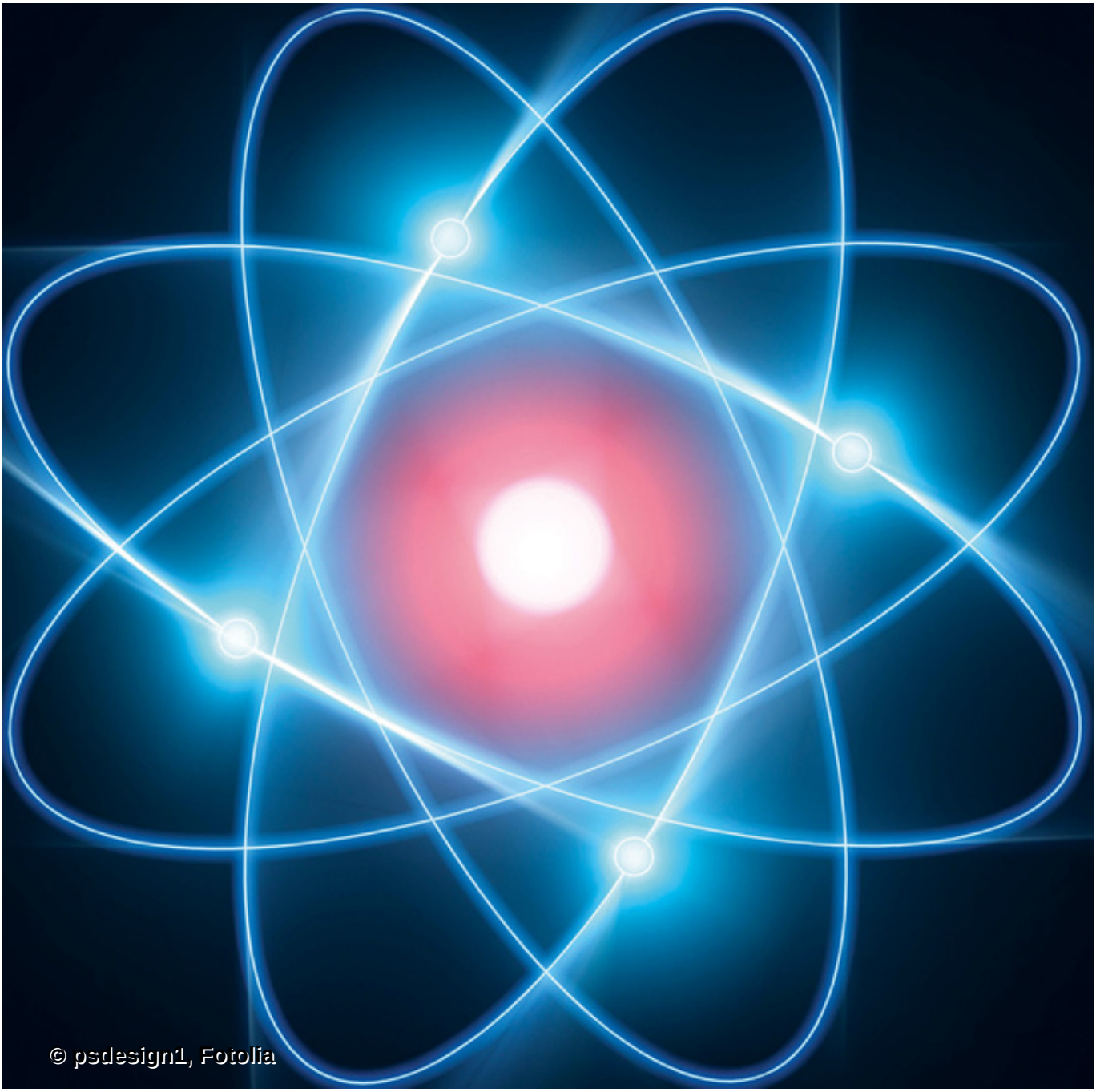
Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 101

Kern-Technik

Viele Applikationen, die Sensoren und Aktoren über GPIOs ansteuern, benutzen ein längst abgekündigtes Interface. Das ist ein bisschen verrückt, weil das moderne Kernel-Subsystem Gpiolib und die passende Userland-Bibliothek Libgpiod viel schneller und sicherer arbeiten. *Eva-Katharina Kunst*, *Jürgen Quade*



Neben dem günstigen Preis, seinem offenen Konzept und einer großen Community machen auch die einfachen Anschlussmöglichkeiten die Himbeer-Plattform Raspberry Pi zum Leckerbissen. Man muss kein Embedded-Spezialist sein, um an den GPIOs (siehe

Kasten "General Purpose Input Output") mit wenig Aufwand LEDs, Taster, Schalter, Sensoren oder Motoren anzuschließen. Den Software-technischen Zugriff haben dann Bastler, Maker und Kernelentwickler so vereinfacht, dass es keines Informatik-Abschlusses bedarf, um Temperaturwerte einzulesen oder LEDs und Motoren präzise zu steuern.

Die Python-Programmierer nehmen hierzu gerne die Wiring-Pi-Bibliothek [1] zur Hand, die ihnen das Ansteuern à la Arduino-Interface öffnet. Allerdings ist diese Art des Zugriffs für ein Multithreading- und Multiuser-Betriebssystem nicht optimal, geschweige denn professionell. Die überdies verwirrende Wiring-Pi-Nummerierung der GPIOs versalzt Programmierern das Himbeertörtchen vollends.

