Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 132

Mehrsprachigkeit

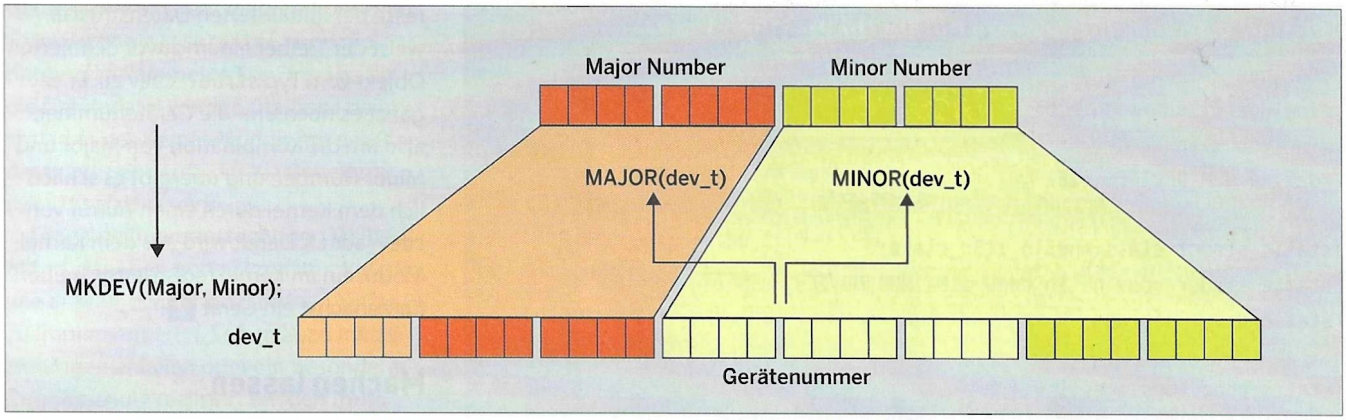
Gut eine Million Geräte pro Treiber oder 20 Bit zur Kodierung von Informationen: Als Bindeglied zwischen Applikation und Treiber lassen sich Gerätedateien für intuitive Interfaces nutzen. Eva-Katharina Kunst, Jürgen Quade

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von freier Software. Jürgen Quade, Professor an der Hochschule Niederrhein, führt auch für Unternehmen Schulungen zu den Themen Treiberprogrammierung und Embedded Linux durch.

Gerätetreiber als Tor zur Hardware sind für Applikationen durch ihre zugehörigen Gerätedateien repräsentiert. Ein bisschen Lesen einer solchen Datei hier und ein bisschen Schreiben in die Datei dort, und schon kennt man die aktuelle CPU-Temperatur oder hat einen Motor in Gang gesetzt. Technisch ist eine Gerätedatei mit zwei Nummern assoziiert, die man klassisch als Major Number sowie Minor Number bezeichnet. Die Major Number bestimmt den zugehörigen Gerätetreiber, der dann die Minor Number auswerten kann. Ein einzelner Treiber bedient damit kunterbunt verschiedenste Geräte oder stellt sein Verhalten auf Basis der Gerätedatei um. Der Character-Device-Treiber mit der Major Number 10 beispielsweise, der sogenannte Misc-Treiber, wertet Mausbewegungen aus, bedient Watchdogs, misst Temperaturen oder steuert LEDs an. Dieser eine Treiber allein verwaltet insgesamt 125 Geräte.

Dass ein Treiber gleich so viele unterschiedliche Geräte bedient, dürfte historisch bedingt sein. In der Unix-Steinzeit waren die Major Number und die Minor Number jeweils 8 Bit breit. Nach Adam



1 Intern verwendet Linux für jedes Gerät eine Gerätenummer.

Riese ergibt das maximal 256 zeichen- und 256 blockorientierte Gerätetreiber. Jeder Treiber konnte bis zu 256 Geräte bedienen oder 256 Verhaltensunterschiede aufweisen. Der Umstand, dass eine Major Number fest einem Treiber zugeordnet war, machte die Sache ein wenig knifflig: Ähnlich wie bei den raren IPv4-Adressen standen für neuzeitliche Anforderungen nicht genügend freie Nummern zur Verfügung. Eine erste Entschärfung brachte die dynamische Nummernvergabe im Linux-Kernel. Wenn allerdings die Major Number nicht von vorneherein bekannt ist, kann die Systemadministration auch die zugehörigen Gerätedateien nicht vorab anlegen.

