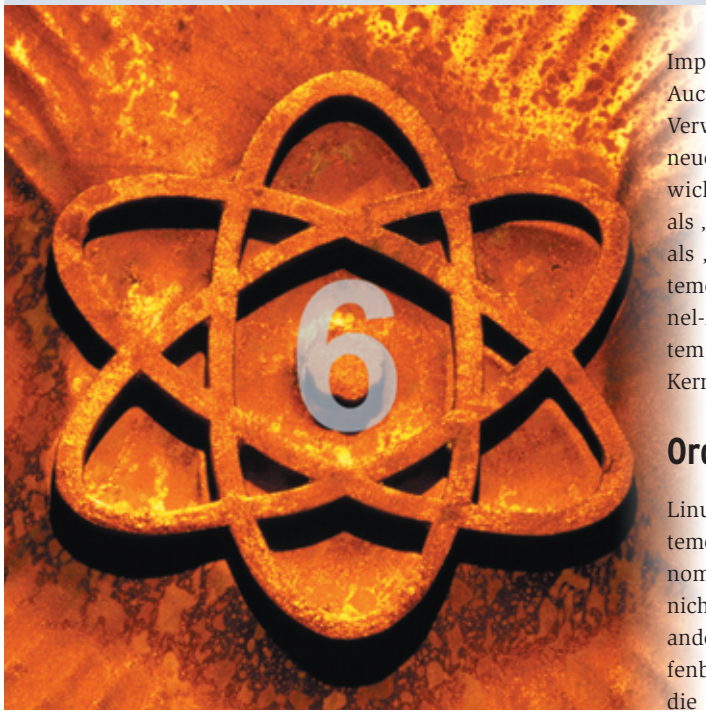
Kernel- und Treiberprogrammierung mit dem künftigen Kernel 2.6 – Folge 6

Kern-Technik

Das neue Gerätemodell ordnet Hardware nach Kategorien und bildet die Struktur im Sys-Filesystem ab. Hält sich der Entwickler daran, unterstützt sein Treiber automatisch Powermanagement. Das Gerätemodell besteht aus einem Kernel-API und einem Laufzeitsystem. Eva-Katharina Kunst, Jürgen Quade



Treiber für Hardware machen den größten Teil des Kernels aus. Linux 2.6 führt neue Schnittstellen ein, mit deren Hilfe sich Treiber besser in das Gesamtsystem integrieren: das Gerätemodell und das Sys-Filesystem. Die Informationen darüber sind allerdings spärlich. Kein Wunder, die Komponenten sind brandneu und stecken mitten in der Entwicklung.

API und Laufzeitsystem

Wer in eigenen Gerätetreibern das neue Modell unterstützen möchte, muss im Linux-Quellcode gelegentlich Variablen, Interfaces und Aufrufsemantik recherchieren. Im Kernelbaum finden sich einige Texte, die das Gerätemodell beschreiben (»Documentation/driver-model«), aber in manchen Details von der Implementierung abweichen. Auch beim Namen herrscht Verwirrung. So erscheint das neue Gerätemodell in der Entwicklerdiskussion teilweise als „device model“, aber auch als „driver model“. Das Gerätemodell ist zugleich ein Kernel-API und ein Laufzeitsystem, das Gerätestrukturen im Kernel verwaltet.

Ordnung durch Sysfs

Linus Torvalds hat das Gerätemodell in den Kernel aufgenommen, obwohl es noch nicht ausgereift ist. Wie auch andere Entwickler setzt er offenbar hohe Erwartungen in die neue Komponente. Sie bringt Ordnung in die Gerätelandschaft, hilft beim Powermanagement und verwaltet künftig – als Ersatz für das Device-Filesystem (Devfs) – die Gerätedateien [1].