General Purpose Input Output

Als ein GPIO bezeichnet der Elektroniker eine über einen IC angeschlossene elektrische Leitung, die eine programmierbare Funktionalität annimmt. Er kann sie klassisch als Eingang oder als Ausgang bestimmen. Die Leitungen, die ein GPIO-Chip bereistellt, sind nummeriert. Von den 54 GPIOs des Raspberry Pi sind 26 auf eine Steckerleiste geführt. Die Pins der Steckerleiste tragen ebenfalls Nummern. Der im Artikel verwendete GPIO-17 befindet sich auf Pin 11 der Steckerleiste, GPIO-18 auf Pin 12.

Falls der Port als Eingang konfiguriert ist, liest der Computer beim Abarbeiten eines zugehörigen Maschinenbefehls eine »1« in ein internes CPU-Register, sofern an der Leitung eine Spannung anliegt. Liegt keine Spannung an, liest er eine »0«. Manche GPIOs gestatten es, sie mit umgekehrter Logik zu programmieren. Eine derartig als Active low konfigurierte Leitung liefert eine »1«, wenn keine Spannung anliegt, und »0«, falls beim Raspberry Pi mindestens 1,6 Volt anliegen.

Als Ausgang konfiguriert setzt die CPU des Microcontrollers die Leitung auf eine definierte Spannung (beim Raspberry Pi beispielsweise 3,3 Volt) oder auf keine Spannung (0 Volt).

Spezielle Anwendungen

Einige GPIO-Leitungen eignen sich nicht nur als Standard-Ein- oder -Ausgänge, sondern können besondere Eigenschaften annehmen, beispielsweise als Teil einer seriellen Schnittstelle oder eines I^2C-Busses fungieren. Oder der Entwickler programmiert per gesetztem Flag einen (dafür geeigneten) GPIO-Chip so, dass sich die elektrischen Eigenschaften des Ausgangs ändern – oft auf Open Drain oder auf Open Source. Das sind offene Ausgänge, die zwischen hochohmig und einem durch die äußere Beschaltung bestimmten elektrischen Zustand wechseln.

Zum alten Eisen

Kern-Technik 70 [2] hatte bereits vor fünf Jahren vorgestellt, wie Linux professionell im Rahmen eines Gerätetreibers auf GPIOs zugreift. Im Fazit der damaligen Ausgabe findet sich der Hinweis, dass die vorgestellte Lösung noch nicht das Nonplusultra ist und die Entwickler es wohl – Linux-like in absehbarer Zeit – durch etwas Besseres ablösen werden.

Und tatsächlich: Heute ist das Sys-FS-Interface zu GPIOs längst als "obsolete" eingestuft, und die vor fünf Jahren hochaktuellen Zugriffsfunktionen im Kernel sind "deprecated" – veraltet und abgekündigt. Im Kernel steuert seit geraumer Zeit das Gpiolib getaufte Subsystem die Hardware. Die alten Interfaces sind on top aufgesetzt, um die Kompatibilität zu früher zu gewährleisten. Wer schneller, kompatibler und vor allem sicherer unterwegs sein will, wählt das neue Interface.

Die Vorteile des neuen Systems zeigen sich bereits bei der Adressierung einer GPIO-Leitung, die jetzt zweistufig erfolgt. Das alte Interface dagegen kennt nur eine einzige Nummer, die bei 0 oder 1 beginnend die vorhandenen GPIO-Ports durchzählt. Das wird dann zum Problem, wenn auf einer Platine mehrere Controller-Bausteine sitzen – im Linux-Kernel sind sie als »gpiochip« bezeichnet.

Der Raspberry Pi 3 etwa besitzt mehr als einen GPIO-Baustein, was im alten System nur mit Tricksen (Zählung über den ersten Chip hinweg aufsteigend) beherrschbar blieb. Im aktuellen Interface dagegen gibt der Programmierer realitätsnah den Chip und dann die Leitung an.

Beim Namen nennen

Das neue Subsystem favorisiert darüber hinaus ein generisches Interface, das die zum Chip gehörige GPIO-Leitung über einen Namen anspricht. Der Name selbst lässt sich zum Beispiel im Devicetree angeben und damit beim Booten zuordnen. Zudem kann er eine Funktionalität referenzieren. So kann der Benutzer beispielsweise die Leitungen namenstechnisch bündeln, die für die serielle Schnittstelle zuständig sind, oder bei entsprechender Auslegung des GPIO-Chips einen I^2C-Bus realisieren. Der eigene Gerätetreiber für ein I^2C-Gerät läuft dann bei entsprechender Konfiguration ohne Codeänderung auf einer anderen Plattform. Das Interface unterstützt es sogar, dass sich ein Gerätetreiber vom Gpiolib-Subsystem eine GPIO-Leitung passender Funktionalität zuweisen lässt.

Neben der modernen Adressierung eines GPIO-Ports ist der neue Zugriffsschutz vorteilhaft. Er verhindert parallele Zugriffe auf eine Leitung, indem er diese mit dem zugreifenden Rechenprozess assoziiert und reserviert. Allerdings schränkt genau dies die Möglichkeiten ein: Das überaus bequeme Lesen einer GPIO-Leitung per »cat« oder das Setzen per »echo« sind damit passé.

