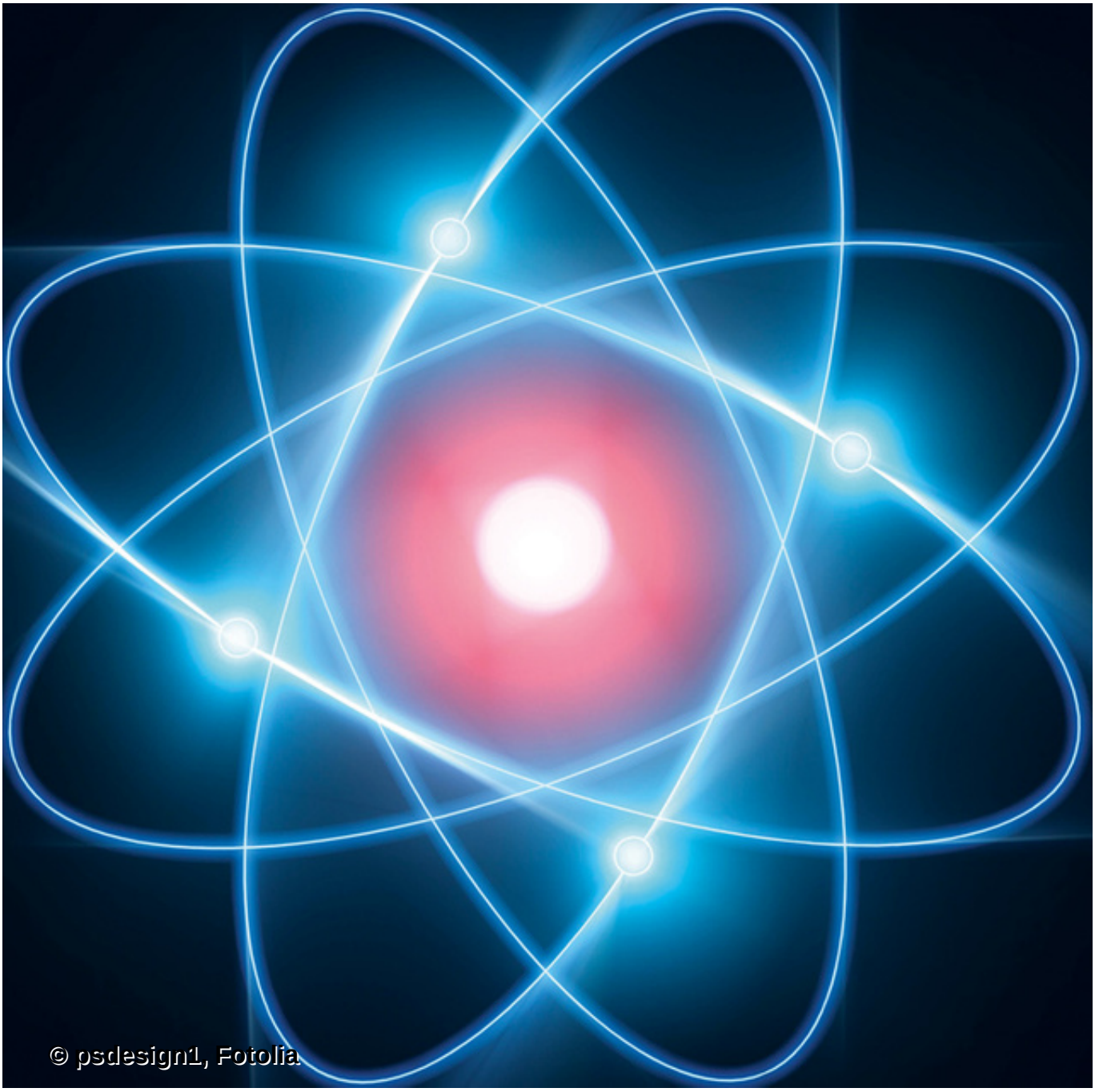
Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 93

