Zusatzmaterial

1 Zeit ohne Internetverbindung einstellen

1.  Terminal starten: `sudo date „1204080014“`
2. Die Zeit wird in folgendem Format angegeben: `MMTThhmmJJ`

Diese Zeiteinstellung gilt nur bis zum nächsten Neustart. Es gibt keine batteriegepufferte Uhr. Sobald der Raspberry Pi eine Internetverbindung hat, wird automatisch die richtige Zeit angezeigt.

2 Software aus dem Pi Store deinstallieren

1. Unter ***My Library*** installiertes Programm auswählen.
2. Auf ***Delete*** klicken.

3 Thunar – erweiterter Dateimanager in den xfce4-goodies

Der bei den xfce4-goodies mitgelieferte Dateimanager Thunar hat deutlich mehr Möglichkeiten, die Anzeige und das Verhalten des Dateimanagers dem persönlichen Geschmack anzupassen.

- Thunar bietet die Möglichkeit, eigene Aktionen zu definieren, die in den Kontextmenüs des Dateimanagers bei bestimmten Dateitypen erscheinen. Auf diese Weise lassen sich z. B. spezielle Betrachter oder Dateiformatkonverter komfortabel in die Desktopoberfläche einbinden.
- Thunar liefert zusätzlich das Tool Bulk-Rename mit, mit dem man nach bestimmten Mustern viele Dateien auf einmal umbenennen oder nummerieren kann. Besonders bei gro-

ßen Fotosammlungen mit von der Kamera automatisch vergebenen Dateinamen ist Bulk-Rename ausgesprochen nützlich.

4 Diashow mit Nokia Windows Phones und PhotoBeamer

Windows Phones von Nokia bieten eine ebenso innovative wie einfache Methode, Fotos vom Handy auf jedem beliebigen Computer anzuzeigen. Man benötigt keinerlei Anmeldung, keine Kabel- oder WLAN-Verbindung, und auch das Betriebssystem des verwendeten PCs ist völlig egal.

1 Mit dem Browser auf dem Raspberry Pi die Seite *www.photobeamer.com* besuchen. Diese Seite funktioniert problemlos mit dem Epiphany- und dem Midori-Browser (aber nicht in Dillo). Sie zeigt einen großformatigen QR-Code an.

2 Die PhotoBeamer-Anwendung auf dem Windows Phone starten und ein Foto auswählen. Die PhotoBeamer-Anwendung auf dem Windows Phone schaltet automatisch die Handykamera ein.

3 Das Windows Phone in Richtung Raspberry Pi-Bildschirm halten und diesen QR-Code scannen.

4 Nach wenigen Sekunden erscheint das ausgewählte Foto auf dem Raspberry Pi-Bildschirm. Jetzt auf dem Windows Phone durch die Fotoalben blättern. Die Bilder werden auf dem Raspberry Pi angezeigt.

5 Nach einer bestimmten Inaktivitätszeit wird die Verbindung automatisch getrennt.

5 GMX-MediaCenter auf dem Raspberry Pi nutzen

GMX bietet jedem Nutzer des kostenlosen Maildiensts 2 GByte Speicherplatz im sogenannten MediaCenter. Mit der Installation der Synchronisationsanwendung für Windows bekommt man noch 4 GByte dazugeschenkt.

Leider hat GMX, wie auch einige andere Cloud-Anbieter, ein Kompatibilitätsproblem und hält sich nicht zu 100 % an den WebDAV-Standard, weshalb die Cloud-Speicher nicht einfach über den Dateimanager von Raspbian eingebunden werden können. Die Lösung bietet ein Dateisystemtreiber, der WebDAV-Laufwerke wie externe Festplatten oder Netzlaufwerke im Linux-Dateisystem einhängt.

1. Dateisystemtreiber installieren: `sudo apt-get install davfs2`
2. Unterverzeichnis für den Mountpunkt im Home-Verzeichnis anlegen: `sudo mkdir /home/pi/gmx`
3. Unterverzeichnis für die Konfigurationsdatei im Home-Verzeichnis anlegen: `sudo mkdir /home/pi/.davfs2`
4. Konfigurationsdatei mit dem Namen `secrets` in diesem Verzeichnis erstellen: `leafpad /home/pi/.davfs2/secrets`
5. Die Datei enthält in einer einzigen Zeile die persönlichen Zugangsdaten für das GMX-MediaCenter, E-Mail-Adresse und Passwort:
   `https://webdav.mc.gmx.net emailadresse@gmx.de passwort`
6. Zugriffsrechte einschränken, damit der Dateisystemtreiber die Konfiguration akzeptiert: `chmod 600 /home/pi/.davfs2/secrets`
7. Das neue Dateisystem und den Mountpunkt in die Datei `/etc/fstab` eintragen: `sudo leafpad /etc/fstab`
8. Folgende Zeile am Ende neu hinzufügen: `https://webdav.mc.gmx.net /home/pi/gmx davfs noauto,user,rw 0 0`

9 Nicht-`root`-Benutzern die Berechtigung geben, WebDAV-Laufwerke einzubinden: `sudo dpkg-reconfigure davfs2`

10 Benutzer `pi` der Gruppe `davfs2` hinzufügen: `sudo adduser pi davfs2`

11 Das WebDAV-Laufwerk im zuvor angelegten Verzeichnis mounten: `sudo mount /home/pi/gmx`

Das GMX-MediaCenter steht nun im Dateimanager wie ein lokales Laufwerk zur Verfügung. Bei einem Neustart wird das GMX-MediaCenter automatisch wieder eingehängt.

6 Betriebssystem auf dem Raspberry Pi installieren

1 SD-Karte mit mindestens 8 GB an einen PC anschließen.

2 Noobs herunterladen unter ***http://www.raspberrypi.org/downloads/*** und entpacken.

3 Alle Dateien auf die formatierte SD-Karte kopieren

4 SD-Karte am Raspberry Pi anstecken und an die Stromversorgung anschließen

7 Python als Taschenrechner

Die auf dem Raspberry Pi vorinstallierte Programmiersprache Python, die für das Pi im Namen verantwortlich ist, lässt sich auch interaktiv verwenden.

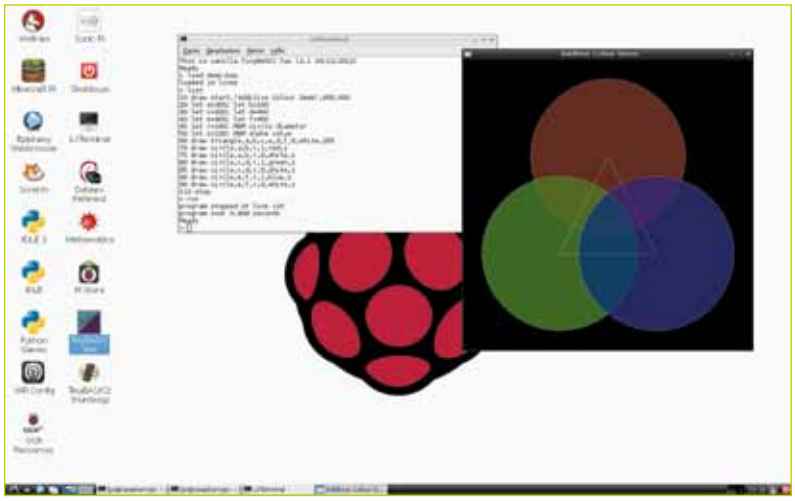
1 IDLE auf dem Desktop starten.

2 Berechnungsformel eingeben. Python rechnet automatisch »Punkt vor Strich«.

8 Programmieren in Basic

Basic wurde auf den ersten Home-Computern der 80er-Jahre als Standardsprache eingesetzt und kann natürlich auch heute noch in den Emulatoren dieser Computer auf dem Raspberry Pi verwendet werden.