Das GPIO-Subsystem ist jetzt über die Gerätedateien »/dev/gpiochip*Nummer_des_GPIO-Chips*« erreichbar und hier im Wesentlichen über einigermaßen hässliche IO-Controls, deren Namen und Parametrierungen mangels anderer Dokumentation im Linux-Quellcode nachzulesen sind (»include/uapi/linux/gpio.h«).

Ebenfalls im Quellcode liegen im Unterverzeichnis »tools/gpio/« drei Beispieltools, die GPIO-relevante Informationen aus dem Kernel kitzeln. Dass das Trio in keiner Linux-Distribution angekommen ist, wundert nicht: Denn mit ihm gelingt weder das richtige Reservieren noch das Steuern von GPIO-Ports.

Gpiolib und Libgpiod

Linux wäre jedoch nicht Linux, wenn nicht schon längst ein pfiffiger Entwickler – in diesem Fall Bartosz Golaszewski – geeignete Werkzeuge gebaut und der Community auf Kernel.org zur Verfügung gestellt hätte [3]. Seine Programmsuite Libgpiod [4] hilft beim Zugriff aus dem Userland auf GPIOs.

Da es auch für die Libgpiod keine fertigen Pakete gibt, muss jeder sie selbst erzeugen. Auf einem frisch installierten Raspbian installiert der Benutzer zunächst die notwendigen Pakete per »apt-get« und lädt danach mit »git« den Libgpiod-Quellcode. Nach dem Wechsel ins entsprechende Verzeichnis können die Konfiguration, der Generierungs- und der Installationsvorgang beginnen. Listing 1 dokumentiert die Abfolge.

Tabelle 1 listet die damit erzeugten Kommandos auf. Das Werkzeug »gpiodetect« zeigt die im Rechner verbauten GPIO-Bausteine mit den jeweiligen Namen und der Anzahl an GPIO-Leitungen an. »gpioinfo« liefert für jeden GPIO-Baustein die zugehörigen GPIOs, inklusive ihrer Namen, der potenziellen User und der momentanen Konfiguration (Ein-/Ausgabe, Active low). Wer eine bestimmte GPIO-Leitung auf Basis des Namens sucht, verwendet »gpiofind«.

Per »gpioget« lassen sich GPIOs lesen und mittels »gpioset« auch auf einen definierten Wert setzen. Beim Setzen ist es wichtig, mit der Option »-m« einen Modus anzugeben. Dieser legt fest, für wie lange beziehungsweise bis zu welchem Ereignis die GPIO-Leitung den übergebenen Wert einstellt. Abhängig von der Parametrierung bleibt der eingestellte Zustand für eine definierte Zeit, eine Benutzereingabe oder bis zum Empfang eines Signals erhalten. »gpiomon« hilft Zustandswechsel an GPIO-Leitungen zu überwachen. Abbildung 1 zeigt die Kommandos in Aktion.

Passend zum Gpiolib-Subsystem des Kernels (Abbildung 2) enthält die Libgpiod im Userland neben Zugriffskommandos auch Bindings für selbst geschriebene Programme in Python, C oder C++. Eine Dokumentation dazu liegt unter [5].

Listing 1: Kommandos zur Generierung der Libgpiod

```
01 root@raspberrypi:~$ sudo su
02 root@raspberrypi:/home/pi# apt-get install autoconf autoconf-archi
03 [...]
04 root@raspberrypi:/home/pi# git clone https://git.kernel.org/pub/sc
05 Klone nach 'libgpiod' ...
06 remote: Counting objects: 3951, done.
07 remote: Total 3951 (delta 0), reused 0 (delta 0)
08 Empfange Objekte: 100% (3951/3951), 575.04 KiB | 0 bytes/s, Fertig
09 Löse Unterschiede auf: 100% (2719/2719), Fertig.
10 root@raspberrypi:/home/pi# cd libgpiod/
11 root@raspberrypi:/home/pi/libgpiod# ./autogen.sh --enable-tools=ye
12 root@raspberrypi:/home/pi/libgpiod# make && make install
```

Tabelle 1: Applikationsprogramme

Programm	Funktion
gpiodetect	Anzeigen der verbauten GPIO-Chips
gpioinfo	Anzeige von Name, Nutzer, Konfiguration der GPIO-Leitungen
gpiofind	Suchen einer GPIO-Leitung
gpioget	Einlesen der angegebenen GPIO-Leitung
gpioset	GPIO-Leitung auf »0« oder »1« setzen
gpiomon	Schlafen bis zum Zustandswechsel an GPIO-Leitungen

```
pi@raspberrypi:~ $ gpiodetect
gpiochip0 [pinctrl-bcm2835] (54 lines)
gpiochip1 [brcmexp-gpio] (8 lines)
gpiochip2 [brcmvirt-gpio] (2 lines)
pi@raspberrypi:~ $ gpioinfo pinctrl-bcm2835 | head
gpiochip0 - 54 lines:
        line   0:      unnamed       unused   input  active-high
        line   1:      unnamed       unused   input  active-high
        line   2:      unnamed       unused   input  active-high
        line   3:      unnamed       unused   input  active-high
        line   4:      unnamed       unused   input  active-high
        line   5:      unnamed       unused   input  active-high
        line   6:      unnamed       unused   input  active-high
        line   7:      unnamed       unused   input  active-high
        line   8:      unnamed       unused   input  active-high
pi@raspberrypi:~ $ gpioset -m time -s 3 pinctrl-bcm2835 17=1
pi@raspberrypi:~ $ gpioget -h
Usage: gpioget [OPTIONS] <chip name/number> <offset 1> <offset 2> ...
Read line value(s) from a GPIO chip
Options:
  -h, --help:           display this message and exit
  -v, --version:        display the version and exit
  -l, --active-low:     set the line active state to low
pi@raspberrypi:~ $
```

Abbildung 1: Die Kommandos der Libgpiod ermöglichen einen performanten Zugriff.

