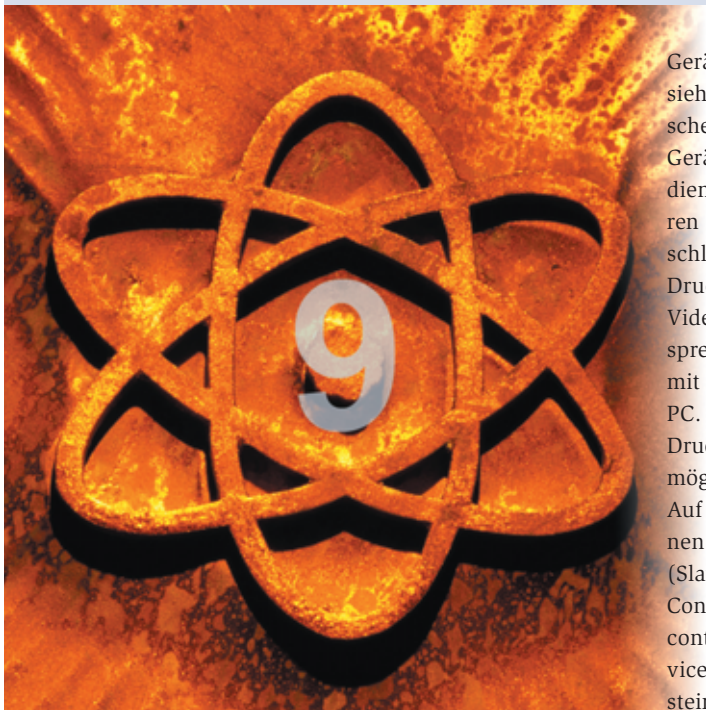
Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 9

Kern-Technik

USB hat die klassische serielle Schnittstelle als Allround-Interface abgelöst und macht mit Version 2.0 den Highspeed-Bussen Konkurrenz. Kernel 2.6 unterstützt schon ein breites Spektrum an Hardware und hilft unbekannte USB-Geräte direkt zu programmieren. Eva-Katharina Kunst, Jürgen Quade



Version 2.0 des Universal Serial Bus (USB) bringt attraktive Bandbreiten: Erreichte schon USB 1.1 Datenraten bis zu 12 MBit/s, sind jetzt 480 MBit/s das Maß aller Dinge [1]. Neben schneller Übertragung steht USB auch für die große Vielfalt moderner Peripheriegeräte. Grund genug für einen Blick auf die Treiberprogrammierung für USB-Devices, siehe [2]. Für den Zugriff auf USB-Geräte ist nicht unbedingt ein Kernelmodul nötig, denn mit der Libusb steht eine Bibliothek zur Verfügung, die den Zugriff aus dem Userspace ermöglicht [3]. Dieser Artikel zeigt, wie es im Kernelspace geht, sollte es einmal nötig sein.

Der USB verbindet Peripheriegeräte mit dem Rechner. Physisch ist USB als Baum strukturiert, mit dem USB-Controller als Wurzel (Master), den Hubs als Ästen (Abzweigungen) und den einzelnen Geräten als Blättern (Slaves), siehe Abbildung 1. Inzwischen gibt es kaum noch eine Geräteart, die USB nicht bedient – Mäuse oder Tastaturen eignen sich zum USB-Anschluss ebenso wie Scanner, Drucker, Digital- oder auch Videokameras. Die Endgeräte sprechen jedoch immer nur mit einem Master, meist dem PC. Den Scanner direkt am Drucker anschließen ist nicht möglich.

Auf Hardware-Ebene bedienen Master und Endgeräte (Slaves) den Bus über einen Controllerbaustein: den Hostcontroller respektive den Devicecontroller. Dieser Baustein wird durch den USB-Controller-Treiber angesteuert, der wiederum die IO-Ports der Hardware anspricht, siehe [4].

Meister und Gehilfe

Ähnlich wie bei Netzwerkkarten ist auch für Controllerbausteine unterschiedlichen Typs ein eigener Treiber erforderlich, siehe Abbildung 2. Der Controller-Treiber wird im Normalfall durch das USB-Subsystem angesteuert, das das Gerätemanagement übernimmt: So ist es dafür zuständig, Geräte zu finden, zu identifizieren und sie einem Treiber zuzuordnen. Darüber hinaus realisiert das USB-Subsystem eine Controller-unabhängige Schnittstelle für den gerätespezifischen Treiber.