TinyBASIC2 ist ein einfacher Basic-Interpreter, der Basic, wie man es von früher kennt, direkt unter Raspbian nutzbar macht. Im Sinne eines sauberen Programmcodes wurde der GOTO-Befehl abgeschaltet. Mit TinyBASIC2 lassen sich auf sehr einfache Weise Grafiken programmieren.



TinyBASIC installieren:

1 `sudo apt-get install libsdl-gfx1.2-4`

2 `wget http://www.staff.city.ac.uk/afl/tinybasic/tinybasic_2.1-1_armhf.deb`

3 `sudo dpkg -i tinybasic_2.1-1_armhf.deb`

4 `cp /usr/share/applications/tinybasic*.desktop ~/Desktop`

5 TinyBASIC2 per Klick auf das Desktopsymbol starten. Im Startmenü unter ***Entwicklung*** ist TinyBASI2 ebenfalls zu finden.

Humbug emuliert den Transam Triton von 1978

Zusätzlich zum normalen Modus kann TinyBASIC im Humbug-Modus genutzt werden und damit einen der ersten englischen Home-Computer mit seinem speziellen grafischen Zeichensatz und direktem Schreibzugriff auf den Bildschirmspeicher emulieren.

9 Objektorientiertes Basic Gambas

Bei Linux-Anwendern erfreut sich die Basic-Entwicklungsumgebung Gambas (*gambas.sourceforge.net/de/main.html*) großer Beliebtheit. Gambas arbeitet ähnlich wie Visual Basic objektorientiert. Damit lassen sich auf relativ einfache Weise grafische Oberflächen für Programme erstellen.

1  Gambas installieren:

```
sudo apt-get install gambas3
```

2 Gambas trägt sich nicht ins Startmenü ein, deshalb über ein Kommandozeilenfenster starten: `gambas3`

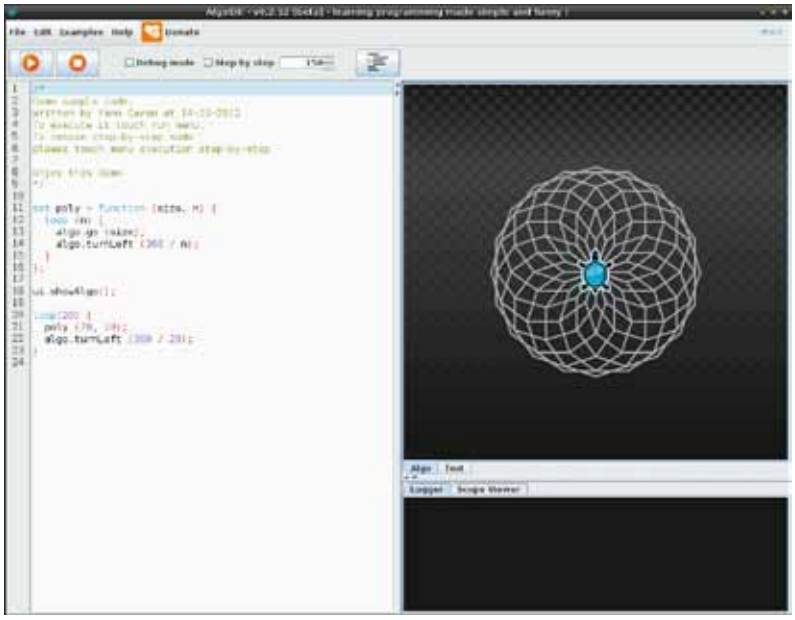
3 

Gambas benötigt einiges an Hardwareressourcen. Dementsprechend dauert bereits der Start einige Zeit.

Algoid ist eine einfache Programmiersprache speziell für Grafikanwendungen und Spiele. Algoid basiert auf der Idee der Turtle-Grafik, wie sie bereits vor vielen Jahren in der Programmiersprache Logo verwendet wurde.

10 Programmieren in Algoid

Algoid wird über den Pi Store zum Download angeboten und benötigt das Oracle Java7 JDK. Weitere Informationen unter *www.algoid.net*.



11 Programmieren in Lua

Lua ist eine einfache Skriptsprache zur Automatisierung von Aufgaben, die auch für Webanwendungen genutzt werden kann. Zur schnelleren Verarbeitung können Lua-Skripte in ausführbare Programme kompiliert werden.

1

Lua installieren: `sudo apt-get install lua5.1`

Lua wird unter anderem vom Cloud-Server BarracudaDrive verwendet, der auch auf dem Raspberry Pi lauffähig ist (siehe xxx).

Die diversen Tools für Lua sind am einfachsten über die Synaptic-Paketverwaltung zu finden. Weitere Informationen unter *www.lua.org* oder in der deutschsprachigen Dokumentation: *lua.coders-online.net*.

12 Raspberry Pi für Finanzmathematik

Cobol ist eine der allerältesten Programmiersprachen, wird aber auch heute noch für Spezialanwendungen im Bereich der Finanzmathematik verwendet. In der Version GNU Cobol (früher Open Cobol) ist es für diverse Plattformen wie auch für den Raspberry Pi erhältlich.

1 Open Cobol installieren: `sudo apt-get install open-cobol`

2 Open Cobol arbeitet auf Kommandozeilenebene. `cobc -h` zeigt eine ausführliche Liste aller möglichen Optionen.

Weitere Informationen unter *www.opencobol.org*.

13 Programmieren in Fortran

Fortran gilt als die erste höhere Programmiersprache. Sie wurde bereits in den 50er-Jahren veröffentlicht und wird seitdem bis heute entwickelt. In der Retroszene hat Fortran absoluten Kultstatus. Fortran kann mit allen bekannten Entwicklungsumgebungen wie Geany oder Eclipse genutzt werden. Es existieren Bibliotheken für alle denkbaren Aufgaben sowie diverse Compiler, die bestehende Fortran-Programme in moderne Programmiersprachen übersetzen. Raspbian beinhaltet den `gcc`-Compiler, für den es eine Fortran-95-Komponente gibt.

1 `sudo apt-get install gfortran gfortran-doc`

Weitere Informationen unter *www.fortran.de*.

14 Programmieren in .NET mit Mono

.NET und C# sind typische Sprachen für die Entwicklung von Windows-Programmen. Das Projekt Mono, eine quelloffene Implementierung des .NET Framework, ermöglicht die plattformübergreifende Entwicklung auch auf dem Raspberry Pi.

1 Mono-Runtime installieren:

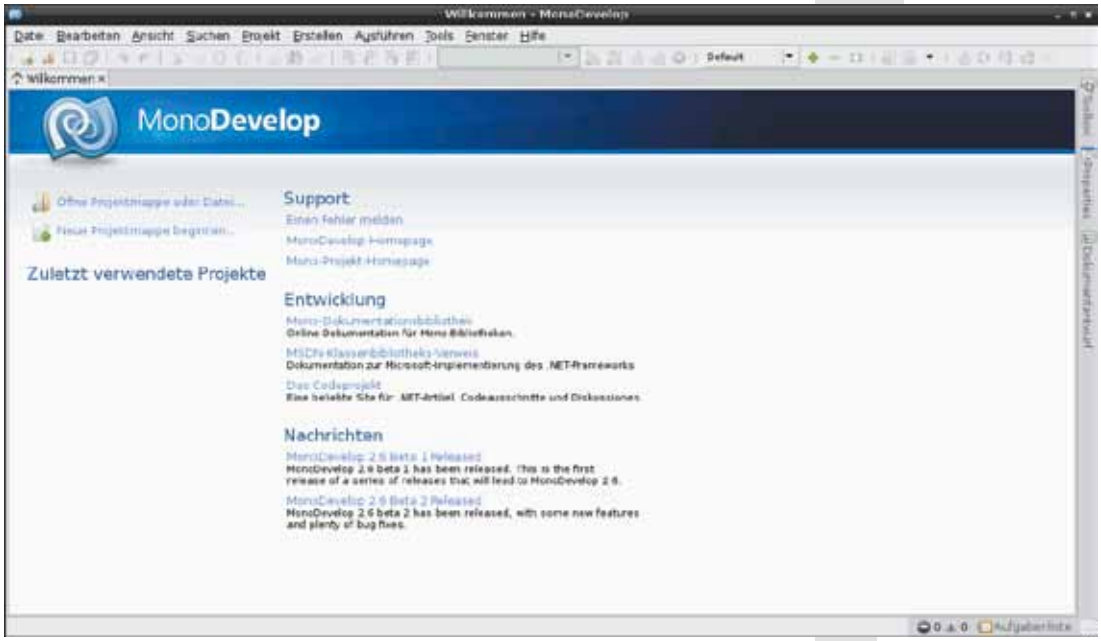
```
sudo apt-get install mono-runtime mono-csharp-shell
```

2 `csharp` startet die kommandozeilenbasierte Shell.