Schluss mit gestern

Linus Torvalds löste diesen gordischen Knoten schließlich durch zwei Maßnahmen: Zum einen fasste er die 8 Bit lange Major Number mit der ebenso langen Minor Number zu einer 32-Bit-Gerätenummer zusammen 1, zum anderen implementierte er einen Mechanismus, durch den die Gerätedateien automatisiert angelegt werden können.

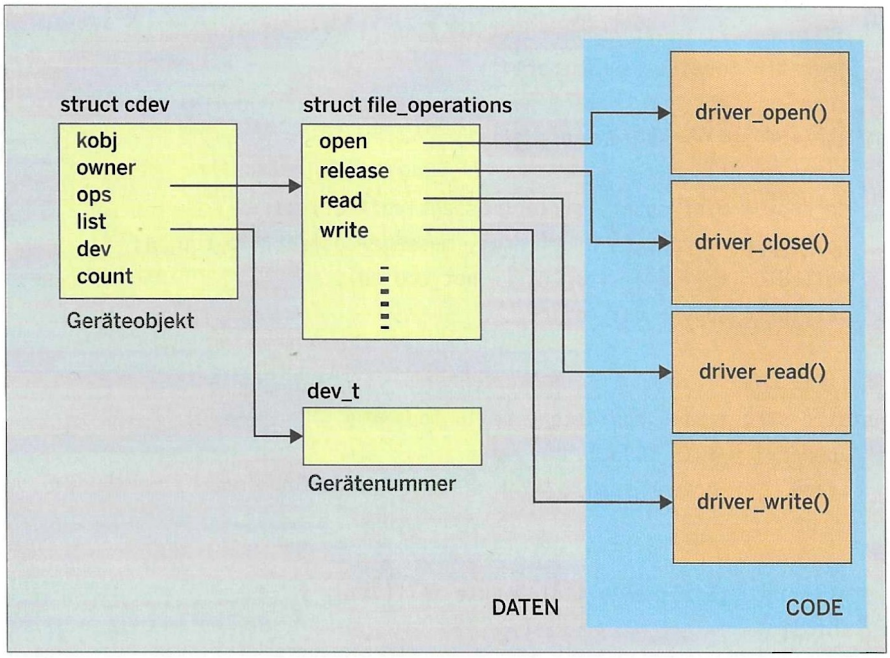
12 Bits der 32 Bit langen Gerätenummer repräsentieren die alte Major Number, sodass Linux jetzt 4095 Treiber unterstützt (die Major Number 0 bleibt reserviert). Die übrigen 20 Bits bilden die Minor Number, sodass jeder Treiber mehr als eine Million Geräte bedienen kann.

Bei einem Gerätetreiber handelt es sich um ein Kernel-Modul, das neben der Initialisierung und Deinitialisierung des Moduls applikationsgetriggerte Zugriffsfunktionen auf virtuelle oder reale Geräte implementiert. Die Modulinitialisierung nutzt der Gerätetreiber, um sich beim Kernel anzumelden. Dazu benötigt er eine oder mehrere Gerätenummern, die er sich über die Kernel-Funktion `alloc_chrdev_region()` zuteilen lassen kann.

Listen erstellen

Später, beim Entladen des Moduls und der Deinitialisierung via `module_exit()` werden die Gerätenummern durch Aufruf der Funktion `unregister_chrdev_region()` wieder freigegeben. Weil im Userland weiterhin die Major und Minor Numbers zum Einsatz kommen, vereinfacht Linux die Umwandlung zwischen den beiden Formaten über die Makros `MKDEV()`, `MAJOR()` und `MINOR()`.

