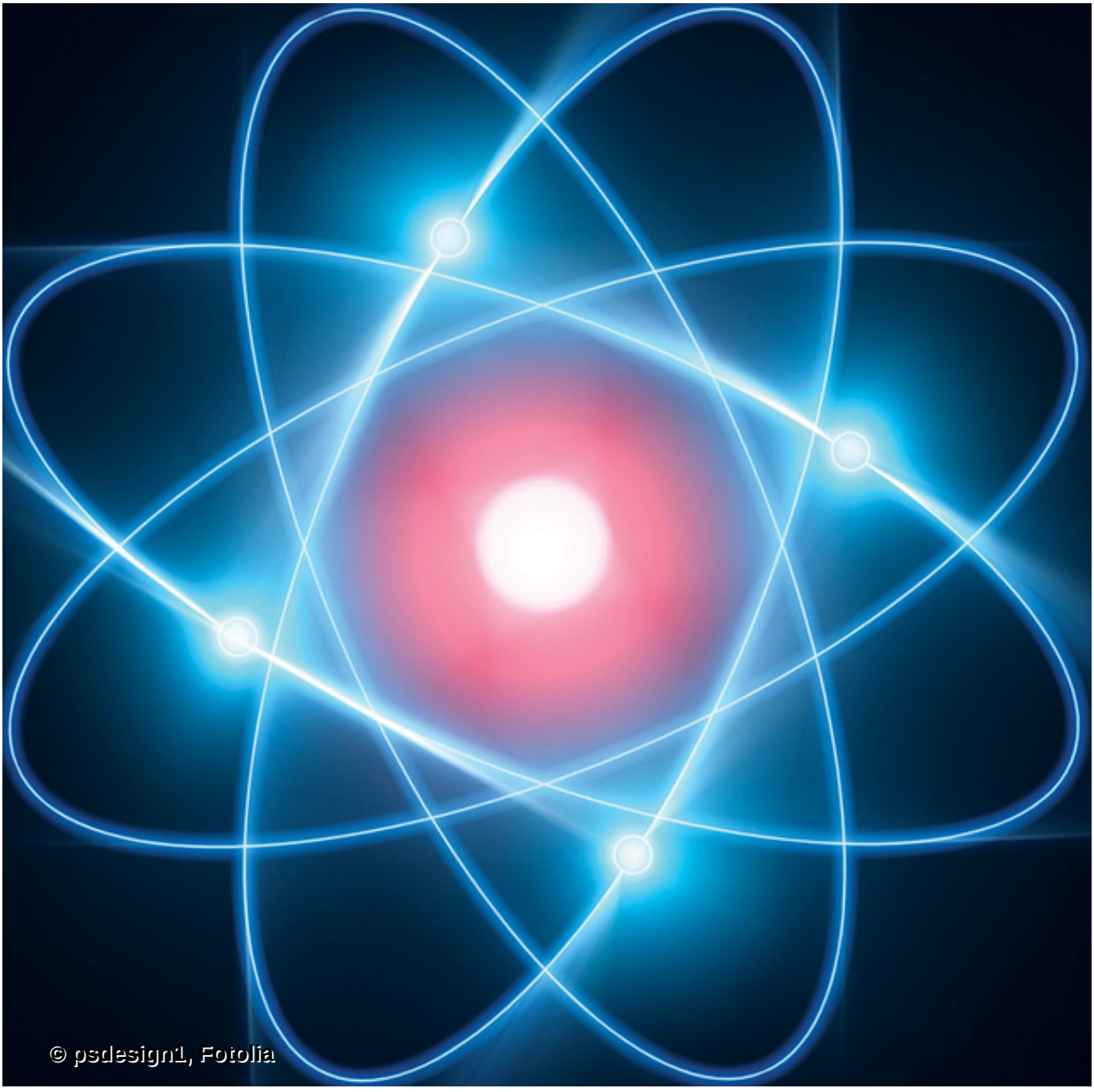
Kernel- und Treiberprogrammierung mit dem Linux-Kernel – Folge 68

Kern-Technik

Die wirklich große Vielfalt der ARM-Hardware stellt Kernelentwickler und Linux-Anwender vor große Herausforderungen. Device Trees und das neue Pincontrol-Subsystem sollen die Probleme lösen. *Jürgen Quade*, *Eva-Katharina Kunst*



Im Linux-Lager hat ARM den x86-Prozessoren längst den Rang abgelaufen – der Schwerpunkt dieser Linux-Magazin-Ausgabe widmet sich der Plattform ausführlich. Durch das Lizenzmodell der Firma, statt Chips die Baupläne dafür zu verkaufen, hat aber eine schwer überschaubare Vielfalt produziert: Statt Einheitsbrei nutzen Hersteller die Freiheit,

sich auf Basis eines Energie-effizienten ARM-Core mit herstellerspezifischer und bedarfsorientierter Peripherie von der Konkurrenz abzugrenzen. Diese Vielfalt spiegelt sich leider auch im Linux-Quellcode wider.