3 Zum komfortablen Entwickeln die MonoDevelop-IDE installieren:

```
sudo apt-get install monodevelop
```

4 MonoDevelop trägt sich im Startmenü unter ***Entwicklung*** ein. Dieses Paket ist über 200 MByte groß, benötigt erhebliche Hardwareressourcen und läuft bei größeren Projekten dementsprechend träge.



Weitere Informationen unter *www.mono-project.com*.

15 Programmieren in Assembler

Alle Programmiersprachen basieren auf menschlicher Logik und setzen diese in Maschinenbefehle für den Prozessor um. Assembler geht den umgekehrten Weg und definiert zu jedem Maschinenbefehl eine für Menschen annähernd lesbare Version. Echte Hardcore-Programmierer, die aus dem Prozessor das Letzte herausholen wollen, schreiben mit Assembler Programme, die der Prozessor direkt umsetzen kann. Raspbian enthält wie alle Linux-Distributionen den Kommandozeilenassembler `as`.

1  `as --help` zeigt alle Optionen und Parameter.

16 Vierzeiliges LCD-Display ansteuern

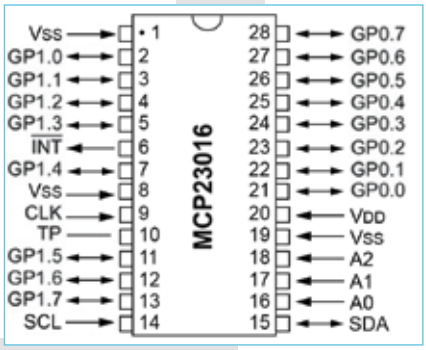
Vierzeilige LCD-Displays verwenden die gleichen 16 Anschlusspins wie zweizeilige. Dies gilt auch für Displays mit Zeilen länger als 16 Zeichen.

Um eine möglichst hohe Kompatibilität zum Standard zu erreichen, verwenden vierzeilige Displays intern auch nur zwei Zeilen. Die dritte Zeile des 20 bzw. 16 Zeichen breiten Displays entspricht der zweiten Hälfte einer 40 bzw. 32 Zeichen langen ersten Zeile. Genauso entspricht die vierte Zeile intern der zweiten Hälfte der zweiten Zeile.

Programmintern müssen also zwei lange Zeilen ausgegeben werden.

17 Der Portexpander MCP23016

Im Internet sind noch viele Schaltungen mit dem älteren Portexpander MCP23016 zu finden. Dieser unterscheidet sich von seinem Nachfolger im Wesentlichen durch eine unübersichtlichere Anordnung der GPIO-Pins.



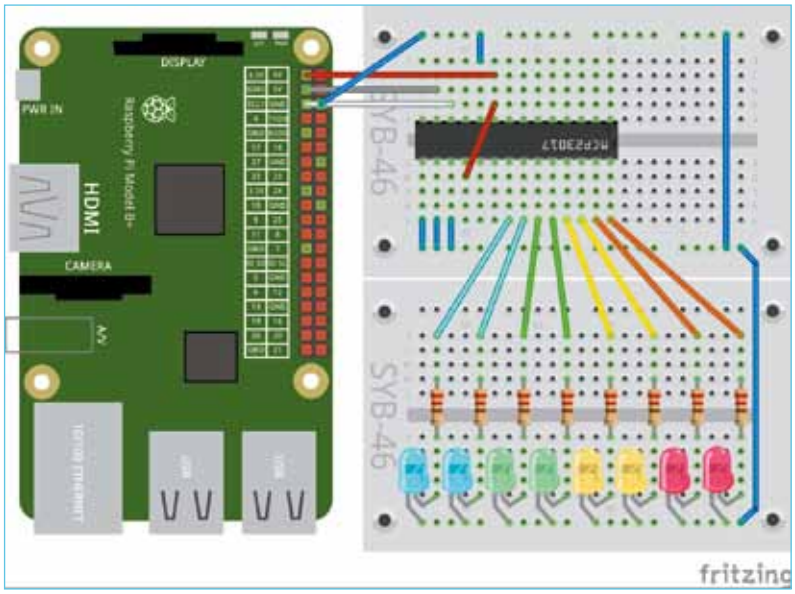
18 Lauflicht mit dem Portexpander (MCP23017)

1 Schaltung aus Portexpander und acht LEDs mit Vorwiderständen wie abgebildet aufbauen, dabei ist die Kerbe des MCP23017-Chips rechts.

2 Die farbigen Drahtbrücken verbinden die farblich passenden LEDs mit den acht Anschlüssen der `GPIOA`-Schnittstelle. Die `GPIOB`-Schnittstelle wird in dieser Schaltung nicht verwendet.

3 Alle blau dargestellten Drahtbrücken sind mit 0 V verbunden, die roten mit +3,3 V – hier zusätzlich zur Stromversorgung auch der RESET-Pin.

4 Die drei kurzen blauen Drahtbrücken verbinden die drei Adressleitungen mit 0 V und geben dem Portexpander damit die i2c-Adresse `0x20`.



Das Programm `i2c_01.py` lässt die LEDs als Lauflicht leuchten.

```
#!/usr/bin/python
import time, smbus

DEVICE = 0x20
IODIRA = 0x00
GPIOA  = 0x12

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)

while True:
  j = 1
  for i in range(8):
    bus.write_byte_data(DEVICE,GPIOA,j)
    j *= 2
    time.sleep(0.1)
```

Programme, die ausschließlich i2c nutzen, brauchen die GPIO-Schnittstelle nicht zu initialisieren. Es ist nicht einmal die RPi.GPIO-Bibliothek nötig. Das Programm muss auch nicht mit `root`-Berechtigung gestartet werden.

So funktioniert es

- Für alle Programme mit i2c muss die Bibliothek `smbus` importiert werden.

- Drei Hardwareadressen werden als Konstanten definiert.

`DEVICE = 0x20`	i2c-Geräteadresse des Portexpanders.
`IODIRA = 0x00`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOA-Ports festlegt.
`GPIOA  = 0x12`	8-Bit-Datenregister der GPIOA-Ports.

- Über ein Objekt namens `bus` vom Typ `smbus.SMBus()` wird der i2c-Bus mit allen angeschlossenen Geräten angesprochen.

```
bus = smbus.SMBus(1)
```

- `bus.write_byte_data()` schreibt ein Byte auf den i2c-Bus. Alle acht Bits eines der beiden GPIO-Ports müssen immer gleichzeitig angesteuert werden. Ein Byte kann binär, dezimal oder hexadezimal angegeben werden.

