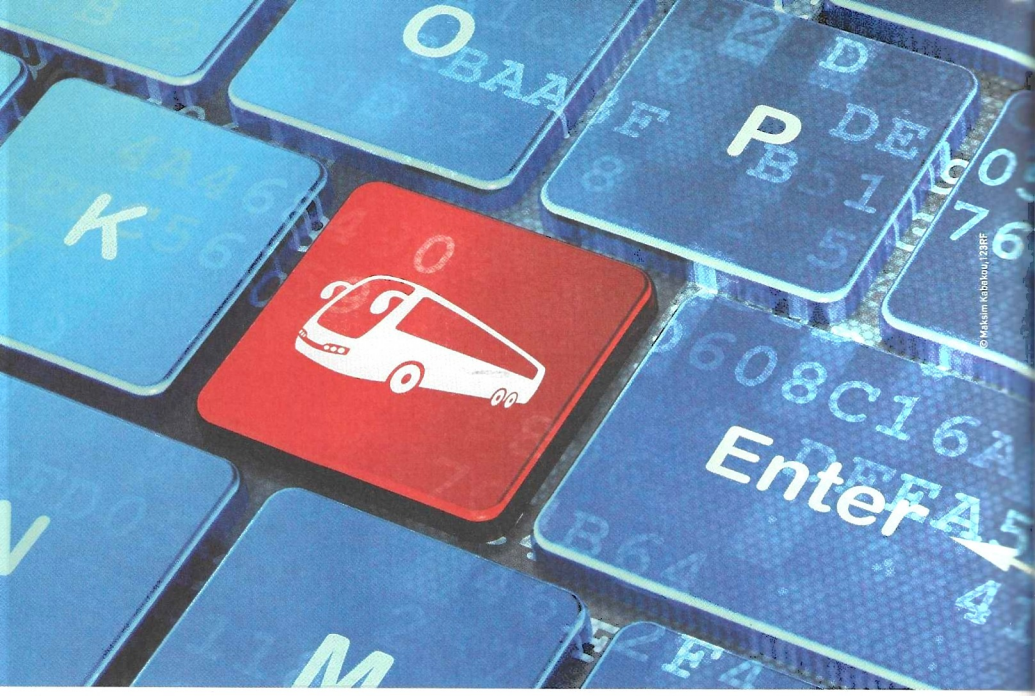


Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 114

Virtuelle Busfahrt

Für Geräte, die über keinen der bekannten Systembusse angeschlossen sind, hält Linux als Alternative den virtuellen Plattformbus bereit. Eva-Katharina Kunst, Jürgen Quade

Das Linux-Ökosystem zur Integration von Hardware wird immer leistungsfähiger. Mithilfe virtueller Dateisysteme informiert sich der Admin über den Zustand der Systemkomponenten, nimmt über dasselbe Interface zur Laufzeit neue Hardware in Betrieb und gliedert sie ins System ein. Technisch funktioniert das freilich nur dann reibungslos, wenn die Entwickler die Schnittstellen kennen und neue Hardware sauber darüber im System verankern sowie beim Entfernen der Hardware die Anker auch wieder einziehen.

Die Mehrzahl der Peripherie wird über USB, PCI, CAN, SPI oder I²C angebunden, also über standardisierte Bussysteme. Dafür bietet Linux Subsysteme an, in die der Kernel-Programmierer seinen Treibercode einklinkt. Die Subsysteme erkennen automatisiert neue Hardware, ordnen sie den passenden Gerätetreibern zu und aktivieren schließlich die Codesequenzen, die die neu angesteckte Hardware bedienen. Die bekommen eine Beschreibung der Peripherie übergeben, also Adresslagen des Speichers oder die Nummern verwendeter Interrupts. Aus Sicht des Treiberprogrammierers ist das alles frei Haus.