Das Gerätemodell bildet ab, wie die Prozessoren eines Systems mit den Controllerbausteinen und diese wiederum mit den Peripheriekarten und mit sonstiger Hardware zusammenhängen. Außerdem verleiht das Modell den zugehörigen Softwarekomponenten eine Struktur, beispielsweise den Gerätetreibern. Anhand der im Gerätemodell gesammelten Information ist der Kernel in der Lage, gezielt Powermanagement zu betreiben. So gibt es die Reihenfolge vor, in der das Betriebssystem die Hardware abschaltet: Zunächst müssen die Geräte an einem Bus heruntergefahren werden, bevor der Bus selbst und schließlich der Prozessor in einen Stromsparzustand übergehen.

Das Sys-Filesystem (Sysfs) ist wie das Proc-Filesystem virtuell: Der Kernel erzeugt die Verzeichnisse und Dateien dynamisch, keine Festplatte muss sie speichern. Der durch das Sys-Filesystem aufgespannte Verzeichnisbaum spiegelt die Struktur der Hardware und der zugehörigen Software des jeweiligen Linux-2.6-Systems wider. Über das Sysfs kommt auch der Anwender mit dem Gerätemodell in Berührung. Er muss es nur noch in den Verzeichnisbaum einhängen:

```
mount -t sysfs sysfs /sys
```

Um in Userspace-Programmen Informationen aus dem Sys-Filesystem zu lesen, gibt es bereits die Bibliothek »libsysfs« [4]. Abbildung 1 zeigt, in welche Kategorien sich Geräte und Treiber aufteilen und welche zusätzlichen Schnittstellen, zum Beispiel für den Download von Firmware [2], zur Verfügung stehen.

Mehrfacheinträge

Entsprechend den Kategorien erscheinen einzelne Geräte und Treiber im Sys-Filesystem an mehreren Stellen. Eine PCI-Netzwerkkarte sortiert das Gerätemodell sowohl in die Kategorie Bus (PCI) als auch in die Kategorie Geräteklasse/Netzwerk ein. Mehrfacheinträge realisiert es mit Hilfe symbolischer Links.

Das Sys-Filesystem zeigt nicht nur die Gerätestruktur. Die Blätter des Baums sind lesbare und beschreibbare Pseudodateien. Wie beim Proc-Filesystem lassen sich hierüber Attribute der Hard- und Software auslesen und setzen. Ob Herstellername, Versionsnummer, Gerätezustand oder die Übertragungsstatistik einer Netzwerkkarte – die Nutzungsmöglichkeiten sind fast unbegrenzt.

Das An- und Abmelden von Geräten und Treibern beim Gerätemodell vollzieht sich oft implizit. PCI-, USB- oder Netzwerksubsystem übernehmen diese Aufgabe. Schon wenn ein Treiber die in [3] vorgestellte Funktion »register_chrdev()« aufruft, erstellt der Kernel einen Eintrag. Dennoch sollte der Programmierer das Gerätemodell explizit unterstützen (siehe Abbildung 2).

Will er les- und schreibbare Attribute erzeugen, sollte er dafür das Gerätemodell verwenden. In anderen Fällen ist es obligatorisch, nämlich wenn

- Geräte nicht über USB, PCI oder über ein anderes, standardisiertes Bussystem angeschlossen werden,
- neue Bussysteme eingefügt oder
- wenn neue Geräteklassen definiert werden sollen.

Der Entwickler muss also in seinem Code den Treiber und das Gerät beim Gerätemodell anmelden und sinnvolle Attribute freischalten.

Treiber anmelden

Wenn sich der Treiber beim PCI-, USB- oder IDE-Subsystem anmeldet, hängt es ihn im Sys-Filesystem automatisch unterhalb des zugehörigen Busobjekts ein. Das Gleiche gilt für I2C- (zwischen Hardware-Chips), EISA- und Microchannel-Busse. In allen anderen Fällen aber muss der Treiber ein neues Bussystem definieren oder sich selbst unterhalb des so genannten Plattformbusses einklinken. Weil innerhalb des Gerätemodells jeder Treiber sich einem Bussystem zuordnen muss, dient der Plattformbus als Container für alle Treiber, die nicht zu anderen Bussen gehören.