- Die nächste Zeile schreibt auf das in der Konstanten `DEVICE` definierte Gerät in alle Bits des Registers `IODIRA` den Wert `0`. Damit werden alle acht `GPIOA`-Ports als Ausgänge definiert. Steht ein Bit auf `1`, würde der entsprechende Port als Eingang definiert.

```
bus.write_byte_data(DEVICE,IODIRA,0x00)
```

- Danach wird in alle Bits des Datenregisters `GPIOA` der Wert `0` geschrieben. Damit werden alle acht `GPIOA`-Ports auf `0` gesetzt, die angeschlossenen LEDs also ausgeschaltet.

```
bus.write_byte_data(DEVICE,GPIOA,0x00)
```

- Für das Lauflicht werden nacheinander die acht LEDs einzeln eingeschaltet. Dazu muss das `GPIOA`-Datenregister nacheinander genau die Zahlenwerte annehmen, bei denen ein einzelnes Bit auf `1` steht. Dazu wird der Bytewert von 1 beginnend bei jedem Schritt verdoppelt, um die passende Zahl zu erhalten.

LED leuchtet	Binär	Dezimal	Hex
1	0000 0001	1	0x01
2	0000 0010	2	0x02
3	0000 0100	4	0x04
4	0000 1000	8	0x08
5	0001 0000	16	0x10
6	0010 0000	32	0x20
7	0100 0000	62	0x40
8	1000 0000	128	0x80

- Innerhalb der Hauptschleife des Programms läuft eine weitere Schleife je achtmal, wobei in jedem Durchlauf eine LED eingeschaltet wird. Dazu wird der aktuelle Wert `j` in das `GPIOA`-Datenregister geschrieben.

```
for i in range(8):
  bus.write_byte_data(DEVICE,GPIOA,j)
```

- In jedem Durchlauf wird der Wert `j` verdoppelt und damit das nächsthöhere Bit auf `1` gesetzt.

```
j *= 2
```

19 CPU-Lastanzeige mit Portexpander (MCP23017)

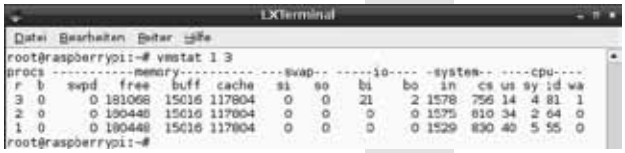
Mit den acht LEDs aus dem Lauflicht (siehe xxx) wird eine Pegelanzeige für die CPU-Last gebaut. Je nach CPU-Auslastung lässt das Programm `cpu.py` eine bis acht LEDs leuchten.

```
#!/usr/bin/python
import time, os, smbus

DEVICE = 0x20
IODIRA = 0x00
GPIOA  = 0x12

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)

while True:
    cpu1 = os.popen("vmstat 1 3").readlines()
    cpu2 = 2 ** int((100 - int(cpu1[4].split()[14])) * 0.08) - 1
    bus.write_byte_data(DEVICE,GPIOA,cpu2)
```

So funktioniert es

- Das Modul `os` zum Zugriff auf Systemfunktionen wird zusätzlich importiert.
- Das Linux-Kommando `vmstat 1 3` liest diverse Statusangaben des Systems aus. Der Parameter `3` bedeutet hier, dass dreimal hintereinander

die Statusdaten gelesen werden. Beim ersten Lesen ergeben sich verfälschte Informationen, da der Aufruf des Kommandos selbst einiges an Systemleistung frisst.



- Die Funktion `os.popen()` schreibt das Ergebnis des Kommandos zeilenweise in das Zeichenkettenarray `cpu1`.

```
cpu1 = os.popen("vmstat 1 3").readlines()
```

- Die nächste Zeile ermittelt den Wert, der mit den LEDs angezeigt werden soll, und speichert diesen in der Variablen `cpu2`. Die dritte Statusabfrage steht nach den beiden Tabellenüberschriften in der fünften Zeile und damit im fünften Element des Arrays, das bei Zählung ab `0` die Nummer `4` trägt. Diese Zeichenkette wird mit der Methode `split()` an den Leerzeichen in einzelne Zeichenketten aufgespalten. Interessant ist das Element `[14]`, das den Inhalt der Spalte `id` enthält, die die ungenutzte CPU-Zeit in Prozent angibt. Subtrahiert man diesen Wert von 100, erhält man die verbrauchte CPU-Zeit in Prozent, also die Auslastung der CPU. Zuvor wird die Zeichenkette mit der `int()`-Funktion in eine Ganzzahl umgewandelt. Eine Multiplikation mit dem Faktor `0,08` ergibt zur Darstellung auf den acht LEDs eine Zahl zwischen 0 und 8 statt zwischen 0 und 100.

```
cpu2 = 2 ** int((100 - int(cpu1[4].split()[14])) * 0.08) - 1
```

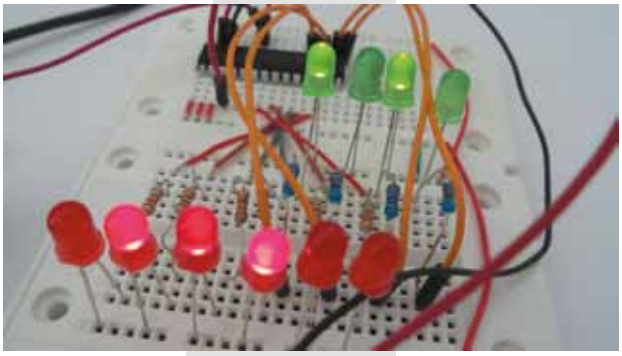
- Zur Darstellung über den Portexpander braucht man 8-Bit-Zahlen. In diesem Fall einer Pegelanzeige sollen LEDs in durchgehender Folge leuchten. Die Tabelle zeigt alle Zahlen zwischen 0 und 8, die in der Binärdarstellung aus einer durchgehenden Folge von Einsen bestehen. Über die Berechnungsfunktion in der rechten Spalte werden die Werte ermittelt, die binärcodiert die entsprechenden LEDs ansteuern. Das Symbol `**` steht in Python für die Exponentialfunktion.

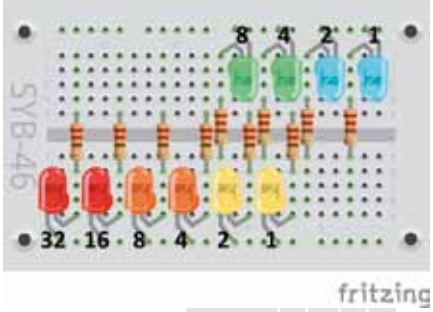
Binär	Dezimal	Berechnungsformel
0000 0000	0	2 ** 0 - 1
0000 0001	1	2 ** 1 - 1
0000 0011	3	2 ** 2 - 1
0000 0111	7	2 ** 3 - 1
0000 1111	15	2 ** 4 - 1
0001 1111	31	2 ** 5 - 1
0011 1111	63	2 ** 6 - 1
0111 1111	127	2 ** 7 - 1
1111 1111	255	2 ** 8 - 1

- Das in der Variablen `cpu2` gespeicherte Ergebnis wird auf den `GPIOA`-Port des Portexpanders geschrieben und mit den LEDs angezeigt. Danach startet die Endlosschleife neu und ermittelt wiederum die aktuelle CPU-Last.

```
bus.write_byte_data(DEVICE,GPIOA,cpu2)
```