Sowohl für den Host als auch für die Peripherie bringt Linux ein solches USB-Subsystem mit. Für die meisten erhältlichen Controller gibt es bereits Linux-Treiber. Auf dieser Ebene ist also glücklicherweise nicht mehr viel zu tun.

Anders liegt der Fall bei einem gerätespezifischen Treiber. Er sorgt beispielsweise bei einem USB-Scanner dafür, die richtige Gammakurve zu laden oder die Scannerlampe einzuschalten. Ein solcher Treiber, der im Gerät selbst werkelt, hängt stark von der verwendeten Hardware ab. Man nennt ihn Gadget-Treiber. Er ist vor allem für Entwickler eingebetteter Systeme von Interesse, eine Einführung hierzu findet sich in [5]. Der Artikel beschränkt sich den gerätespezifischen Host-Treiber des Masters.

Ab geht die Post

Der Host-Treiber spricht mit seinem USB-Gerät über eine Paketschnittstelle. Ein Paket heißt URB (USB Request Block) und enthält den Adressaten, Transferinformationen und die Daten selbst. Die Transferinformation gibt die Art der Übertragung an, bei USB eine von vier Möglichkeiten:

- Kontroll-Transfer (Control): Kurze Pakete mit Konfigurationsdaten für das Gerät beziehungsweise Statusinformationen vom Gerät.
- Massen-Transfer (Bulk): Pakete zum normalen Datenaustausch zwischen Controller und Gerät.
- Interrupt-Transfer (Int): Periodischer Datentransfer zwischen Controller und Gerät.
- Isochroner Transfer (Iso): Übertragung in Realzeit und mit fester Bandbreite.

Eine Gerätenummer, eine Interfacenummer und ein Endpunkt definieren den Adressaten einer solchen Nachricht.

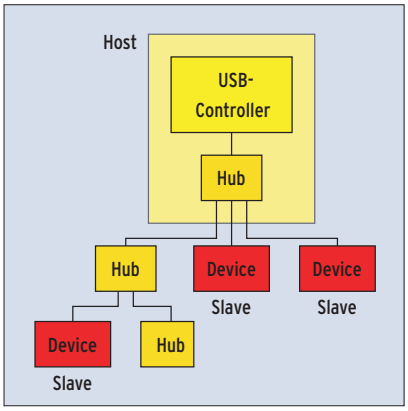


Abbildung 1: USB in Baumform: Geräte hängen über einen oder mehrere Hubs am Host.

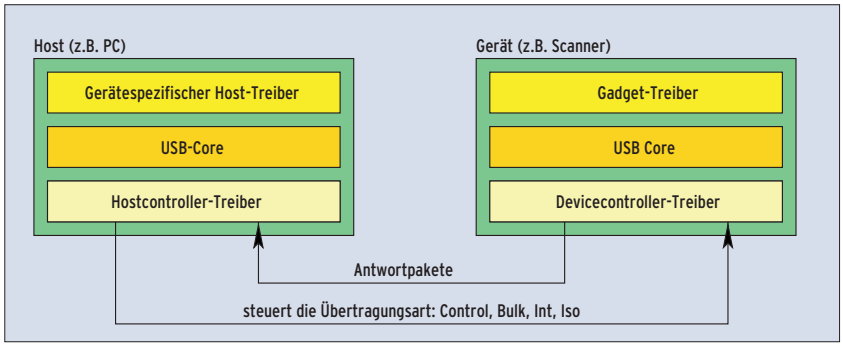


Abbildung 2: Im Host und im Endgerät finden sich jeweils drei Schichten für die USB-Kommunikation.

Der grundlegende Aufbau des USB-Systems unter Linux ähnelt dem von PCI: Da USB-Geräte ebenfalls hotpluggable sind, ist die Treiberinitialisierung von der Geräteinitialisierung getrennt. Laden Anwender oder der Modul-Autoloader den Treiber, ruft der Kernel die Funktion »ModInit()« auf. Sie registriert den Treiber mit »usb_register()« beim USB-Subsystem und zeigt diesem an, für welche Geräte er zuständig ist, siehe Listing 1, Zeilen 86 bis 90.

