

# USB-Gadget

## Framework für die Entwicklung von Gerätetreibern unter Linux

Die USB-Schnittstellen auf der Host-Seite sind sehr genau standardisiert – nicht nur der USB-Anschluss selbst, sondern auch die Zugriffsmechanismen auf die USB-Chips im PC. Das ist auf der Geräteseite leider nicht der Fall. Hier gibt es eine große Vielfalt von USB-Chips mit jeweils eigenen Schnittstellen. Das „Gadget“-Framework von Linux verleiht den Gerätetreibern immerhin eine gewisse Modularität, so dass eine Entwicklung oder Anpassung von USB-Treibern einfacher wird.

Von Thomas Brinker

USB hat sich als vielseitiger Standard für Kurzstreckenverbindungen in den letzten Jahren etabliert. Zunächst war es als Ersatz für PS/2, serielle und parallele Schnittstellen konzipiert worden. Inzwischen sind schnelle Verbindungen zu Festplatten und Streaming Devices wie Videokameras als typisches Einsatzfeld von USB hinzugekommen. Im Umfeld von USB wurde nicht nur ein serielles Protokoll standardisiert, sondern auch mehrere Geräteklassen und sogar die detaillierte Ansteuerung der USB-Host-Controller.

Linux verfügt über leistungsstarke und zuverlässige Treiber für alle Host-Controller-Standards und über Treiber für fast alle Geräteklassen (Tabelle 1). Das USB-Framework ermöglicht zudem die schnelle und leichte Entwicklung neuer Treiber. Für alle USB-Geräteklassen sind bereits hostseitig Trei-

ber vorhanden. Auch für die allermeisten USB-Seriell-Adapter, für viele USB-Kameras und andere Geräte gibt es bereits USB-Treiber und es werden ständig mehr. Nur in den allerseltensten Fällen ist es notwendig, Linux um einen Treiber für die Hostseite zu ergänzen. Die ausführliche Dokumentation und viele Beispiele erleichtern hierbei die Arbeit.

An jedem USB-Host-Controller können bis zu 127 Geräte angeschlossen werden. Jedes dieser Geräte muss mindestens einen Endpunkt haben, den Endpunkt Null – bis zu 16 Endpunkte sind möglich. Selten werden mehr als vier Endpunkte realisiert. In der Hardware ist für jeden Endpunkt ein FIFO angelegt, aus dem heraus die Daten in das FIFO des gleichen Endpunktes des Kommunikationspartners übertragen werden. Üblicherweise wird nur der Endpunkt Null bidirektional verwendet. Für alle anderen wird die Kommunikationsrichtung in der Initialisierungsphase festgelegt. Die Kommunikation erfolgt immer zwischen Host und Client (Bild 1). Die Richtung wird immer aus Sicht des Host benannt. Gibt der Host Daten aus, so spricht man von einer OUT-Transaktion, liest er Daten ein, so ist es eine IN-Transaktion. Für das Gerät ergibt sich also eine scheinbare Umkehrung der Be-

griffe IN und OUT. Man spricht von einer IN-Transaktion, wenn das Gerät Daten ausgibt und von einer OUT-Transaktion, wenn das Gerät Daten einliest.

### Neues Gerät erkannt – was nun?

Sobald ein neues USB-Gerät vom Host erkannt wird, fragt dieser einige Informationen vom Gerät ab und vergibt eine Adresse, die für die weitere Kommunikation verwendet wird. Hierauf folgend werden dann weitere Informationen in Form von Deskriptoren vom

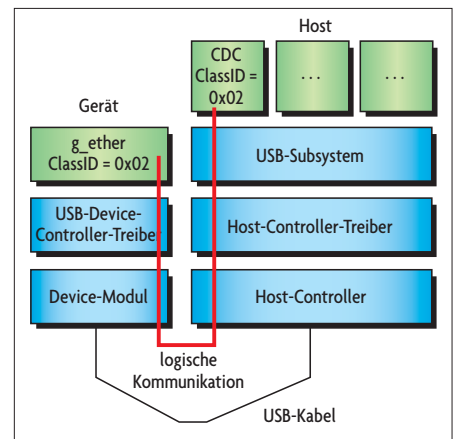


Bild 1. Datenpfad zwischen USB-Gerät und USB-Host. Die Kommunikationsrichtung wird immer aus Sicht des Host benannt. Gibt der Host Daten aus, so spricht man von einer OUT-Transaktion.

