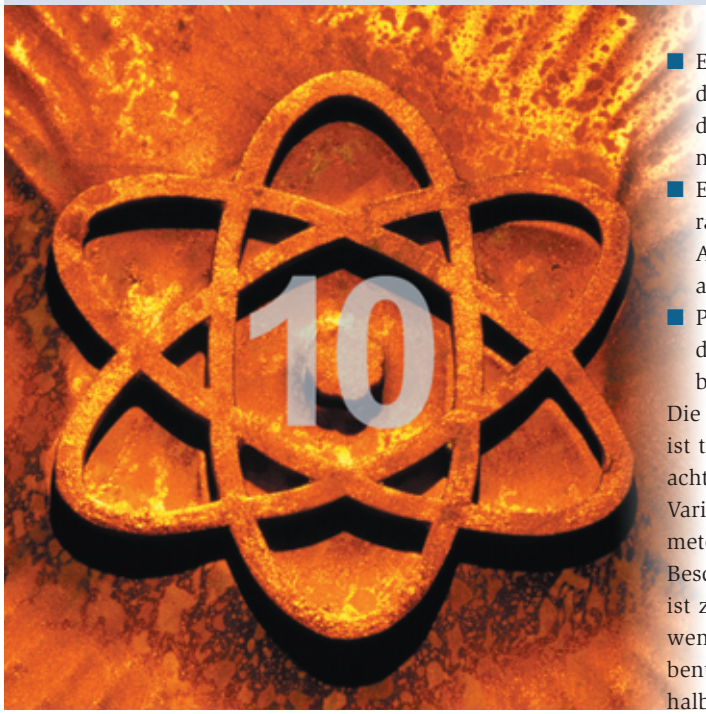
Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 10

Kern-Technik

Aus Platz- und Übersichtsgründen lagert der Kernel seine Funktionalität größtenteils in Module aus. Dieser Vorgang lässt sich über Parameter fein einstellen. Linux 2.6 bietet Typsicherheit für Modulparameter und erlaubt es dem Kernelprogrammierer, dafür eigene Datentypen zu verwenden. Eva-Katharina Kunst, Jürgen Quade



Auch bei Modulen bringt Linux 2.6 im Vergleich zu 2.4 kleine Veränderungen mit: Kernelprogrammierer müssen Parameter nun anders implementieren, können aber eigene Datentypen definieren. Wenn der neue Kernel dynamische Module lädt und dabei Parameter übergeben bekommt, kann er sie auf ihre Gültigkeit überprüfen. IO-Adressen, Speicherbereiche, Gerätemerkmale und Betriebsarten sind nur einige Anwendungsbeispiele dafür.

Die Parameter eines Moduls bestehen aus zwei Komponenten: dem Namen und dem Wert. Die dafür vordefinierten Datentypen zeigt Tabelle 1. Den Datentyp »byte«, im Kernel 2.4 noch vorhanden, gibt es in 2.6 nicht mehr. Um in einem eigenen Modul einen Parameter zu implementieren, muss der Programmierer Folgendes tun:

- Eine Variable definieren, die im laufenden Modul den Parameterwert aufnimmt.
- Eine Beschreibung des Parameters erstellen, die der Anwender des Moduls auslesen kann.
- Per Makro den Parameter dem Kernel bekannt geben.

Die Definition der Variablen ist trivial. Hier ist nur zu beachten, dass der Datentyp der Variablen mit dem des Parameters übereinstimmt. Die Beschreibung des Parameters ist zwar nicht zwingend notwendig, aber für den Modulbenutzer oft nützlich, deshalb sollte der Kernelprogrammierer diese Zusatzarbeit leisten. Dazu übergibt er dem Makro »MODULE_PARM_DESCR()« den beschreibenden Text. Der Anwender liest diese Beschreibung mit dem Programm »mod_info«.

Makros für alle

Um den Parameter dem Kernel bekannt zu machen, stehen gleich zwei Makros zur Auswahl. Am einfachsten ist es, wenn der Name des Parameters mit dem Namen der Variablen übereinstimmt. In diesem Fall kommt das Makro »module_param()« zum Einsatz. Es bekommt den Variablen- beziehungsweise den Parameternamen übergeben, den Datentyp (siehe Tabelle 1) und einen Dateizugriffsmodus. In einer der kommenden Kernelversionen wird das Gerätemodell [1] nämlich eine Gerätedatei mit dem Namen des Parameters erstellen. Der Zugriffsmodus auf diese Gerätedatei wird dann dem an dieser Stelle angegebenen Zugriffsmodus entsprechen.