Dazu dient die Struktur »usb_dev_id«, deren Aufbau und Verwendung der Kasten „USB-Gerätemerkmale“ näher beschreibt. Die Funktion »usb_register()« übernimmt neben den Gerätemerkmalen in der »struct usb_driver« noch weitere Informationen: den Treibernamen, die Adresse der Funktion zur Geräteinitialisierung »probe()« und die Adresse der Deinitialisierungsfunktion »disconnect()«, siehe Zeile 77 bis 82.

Sobald das USB-Subsystem ein Gerät ortet, das der vom Treiber übergebenen Gerätekennung entspricht, initialisiert es das Device. Die »probe()«-Funktion überprüft, ob das Gerät vom Treiber bedient wird. Falls ja, muss der Kernel die Userseite darüber informieren, dass ein neues Gerät vorhanden ist.

Wie die Applikation auf die neue Peripherie zugreift, hängt von der Art des Geräts ab. Handelt es sich beispielsweise um einen USB-Netzwerkadapter, meldet der Treiber ihn beim Netzwerk-Subsystem an. Gehört das Gerät jedoch keiner derartigen Klasse an, kann es als Character-Device angesprochen werden.

Kernel, übernehmen Sie!

Weil Major-Nummern knapp sind, meldet das USB-Subsystem im Kernel 2.6 sich stellvertretend für alle USB-Geräte als zeichenorientierter Treiber mit der Major-Nummer 180 an. Im Kernel 2.4 hatte das USB-Subsystem noch jedem Treiber eine eigene Minor-Nummer zugeteilt. Beim neuen Kernel muss ein Treiber diese Minor-Nummer explizit anfordern. Dazu spezifiziert der Treiber das Gerät während der Initialisierung in der Datenstruktur »struct usb_class_driver«, siehe Listing 1, Zeilen 48 bis 53.

Zu den Feldern der Struktur zählen der Treibername und die Zugriffs- und Geräteart (zum Beispiel zeichenorientiertes Gerät). Zusätzlich muss das Modul die Tabelle mit Funktionen »struct file_operations« übergeben, die der Kernel aufruft, wenn eine Applikation auf die zugehörige Gerätedatei zugreift. Achtung: Von diesen Funktionen ist zumindest die Funktion »DriverOpen()« zu implementieren, selbst wenn sie nur aus der Zeile »return 0« besteht.

Ruft der Treiber »usb_register_dev()« auf, teilt ihm das USB-System dynamisch eine Minor-Nummer zu, die sich dann im Feld »minor« befindet. Für den Zugriff auf das Device muss eine Gerätedatei mit dieser Minor-Nummer existieren. Ist das Device-Filesystem aktiviert, legt es diese Gerätedatei automatisch im Verzeichnis »/dev/« an, ansonsten ist hier Handarbeit gefordert (Aufruf von »mknod«, siehe Kasten „Ärmel hochkrempeln“).

Blockweise Requests

Nun können Anwendungen auf das USB-Gerät zugreifen. Der Treiber schickt dann USB Request Blocks (URBs) an das Gerät und empfängt entsprechende Antworten. In einem URB, definiert in »include/linux/usb.h«, trägt die Anwendung alle Parameter ein, die der USB-Core dazu benötigt, den Datentransfer zwischen Controller und Peripheriegeräten abzuwickeln.

Dazu gehören die Endadresse (Gerät, Interface, Endpunkt), die eigentlichen Daten, die Länge der zu übertragenden Daten, ein Timeout für die Übertragung sowie die Adresse einer Callback-Funktion. Die ist nötig, weil USB asynchron arbeitet. Eine Instanz – etwa die Lese-

USB-Gerätemerkmale

Ein USB-Gerät besitzt bis zu neun Merkmale: Produkt-ID, Versionsnummernbereich (Low und High), Geräteklasse, Geräteunterklasse, Geräteprotokoll, Interfaceklasse, Interfaceunterklasse und Interfaceprotokoll. Ein Treiber muss nicht alle Merkmale belegen. Vielmehr ist es möglich, ein Gerät nur aufgrund einiger Merkmale auszuwählen.