Gerät abgefragt, mit Hilfe derer ein passender Treiber auf dem Host gesucht werden kann. Diesen Vorgang, der für alle USB-Devices gleich ist, nennt man „Enumeration“.

Eine besondere Rolle kommt hierbei dem „device descriptor“ zu. Neben vielen anderen Spezifika über das Gerät enthält er auch drei IDs, die auf der Suche nach dem passenden Treiber helfen (Tabelle 2). Jeder Hersteller hat eine eigene „VendorID“ und kann beliebig „DeviceIDs“ vergeben. Gehört das Gerät zu einer bestimmten Klasse, so wird eine passende „ClassID“ ebenfalls im device descriptor festgelegt. Über die Klassen kann in der Enumeration ein bestimmter Klassentreiber

Klasse	Geräte-Beispiel	Linux-Host-Treiber	Linux-Gadget-Treiber
Audio	USB-Sound	ja	nein
CDC	Netzwerkverbindung	ja	ja
HID	Tastatur, Maus u.ä.	ja	nein
Mass-Storage	USB-Stick	ja	ja
RNDIS <sup>1)</sup>	Netzwerkverbindung	ja	ja
seriell <sup>2)</sup>	RS-232-to-USB-Konv.	ja	ja

<sup>1)</sup> Spezifikation zur Netzwerkverbindung von Microsoft, konkurriert mit CDC  
<sup>2)</sup> keine Klasse im engeren Sinne

Tabelle 1. Linux-Unterstützung für die wichtigsten USB-Geräteklassen

Feld	Bit	Beschreibung
ClassID	8	Geräteklasse (z.B. Mass-Storage = 0x08; CDC = 0x02; Hersteller spezifisch = 0xff)
VendorID	16	ID des Herstellers, von USB.org vergeben
DeviceID	16	ID des Produktes, vom Hersteller vergeben

! **Tabelle 2. Auszug aus der Liste der „USB device descriptors“**

geladen werden. Dieser ist üblicherweise fester Bestandteil des Betriebssystems, sodass eine Treiberinstallation nicht erforderlich ist (**Bild 2**). Richtiges Plug-and-Play ist also möglich. Kann ein Gerät keiner bestimmten Klasse zugeordnet werden, so steht im Feld für die ClassID der Wert „0xff“, und der Treiber wird abhängig von Vendor- und DeviceID gesucht. Bei der ersten Benutzung müssen dann meistens Treiber nachinstalliert werden.

**Framework für beliebige Embedded-Systeme als USB-Geräte**

Wie sieht es jedoch aus, wenn Embedded-Systeme als USB-Geräte auftreten sollen? – Die Antwort für Linux lautet: USB-Gadget. USB-Gadget ist ein Framework für Treiber von Hardware-USB-Geräte-Modulen. Das Framework ist in aktuellen Linux-Versionen 2.4 und 2.6 enthalten. Es lässt sich in drei Schichten unterteilen (**Bild 3**):

- ▶ „USB Device Controller (UDC)“-Treiber,
- ▶ Gadget-Treiber und
- ▶ andere Kernel-Infrastrukturen.

