo -i
a2enmod ssl
service apache2 restart
a2ensite default-ssl.conf
service apache2 restart
```

The SSL module needs some keys, let's create them:

```
cd /etc/apache2
mkdir cert0C
cd cert0C
openssl genrsa -out owncloud.key
1024
openssl req -new -key owncloud.key
-out owncloud.csr
```

Worldwide access?

To access your cloud from the computer in your office — or from anywhere on the planet — you will ideally need a dynamic DNS address. For this I advise you to use the service 'NO-IP' [3]. Get onto the site and register your desired domain name. It's free but the only downside is that every month you receive a message to continue to use your account. Installation is simple enough. From your Raspberry Pi, execute the commands below in a working directory of your choice:

```
sudo wget https://www.noip.com/client/linux/noip-
duc-linux.tar.gz
tar xf noip-duc-linux.tar.gz
```

Verify the version of noip (2.1.9-1 in our case) and modify the following command if necessary:

```
cd noip-2.1.9-1
```

Before starting installation, make sure you know your login and password for NO-IP:

```
sudo make install
```

Now we need a script to launch the service at startup. Download the script 'noip' at [4] and then copy it into /etc/init.d/. To get the 'noip' to launch at startup you must type the following lines:

```
sudo chmod 755 /etc/init.d/noip
sudo update-rc.d noip defaults
sudo service noip configure
```

Start the service manually:

```
sudo service noip start
```

You can verify the status of the service with the command:

```
sudo service noip status
```

Finally, to be able to use OwnCloud from the outside, you need to modify a parameter in the file /mnt/usb/owncloud/config/config.php:

```
'trusted_domains' =>
array (
0 => '192.168.yyy.xxx',
1 => 'votre_nom_domaine',
),
```

On your local network, you should use your internal address (for example:

```
https://192.168.yyy.xxx/owncloud)
```

to set up your synchronisation client.

After having configured your box (see [3]), you can now share your files with the whole world!

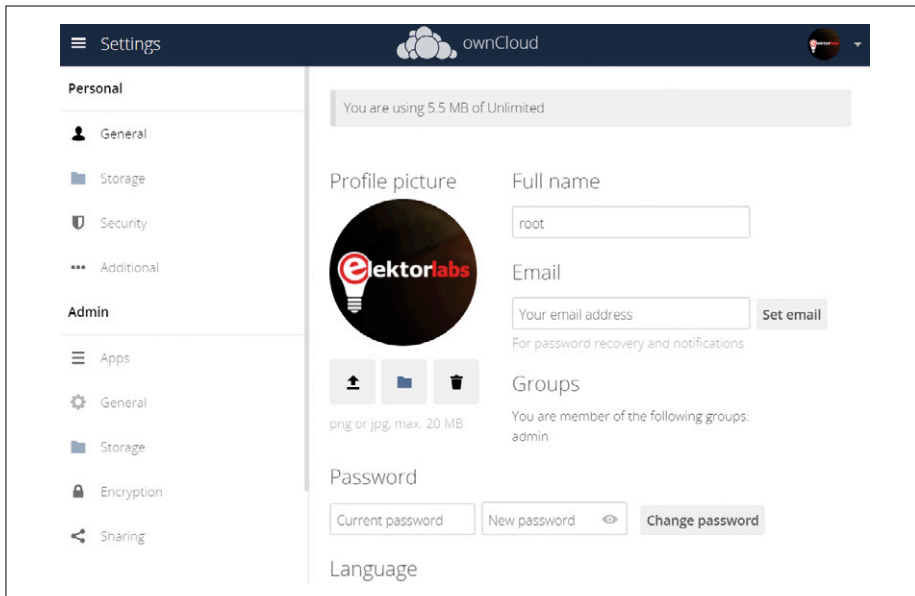


Figure 5. That's it, here we are with the dashboard of our cloud.
Now it's up to you to manage your cloud.

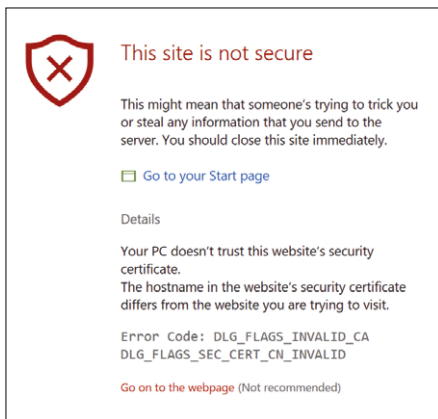


Figure 6. Don't panic! You can have confidence in yourself and your own systems, right?

You will have to respond to a certain number of questions to complete the certificate — be sure to note your responses somewhere just in case (or accept the default values).
Then:

```
openssl x509 -req -days 365 -in
owncloud.csr -signkey owncloud.
key -out owncloud.crt
```

We copy the keys:

```
cp owncloud.crt /etc/ssl/certs
cp owncloud.key /etc/ssl/private
```

A second ownCloud configuration is needed:

```
nano /etc/apache2/sites-available/
owncloud.conf
```

with contents as follows:

```
<VirtualHost *:443>
DocumentRoot /var/www
SSLEngine On
SSLOptions +FakeBasicAuth
+ExportCertData +StrictRequire
SSLCertificateFile /etc/ssl/certs/
owncloud.crt
SSLCertificateKeyFile /etc/ssl/
private/owncloud.key
</VirtualHost>
```

COMPONENT LIST

Resistors
All 0805, 0.1 W
R1,R3 = 1kΩ 1%
R2 = 2.87kΩ 1%

Capacitors
C1,C2 = 220μF, 50V, SMD, 10mm diam.
C3,C4,C5 = 22μF, 35V, SMD, 5mm diam.
C6,C7 = 100nF, 0805
C8 = 10nF, 0805

Inductor
L1 = 33μH, 3.4A, 70mΩ, e.g. Würth 74457133

Semiconductors
D1,D2 = VS-6TQ045SPBF
IC1 = LM22678TJ
LED1 = green, 3mm

Miscellaneous
K1 = DC barrel socket, 5A, 2.1mm diam.

K2,K3 = USB connector type A
K4 = USB connector type B
Hammond case 1593KBK

PCB # 160494-1



a potential danger (**Figure 6**) because the signature that we've just created is not certified. You can ignore this error (or add an exception) and carry on. This will not in any case affect the function of your cloud.

What now?

You are now in possession of your very own cloud, of a good size, over which you have total control. You can choose to switch it on or off whenever you want, launch an automatic backup of your cloud, etc. It's you who is in command, it's you who is piloting the cloud. ◀

(160494)

Press Ctrl-O then Enter to save the file, Ctrl-X to quit the editor.

Finally, we activate the new configuration and restart the Apache2 web server:

```
a2ensite owncloud.conf
apachectl configtest
service apache2 restart
```

You can now connect to your cloud using the address <https://192.168.yyy.xxx/owncloud> (with an 's' on the HTTP). Your browser will warn you of

Web Links

- [1] Raspbian: www.raspberrypi.org
- [2] ownCloud: owncloud.org
- [3] Service NO-IP: www.noip.com
- [4] Page for this article: www.elektormagazine.com/160494



FROM THE STORE

→ 160494-1

Power supply printed circuit board, bare

Advertisement

The Connected Greenhouse

IoT demonstration project using MQTT and Node-RED

By **Walter Trojan** (Germany)



Imagine that you have decided to build a complex IoT (Internet of Things) project, such as a system for monitoring and controlling the greenhouses in a nursery. Among other things, this might involve stabilizing the interior temperature and lighting at preset target levels and monitoring the humidity of the interior atmosphere and the moisture level in the soil in which the plants are growing. If these values go outside a specified range an alarm must be raised, for example by sending an e-mail to the nursery manager. Not a trivial undertaking, you might think; but these days the availability of cheap hardware and free software makes it easy to tackle an intricate IoT project of this type.

A professional approaching a project like this would start with a functional specification document, including a detailed definition of the solution and expected costs and delivery schedules. Unfortunately it is not possible at the outset to anticipate all possible difficulties and risks that might be encountered, and so it is worthwhile to carry out an initial feasibility study. This article describes just such a feasibility study, employing a model greenhouse populated with a real living plant, and looks at the hardware and software components that could be used. For reasons of space we cannot go into the full constructional details, but we can give some suggestions that might come in handy for your own projects. In the project we use familiar hardware such as the Raspberry Pi and the ESP8266 [1], along with cutting-edge software technologies such as MQTT and Node-RED. MQTT is a bandwidth-efficient IoT communications protocol, which typically runs on top of TCP/IP, and Node-RED [2] is a graphical programming system that is ideal for building prototypes and complex applications.

Project overview

It is of course well within the capability of a Raspberry Pi or an ESP8266 to monitor and control a model greenhouse. However, that would not capture the potential scale of the real project, which involves many greenhouses. For that reason we share the work among several devices, connected to one another using MQTT running over a WLAN.

Figure 1 shows the overall structure of the project. The active components (the Raspberry Pi and the ESP8266-based sensor and actuator boards) are connected together using a WLAN. These components exchange readings and commands using the MQTT protocol. The master of ceremonies is the Raspberry Pi, on which the Mosquitto MQTT broker software and the Node-RED application are installed. The control and monitoring process involves the following steps, executed cyclically.

- The sensor board measures the desired environmental parameters (temperature, light and humidity) and publishes the readings. This involves sending a message to the MQTT broker.
- The broker checks whether the message is of interest to other devices

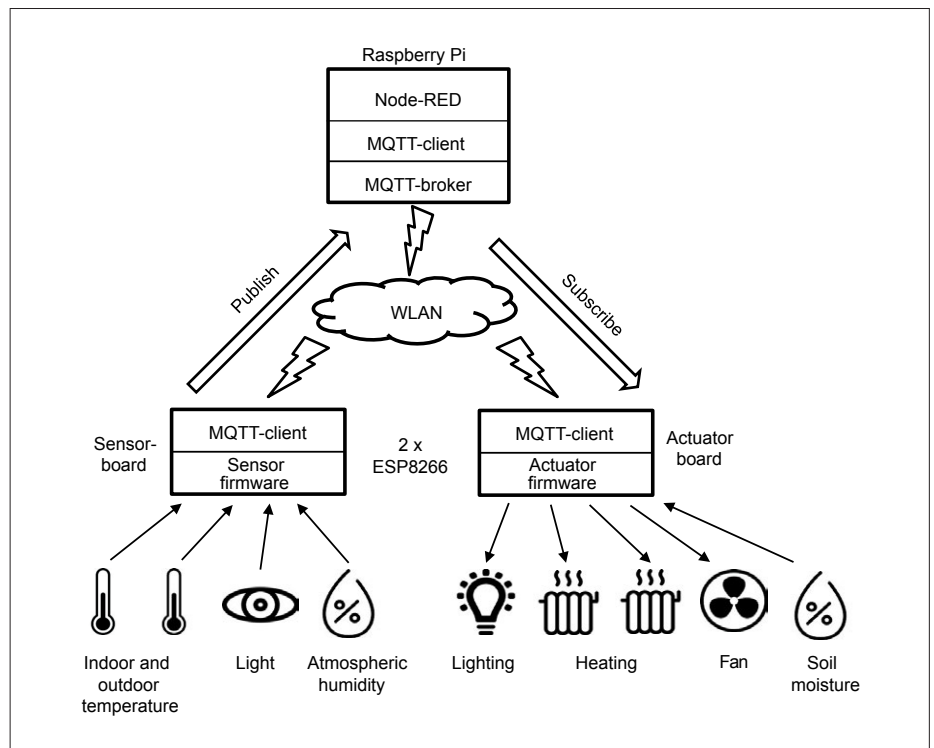


Figure 1. Configuration of the model greenhouse.

in the network and if necessary forwards it.

- The Node-RED application, which is subscribed to the message, processes it and, if required, publishes commands in response.
- The actuator board, which is subscribed to specific command messages from the Node-RED application, receives any commands that relate to it and executes them.

Even from this high-level view of the system it is possible to see how the MQTT network operates. At the center there is always a broker whose job is to mediate the transfer of messages over the network. The network can connect together a practically unlimited number of devices. Some network participants publish messages without knowing which participants will read it; others subscribe to these messages and can process their contents, without knowing from where they originated. The details of this process are explained in more detail below.