Mit Struktur

Einen nicht zu verachtenden Vorteil dieser Technik stellt die sich damit ergebende, strukturierte Software-Architektur dar: Die Beschreibung der Geräteparameter (Devices) ist sauber vom Code für den

Die Autoren

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

Zugriff auf die Geräte (Driver) getrennt. Peripherie, die sich keinem der unterstützten Bus- beziehungsweise Subsysteme zuordnen lässt, bleibt zunächst außen vor. Dazu zählen Geräte, die beispielsweise als System on Chip (SoC) direkt am Systembus hängen oder aber, wie die GPIOs beim Raspberry Pi, indirekt mit der CPU verbunden sind. Für diese Geräte hob Chefentwickler Linux Torvalds schon vor langer Zeit den Plattformbus aus der Taufe. Das virtuelle Bussystem dient als Sammelbecken für heimatlose Hardware und trennt Gerätebeschreibung von Treibercode 1. Eine Auswahl seiner Funktionen zeigt die Tabelle Funktionen des Platform-Device-Subsystems.

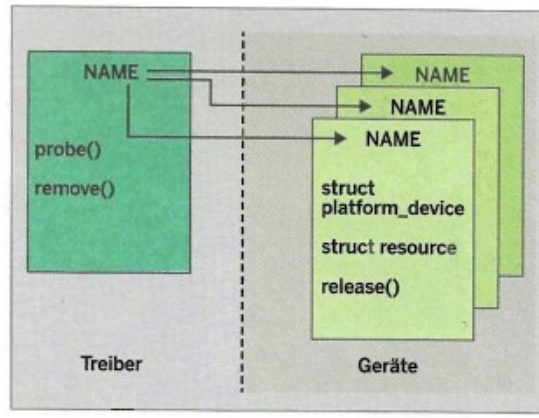
So meldet sich ein Plattformtreiber mit den in der Struktur `struct platform_driver` hinterlegten Funktionen und Attributen über die Funktion `int platform_driver_register(struct platform_driver *)` beim Plattformsubsystem an. Eine zur Übernahme eines Geräts benötigte Gerätebeschreibung lässt sich mithilfe der Funktion `int platform_device_register(struct platform_device *)` im Kernel hinterlegen 2.

Loslassen als Pflicht

Zur Gerätebeschreibung gehört zudem obligatorisch eine `release`-Funktion, die dazu dient, eine Race Condition zu verhindern. Der Kernel ruft sie auf, wenn intern alle mit dem Plattformgerät verbundenen Referenzen aufgelöst wurden, oder – um es konkreter zu formulieren – wenn alle mit dem Plattformgerät verbundenen Einträge im Sys-Filesystem wieder entfernt wurden. Das macht der Kernel freilich erst dann, wenn keine Instanz mehr auf diese Einträge zugreift. Dasselbe gilt für den Gerätetreiber: So darf der Treibercode (`hello_device.c`) ebenfalls nicht vorzeitig aus dem Speicher verschwinden.

Ein Completion-Objekt (siehe Kasten Completion-Objekt) löst dieses Dilemma. Auf den Aufruf der Release-Funktion über `wait_for_completion()` hin wird in der Funktion `hello_device_exit()` zunächst geschlafen. Die Release-Funktion selbst signalisiert dann per `complete()`, dass alle potenziellen Zugriffe abgeschlossen und die zugehörigen Datenstrukturen aus dem Kernel entfernt wurden. Im Beispiel aus Listing 1 beendet erst der Aufruf von `complete()` die Funktion `hello_device_exit()`, und erst danach entfernt der Kernel den gesamten Treibercode aus dem Hauptspeicher.

Die Zuordnung zwischen Gerät (Device) und Plattformtreiber vollzieht sich über den in der `struct platform_device` hinterlegten Namen. Dazu setzt der Kernel auf einen einfachen Textvergleich. Alternativ kann man auch eine Tabelle mit IDs (Strings) verwenden, die man in einer `struct platform_device_id` spezifiziert.



1 Der Treibercode und die Gerätebeschreibung sind beim Plattformbus voneinander getrennt.

Diese Struktur wird sowohl vom Gerätetreiber selbst (`id_table`) als auch von den Geräten (`id_entry`) definiert.