Die Funktionen und Makros für Modulparameter sind in »linux/moduleparam.h« definiert. Listing 1 zeigt hierzu ein Beispiel: ein Modul, dem der Benutzer beim Laden den Parameter »myint« vom Typ »int« übergeben kann. Aus »param1.c« (Listing 1) erzeugt der Compiler nach Anleitung des Makefile (Listing 2) das Modul »param1.ko«, das der Befehl »insmod« in den laufenden Kernel lädt (Abbildung 1). Liegt der angegebene Parameter außerhalb des Definitionsbereichs, wird das Modul nicht geladen. So entspricht der Wert »abcd« nicht der Definition eines Integer-Parameters.

Nur mit einem gültigem Parameter wie »myint=1234« lässt sich das Modul laden. Das Syslog (meist »/var/log/messages«) zeigt den Wert des Parameters, den das Modul mit »printk()« ausgibt (Lis-

Tabelle 1: Mögliche Datentypen für Modulparameter

Bezeichnung	Beschreibung
short	Short-Wert mit Vorzeichen
ushort	Short-Wert ohne Vorzeichen
int	Integer-Wert mit Vorzeichen
uint	Integer-Wert ohne Vorzeichen
long	Long-Wert mit Vorzeichen
ulong	Long-Wert ohne Vorzeichen
charp	Zeiger auf einen String
bool	boolescher Wert, mögliche Werte sind »y«, »Y«, »n«, »N«, »0«, »1«
Array	Feld eines der Standard-Datentypen
String	Zeichenfeld

ting 1, Zeile 12). Der Befehl »rmmod param1« entfernt das Modul wieder.

Stimmen die Namen von Variable und Parameter hingegen nicht überein, hilft das Makro »module_param_named()« weiter. Es besitzt als erstes Argument den Namen des Modulparameters, als zweites den Variablennamen im Modul. Die letzten beiden Parameter von »module_param_named()« repräsentieren wieder den zugehörigen Datentyp und die Zugriffsrechte. Listing 3 zeigt die Verwendung des Makros.

Für einen Test lässt sich das Modul wieder wie oben beschrieben laden. Nur lautet jetzt der Name des Modulparameters »maxtime«, während die modulinterne Variable den alten Namen »myint« weiterhin behält.

Felder und Strings

Wie Tabelle 1 zeigt, stehen neben den Standarddatentypen auch Feld- beziehungsweise String-Datentypen zur Auswahl. Sie in einem Kernelmodul einzusetzen ist etwas schwieriger, doch auch für sie existieren jeweils eigene Makros. Außerdem braucht das Modul für Strings und Felder zwei Variablen, denn der Kernel muss über Feld- und Stringlänge Buch führen.

Zusätzlich teilt er dem Treiber mit, wie viele Elemente des Parameters der Anwender tatsächlich beim Aufruf verwendet hat. Leider ist hierbei nicht zu unterscheiden, ob der Benutzer sämtliche Elemente vorbelegt hat oder ob der Parameter gänzlich unbesetzt blieb.

Das Makro zur Übergabe eines Felds als Modulparameter heißt »module_param_array()«. Es besitzt ebenfalls vier Parameter: Variablen- beziehungsweise Parametername, Datentyp des Feldes, Anzahl der Feldelemente und schließlich die Zugriffsrechte. Eine Variante, bei der Variablenname und Parametername nicht übereinstimmen müssen, existiert ebenfalls, und zwar »module_param_array_named()«. Listing 4 zeigt den Einsatz der Makros anhand eines Beispiels. Es wertet die Zählervariable aus und zeigt die Anzahl der vom Anwender übergebenen Variablen an.

Für Strings als Modulparameter stehen die beiden Makros »module_param_string()« und »module_param_string_named()« zur Verfügung. Ihre Parameter entsprechen denen der Feldmakros. Listing 5 zeigt ihre Anwendung.

Typ-Überprüfung

Der Modullader überprüft, ob Parameter und Datentyp übereinstimmen. Ist dies nicht der Fall, entfernt er das Modul gleich wieder aus dem Kernel. Das gilt auch für die selbst definierten Datentypen, doch es ist schwieriger, die Typüberprüfung zu implementieren: Der Programmierer muss dazu eine so genannte Set-Funktion bereitstellen, die überprüft, ob der angegebene Parameterwert in den Wertebereich des Datentyps passt oder nicht. Die Get-Funktion konvertiert dagegen ein übergebenes Datum wieder in die ursprüngliche Darstellung. Ein kurzes Beispiel zeigt, wie man die Funktionen verwendet.