Our model greenhouse

The front view of the model greenhouse is shown in our lead photograph. It is mostly made of wood and acrylic. Its dimensions are such that a medium-sized houseplant will fit inside it: the base plate is a wood offcut 0.8 cm thick

and measuring 50 cm by 30 cm, and to this are fixed 5 mm acrylic sheets 30 cm high to form the walls. The right-angle joints are made using lengths of 1.5 cm square cross-section wooden strips to which the various sheets are screwed. The detachable roof is also made from sheet acrylic, joined at the ridge using a triangular cross-section wooden strip. A handle fixed to the middle of the ridge makes the model greenhouse portable. The roof is firmly attached to the walls using mending plates and four M6 bolts. Two halogen heater lamps (12 V, 10 W) are mounted in diagonally-opposite corners of the enclosure, and the two processor boards are mounted in the middle of the short edges. A strip of white LEDs in the roof ridge provides bright illumination for the plant, while a fan on the side of the enclosure provides ventilation. The power supply is also mounted on the side of the enclosure. A two-pin sensor stuck into the soil measures its moisture level. In the photograph the heating lamps and the LEDs are at full intensity, which should be keeping the plant nice and cosy.

What do we need?

Having outlined the system we can now look at the particular hardware and software components that we will need in more detail.

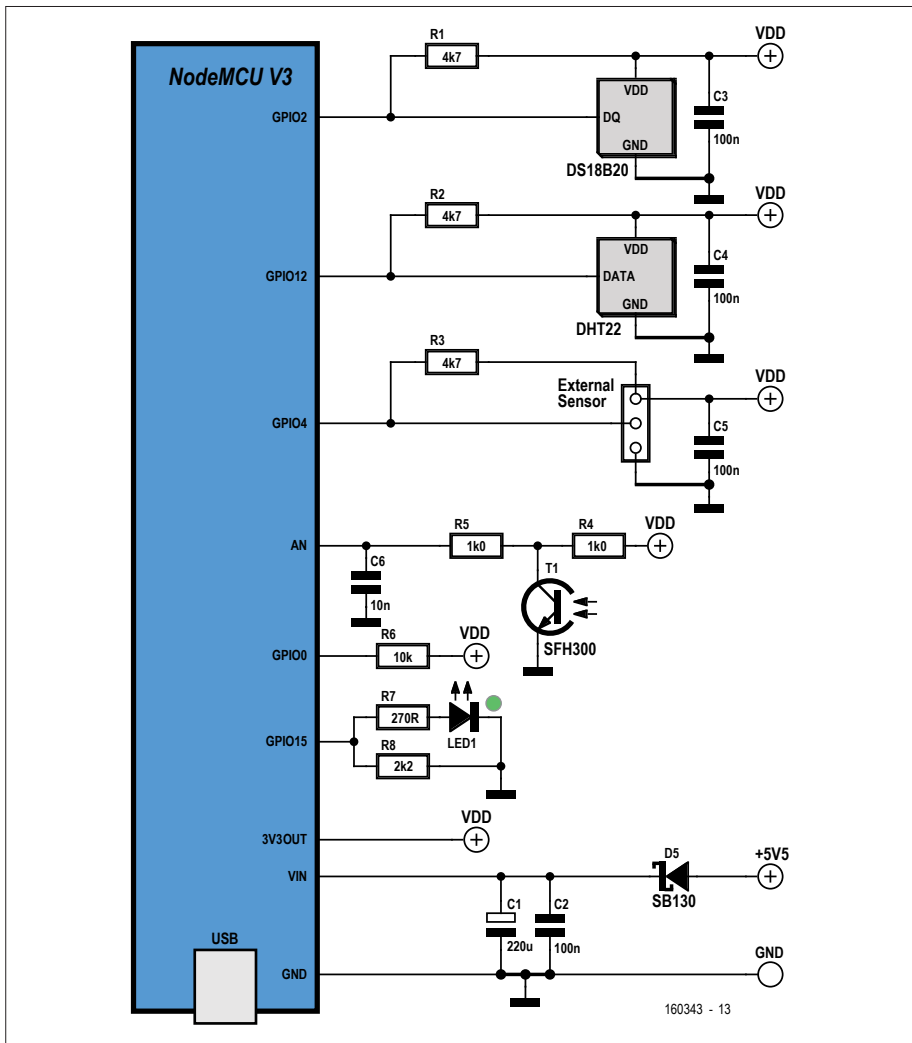


Figure 2. Circuit of the sensor board.

Raspberry Pi

We will be using a Raspberry Pi 3 with a four-core processor and 1 GB of main memory, which gives us plenty of processing power to spare. The most up-to-date Raspbian operating system, Raspbian Stretch, is stored on an 8-GB SD card. The system comes with a WLAN chip fitted and thus is easily connected to the wireless network. Fortunately Node-RED is already pre-installed in Raspbian so we only need to install the Mosquitto broker and the Node-RED application. When configuring the system, and for debugging, it is helpful to connect a USB mouse and keyboard and an HDMI monitor, but these are not needed for normal day-to-day operation: instead it is possible to log in to the Raspberry Pi using the 'ssh' command from an external PC to make any adjustments that might be necessary.

Sensor board

This board (see **Figure 2**) consists of an ESP8266 on a NodeMCU V3 board and handles the interface to the sensors to measure environmental parameters. The indoor temperature sensor is a DS18B20, the humidity sensor is a DHT22 and a phototransistor is used to measure the ambient light level. An external input allows a further DS18B20 to be added to measure outdoor temperature. If the outdoor temperature falls below a specified threshold, for example 4 °C, the operator can be sent an e-mail warning of the risk of frost. The temperature sensor built into the DHT22 is not used, although it would be possible to use it to check the measurement from the DS18B20. The structure of the firmware for the ESP8266 is straightforward. The sensors are polled every 10 s within a loop, and the readings are published using MQTT. The sensors are read using their drivers.

The program was developed in C using Eclipse and in C++ using the Arduino environment: you can download it from the Elektor website at [3] and experiment further with it.

Actuator board

The ESP8266 NodeMCU board has four high-current outputs A1 to A4 to control the lighting, heating and ventilation of the greenhouse: see **Figure 3**. Each of these outputs is pulse-width modulated with a duty cycle from 0 % to 100 % and can run at up to 12 V. Lighting is provided by several LED strips populated with bright white LEDs. It is known that plants prefer blue and red light; since white light contains all colours, the plants should thrive. Heating is provided by two 12 V halogen lamps each with a power rating of 10 W, switched by separate outputs from the board. If the control algorithm demands only a small amount of heating power, the lamps are dimmed by adjusting the PWM signals. At elevated levels of temperature or humidity the 12 V fan is switched on to provide cooling and air circulation.