Anders als bei PCI gibt es allerdings für die einzelnen Attribute keine Wildcards. Stattdessen gibt ein Bitfeld in der Struktur »struct usb_device_id« an, ob ein bestimmtes Merkmal zur Identifikation herangezogen werden soll oder nicht (so genannte Match-Flags). Ein gesetztes Bit sorgt dafür, dass das USB-Subsystem das Merkmal auswertet.

Unterstützung durch Makros

Der Programmierer muss die notwendigen Bit-Operationen nicht von Hand durchführen, die Headerdatei »linux/usb.h« enthält dafür Makros. »USB_DEVICE()« bekommt beispielsweise die Parameter Herstellerkennung und Produkt-ID übergeben und erstellt aus diesen Angaben eine gültige Version der »struct usb_device_id«. Ähnlich übernimmt »USB_DEVICE_VER()« zusätzlich zu Herstellerkennung und Produkt-ID den Versionsnummernbereich.

funktion in einem Treiber – übergibt dem USB-Subsystem einen URB. Während der dort spezifizierte Auftrag abgearbeitet wird, kann die Treiberfunktion weiterlaufen, blockiert also nicht. Sobald das Subsystem mit dem Auftrag fertig ist, ruft es die Callback-Funktion auf.

USB ruft zurück

Ein URB wird durch die Funktion »usb _alloc_urb()« erzeugt. Ist die URB-Datenstruktur angelegt, muss das Modul als Nächstes deren Felder ausfüllen (Initialisierung des URB). Die Headerdatei »include/linux/usb.h« definiert für die Transferarten Control, Bulk und Int entsprechende Makros: »usb_fill_control _urb()«, »usb_fill_bulk_urb()« sowie »usb_fill_int_usb()«. Ist der URB fertig gestellt, schickt der Treiber ihn ab. Die Funktion »usb_submit_urb()« startet den Transfervorgang und »usb_unlink_ urb()« bricht ihn notfalls wieder ab. Die Callback-Funktion (Element »complete«) übernimmt gegebenenfalls das Ergebnis und gibt den URB mit Hilfe der Funktion »usb_free_urb()« wieder frei.

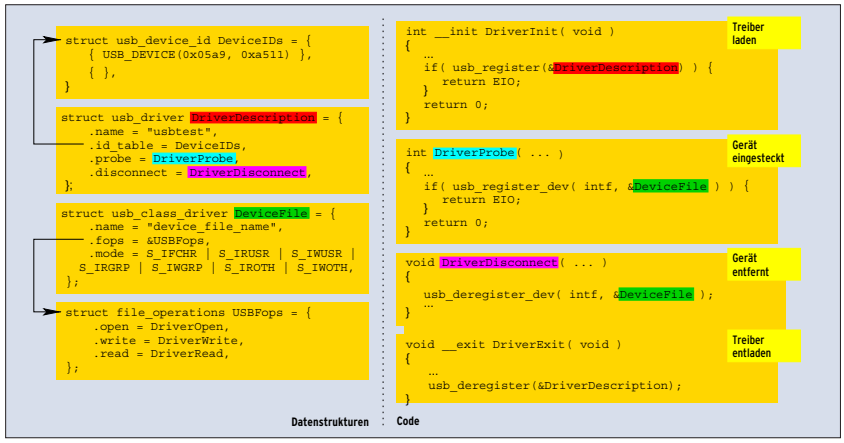


Abbildung 3: Funktionen eines USB-Host-Treibers mit den von ihm benutzten Datenstrukturen.