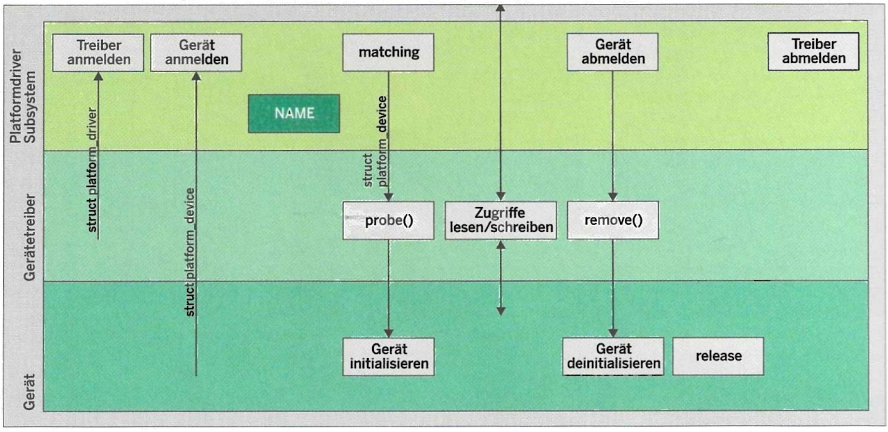
Reale Geräte spezifizieren außerdem ihre Hardware-Ressourcen in einer `struct resource`. Bei dem in Listing 1 definierten virtuellen Gerät *Hello* werden beispielhaft Ressourcen für IO-Speicher sowie für einen Interrupt reserviert. Die Anzahl der Ressourcen ermittelt das Makro `ARRAY_SIZE()` automatisiert. Virtuelle Geräte benötigen eigentlich keine Ressourcen; folgerichtig könnte man in diesem Fall das Attribut `num_resources` auf null setzen.

Den Kernel machen lassen

Ein Gerät hat darüber hinaus die Möglichkeit, weitere Attribute festzulegen. Pfiffige

Funktionen des Platform-Device-Subsystems

Funktion	Kurzbeschreibung
`module_platform_driver();`	Makro, das zur Initialisierungs- und zur Deinitialisierungsfunktion expandiert
`int platform_driver_register(struct platform_driver *drv);`	Plattformtreiber anmelden
`void platform_driver_unregister(struct platform_driver *drv);`	Plattformtreiber abmelden
`int platform_device_register(struct platform_device *);`	Plattformgerät anmelden
`void platform_device_unregister(struct platform_device *);`	Plattformgerät abmelden
`struct resource *platform_get_resource(struct platform_device *pdev, unsigned int type, unsigned int n);`	Daten zur Ressource auslesen
`struct resource *platform_get_resource_byname(struct platform_device *pdev, unsigned int type, const char *name);`	Daten zur Ressource auslesen
`int platform_get_irq(struct platform_device *pdev, unsigned int n);`	Interrupt-Nummer auslesen



2 **Das Zusammenspiel zwischen Kernel-Subsystem, Plattformtreiber und Gerät in der Übersicht.**

Entwickler nutzen das, um automatisiert und ohne zusätzlichen Aufwand eine Gerätedatei anlegen zu lassen, über die eine Applikation auf die Hardware zugreift. Der Entwickler lässt sich hierfür vom Kernel eine Gerätenummer zuweisen (Funktion `alloc_chrdev_region`) und belegt das Attribut `dev.devt` damit. Den Rest erledigt das Plattformsubsystem in Zusammenarbeit mit dem im Userland laufenden Udevd.

Der Name der Gerätedatei entspricht dem Namen des Geräts plus einer laufenden Nummer. Gibt es nur ein Gerät – in diesem Fall ist die Nummer obsolet – wird das Attribut `id` der Struktur `struct platform_device` auf –1 gesetzt. Vermeiden Sie einen klassischen Programmierfehler: Vergessen Sie nicht, im Fehlerfall oder beim Entladen des Treibers die reservierte Gerätenummer wieder freizugeben.

Geräte spezifizieren also im Wesentlichen ihren Namen sowie die Release-Funktion und übergeben sie dem Plattformsubsystem. Plattformtreiber definieren neben ihrem Namen die Adresse einer `probe`- und einer `remove`-Funktion. Daneben können sie noch Adressen für Power-Management-Funktionen übergeben.