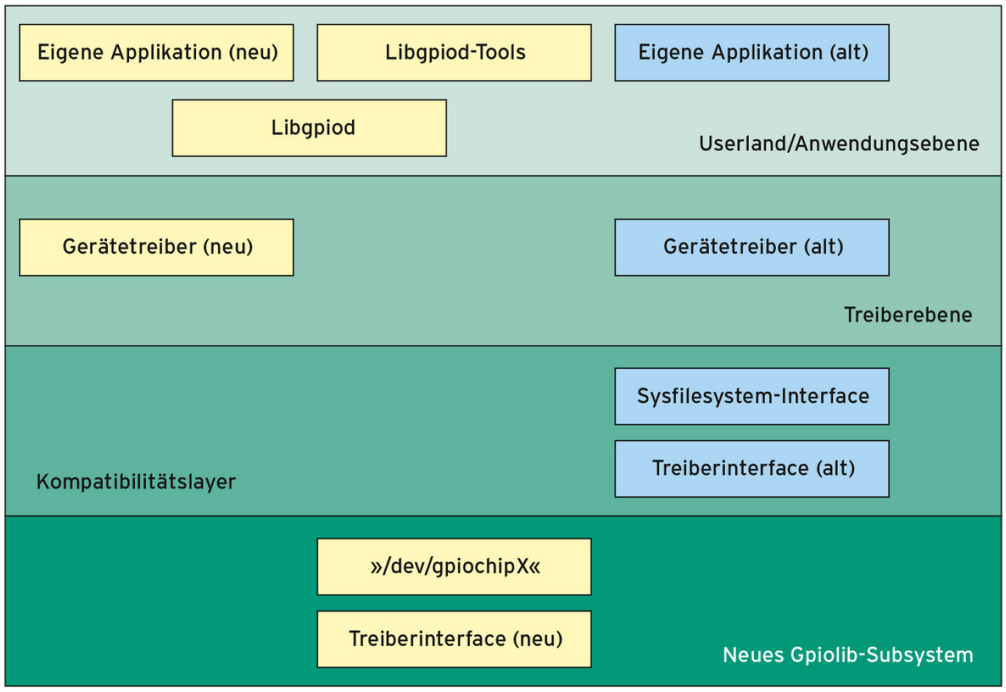


Abbildung 2: Die Architektur des Gpiolib-Subsystems. Die hellblauen Komponenten sind abgekündigt.

Profis am Werk

Der professionellste und schnellste Zugriff auf GPIO-Leitungen lässt sich jedoch im Rahmen eines Gerätetreibers implementieren. Der Gerätetreiber reserviert eine oder mehrere GPIO-Leitungen, konfiguriert sie als Ausgang, Eingang oder auch Interrupt-Eingang und greift dann auf die Leitungen exklusiv zu. Braucht der Gerätetreiber die Leitungen irgendwann nicht mehr, gibt er sie wieder frei.

Der Kernel referenziert eine GPIO-Leitung nicht mehr wie früher über deren Nummer, sondern über einen GPIO-Deskriptor – ganz ähnlich einem Filedeskriptor. Dahinter steckt ein objektorientiertes Modell, das dem Deskriptor Zugriffsmethoden zuordnet. Um einen

oder gleich mehrere GPIO-Deskriptoren zu erhalten, ruft der Programmierer eine der Reservierungsfunktionen aus Tabelle 2 auf.

Will der Treiberprogrammierer mit einem Funktionsaufruf direkt mehrere GPIO-Leitungen eines Bausteins reservieren, bekommt er die Adresse eines Deskriptoren-Felds (»gpiod_get_array()«) zurück. Es ist auch möglich, eine Leitung rein auf Basis der benötigten Funktionalität – zum Beispiel für I^2C – zu reservieren (»gpiod_get_optional()«).

Allerdings sind beim Raspberry Pi bisher keine Namen für die GPIO-Leitungen hinterlegt. Damit bleibt dem Programmierer nur die Reservieren-Funktion »gpio_to_desc()«, die die Nummer der Leitung übergeben bekommt und den zugehörigen GPIO-Deskriptor zurückliefert.