Der Treiber meldet sich nicht nur mit der Gerätenummer an, sondern auch mit einer Liste von Adressen für Zugriffsfunktionen. Zu ihnen gehören beispielsweise `driver_open()`, `driver_read()`, `driver_write()` und `driver_close()`). Sie ruft der Kernel dann jedes Mal auf, wenn eine Applikation per `open()`, `read()`, `write()` oder `close()` auf die Gerätedatei zugreift. Die Datenstruktur, die die Adressen der Zugriffsfunktionen aufnimmt, hat den Typ `struct file_operations`. Die Ad-



2 Jedes Gerät wird durch eine Datenstruktur `cdev` repräsentiert.

Listing 1: Treiber für mehrere Geräte (hello-i18n.c) (Fortsetzung auf S. XX)

```
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/cdev.h>
#define NUM_DEVICES (16)

static struct class *hello_i18n_class;
static struct cdev hello_cdev_i18n[NUM_DEVICES];
static dev_t dev_num;

static char *messages[] = {
  "Hallo, Welt",
  "Hello, world",
  "Hola, mundo",
  "Hallo, wereld" };

static char *formatstring[] = {
  "%s\n",
  "<html>\n<h1>%s</h1>\n</html>\n",
  "{\n\t\"message\": \"%s\"\n}\n",
  "<message>%s</message>\n" };

static char *languages[] = {
  "german", "english", "spanish", "dutch" };
static char *formats[] = {
  "text", "html", "json", "xml" };

static ssize_t driver_read( struct file *instance, char __user *user,
size_t count, loff_t *offset ) {
  unsigned long not_copied, to_copy;
  char *format;
  char message[256];
  unsigned int minor = iminor( file_inode(instance) );
  format = formatstring[minor%4];
  snprintf( message, sizeof(message), format, messages[(minor>>2)%4]);
  if (*offset >= strlen(message)+1)
    return 0;
  to_copy = min( count, strlen(message+*offset)+1);
  not_copied = copy_to_user( user, message+*offset, to_copy );
  *offset = *offset + (to_copy - not_copied);
  return to_copy - not_copied;
}

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

static int __init hello_i18l_module_init(void) {
  int i;
  // Gerätenummern zuteilen lassen
  if (alloc_chrdev_region(&dev_num, 0, NUM_DEVICES, "hello_i18n") < 0) {
```

resse der initialisierten Datenstruktur weist der Treiber einem zuvor definierten Objekt vom Typ `struct cdev` zu. Er ergänzt es noch um -die Gerätenummer, also um die Kombination von Major und Minor Number, und übergibt es schließlich dem Kernel durch einen Aufruf von `cdev_add()`. Damit wird aus dem Kernel-Modul ein im Kernel registrierter Treiber für zunächst ein Gerät 2.

Machen lassen

Der Treiber ist damit im System, aber die Gerätedatei für den Zugriff fehlt noch. Kennt man die Gerätenummer, etwa durch Lesen der Datei `/proc/devices`, lässt sich die Gerätedatei wie in der Unix-Steinzeit über den Befehl `mknod` anlegen. Eleganter und zeitgemäß legt stattdessen der Kernel die Datei selbst an.

An dieser Stelle kommt das Sys-Filesystem ins Spiel. Darin legt der Gerätetreiber einen Eintrag an, über den er die Gerätenummer bekanntgibt. Zuvor erzeugt er jedoch der Ordnung halber typischerweise im Sys-Filesystem eine eigene Geräteklasse mit den relevanten Informationen (Gerätename, Major Number, Minor Number). Ein Rechenprozess, ein Daemon, überwacht das Sys-Filesystem ständig, erkennt den neuen Eintrag und legt daraufhin die Gerätedatei an.