Die `probe`-Funktion ruft Linux auf, sobald sich ein Gerät mit dem Namen des Treibers anmeldet. Die Funktion bekommt alle zum Gerät gehörenden Daten übergeben, insbesondere die vom Gerät benötigten Ressourcen. Gibt die Funktion `probe` des eigentlichen Treibers null zurück, gehören Treiber und Gerät zusammen, und das Linux-Gerätemodell erstellt die zugehörige Verknüpfung im Sys-Filesystem **4**. Einen Rückgabewert ungleich null wertet der Kernel hingegen als Fehler, und das Device wird durch diesen Treiber nicht bedient.

Completion-Objekt

Beim Completion-Objekt (`struct completion`) handelt es sich um ein Synchronisationselement, mit dem das Ende einer Codesequenz abgewartet wird. Der zu überwachende Code meldet per `void complete(struct completion *)` die finale Abarbeitung. Über die Methode `void wait_for_completion(struct completion *)` schläft eine Instanz, falls das Ende der Abarbeitung noch nicht signalisiert wurde. Technisch verbirgt sich hinter dem Completion-Objekt vereinfacht dargestellt eine Integer-Variable, die anfänglich mit null initialisiert ist. Diese inkrementiert `complete()` **3** und weckt darüber hinaus eine möglicherweise auf das Objekt schlafende Instanz auf. `wait_for_completion()` überprüft den Wert der Variablen. Ist die Variable null, wird die Instanz schlafen gelegt und die Variable dekrementiert.

Ressource auslesen

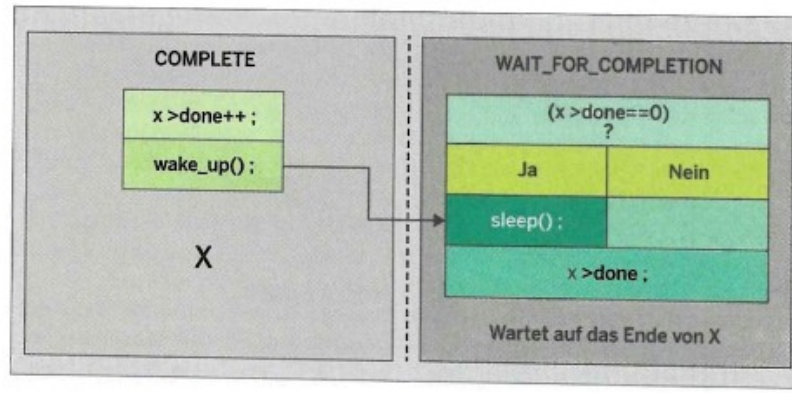
Innerhalb der `probe`-Funktion kann der Treiber die notwendigen Initialisierungen vornehmen, insbesondere die benötigten Ressourcen reservieren. Um die Adresslagen und Interrupt-Nummern aus der beim Aufruf der `probe`-Funktion übergebenen Gerätebeschreibung auszulesen, stellt der Kernel die Funktionen `platform_get_resource()`, `platform_get_resource_byname()` und `platform_get_irq()` zur Verfügung. Linux unterscheidet dabei die Ressourcentypen IO, Memory, Register, Interrupts, DMA und Bus. Die Funktionen liefern jedoch nur die Werte zurück; das Reservieren etwa über eine Funktion `request_irq()` oder `request_mem_region()` obliegt dem Treiber.

Soll das Gerät als Character Device Anwendungen für den lesenden und schreibenden Zugriff zur Verfügung stehen, muss der Gerätetreiber noch die entsprechenden Lese- und Schreib-

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/44070

funktionen bereitstellen und deren Adressen in einer `struct fileoperations` eintragen – das Standardvorgehen bei zeichenorientierten Treibern. Um dem Kernel die Liste mit den Zugriffsfunktionen übergeben zu können, benötigt man ein Objekt vom Typ `struct cdev`. Das erhält der Programmierer durch Aufruf von `cdev_alloc()`. Das Attribut `ops` wird mit der Adresse der Funktionsliste initialisiert und schließlich dem Kernel durch Aufruf von `cdev_add()` übergeben. Schlägt Letzteres fehl, erfolgt die Freigabe durch einen Aufruf der Funktion `kobject_put()`. Daneben erhält `cdev_add` auch die Gerätenummer übergeben, die man der Datenstruktur zum Gerät entnimmt.