Kern-Technik

In Device Trees lagern Hardware-Spezifikationen, damit die Treiber beim Booten erfahren, worum sie sich kümmern müssen. Wegen der Aufsteckboards für Raspberry und Co. legen die Entwickler nun dynamische Device Trees nach, die lange nach dem Booten Peripherie-Neuankömmlinge willkommen heißen.
Eva-Katharina Kunst, *Jürgen Quade*



Viele der rund 200 Chamäleon-Arten auf der Welt wechseln ihre Hautfarben und -muster passend zur Umwelt. Meist geht es den "Erdlöwen" nicht in erster Linie um Tarnung,

sondern um Balz oder Verteidigung sowie eine nützliche Reaktion auf sich ändernde Temperaturen, Sonneneinstrahlung, Tageszeiten oder Luftfeuchtigkeit.

Auch dem Linux-Kernel haben seine Entwickler auf Basis der Device Trees eine Art Chamäleon-Modus eingepflanzt, der dem Betriebssystem hilft sich an spontan geänderte Hardware-Gegebenheiten anzupassen, also an Peripherie-Modifikationen, die lange nach dem Booten passieren. Anders als die Chamäleons, die nach dem Washingtoner Artenschutz-Übereinkommen als Lebensraum-gefährdet eingestuft sind, entwickeln sich Device Trees in Linux sehr gedeihlich und sollen darum in dieser Kern-Technik-Folge ihr "Bäumchen, wechsle dich!" vorführen.

Dem guten Entwicklergrundsatz gemäß, Daten von Code zu trennen, beschreiben Device Trees Daten (die Hardware), die Code im Gerätetreiber verarbeitet. Der Device Tree spezifiziert, über welche Adresse die serielle Schnittstelle oder der I2C-Bus zu finden ist, an welche GPIOs der Temperatursensor seine Daten liefert und welchen Interrupt die jeweilige Peripherie verwendet.

Anhand dieser Beschreibung lädt Linux zum einen beim Booten die zur Hardware passenden Gerätetreiber. Zum anderen greifen die Gerätetreiber ihrerseits beim Kernel die im Device Tree hinterlegten Informationen – im Wesentlichen Hardware-Adressen – ab, um mit der Peripherie arbeiten zu können.

Device Trees hatte die Kern-Technik-Reihe zwar bereits 2013 vorgestellt [1], in den vier verstrichenen Jahren ist die Entwicklung auf diesem Gebiet aber mutig vorangeschritten. So sind Device Trees seit Kernel 4.4 zum Must have geworden. Hinzu kamen wichtige Erweiterungen: Dynamische Device Trees und Overlays. (Letztere haben außer ihrem Namen nichts mit dem Overlay-Filesystem der vorigen Kern-Technik gemeinsam.)

Mit den Neuerungen trägt Linux dem Umstand Rechnung, dass sich immer mehr Hardware erst nach dem Bootzeitpunkt zu erkennen gibt. Als Beispiel sind standardisierte Erweiterungsboards – so genannte Capes oder Hats – für Raspberry Pi, Wandboard oder Beagleboard zu nennen, deren Ein- und Ausgabepins – die GPIOs – der Kernel konfigurieren soll, wie es nötig wird.

Für so viel Dynamik sind Device Trees der ersten Generation zu statisch und der Aufwand sie zu erstellen, passende auszuwählen und zu pflegen zu groß. Folglich haben die Entwickler gezaubert und Device Tree Overlays aus dem Hut gezogen. Ihr modularer Ansatz beschreibt die Basishardware (zum Beispiel Raspberry Pi Model 2) und die optionalen Komponenten separat. Beim Booten entsteht aus den Beschreibungen der Device Tree, den der Kernel übergeben bekommt.

So zeigt Listing 1 beispielsweise ein Overlay, das die GPIOs 7 und 8 des Raspberry Pi mit Pull-down- und der GPIO 9 mit Pull-up-Widerständen versieht; es aktiviert also eine Eigenschaft, die in der Hardware des Raspberry Pi verborgen ist.