Jede ARM-Plattform bringt eigene Treiber und Module ein und bläht den Kernel-Quellcode zusehends auf. Während auf einem PC serielle Schnittstellen immer an den Adressen 0x3f8 und 0x2f8 zu finden sind, unterscheiden sich bei den ARM-SoCs Adressenlagen und zugehörige Interrupts regelmäßig, oft auch die Ansteuerlogik. Und: Eine in Hard- und Firmware realisierte Konfigurationsschnittstelle wie PCI hat sich in der ARM-Szene bisher ebenfalls nicht etabliert.

Ein aus dem Quellcode übersetzter Kernel ist deshalb nur auf einer spezifischen Plattform lauffähig. Das auf PCs gültige Linux-Motto "Compile once, run everywhere" ist auf dem Weg zu ARM verloren gegangen.

2010 zog Linus Torvalds die Reißleine und forderte die ARM-Entwickler auf, den Quellcode zu konsolidieren [1]. Kaum ein Jahr später präsentierten die Programmierer tatsächlich den ersten Kernel, der auf mehreren ARM-Plattformen lauffähig ist. Dabei griffen sie auf einen Trick der Entwickler der Linux-Version für Power-PC zurück: Die hatten die jeweilige Hardwarekonfiguration weder fest in den Kernel einkompiliert noch als ewig langen Bootparameter übergeben, sondern stellten sie vor dem Startup als im Hauptspeicher abgelegte Datenstruktur bereit. Diese Technik ist unter dem Namen Device Tree bekannt. Parallel dazu entwickelte die ARM-Community neue Subsysteme wie das des Pincontrol (siehe Kasten "Pincontrol-Subsystem")

Pincontrol-Subsystem

Die Pins moderner ARM-Prozessoren sind oft multifunktional ausgelegt: Je nach Konfiguration kann der Entwickler sie als normale Ein- und Ausgabeleitungen verwenden, die eine LED ansteuern oder einen Taster abfragen, oder sie als Daten- oder Taktleitungen von I2C-Interfaces (Inter-Integrated Circuit) oder SPI (Serial Peripheral Interface) beschalten.

Zudem gestatten die Chips die elektrische Eigenschaft eines Pins vorzugeben, beispielsweise lässt sich ein Pullup- oder Pulldown-Widerstand an den Eingang schalten oder die Stromstärke am Ausgang konfigurieren. Die Konfiguration sowie das Management dieser Pins übernimmt das mit Kernel 3.1 eingeführte Pincontrol-Subsystem, detailliert dokumentiert in den Kernelquellen [4].

Um die Pins zu verwalten, registriert der Entwickler zunächst sämtliche Pins. Dabei weist er jedem Pin eine Nummer und einen Namen zu (»PINCTRL_PIN(0, "A1")«). Danach darf er jeden konfigurieren. Außerdem gruppiert er Pins, beispielsweise jene, die zusammen ein I2C-Interface bilden (Pin-Multiplexing). Dass dieselben Pins zu mehreren Gruppen gehören, ist für das Subsystem unproblematisch – im Gegenteil, es verhindert sogar die versehentliche Doppelnutzung.

Blobs im Hauptspeicher

Eine Board-Firmware oder der Bootloader kopieren die Device Trees in den Hauptspeicher. (Theoretisch kann die Firmware Device Trees sogar erzeugen.) Folglich muss nicht nur der Kernel, sondern auch die sonstige Systemsoftware Device Trees unterstützen. Das ist gottlob der Fall: Egal ob Grub, Lilo oder "Das U-Boot" [2] – die

gängigen Bootloader sind schon längst auf die Technik umgestellt. Auch der proprietäre Bootloader der ebenso beliebten wie günstigen ARM-Einsteiger-Platine Raspberry Pi macht keine Ausnahme (Abbildung 1).

Jeder entsprechend konfigurierte Bootloader lädt den Device Tree Blob neben dem Kernel und dem Root-Filesystem in den Hauptspeicher und gibt danach die Kontrolle an Linux ab (siehe Kasten "Device Tree für den Raspberry Pi"). Alternativ lässt sich der Device Tree dem Kernel-Quellcode anhängen, was allerdings ganz und gar nicht mit der Idee eines einheitlichen Kernels für unterschiedliche Hardwareplattformen harmoniert.