Tabelle 2: Kernelfunktionen der Gpiolib

Ausgewählte Kernelfunktionen
Reservierung
struct gpio_desc *gpiod_get(struct device *dev, const char *con_id, enum gpiod_flags flags)
struct gpio_desc *gpiod_get_index(struct device *dev, const char *con_id, unsigned int idx, enum gpiod_flags flags)
struct gpio_desc *gpiod_get_optional(struct device *dev, const char *con_id, enum gpiod_flags flags)
struct gpio_descs *__must_check gpiod_get_array(struct device *dev, const char *con_id, enum gpiod_flags flags)
struct gpio_desc *gpio_to_desc(unsigned gpio)
Konfiguration
int gpiod_direction_input(struct gpio_desc *desc)
int gpiod_direction_output(struct gpio_desc *desc, int value)
int gpiod_get_direction(const struct gpio_desc *desc)
int gpiod_to_irq(const struct gpio_desc *desc)
Zugriffsfunktionen
int gpiod_get_value(const struct gpio_desc *desc)
void gpiod_set_value(struct gpio_desc *desc, int value)
int gpiod_cansleep(const struct gpio_desc *desc)
int gpiod_get_value_cansleep(const struct gpio_desc *desc)
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)
int gpiod_get_raw_value(const struct gpio_desc *desc)
void gpiod_set_raw_value(struct gpio_desc *desc, int value)
Freigabe
void gpiod_put(struct gpio_desc *desc)
void gpiod_put_array(struct gpio_descs *descs)

Richtungweisend

Ist die GPIO-Leitung reserviert, muss der Programmierer durch Aufruf der Funktion »gpiod_direction_input()« oder »gpiod_direction_output()« die Richtung konfigurieren. Danach darf der Zugriff erfolgen. Wer festzustellen will, ob an einem GPIO eine »1« anliegt oder eine »0«, ruft typisch die Funktion »gpiod_get_value()« auf. Um Spannung oder eben keine Spannung auf die Leitung zu legen, dafür dient »gpiod_set_value()«.

Dauert der Zugriff auf ein GPIO länger, beispielsweise weil es an einem I^2C hängt, kann man dem Betriebssystem auch die Möglichkeit anbieten, beim GPIO-Zugriff eine kurze Schlafpause einzulegen. Das Nickerchen ist natürlich nur dann gestattet, wenn der Aufruf nicht aus einem Interruptkontext heraus erfolgt, in dem Schlafen tabu ist.

Darüber hinaus existieren Möglichkeiten, mit einem einzelnen Funktionsaufruf mehrere GPIOs gleichzeitig anzusteuern. Hier kommen die Array-Funktionen zum Einsatz. Die Zugriffsfunktionen mit dem »raw« im Namen greifen direkt auf die GPIO-Leitungen zurück und berücksichtigen damit nicht mögliche Active-low-Zustände. Benötigt sein Programm die GPIO-Leitung nicht mehr, gibt der Programmierer die Ressource per »gpio_put()« wieder zurück.

Listing 2 zeigt in der Funktion »config_gpios()« beim Raspberry Pi den Zugriff auf die GPIO-Leitungen 17 und 18 über einen Gerätetreiber. Die Funktion »gpiodtest_request()« reserviert und konfiguriert GPIO-17 als Output und GPIO-18 als Input. Zusätzlich legt das Programm GPIO-18 als Interrupteingang fest. Die zugehörige Interrupt-Service-Routine trägt den Namen »rpi_gpio_isr()«.

Da der Raspberry Pi leider keine Namen für die GPIOs zugeordnet hat, erfolgt die Anforderung der GPIO-Leitung über deren Nummer per »gpio_to_desc()«. Der Zugriff selbst ist dann sehr gradlinig und in den Funktionen »driver_read()« und »driver_write()« implementiert. Um den Code auszuprobieren, muss der Raspberry-Rootuser die Kernel-Header per

```
apt-get install raspberrypi-kernel-headers
```

installieren. Zusätzlich zum Quellcode (»gpiodtest.c«) bedarf es eines einfachen Makefile (Listing 3). Jetzt reicht ein »make«, um den Gerätetreiber herzustellen. Wer ein zweites Terminalfenster öffnet und dort »sudo tailf /var/log/kern.log« eingibt, kann die Aktivitäten des Gerätetreibers anhand der »printk«- und »dev_err«-Ausgaben verfolgen.

Den erfolgreich kompilierten Treiber lädt »insmod gpiodtest.ko« in den Kernel (Abbildung 3). Für einen ersten Test reicht es, per Echo-Kommando eine »1« auf die seit dem Modul-Laden bereitstehende Gerätedatei »/dev/gpiodtest« zu schreiben. Wer am GPIO-17 über einen Vorwiderstand eine LED angeschlossen hat (Abbildung 4), sieht sie nun leuchten. Eine »0« lässt das LED-Licht erlöschen.

Zum Testen des Eingangs ziehen Bastler GPIO-18 per Widerstand auf 3,3 Volt (Pin 1 der Steckerleiste, Abbildung 5) und verbinden einen Schalter über einen Widerstand mit Masse. Wenn sie jetzt den Schalter betätigen, zieht das die Leitung auf Masse und ein Interrupt wird ausgelöst, was zu einer Meldung in den Kernelmessages (»/var/log/kern.log«) führt. Das Kommando »rmmod gpiodtest« entlädt den Gerätetreiber.

Abbildung 3: Den Gerätetreiber generieren und testen.