Listing 1: Treibergerüst für USB

```
01 #include <linux/fs.h>
02 #include <linux/version.h>
03 #include <linux/module.h>
04 #include <linux/init.h>
05 #include <linux/usb.h>
06 #include <asm/uaccess.h>
07
08 MODULE_LICENSE( "GPL" );
09
10 #define USB_VENDOR_ID 0x05a9
11 #define USB_DEVICE_ID 0xa511
12
13 struct usb_device *dev;
14
15 static ssize_t usbtest_read( struct file
   *Instanz, char *buffer, size_t count,
16                 loff_t *ofs )
17 {
18    __u16 status;
19    char pbuf[20];
20
21    if( usb_control_msg(dev, usb_rcvctrlpipe(dev,
   0), USB_REQ_GET_STATUS,
22                            USB_DIR_IN |
   USB_TYPE_STANDARD | USB_RECIP_INTERFACE,
23                            0, 0, &status,
   sizeof(status), 5*HZ) < 0 )
24           return -EIO;
25    snprintf( pbuf, sizeof(pbuf), "status=%d\n",
   status );
26    if( strlen(pbuf) < count )
27           count = strlen(pbuf);
28    count -= copy_to_user(buffer,pbuf,count);
29    return count;
30 }
31
32 static int usbtest_open( struct inode
   *devicefile, struct file *Instanz )
33 {
34    return 0;
35 }
36
37 static struct file_operations USBFops = {
38    .owner = THIS_MODULE,
39    .open  = usbtest_open,
40    .read  = usbtest_read,
41 };
42
43 static struct usb_device_id usbid [] = {
44    { USB_DEVICE(USB_VENDOR_ID, USB_DEVICE_ID), },
45    { }                  /* Terminating entry */
46 };
47
48 struct usb_class_driver ClassDescr = {
49    .name = "usbtest",
50    .fops = &USBFops,
51    .mode = S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP
   | S_IWGRP | S_IROTH | S_IWOTH,
52    .minor_base = 16,
53 };
54
55 static int usbtest_probe(struct usb_interface
   *interface,
56         const struct usb_device_id *id)
57 {
58    dev = interface_to_usbdev(interface);
59    printk("USBTEST: 0x%4.4x|0x%4.4x, if=%p\n",
   dev->descriptor.idVendor,
60           dev->descriptor.idProduct, interface
   );
61    if(dev->descriptor.idVendor==USB_VENDOR_ID
62           && dev->descriptor.idProduct ==
   USB_DEVICE_ID) {
63           if( usb_register_dev( interface,
   &ClassDescr ) ) {
64                  return -EIO;
65           }
66           printk("got minor= %d\n", interface-
   >minor );
67           return 0;
68    }
69    return -ENODEV;
70 }
71
72 static void usbtest_disconnect( struct
   usb_interface *iface )
73 {
74    usb_deregister_dev( iface, &ClassDescr );
75 }
76
77 static struct usb_driver usbtest = {
78    .name= "usbtest",
79    .id_table= usbid,
80    .probe= usbtest_probe,
81    .disconnect= usbtest_disconnect,
82 };
83
84 static int __init usbtest_init(void)
85 {
86    if( usb_register(&usbtest) ) {
87           printk("usbtest: unable to register
   usb driver\n");
88           return -EIO;
89    }
90    return 0;
91 }
92
93 static void __exit usbtest_exit(void)
94 {
95    usb_deregister(&usbtest);
96 }
97
98 module_init(usbtest_init);
99 module_exit(usbtest_exit);
```

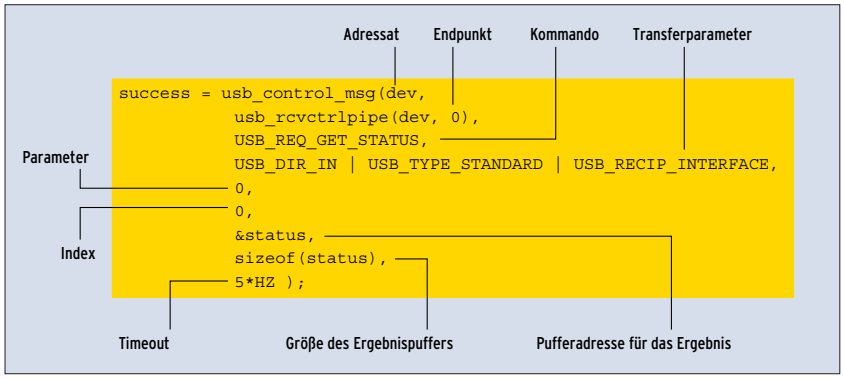


Abbildung 4: High-Level-Funktionen wie »usb_control_msg()« helfen dabei, URBs zu definieren und an Geräte zu versenden. Sie enthalten den Adressaten, die Transferparameter und einen Puffer für den Rückgabewert.

Für die Transferarten Control und Bulk gibt es Hilfsfunktionen, die das Erstellen und Verschicken eines URB übernehmen: »usb_control_msg()« oder für Bulk »usb_bulk_msg()«. Die Funktionen schicken die Daten an einen definierten Endpunkt und warten, bis der Transfer entweder beendet ist oder ein Timeout erfolgt. Sie geben anschließend die Anzahl der übertragenen Bytes zurück, im Fehlerfall einen negativen Wert.