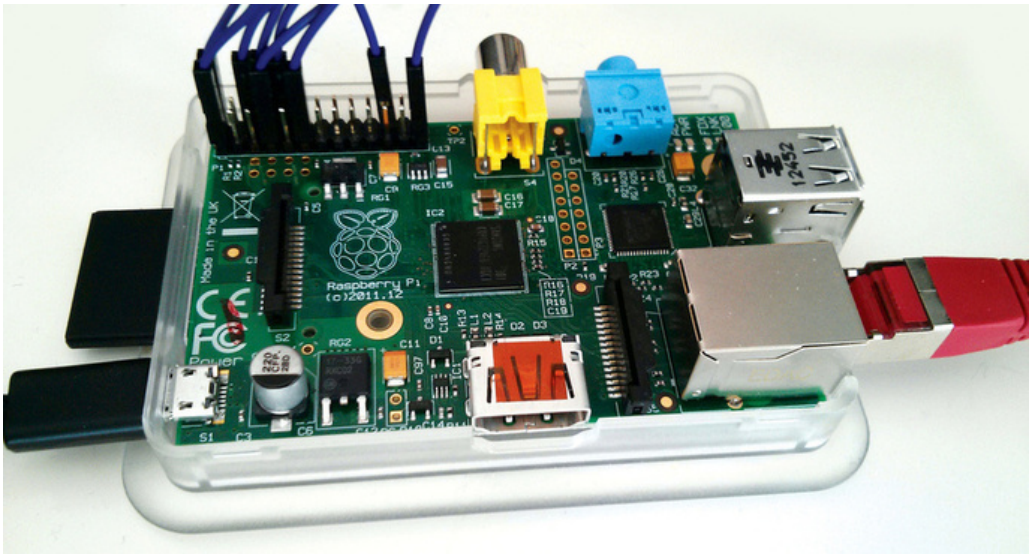


Abbildung 1: Die bestückte Raspberry-Pi-Platine als preiswertes Sprungbrett in die ARM-Technik.

Device Tree für den Raspberry Pi

So wie Raspbian in der Version »2013-02-09-wheezy-raspbian.img« [5] setzen viele Raspberry-Pi-Distributionen auf einen modifizierten Kernel der Version 3.6.y. Obwohl dieser Kernel bereits erste Anpassungen und Treiber für den vom Raspberry Pi verwendeten SoC BCM 2835 enthält, unterstützt er Device Trees nicht. Diese haben erst mit Kernel 3.7 Einzug gehalten, und auch dort erweist sich der Device Tree »bcm2835-rpi-b.dts« als rudimentär. Selbst der in Entwicklung befindliche Kernel 3.9 unterstützt den Raspberry Pi nur unvollständig, der Device Tree ist aber bereits umfangreicher gefüllt (Listing 1 und Abbildung 3).

Ohne Patches lässt sich derzeit auch aus dem aktuellen Standard-Linux-Quellcode kein vollständig lauffähiger Kernel für den Raspberry Pi generieren. Vor allem fehlt umfassender Support für USB und Grafik (Videocore). Hinweise, wie ein Device-Tree-fähiger Kernel zu bauen ist, gibt [6]. Das Hauptproblem bleibt, geeignet gepatchte Kernelquellen inklusive der zugehörigen Kernelkonfiguration zu finden.

Das sollte sich in naher Zukunft bessern. Die zum Raspberry Pi gehörende Default-Kernelkonfiguration liegt in vielen Kernelquellen im Verzeichnis »arch/arm/configs/«. In dem Namen tauchen »bcm2835« oder »rpi« auf. Sie zu aktivieren gelingt im Rootverzeichnis der Kernelquellen beispielsweise mit »make ARCH=arm bcmrpi_defconfig«.

Der Entwickler prüft und ergänzt gegebenenfalls die Konfiguration (Aufruf durch »make ARCH=arm menuconfig«) auf die Unterstützung des Device Tree generell

(Auswahl »Boot options«) und im Proc-Verzeichnis (Auswahl »Device Drivers | Device Tree und Open Firmware support«, Abbildung 2).