20 Binäruhr mit Portexpander (MCP23017)



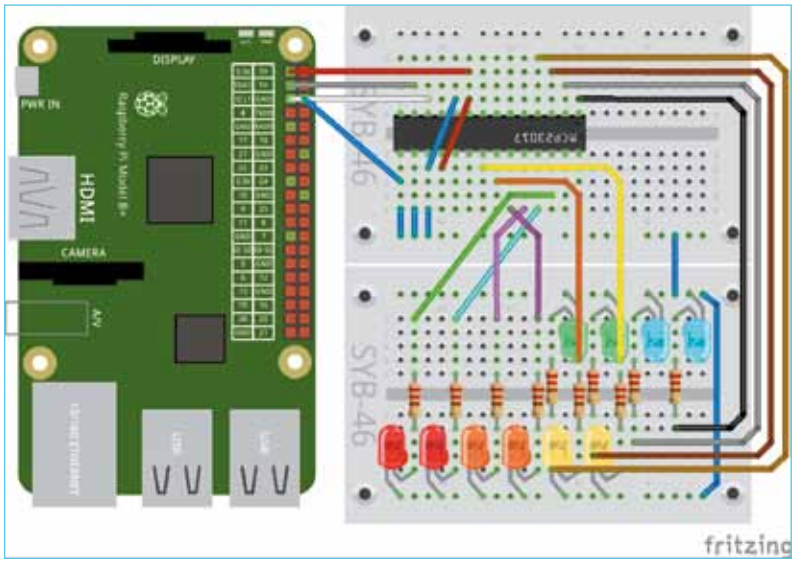


Binäre Uhren zeigen die Uhrzeit in binärcodierter Form anstatt in Ziffern an. Auf der abgebildeten Uhr ist es 10:28. Solche Uhren sehen einfach cool aus, obwohl oder gerade weil das Ablesen für Anfänger etwas gewöhnungsbedürftig ist. So bietet zum Beispiel der Uhrenhersteller »01 the one« (*www.01theone.com*) solche Binäruhren als Armbanduhren an.

Das Prinzip der Zeitdarstellung ist einfach. Jede LED steht für ein Bit der Binärzahl. Zur Darstellung der maximal zwölf Stunden werden vier LEDs benötigt (8, 4, 2, 1), zur Darstellung der maximal 59 Minuten sechs LEDs (32, 16, 8, 4, 2, 1). Man braucht nur die Werte der leuchtenden LEDs zusammenzuaddieren und die Stunden und Minuten im 12-Stunden-Format., dann hat man die Uhrzeit

Aus zehn LEDs mit Vorwiderständen und einem MCP23017-Portexpander lässt sich eine Binäruhr bauen.

Die Anschlussdrähte der LEDs verlaufen diesmal nicht parallel von den Vorwiderständen zu den Ports des Portexpanders, sondern genau über Kreuz. Dies hat den Vorteil, dass die LED ganz rechts mit dem niedrigsten Bitwert auch mit dem Port mit dem niedrigsten Bitwert verbunden ist. Die weiteren LEDs folgen mit aufsteigenden Bitwerten, was die Programmierung deutlich vereinfacht. Für Stunden und Minuten werden die beiden voneinander unabhängigen GPIO-Schnittstellen des MCP23017 verwendet.



LCD	Port am Portexpander
Stunde 8	GPIOB3
Stunde 4	GPIOB2
Stunde 2	GPIOB1
Stunde 1	GPIOB0
Minute 32	GPIOA5
Minute 16	GPIOA4
Minute 8	GPIOA3
Minute 4	GPIOA2
Minute 2	GPIOA1
Minute 1	GPIOA0

Das Programm `binaeruhr.py` ist sehr einfach, da der Portexpander von sich aus bereits mit binären Daten arbeitet.

```
#!/usr/bin/python
import time, smbus

DEVICE = 0x20
IODIRA = 0x00
IODIRB = 0x01
GPIOA  = 0x12
GPIOB  = 0x13

bus = smbus.SMBus(1)
bus.write_byte_data(DEVICE,IODIRA,0x00)
```

```
bus.write_byte_data(DEVICE,IODIRB,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)
bus.write_byte_data(DEVICE,GPIOB,0x00)

m1 = 60

while True:
  zeit = time.localtime()
  m = zeit.tm_min
  h = zeit.tm_hour
  if h > 12:
    h = h - 12
  if m1 <> m:
    bus.write_byte_data(DEVICE,GPIOB,h)
    bus.write_byte_data(DEVICE,GPIOA,m)
    m1 = m
  time.sleep(1)
```

So funktioniert es

- Für die Adressen beider GPIO-Schnittstellen werden Konstanten definiert, und diese werden anschließend initialisiert.

`DEVICE = 0x20`	i2c-Geräteadresse des Portexpanders.
`IODIRA = 0x00`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOA-Ports festlegt.
`IODIRB = 0x01`	8-Bit-Register, das die Richtung (Ausgang/Eingang) der GPIOB-Ports festlegt.
`GPIOA = 0x12`	8-Bit-Datenregister der GPIOA-Ports.
`GPIOB = 0x13`	8-Bit-Datenregister der GPIOB-Ports.

- In einer Endlosschleife wird die aktuelle Zeit ausgelesen, und die beiden für die Digitaluhr relevanten Werte, Minuten und Stunden, werden aus der Struktur in die Variablen `m` und `h` geschrieben.

```
while True:
  zeit = time.localtime()
  m = zeit.tm_min
  h = zeit.tm_hour
```

- Ist die Stundenangabe im 24-Stunden-Format größer als 12, wird 12 subtrahiert, um die Zeit im 12-Stunden-Format zu speichern.

- Hat die aktuelle Minute `m` einen anderen Wert als die zuletzt dargestellte Minute `m1`, wird die Stunde auf den Port `GPIOB` und die Minute auf den Port `GPIOA` geschrieben. Hier können direkt die Zahlenwerte verwendet werden, der Portexpander rechnet diese automatisch in Binärzahlen

um und gibt sie entsprechend auf den Ports aus. Danach wird `m1` auf die gerade dargestellte Minute gesetzt.

```
if m1 <> m:
  bus.write_byte_data(DEVICE,GPIOB,h)
  bus.write_byte_data(DEVICE,GPIOA,m)
  m1 = m
```

- Vor dem ersten Start der Schleife bekommt `m1` den Wert `60`, den die Minutenanzeige im laufenden Betrieb nie erreicht. Damit wird sichergestellt, dass unabhängig von der aktuellen Uhrzeit bereits im ersten Schleifendurchlauf eine neue Zeit ermittelt und über die LEDs angezeigt wird.

- Am Ende der Schleife wartet das Programm eine Sekunde. Damit wird verhindert, dass die Schleife in Intervallen von Sekundenbruchteilen immer wieder aufgerufen wird und den Prozessor so stark auslastet, dass der Raspberry Pi für nichts anderes als dieses Programm zu nutzen ist.

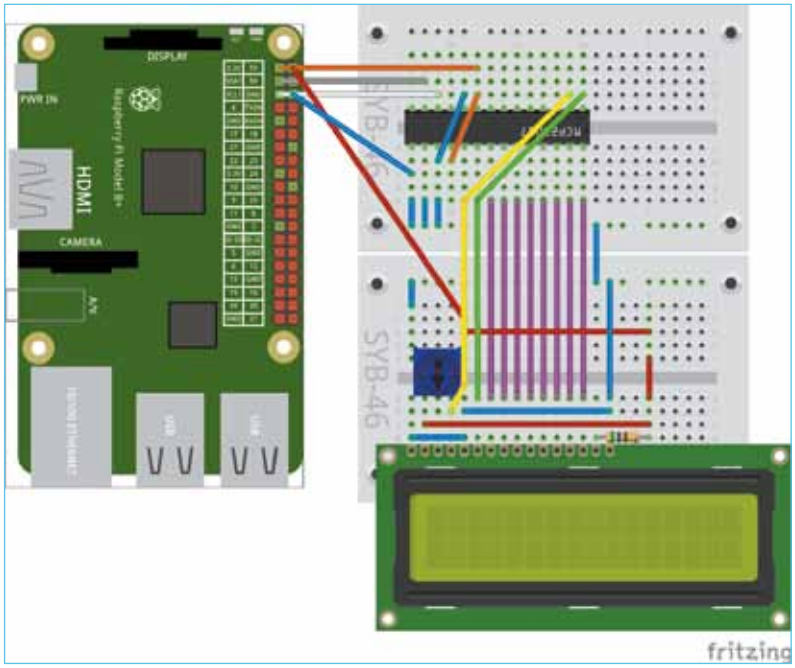
21 i2c-LCD-Displays (MCP23017) anschließen

LCD-Display und Portexpander verarbeiten beide bitcodierte Daten mit bis zu 8 Bit Breite. Ein LCD-Display am Portexpander benötigt am Raspberry Pi nur noch den i2c-Bus und nicht mehr sechs GPIO-Port. Ganz nebenbei wird das Programm auch noch einfacher.