Das Einhängen des Treibers unterhalb des Plattformbusses folgt dem Standard: Zunächst definiert und initialisiert der Programmierer ein Objekt, in der Sprache C also eine Datenstruktur. Dann übergibt er dieses Objekt dem Kernel, der den Rest erledigt. Wird der Treiber nicht mehr gebraucht, muss er sich beim Kernel abmelden. Der Treiber in Listing 1 beispielsweise definiert die Datenstruktur »struct device_driver« und belegt initial die Elemente »name« und »bus« (Zeilen 29 bis 32). Der Datentyp »struct device_driver« und das Symbol »platform_bus_type« sind in der Headerdatei »linux/device.h« definiert.

Bei der Initialisierung übergibt der Treiber durch den Aufruf der Funktion »dri-

Listing 1: Treiber für ein virtuelles Gerät

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/device.h>
06 #include <linux/completion.h>
07
08 #define DRIVER_MAJOR 240
09
10 MODULE_LICENSE("GPL");
11
12 static struct file_operations Fops;
13 static DECLARE_COMPLETION( DevObjectIsFree );
14 static int Frequenz; // Zustandsvariable des Gerätes
15
16 static void mydevice_release( struct device *dev )
17 {
18     complete( &DevObjectIsFree );
19 }
20
21 struct platform_device mydevice = {
22     .name   = "MyDevice",
23     .id     = 0,
24     .dev = {
25         .release = mydevice_release,
26     }
27 };
28
29 static struct device_driver mydriver = {
30     .name = "MyDevDrv",
31     .bus = &platform_bus_type,
32 };
33
34 static ssize_t ReadFreq( struct device *dev, char *buf )
35 {
36     sprintf(buf, "Frequenz: %d", Frequenz );
37     return strlen(buf)+1;
38 }
39
40 static ssize_t WriteFreq( struct device *dev, const char *buf, size_t
   count )
41 {
42     Frequenz = simple_strtoul( buf, NULL, 0 );
43     return strlen(buf)+1;
44 }
45
46 static DEVICE_ATTR( freq ,S_IRUGO|S_IWUGO, ReadFreq, WriteFreq );
47
48
49 static int __init DrvInit(void)
50 {
51     if(register_chrdev(DRIVER_MAJOR, "MyDevice", &Fops) == 0) {
52         driver_register(&mydriver);           // register the driver
53         platform_device_register( &mydevice );// register the device
54         mydevice.dev.driver = &mydriver;      // now tie them together
55         device_bind_driver( &mydevice.dev );  // links the driver to
                                                    the device
56         device_create_file( &mydevice.dev, &dev_attr_freq ); //
   attributes
57         return 0;
58     }
59     return -EIO;
60 }
61
62 static void __exit DrvExit(void)
63 {
64     device_remove_file( &mydevice.dev, &dev_attr_freq );
65     device_release_driver( &mydevice.dev );
66     platform_device_unregister( &mydevice );
67     driver_unregister(&mydriver);
68     unregister_chrdev(DRIVER_MAJOR,"MyDevice");
69     wait_for_completion( &DevObjectIsFree );
70 }
71
72 module_init( DrvInit );
73 module_exit( DrvExit );