Der UDC-Treiber kommuniziert direkt mit dem USB-Controllerchip des Gerätes. Je nach Art des Controllers werden schon viele Aufgaben direkt in der Hardware erledigt. Üblicherweise stellt die Hardware eine bestimmte Zahl von Endpoints bereit, denen ein FIFO festgelegter Tiefe zugeordnet ist. Je nachdem, wieviel schon in der Hard-

Modul
NetChip 2280
PXA 25x or IXP 4xx
Toshiba TC86C001 „Goku-S“
LH7A40X
OMAP USB Device Controller

! **Tabelle 3. Vom Linux-Kernel unterstützte USB-Geräte-Module**

ware abgewickelt wird, steigt oder fällt der Aufwand für die Implementierung des Treibers, damit dieser den Anforderungen der USB-Gadget-API genügt. Mit diesem API reagiert das Gerät auf

Abfragen des USB-Host, sendet und empfängt Daten über USB-Endpoints und nutzt das Power-Management.

Leider gibt es keine einheitlichen Standards für USB-Device-Controller-Hardware, wie es sie für die Host-Controller gibt. Dies macht in vielen Fällen eine komplette Neuentwicklung des Treibers nötig. Für viele USB-Devices gibt es aber bereits Treiber, die in den Linux-Kernel eingeflossen sind, und es werden ständig mehr. **Tabelle 3** gibt eine Übersicht über die bereits unterstützten Geräte-Module.

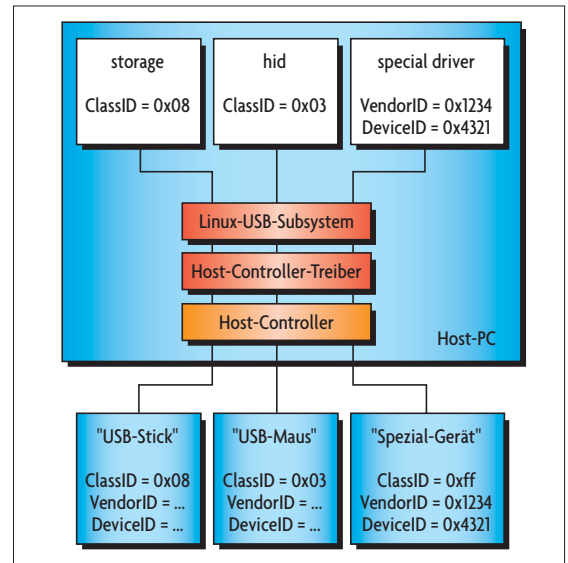
Der Entwicklungsaufwand eines UDC-Treibers sollte keinesfalls unterschätzt werden. Die im Kernel bereits vorhandenen Treiber kommen auf jeweils 2000 bis 3000 Zeilen. Etwa 20 Funktionen sind zu implementieren und ein tieferes Verständnis von USB ist unverzichtbar. Nach gründlicher Einarbeitung stellt dies jedoch keine allzu große Hürde für einen versierten Programmierer dar. Zum Abschluss der Arbeiten am Treiber ist die systematische Verifikation unabdingbar. Hierfür stehen recht viele und qualitativ hochwertige Tools zur Verfügung.

**Gadget-Treiber für nahezu jede Geräteklasse**

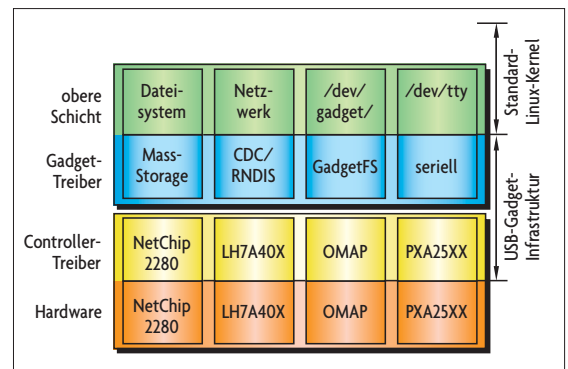
Der Gadget-Treiber wird logisch über dem UDC-Treiber angeordnet und greift auf dessen API zu. Sein Verhalten definiert die genaue Funktion, die über USB realisiert werden soll. Es gibt Gadget-Treiber für fast jede Geräteklasse, die für USB definiert wurde. Wird nun der Gadget-Treiber für USB-Massenspeicher geladen, dann können Speichergeräte wie z.B. ein Memorystick benutzt werden. Wird der Gadget-Treiber für CDC geladen, so ist es möglich, eine Netzwerkverbindung über USB aufzubauen. Die Gadget-Treiber schließen sich gegenseitig aus. Pro USB Device Controller kann immer nur ein Gadget-Treiber geladen sein. Es ist jedoch