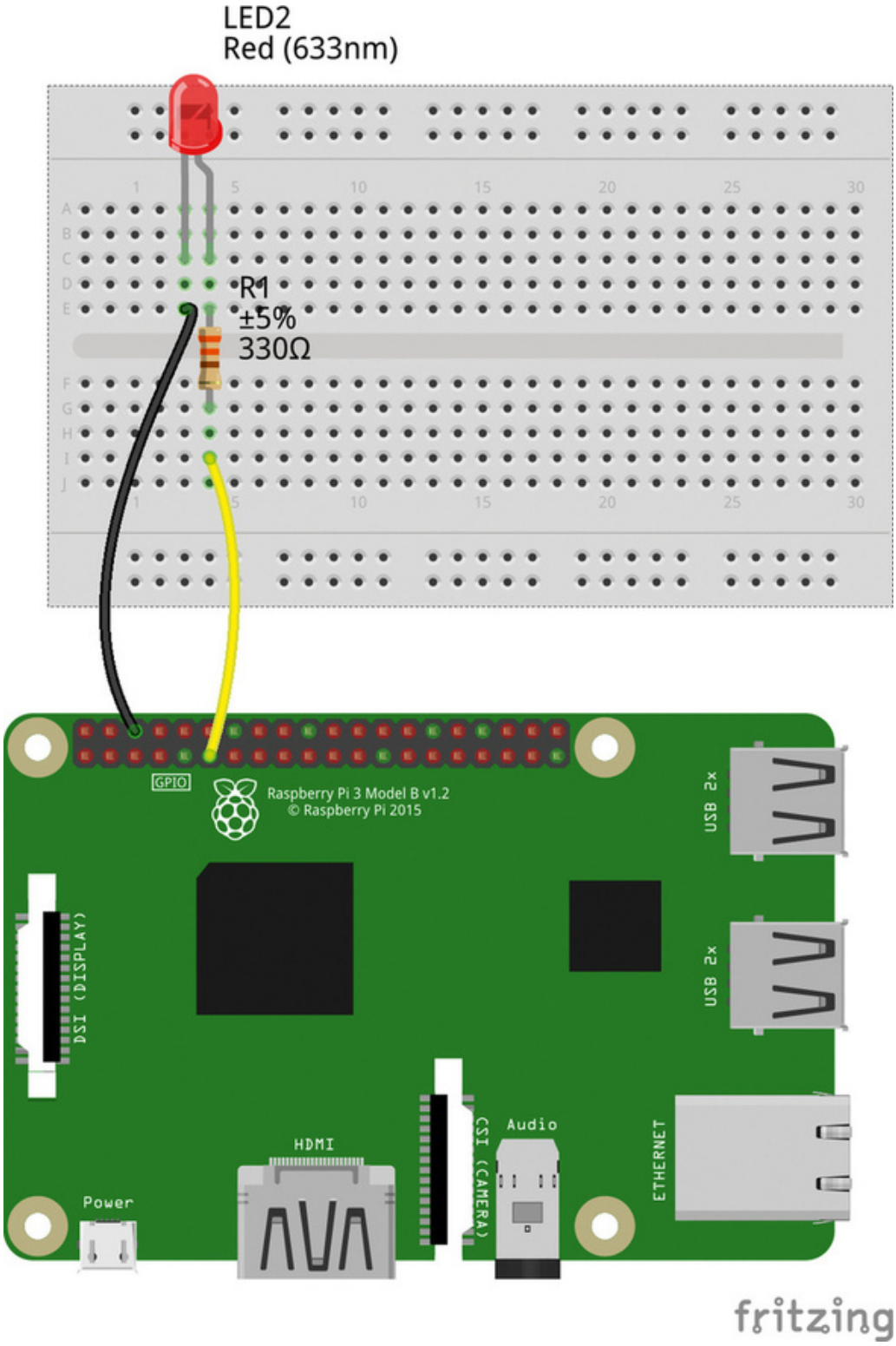


Abbildung 4: **Diese einfache Schaltung reicht zum Test bereits aus.**

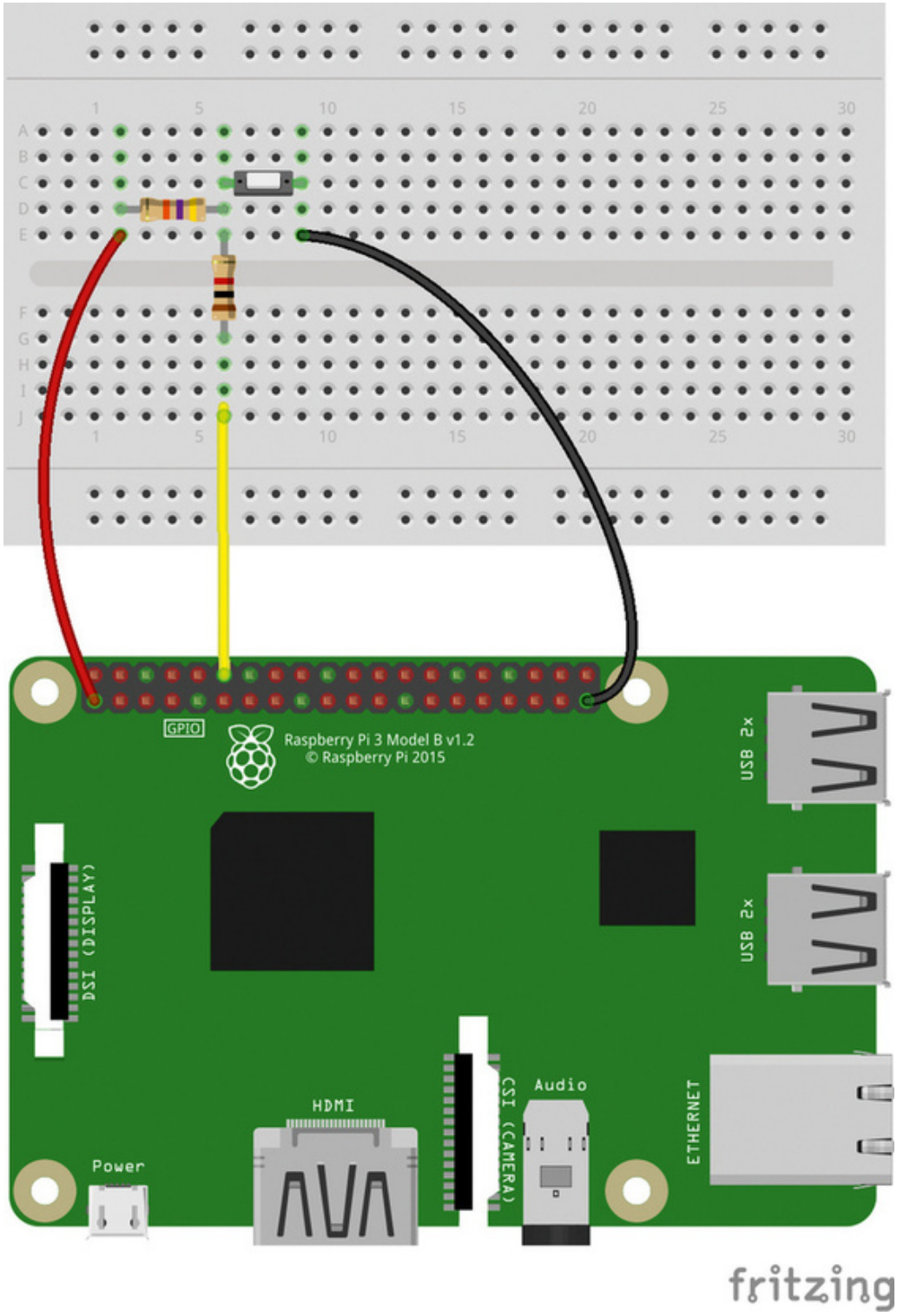


Abbildung 5: **Eine Schaltung zum Testen von Input und Interrupt.**

Listing 2: GPIO-Treiber (*gpiodtest.c*)

```
001 #include <linux/module.h>
002 #include <linux/fs.h>
003 #include <linux/device.h>
004 #include <linux/cdev.h>
005 #include <linux/uaccess.h>
006 #include <linux/interrupt.h>
007 #include <linux/gpio/consumer.h>
008
009 static dev_t gpiodtest_dev_number;
010 static struct cdev *driver_object;
011 static struct class *gpiodtest_class;
```

```
012 static struct device *gpiodtest_dev;
013
014 static struct gpio_desc *gpio17, *gpio18;
015 static int irq_18;
016
017 static irqreturn_t rpi_gpio_isr( int irq, void *data )
018 {
019     printk("rpi_gpio_isr( %d, %p )\n", irq, data );
020     return IRQ_HANDLED;
021 }
022
023 static int config_gpios( void )
024 {
025     int err;
026
027     gpio17 = gpio_to_desc( 17 );
028     if (IS_ERR(gpio17)) {
029         dev_err(gpiodtest_dev, "can't acquire GPIO17\n");
030         return -EIO;
031     }
032     err = gpiod_direction_output( gpio17, 0 );
033     if (err<0) {
034         dev_err(gpiodtest_dev,
035                 "setting direction failed\n");
036         gpiod_put( gpio17 );
037         return -EIO;
038     }
039
040     gpio18 = gpio_to_desc( 18 );
041     if (IS_ERR(gpio18)) {
042         dev_err(gpiodtest_dev, "can't acquire GPIO18\n");
043         return -EIO;
044     }
045     err = gpiod_direction_input( gpio18 );
046     if (err<0) {
047         dev_err(gpiodtest_dev,
048                 "setting direction failed\n");
049         gpiod_put( gpio17 );
050         gpiod_put( gpio18 );
051         return -EIO;
052     }
053     irq_18 = gpiod_to_irq( gpio18 );
054     dev_info(gpiodtest_dev, "using irq: %d\n", irq_18);
055
056     err = request_irq( irq_18, rpi_gpio_isr,
057         IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
058         "gpiodtest-18", driver_object);
059     if (err) {
060         dev_err(gpiodtest_dev,
061                 "request_irq failed with %d\n",err);
062         gpiod_put( gpio17 );
063         gpiod_put( gpio18 );
064         return -EIO;
065     }
066     return 0;
067 }
068
069 static int free_gpios( void )
070 {
071     if (gpio17)
072         gpiod_put( gpio17 );
```

```
073     if (gpio18)
074         gpiod_put( gpio18 );
075     free_irq(irq_18, driver_object);
076     return 0;
077 }
078
079 static ssize_t driver_write( struct file *instanz,