Since the analogue input to the NodeMCU on the sensor board is used by the phototransistor, the soil moisture level sensor must be connected to the otherwise unused analogue input on the actuator board. The sensor itself consists of two electrodes that are pushed into the soil; the resistance between these electrodes is measured and converted to an analogue voltage. A subsequent operational amplifier on a small adapter board converts the voltage range to 0 V to 3.3 V. The firmware determines the moisture level from this voltage and sends the result to the central controller using MQTT. In turn the central controller can send a warning e-mail to the operator if the soil is too dry.

The firmware for the actuator board was also developed in C using Eclipse and in C++ using the Arduino environment, and again the code can be downloaded from the Elektor website. The code receives commands from the Node-RED application over MQTT and generates the required PWM signals on the high-current outputs. It also periodically communicates the soil moisture level readings to the application.

Completing the circuit

The power supply is straightforward: 18 V DC from a laptop adapter is

taken to two DC-to-DC converters (see **Figure 4**). One of these produces 5.5 V to power the NodeMCU boards: the extra 0.5 V compensates for the forward voltage drop across the reverse polarity protection diodes on the actuator and sensor boards. The second DC-to-DC converter supplies the power outputs at 12 V. The input to the power supply circuitry is also protected against a voltage of reverse polarity being applied by diode D1. Since the heaters alone can dissipate as much as 20 W at full power, a reasonably capable laptop adapter is required: we could recommend one specified to deliver at least 3 A.

A quick look at MQTT

In contrast to the request-response architecture of the widely-known HTTP protocol, MQTT is based on a publish-subscribe architecture, based around a central broker. Devices which have a message to send simply ‘publish’ their message to the broker. The broker then forwards these messages to any other devices that require them, where they are processed. The nodes that wish to receive messages must ‘subscribe’ to them. Communication can happen in both directions: an individual device on the network can simultaneously be a publisher and a subscriber. A node might be a simple microcontroller, a PC or a Linux server. The only requirements are the presence of a TCP/IP stack and an implementation of the MQTT protocol. The addressing scheme for publishing and subscribing to messages uses a mechanism called a ‘topic’. This is a string which acts as a kind of ‘subject’ field for a message, and which takes a form resembling a URL. A temperature sensor in a study might, for example, publish its current reading using the topic string ‘MyHouse/Study/Temperature’, or using a shorter string such as ‘Test1’. The payload and other parameters are transmitted along with the topic string. MQTT is data-agnostic: the payload can be binary data, text or even XML- or JSON-format data structures.

The reliability of message transfer can be configured at one of three levels as follows.

- Level 0: no particular guarantees (‘fire and forget’)
- Level 1: guaranteed transfer of at least one copy of the message (but multiple copies might be transferred)

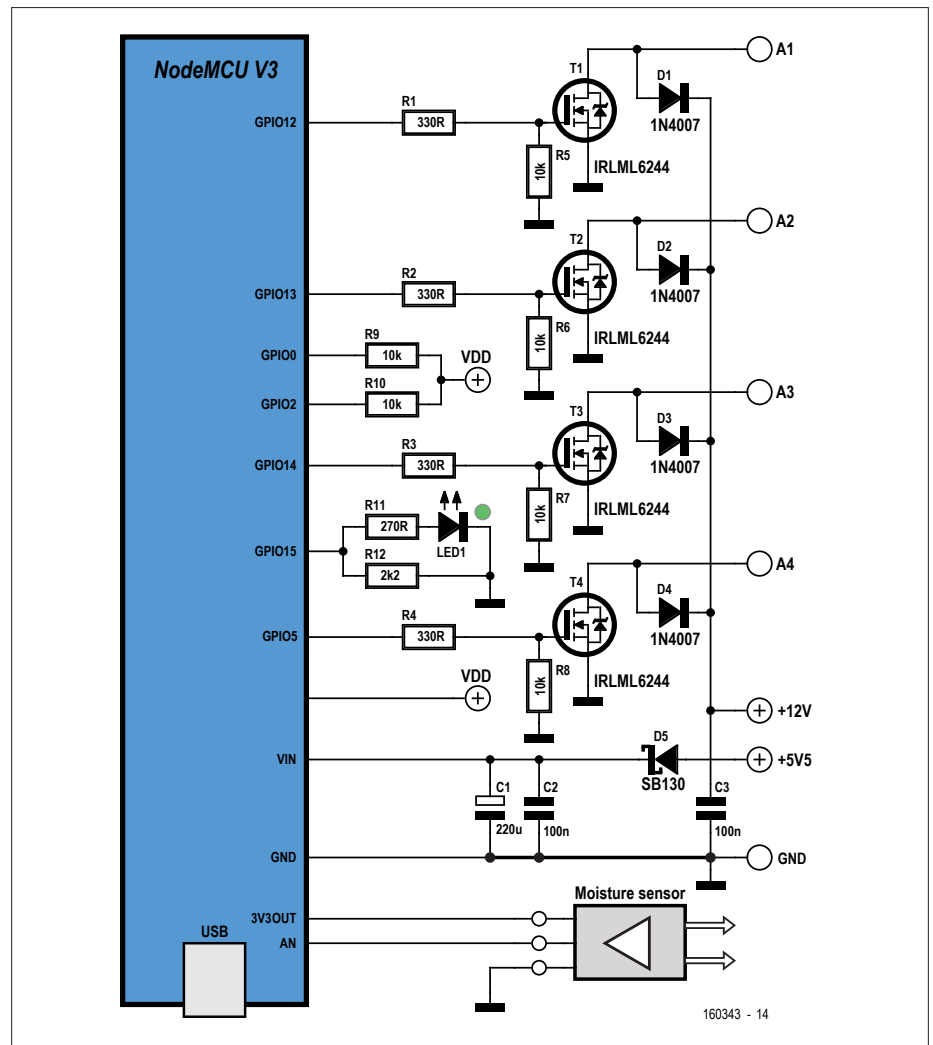


Figure 3. Circuit of the actuator board.

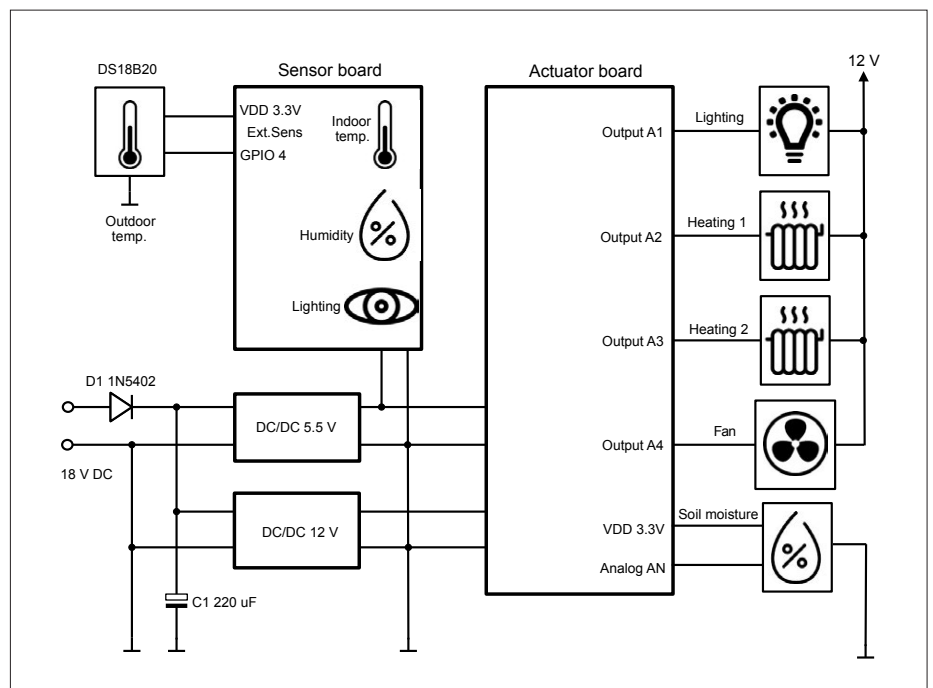


Figure 4. Block diagram of the greenhouse control system.

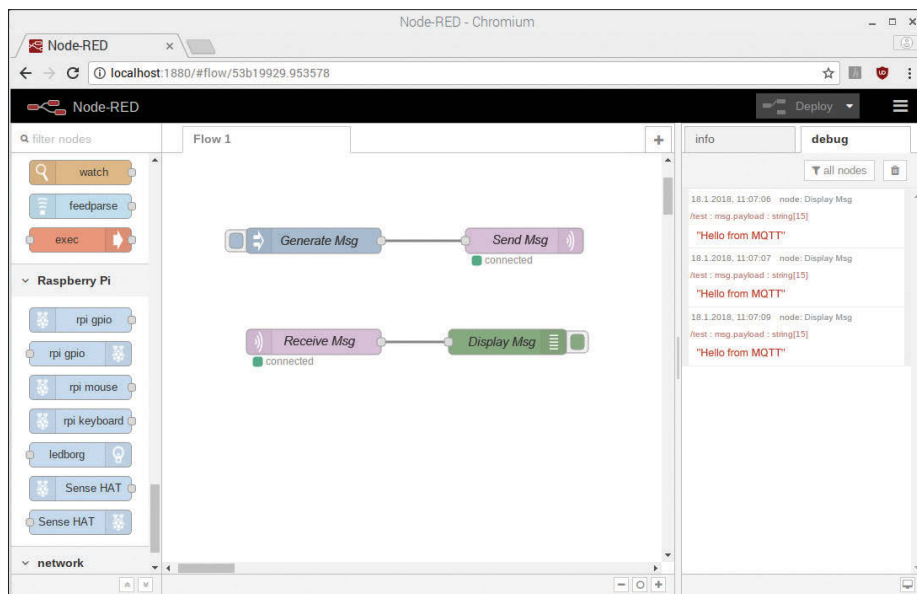


Figure 5. Our first Node-RED flow using MQTT.

- Level 2: guaranteed transfer of exactly one copy of the message

And, perhaps surprisingly, MQTT allows a node to lodge its 'last will and testament' with the broker! What this means is that should the node fail, the broker will execute a specified series of commands. This might, for example, generate an alert for the administrator responsible for maintaining the system. Now, rather than waxing lyrical over many pages on the subjects of message filtering, multi-layered security and features for protecting against failures, I refer you to the wealth of information about MQTT available on the Internet.

MQTT on the Raspberry Pi

For this project I have selected the free Mosquitto [4] implementation by the Eclipse Foundation from among the many available MQTT brokers. The most recent version (1.4.10-3) can be installed on your Raspberry Pi using the following steps.