Abbildung 4 zeigt, wie eine Statusanfrage verschickt wird. Von den drei Adressangaben (Geräte, Interface und Endpunkt) erscheinen hier nur zwei, denn das Interface ist bereits per Gerät »dev« ausgewählt. Der Endpunkt ist eine Unix-Pipe, sie wird durch ein Bitfeld repräsentiert, das die Geräteadresse, die Endpunktnummer, die Transferrichtung und die Transferart angibt. Für die Pipe-Funktionalität stehen Makros zur Verfügung, deren Name sich zusammensetzt aus: dem Präfix »usb_«, der Transferrichtung »snd« oder »rcv«, der Transferart »ctrl«, »bulk«, »iso« oder »int« sowie dem Abschluss »pipe«. Bei dem in der Abbildung verwendeten »usb_rcvctrlpipe« handelt es sich also um den Empfang (»rcv«) von Statusinformationen (»ctrl«) des Geräts.

Ein weiterer Parameter gibt den Befehl an, den das Gerät ausführen soll. Das können gerätespezifische Kommandos wie »USB_TYPE_VENDOR« oder Standardkommandos sein. Dafür gibt es folgende USB-Sendefunktionen:

- usb_get_descriptor()
- usb_get_device_descriptor()
- usb_get_status()
- usb_get_string()
- usb_string()
- usb_set_configuration()
- usb_get_interface()

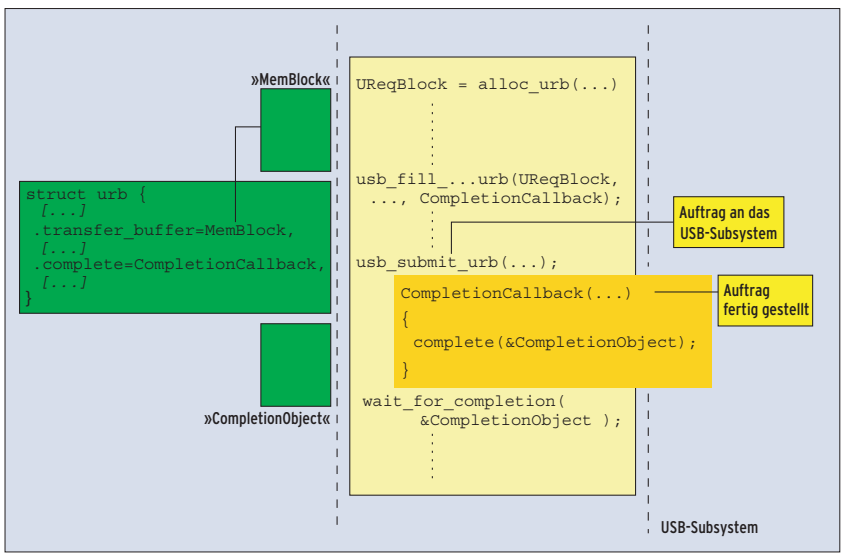


Abbildung 5: Bei asynchroner USB-Kommunikation legt der Treiber den URB an und übergibt eine Callback-Funktion, die das USB-Subsystem aufruft, wenn es den Request abgearbeitet hat.

Der nächste Parameter besteht aus der Transferrichtung aus Sicht des Hosts, der Art des Kommandos und einer Adressatenauswahl. Die Transferrichtung ist »USB_DIR_IN« oder »USB_DIR_OUT«. Als Adressaten stehen zur Auswahl: »USB_RECIP_MASK«, »USB_RECIP_DEVICE«, »USB_RECIP_INTERFACE«, »USB_RECIP_ENDPOINT« oder »USB_RECIP_OTHER«. Dann folgen noch ein optionaler Parameter sowie ein Index. Bei einem Kommando an das Gerät ist der Index »0«. Ein Befehl an das Interface enthält die Interfacenummer, ein Kommando an den Endpunkt und die Endpunktnummer.

Der nächste Parameter spezifiziert den Speicherbereich, in dem der Gerätetreiber das Ergebnis ablegt. Die beiden letzten Parameter bezeichnen die Speicherlänge und den Timeout des Requests.