Listing 1: Raspberry-Pi-Overlay zum Aktivieren von Pull-up- und Pull-down-Widerständen

```
01 /dts-v1/;
02 /plugin/;
03
04 / {
05     compatible = "brcm,bcm2708", "brcm,bcm2709";
```

```
06
07      fragment@0 {
08          target = <&gpio>;
09          __overlay__ {
10              pinctrl-names = "default";
11              pinctrl-0 = <&my_pins>;
12
13              my_pins: my_pins {
14                  brcm,pins = <7 8 9>;      /* gpio no. */
15                  brcm,function = <0 0 0>; /* 0:in, 1:out */
16                  brcm,pull = <1 1 2>;      /* 2:up 1:down 0:none
17              };
18          };
19      };
20 };
```

Baumschule

Wer einen Device Tree pflanzen will, tut dies in einer Datei mit der Erweiterung ».dts« (Device Tree Source). Ein Compiler übersetzt sie danach in einen so genannten Device Tree Blob – also ein binäres Datenobjekt (Abbildung 1). Technisch ist ein Device Tree ein Baum, der aus Knoten (Nodes) besteht. Mit einem Klammerpaar »{}« eingefasste Attribute (Properties) beschreiben einen Knoten. Die Spezifikation von Kindknoten erfolgt ebenso innerhalb der Beschreibung des Elternknotens. So entsteht das hierarchische Verhältnis innerhalb des Baumes.

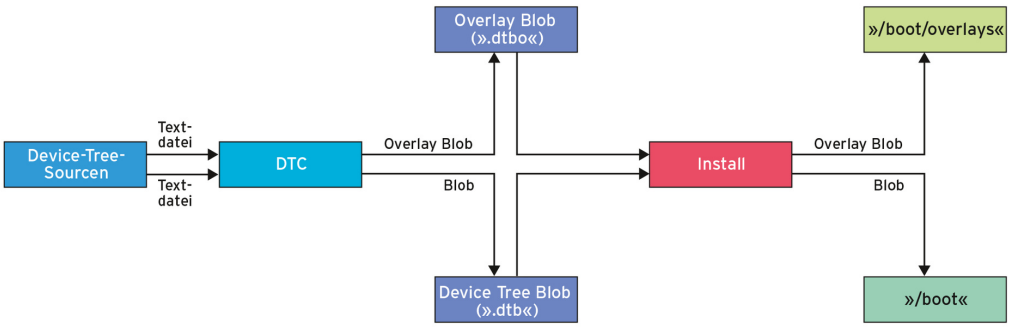


Abbildung 1: Aus den Quelldateien erzeugt der Device Tree Compiler binäre Datenobjekte (Blobs).

Der Name eines Knotens, im Beispiel »my_pins«, steht vor der geschweiften Klammer. Die den Knoten beschreibenden Attribute bekommen einen Wert zugewiesen (Key-Value-Paar). Als Wert erlaubt sind Strings, Zahlen oder Byte-Felder. Strings sind in Hochkommata gesetzt, Zahlenwerte erkennt der Device Tree Compiler an den spitzen Klammern. Eckige Klammern umzäunen die Byte-Felder. Attribute dürfen auch mehrere Werte zugewiesen bekommen.

Bei Overlays, also einem modularen Device Tree, dessen Beschreibung in separaten Dateien erfolgt, ordnen Linuxer Knoten über das Attribut mit dem vorgegebenen Schlüsselwort »target« direkt einem Elternknoten zu. Das erspart ihnen die Baumstruktur anzugeben. Ein Subknoten mit dem festen Namen »__overlay__« spezifiziert zugehörige Attribute und mögliche Kindknoten.

Beziehungen zwischen Knoten entstehen über Referenzen. Technisch nummeriert man dazu die Knoten mit einem 32-Bit-Wert durch. Dieser Zählwert heißt Phandle. Ist der

Phandle beim Schreiben einer Device-Tree-Beschreibung nicht bekannt, erfolgt die Referenzierung durch Angabe des symbolischen Namens mit vorgestelltem Kaufmanns-Und, in Listing 1: »<&my_pins>«.