```
wget http://repo.mosquitto.org/
  debian/mosquitto-repo.gpg.key
```

```
sudo apt-key add mosquitto-repo.
  gpg.key
```

```
cd /etc/apt/sources.list.d/
```

```
sudo wget http://repo.mosquitto.
```

```
org/debian/mosquitto-stretch.
  list
```

```
sudo apt-get update
```

```
sudo apt-get install mosquitto
  mosquitto-clients
```

Mosquitto runs on the Raspberry Pi as a demon and starts up automatically. To test whether Mosquitto is running use the following command.

```
sudo service mosquitto status
```

The service can be stopped and started as follows.

```
sudo service mosquitto stop
```

```
sudo service mosquitto start
```

The following quick test will verify that everything is working. Open two terminal windows on your Raspberry Pi; in the first window subscribe to the topic 'Test' as follows.

```
mosquitto_sub -t test
```

In the second window publish the text 'Hello' under the same topic.

```
mosquitto_pub -t test -m Hello
```

The text 'Hello' should now appear in

the first window. The subscription can be canceled by pressing control-C. The basic components are now in place, and we can start to build our greenhouse controller application on top of them.

Graphical programming using Node-RED

In order to minimize the effort involved in building the application a slightly unusual tool was employed. Rather than working with lines of program source text, I used a graphical editor to connect together functional blocks to form the application. Because in this approach messages 'flow' through a network of connections the programs are called 'Flows'. The (free) tool was developed in 2013 by IBM; it is now open source and is called Node-RED [2].

Node-RED is a convenient way to create applications for the Internet of Things as well as in other areas. The tool consists of a run-time environment, a graphical editor for program development, a dashboard for communicating with the application and a wide range of preconfigured functional blocks. These blocks, called 'nodes', can be rather powerful: for example, they can represent MQTT publishing or subscribing clients, web browsers or e-mail subsystems. The nodes have inputs and outputs, process incoming messages, carry out various actions and forward the messages (potentially modified) on their outputs to other nodes.

The structure of an application is described in the graphical editor by placing the required nodes on the canvas and then connecting them with message paths. The flow of messages is triggered by an event, and ends when a node does not produce an output message in response to its input.

Node-RED runs on the JavaScript-based platform Node.js, which in turn is built upon the Google Chrome V8 JavaScript engine. If a node must be configured to carry out some custom task, then this can be done using some JavaScript code: even in Node-RED a little bit of classical programming can be necessary. The graphical editor used for program development and the application interface are both implemented in the browser.

Node-RED has rapidly established itself as a general-purpose development tool for IoT applications. Over recent times a quickly-growing band of users have adopted Node-RED and the

highly-engaged developer community ensures continuous enhancements and maintenance. Today over 1400 special-purpose node designs and example flows are available for download. If you want to know more, again there is a wealth of information available on the Internet.

Our first MQTT flow

Fortunately Node-RED is fully integrated into Raspbian, and the run-time environment can be launched under 'Menu/Programming/Node-RED'.

This opens a terminal window with a few start-up messages that are not relevant to us at the moment. Node-RED can be stopped when it has finished doing its tasks using the control-C key combination. The graphical program editor runs in the Chromium browser: go to the address 'localhost:1880' and you should see the following windows.

On the left, the **nodes** window. This displays over 60 nodes under various categories including input, output, function, social, storage, analysis, advanced, Raspberry Pi and network. The special nodes for use with the Raspberry Pi are particularly useful as they allow convenient access to GPIOs, to the keyboard and mouse, as well as to LEDs and sensors.

In the middle, the **program** window. The required nodes can be dragged into this window, connected together, and then configured.

On the right, the **info** window. This has tabs labelled 'info' and 'debug': 'info' gives information about the selected node while 'debug' shows debug information.