Für den Anschluss am Portexpander empfiehlt sich der 8-Bit-Modus des LCD-Displays, da hier die Zeichen byteweise anstatt in zwei Blöcken auf das Display geschrieben werden können, was das Programm deutlich vereinfacht. Die acht Datenleitungen des `GPIOA`-Ports werden eins zu eins mit den Datenleitungen des LCD-Displays verbunden. Zwei der Leitungen des `GPIOB`-Ports werden für die Steuersignale ***RS*** und ***E*** verwendet.

LCD	Port am Portexpander
D0	GPIOA 0
D1	GPIOA 1
D2	GPIOA 2
D3	GPIOA 3
D4	GPIOA 4
D5	GPIOA 5
D6	GPIOA 6
D7	GPIOA 7

E	GPIOB 0
RS	GPIOB 1



Das Programm `i2c_display08.py` übernimmt die grundlegende Struktur der Statusanzeige.

```
#!/usr/bin/python
import time, smbus, subprocess
bus = smbus.SMBus(1)

DEVICE = 0x20
IODIRA = 0x00
IODIRB = 0x01
GPIOA  = 0x12
GPIOB  = 0x13

LCD_WIDTH  = 16
LCD_LINE_1 = 0x80
LCD_LINE_2 = 0xC0
LCD_CHR = 1
LCD_CMD = 0
LCD_RS  = 0x02
LCD_E   = 0x01
E_PULSE = 0.00005
E_DELAY = 0.00005
```

```
pause = 2

def lcd_byte(bits, mode):
  bus.write_byte_data(DEVICE,GPIOB,LCD_RS*mode)
  bus.write_byte_data(DEVICE,GPIOA,bits)
  time.sleep(E_DELAY)
  bus.write_byte_data(DEVICE,GPIOB,LCD_E+LCD_RS*mode)
  time.sleep(E_PULSE)
  bus.write_byte_data(DEVICE,GPIOB,LCD_RS*mode)
  time.sleep(E_DELAY)

def lcd_string(message):
  message = message.ljust(LCD_WIDTH," ")
  for i in range(LCD_WIDTH):
    lcd_byte(ord(message[i]),LCD_CHR)

def lcd_anzeige(z1, z2):
  lcd_byte(LCD_LINE_1, LCD_CMD)
  lcd_string(z1)
  lcd_byte(LCD_LINE_2, LCD_CMD)
  lcd_string(z2)

bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,IODIRB,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)
bus.write_byte_data(DEVICE,GPIOB,0x00)

LCD_INIT = [0x33, 0x32, 0x38, 0x0C, 0x06, 0x01]
for i in LCD_INIT:
  lcd_byte(i,LCD_CMD)

while True:
  zeile1 = time.asctime()
  zeile2 = „IP:“ + subprocess.check_output([„hostname“,“-I“])
  lcd_anzeige(zeile1, zeile2)
  time.sleep(pause)
```

So funktioniert es

- Die Konstanten für das LCD-Display werden weitestgehend übernommen, um die Änderungen am Programmcode so gering wie möglich zu halten. Die beiden Signale `LCD_RS` (Register Select) und `LCE_E` (Enable) sind jetzt keine GPIO-Pins mehr, sondern Bitcodes, die über den `GPIOB`-Port an die entsprechenden Steuerleitungen des LCD-Displays gesendet werden.

```
LCD_RS = 0x02
LCD_E  = 0x01
```

- Der Funktionsaufruf `lcd_byte(bits, mode)` bleibt kompatibel zum ursprünglichen Programm, damit braucht das Hauptprogramm nicht verändert zu werden.

- Zur Modusumschaltung wird das Bitmuster für das zweite Bit des `GPIOB`-Ports (`0x02`), das in der Konstanten `LCD_RS` gespeichert ist, mit dem Parameter `mode` (0 oder 1) multipliziert. Auf diese Weise wird dieses Bit für den Kommandomodus ausgeschaltet und für den Zeichenmodus eingeschaltet.

```
bus.write_byte_data(DEVICE,GPIOB,LCD_RS*mode)
```

- Die nächste Zeile schreibt das an das Display zu sendende Byte auf den `GPIOA`-Port. Bei Verwendung des Portexpanders entfällt die Umrechnung des Zeichens in Bitcode.

```
bus.write_byte_data(DEVICE,GPIOA,bits)
```

- Jetzt braucht das Display das Enable-Signal, einen Wechsel von `1` auf `0` auf der `E`-Leitung, um die Daten auf den Datenleitungen einzulesen und darzustellen. Nach einer kurzen Verzögerung, um die Displayträgheit zu überbrücken, wird das in der Konstanten `LCD_E` festgelegte erste Bit des `GPIOB`-Ports auf `1` gesetzt. Da auf dem Portexpander immer ein komplettes Byte auf einmal ausgegeben werden muss und der zuvor gesetzte Modus nicht verändert werden darf, wird dieser Modus zum Wert `LCD_E` addiert und so wieder mitgesendet.

```
time.sleep(E_DELAY)
bus.write_byte_data(DEVICE,GPIOB,LCD_E+LCD_RS*mode)
```

- Nach einer weiteren kurzen Verzögerung wird das Bit `LCD_E` wieder auf `0` gesetzt. Dazu sendet die Funktion einfach nur den aktuellen Modus `0x02` oder `0x00` an das Display. Dabei ist das Bit `LCD_E` automatisch `0`.

```
time.sleep(E_PULSE)
bus.write_byte_data(DEVICE,GPIOB,LCD_RS*mode)
```

- Die Funktionen `lcd_string()` und `lcd_anzeige()` greifen nicht direkt auf die Hardware des LCD-Displays zu und können unverändert übernommen werden.

- Nach der Definition der Funktionen beginnt das eigentliche Programm mit der Initialisierung der beiden Ports des Portexpanders, `GPIOA` und `GPIOB`.

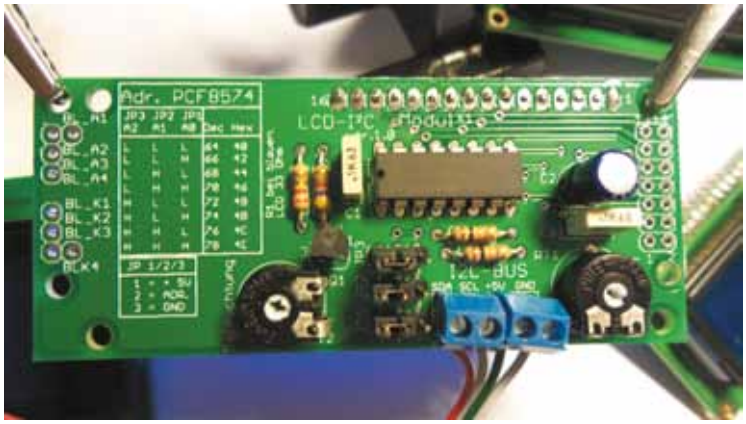
```
bus.write_byte_data(DEVICE,IODIRA,0x00)
bus.write_byte_data(DEVICE,IODIRB,0x00)
bus.write_byte_data(DEVICE,GPIOA,0x00)
bus.write_byte_data(DEVICE,GPIOB,0x00)
```

- Der Initialisierungsstring für das Display enthält diesmal den Code `0x38` für den 8-Bit-Modus.

