Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 134

Fruchtbarer Baum

Beim Zugriff auf per GPIO angebundene Hardware setzen Profis auf die Kombination aus Gerätetreibern, der Gpiolib und dem Devicetree. Eva-Katharina Kunst, Jürgen Quade

Linux ist in Bewegung – grundsätzlich ja positiv, aber manchmal schmerzt es: Zum Beispiel dann, wenn der bislang funktionierende Treibercode mit der neuen Kernel-Version auf dem Raspberry Pi einen seltsamen Fehlercode wirft. Ein kurzes Debugging offenbart: Das Reservieren eines GPIOs per `gpio_to_desc()` quittiert den Aufruf mit einem Null-Pointer.

Bei GPIOs (General Purpose Input Output) handelt es sich um Leitungen, die man softwaregesteuert auf Null oder auf Eins setzt, auf Spannung oder auf keine Spannung. Verbindet man eine solche Leitung etwa mit einem Widerstand und einer LED, kann man die LED softwaregesteuert an- und ausschalten. Mit einem Taster verbunden, lässt sich einlesen, ob an der Leitung eine Spannung anliegt.

Spätestens seit Erscheinen des RasPi sind GPIOs einem breiteren Publikum bekannt, das in Heimlaboren mit dem Mini-Rechner Bewässerungssysteme, kleine Roboter, Lichtorgeln oder Laserschwerter realisiert. Während viele Maker GPIOs über Frameworks ansteuern, nehmen Profis die Diretissma über Gerätetreiber.

Linus Torvalds hat schon vor Jahren den Zugriff auf GPIOs innerhalb des Kernels auf ein Deskriptor-Interface umgestellt, das als Gpiolib bezeichnet wird. Zugriffe wie Lesen (`gpiod_get()`) und Schreiben (`gpiod_set()`) finden dabei nicht mehr direkt durch Angabe der GPIO-Nummer statt. Stattdessen dient die GPIO-Nummer als Attribut eines Datenobjekts, des Deskriptors, das der Treiber beispielsweise über die erwähnte Funktion `gpio_to_desc()` beim Kernel anfordert. Normalerweise gibt die Funktion `gpio_to_desc()` unproblematisch den zur übergebenen GPIO-Nummer gehörenden GPIO-Deskriptor zurück.

Mottenkiste, Trick 17

Das Problem liegt darin, dass man GPIOs genau genommen über zwei Adressinfos auswählen muss: den Controller-Chip und die GPIO-Nummer an diesem Controller. Um hier mit einem Parameter auszukommen, hat sich die Linux-Gemeinde des Controller-spezifischen Offset-Tricks bedient. In der aktuellen Kernel-Version wurde der Offset für den ersten Controller des Raspberry Pi 4 neu auf 512 festgelegt. Um den zu *GPIO17* gehörenden Deskriptor zu reservieren, muss man die Funktion ab Kernel 6.6 mit dem Wert 529 aufrufen. Schwupp, schon gibt es keine Fehlermeldung mehr, und der Gerätetreiber greift erfolgreich auf die GPIOs zu.

Dass der Offset 512 beträgt, lässt sich im Übrigen weder durch den Aufruf des

Programms `gpiodetect` noch über `gpioinfo` ablesen, mit denen sich aus dem Userland heraus Informationen zu den GPIOs anzeigen lassen. Erst das Auslesen der Datei `/sys/kernel/debug/gpio` gibt die intern verwendeten GPIO-Nummern preis (Listing 1).

Optimal geht anders

Das Verwenden der Funktion `gpio_to_desc()` stellt allerdings ohnehin eine suboptimale Lösung dar. Die Funktion ist zwar einerseits sehr schlank, trennt aber andererseits den Code nicht von den Daten. Dabei favorisiert Linux doch schon seit Langem die saubere Trennung durch Verwendung der Devicetrees.