Our first test will demonstrate the operation of Node-RED in the context of MQTT (see **Figure 5**). In this example we have four nodes connected in two flows. In the upper flow the Inject node 'Generate Msg' creates the MQTT payload text 'Hello from MQTT' and sends it to the MQTT Output node 'Send Msg'. This node publishes the message under the topic '/test' to the MQTT broker; and that is the end of this flow.

The lower flow starts with the MQTT Input node called 'Receive Msg', which is subscribed to the topic '/test'. When a message arrives under this topic, the flow forwards it to the Debug node called 'Display Msg'. In the example shown the

button on the 'Generate Msg' node has been clicked three times, resulting in three messages being displayed in the debug window on the right.

The parameter settings for any of the nodes described above can be displayed for editing in a window by double-clicking on it. The parameters should be set as follows.

For the Inject node, select 'a-z string' for the payload and enter 'Hello from MQTT'. Set the name to 'Generate Msg' and then click 'Done'.

For the MQTT Output node, edit the server configuration to set the server to 'localhost' and the port number to 1883; then click on 'Add' or 'Update'. Set the topic to '/test', 'QoS' to zero, and the name of the node to 'Send Msg'. Then click on 'Done'.

For the MQTT Input node, set the topic to '/test', 'QoS' to zero, and the name of the node to 'Receive Msg'. Then click on 'Done'.

No parameters need to be set for the Debug node.

When this first test flow has been tested successfully we can start to experiment with more complicated flows.

Putting it all together

We are now at the point where we can define the parameters for our MQTT communications. There are four channels on the actuator board for controlling the lighting, the fan and the two heaters, each of which has a PWM signal which is controlled by a variable in the range from 0 to 255. With an eye towards future expansion to the full-scale system,

we will name the corresponding topics 'actor1/x' where 'x' is the channel number. When we add further actuators their topics will be of the form 'actor2/x', 'actor3/x' and so on. The payload for each of these topics could just be a single number in the range from 0 to 255; but in the interests of clarity and to simplify debugging we will add the prefix 'chx_' where again 'x' is the channel number.

Table 1 shows a few examples.

The sensor board delivers readings for ambient light intensity, humidity and indoor and outdoor temperature. The topic strings we use will be 'sens1/' plus the sensor type. The range of possible values varies considerably from sensor to sensor: for example, the light sensor returns values from 0 to 1023, where 0 corresponds to maximum brightness and 1023 to complete darkness. The humidity sensor returns the relative humidity in percent, and the two digital temperature sensors return readings in Celsius, with negative values suppressed.

The actuator board hosts the soil moisture sensor: for this we use the topic 'sens2'. Again in this case the range of possible values runs from 0 to 1023, where 0 corresponds to maximum moisture content and 1023 to zero moisture content. The lighting and soil moisture readings are converted into percentage values in the central controller.

Installing the greenhouse flow

Before we can install the (somewhat complex) greenhouse flow a little more groundwork is necessary. First install the

Table 1. Example MQTT messages.

Function	Topic	Payload	Range	Example
PWM signal on A1 (light)	actor1/ch0	ch0_xxx	0 to 255	ch0_64
PWM signal on A2 (fan)	actor1/ch1	ch1_xxx	0 to 255	ch1_255
PWM signal on A3 (heater 1)	actor1/ch2	ch2_xxx	0 to 255	ch2_0
PWM signal on A4 (heater 2)	actor1/ch3	ch3_xxx	0 to 255	ch3_128
Light intensity	sens1/lght	lght_xxxx	0 to 1023	lght_500
Indoor air humidity	sens1/humi	humi_xxx	0 to 100	humi_80
Indoor temperature	sens1/tpin	tpin_xxx	0 to 100	tpin_27
Outdoor temperature	sens1/tpau	tpau_xxx	0 to 100	tpau_22
Soil moisture level	sens2/erde	erde_xxxx	0 to 1023	erde_300

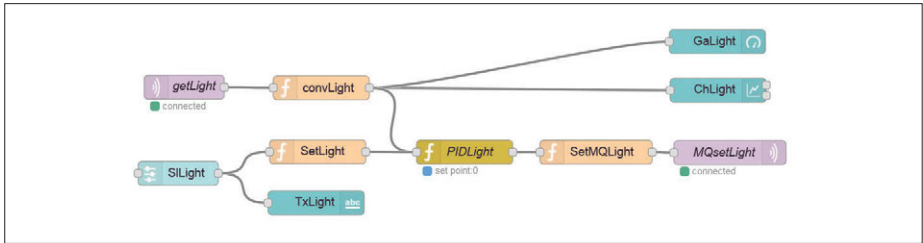


Figure 6. Flow for lighting control.

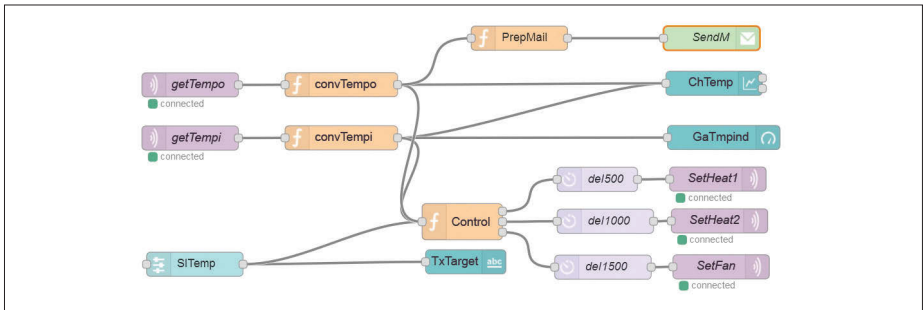


Figure 7. Flow for temperature control.

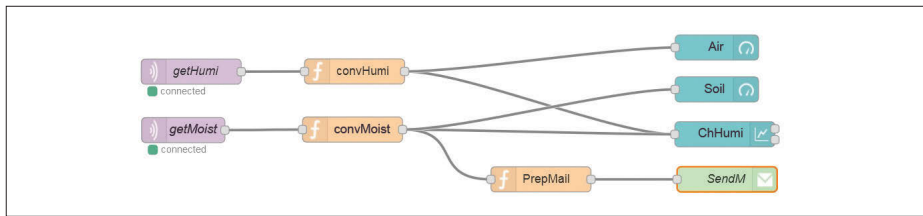


Figure 8. Flow for humidity and soil moisture monitoring.

npm package manager on the Raspberry Pi using the following command.

```
sudo apt-get install npm
```

Then move to the Node-RED directory ('/node-red') and install the Node-RED dashboard. This gives a user-friendly graphical interface to the Node-RED system.

```
npm i node-red-dashboard
```

And now we use npm again to install the PID controller node as follows.