Diesen Daemon gibt es in unterschiedlichen Ausprägungen. Auf Standardsystemen werkelt das komplex konfigurierbare Udevd, das Systemd startet und dessen Konfiguration sich im Verzeichnis `/etc/udev/` befindet. Im Embedded-Bereich trifft man in der Regel auf das schlankere Mdev. Allerdings kommt Linux auch ohne Mdev und Udevd zurecht, denn der Kernel-Thread `kdevtmpfs` legt unabhängig von einer Unterstützung im Userland die Gerätedatei an.

Alle Geräte einzeln

Zum Anlegen der Geräteklasse dient im Kernel die Funktion `class_create()`, zum Anlegen der Geräteeinträge `device_create()`. Letzterer gibt man neben der Adresse des Klassenobjekts die Gerätenummer sowie den Namen des Geräts mit. Die Gerätenummer lässt sich über das Makro `MKDEV()` aus Major und Minor Number zusammenbauen. Der

Gerätename entsteht elegant über einen Formatstring à la `sprintf()` und die optionalen Parameter. Die Einträge für die Gerätedatei werden übrigens beim Entladen des Kernel-Moduls mit `device_destroy()` entfernt, der Klasseneintrag per `class_destroy()`.

Für einen funktionstüchtigen Treiber gilt es, daneben noch Zugriffsfunktionen wie `driver_read()` oder `driver_write()` zu implementieren. Sofern auch Initialisierungen anfallen oder ein besonderer Zugriffsschutz realisiert werden muss, braucht es daneben noch `driver_open()` sowie `driver_close()`.

Transfer kodieren

Die Treiberfunktion `driver_read()` kopiert mittels der Funktion `copy_to_user()` Daten aus dem Adressraum des Kernels (Kernel Space) in den der Applikation (User Space). Gemäß dem Schema „wohin, woher, wie viel" gibt man zuerst die Zieladresse in der Applikation an, danach die Adresse im Kernel.

Gibt die Funktion eine Null zurück, sind alle Daten erfolgreich kopiert. Ansonsten entspricht der Rückgabewert der Anzahl von Bytes, die nicht kopiert werden konnten. Das kommt vor, wenn die Applikation absichtlich oder unabsichtlich dem Kernel vorbehaltene Adressen nutzt. Zudem ist sicherzustellen, dass die Funktion nicht mehr Daten kopiert, als Platz im Kernel respektive im User Space dafür verfügbar ist. Das wird mittels Minimum-Funktion abgesichert.

Zu guter Letzt addiert der Treiber die Anzahl der transferierten Bytes auf einen Offset-Zähler. Dieser Wert lässt sich nutzen, um festzustellen, ab welchem Index eine Applikation Daten lesen möchte beziehungsweise ob sie bereits alle Daten konsumiert hat. Für die Applikation ist das Lesen beendet, wenn die Funktion eine Null liefert. Ansonsten gibt `driver_read()` die Anzahl der erfolgreich in den Userspace kopierten Bytes zurück.

Kreative Namenswahl

Das vorgestellte Szenario eines Treibers berücksichtigt noch nicht die vielfältigen Möglichkeiten, die sich durch die Verknüpfung des Treibers mit mehreren Gerätedateien ergibt. Immerhin stehen