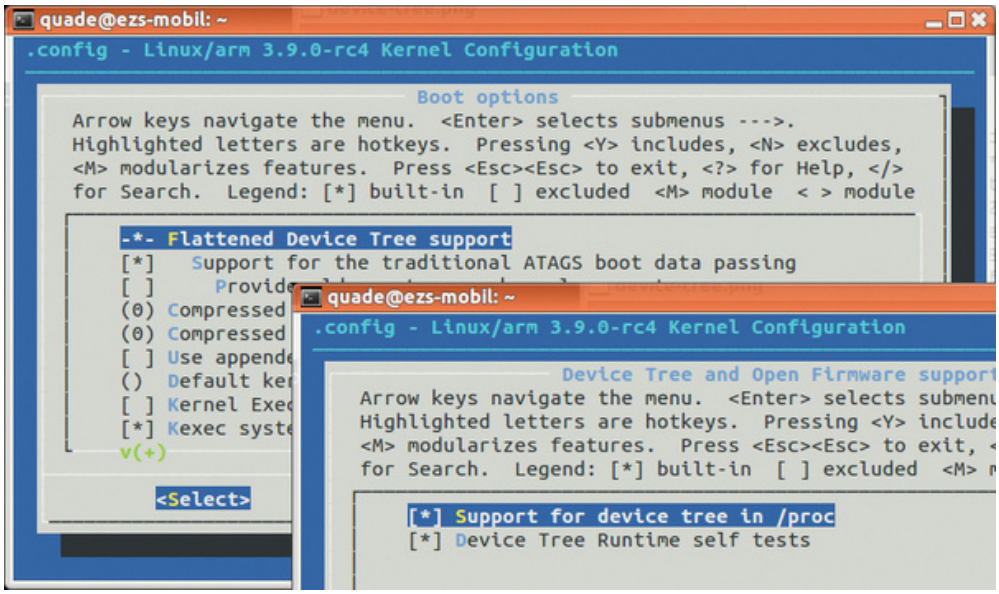


Abbildung 2: Im Zuge der Kernelkonfiguration des Raspberry Pi ist die Device-Tree-Unterstützung zu prüfen.

Den Device Tree Blob erzeugen

Um mit Hilfe des Device Tree Compilers (DTC) den Device Tree Blob zu erzeugen, ruft der Entwickler nach dem Generieren des Kernels im Kernelquellen-Root »make dtbs« auf. Der Device Tree Blob befindet sich nach erfolgreichem Übersetzen im Verzeichnis »arch/arm/boot/dts/«. Ihn, also zum Beispiel »bcm2835-rpi-b.dtb«, kopieren Raspberry-Pi-Besitzer auf ihre SD-Karte in das Verzeichnis »/boot«. An gleicher Stelle ergänzen sie die Datei »config.txt« um die folgenden Zeilen:

```
device_tree=bcm2835.dtb
device_tree_address=0x100
kernel_address=0x8000
disable_commandline_tags=1
init_uart_baud=115200
init_uart_clock=3000000
init_emmc_clock=50000000
```

Diese Variablen legen fest, an welchen Adressen der Bootloader im Hauptspeicher den Device Tree und den Kernelcode ablegen soll. Die Adresse des Device Tree bekommt dann die ARM-CPU bei der Programmübergabe an den Kernel in dem internen Register »r1« übergeben. Das Generieren eines Kernels für den Raspberry Pi wird übrigens Thema einer der nächsten Kern-Technik-Folgen sein.

Listing 1: Device Tree des Raspberry Pi (gekürzt)

```
01 /include/ "skeleton.dtsi"
02
03 / {
04     compatible = "brcm,bcm2835";
05     model = "BCM2835";
06     interrupt-parent = <&intc>;
07
```

```
08      chosen {
09          bootargs = "earlyprintk ... rootfstype=ext4";
10      };
11
12      soc {
13          compatible = "simple-bus";
14          #address-cells = <1>;
15          #size-cells = <1>;
16          ranges = <0x7e000000 0x20000000 0x02000000>;
17
18          timer {
19              compatible = "brcm,bcm2835-system-timer";
20              reg = <0x7e003000 0x1000>;
21              interrupts = <1 0>, <1 1>, <1 2>, <1 3>;
22              clock-frequency = <1000000>;
23          };
24
25          intc: interrupt-controller {
26              compatible = "brcm,bcm2835-armctrl-ic";
27              [...]
28          };
29
30          watchdog {
31              compatible = "brcm,bcm2835-pm-wdt";
32              reg = <0x7e100000 0x28>;
33          };
34
35          uart@20201000 {
36              compatible = "brcm,bcm2835-pl011", "arm,pl011";
37              reg = <0x7e201000 0x1000>;
38              interrupts = <2 25>;
39              clock-frequency = <3000000>;
40          };
41
42          gpio: gpio {
43              compatible = "brcm,bcm2835-gpio";
44              reg = <0x7e200000 0xb4>;
45              [...]
46              gpio-controller;
47              #gpio-cells = <2>;
48              interrupt-controller;
49              #interrupt-cells = <2>;
50          };
51
52          i2c0: i2c@20205000 {
53              [...]
54          };
55          [...]
56          sdhci: sdhci {
57              [...]
58              clocks = <&clk_mmc>;
59              status = "disabled";
60          };
61      };
62
63      clocks {
64          [...]
65          clk_mmc: mmc {
66              [...]
67          };
68          [...]
```

```
69     };
70 };
```