080         const char __user *user,
081         size_t count, loff_t *offset )
082 {
083     unsigned long not_copied=0, to_copy=0;
084     int value=0;
085
086     to_copy = min( count, sizeof(value) );
087     not_copied=copy_from_user(&value,user,to_copy);
088     gpiod_set_value( gpio17, value );
089     *offset += to_copy-not_copied;
090     return to_copy-not_copied;
091 }
092
093 static ssize_t driver_read( struct file *instanz,
094         char __user *user,
095         size_t count, loff_t *offset )
096 {
097     unsigned long not_copied, to_copy;
098     int value;
099
100     value = gpiod_get_value( gpio18 );
101
102     to_copy = min( count, sizeof(value) );
103     not_copied=copy_to_user(user,&value,to_copy);
104     *offset += to_copy-not_copied;
105     return to_copy-not_copied;
106 }
107
108 static struct file_operations fops = {
109     .owner= THIS_MODULE,
110     .read= driver_read,
111     .write= driver_write,
112 };
113
114 static int __init mod_init( void )
115 {
116     if (alloc_chrdev_region(&gpiodtest_dev_number,
117         0,1,"Gpiodtest")<0)
118         return -EIO;
119     driver_object = cdev_alloc();
120     if (driver_object==NULL)
121         goto free_device_number;
122     driver_object->owner = THIS_MODULE;
123     driver_object->ops = &fops;
124     if (cdev_add(driver_object,gpiodtest_dev_number,1))
125         goto free_cdev;
126     gpiodtest_class = class_create( THIS_MODULE, "Gpiodtest" );
127     if (IS_ERR( gpiodtest_class )) {
128         pr_err( "gpiodtest: no udev support\n");
129         goto free_cdev;
130     }
131     gpiodtest_dev = device_create(gpiodtest_class,NULL,
132         gpiodtest_dev_number, NULL, "%s", "gpiodtest" );
133     if (IS_ERR( gpiodtest_dev )) {
```

```
134         pr_err( "gpiodtest: device_create failed\n");
135         goto free_class;
136     }
137     config_gpios();
138     return 0;
139 free_class:
140     class_destroy( gpiodtest_class );
141 free_cdev:
142     kobject_put( &driver_object->kobj );
143 free_device_number:
144     unregister_chrdev_region( gpiodtest_dev_number, 1 );
145     return -EIO;
146 }
147
148 static void __exit mod_exit( void )
149 {
150     free_gpios();
151     device_destroy( gpiodtest_class, gpiodtest_dev_number );
152     class_destroy( gpiodtest_class );
153     cdev_del( driver_object );
154     unregister_chrdev_region( gpiodtest_dev_number, 1 );
155     return;
156 }
157
158 module_init( mod_init );
159 module_exit( mod_exit );
160
161 MODULE_AUTHOR("Juergen Quade");
162 MODULE_LICENSE("GPL");
```