Wird das Gerät wieder entfernt, gibt die `remove`-Funktion das Geräteobjekt (`struct cdev`) durch Aufruf von `cdev_del()` wieder frei – das war's.



3 **Die interne Realisierung des Completion-Objekts.**

Um die Trennung zwischen Treiber und Gerät besser demonstrieren zu können, zeigen Listing 1 und Listing 2 jeweils die getrennten Kernel-Quellcodes für den Treiber und für das Gerät. Der Code für das Gerät besteht aus nur drei Funktionen: `hello_device_init()` wird beim Laden des Gerätetreibers aufgerufen, `hello_de-`

Listing 1: `hello_device.c`

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/platform_device.h>

#define DEVICE_NAME "hello"
#define HELLO_MEM_START 0x10000
#define HELLO_MEM_END   0x1ffff
#define HELLO_IRQ       42

static DECLARE_COMPLETION(
dev_obj_is_free);

static void hello_release(
struct device *dev)
{
  printk("DEVICE: hello_
release()\n");
  complete(&dev_obj_is_free);
}

static struct resource
hello_resources[] = {
  {
    .start  = HELLO_MEM_START,
    .end    = HELLO_MEM_END,
    .flags  = IORESOURCE_MEM,
  },
  {
    .start  = HELLO_IRQ,
    .end    = HELLO_IRQ,
    .flags  = IORESOURCE_IRQ,
  }
};

static struct platform_device
hello_device = {
  .name           = DEVICE_NAME,
  .id             = -1, // don't
add a number to the device name
  .num_resources  = ARRAY_SIZE(
hello_resources),
  .resource       = hello_
resources,
  .dev = {
    .release = hello_release,
  }
};

static int __init hello_device_
init(void)
{
  int hello_device_number;

  printk(KERN_INFO "DEVICE:
platform_device_register(%s)\n",
DEVICE_NAME);
  if (alloc_chrdev_region(&hello_
device_number, 0, 1, "hello") < 0)
    return -1;
  hello_device.dev.devt =
hello_device_number;
  platform_device_register(
&hello_device);
  return 0;
}

static void __exit
hello_device_exit(void)
{
  printk(KERN_INFO "DEVICE:
platform_device_unregister(%s)\n",
DEVICE_NAME);
  unregister_chrdev_region(
hello_device.dev.devt, 1);
  platform_device_unregister(
&hello_device);
  wait_for_completion(
&dev_obj_is_free);
}