Technik für alle

Device Trees sind für Linux-Experten also in mehrfacher Hinsicht bedeutsam: Der Systemarchitekt erzeugt den zu seiner Plattform gehörenden Device Tree. Der Admin wählt die binäre Repräsentation des Device Tree zum Booten aus und der Treiberprogrammierer unterstützt in seinem Kernelcode die Konfiguration über Device Trees.

Die Device-Tree-Beschreibungen der einzelnen ARM-SoCs und ARM-Boards befinden sich im Linux-Quellcodeverzeichnis unter »arch/arm/boot/dts«. Dateien mit der Datei-Erweiterung ».dtsi« enthalten die Definitionen für einen bestimmten SoC, beispielsweise den im Raspberry Pi zum Einsatz kommenden "High Definition 1080p Embedded Multimedia Applications Processor" BCM 2835 von Broadcom [3].

Boards mit diesem SoC enthalten in ihrer Device-Tree-Beschreibung die Datei »bcm2835.dtsi«. Die Beschreibung für das Board hat die Datei-Erweiterung ».dts«, für den Raspberry Pi heißt die Datei beispielsweise »bcm2835-rpi-b.dts«, die es seit Kernelversion 3.7 gibt. Listing 1 zeigt Ausschnitte aus der SoC-Konfiguration von Kernel 3.9-rc4.

Kleine Baumschule

Der Baum startet mit »/«, also der Wurzel. Die auftretenden geschweiften Klammern bilden jeweils einen neuen Ast, dessen Bezeichnung vor der Klammer steht. Der BCM-2835-Baum hat drei Hauptäste: »chosen«, »soc« und »clocks«. Alle Angaben, die mit einem Doppelkreuz beginnen, spezifizieren Formate. So bedeutet »#address-cells = <1>«, dass Adressen aus einem einzelnen 32-Bit-Wert bestehen, »<2>« steht für zwei 32-Bit-Werte und damit eine 64-Bit-Adresse. Unter »<reg>« firmieren Adressenangaben in der Form *Startadresse Länge*.

Beziehungswissenschaften

Von besonderer Bedeutung ist das Schlüsselwort »compatible«, das die Beziehung zwischen dem Device-Tree-Eintrag und einer Kernelkomponente herstellt – typischerweise einem Treiber. Neben den vordefinierten Schlüsselwörtern wie »compatible«, »reg«, »interrupts« oder auch »ranges« darf der Entwickler auch eigene Variablen definieren und mit Werten versehen. Detaillierte Informationen zu Schlüsselwörtern und deren Syntax liefert [7].

Falls im Kernel konfiguriert, lässt sich aus dem Userland der vollständige Device Tree nach dem Booten auslesen. Er steckt im Verzeichnis »/proc/device-tree«. Die einzelnen Äste sind dabei als Unterverzeichnisse ausgeprägt (Abbildung 3), wobei die Include-Datei die zusätzlichen Äste »memory« und »aliases« in »skeleton.dtsi« definiert.

Die Device-Tree-Beschreibung eines Boards übersetzt schließlich ein Device Tree Compiler (DTC). Das Ergebnis ist ein kompakter Binary-Blob (Device Tree Blob, DTB). Der Code des Compilers findet sich in den Kernelquellen unterhalb des Verzeichnisses »scripts/dts/«.

Abbildung 3: **Der Device Tree am Beispiel des Raspberry Pi.**

Open Firmware

Im Kernel gibt es einen Interpreter, der den Blob beim Booten interpretiert, einen Baum aus »struct device_node« aufbaut, die zugehörigen Treiber aktiviert und mit den passenden Konfigurationsinformationen versorgt (Abbildung 4). Der Quellcode zum Device-Tree-Interpreter findet sich in den Linux-Kernelquellen unter »drivers/of«, »of« steht für Open Firmware, einer von der Firma Sun, dann Oracle, entworfenen Schnittstellen-Spezifikation. Die im Linux-Kernel eingesetzte Spezialvariante nennt sich Flattened Device Tree (FDT).