```

ver_register()« das Objekt an den Kernel (Zeile 52). Sobald »driver_register()« abgearbeitet ist, erscheint in einem gemounteten Sys-Filesystem das neue Verzeichnis »/sys/bus/platform/drivers/MyDevDrv/«. Die Funktion »driver_unregister()« entfernt diesen Eintrag wieder (Zeile 67).

Geräte anmelden

Module müssen nicht nur den Treiber beim Laufzeitsystem des Gerätemodells anmelden, sondern auch das Gerät selbst, wenn es durch den Treiber und nicht durch ein PCI-, USB- oder sonstiges Subsystem automatisch erkannt wird. Solche Geräte werden im Modell als Plattform- oder Systemgeräte bezeichnet. Systemgeräte sind im Wesentlichen die Bausteine des Rechnerkerns selbst, also die CPU, der Interrupt-Controller oder auch der Timer. Einträge hierfür finden sich unter dem Ordner »/sys/devices/system/«. Alle übrigen Geräte sind Plattformgeräte. Für sie werden die Einträge im Verzeichnis »/sys/devices/legacy/« erstellt.

Ein Modul definiert ein Gerät als Objekt und initialisiert es bei dessen Anmeldung, genauso wie eben bei den Treibern gezeigt (siehe Listing 1, Zeilen 21 bis 27). Allerdings muss der Code für ein Gerät zudem eine »release()«-Funktion enthalten (Zeile 16). Diese ruft der Kernel auf, sobald das Gerätemodell das Geräteobjekt nicht mehr benötigt. Das Modul wiederum muss sicherstellen, dass der zum Objekt gehörige Speicher erst nach Aufruf der »release()«-Funktion freigegeben wird. Das lässt sich am einfachsten mit einem Completion-Objekt [3] bewerkstelligen (Zeilen 13, 18 und 69), auf das bei der Deinitialisierung des Treibers gewartet wird.

Die Funktion »platform_device_register()« übergibt das Objekt dem Gerätemodell (Zeile 53). Daraufhin erstellt der Kernel das Verzeichnis »/sys/devices/legacy/MyDevice/«. Die Routine »platform_device_unregister()« lässt ihn den Verzeichniseintrag anschließend wieder entfernen (Zeile 66).

So einfach ist es, Treiber- und Geräteobjekte im Gerätemodell zu registrieren – allerdings stehen beide Objekte noch in keiner Beziehung zueinander. Das Gerätemodell weiß also noch nicht, dass das Gerät durch den Beispieltreiber bedient wird, und umgekehrt, dass dieser Treiber für das soeben definierte Gerät verantwortlich ist.

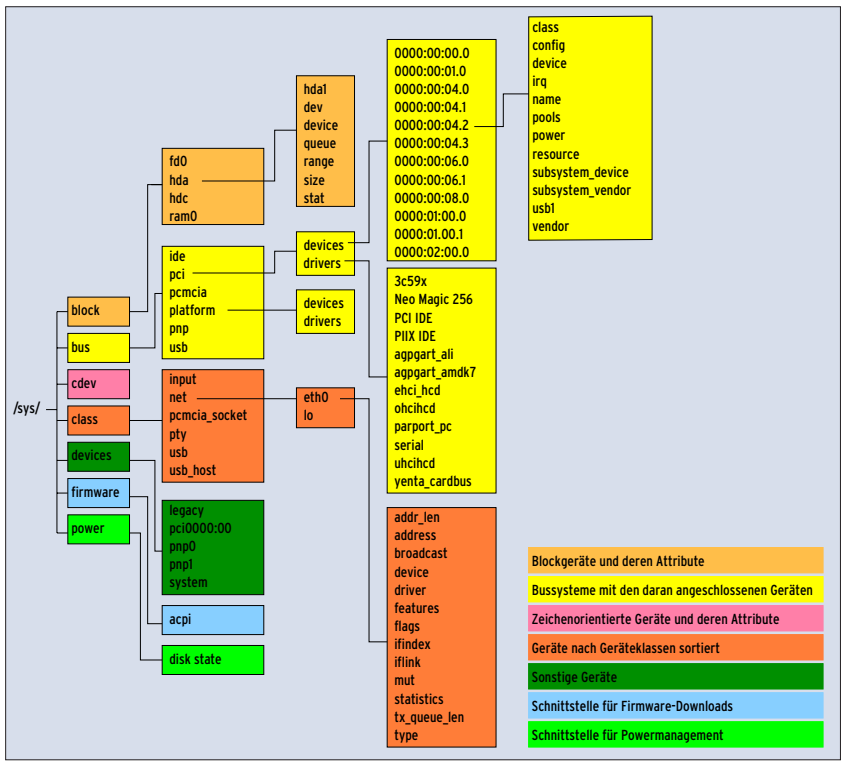


Abbildung 1: Das Sys-Filesystem kategorisiert die Informationen des Gerätemodells und stellt sie in einem Verzeichnisbaum dar. Für jedes Objekt, etwa einen Treiber, existiert ein eigenes Verzeichnis. Die Objektattribute, zum Beispiel Zustandsdaten des Treibers, sind als Dateien repräsentiert.

Verknüpfungspunkte zum Geräteobjekt schaffen

Die Funktion »device_bind_driver()« stellt diese Verbindung her. Parameter der Funktion ist die Adresse des Geräteobjekts. Vor ihrem Aufruf muss allerdings die Adresse des Treiberobjekts »mydriver« im Geräteobjekt »mydevice« eingetragen sein:

```
mydevice.dev.driver = &mydriver;
device_bind_driver( &mydevice.dev );
```

Das Gerätemodell erzeugt einen Link vom Verzeichnis »/sys/bus/platform/drivers/MyDriver« hin zu »/sys/devices/legacy/MyDevice«.

Bisher hat das Gerätemodell Verzeichniseinträge im Sys-Filesystem verändert. Richtig spannend wird es, wenn sich Parameter und Zustände von Gerät und Treiber über Attributdateien einfach lesen und setzen lassen. Dazu ist zweierlei notwendig: Erstens sind die Funktionen zu definieren, die der Kernel aufruft, wenn der Anwender ein Attribut liest oder schreibt. Zweitens müssen die Attributdateien als solche im Sys-Filesystem angelegt werden (siehe Kasten „Attribute programmieren“).

Lesefunktionen für Attributdateien besitzen immer zwei Parameter. Der erste spezifiziert das zugehörige Objekt, beispielsweise das Geräteobjekt oder das Treiberobjekt. Der zweite Parameter ist die Adresse eines Speicherbereichs, in dem die Attributdaten abgelegt werden sollen. Da normalerweise über das Sys-Filesystem keine großen Datenmengen ausgetauscht werden, ist dieser Puffer in seiner Länge auf eine Speicherseite limitiert. Durch diese feste Grenze ist es einfacher, in eigenen Treibern das Sys-Filesystem zu unterstützen.

Daten, die Treiber und Userspace austauschen, sollten in Ascii kodiert sein. Auf diese Weise kann man mit Systemprogrammen einfach auf die Attribute zugreifen. So schreibt »echo "1" > /sys/legacy/MyDevice/attr« den Wert »1« in das Attribut »attr«, während »cat /sys/legacy/MyDevice/attr« den entsprechen-

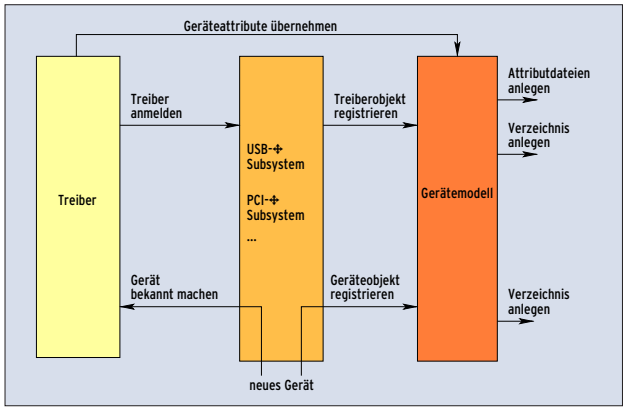


Abbildung 2: Wenn ein Treiber eigene Attribute unterstützen möchte oder wenn er sich nicht bei einem Treiber-Subsystem (wie PCI, USB) anmeldet, muss der Programmierer das Gerätemodell explizit ansprechen.

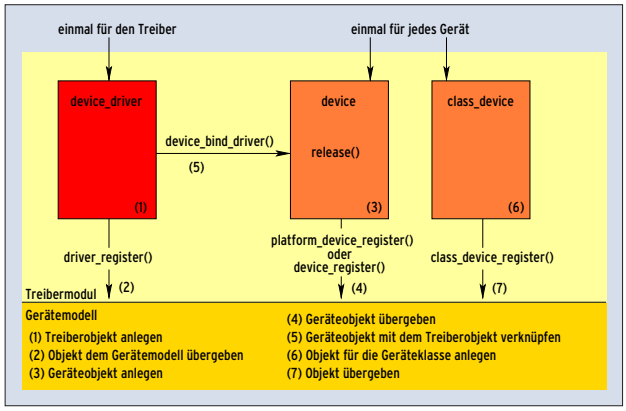


Abbildung 3: Ein Treibermodul muss Objekte für den Treiber selbst und für jedes Gerät definieren und dem Gerätemodell übergeben. »device_bind_driver()« ordnet beide Objekte einander zu.

den Wert liest und ausgibt. Schreibfunktionen besitzen neben den beiden erwähnten noch einen dritten Parameter, der die Anzahl der zu schreibenden Bytes angibt. Übrigens transferieren die Lese- und die Schreibfunktion Daten nur innerhalb des Kernelspace. Für den Transfer zum Userspace ist das Gerätemodell selbst verantwortlich.

Sind die Zugriffsfunktionen auf die Attributdateien definiert, müssen die Dateien im Sysfs erstellt werden. Die beiden Makros »DRIVER_ATTR()« und »DEVICE_ATTR()« definieren und initialisieren die Objekte, die die Funktionen »driver_create_file()« und »device_create_file()« dann dem Gerätemodell übergeben.

Attribute im Sys-Filesystem

Die Namen der über die Makros »DRIVER_ATTR()« und »DEVICE_ATTR()« erzeugten Objekte ergeben sich aus dem ersten Parameter mit dem Vorsatz »driver_attr_« beziehungsweise »device_attr_«. So definiert die folgende Zeile das Objekt »driver_attr_version«:

```
static DRIVER_ATTR( version, S_IRUGO,
   ShowVersion, NULL );
```

Der zweite Parameter spezifiziert die Zugriffsrechte auf die Attributdatei. Die Headerdatei »linux/stat.h« deklariert die möglichen Werte (siehe Tabelle 1). Da dieses Beispiel nur die Lesefunktion »ShowVersion()« verwendet, setzt das Makro auch nur Leserechte (hier für den User, die Gruppe und für die Übrigen). Der letzte Parameter ist »NULL« und verweist damit nicht auf eine Schreibfunktion, die sonst hier erscheinen würde. Die Bedeutung der Parameter gilt für »DEVICE_ATTR()« entsprechend (siehe Kasten „Attribute programmieren" und Listing 1).

Die Funktionen »driver_remove_file()« und »device_remove_file()« entfernen die angelegten Dateien wieder, wenn der Treiber beziehungsweise das Gerät nicht mehr gebraucht wird.

Attributdateien für PCI

Bei PCI-Geräten enthalten die Datenstrukturen »struct pci_driver« und »struct pci_dev« jeweils ein »struct device_driver« und ein »struct device«. Kein Problem also, auch in diesem Fall Attributdateien zu erzeugen. Bei der Treiberinitialisierung legt »driver_create_file()« die Attributdateien des Treibers an, bei der Geräte-Initialisierung »device_create_file()« die des Geräts.