Der selbst definierte Datentyp soll nur gerade Werte und die Zahl 0 zulassen (0, 2, 4, 6 …) und deshalb »even« heißen. Die Funktion, die den übergebenen Parameterwert überprüft, soll »param_set_even()« heißen. Sie bekommt vom Kernel je einen Zeiger auf einen String und auf ein Element vom Typ »struct kernel_param« übergeben. Diese Struktur repräsentiert Kernel-intern die Modulparameter. Sie enthält unter anderem einen Zeiger auf die Variable, in die der Modullader den Wert abgelegt (das Element »void *arg«). Sobald der Modullader ein

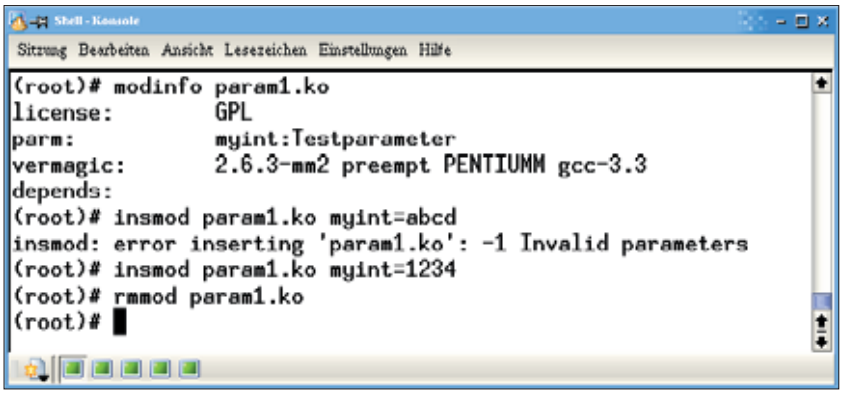


Abbildung 1: Der Versuch, mit »insmod« das Beispiel aus Listing 1 in den laufenden Kernel zu laden, funktioniert nur, wenn der Parameter innerhalb des definierten Wertebereichs liegt.

Listing 1: Parameter mit »module_param()«

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03
04 MODULE_LICENSE("GPL");
05 MODULE_PARM_DESC(myint, "Testparameter");
06
07 static int myint;
08 module_param( myint, int, 666 );
09
10 static int __init mod_init(void)
11 {
12         printk("myint = %d\n", myint);
13         return 0;
14 }
15
16 static void __exit mod_exit(void)
17 {
18 }
19 module_init( mod_init );
20 module_exit( mod_exit );
```

Listing 2: Makefile für »param1.c«

```
01 TARGET=param1
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
```

Listing 3: Makro »module_param_named()«