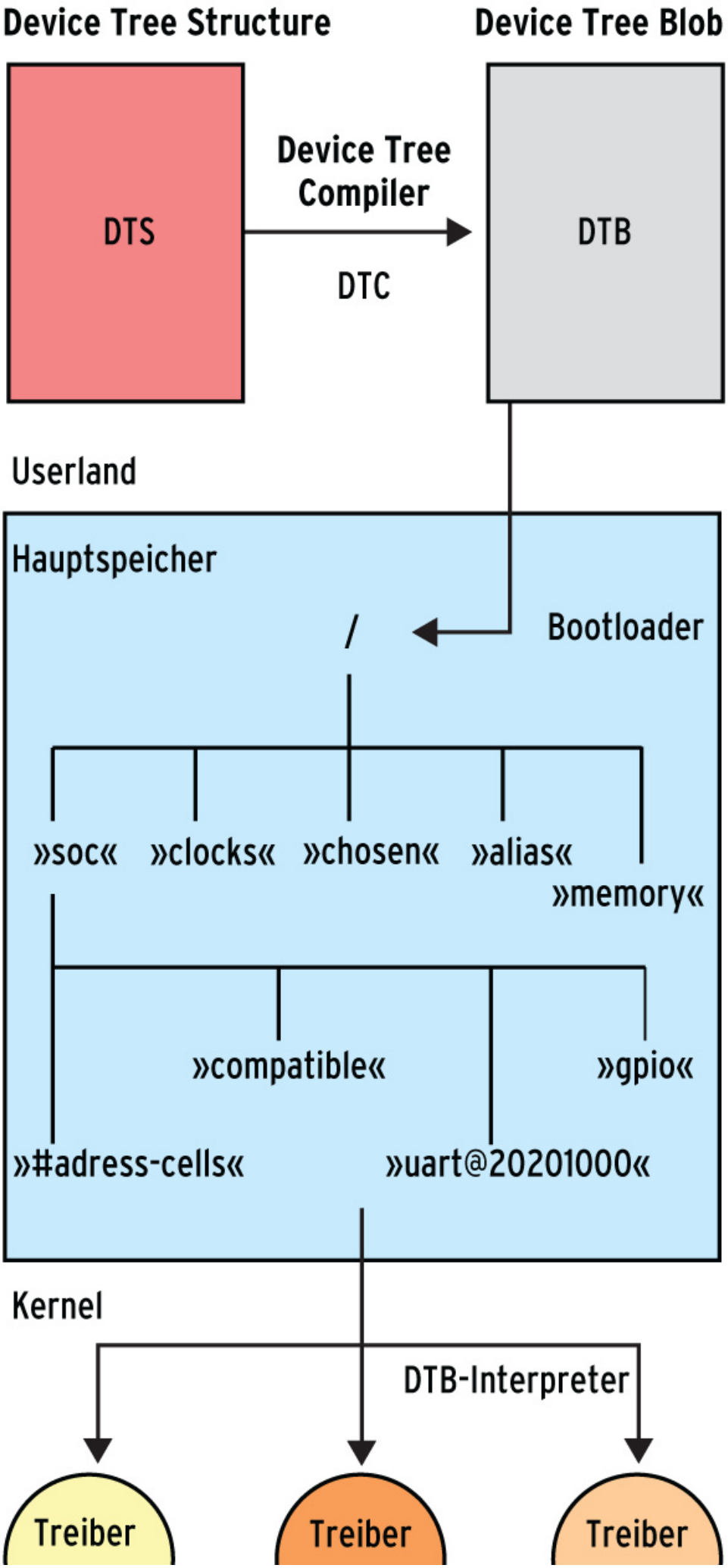

Device Tree Structure
Device Tree Blob
DTS
Device Tree Compiler
DTC
DTB
Userland
Hauptspeicher
/
Bootloader
»soc«
»clocks«
»chosen«
»alias«
»memory«
»compatible«
»gpio«
»#adress-cells«
»uart@20201000«
Kernel
DTB-Interpreter
Treiber
Treiber
Treiber

Abbildung 4: Das Kompilat der DTS-Datei landet per Bootloader im Hauptspeicher.

Außerdem stellt der Kernel eine Reihe von Funktionen bereit, um den Baum zu durchwandern, nach bestimmten Ästen zu suchen und den Zugriff auf die enthaltenen Informationen zu erleichtern. Diese Funktionen verwenden Treiberprogrammierer, die Device Trees supporten, um damit Konfigurationsinformationen auszulesen.

Um den richtigen Treiber auf Basis der Informationen des Device Tree zu aktivieren, benötigt der Interpreter eine Matchtable, wie Linux sie bereits bei USB- oder bei PCI-Geräten einsetzt (Abbildung 5). Die Tabelle ordnet jedem Treiber eine Compatible-ID zu. Diese ID besteht aus einem Herstellernamen und einer Treiberreferenz, beispielsweise »brcm,bcm2835-gpio« wie in Listing 2 zu sehen.