Zentral ist das Attribut »compatible«. Es stellt die Verbindung zwischen Knoten und seinem Treiber her, indem ein zugehöriger Treiber den gleichen Compatible-String spezifiziert. Die Syntax dafür ist recht komplex und [2] zu entnehmen.

Bäumchen, wechsle dich

Mit Hilfe des Device Tree Compilers erzeugen Linuxer aus einer Device-Tree-Beschreibung den Blob (Datei-Erweiterung ».dtbo«). Dank dynamischer Device Trees und dem Kommando »dtoverlay« ergänzt der Blob den vorhandenen Device Tree oder überschreiben ihn.

Kleinere Änderungen am Device Tree gelingen auf einer ARM-Plattform mit »dtparam«. Das Kommando auf der Konsole ohne Parameter listet Attribute auf, die sich für Modifikationen per »dtparam« prinzipiell empfänglich zeigen. Insbesondere lassen sich damit Komponenten wie zum Beispiel Audiohardware in Betrieb setzen oder deaktivieren.

Standard-x86-Distributionen haben die Kommandos »dtparam« oder »dtoverlay« übrigens nicht an Bord, weil sich für die vergleichsweise uniforme PC-Welt Device Trees (noch) nicht lohnen. Besitzer eines Raspberry Pi dagegen fügen Device Tree Overlays mit der Bootloader-Datei »/boot/config.txt« hinzu oder entfernen sie aus dem System (Abbildung 2).

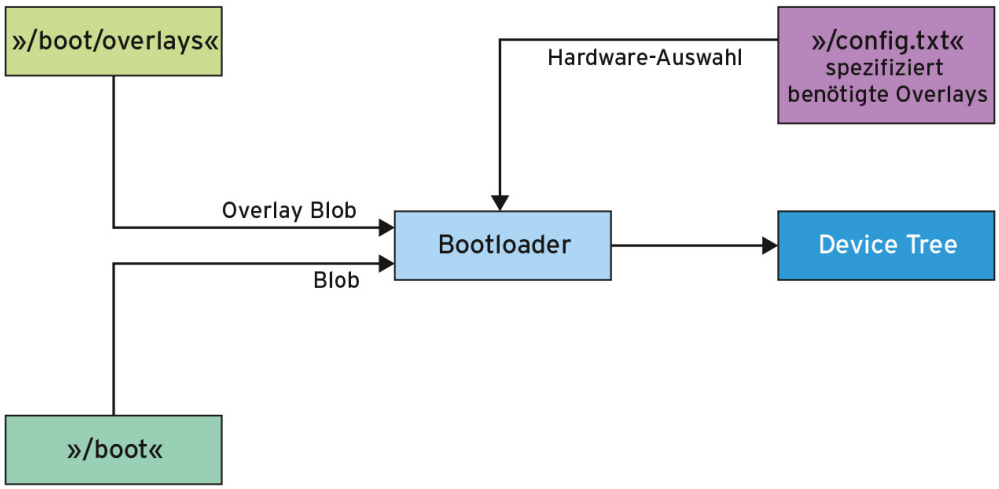


Abbildung 2: Der Bootloader baut aus den Blobs den Baum zusammen.

Um ein Overlay zu etablieren, kopiert Root den zugehörigen Device Tree Blob in das Verzeichnis »/boot/overlays/«, wenn er nicht bereits dort liegt. In der »/boot/config.txt« ergänzt er dann:

```
dtoverlay=Name-der-Blobdatei-ohne-Dateierweiterung
```

Sind innerhalb des im Blob spezifizierten Knotens Attribute anzupassen, folgen »dtparam«-Kommandos. Wer nur einzelne Anpassungen an einem Blob vorzunehmen gedenkt, lässt anstelle der separaten »dtparam«-Kommandos per

```
dtoverlay=w1-gpio,gpiopin=4
dtoverlay=w1-gpio-2,gpiopin=24
```

die zu modifizierenden Attribute gleich dem Kommando »dtoverlay« folgen.