Im Kernel ist der Devicetree eine hierarchisch organisierte Datenstruktur, die möglichst alle Hardwareinformationen enthält. Dazu gehören beispielsweise die Adressen der Funktionsregister, die zur Verfügung stehenden Interrupts oder die DMA-Kanäle. Das System legt diese Datenstruktur beim Booten an, indem der Bootloader einen sogenannten Devicetree-Blob im Speicher ablegt.

Den Blob, also den strukturierten, vom Kernel interpretierbaren Datenhaufen, generiert ein Devicetree-Compiler aus einer menschenlesbaren Beschreibung im JSON-Format. Der Devicetree selbst besteht aus hierarchisch angeordneten Knoten, die in ihren Attributen die relevanten Informationen speichern, unter anderem zum Auffinden eines Knoten-

Listing 1: Offsets im Sys-Filesystem

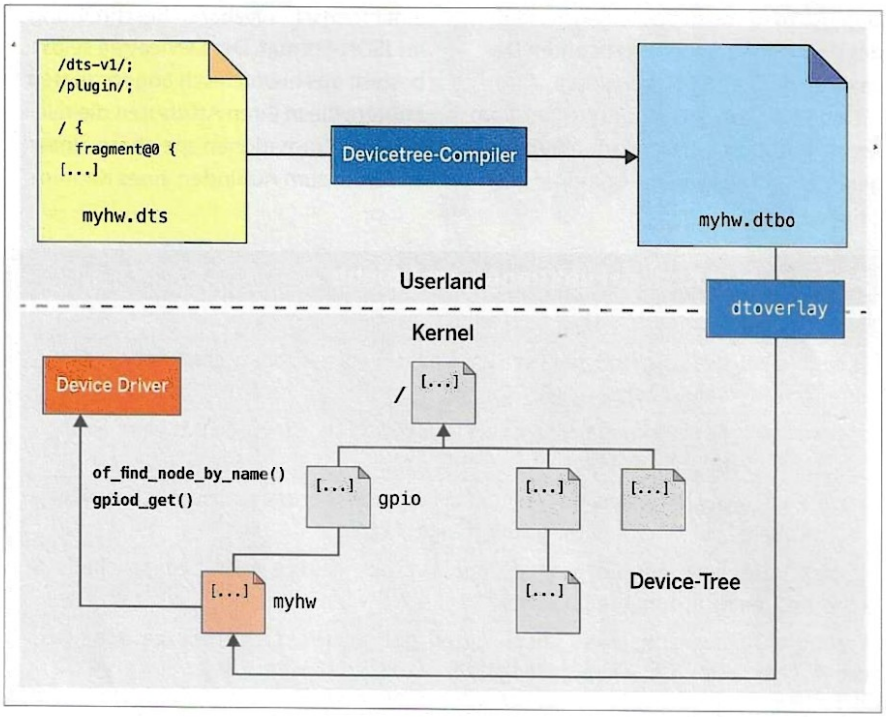
```
# cat /sys/kernel/debug/gpio
gpiochip0: GPIOs 512-569, parent: platform/fe200000.gpio, pinctrl-bcm2711:
 gpio-512 (ID_SDA              )
 gpio-513 (ID_SCL              )
 gpio-514 (GPIO2               )
 gpio-515 (GPIO3               )
 gpio-516 (GPIO4               )
 gpio-517 (GPIO5               )
 gpio-518 (GPIO6               )
[...]
```

Ausgewählte Kernel-Funktionen der Gpiolib

Reservierung
`struct gpio_desc *gpiod_get(struct device *dev, const char *con_id, enum gpiod_flags flags)`
`struct gpio_desc *devm_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 devm_gpiod_put(struct device *dev, struct gpio_desc *desc)`
`void gpiod_put_array(struct gpio_descs *descs)`

Listing 2: Devicetree – Ressourcen (myhw.dts)

```
/dts-v1/;
/plugin/;
/ {
  fragment@0 {
    target = <&gpio>;
    __overlay__ {
      myhw {
        compatible = "myhw,hwtest";
        my-irq-gpios = <&gpio 4 0>;
        my-trigger-gpios = <&gpio 17 0>;
        my-array-gpios =  <&gpio 10 0>,
          <&gpio 11 0>,
          <&gpio 12 0>,
          <&gpio 13 0>,
      };
    };
  };
};
```