ohne weiteres möglich, einen Treiber zu entfernen und danach einen anderen zu laden.

Das Ersetzen eines Treibers ist durch die Modularisierung des Kernel leicht realisierbar. Es ist sehr leicht möglich, unter Linux bestimmte Treiber als Modul zu erstellen, das dann später zur Laufzeit nachgeladen und genauso leicht auch wieder entfernt werden



! **Bild 2. Geräte und Treiber werden vom USB-Subsystem durch die IDs zugeordnet.**



! **Bild 3. Schichtenaufbau des USB-Gadget-Framework.**

kann. Sind die Gadget-Treiber in solche Module aufgeteilt, so kann durch Laden der passenden Module eine bestimmte Funktion realisiert werden. In den allermeisten Fällen ist die Neuentwicklung eines Gadget-Treibers nicht erforderlich. Nahezu alle relevanten Klassen werden bereits abgedeckt.

Je nachdem, welcher Gadget-Treiber geladen ist, schließen sich nach oben hin weitere Schichten des Linux-Kernel an, für Massenspeicher ist es das Filesystem, für CDC der Netz-

werkstack und serielle Gadget-Treiber sind im „serial“-Subsystem verankert.

## ■ Systematische Verifikation

Die Funktion des USB-Subsystems auf Client- und Hostseite kann umfassend verifiziert werden. Hierbei hilft in erster Linie der Gadget-Treiber „g\_zero“ mit seiner Host-Gegenstelle „usbtest“. Als Hostrechner ist eine Linuxmaschine erforderlich, die den USB-Treiber „usbtest“ enthält. Die meisten Linux-Distributionen enthalten diesen Treiber standardmäßig.

Zu Beginn der Verifikation wird der Client nur mit dem UDC-Treiber gestartet. Sobald dieses System mit dem Host verbunden wird, passiert zunächst noch nichts. Denn solange keine eindeutige USB-Funktion des

```
# generate empty file sized 16MB
dd if=/dev/zero of=/tmp/image bs=1M count=16
# map file to free loop device
losetup /dev/loop1 /tmp/image
# create a fat filesystem
mkfs -t vfat /dev/loop1
# mount it
mount /dev/loop1 /mnt/
# copy some files to it
cp <files> /mnt/
# unmount it
umount /mnt/
# terminate the loop
losetup -d /dev/loop1
#now /tmp/image can be used
```

### I Erzeugung eines Dateisystemimages für den Massenspeicher-Treiber

Client durch den Gadget-Treiber festgelegt ist, muss das USB-Gerät für den Host unsichtbar sein. Dies wird durch Abschalten des Pull-Up-Widerstandes erreicht, welcher auch die Geschwindigkeit codiert.

Sobald ein Gadget-Treiber – zur Verifikation eignet sich „g\_zero“ am besten – geladen wird, wird der Pull-Up-Widerstand durch den UDC-Treiber auf 3,3 V gelegt und signalisiert so dem Host, dass er mit der Enumeration beginnen kann. Während der Enumeration wird zunächst der device descriptor durch den Host abgeholt, oftmals gar nicht vollständig, denn der Host führt sehr schnell einen Reset durch. Dies ist ein normales, wenn auch ungewöhnliches Verhalten. Direkt nach dem Reset wird dem Device eine 7 bit breite USB-Adresse zugeordnet, über die es nun ausschließlich

angesprochen wird. Erst jetzt werden nahezu alle Deskriptoren vom Host abgefragt. Unter anderem wird ein weiteres Mal der device descriptor (Tabelle 2) übertragen. Dieser enthält IDs, mittels derer der Host nach einem passenden Treiber suchen kann. Der Gadget-Treiber „g\_zero“ definiert einen device descriptor, dessen IDs zum Laden des Treibers „usbtest“ auf der Linux-Hostseite führt.