Schnittstelle zum Treiber

Hardware beschreiben ist die eine Seite, sie in einem Gerätetreiber auszulesen und auszuwerten die andere. Glücklicherweise unterstützt der Linux-Kernel den Programmierer auch dabei, denn er repräsentiert jedes Gerät intern über ein Geräteobjekt, die Datenstruktur »struct device«. Das Objekt wiederum gewährt Zugriff auf den zugehörigen Eintrag im Geräte-Baum – falls der Kernel die Beziehung zwischen Geräteobjekt und Device Tree selbstständig herstellen kann.

Völlig unabhängig vom Erfolg der Vorarbeit des Kernels kann der Programmierer die Funktion »of_find_node_by_path(char *nodename)« verwenden. Unter Angabe des Knotennamens bekommt er den Zeiger auf die zugehörige Datenstruktur »struct device_node« zurückgegeben. Auf die Elemente im Knoten wiederum kann er per »of_get_property()« unter Angabe des Attribut-Namens zugreifen.

Sollten sich hinter einem Attribut 16- oder 32-Bit-Werte verstecken, sind diese im Big-Endian-Format abgelegt. Funktionen wie »be16_to_cpup()«, »be32_to_cpup()« oder »be64_to_cpup()« überführen die Zahlenwerte dann in das Datenablageformat des verarbeitenden Rechners. Die Funktionen bekommen einen Pointer übergeben und liefern den Wert mit der jeweiligen Bitbreite im so genannten CPU-Format zurück.

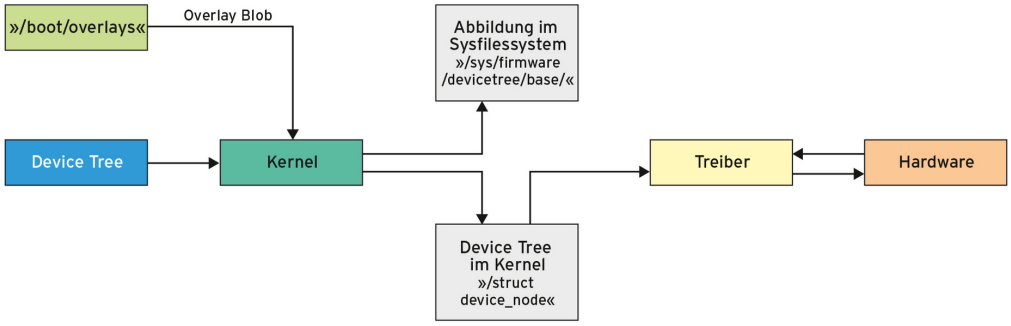


Abbildung 3: Der Kernel lässt auch nach dem Booten Änderungen am Device Tree zu.

Wie bereits angedeutet, kann der Treiberprogrammierer dem Kernel sogar die Chance geben, den zu einem Gerät gehörenden Treiber automatisch zu laden. Dazu erzeugt er eine so genannte Matchtabelle, die einen der Strings enthält, die im Device Tree unter dem Attribut »compatible« abgelegt sind. Außerdem ruft er im Quellcode des Gerätetreibers das Makro »MODULE_DEVICE_TABLE()« auf:

```
[...]
static struct of_device_id
   linmag_match[] = {
   { .compatible = "linmag" },
   {}
};
MODULE_DEVICE_TABLE( of, linmag_match );
```

Grob betrachtet trägt das Makro String und Treibernamen in die zum jeweiligen Kernel gehörende Datei »modules.aliases« ein. In ihr schlägt der Kernel bei einer anstehenden Geräte-Identifikation – für die ist er selbst nicht zuständig – den zugehörigen Treiber nach und lädt das Kernelmodul.

Eine Probe-Aufforstung

Um die Mechanismen einmal durchzuspielen, wird das folgende Bespiel ein Overlay erstellen und laden, ein Gerätetreiber wird die darin spezifizierten Attribute einlesen. Ziel ist es, das Attribut »string-property« eines Read-Aufrufs an eine Applikation weiterzureichen. Die zugehörige Device-Tree-Quellcodedatei »linmag.dts« ist in Listing 2 abgedruckt. Zur Veranschaulichung definiert der Code neben der String Property weitere Attribute und den Kindknoten »foo«.