1 Von der Beschreibung bis zum Treiber: der Informationsfluss im Kernel.

Listing 3: Zugriff auf GPIOs (myhw.c)

```
#include <linux/module.h>
#include <linux/cdev.h>
#include <linux/interrupt.h>
#include <linux/gpio/consumer.h>
#include <linux/of_device.h>
static dev_t myhw_dev_number;
static struct cdev driver_object;
static struct class *myhw_class;
static struct device *myhw_dev;
static struct gpio_desc *gpio_irq, *gpio_trigger;
static int myhw_irq;
static irqreturn_t hard_isr(int irq, void *data) {
  return IRQ_WAKE_THREAD;
}
static irqreturn_t isr_thread(int irq, void *data) {
  pr_info("isr_thread(%d, %p)\n", irq, data);
  return IRQ_HANDLED;
}
static int config_gpios(void) {
  int err;
  struct device_node *nodeptr = myhw_dev->of_node;
  nodeptr = of_find_node_by_name(nodeptr, "myhw");
  myhw_dev->of_node = nodeptr;
  pr_info("config_gpios(%p, %p)\n", myhw_dev, nodeptr);
  gpio_trigger = gpiod_get(myhw_dev, "my-trigger", GPIOD_OUT_LOW);
  if (IS_ERR(gpio_trigger)) {
    pr_err("gpiod_get failed with my-trigger\n");
```

namens. Der fixe Name des Wurzelknotens lautet ähnlich wie im Dateisystem `/`. Im Attribut `compatible` wird zudem abgelegt, für welche Hardware Knoten und deren Subknoten relevant sind.

Ursprünglich war der Devicetree rein statisch, seit Langem lässt er sich jedoch zur Laufzeit verändern und erweitern. Im einfachsten Fall tauscht man nur die Parameter eines Knotens mithilfe des Kommandos `dtparam` aus. Ansonsten kommen sogenannte Overlays zum Zuge, die die komplette Baumstruktur mithilfe von `dtoverlay` verändern. Die Overlays spezifiziert man ebenfalls im JSON-Format. Sie verwenden aber die beiden Schlüsselworte `fragment` und `overlay`, um eine leichtere Einsortierung der modifizierten Knoten im Baum zu ermöglichen (siehe auch Listing 2).

Um nicht die gesamte Baumhierarchie im Overlay spezifizieren zu müssen, gibt es das Attribut `target`, das direkt auf den relevanten Knoten verweist. Auch die Beschreibung eines Overlays überführt der Devicetree-Compiler in einen Blob, in diesem Fall in ein `dtbo`. Den Devicetree-Overlay-Blob laden Sie per `dtoverlay` in den Kernel und können ihn mit demselben Kommando unter Zusatz der Option `-r` auch wieder entladen 1.

Her mit der Info

Innerhalb des Kernels greifen die Gerätetreiber auf die Knoten des Devicetrees zu und lassen sich mit den für ihre Funktion notwendigen Parametern versorgen. Dazu dienen Funktionen wie `of_find_node_by_name(char *nodename)`, die die Adresse des Knotens mit dem Namen `nodename` ermittelt, und `of_get_property(struct node *nodeptr, char *propname, int size)`, die den Wert eines zum Knoten gehörenden Attributs ausliest. Der Datentyp des ausgelesenen Attributs muss dem Gerätetreiber bekannt sein beziehungsweise wird vom Gerätetreiber sogar vorgegeben.

Das GPIO-Subsystem ist auf die Zusammenarbeit mit dem Devicetree hin optimiert. Die Funktion `gpiod_get()` (siehe Tabelle Ausgewählte Kernel-Funktionen der Gpiolib), die einen GPIO-Deskriptor reserviert, erwartet in Form des Parameters `con_id` den Namen eines Devicetree-Attributs. Allerdings enthält

die Programmierung hier eine Falltür, die sich im Kleingedruckten der Linux-Kernel-Dokumentation im Abschnitt „GPIO Mappings" findet: Der Name eines GPIOs muss das Suffix `-gpio` oder `-gpios` haben. Die `con_id` selbst jedoch muss man ohne dieses Suffix angeben.