module_init(hello_device_init);
module_exit(hello_device_exit);
MODULE_LICENSE("GPL");
```

vice_exit beim Entladen. Die Release-Funktion `hello_release()` besteht, wie bereits erwähnt, im Wesentlichen aus dem Aufruf von `complete()`.

Der Quellcode zum Gerätetreiber (Listing 2) weist eine Besonderheit auf: Es fehlen die für ein Kernel-Modul üblichen Makros `module_init()` und `module_exit()` inklusive der davon referenzierten Funktionen. Tatsächlich werden die (De-)Initialisierungsfunktionen des Kernel-Moduls über das Makro `module_platform_driver()` generiert. Das macht den Code übersichtlicher, findet in diesen beiden Funktionen im konkreten Fall doch nur das An- und Abmelden beim Plattformsubsystem statt. Dem Makro `module_platform_driver()` übergibt man den Namen der Datenstruktur, die die probe- und die remove-Funktion sowie – ganz wichtig – den Namen des Treibers zum Matching mit den Geräten enthält.

Hello again

Der vorliegende Treiber realisiert das virtuelle Gerät `/dev/hello`, das beim lesenden Zugriff den String `hello, world` zurückgibt. Die Lesefunktion `driver_read()` kopiert den String `hello, world` per `copy_to_user()` vom Kernelspace in den Speicher der Applikation, korrigiert den internen Lese-Offset, und gibt die Anzahl der kopierten Bytes zurück. Die am Anfang der Funktion stehende Minimumfunktion stellt sicher, dass nicht mehr Daten kopiert werden, als zur Verfügung stehen respektive die Applikation angefordert hat.

Stellen Sie vorab sicher, dass die zum Generieren von Kernel-Code notwendigen Dateien auf dem System vorhanden sind. Auf einem Raspberry Pi beispiels-

Listing 2: `hello_driver.c`

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include <linux/cdev.h>
#include <linux/platform_device.h>
#include <linux/of.h>

#define DRIVER_NAME "hello"

static char hello_world[]=
"hello, world\n";
static struct cdev *cdevobj;

static ssize_t driver_read(struct
file *instanz, char __user *user,
size_t count, loff_t *offset)
{
  unsigned long not_copied,
to_copy;

  printk(KERN_INFO "DRIVER:
driver_read()\n");
  to_copy =
min(count,strlen(hello_world)+1);
// incl. terminating zero
  not_copied = copy_to_user(user,
hello_world, to_copy);
  *offset = *offset +
(to_copy-not_copied);
  return to_copy - not_copied;

}

struct file_operations hello_fops
= {
  .owner = THIS_MODULE,
  .read  = driver_read,
};

static int hello_probe(struct
platform_device *pdev)
{
  struct resource *hello_res;
  int irq;

  printk(KERN_INFO "DRIVER:
hello_probe()\n");

  cdevobj = cdev_alloc();
  if (cdevobj==NULL) {
    return -EIO;
  }
  cdevobj->ops    = &hello_fops;
  cdevobj->owner = THIS_MODULE;
  if (cdev_add(cdevobj,pdev->dev.
devt,1)) {
    kobject_put( &cdevobj->kobj
);
    return -EIO;
  }

  // get resources
  hello_res = platform_get_
resource(pdev, IORESOURCE_MEM, 0);
  if (hello_res) {
    printk(KERN_INFO "DRIVER:
Memory: %llx - %llx\n", hello_
res->start, hello_res->end);
  }
  irq = platform_get_irq(
pdev, 0);
  if (irq) {
    printk(KERN_INFO "DRIVER:
IRQ: %d\n", irq);
  }
  return 0;
}

static int hello_remove(
struct platform_device *pdev)
{
  printk(KERN_INFO "DRIVER:
hello_remove()\n");
  cdev_del(cdevobj);
  return 0;
}

static struct platform_driver
hello_driver =
{
  .probe = hello_probe,
  .remove = hello_remove,
  .driver = {
    .name = DRIVER_NAME,
  }
};

module_platform_driver(
hello_driver);

MODULE_LICENSE("GPL");
```

weise rufen Sie dazu sudo apt install build-essentials bison flex raspberrypi-kernel-headers auf. Dann kopieren Sie die Quellcodedateien hello_device.c (Listing 1), hello_driver.c (Listing 2) und das Makefile (Listing 3) in ein eigenes Verzeichnis und rufen dann nur make auf. In Abbildung 5 sehen Sie den Vorgang, wenn alles gut geht (ohne die Ausgaben der Funktion driver_read()).

Die beiden Treiber laden Sie per sudo insmod hello_driver.ko und sudo insmod hello_device.ko. Idealerweise öffnen Sie parallel ein zweites Terminal und beobachten dort durch Aufruf des Kommandos sudo tail -f /var/log/kern.log die Ausgaben der Kernel-Module und damit die internen Abläufe.