```
npm i node-red-node-pidcontrol
```

The Raspberry Pi must now be rebooted and Node-RED launched again. In the Node-RED editor you will see a row of new nodes for creating graphical output. We can now proceed to load the flow for the greenhouse application. Node-

RED flows are stored in a JSON-based format and they can be imported and exported via the clipboard. Copy the file Greenhouse_Flow2.json from the download page on the Elektor website [3] onto the Raspberry Pi, open it using the Geany editor, copy the contents of the file to the clipboard and then import it into Node-RED using the Import/Clipboard menu option as 'Flow2'. You will now be able to inspect the overall flow with the component flows for handling light, temperature and soil moisture content.

PID control for lighting

The lights in the greenhouse are controlled smoothly using a PID controller to maintain a chosen constant overall ambient light intensity (Figure 6). 'PID' stands for 'Proportional-Integral-Differential', a control algorithm that, assuming its parameters are appropriately set, guarantees good

tracking of the target value. The mathematics behind the controller is implemented in the PIDLight node; the behavior of the controller is determined by three constants giving the coefficients of the proportional, integral and differential terms in the control loop.

In this flow the current ambient lighting level is periodically received by the getLight node over MQTT. The value is normalized in the convLight node and then passed to the PIDLight controller. The target level is set using the SLight slider node, and this value is fed into the controller where it is compared with the current reading. The output of the controller is a correction signal which is converted into an MQTT payload in the SetMQLight node. The actuator board receives this message and adjusts the brightness of the lighting accordingly. The 'function' nodes, which are marked by a script letter 'f', each include a few lines of JavaScript to implement their specific task. The blue nodes are graphical input and output elements that appear on the dashboard, such as sliders, text boxes, gauges and charts.

Two-input temperature controller

In this component flow (see Figure 7) the current outdoor and indoor temperature readings are received over MQTT. If the outdoor temperature falls below 4 °C a warning message is prepared in the PrepMail node and sent to the operator in the SendM node. Temperature readings, after normalization, are fed to the two-input Regulator node, where they are compared with the target value set by the SiTemp slider node. The Regulator node computes the control values for the heaters and for the fan and sends them to the actuator board using MQTT.

This controller does not use any difficult mathematics. Instead, a simple series of thresholds are defined and the power to the heating lamps is adjusted in discrete steps. Depending on the indoor and outdoor temperature levels the fan will contribute either to heating or to cooling the greenhouse. In order to avoid large spikes in current draw when several loads are switched simultaneously, the MQTT messages are sent with delays of between 500 ms and 1500 ms. Again, information about the status of the flow is displayed on the dashboard.

Soil moisture level monitoring

The air humidity and the soil moisture level readings are normalized and displayed on the dashboard. If the soil moisture level falls below 20 % a message is prepared in the PrepMail node and sent to the operator in the SendM node (see **Figure 8**).

System overview

The aim of this project is to display all the data relevant to the operation of the greenhouse in one place, and to test the behaviour of the controllers. To this end we make use of the dashboard and nodes available in Node-RED for creating graphical displays: these need to be specially installed as we described above. The dashboard can be displayed by opening a new tab in the browser and entering the following address.

<http://localhost:1880/ui>

This window (see **Figure 9**) just shows the graphical elements themselves. To configure the parameters of a dashboard node double-click on it in the Node-RED editor window.

Across the top of the display there are three long-term charts showing the measured temperature, humidity and lighting levels. In this example we have set the time period displayed to just 5 minutes so that even small changes in temperature can be seen. In practice the period would be set to several hours, or perhaps one would implement a combination of short-term and long-term charts. It is possible to display more than one variable on a chart, and since version 2.3.2 of the dashboard it is possible also to specify the colours of the lines. The current indoor temperature, humidity and light levels are also displayed using a selection of indicator designs.

At the bottom of the display there are two sliders to set the target values for indoor temperature and light level. The screenshot shows how the dashboard appears on an Android tablet; on a

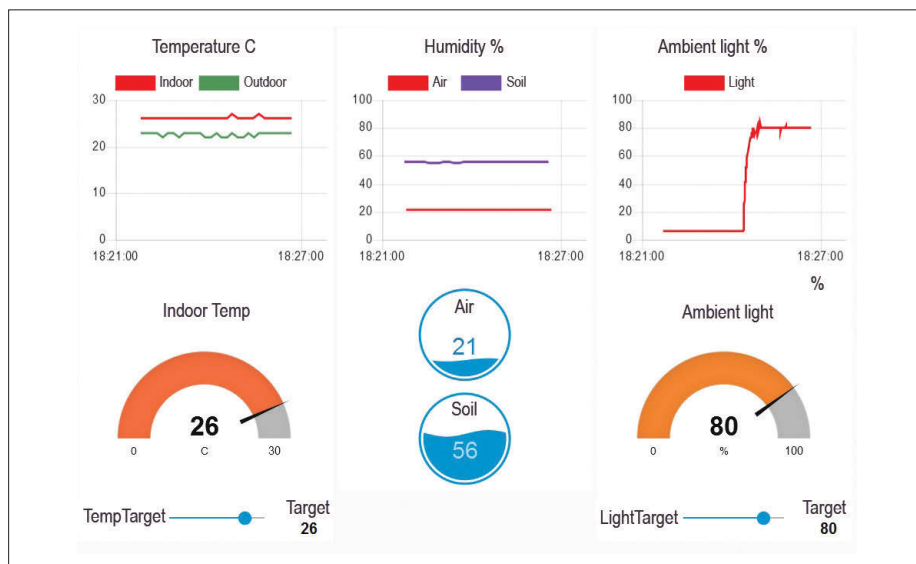


Figure 9. Operating parameters on the dashboard.

smartphone the graphical elements are presented vertically, but the result is still perfectly legible.

Learning all the time...