```
01 ...
02 static int myint;
03 module_param_named(maxtime, myint, int, 666);
04
05 static int __init mod_init(void)
06 {
07    printk("myint = %d\n", myint);
08    return 0;
09 }
10 ...
```

Modul mit einem selbst definierten Parameter lädt, ruft der Kernel die Funktion »param_set_even()« auf. Sie konvertiert den als String übergebenen Parameter in den internen Datentyp und prüft, ob er innerhalb des Wertebereichs liegt, und speichert einen gültigen Wert ab.

Listing 6 zeigt in den Zeilen 13 bis 17 die Implementierung. Der Wert selbst wird durch »simple_strtoul()« in einen Integerwert konvertiert (Zeile 13). Die folgende Abfrage überprüft mit Hilfe der Modulo-Funktion, ob der Wert gerade oder ungerade ist (Zeile 14). Wenn er ungerade ist, gibt die Testfunktion »-EINVAL« (ungültiger Wert) zurück (Zeile 15), sonst »0«.

Die Get-Funktion hat die Aufgabe, die interne Form des Parameters zurück in eine Ascii-Kodierung zu transformieren. Das geht am einfachsten mit »sprintf()«, wobei nur sicherzustellen ist, dass nicht mehr Bytes geschrieben werden als in einer Speicherseite Platz haben (Zeile 22). Die Adresse, an die »sprintf()« das Ergebnis schreiben soll, ist als Parameter der Get-Funktion übergeben worden: »char *pbuf«.

Neben den beiden Funktionen muss der Programmierer noch ein Makro definieren, das unter Umständen eine Prüfung des Typs durch den Compiler ermöglichen könnte. In fast allen Fällen kann das Makro aber auch einfach leer bleiben.

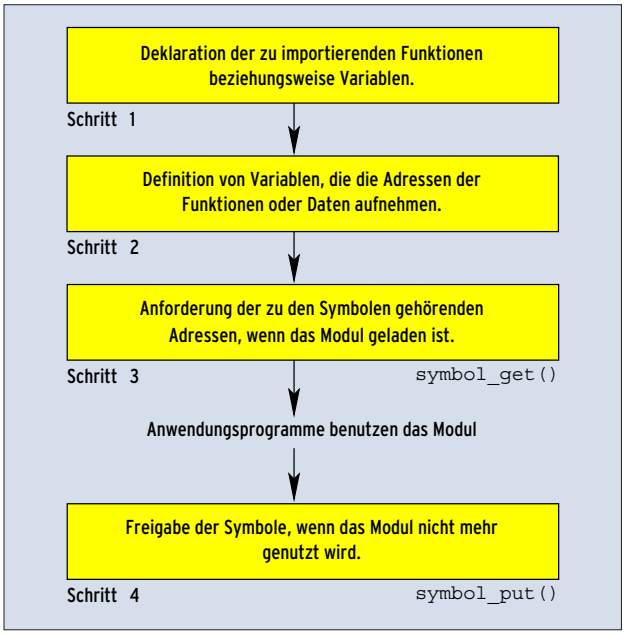


Abbildung 2: Vier Schritte zur Implementierung von Intermodul-Kommunikation.

Listing 4: Übergabe von Feldern

```
01 #include <linux/module.h>
02 #include <linux/moduleparam.h>
03
04 MODULE_LICENSE("GPL");
05
06 static int intarray[4];
07 static int intarraycount=4;
08 module_param_array(intarray, int, intarraycount, 666);
09 MODULE_PARM_DESC(intarray, "Ein Feld mit bis zu 4 Int");
10
11 static int __init mod_init(void)
12 {
13    printk("intarraycount=%d\n", intarraycount);
14    for(; intarraycount; intarraycount--)
15        printk("%d: %d\n", intarraycount,
16        intarray[intarraycount - 1]);
17    return 0;
18 }
19
20 static void __exit mod_exit(void)
21 {
22 }
23 module_init(mod_init);
24 module_exit(mod_exit);
```

Listing 5: Definition eines String-Parameters