Mit Hilfe dieses Treibers auf dem Host können nun Tests durchgeführt werden. Es besteht die Möglichkeit, Datenpakete hin und her zu senden und viele Varianten der Enumeration zu überprüfen. Außerdem wird das Verhalten im Fehlerfall verifiziert. Die Website <http://www.linux-usb.org/usbtest/index.html#gadgets> gibt hierzu genauere Informationen, auch findet man hier einige Skripts und Programme, die das Testen wesentlich erleichtern. Das Kernelmodul „g\_zero“ kann mittels „rmmod gzero“ auf dem Client wieder entladen werden. Der USB-Host stellt einen Disconnect fest und entlädt seinerseits den Treiber „usbtest“.

Zur weiteren Verifikation gibt es von USB.org das Testprogramm „USBCV“. Es kann auf einem Windows-Host mit USB 2.0 installiert werden. Das Gerät muss dann über einen Hub an den Hostcontroller angeschlossen werden. Nach Ende der Prüfung erzeugt das Programm ein HTML-Dokument, das dann als Protokoll dienen kann.

## ■ Nutzung der Gadget-Treiber

Erst wenn alle vorgesehenen Tests reproduzierbar erfolgreich durchgeführt wurden, lohnt es sich, einen anderen USB-Gadget-Treiber zu versuchen. Am einfachsten kann ein weiterer Test mit dem USB-Gadget-Treiber für die Klasse CDC durchgeführt werden. Dabei wird eine Netzwerkverbindung zwischen dem USB-Host und dem Gerät hergestellt.

Das erforderliche Kernelmodul „g\_ether“ wird mit „insmod g\_ether“ geladen; sofort beginnt der Host mit der Enumeration. Da „g\_ether“ einen device descriptor mit entsprechender Klassen-ID enthält, lädt der Host den Treiber „usbnet“. Sowohl auf dem

Gerät als auch auf dem Host sind damit zusätzliche Netzwerkinterfaces angelegt worden. Sie sind in der Ausgabe von „ifconfig -a“ an dem Prefix „usb“ zu erkennen. Diese Interfaces stehen miteinander in Verbindung. Wird ihnen beispielsweise mittels „ifconfig usb0 192.168.1.x“ jeweils eine andere IP-Adresse aus demselben Netzwerksegment zugeteilt, können IP-Pakete zwischen Host und Gerät ausgetauscht werden. Mittels eines wechselseitigen „ping“ auf die jeweilige Gegenstelle kann schnell die Funktion überprüft werden.

Für USB-Geräte der Klasse CDC steht nur unter Linux mit „usbnet“ ein adäquater Treiber zur Verfügung. Soll jedoch ein Windows-Rechner als Host verwendet werden, so muss das Device RNDIS beherrschen. Der Gadget-Treiber „g\_ether“ verfügt über die notwendige RNDIS-Unterstützung. Je nachdem, welche Art Host vorliegt, verwendet „g\_ether“ CDC oder RNDIS. Da RNDIS aber keine Geräteklasse im Sinne von USB ist, erfolgt die Zuordnung zwischen Gerät und Treiber auf dem Host nicht über eine KlassenID, sondern über die Vendor- und DeviceID. Klassisches Plug-and-Play ist mit Windows und USB-Netzwerk somit leider nicht möglich und die Installation einer inf-Datei unter Windows ist erforderlich. Im Linux-Kernelsource ist hierfür bereits eine Vorlage vorhanden unter Documentation/usb/linux.inf.