The model greenhouse demonstrates that the full-scale project is well within the capabilities of the Raspberry Pi and ESP8266 hardware and the MQTT and Node-RED software. The combination of MQTT and Node-RED makes it easy and quick to implement the application logic. It is particularly useful for debugging to be able to see all the messages (or just the messages of interest) on the PC by simply using the MQTT subscription mechanism.

However, a couple of modifications will be necessary. The central Raspberry Pi controller is a 'single point of failure' and reliability will have to be improved by using multiple brokers with MQTT bridges or a cluster configuration. Means will have to be found to protect the network from external attack, which could compromise its operation. One approach to this would be to use TLS (transport layer security) encryption for all communications. These precautions

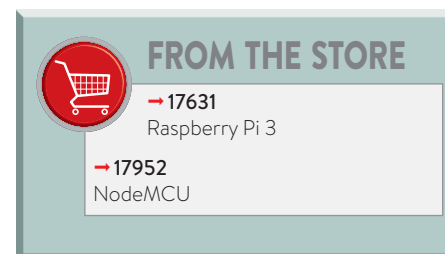
are all perfectly feasible, but require know-how and extensive experimentation and testing. But that's what makes the job of an engineer so fascinating!

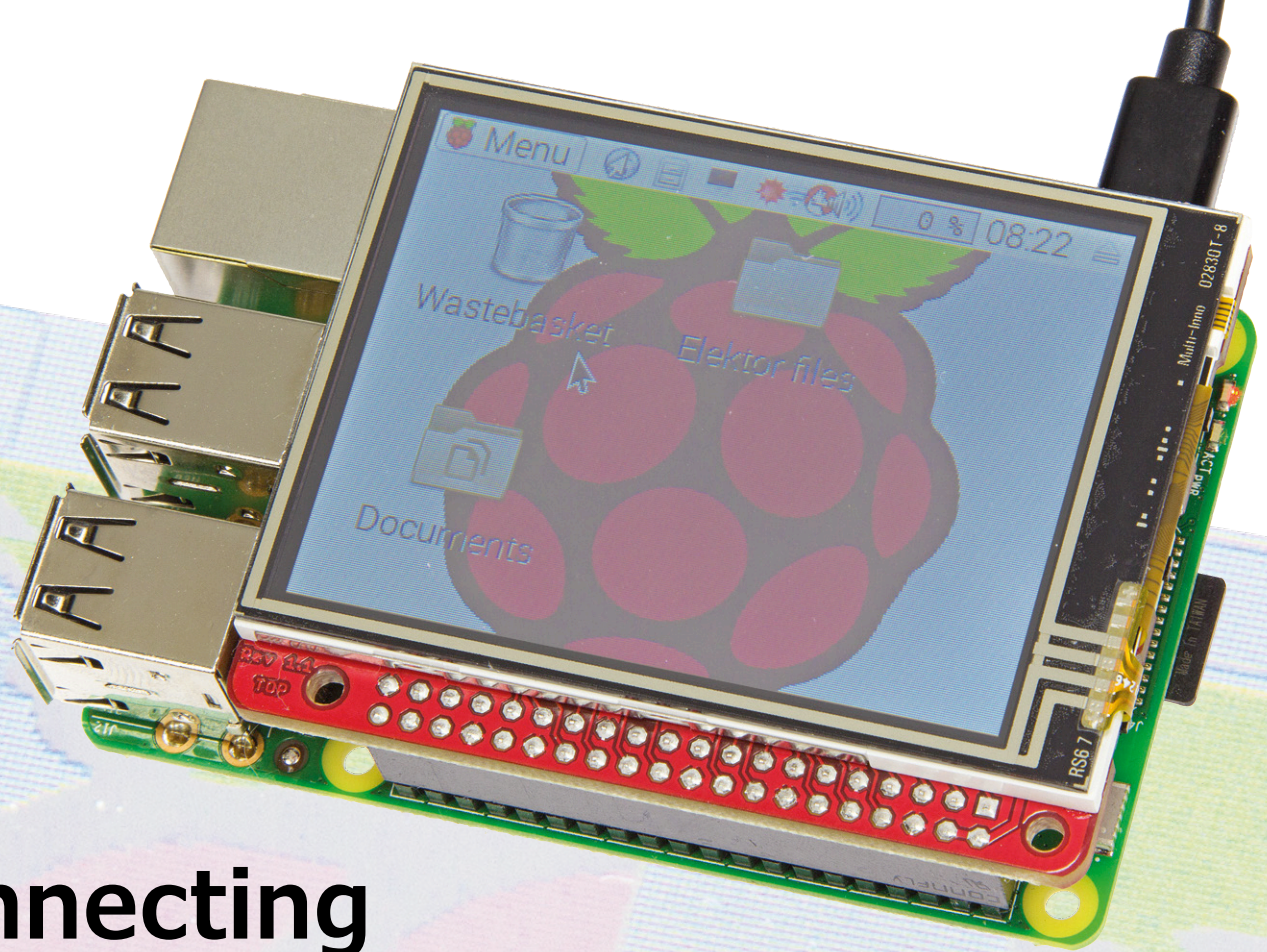
So that interested readers can quickly carry out some initial tests on a Raspberry Pi, I have created a complete Raspbian Stretch image with broker, dashboard and flow pre-installed. The image is available from the Elektor website [3]. It should be noted that this article is classified as Homelab hence was not processed by Elektor Labs. Consequently some parts of the program are in German. After unpacking the image it can be copied directly to an SD card from which the Raspberry Pi will boot. A C++ program for the Arduino IDE and a C version for the CHERTS SDK [5] are also available for flashing the ESP8266 firmware: make sure you read the notes in the README file. Happy experimenting! ◀

(160434)

Web Links

- [1] ESP8266: <https://bbs.espressif.com/>
- [2] Node-RED: <https://nodered.org>
- [3] Software download: www.elektormagazine.com/160434
- [4] Mosquitto: <http://mosquitto.org>
- [5] CHERTS SDK: www.esp8266.com/viewtopic.php?f=9&t=820





Connecting an LCD to an Rpi

How easy (or difficult) can that be?

By **Luc Lemmens** (Elektor Labs)

There are large numbers of add-on boards available for the Raspberry Pi (the so-called 'HATs', Hardware Attached on Top) for a diverse range of applications. Some work immediately once you connect them, others have to be configured first or be provided with the correct software. This is not always immediately obvious...

Particularly in the case of a display module it would be handy to know what you need to do in order to conjure up an image on the screen, even if it is only for the purpose to show whether it is working or not. For some time now, Elektor have been selling a small touch-display for the Raspberry Pi [1] in their Store. The small color screen with 320 x 240 pixels and a diagonal of 2.8 inches is provided with a resistive touch panel and is very affordable. This display module is the same size as the Rpi. That makes this combination quite compact and can therefore be easily built into something.

Several buyers of this display asked us what software is required in order to work with this display; the standard Linux distributions from the Raspberry Pi organization apparently don't support this display.

The manufacturer of this display-module, the German company Watterott, has made a github website where the necessary information and software are made available [2]. Among the downloads from this page are ready-to-run SD-card images

(Raspbian/Debian), with support for this touch display already integrated. With these you can simply make a new SD card. For an existing installation you can download a script from the same website that will install the FBTFT Framebuffer for the display. We have tested that script here in the labs and it works perfectly. In the event that you prefer to do everything yourself or if the script doesn't work, there is also an explanation as to how you can add the drivers manually. In any case, note that the touch screen needs to be calibrated.

A few other projects are also mentioned on the same website, built around the combination of Rpi/touch-display, such as a web radio and a mini TV. Certainly worth the effort to take a closer look!

(150824)

Web Links

[1] www.elektor.nl/touch-display-for-raspberry-pi

[2] <https://github.com/watterott/RPi-Display>

Join the Elektor Community

Take out a GOLD Membership now!



- GOLD € 2,45/week**
- ✓ Access to Elektor Archive
 - ✓ 10% Discount in Elektor Store
 - ✓ 6x Elektor Magazine (Print)
 - ✓ 6x Elektor Magazine (PDF)
 - ✓ 6x Elektor Business Magazine (PDF)
 - ✓ Exclusive Offers
 - ✓ Access to over 1,000 Gerber files
 - ✓ Elektor Annual DVD

Also available:

The all-paperless GREEN Membership!



GREEN € 1,78/week

- ✓ Access to Elektor Archive
- ✓ 10% Discount in Elektor Store
- ✓ 6x Elektor Magazine (PDF)
- ✓ 6x Elektor Business Magazine (PDF)
- ✓ Exclusive Offers
- ✓ Access to over 1,000 Gerber files



The screenshot shows the Elektor Store website. It features a search bar, navigation links (HOME, DOWNLOADS, NEWS, FORUM, CONTACT), and a shopping cart icon. A prominent banner advertises the 'Sand Clock' (Arduino based eye-catcher) for €119.95, now €79.95, with a 'Save €40 Now!' offer. Other sections include 'SPECIAL OFFER' for the 'Edition 5/2017' and 'LAB EQUIPMENT' for the 'Pi-TOP'.

www.elektor.com/member