Listing 2 zeigt ein Devicetree-Overlay-File, das ein erstes Fragment `fragment@0` definiert. Das dort spezifizierte Attribut `target` zeigt, dass sich die nachfolgenden Angaben auf den Knoten respektive den Teilbaum beziehen, der die GPIOs definiert. Daran wird ein neuer Knoten mit dem Namen `myhw` angehängt, der die Attribute `my-irq-gpios`, `my-trigger-gpios` und `my-array-gpios` besitzt. Ersteres bestimmt *GPIO4* als Interrupt-Eingang und das zweite *GPIO17* als Trigger, mit dem man bei einer Schaltung wie der in Abbildung 1 den Interrupt softwaregesteuert auslösen kann.

Konfiguration inklusive

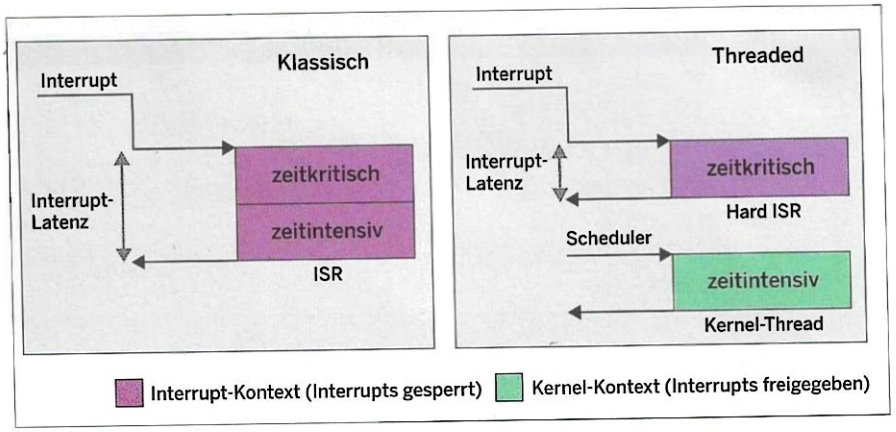
Einen dazu passenden Gerätetreiber finden Sie in Listing 3. Die dort implementierte Funktion `config_gpios()` ermittelt per `of_find_node_by_name()` den Knoten im Devicetree. Die Adresse des Knotens wird mit der Datenstruktur `struct device` verknüpft, die im Sys-Filesystem das Gerät repräsentiert und dafür ein gesondertes Feld bereitstellt. Danach bekommt die Funktion `gpiod_get()` den Namen des GPIOs (ohne das Suffix `-gpios`) übergeben, um den zugehörigen Deskriptor zu erhalten. Hier wird bemerkenswerterweise ein GPIO durch Angabe eines Flags direkt mit konfiguriert – anders, als wenn man sich den Deskriptor per `gpio_to_desc()` holt.

Wie sich aus dem Codebeispiel leicht erschließen lässt, konfiguriert die Angabe von `GPIOD_IN` einen Eingabe-Pin sowie die Angabe von `GPIOD_OUT_LOW` einen Ausgabe-Pin. Letzterer hat initial an seiner Leitung 0 Volt anliegen. Bei `GPIOD_OUT_HIGH` lägen direkt nach dem Ausführen 3,3 Volt an. Zusätzlich können Sie den Ausgang als Open Drain ausführen (`GPIOD_OUT_LOW_OPEN_DRAIN` respektive `GPIOD_OUT_HIGH_OPEN_DRAIN`). Soll an dieser Stelle keine Konfiguration stattfinden, setzen Sie `GPIOD_ASIS` ein.