Listing 2: Overlay *linmag.dts*

```
01 /dts-v1/;
02 /plugin/;
03
04 / {
05    compatible = "linmag";
06
07    fragment@0 {
08       target = <&gpio>;
09       __overlay__ {
10          #address-cells = <1>;
11          #size-cells = <0>;
12          string-property = "Linux Magazin\n";
13          string-list-property = "Kern Technik", "93";
14          bytes = /bits/ 8 <0x45 0x76 0x61>;
15          u16 = /bits/ 16 <0x7377>;
16          u32 = /bits/ 32 <0x45657479>;
17          status = "on";
18          foo {
19               compatible = "linmag";
20               my_string = "Hello, World";
21               status = "on";
22          };
23      };
24    };
25 };
```

Um aus der Overlay-Datei den Blob zu generieren und zu laden, ruft der Programmierer auf dem Raspberry Pi den Compiler »dtc« und danach das Lade-Programm »dtoverlay« auf, die Details zeugt Abbildung 4. Dass der Compiler bei der Übersetzung eine Warnung ausspuckt, gibt keinen Anlass zur Sorge.

Abbildung 4: Das Sysfile-System spiegelt Änderungen am Device Tree wider.

Der Erfolg dieser Aktion lässt sich im Sys-Filesystem verifizieren (Abbildung 4), denn ein Baum eignet sich hervorragend zur Darstellung in einer Verzeichnis- und Dateistruktur wie dem Sys-FS. Der Einstiegspunkt dort ist das Verzeichnis »/sys/firmware/devicetree/base/«.