Die Knoten im Device Tree enthalten ebenfalls diese Compatible-ID, sodass der Kernel für jeden Knoten den in der Matchtable spezifizierten zugehörigen Treiber laden kann. Der Treiber bekommt dabei für jedes von ihm zu bedienende Gerät die Device-Tree-Informationen in Form einer Referenz auf den zugehörigen Ast (»struct device_node«) übergeben. Die Referenz ist in der Datenstruktur »struct node« eingebettet, die etwa bei einer Treiber-Probe-Funktion standardmäßig als Parameter auftaucht.

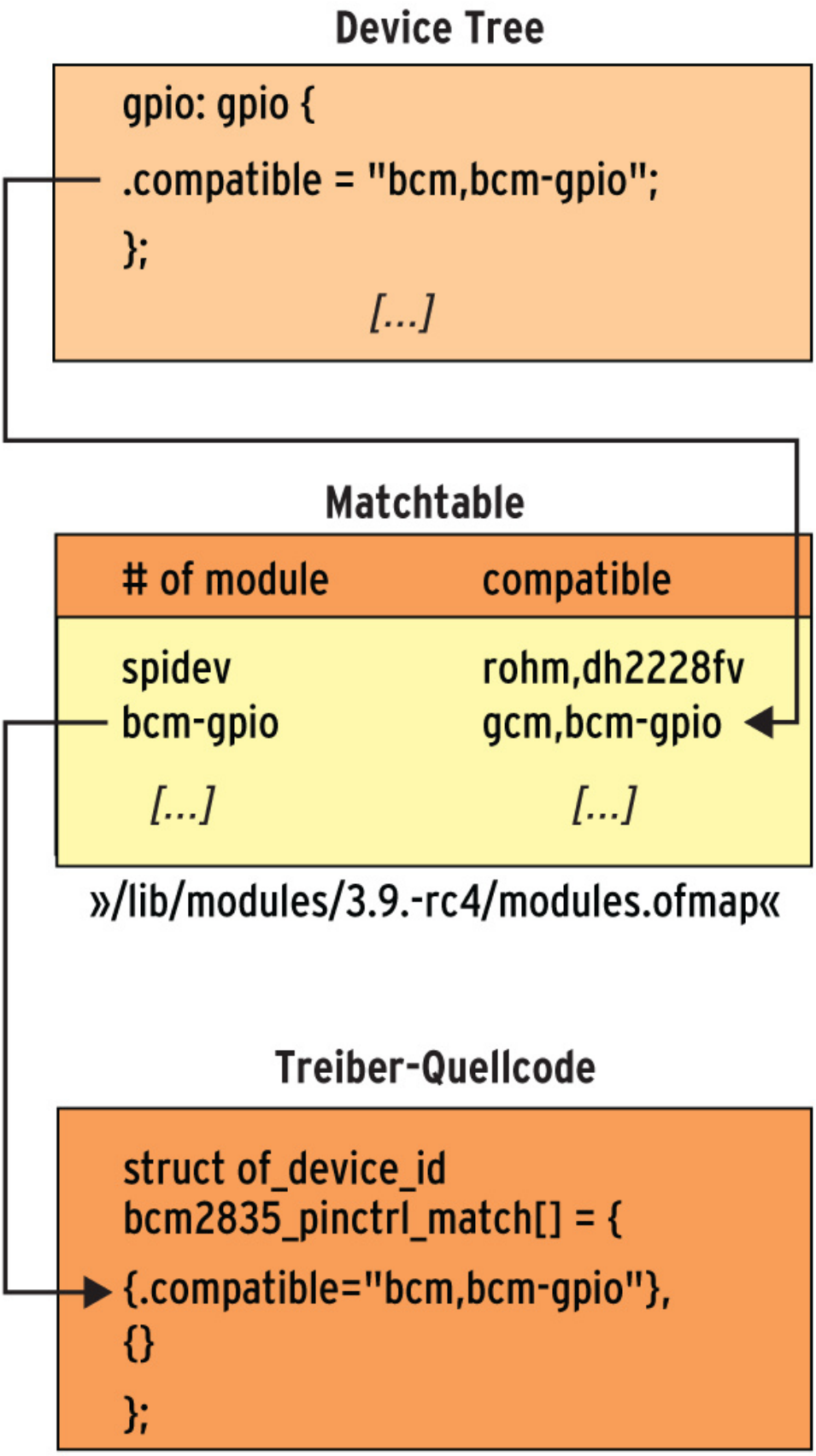


Abbildung 5: Die Matchtable bildet das Bindeglied zwischen Device Tree und Treiber.