Für USB-Clients der Klasse Massenspeicher stellt sich dies ganz anders dar. Der Host-Treiber kann sowohl von Windows als auch von Linux über die KlassenID gefunden werden. Eine Installation auf dem Host ist genauso wie bei einem USB-Stick nicht notwendig. Das Modul „g\_file\_storage“ enthält hierfür die erforderlichen Funktionen und bedarf mindestens des Parameters „file“. Dieser Parameter enthält die Pfadangabe einer Datei, die als Hintergrundspeicher dient. Hierin werden alle Daten gespeichert, die dem Host über USB zur Verfügung gestellt werden. Üblicherweise enthält die Datei ein FAT-Dateisystem, welches sich leicht mit den Befehlen im obigen **Listing** erzeugen lässt. Das Laden des Moduls sieht dann wie folgt aus:

```
insmod g_file_storage file=/tmp/image
```

Sobald die Enumeration abgeschlossen wurde, bindet der Host üblicherweise den Inhalt in seine Verzeichnishierarchie mit ein; sei es als zusätzliches Laufwerk mit entsprechendem Buchstaben unter Windows oder als Verzeichnis im Ordner /media/, wie in vielen Linux-Distributionen üblich. Nun können beliebige Dateien hinzugefügt oder gelöscht werden, je nachdem, wie groß die Image-Datei auf dem USB-Gerät ist.

### ■ Fallstricke für die Anwendungen

Es ist nun sehr verlockend, über diese Verbindung Dateien zwischen Host und Gerät auszutauschen. Leider gibt es hierbei einige Fallstricke. Die Klasse „Mass Storage“ ist grundsätzlich nicht für den mehrfachen und gleichzeitigen Zugriff ausgelegt. Sie verfügt über kein Kohärenz-Protokoll, um den Kommunikationspartner über geänderte Dateien zu unterrichten. Der USB-Host kann also ein ganz anderes Bild vom Inhalt des Speichers haben, als der Client es hat, und umgekehrt. Es sollte grundsätzlich vermieden werden, die Daten von Host und Client gleichzeitig zu benutzen. Gegen eine zeitlich abgegrenzte Nutzung ist dagegen nichts einzuwenden. Es ist jedoch schwierig, den gegenseitigen Ausschluss zu jedem Zeitpunkt sicherzustellen.

Eine weitere Möglichkeit der Kommunikation über USB bietet der Gadget-Treiber „g\_serial“. Der USB-Client kann mit „g\_serial“ wie ein USB-to-Serial-Konverter eingesetzt werden. Nach Enumeration und Laden des passenden Treibers auf dem Host sind auf Host und Client virtuelle serielle Schnittstellen eingerichtet worden. Eine Datenübertragung zwischen den Schnittstellen ist wie mit einem Null-Modem-Kabel möglich. Diese Vorgehensweise bietet sich oft an, wenn es schon ein bewährtes serielles Protokoll gibt und dieses möglichst ohne Änderungen übernommen werden soll.

Das Gegenteil hierzu bietet der Gadget-Treiber „gadgetfs“ an. Mit diesem

Treiber kann das USB-Interface komplett aus der Applikation heraus genutzt werden.

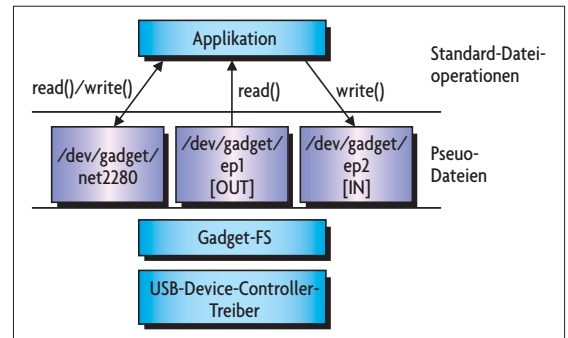
Als Schnittstelle zur Applikation benutzt Gadgetfs einige Pseudo-Dateien (Bild 4), die üblicherweise unter /dev/gadget/ zu finden sind. Durch Schreiben von Deskriptoren in die Datei /dev/gadget/<udc-name> wird das Gerät konfiguriert. Die Datenübertragung über die Endpunkte erfolgt dann mittels der Dateien /dev/gadget/ep<nummer>. Ist ein Endpunkt als OUT konfiguriert, so kann von ihm nur gelesen werden, IN-Endpunkte können nur beschrieben werden.