Listing 3: Gerätetreiber *string.c*

```
001 #include <linux/module.h>
002 #include <linux/fs.h>
003 #include <linux/cdev.h>
004 #include <linux/platform_device.h>
005 #include <linux/of_device.h>
006 #include <linux/of_platform.h>
007
008 static dev_t linuxmag_number;
009 static struct cdev *driver_object;
010 static struct class *linmag_class;
011 static struct platform_device_id pdi;
012
013 static struct file_operations fops;
014 static DECLARE_COMPLETION( dev_obj_is_free );
015
016 static int linuxmag_probe_device( struct platform_device *pde
017 {
018     struct device *dev = &pdev->dev;
019     struct device_node *nodeptr = dev->of_node;
020     const void *prop_value;
021     int size, i;
022     u16 value;
023     const __be16 *beptr;
024
025     pr_info("linuxmag_probe_device( %p, %p )\n", pdev, nodept
026     pr_info("pd->id: %d\n", pdev->id );
027     pr_info("pd->name: %s\n", pdev->name );
028     nodeptr = of_find_node_by_path("gpio");
029
030     prop_value = of_get_property( nodeptr, "string-property"
```

```
030     prop_value = of_get_property( nodeptr, "string-property",
031     printk("linuxmag_probe_device(): prop_value=%p, size: %d\
032         prop_value, size);
033     if (prop_value)
034         printk("prop_value: %s\n", (char *)prop_value);
035
036     prop_value = of_get_property( nodeptr, "bytes", &size );
037     printk("linuxmag_probe_device(): prop_value=%p, size: %d\
038         prop_value, size);
039     printk("byte array: [ ");
040     for ( i=0; i<size; i++ ) {
041         printk("%x ", *(char *)(prop_value+i));
042     }
043     printk(" ]\n");
044
045     beptr = of_get_property( nodeptr, "u16", &size );
046     printk("linuxmag_probe_device(): prop_value=%p, size: %d\
047         prop_value, size);
048     if (prop_value) {
049         value = be16_to_cpup( beptr );
050         printk("value: %x\n", value);
051     }
052     return 0;
053 }
054
055 static int linuxmag_remove_device( struct platform_device *pd
056 {
057     return 0;
058 }
059
060 static void linuxmag_release( struct device *dev )
061 {
062     complete( &dev_obj_is_free );
063 }
064
065 struct platform_device linuxmag = {
066     .name  = "linmag", /* driver identification */
067     .id    = 0,
068     .dev = {
069         .release = linuxmag_release,
070     }
071 };
072
073 static struct of_device_id linmag_match[] = {
074     { .compatible = "linmag" },
075     {}
076 };
077
078 static struct platform_driver mydriver = {
079     .probe = linuxmag_probe_device,
080     .remove = linuxmag_remove_device,
081     .driver = {
082         .name = "linmag_driver",
083         .of_match_table = linmag_match,
084     }
085 };
086 MODULE_DEVICE_TABLE( of, linmag_match );
087
088 static int __init mod_init(void)
```

```
088 static int __init mod_init(void)
089 {
090     pr_info("mod_init()\n");
091     strcpy( pdi.name, "linmag" );
092     mydriver.id_table = &pdi;
093     if (platform_driver_register(&mydriver)!=0) {
094         pr_err("driver_register failed\n");
095         return -EIO;
096     }
097     if (alloc_chrdev_region(&linuxmag_number,0,1,"linmag")<0)
098         return -EIO;
099     driver_object = cdev_alloc();
100     if (driver_object==NULL)
101         goto free_device_number;
102     driver_object->owner = THIS_MODULE;
103     driver_object->ops = &fops;
104     if (cdev_add(driver_object,linuxmag_number,1))
105         goto free_cdev;
106     linmag_class = class_create( THIS_MODULE, "linuxmag" );
107     if (IS_ERR(linmag_class)) {
108         printk("linuxmag: no udev support.\n");
109         goto free_cdev;
110     }
111     linuxmag.dev.devt = linuxmag_number;
112     platform_device_register( &linuxmag );
113     return 0;
114
115 free_cdev:
116     kobject_put( &driver_object->kobj );
117 free_device_number:
118     unregister_chrdev_region( linuxmag_number, 3 );
119     return -EIO;
120 }
121
122 static void __exit mod_exit(void)
123 {
124     pr_info("mod_exit()\n");
125     device_release_driver( &linuxmag.dev );
126     platform_device_unregister( &linuxmag );
127     class_destroy( linmag_class );
128     cdev_del( driver_object );
129     unregister_chrdev_region( linuxmag_number, 1 );
130     platform_driver_unregister(&mydriver);
131     wait_for_completion( &dev_obj_is_free );
132 }
133
134 module_init( mod_init );
135 module_exit( mod_exit );
136 MODULE_LICENSE("GPL");
```

Der vom Beispielcode modifizierte Knoten »gpio« befindet sich unterhalb des Knotens »soc« und präsentiert sich im Namen um die zugehörige Hardware-Adresse erweitert. Wer in das Verzeichnis wechselt, findet alle Attribute als Dateien (»string-property«, »u16« oder »bytes«), auf die er mit den Kommandos »cat« und »hd« (Hexdump) zugreifen kann. Der Kindknoten »foo« ist, wie zu erwarten war, als Unterverzeichnis ausgeprägt.

Nutzholz

Listing 3 zeigt einen Gerätetreiber, der die Attribute des Device Tree ausliest und auf einen virtuellen »Hello, World«-Beispieltreiber zurückgeht. Die modifizierte Variante gibt dagegen den Text aus, der im Device Tree im Attribut »string-property« abgelegt ist.