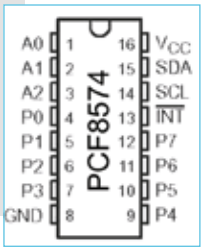
```
LCD_INIT = [0x33, 0x32, 0x38, 0x0C, 0x06, 0x01]
```

- Alle weiteren Programmkomponenten werden unverändert übernommen.

22 Der Portexpander PCF8574/PCF8574A



Der Chip PCF8574 ist ein besonders einfacher und kostengünstiger Portexpander, der acht Ports zur Verfügung stellt und in vielen vorkonfigurierten i2c-Displays eingebaut ist.



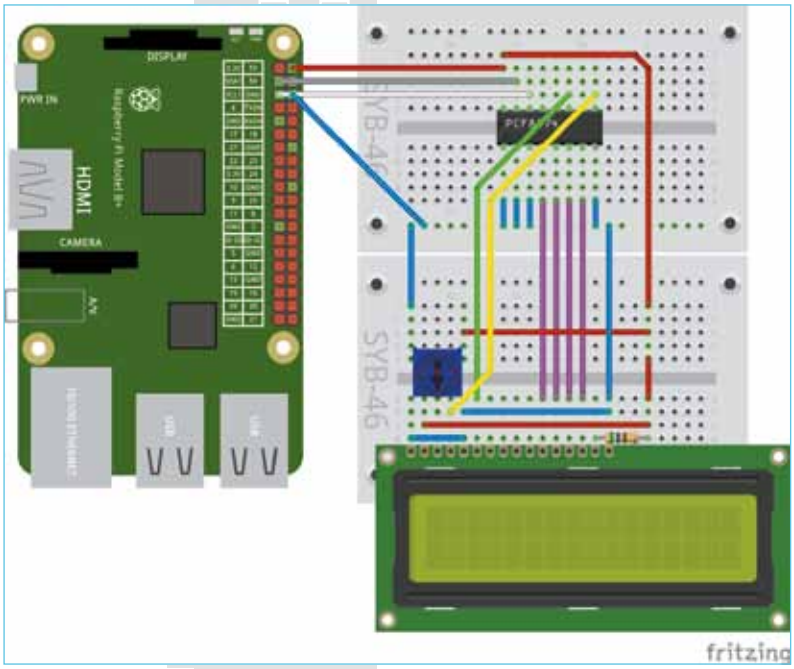
- Die Anschlüsse `P0...P7` sind eine 8 Bit breite GPIO-Schnittstelle, die binärcodiert angesteuert wird.

- Die Anschlüsse `VCC` und `GND` sind die Stromversorgung für den Portexpander. `VCC` kann an +3,3 V oder +5 V angeschlossen werden, damit kann die gleiche Stromversorgung auch für das Display genutzt werden. `GND` wird an 0 V angeschlossen.

- Die Anschlüsse `A0`, `A1`, `A2` legen die Adresse des Portexpanders fest. An den drei Anschlüssen lassen sich mit `HIGH`- oder `LOW`-Signalen binärcodiert die Zahlen von 0 bis 7 darstellen. Es gibt zwei Versionen des Chips, die sich nur in den Adressen unterscheiden:

Chip	Adressschema	Mögliche Adressen
PCF8574	0010 0A2A1A0	0x20 ... 0x27
PCF8574A	0011 1A2A1A0	0x38 ... 0x3F

Für die Programmierung ist es wichtig, das interne Anschlussschema zwischen LCD-Display und Portexpander zu kennen. Das nächste Programm verwendet ein weitverbreitetes Anschlussschema vorgefertigter Displaymodule:



LCD	Port am Portexpander
D4	P0
D5	P1
D6	P2
D7	P3
E	P6
RS	P4

23 i2c-LCD-Displays (PCF8574) ansteuern

Das Programm `i2c_display08.py` zeigt die gleichen Informationen wie das vorherige Programm auf einem Display an, das über den Portexpander PCF8574 gesteuert wird.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import time, smbus, subprocess
bus = smbus.SMBus(1)
```

```
DEVICE     = 0x27
LCD_WIDTH  = 16
LCD_LINE_1 = 0x80
LCD_LINE_2 = 0xC0
LCD_CHR = 1
LCD_CMD = 0
LCD_RS  = 0x10
LCD_E   = 0x40
E_PULSE = 0.00005
E_DELAY = 0.00005
pause = 2

def lcd_byte(bits, mode):
  bus.write_byte(DEVICE,LCD_RS*mode+(bits>>4)+LCD_E)
  time.sleep(E_PULSE)
  bus.write_byte(DEVICE,LCD_RS*mode)
  time.sleep(E_DELAY)
  bus.write_byte(DEVICE,LCD_RS*mode+(bits&0x0F)+LCD_E)
  time.sleep(E_PULSE)
  bus.write_byte(DEVICE,LCD_RS*mode)
  time.sleep(E_DELAY)

def lcd_string(message):
  message = message.ljust(LCD_WIDTH," ")
  for i in range(LCD_WIDTH):
    lcd_byte(ord(message[i]),LCD_CHR)

def lcd_anzeige(z1, z2):
  lcd_byte(LCD_LINE_1, LCD_CMD)
  lcd_string(z1)
  lcd_byte(LCD_LINE_2, LCD_CMD)
  lcd_string(z2)

LCD_INIT = [0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]
for i in LCD_INIT:
  lcd_byte(i,LCD_CMD)

while True:
  zeile1 = time.asctime()
  zeile2 = „IP:“ + subprocess.check_output([„hostname“,“-I“])
  lcd_anzeige(zeile1, zeile2)
  time.sleep(pause)
```

So funktioniert es

- Bei den Definitionen ist nur noch die Geräteadresse nötig, keine Adressen für zwei GPIO-Ports. `LCD_RS` und `LCD_E` enthalten die Bitmuster für die beiden Steuersignale. Je nach Schaltplan müssen diese entsprechend geändert werden.

- In der Funktion `lcd_byte()` werden Daten im 4-Bit-Modus übertragen. Dazu werden die entsprechenden Steuersignale addiert, und alles zusammen wird in zwei Blöcken auf dem 8-Bit-GPIO-Port übertragen. Im ersten Block werden die höheren vier Bits mit Rightshift (`>>`) auf die unteren vier Stellen verschoben. Im zweiten Block werden die oberen vier Bits durch bitweises UND (`&`) mit `0x0F` auf `0` gesetzt. In dem Moment, in dem `LCD_E` von `1` auf `0` wechselt, werden die zuvor auf den Datenregistern vorhandenen Bits auf das Display geschrieben. Die Tabelle zeigt die Bitmuster in der Übersicht:

Zeichen, Beispiel: „C“	0100 0011
LCD_RS	0001 0000
LCD_E	0100 0000
LCD_RS*LCD_CHR+(bits>>4)+LCD_E	0101 0100
LCD_RS*LCD_CHR+(bits&0x0F)+LCD_E	0101 0011
LCD_RS*LCD_CHR	0001 0000

- Der Initialisierungsstring setzt den 4-Bit-Modus auf dem Display:

```
LCD_INIT = [0x33, 0x32, 0x28, 0x0C, 0x06, 0x01]
```

24 Das BerryClip-Demoprogramm

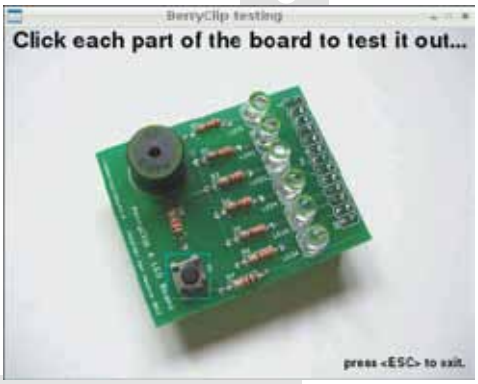
Der Hersteller des BerryClip liefert ein grafisches Demo- und Testprogramm auf Basis von Python und PyGame.

1 Programm installieren:

```
wget https://bitbucket.org/vinnievg/berryclip/raw/
default/6LED/testclip/testclip.py
wget https://bitbucket.org/vinnievg/berryclip/raw/
default/6LED/testclip/data/berryClip.JPG
mkdir data
mv berryClip.JPG ~/data
```

2 Programm starten: `sudo python testclip.py`

3 Ein Klick auf eine LED lässt diese leuchten, ein zweiter Klick schaltet sie wieder aus. Anstatt zu klicken, kann man auch eine der Zifferntasten [1] bis [6] drücken.



25 GPIO aus der Linux-Shell nutzen

GPIO-Ports können auch ohne Python direkt aus der Linux-Shell angesprochen werden. Das Skript `berryclip_test.sh` lässt alle sechs LEDs auf dem BerryClip für fünf Sekunden aufleuchten. Auch auf Kommandozeilenebene braucht man `root`-Rechte zum Zugriff auf die GPIO-Schnittstelle.

```
#!/bin/bash

echo “4”  > /sys/class/gpio/export
echo “17” > /sys/class/gpio/export
echo “22” > /sys/class/gpio/export
echo “10” > /sys/class/gpio/export
echo “9”  > /sys/class/gpio/export
echo “11” > /sys/class/gpio/export
echo “out” > /sys/class/gpio/gpio4/direction
echo “out” > /sys/class/gpio/gpio17/direction
echo “out” > /sys/class/gpio/gpio22/direction
echo “out” > /sys/class/gpio/gpio10/direction
echo “out” > /sys/class/gpio/gpio9/direction
echo “out” > /sys/class/gpio/gpio11/direction
echo “1” > /sys/class/gpio/gpio4/value
echo “1” > /sys/class/gpio/gpio17/value
echo “1” > /sys/class/gpio/gpio22/value
echo “1” > /sys/class/gpio/gpio10/value
echo “1” > /sys/class/gpio/gpio9/value
echo “1” > /sys/class/gpio/gpio11/value
sleep 5.0
echo “0” > /sys/class/gpio/gpio4/value
echo “0” > /sys/class/gpio/gpio17/value
echo “0” > /sys/class/gpio/gpio22/value
```

```
echo “0” > /sys/class/gpio/gpio10/value
echo “0” > /sys/class/gpio/gpio9/value
echo “0” > /sys/class/gpio/gpio11/value
echo “4”  > /sys/class/gpio/unexport
echo “17” > /sys/class/gpio/unexport
echo “22” > /sys/class/gpio/unexport
echo “10” > /sys/class/gpio/unexport
echo “9”  > /sys/class/gpio/unexport
echo “11” > /sys/class/gpio/unexport
```

So funktioniert es

- Die Ports werden initialisiert und entsprechende Verzeichnisse in der Linux-Verzeichnisstruktur angelegt.

```
echo „4“  > /sys/class/gpio/export
echo “17” > /sys/class/gpio/export
...
```

- Die Ports werden über die Datei `direction` als Ausgänge definiert.

```
echo „out“ > /sys/class/gpio/gpio4/direction
...
```

- Der Wert `1` schaltet die Ausgänge auf `HIGH` und die angeschlossenen LEDs ein.

```
echo „1“ > /sys/class/gpio/gpio4/value
...
```

- Der Wert `0` schaltet die Ausgänge auf `LOW` und die angeschlossenen LEDs aus.

```
echo „0“ > /sys/class/gpio/gpio4/value
...
```

- Wenn man einen GPIO-Pin vorläufig nicht mehr verwenden möchte, sollte man ihn wieder deaktivieren, da es zu Fehlermeldungen kommt, wenn ein bereits aktiver Pin z. B. durch ein Skript nochmals aktiviert wird:

```
echo „4“  > /sys/class/gpio/unexport
...
```

26 Erweiterungsplatine PiFace digital

PiFace Digital ist eine Erweiterungsplatine, die den Raspberry Pi auf einfache Weise mit der Außenwelt verbindet. Die Platine bietet je acht digitale Ein- und Ausgänge, acht LEDs, vier Taster sowie zwei Relais zum Schalten stärkerer Stromverbraucher. Alle Anschlüsse verwenden Schraubklemmen, sodass kein Löten nötig ist.

Das PiFace Digital nutzt die GPIO-Ports nicht direkt, sondern über einen MCP23S17-Port-expander-Chip, der über eine i2c-Schnittstelle mit dem Raspberry Pi kommuniziert. Auf diese Weise werden nur zwei GPIO-Pins benötigt. Der Hersteller liefert bei *www.piface.org.uk* Funktionsbibliotheken zur einfachen Ansteuerung der Platine in Python, C und Scratch zum Download. Bei *github.com/piface/pifacedigital-emulator* wird ein Emulator für das PiFace Digital angeboten.

27 Der Pi-LITE-Emulator

Der Hersteller des Pi-LITE liefert mit den Programmierbeispielen auch einen Emulator, mit dem sich Laufschriften und Steuerungsbefehle auf dem Raspberry Pi testen lassen, ohne dass die Pi-LITE-Platine angeschlossen sein muss.

1 Das Skript `~/git/PiLite/Python_Examples/PiLiteEmulator.py` aus der Python-IDE oder per Doppelklick aus dem Dateimanager starten. Keine Superuserrechte nötig.

2 Im Feld ***Command Entry*** einen Text eingeben, dieser wird im Emulator angezeigt.

3 Die Zeichenfolge `$$$` schaltet das Pi-LITE (und auch den Emulator) in den Kommandomodus um. Jetzt können die unter *bit.ly/pilite03* beschriebenen Befehle genutzt werden.

28 Erweiterungsplatine Gertboard

Das Gertboard ist eine der umfangreichsten Erweiterungsplatinen und größer als der Raspberry Pi selbst – mit zwölf gepufferten Ausgängen, LEDs und drei Tastern. Dazu sind ein Digital-Analog-Wandler, ein Analog-Digital-Wandler sowie ein Motorcontroller auf der Platine verbaut. Zur Steuerung gibt es einen ATmega-Mikrocontroller wie auf dem bekannten Arduino, der auch über die Arduino-IDE gesteuert wird.

29 Erweiterungsplatine GertDuino

Gert van Loo, der Erfinder des Gertboards, entwickelte Ende des Jahres 2013 eine neuartige Platine. Der GertDuino bringt die beiden Welten von Raspberry Pi und Arduino zusammen. GertDuino ist eine Arduino-Uno-kompatible Platine mit ATmega-328-Chip, zwei Tasten und sechs LEDs, die auf den GPIO-Port des Raspberry Pi gesteckt wird. Beide Geräte arbeiten eigenständig und kommunizieren nur über den GPIO-Port miteinander.

Das GertDuino-Board verfügt über Arduino-kompatible Steckanschlüsse, auf denen die zahlreichen verfügbaren Shields genutzt werden können. Die Kombination ermöglicht es, Sensordaten mit dem Arduino zu erfassen und auf dem Raspberry Pi mit entsprechender Software auszuwerten. Weiterhin lässt sich der Arduino direkt vom Raspberry Pi programmieren, ohne dass ein PC benötigt wird.