Listing 1: Treiber für mehrere Geräte (hello-i18n.c) (Fortsetzung von S. XX)

```
    pr_err("Failed to allocate device numbers\n");
    return -ENOMEM;
  }
  // Erstellen der Geräteklasse
  hello_i18n_class = class_create(THIS_MODULE, "hello_i18n");
  if (IS_ERR(hello_i18n_class)) {
    pr_err("class_create failed\n");
    unregister_chrdev_region(dev_num, NUM_DEVICES);
    return PTR_ERR(hello_i18n_class);
  }
  for (i = 0; i < NUM_DEVICES; ++i) {
    cdev_init(&hello_cdev_i18n[i], &hello_fops);
    hello_cdev_i18n[i].owner = THIS_MODULE;
    if (cdev_add(&hello_cdev_i18n[i],MKDEV(MAJOR(dev_num),i),1)<0) {
      pr_err("cdev_add failed\n");
      goto device_cleanup;
    }
    // Erstellung der Gerätedatei initiieren
    device_create(hello_i18n_class, NULL, MKDEV(MAJOR(dev_num), i),
      NULL, "hello-%s-%s",
      languages[(i>>2)%4],
      formats[i%4] );
  }
  pr_info("Module loaded successfully\n");
  return 0;
device_cleanup:
  for (i--; i >= 0; i--) {
    device_destroy(hello_i18n_class, MKDEV(MAJOR(dev_num), i));
    cdev_del(&hello_cdev_i18n[i]);
  }
  class_destroy(hello_i18n_class);
  unregister_chrdev_region(dev_num, NUM_DEVICES);
  return -ENOMEM;
}

static void __exit hello_i18l_module_exit(void) {
  int i;
  for (i = 0; i < NUM_DEVICES; ++i) {
      device_destroy(hello_i18n_class, MKDEV(MAJOR(dev_num), i));
  }
  class_destroy(hello_i18n_class);
  for (i = 0; i < NUM_DEVICES; ++i) {
      cdev_del(&hello_cdev_i18n[i]);
  }
  unregister_chrdev_region(dev_num, NUM_DEVICES);
  pr_info("Module unloaded successfully\n");
}

module_init(hello_i18l_module_init);
module_exit(hello_i18l_module_exit);
MODULE_LICENSE("GPL");
```

hier 20 Bits zur Verfügung, die es einem Treiber ermöglichen, 2^{20} Geräte zu bedienen, also gut eine Million. Alternativ lassen sich die 20 Bits nutzen, um Funktionalitäten bitweise zu kodieren.

So wäre ein Treiber zur Temperaturmessung denkbar, der die Temperatur abhängig von der Minor Number entweder in Kelvin, Grad Celsius oder Fahrenheit zurückgibt. Bei einem Treiber für eine serielle Schnittstelle könnte man drei Bits der Minor Number nutzen, um damit eine von acht Übertragungsgeschwindigkeiten auszuwählen. Über je ein weiteres Bit ließen sich die Parität und Anzahl der Datenbits auswählen.

Wir implementieren als Beispiel einen internationalen Hello-World-Treiber, der sich über die Minor Number bezüglich der Ausgabesprache (Deutsch, Englisch, Spanisch, Niederländisch) und des Ausgabeformats (Text, HTML, JSON, XML) konfigurieren lässt. Zunächst müssen wir bei der Modulinitialisierung die zugehörigen Device-Nummern reservieren. Für die Unterstützung von vier Sprachen reichen zwei Bits aus, weitere zwei Bits decken die vier Ausgabeformate ab.

Dazu brauchen wir 16 Minor Numbers 3. Zu jeder davon gehört eine Gerätedatei mit einem möglichst sprechenden Namen. Deshalb definiert der Quellcode zwei Felder für die Namen der unterstützten Sprachen (`languages`) sowie die Formate (`formats`). Zudem benötigen wir für die eigentliche Ausgabe die Nachricht in der passenden Sprache (Variable `messages`) und einen Format-String (Variable `formatstring`), um die Nachricht im Wunschformat ausgeben zu können.

Bits schieben

Der Name der Gerätedatei entsteht mit etwas Modulo-Trickserei. Die Modulo-Funktion gibt den Rest zurück, der bei einer ganzzahligen Division übrig bleibt. Die unteren beiden Bits, also vier Varianten, definieren das Format. Daher lässt sich per Modulo 4 die Auswahl bezüglich des Formatnamens (Variable `formats`) treffen. Die nächsten zwei Bits spezifizieren die Sprache. Wir schieben unsere Minor Number um zwei Bits nach rechts, um dann erneut den Modulo-Trick anwenden zu können und schließlich den Namen der Sprache zu erhalten.

Beim Aufräumen ist übrigens Vorsicht geboten. Schlägt aus welchen Gründen auch immer das Anlegen der Gerätedatei fehl, gilt es, die bis dahin erfolgreich absolvierten Schritte wieder rückgängig zu machen – aber eben auch nur diese. Deswegen wird beim `device_cleanup` die Schleife mit dem Aufruf von `device_destroy()` vom aktuellen Index an rückwärts durchlaufen.