```
driver_create_file( &pcidrv.driver,
    &driver_attr_mytext );
device_create_file( &pcidev->dev,
    &dev_attr_mytext );
```

Das Vorgehen, um ein Gerät bei einer Geräteklasse anzumelden, ist vertraut:

Attribute programmieren

Das Makro »DEVICE_ATTR()« (Listing 2, Zeile 11) definiert mit »freq« die Datenstruktur »dev_attr_freq« (»dev_attr_« wird dem Variablennamen vorangestellt). Es erlaubt lesenden und schreibender Zugriff und legt die Zugriffsfunktionen fest. Während der Geräte-Initialisierung in »DeviceProbe()« legt »device_create_file()« (Zeile 16) die Attributdatei an.

Der Kernel ruft »WriteFreq()« auf, wenn eine Applikation in die Attributdatei »freq« schreibt. Die zu schreibenden Daten befinden sich bereits im Kernelspace. Liest eine Applikation die Attributdatei »freq«, ruft der Kernel »ReadFreq()« auf. Die angeforderten Daten können direkt in den Puffer »buf« geschrieben werden. Die Funktion gibt die Anzahl der gelesenen Bytes zurück.

Bei der Geräte-Deinitialisierung in »DeviceRemove()« entfernt »device_remove_file()« (Zeile 23) die Attributdatei wieder aus dem Sys-Filesystem.

Listing 2: Attribute programmieren

```
01 static ssize_t ReadFreq( struct device *dev, char *buf )
02 {
03     ...
04 }
05
06 static ssize_t WriteFreq( struct device *dev, const char
   *buf, size_t count )
07 {
08     ...
09 }
10
11 static DEVICE_ATTR( freq, S_IRUGO|S_IWUGO, ReadFreq,
   WriteFreq );
12
13 static int DeviceProbe( ... )
14 {
15     ...
16     device_create_file( &mydevice.dev, &dev_attr_freq );
17     ...
18 }
19
20 static int DeviceRemove( ... )
21 {
22     ...
23     device_remove_file(&mydevice.dev, &dev_attr_freq );
24     ...
25 }
```

Meldet sich ein Treiber bei einem Subsystem – zum Beispiel beim Netzwerksubsystem – an, sorgt dieses für die Registrierung beim Gerätemodell.

Geräteklassen

Gibt es dagegen kein zugehöriges Subsystem, muss der Treiberentwickler selbst Hand anlegen – etwa bei Geräten, die zu den Klassen »input« oder »pcmcia _socket« gehören. Dazu definiert er ein Objekt, initialisiert es und übergibt es dem Gerätemodell. Die Übergabe erfolgt während der Geräte-Initialisierung. Dabei geht der Kernelprogrammierer folgendermaßen vor:

- Er definiert ein Objekt von Typ »struct class_device«, zum Beispiel mit Namen »myclassdev«. Es dient als Container, um ein Geräte- und ein Treiberobjekt an das Gerätemodell zu übergeben.
- Um Geräte- und Treiberobjekt miteinander zu verknüpfen, trägt er die Adresse des Treiberobjekts »mydriver« (Typ »struct device_driver«) im Geräteobjekt »mydevice« ein (»struct device«).
- Jetzt trägt er die Adresse von »mydevice« in »myclassdev.dev« ein.
- Nun muss er noch festlegen, unter welcher Klasse (also in welchem Sysfs-Verzeichnis) der neue Eintrag steht. Für Input-Klassen schreibt er die Adresse der globalen Variablen »input_class« (definiert in »linux/input.h«) nach »myclassdev.class«.
- Er kopiert den Verzeichnisnamen, unter dem die Einträge später auftauchen, in »myclassdev.class_id«.
- Er übergibt mit »class_device_register()« das Objekt an den Kernel:

```
static struct class_device myclassdev;
...
    mydevice->driver = &mydriver;
    myclassdev.dev = &mydevice;
    myclassdev.class = &input_class;
    strlcpy( (void *)&myclassdev.class_id,
     "MyDev", 6 );
    class_device_register( &myclassdev );
```

Das Gerätemodell legt daraufhin das Verzeichnis »MyDev« unterhalb von »/sys/class/input« an und setzt symbolische Links auf die entsprechenden Geräte- und Treiberobjekte:

```
device -> ../../../devices/pci0000:00/
0000:00:08.0
driver -> ../../../bus/pci/drivers/pci_drv
```

Auch ein Class-Device kann Attribute besitzen. Die Programmierung unterscheidet sich allein im Namensvorsatz für Makros und Funktionen von dem bereits dargestellten Ansatz (»CLASS« statt »DRIVER« oder »DEVICE«).

Interessanter ist es allerdings, eine eigene Klasse zu definieren. Hierfür sind eine Hotplug- sowie eine Release-Funktion zu implementieren und beide über ein Klassen-Objekt dem Linux-Gerätemodell zu übergeben. Die Hotplug-Funktion wird aufgerufen, sobald sich ein Treiber bei der Klasse anmeldet, die Release-Funktion, wenn sich der Treiber wieder abmeldet.

Tabelle 1: Zugriffsrechte

Symbol	Bedeutung
S_IRWXU	Lesen, Schreiben und Ausführen für den User
S_IRUSR	Lesen für den User
S_IWUSR	Schreiben für den User
S_IXUSR	Ausführen für den User
S_IRWXG	Lesen, Schreiben und Ausführen für die Gruppe
S_IRGRP	Lesen für die Gruppe
S_IWGRP	Schreiben für die Gruppe
S_IXGRP	Ausführen für die Gruppe
S_IRWXO	Lesen, Schreiben und Ausführen für Übrige
S_IROTH	Lesen für Übrige
S_IWOTH	Schreiben für Übrige
S_IXOTH	Ausführen für Übrige
S_IRWXUGO	Kombination von »S_IRWXU|S_IRWXG|S_IRWXO«
S_IALLUGO	Kombination von »S_ISUID|S_ISGID|S_ISVTX|S_IRWXUGO«
S_IRUGO	Kombination von »S_IRUSR|S_IRGRP|S_IROTH«
S_IWUGO	Kombination von »S_IWUSR|S_IWGRP|S_IWOTH«
S_IXUGO	Kombination von »S_IXUSR|S_IXGRP|S_IXOTH«

Auf dem FTP-Server [5] ist als Beispiel ein kompletter Treiber zu finden, der die eigene Geräteklasse »GameClass« implementiert. Innerhalb der Klasse erzeugt er ein Klassengerät »GameDevice« und ein virtuelles Gerät »GameClassDevice«.

Peripheriebusse definieren

Eigene Busse definieren ist noch unkomplizierter: Der Programmierer legt ein Busobjekt an, initialisiert es mit dem Namen und registriert es mit der Funktion »bus_register()«:

```
struct bus_type can = {
    .name = "CAN",
};
static int __init MyModulInit(void)
{   ...
    bus_register( &can );
```

Das Konzept des Gerätemodells ist klar strukturiert, aber da der zugehörige Kernelcode noch nicht ganz ausgereift ist, führen kleine eigene Programmierfehler schnell zum Absturz des Systems – dann hilft nur noch ein Reboot.

Fazit und Vorschau

Sich ins neue Gerätemodell einarbeiten lohnt für jeden Kernelprogrammierer. Das Modell ist zukunftsfähig: Es erlaubt ordentliches Powermanagement, zudem wird das Sys-Filesystem viele Aufgaben des überforderten Proc-Filesystems übernehmen. So wird die nächste Kern-Technik-Folge zeigen, wie das Proc-Filesystem in seinem angestammten Bereich funktioniert und wie man es in eigenen Kernelmodulen einsetzt. *(ofr)*

Infos:

[1] Greg Kroah-Hartmann, „udev – A Userspace Implementation of devfs“: [http://archive.linuxsymposium.org/ols2003/Proceedings/All-Reprints/Reprint-Kroah-Hartman-OLS2003.pdf]
[2] Manual Estrada Sainz, „ Firmware Class“, Dokumentation in den Kernelquellen: »Documentation/firmware_class«.
[3] Eva-Katharina Kunst und Jürgen Quade: „Kern-Technik“, Folge 4, Linux-Magazin 10/03, S. 81
[4] Libsysfs: [http://linux-diag.sf.net].
[5] Listings: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/01/Kern-Technik/]