Einmal reservierte Ressourcen gilt es später wieder freizugeben. Dazu genügt

Listing 3: Zugriff auf GPIOs (myhw.c) (Fortsetzung von S. 74)

```
    return -EIO;
  }
  pr_info("gpio-trigger configured...");
  gpio_irq = gpiod_get(myhw_dev, "my-irq", GPIOD_IN);
  if (IS_ERR(gpio_irq)) {
      pr_err("gpiod_get failed with my-irq\n");
      return -EIO;
  }
  myhw_irq = gpiod_to_irq(gpio_irq);
  dev_info(myhw_dev, "using irq: %d\n", myhw_irq);
  err = request_threaded_irq(myhw_irq, hard_isr,isr_thread,
      IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
      "myhw", &driver_object);
  if (err) {
      pr_err("request_irq failed with %d\n",err);
      gpiod_put(gpio_irq);
      gpiod_put(gpio_trigger);
      return -EIO;
  }
  return 0;
}
static int free_gpios(void) {
  if (gpio_irq)
      gpiod_put(gpio_irq);
  if (gpio_trigger)
      gpiod_put(gpio_trigger);
  free_irq(myhw_irq, &driver_object);
  return 0;
}
static ssize_t device_write(struct file *instanz, const char __user
*user, size_t count, loff_t *offset) {
  unsigned long not_copied=0, to_copy=0;
  int value=0;
  to_copy = min(count, sizeof(value));
  not_copied=copy_from_user(&value,user,to_copy);
  pr_info("device_write(%d)", value);
  gpiod_set_value(gpio_trigger, value);
  *offset += to_copy-not_copied;
  return to_copy-not_copied;
}
static ssize_t device_read(struct file *instanz, char __user *user,
size_t count, loff_t *offset) {
  unsigned long not_copied, to_copy;
  int value;
  value = gpiod_get_value(gpio_irq);
  to_copy = min(count, sizeof(value));
  not_copied=copy_to_user(user,&value,to_copy);
  *offset += to_copy-not_copied;
  return to_copy-not_copied;
}
```



2 Kürzere Latenzzeiten dank Threaded-Interrupts.

Listing 3: Zugriff auf GPIOs (myhw.c) (Fortsetzung von S. 75)

```
static struct file_operations fops = {
  .owner    = THIS_MODULE,
  .read     = device_read,
  .write    = device_write,
};
static int __init mod_init(void) {
  if (alloc_chrdev_region(&myhw_dev_number,0,1,"myhw")<0)
    return -EIO;
  cdev_init(&driver_object, &fops);
  driver_object.owner = THIS_MODULE;
  if (cdev_add(&driver_object,myhw_dev_number,1))
    goto free_devnum;
  myhw_class = class_create("myhw");
  if (IS_ERR(myhw_class)) {
    pr_err("myhw: no udev support\n");
    goto free_cdev;
  }
  myhw_dev = device_create(myhw_class,NULL,myhw_dev_number,
    NULL, "%s", "myhw");
  if (IS_ERR(myhw_dev)) {
    pr_err("myhw: device_create failed\n");
    goto free_class;
  }
  if (config_gpios())
    goto free_device;
  return 0;
free_device:
  device_destroy(myhw_class, myhw_dev_number);
free_class:
  class_destroy(myhw_class);
free_cdev:
  cdev_del(&driver_object);
free_devnum:
  unregister_chrdev_region(myhw_dev_number, 1);
  return -EIO;
```

ein Aufruf von `gpiod_put()` unter Angabe des Deskriptors beim Entladen des Treibers. Wenn Sie die Header-Datei `consumer.h` im Linux-Quellcode studieren, dürfte Ihnen auffallen, dass eine Variante zum Reservieren eines GPIO-Deskriptors namens `devm_gpiod_get()` existiert. Das Präfix `devm_` steht für Device Management und soll beim Programmieren die Freigabe von Ressourcen ersparen. Dazu wird die reservierte Ressource mit einem Geräteobjekt (`struct device`) verknüpft.