```
01 ...
02 static char string[10];
03 module_param_string( optionname, string,
   sizeof(string), 666 );
04
05 static int __init mod_init(void)
06 {
07    if( string[0] )
08       printk("string: %s\n", string);
09    ...
```

Namen zusammensetzen

Der Name dieses Makros leitet sich aus dem Präfix »param_check_« und dem selbst erteilten Namen des Parameters ab. Damit ergibt sich das in Listing 6 in Zeile 7 dargestellte Makro »param_check_even()«. Jetzt fehlt nur noch der Parameter, den das Makro »module_param()« nach dem bekanntem Schema definiert (Zeile 26). Der Versuch, das Modul einmal mit dem Parameter »para=5« zu laden, schlägt wie erwartet fehl. Ein Aufruf des Moduls mit »para=2« hingegen ist erfolgreich.

Parameter stehen natürlich nicht nur Modulen zur Verfügung. Auch einem in den Kernel gelinkten Treiber lassen sich beim Booten Parameter übergeben. Die Implementierung erfolgt wie beim Modultreiber, nur der Aufruf gestaltet sich etwas anders: Vor dem Parameternamen steht der Name des Moduls. Beispielsweise steht bei »psmouse.proto=bare« der Name »psmouse« für das Mausmodul, »proto« ist der Parametername und »bare« dessen Wert.

Import und Export

Module bekommen aber nicht nur beim Laden Parameter übergeben, sondern interagieren auch mit anderen Modulen. Auf der einen Seite stellen sie Variablen

Lizenz zum Linken

Ob der Quellcode von Modulen, die der Linux-Kernel einbindet, veröffentlicht werden muss oder nicht, ist schon seit jeher eine Streitfrage. Im Grunde geht es darum, ob ein Modul den Kernel nur nutzt oder ihn erweitert. Linus Torvalds erkennt die rechtliche Grauzone an, auch wenn er klar Stellung bezieht. Seiner Meinung nach muss der Code eines jeden nach außen gegebenen Moduls veröffentlicht werden. Support gibt es ohnehin von der Entwicklergemeinde nicht mehr, wenn der Kernel „tainted" ist, wenn sich also im Kernel ein Modul ohne passende Lizenz befindet.

Darüber hinaus können die Entwickler selbst definieren, ob eine Funktion von einem Binary-only-Treiber (also einem Treiber ohne Sourcecode) genutzt werden darf oder nur von einem Treiber mit einer passenden Lizenz (wie die GNU Public License GPL). Dies geschieht über die im Artikel vorgestellten Makros »EXPORT_SYMBOL()« und »EXPORT_SYMBOL_GPL()«. Im Kernel 2.6.3 gibt es über 300 Funktionen, die nur von einem GPL-konformen Treiber genutzt werden können.

Um diese Einschränkung zu umgehen, tricksen Hersteller von Binary-only-Treibern oft: Sie unterteilen ihren Treiber in einen Open-Source- und einen Binary-only-Teil. Der Open-Source-Teil führt ein Mapping von geschützten Funktionen auf neue Funktionen durch, die dann das Makro »EXPORT_SYMBOL()« auch an Binary-only-Treiber durchreicht.

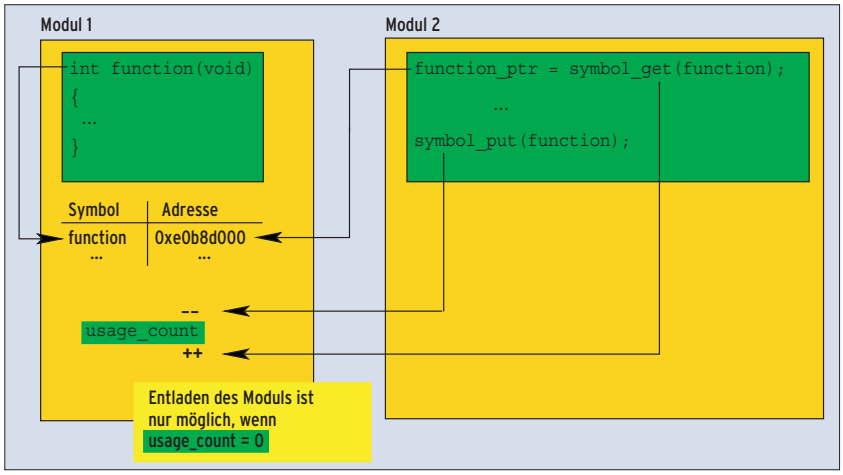


Abbildung 3: Die Makros »symbol_get()« und »symbol_put()« verändern den »usage_count« des exportierenden Moduls.

sowie Funktionen zur Verfügung, auf der anderen nutzen sie die des Kernels und anderer Module.

Das Bereitstellen von Variablen und Funktionen – das Exportieren – gestaltet sich einfach. Eine zu exportierende Variable oder Funktion geben die Makros »EXPORT_SYMBOL()« oder »EXPORT_SYMBOL_GPL()« bekannt. Beim Einsatz von »EXPORT_SYMBOL()« darf jede andere Kernelkomponente das Symbol verwenden. Wird dagegen »EXPORT_SYMBOL_GPL()« gewählt, kann das exportierte Symbol nur von Modulen genutzt werden, die ihrerseits unter einer BSD- oder GPL-Lizenz stehen (siehe Kasten „Lizenz zum Linken“).

Komplexer wird die Intermodul-Kommunikation allerdings, wenn es zwischen dem exportierenden und dem importierenden Modul keine hierarchische Abhängigkeit gibt. Dazu das folgende Szenario: Modul 1 exportiert ein Symbol »good_bye« und möchte zugleich ein Symbol »hello« importieren. Modul 2 wiederum importiert ein Symbol »good_bye« und exportiert »hello«.

Der Versuch, das Modul 1 zu laden, misslingt, da für das Symbol »hello« keine Adresse, also keine Repräsentation im Kernel gefunden werden kann. Modul 2 lässt sich ebenfalls nicht laden, denn bei ihm kann der Kernel das Symbol »good_bye« nicht auflösen – das Modul 1, das die zum Symbol gehörige Funktion oder Variable zur Verfügung stellt, ist ja nicht geladen.

Modul an Modul

Kernel 2.4 besitzt für diesen Fall die Funktionen »inter_module_register()«, »inter_module_unregister()«, »inter_module_get()« und »inter_module_put()«. Bei Linux 2.6 fehlen diese Funktionen, da sie nicht vernünftig gegen Race Conditions abzusichern sind.

An ihre Stelle treten die Makros beziehungsweise Inline-Funktionen »symbol_get()«, »symbol_put()« (siehe Abbildung 2) und »symbol_put_addr()«. Sie sind übrigens vom Kernel mit »EXPORT_SYMBOL_GPL()« exportiert, also können

Listing 6: Einschränkung von Parameterwerten

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <linux/moduleparam.h>
04
05 MODULE_LICENSE("GPL");
06
07 #define param_check_even(name, p)
08
09 int param_set_even(const char *pbuf, struct
   kernel_param *kp)
10 {
11    int number;
12
13    number = simple_strtoul(pbuf, NULL, 0);
14    if(number % 2) // ungerade?
15            return -EINVAL;
16    *(int *)kp->arg = number;
17    return 0;
18 }
19
20 int param_get_even(char *pbuf, struct
   kernel_param *kp)
21 {
22    return sprintf((char *)pbuf, "%d", *(int
   *)kp->arg);
23 }
24
25 static int para;
26 module_param(para, even, 0666);
27
28 static int __init mod_init(void)
29 {
30    printk("para = %d\n", para );
31    return 0;
32 }
33
34 static void __exit mod_exit(void)
35 {
36 }
37 module_init( mod_init );
38 module_exit( mod_exit );
```