Listing 2: Treiber identifizieren

```
01 #ifdef CONFIG_OF
02 static struct
```

```
03   of_device_id bcm2835_pinctrl_match[] = {
04   { .compatible = "brcm,bcm2835-gpio" },
05   {}
06 };
07 MODULE_DEVICE_TABLE(of,bcm_pinctrl_match);
08
09 static struct platform_driver
10   bcm_pinctrl_drvr = {
11   .probe = bcm2835_pinctrl_probe,
12   .remove = bcm2835_pinctrl_remove,
13   .driver = {
14     .name = MODULE_NAME,
15     .owner = THIS_MODULE,
16     .of_match_table = bcm_pinctrl_match,
17   },
18 };
19 module_platform_driver(bcm_pinctrl_drvr);
20 #endif
```

Hardware für den Zugriff reservieren

Diese Probe-Funktion ist in typischen Gerätetreibern für die Identifikation und Übernahme der Hardware zuständig (Listing 3). Zum Auslesen der Daten aus dem Device-Tree-Ast und der Ablage in einer Treiber-internen Variablen verwendet der Programmierer beispielsweise »of_address_to_resource()«.

Um den exklusiven Zugriff organisieren zu können, muss der Programmierer die jeweiligen Ressourcen danach auf bekannte Art und Weise, zum Beispiel über »request_mem_region()«, beim Kernel reservieren [8]. Sind mehrere gleichartige Ressourcen spezifiziert, entscheidet die Reihenfolge über die Zuordnung. Die »0« in Listing 3, Zeile 9, steht damit für die erste Ressource.

Das Auslesen eigener, im bisherigen Device-Tree-Format nicht spezifizierter Informationen gelingt recht einfach über die Funktion »of_get_property()«. Sie liefert von dem übergebenen Ast den mit dem Namen hinterlegten Wert in Form einer Speicheradresse (»mux«) und der Länge des zugehörigen Speicherbereichs (»&size«) zurück:

```
mux = of_get_property(node_ptr, PCS_MUX_BITS_NAME, &size);
```

Den Speicher hinter der zurückgegebenen Adresse verwaltet im Übrigen der Kernel – ihn freizugeben, sollte darum der Treiberprogrammierer tunlichst unterlassen. Weitere Hinweise zur Device-Tree-Unterstützung finden Entwickler in der durchgehend lesenswerten Dokumentation von Xillybus [9].

Listing 3: Probe-Funktion im Gerätetreiber

```
01 static int bcm2835_pinctrl_probe(struct platform_device *pdev)
02 {
03   struct device *dev = &pdev->dev;
04   struct device_node *np = dev->of_node;
05   [...]
06   struct resource iomem;
07   [...]
08
```

```
09   err=of_address_to_resource(np,0,&iomem);
10   if (err) {
11     dev_err(dev,"could not get IO mem\n");
12     return err;
13   }
14
15   pc->base=devm_ioremap_resource(dev, &iomem);
16   [...]
```

Ein ARM-Linux

Device Trees und das ARM-Pincontrol-Subsystem bilden elegante Methoden, um Hardware-Unterschiede zu nivellieren. Um dem "Compile once, run everywhere" auf ARM-Systemen näher zu rücken, muss der Kernelcode aber Device Trees durchgängig unterstützten. Halten sich genügend Entwickler daran, entsteht hoffentlich bald ein Linux, das auf Chrome-Books genauso läuft wie auf verschiedenen Smartphones. (jk)

Infos

1. Jake Edge, "ARM and defconfig files": [http://lwn.net/Articles/391372/]
2. Das U-Boot: [http://www.denx.de/wiki/U-Boot]
3. BCM 2835: [http://www.broadcom.com/products/BCM2835]
4. Pincontrol-Subsystem – Dokumentation im Kernel-Quellcode: [http://lxr.free-electrons.com/source/Documentation/pinctrl.txt]
5. Raspbian: [http://www.raspbian.org]
6. "How to boot using device tree": [https://github.com/raspberrypi/linux/wiki/How-to-boot-using-device-tree]
7. Device Tree Usage: [http://devicetree.org/Device_Tree_Usage]
8. J. Quade, K. Kunst, "Linux-Treiber entwickeln", 3. Auflage: Dpunkt-Verlag 2011; S. 113 ff.
9. "A Tutorial on the Device Tree (Zynq)": [http://xillybus.com/tutorials/device-tree-zynq-1]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Inzwischen ist die dritte Auflage ihres Buches "Linux Treiber entwickeln" erschienen.