Wenn man beim Entladen des Treibers das Device-Objekt durch den Aufruf von `device_destroy()` freigibt, obliegt es dem Geräteobjekt, noch vor dem eigenen Ende die verknüpften Ressourcen eine nach der anderen freizugeben. Das macht die Funktion `devm_gpiod_put()` im Grunde genommen überflüssig. Die Automatisierung der Ressourcenverwaltung funktioniert übrigens auch mit anderen Ressourcentypen wie dynamisch reserviertem Speicher (`devm_kzalloc()`). Der Einsatz ergibt freilich nur dann Sinn, wenn Ressourcen über eine längere Zeit reserviert bleiben sollen.

Um den Beispieltreiber auszuprobieren, generieren Sie als Erstes durch Aufruf des Devicetree-Compilers aus der Hardwarebeschreibung `myhw.dts` den Devicetree-Overlay-Blob `myhw.dtbo`. Den Treiber `myhw.c` übersetzen Sie mittels eines passenden Makefiles (Listing 4) zum Kernel-Objekt `myhw.ko`. Mit Root-Rechten ausgestattet, laden Sie dann den Devicetree-Blob `myhw.dtbo` in den Pi-OS-Kernel, es folgt der Gerätetreiber selbst.

Lädt der Treiber fehlerfrei, ist das ein gutes Zeichen: Dann war die Reservierung der GPIOs erfolgreich. Grundsätzlich lassen sich Erfolg und Misserfolg jedoch am besten in einem separaten Terminalfenster verfolgen, in dem das Kommando `dmesg --follow` läuft. Im Fehlerfall finden Sie so am ehesten mögliche Ursachen für das Scheitern.

Der Beispieltreiber demonstriert außer dem Reservieren auch das Konfigurieren der Leitung `my-irq` als Interrupt-Eingang. Der soll bei Flankenwechseln eine Interrupt-Service-Routine (ISR) aktivieren, die hier in moderner Form direkt als Threaded-Interrupt ausgeführt ist und damit aus einem echten Interrupt-Teil (`my_hard-isr()`) und dem anschließend aktivierten Kernel-Thread (`my_isrthread()`) be-

```
quade@raspi-dev:~/linux-mag $ ls
Makefile  myhw.c  myhw.dts
quade@raspi-dev:~/linux-mag $ make myhw.dtbo
dtc -@ -W no-unit_address_vs_reg -I dts -O dtb -o myhw.dtbo myhw.dts
quade@raspi-dev:~/linux-mag $ make
make    -C /lib/modules/6.6.20+rpt-rpi-v8/build M=/home/quade/linux-mag modules
make[1]: Entering directory '/usr/src/linux-headers-6.6.20+rpt-rpi-v8'
  CC [M]  /home/quade/linux-mag/myhw.o
  MODPOST /home/quade/linux-mag/Module.symvers
  CC [M]  /home/quade/linux-mag/myhw.mod.o
  LD [M]  /home/quade/linux-mag/myhw.ko
make[1]: Leaving directory '/usr/src/linux-headers-6.6.20+rpt-rpi-v8'
quade@raspi-dev:~/linux-mag $ sudo su
root@raspi-dev:/home/quade/linux-mag# dtoverlay myhw.dtbo
root@raspi-dev:/home/quade/linux-mag# dtoverlay -l
Overlays (in load order):
0:  myhw
root@raspi-dev:/home/quade/linux-mag# insmod myhw.ko
root@raspi-dev:/home/quade/linux-mag# dmesg -H | tail -n 3
[Apr23 20:45] config_gpios( 00000000da03bc8f, 000000009ed191b )
[  +0.000091] gpio-trigger configured...
[  +0.000038] myhw myhw: using irq: 55
root@raspi-dev:/home/quade/linux-mag# echo -n -e "\x1" >/dev/myhw
root@raspi-dev:/home/quade/linux-mag# echo -n -e "\x0" >/dev/myhw
root@raspi-dev:/home/quade/linux-mag# dmesg -H | tail -n 5
[  +0.000038] myhw myhw: using irq: 55
[Apr23 20:46] device_write( 1 )
[  +0.000069] isr_thread( 55, 0000000054e8d4bc )
[  +4.282804] device_write( 0 )
[  +0.000066] isr_thread( 55, 0000000054e8d4bc )
root@raspi-dev:/home/quade/linux-mag#
```

3 So funktioniert der treiberbasierte GPIO-Zugriff im Test.

steht. Threaded-Interrupts bieten kürzere Latenzzeiten und ermöglichen ein Priorisieren der Interrupts **2**. Außerdem kann ein Interrupt-Thread Funktionen wie ein Schlafenlegen nutzen.