Massenversand

Beim Bulk-Transfer heißt die Funktion entsprechend »usb_bulk_msg()«. Anders als »usb_control_msg()« besitzt sie nur die folgenden sechs Parameter: das Gerät, die Pipe, den Speicherblock samt Daten, die Länge des Speicherblocks, die Adresse einer Variablen, die das Ergebnis aufnimmt, sowie einen Timeout:

```
ult = usb_bulk_msg(dev, usb_rcvbulkpipe
  (dev, 0), buffer, count, &count, HZ*2 );
```

Wie »usb_control_msg()« versendet die »usb_bulk_msg()« das Paket und wartet auf eine Antwort. Achtung: Da die beiden Funktionen synchron arbeiten, dürfen sie nicht im Interrupt-Kontext eingesetzt werden. Bei asynchroner Kommunikation (Interrupt- und Isochron-Transfer) wartet das Modul nicht auf das Er-

Listing 2: Makefile zum Treibergerüst

```
01 TARGET=usbtest
02
03 ifneq ($(KERNELRELEASE),)
04 obj-m   := ${TARGET}.o
05
06 else
07 KDIR    := /lib/modules/$(shell uname -r)/build
08 PWD     := $(shell pwd)
09
10 default:
11         $(MAKE) -C $(KDIR)      SUBDIRS=$(PWD) modules
12 endif
```

Ärmel hochkrempeln

Besitzen Sie ein USB-Gerät, können Sie die beschriebenen Grundmechanismen eines USB-Treibers testen. Den Code des Treibers finden Sie in Listing 1. Im Quelltext tragen Sie die Hersteller-ID und die Produkt-ID (Zeilen 10 und 11) Ihres spezifischen Geräts ein. Außerdem müssen Sie noch dafür sorgen, dass der für das Gerät ursprünglich vorgesehene Treiber nicht geladen wird. Am besten gehen Sie folgendermaßen vor:

1. Stellen Sie sicher, dass Ihr Kernel USB unterstützt: Dazu muss USB-Support im Kernel konfiguriert sein, siehe auch [7]. Falls der USB-Support über Module erfolgt, stellen Sie durch Aufruf von »lsmod« fest, ob die Hostcontroller-Treiber mit den Namen »ehci_hcd«, »uhci_hcd« oder »ohci_hcd« geladen sind. Falls nicht, laden Sie den passenden mit »modprobe uhci_hcd«, »modprobe ehci_hcd« oder »modprobe ohci_hcd«.

2. Identifizieren Sie die Hersteller- und Geräte-ID Ihrer Peripherie. Dafür ist das Programm »lsusb« hilfreich. Es bezieht seine Informationen aus dem USB-Filesystem, das Sie per »mount -t usbfs none /proc/bus/usb« in den Verzeichnisbaum einhängen (falls nicht ohnehin bereits geschehen).
Stecken Sie das Gerät an den USB-Bus und rufen »lsusb« auf. In der Ausgabe sollten Sie Ihre Peripherie wiederfinden:

```
Bus 004 Device 001: ID 0000:0000
Bus 003 Device 032: ID 05a9:a511
   OmniVision Technologies, Inc. [...]
Bus 003 Device 001: ID 0000:0000
Bus 002 Device 001: ID 0000:0000
Bus 001 Device 001: ID 0000:0000
```

Das Beispiel verwendet eine Webcam zum Test der Mechanismen innerhalb des USB-Subsystems. Die Ausgabe zeigt, dass die Webcam die Hersteller-ID »0x05a9« und die Produkt-ID »0xa511« hat. In Listing 1 sind diese IDs bereits eingetragen.

3. Damit das Hotplug-System den ursprünglich vorgesehenen Treiber nicht automatisch lädt, tragen Sie ihn in »/etc/hotplug/blacklist« ein. Sie finden den Treiber über das File »/lib/modules/*Kernelversion*/modules.usbmap«.