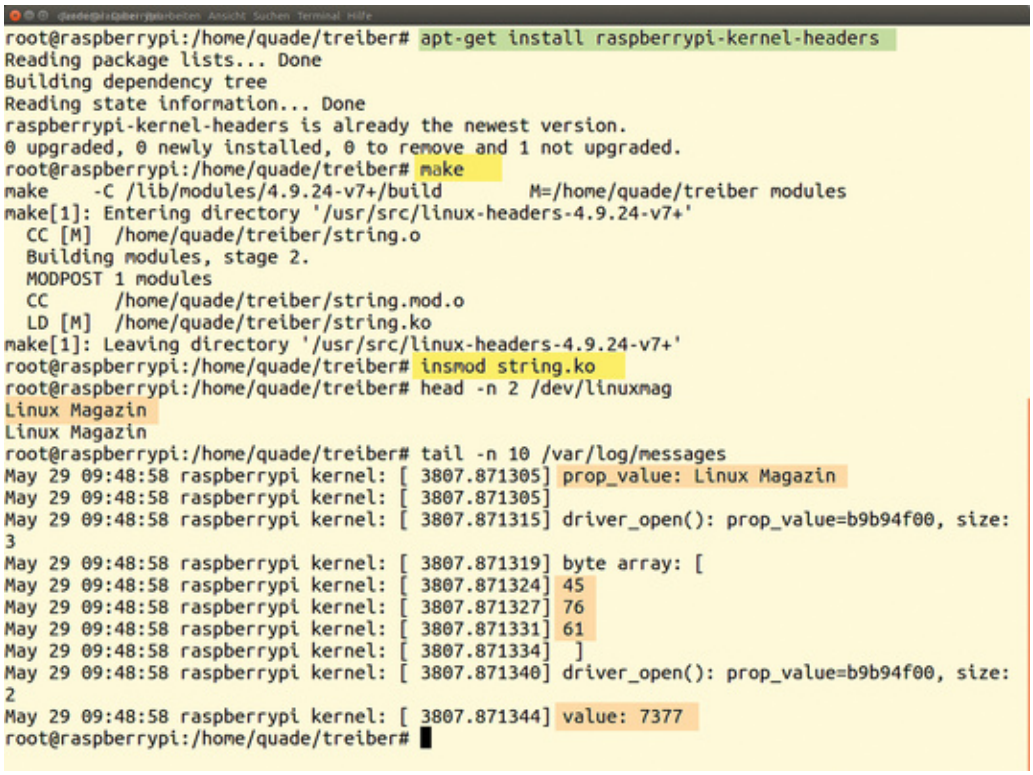


Abbildung 5: So geht's: Den Treiber erzeugen und ausprobieren.

Öffnet eine Applikation die zum Treiber gehörende Gerätedatei »/dev/linuxmag«, ruft Linux die Funktion »driver_open()« auf. Die sucht per »of_find_node_by_path()« in Zeile 28 den Knoten »gpio« und liest gleich danach mit »of_get_property()« die gesuchten Attribute aus. Während der Inhalt der Attribute »u16« und »bytes« nur im Syslog landet, kopiert die Funktion die String Property und übergibt den String auf Nachfrage (Funktion »driver_read()«) der Applikation. Wer den Treiber testen will, installiert die Kernelquellen

```
sudo apt-get install raspberrypi-kernel-headers
```

und generiert den Treiber (Abbildung 5). Dazu kann er das Makefile aus Listing 4 dienstverpflichten. »insmod string.ko« lädt den Neuen. Testen lässt er sich per:

```
sudo cat /dev/linuxmag
```

Wenn jetzt ein »Linux-Magazin« nach dem anderen über das Terminalfenster rauscht, ist eine Chamäleon-Art mehr im Linux-Biotop lebensfähig. *(jk)*

Listing 4: Makefile

```
01 obj-m   := string.o
02 KDIR    := /lib/modules/$(shell uname -r)/build
03 PWD     := $(shell pwd)
04
05 default:
06         $(MAKE) -C $(KDIR)      M=$(PWD) modules
```

Infos

1. Quade, Kunst, "Kern-Technik – Folge 68", Device Tree: Linux-Magazin 06/13, S. 76
2. Device-Tree-Referenz: [http://elinux.org/Device_Tree_Reference]
3. "Technical Note: About the Device Tree": [http://www.ofitselfso.com/BeagleNotes/AboutTheDeviceTree.pdf]

Die Autoren

Eva-Katharina Kunst ist seit den Anfängen von Linux Fan von Open Source. Jürgen Quade ist Professor an der Hochschule Niederrhein. Ihr gemeinsames Buch "Linux-Treiber entwickeln" ist Ende 2015 in vierter Auflage erschienen.