Verbinden Sie also die beiden GPIO-Leitungen über einen 10-kOhm-Widerstand miteinander, können Sie durch Schreiben auf die vom Treiber erstellte Gerätedatei den Interrupt auslösen und über den Log-Mechanismus im Terminalfenster anzeigen. So sehen Sie im Beispielcode auch die Zugriffe auf die GPIOs per `gpiod_get()` und `gpiod_put()`.

Die vom Treiber erstellte Gerätedatei trägt den Namen `myhw`. Zum Schreiben darauf aus dem Userland heraus kommt das Kommando `echo` zum Einsatz. Damit eine Flanke entsteht, müssen Sie Echo zweimal aufrufen **3**.

Gruppenarbeit

Benötigen Sie in einer Anwendung mehrere GPIOs, müssen diese nicht alle einzeln reservieren. Sowohl die Devicetree-Beschreibung als auch das GPIO-Interface im Kernel unterstützen GPIO-Arrays.

Im Devicetree werden die GPIOs, durch Kommas getrennt, einem einzelnen Namen zugeordnet (siehe Listing 2). Um mit einem Aufruf für alle unter einem Namen definierten GPIOs jeweils einen Deskriptor zu erhalten, verwenden Sie die Funktion `gpiod_get_array()` oder eine Variante davon. Sie erhalten eine Datenstruktur zurück, die ein Feld von Deskriptoren referenziert (man beachte den Plural in der Strukturdeklaration `struct gpiod_descs`). Stattdessen können Sie die Deskriptoren auch einzeln abholen, indem Sie einen Index mit angeben (`gpiod_get_index()`), dessen Zählung informatiktypisch bei null startet.

Die bisher genannten Reservierungsfunktionen geben entweder gültige Deskriptoren oder einen Fehlercode zurück, der über das Makro `IS_ERR()` geprüft wird. Es gibt auch Varianten dieser Funktionen, die ein `optional` im Namen tragen. Sie retournieren einen Null-Zeiger, falls das Reservieren fehlschlägt.

Fazit

Linux-typisch handelt es sich beim hier Gezeigten nur um einen Ausschnitt aus der überbordenden Funktionalität des Kernels, sowohl bezüglich des Devicetrees und des Zugriffs auf die dort abgelegten Informationen als auch bezüglich der Gpiolib-Schnittstelle. Es gibt auch hier deutlich mehr zu erkunden – also immer ran an den Linux-Quellcode! (*jlu*)

Listing 4: Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := myhw.o
else
KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)
default:
  $(MAKE) -C $(KDIR) M=$(PWD) modules
endif
%.dtbo: %.dts
  dtc -@ -W no-unit_address_vs_reg -I dts -O dtb -o $@ $<
clean:
  rm -rf *.ko .*.cmd *.mod.c .tmp_versions
  rm -rf modules.order *.mod.o
  rm -rf Module.symvers
```

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/50454

Listing 3: Zugriff auf GPIOs (myhw.c) (Fortsetzung von S. 76)

```
}
static void __exit mod_exit(void) {
  free_gpios();
  device_destroy(myhw_class, myhw_dev_number);
  class_destroy(myhw_class);
  cdev_del(&driver_object);
  unregister_chrdev_region(myhw_dev_number, 1);
  return;
}
module_init(mod_init);
module_exit(mod_exit);
MODULE_LICENSE("GPL");
```

Weitere Infos und interessante Links
www.lm-online.de/qr/50454

Die Autoren

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