Den Trick mit der Modulo-Funktion wenden wir auch bei `driver_read()` an,

Listing 2: Makefile

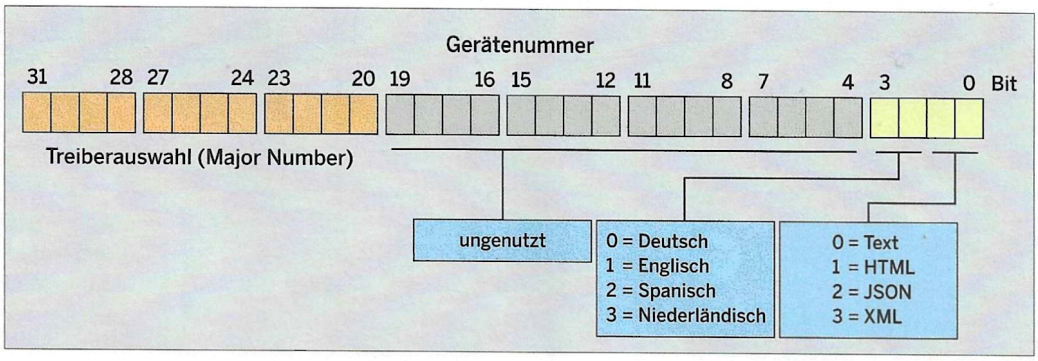
```
ifneq ($(KERNELRELEASE),)
obj-m := hello-i18n.o
else
KDIR  := /lib/modules/$(shell uname -r)/build
#KDIR := /home/quade/embedded/raspi/linux/
PWD   := $(shell pwd)
default:
  $(MAKE) -C $(KDIR) M=$(PWD) modules
endif

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

Listing 3: Linux

```
[...]
struct _instance_data {
  char *message;
  char *format;
};

static int driver_open(struct inode *instance,
struct file *file)
{
  unsigned int minor;
  struct _instance_data *id;

  id = kmalloc( sizeof(struct _instance_data), GFP_
USER );
  if (id==NULL)
    return -ENOMEM;

  minor = iminor( instance );
  id->format = formatstring[minor%4];
  id->message = messages[(minor>>2)%4];
  file->private_data = (void *)id;
  return 0;
}