Nach dem Laden der Module existiert die Gerätedatei /dev/hello. Den Zugriff testen Sie per sudo cat /dev/hello, wobei Sie die Ausgabe über [Strg]+[C] beenden. Um die beiden Treibergeister wieder loszuwerden, bemühen Sie die Kommandos sudo rmmod hello_device und sudo rmmod hello_driver.

```
root@raspberrypi:/sys/bus/platform/devices/hello# ls -lrt
total 0
-rw-r--r-- 1 root root 4096 Dec 30 20:57 uevent
drwxr-xr-x 2 root root    0 Dec 30 20:58 power
lrwxrwxrwx 1 root root    0 Dec 30 20:58 subsystem -> ../../../bus/platform
-r--r--r-- 1 root root 4096 Dec 30 20:58 modalias
-rw-r--r-- 1 root root 4096 Dec 30 20:58 driver_override
lrwxrwxrwx 1 root root    0 Dec 30 20:58 driver -> ../../../bus/platform/drivers/hello
-r--r--r-- 1 root root 4096 Dec 30 20:58 dev
root@raspberrypi:/sys/bus/platform/devices/hello# cat dev
234:0
```

4 Die Einträge zum Gerät im Sys-Filesystem werden automatisiert angelegt.

```
root@raspberrypi:/home/quade/platform# make
make    -C /lib/modules/5.4.79-v7l+/build       M=/home/quade/platform modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.79-v7l+'
  CC [M]  /home/quade/platform/hello_device.o
  CC [M]  /home/quade/platform/hello_driver.o
  Building modules, stage 2.
  MODPOST 2 modules
  CC [M]  /home/quade/platform/hello_device.mod.o
  LD [M]  /home/quade/platform/hello_device.ko
  CC [M]  /home/quade/platform/hello_driver.mod.o
  LD [M]  /home/quade/platform/hello_driver.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.79-v7l+'
root@raspberrypi:/home/quade/platform# insmod hello_device.ko
root@raspberrypi:/home/quade/platform# insmod hello_driver.ko
root@raspberrypi:/home/quade/platform# tail -n2 /var/log/kern.log
Dec 30 20:45:49 raspberrypi kernel: [22025.170645] DEVICE: platform_device_register( hello )
Dec 30 20:45:54 raspberrypi kernel: [22030.280422] DRIVER: hello_probe()
root@raspberrypi:/home/quade/platform# dd if=/dev/hello count=3 2>/dev/null # dreimal lesen
hello, world
hello, world
hello, world
root@raspberrypi:/home/quade/platform# rmmod hello_device
root@raspberrypi:/home/quade/platform# rmmod hello_driver
root@raspberrypi:/home/quade/platform# tail -n6 /var/log/kern.log
Dec 30 20:46:17 raspberrypi kernel: [22052.804941] DRIVER: driver_read()
Dec 30 20:46:17 raspberrypi kernel: [22052.804979] DRIVER: driver_read()
Dec 30 20:46:17 raspberrypi kernel: [22052.804995] DRIVER: driver_read()
Dec 30 20:46:25 raspberrypi kernel: [22060.736963] DEVICE: platform_device_unregister( hello )
Dec 30 20:46:25 raspberrypi kernel: [22060.737196] DRIVER: hello_remove()
Dec 30 20:46:25 raspberrypi kernel: [22060.737319] DEVICE: hello_release()
```

5 Der Beispieltreiber verdeutlicht die Interaktionen.

Fazit

Wer systemkonform Hardware in den Linux-Kernel integriert, die nicht über einen klassischen Bus angekoppelt wird, der setzt auf den virtuellen Plattformbus. Der ermöglicht die Trennung des Codes, des eigentlichen Gerätetreibers und der Daten, die das Device beschreiben. Diese saubere Struktur beschert eine übersichtliche Codebasis. Seine volle Stärke spielt der Ansatz erst in Verbindung mit einem Device Tree aus – aber das wird Thema einer künftigen Kern-Technik-Folge. (*jlu*)

Weitere Infos und interessante Links
www.lm-online.de/qr/44070

Listing 3: Makefile

```
ifneq ($(KERNELRELEASE),)
  obj-m := hello_device.o hello_driver.o
else
  KDIR := /lib/modules/$(shell uname -r)/build
  PWD  := $(shell pwd)
default:
  $(MAKE)  -C $(KDIR)  M=$(PWD) modules
endif

clean:
  rm -rf *.ko *.mod *.mod.* *.o modules.order
  rm -rf Module.symvers .*.cmd
```