4. Wenn Sie den Quellcode modifiziert und den Treiber übersetzt haben, laden Sie diesen mit »insmod usbtest.ko«. Falls Sie das Device-Filesystem aktiviert haben, wird vom USB-Subsystem automatisch eine passende Gerätedatei angelegt.
Falls nicht, legen Sie die Gerätedatei selbst an. Bei USB ist die Major-Nummer immer »180«. Die Minor-Nummer hat der Treiber in den Syslogs ausgegeben, innerhalb des Treibers wird die Minor-Nummer »16« angefordert.

```
mknod devicefile c 180 16
```

5. Der Treiber ist so implementiert, dass – wenn Sie von der Gerätedatei mit »cat <devicefile« lesen – er den Status zurückgibt:

```
status = 0
...
```

Falls Sie das Gerät jetzt abstecken, bricht »cat« mit einem Ein-/Ausgabefehler ab.

gebnis. Hier muss der Programmierer selbst URBs anlegen, initialisieren und per »usb_submit_urb()« verschicken (Abbildung 5). Das Modul legt den URB an und initialisiert die Transportart. Dabei übergibt es eine Callback-Funktion, die vom USB-Subsystem aufgerufen wird, sobald es den Auftrag abgearbeitet hat. Zur Synchronisation der Aufträge dient ein Completion-Objekt [6].

Und tschüs!

Wird das Gerät abgeklemmt, informiert das USB-Subsystem den Treiber über die »disconnect()«-Funktion, die der Treiber bereits bei der Anmeldung übergeben hat (im Element ».disconnect« der Datenstruktur »usb_driver«, siehe Abbildung 3). Die Funktion beendet alle aktiven Zugriffe auf das Gerät und vermerkt, dass es entfernt worden ist.
Das Subsystem ruft die Funktion »disconnect()« allerdings auch dann auf, wenn das Treibermodul entladen wird. Im einfachsten Fall meldet sich der Treiber wieder bei jenem USB-Subsystem ab, bei dem er sich bei der Geräteinitialisierung zuvor registriert hatte. Dann reicht zum Abmelden die Funktion »usb_deregister_dev()« aufzurufen, siehe Listing 1, Zeilen 72 bis 74. Für den nun fälligen Test des Treibers beschreibt der Mini-Workshop im Kasten „Ärmel hochkrempeln“ alles Notwendige.

Portierung auf Kernel 2.6

Wie in Kernel 2.4 meldet sich der USB-Treiber mit »usb_register()« beim USB-Subsystem an. Allerdings hat sich die als Parameter übergebene »struct usb_driver()« geändert: Der Treiber nutzt nicht mehr automatisch die Major-Nummer des USB-Subsystems. Soll der Treiber auf die Major-Nummer zurückgreifen, muss er unter Kernel 2.6 die »usb_class_driver« ausfüllen, darunter einen Zeiger auf die »struct file_operations«. Für Treiber, die die Major-Nummer nicht verwenden, entfällt dieser Schritt.
Die Funktion »usb_register_dev()« übergibt die »struct usb_class_driver« dem USB-System. Ist das Device-Filesystem aktiviert, erstellt Kernel 2.6 automatisch eine Gerätedatei. Mit »usb_deregister_dev()« gibt der Treiber die ihm zugewiesene Minor-Nummer wieder frei.
Die Funktion »usb_submit_urb()« ist in Kernel 2.6 um einen Parameter reicher: Nun steuern Flags die Speicherreservierung. So lässt sich mit »GFP_KERNEL« das von der älteren Version bekannte Verhalten wieder herstellen. Auch die »probe()«-Funktion erscheint in neuem Gewand. Der erste Parameter ist nun vom Typ »usb_interface«:

```
void *USBDriverProbe( struct usb_device
   *dev, unsigned intf, const struct
   usb_device_id *id );
```

Vorschau

Die nächste Folge der Kern-Technik erklärt, wie die Parameterübergabe an dynamische Kernelmodule unter Linux 2.6 funktioniert und wie Module untereinander Daten und Funktionen austauschen können. *(ofr)* ■

Infos:

[1] USB-Spezifikationen: [http://www.usb.org/developers/docs]
[2] David Brownell, „Linux and USB 2.0“: [http://www.linux-usb.org/usb2.html]
[3] Johannes Erdfelt, „Libusb Developers Guide“: [http://http://libusb.sourceforge.net/]
[4] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 3: Linux-Magazin 10/03, S. 81
[5] David Brownell, „USB Gadget API for Linux“: [http://libusb.sourceforge.net/doc/]
[6] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 4: Linux-Magazin 11/03, S. 96
[7] Eva-Katharina Kunst, Jürgen Quade, „Meister-Installateur“: Linux-Magazin 2/04, S. 28