Gadgetfs bietet sich immer dann an, wenn es gilt, ein spezielles neues Protokoll über USB zu realisieren. In vielen Fällen ist dies jedoch nicht erforderlich, da die USB-Klassen CDC (Netzwerk) und Massenspeicher sowie die Pseudo-Klasse „serial“ genügend Funktionen bieten. Ausnahmen bilden Anwendungen, in denen die Übertragung von großen Datenströmen in Echtzeit erforderlich ist, wie es bei Audio- und Videodaten oft gefordert wird.

### ■ USB On-The-Go

USB-OTG (On-The-Go) ermöglicht einem System, sowohl in die Rolle des Host als auch des Gerätes zu schlüpfen – je nachdem, womit es verbunden wird. Sollten zwei OTG-Systeme aufeinander treffen, so werden durch einen speziellen Algorithmus die Rollen verteilt. Die Programmierschnittstellen von Linux sind hierfür bereits vorbereitet. Abhängig von der Art der Grenzstelle kommt entweder ein UDC- oder ein Hostcontroller-Treiber zum Einsatz. Für Bausteine der OMAP-Familie von Texas Instruments sind alle benötigten Teile bereits seit Kernel 2.6.9 implementiert.

Das gesamte USB-Gadget-Framework macht einen sehr sauberen und übersichtlichen Eindruck; eine ausführliche Dokumentation ist verfügbar. Im praktischen Einsatz hat sich das Framework schon in mehreren Projekten bewährt. Was die Gadget-Schicht angeht, so ist das Treiberangebot des Linux-Kernel nahezu vollständig. Ausnahmen bilden Audio- und Video-Gadget-Treiber, die jedoch zurzeit noch eher selten erforderlich sind.



! Bild 4. Verwendung des USB-Gadget-Treibers durch eine Anwendung.

Die Zahl der UDC-Treiber ist dagegen leider etwas klein – eher selten ist ein passender dabei. Hier ist oftmals noch Entwicklungsarbeit nötig. Mit der dem Linux-Kernel beiliegenden Dokumentation und den anderen Treibern in Linux als Vorlagen ist dies jedoch mit vertretbarem Aufwand realisierbar. Der Ausbau von USB-Gadget, um USB-OTG zu unterstützen, ist bereits erfolgt, und seine Benutzbarkeit ist bewiesen worden. Linux ist also OTG-ready.

Das USB-Gadget-Framework ist zudem ein typisches Beispiel dafür, wie wichtig eine sinnvolle horizontale Unterteilung der Software-Module ist. Wer einmal einen UDC-Treiber entwickelt hat, kann frei entscheiden, ob der Client nun ein USB-Stick oder ein Netzwerkmodul oder etwas ganz anderes sein soll. Auch während der Laufzeit ist durch Laden und Entladen der Module leicht ein Wechsel möglich.

jk



**Thomas Brinker**

studierte an der Technischen Universität Berlin Technische Informatik. Seit 2005 ist er bei der emlix GmbH in Göttingen als Embedded Linux Systemingenieur beschäftigt und dort an der hardwarenahen Entwicklung von Produkten für die Medizintechnik, Automatisierungstechnik und Datentechnik beteiligt. Er arbeitet im Bereich der Kernel- und Treiberentwicklung und unterstützt Kunden bei deren Applikationsentwicklung als Berater.

[tb@emlix.com](mailto:tb@emlix.com)