static int driver_close(struct inode *instance,
struct file *file )
{
  kfree( file->private_data );
  return 0;
}
[...]
```

um die eigentliche Ausgabe zu erzeugen. Dazu werden der Formatstring und die zugehörige Nachricht ausgewählt und beides per `snprintf()` miteinander zur eigentlichen Ausgabe verquickt.

Den Quellcode `hello-i18n.c` aus Listing 1 übersetzen Sie mit einem Make-Aufruf mithilfe des Makefiles aus Listing 2 zum Kernel-Modul, das Sie dann per `insmod hello-i18n.ko` in den Kernel laden. Dank `kdevtmpfs` entstehen dabei die Gerätedateien im Verzeichnis `/dev`. Per `cat /dev/hello-german-json` etwa lässt sich die Nachricht in Deutsch und im JSON-Format auslesen 4. Allerdings müssen Sie dazu Root-Rechte besitzen.



3 **Die Funktionen des Treibers sind in der Minor Number kodiert.**

Komplexe Realitäten

In der vorgestellten Implementierung extrahiert bei jedem Aufruf der Funktion `driver_read()` ein Aufruf des Makros `iminor()` die Minor Number. Sie ist ein Attribut des Inodes, also der Datenstruktur, die alle Infos zur Datei enthält. Ein Zeiger auf den Inode wiederum befindet sich in der `struct file` und lässt sich mit überschaubarem Rechenaufwand über das Makro `file_inode()` ermitteln.

Das wahre Treiber-Leben gestaltet sich aber oft komplexer. Darum nimmt man die stets gleichen Vorgänge üblicherweise einmalig beim Öffnen der Gerätedatei vor und verknüpft das Ergebnis mit der zugreifenden Applikationsinstanz. Letztere repräsentiert bei `driver_read()` der Parameter vom Typ `struct file`, den der Kernel für jede Instanz gesondert anlegt. Linus Torvalds hat dieser Datenstruktur das Attribut `private_data` spendiert, an das man praktischerweise instanzenspezifische Parameter anhängen kann.

Optimierung

Folglich deklariert ein Treiber traditionellerweise eine Datenstruktur, die alle instanzenspezifischen Parameter aufnimmt. Innerhalb von `driver_open()` wird per `kmalloc()` Speicher für die Daten reserviert und mit den benötigten Inhalten gefüllt. Innerhalb der Zugriffsfunktionen `driver_read()` oder auch `driver_write()` kann man dann direkt auf diese Struktur zugreifen. Bei einem `driver_close()` gibt ein Aufruf von `kfree()` sie wieder frei (Listing 3).

Das alles ergibt einen funktionierenden ersten Gerätetreiber, der mit mehreren Gerätedateien zurechtkommt und sich gut als Basis für weitere Entwicklungen eignet. Insbesondere der Zugriff auf Hardware fehlt hier bislang.

Noch eine Anmerkung zum Schluss: Im Beispiel haben wir die Datenstruktur `struct cdev` statisch im Treiber reserviert. Es gibt eine alternative dynamische Variante, bei der ein Aufruf von `cdev_alloc()` das Objekt reserviert und initialisiert. Zwar fällt dann ein `cdev_init()` flach, aber es wird eine Fehlerbehandlung mehr nötig. Das macht die Sache also sicher nicht übersichtlicher. (*jlu*)

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/50222

Weitere Infos und interessante Links
www.lm-online.de/qr/50222

```
quade@raspi:~/linux-magazin $ make
make    -C /lib/modules/6.1.0-rpi7-rpi-v8/build M=/home/quade/linux-magazin modules
make[1]: Entering directory '/usr/src/linux-headers-6.1.0-rpi7-rpi-v8'
  CC [M]  /home/quade/linux-magazin/hello-i18n.o
  MODPOST /home/quade/linux-magazin/Module.symvers
  CC [M]  /home/quade/linux-magazin/hello-i18n.mod.o
  LD [M]  /home/quade/linux-magazin/hello-i18n.ko
make[1]: Leaving directory '/usr/src/linux-headers-6.1.0-rpi7-rpi-v8'
quade@raspi:~/linux-magazin $ sudo insmod hello-i18n.ko
quade@raspi:~/linux-magazin $ ls /dev/hello*
/dev/hello-dutch-html    /dev/hello-english-text  /dev/hello-spanish-html
/dev/hello-dutch-json    /dev/hello-english-xml   /dev/hello-spanish-json
/dev/hello-dutch-text    /dev/hello-german-html   /dev/hello-spanish-text
/dev/hello-dutch-xml     /dev/hello-german-json   /dev/hello-spanish-xml
/dev/hello-english-html  /dev/hello-german-text
/dev/hello-english-json  /dev/hello-german-xml
quade@raspi:~/linux-magazin $ ls -l /dev/hello-german-*
crw------- 1 root root 236, 1 Dec 30 18:03 /dev/hello-german-html
crw------- 1 root root 236, 2 Dec 30 18:03 /dev/hello-german-json
crw------- 1 root root 236, 0 Dec 30 18:03 /dev/hello-german-text
crw------- 1 root root 236, 3 Dec 30 18:03 /dev/hello-german-xml
quade@raspi:~/linux-magazin $ sudo cat /dev/hello-german-json
{
        "message": "Hallo, Welt"
}
quade@raspi:~/linux-magazin $ sudo cat /dev/hello-dutch-text
Hallo, wereld
quade@raspi:~/linux-magazin $ sudo cat /dev/hello-english-html
<html>
<h1>Hello, world</h1>
</html>
quade@raspi:~/linux-magazin $
```

4 **Die vielen Gerätedateien gehören allesamt zum selben Treiber.**