Listing 7: »icm.ko«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03 #include <asm/io.h>
04
05 MODULE_LICENSE("GPL");
06
07 static char *textbuf = "Hallo Treiber";
08
09 void ton_an(u16 tonwert)
10 {
11    s8 save;
12
13    if( tonwert ) {
14       tonwert = CLOCK_TICK_RATE/tonwert;
15       printk("ton_an(0x%x)\n", tonwert);
16       outb( 0xb6, 0x43 );
17       outb_p(tonwert & 0xff, 0x42);
18       outb((tonwert>>8) & 0xff, 0x42);
19       save = inb( 0x61 );
20       outb(save | 0x03, 0x61);
21    } else {
22       outb(inb_p(0x61) & 0xFC, 0x61);
23    }
24 }
25
26 static int __init buf_init(void)
27 {
28    printk(textbuf);
29    return 0;
30 }
31
32 static void __exit buf_exit(void)
33 {
34    outb(inb_p(0x61) & 0xFC, 0x61);
35 }
36
37 module_init(buf_init);
38 module_exit(buf_exit);
39 EXPORT_SYMBOL_GPL(ton_an);
40 EXPORT_SYMBOL_GPL(textbuf);
```

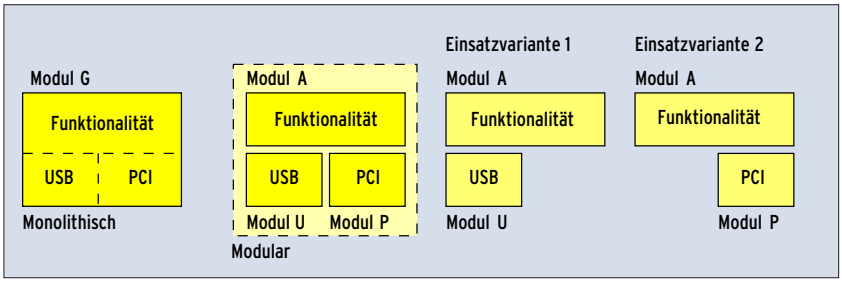


Abbildung 4: Die modulare Lösung trennt USB-, PCI- (Module U und P) und Anwendungsfunktionen (Modul A). Alternativ dazu packt der Programmierer alles auf einmal in Modul G.

nur jene Treiber sie nutzen, die selbst unter einer BSD-Lizenz oder der GNU General Public License stehen.

Symbole finden

Da Symbole nicht beim Laden aufgelöst werden können, geschieht dies zur Laufzeit. Will ein Modul ein Symbol importieren, muss es dazu Speicher (einen Pointer) reservieren, der später die Adresse des Symbols aufnimmt. Diese Adresse fordert das Makro »symbol_get()« an, dessen einziger Parameter der Name des Symbols ist. Das Makro »symbol_get()« baut den Namen des Symbols so zu einem String um, dass innerhalb des Kernels die zugehörige Adresse gesucht werden kann. Ist das Symbol aufgelöst, gibt die Funktion die Adresse zurück, sonst »NULL«.

Darüber hinaus zählt die Funktion »symbol_get()«, wie viele andere Komponenten auf das angeforderte Symbol zugreifen. Solange ein Zählwert größer null vorliegt, lässt sich das exportierende Modul nicht entladen. Ein Aufruf von »symbol_put()« oder alternativ »symbol_put_addr()« dekrementiert den Zähler wieder (Abbildung 3). Zu jedem »symbol_get()« gehört daher ein »symbol_put()« oder ein »symbol_put_addr()«. Das Makro »symbol_put()« besitzt als Parameter den Namen des Symbols, während »symbol_put_addr()« das Symbol über seine Adresse findet.

Die Listing 7 und 8 zeigen die Implementierung zweier Module »icm.ko« und »stacked.ko«. Dabei nutzt »stacked.ko« eine Funktion und eine Variable von »icm.ko«, lässt sich aber auch dann laden, wenn sich das Modul »icm.ko« noch nicht im Speicher befindet. Lädt der Anwender per »insmod« nur das Modul »stacked.ko«, erscheint im Syslog die Meldung, dass der Kernel die beiden Symbole »ton_an« und »textbuf« nicht finden konnte. Lädt er zunächst »icm.ko« und anschließend »stacked.ko«, müsste der PC-Lautsprecher einen Ton von sich geben. Icm basiert nämlich auf dem in [2] vorgestellten Speaker-Treiber. Das Entladen von »stacked.ko« schaltet den Lautsprecher wieder ab. Das Modul »icm.ko« lässt sich erst entladen, wenn auch »stacked.ko« aus dem Speicher entfernt ist.

Listing 8: »stacked.ko«