Listing 3: Makefile für den Gerätetreiber

```
01 obj-m := gpiodtest.o
02
03 all:
04         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modu
05
06 clean:
07         make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clea
```

Schnelle Rechtecke

Listing 4 zeigt den Code einer Mini-Applikation, die über die Gerätedatei ein Rechtecksignal auf GPIO-17 ausgibt. Ein Raspberry Pi schafft damit eine Frequenz von stattlichen 500 Kilohertz – rund das Doppelte dessen, was ein C-Programm auf derselben Hardware zustande bringt, das die Funktionen der Libgpiod nutzt.

Listing 4: Rechteckgenerator

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <fcntl.h>
04
05 int main( int argc, char **argv, char **envp )
```

```
06 {
07     int fd, value=1;
08     int i;
09
10     fd = open( "/dev/gpiodtest", O_RDWR );
11     if (fd<0) {
12         perror("/dev/gpiodtest");
13         return -1;
14     }
15     for (i=0; i<100000; i++) {
16         value = 1;
17         write( fd, &value, sizeof(value) );
18         value = 0;
19         write( fd, &value, sizeof(value) );
20     }
21     return 0;
22 }
```

Karger Erfolg

Obwohl das Gpiolib-Subsystem bereits seit geraumer Zeit im Kernel implementiert ist, hat es bisher nur wenige Nutzer. Ob es daran liegt, dass das alte Interface bequem ist und es zum neuen neben der kurzen Dokumentation im Linux-Quellcode [6] kaum Beispielcode gibt? Doch braucht das neue Interface noch einiges, um technisch perfekt zu sein.

Zunächst fehlt beim Raspberry Pi eine Zuweisung von Namen zu den GPIO-Leitungen. Gut, das ist nicht dem Subsystem anzulasten. Aber damit eine wirklich generische Nutzung möglich ist, muss eine eindeutige Namensgebung für spezifische Funktionalitäten erfolgen, und die ist nicht in Sicht.

Außerdem: Um alle Möglichkeiten moderner GPIOs auszunutzen, muss der Programmierer auch mit dem neuen Interface auf die Register der Chips zugreifen. Irgendwann wird eine Kern-Technik daher den GPIOs abermals auf die Pins starren müssen. Auf Wiederlesen! (jk)

Infos

1. Wiring Pi: [http://wiringpi.com]
2. Quade, Kunst, "Kern-Technik 70": Linux-Magazin 10/13, S. 102
3. Bartosz Golaszewski, "New GPIO-Interface for User Space", Embedded Linux Conference Europe 2017: [https://schd.ws/hosted_files/osseu17/88/elce2017_new_GPIO_interface.pdf]
4. Craig Peacock, "An Introduction to chardev GPIO and Libgpiod on the Raspberry PI": [https://www.beyondlogic.org/an-introduction-to-chardev-gpio-and-libgpiod-on-the-raspberry-pi/]
5. Git-Repository für Libgpiod: [https://git.kernel.org/pub/scm/libs/libgpiod/libgpiod.git/]
6. General Purpose Input/Output – Descriptor Consumer Interface: [https://www.kernel.org/doc/Documentation/gpio/consumer.txt]

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade ist Professor an der Hochschule Niederrhein und führt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux durch.