```
01 #include <linux/module.h>
02 #include <linux/init.h>
03
04 MODULE_LICENSE("GPL");
05
06 extern void ton_an(u16 tonwert);
07 extern char *textbuf;
08
09 static void (*ton_an_function)(u16 var);
10 static char **textbufptr;
11
12 static int __init buf_init(void)
13 {
14    ton_an_function = symbol_get(ton_an);
15    if (ton_an_function)
16       ton_an_function(330);
17    else
18       printk("can't find address of symbol
   \"ton_an\"\n");
19    textbufptr = symbol_get(textbuf);
20    if(textbufptr)
21       printk("content of 0x%p: \"%s\"\n",
   textbufptr, *textbufptr);
22    else
23       printk("can't find address of symbol
   \"textbufptr\"\n");
24    return 0;
25 }
26
27 static void __exit buf_exit(void)
28 {
29    if (ton_an_function) {
30       ton_an_function(0);
31       symbol_put(ton_an);
32       //symbol_put_addr(ton_an_function); //
   alternativ zu symbol_put
33    }
34    if (textbufptr)
35       symbol_put(textbuf);
36 }
37
38 module_init(buf_init);
39 module_exit(buf_exit);
```

Teile und herrsche?

Die hier vorgestellte Intermodul-Kommunikation sollte Entwickler nicht dazu verleiten, jedes Problem in diverse Module zu zerteilen. Das ist nur sinnvoll, wenn damit die Software besser skaliert oder flexibler wird. Lässt sich beispielsweise Hardware über USB oder PCI anschließen, mag es sinnvoll sein, für beide Busse je ein Modul zu erstellen. Ein drittes Modul deckt dann die eigentliche Funktionalität ab und kommuniziert mit Anwendungen (siehe Abbildung 4). Dieses Anwendungsmodul ließe sich mit einem USB- oder einem PCI-Modul betreiben.

Vorschau

Ein Blick in die Verzeichnisse »/lib/« und »/usr/lib/« zeigt, dass es für Applikationsprogrammierer viele Bibliotheken gibt, um ihnen die Entwicklungsarbeit zu erleichtern. Im Kernel gibt es dagegen keine Bibliotheken im klassischen Sinn. Trotzdem bietet der Kernel dem Entwickler eine Reihe hilfreicher Funktionen. Die wichtigsten stellt die nächste Folge der Kern-Technik vor. *(ofr)* ■

Infos

[1] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 6: Linux-Magazin 1/04, S. 94.

[2] Eva-Katharina Kunst und Jürgen Quade, „Kern-Technik", Folge 3: Linux-Magazin 10/03, S. 81.

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source.