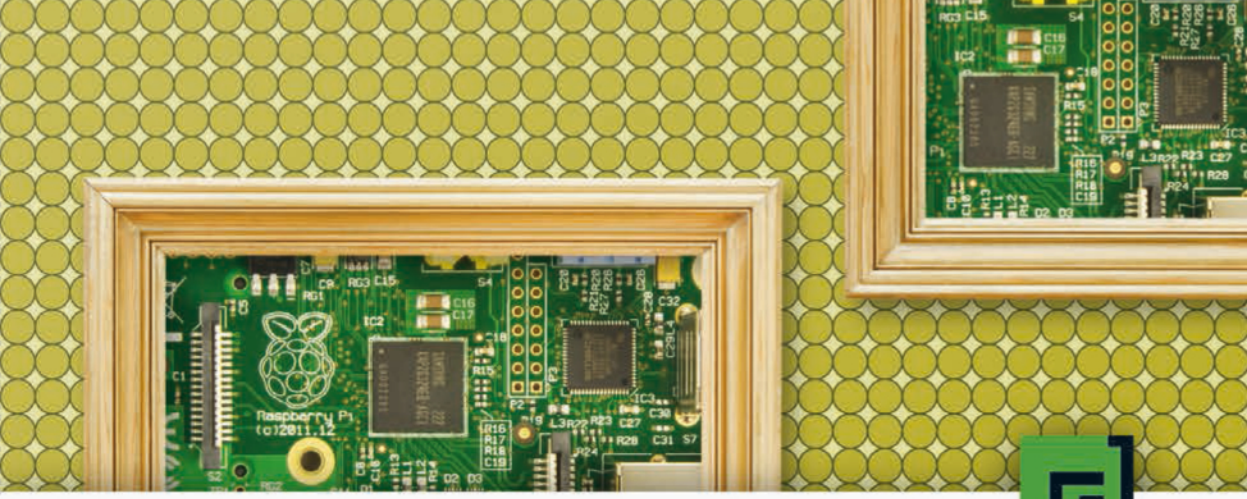


Jürgen Quade

Embedded Linux

lernen mit dem Raspberry Pi

Linux-Systeme selber bauen
und programmieren

dpunkt.verlag

Was sind E-Books von dpunkt?

Unsere E-Books sind Publikationen im PDF- oder EPUB-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken. Sie benötigen zum Ansehen den Acrobat Reader oder ein anderes adäquates Programm bzw. einen E-Book-Reader.

E-Books können Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt.büchern her kennen.

Was darf ich mit dem E-Book tun?

Die Datei ist nicht kopiergeschützt, kann also für den eigenen Bedarf beliebig kopiert werden. Es ist jedoch nicht gestattet, die Datei weiterzugeben oder für andere zugänglich in Netzwerke zu stellen. Sie erwerben also eine Ein-Personen-Nutzungslizenz.

Wenn Sie mehrere Exemplare des gleichen E-Books kaufen, erwerben Sie damit die Lizenz für die entsprechende Anzahl von Nutzern.

Um Missbrauch zu reduzieren, haben wir die PDF-Datei mit einem Wasserzeichen (Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer) versehen.

Bitte beachten Sie, dass die Inhalte der Datei in jedem Fall dem Copyright des Verlages unterliegen.

Wie kann ich E-Books von dpunkt kaufen und bezahlen?

Legen Sie die E-Books in den Warenkorb. (Aus technischen Gründen, können im Warenkorb nur gedruckte Bücher ODER E-Books enthalten sein.)

Downloads und E-Books können sie bei dpunkt per Paypal bezahlen. Wenn Sie noch kein Paypal-Konto haben, können Sie dieses in Minutenschnelle einrichten (den entsprechenden Link erhalten Sie während des Bezahlvorgangs) und so über Ihre Kreditkarte oder per Überweisung bezahlen.

Wie erhalte ich das E-Book von dpunkt?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene Adresse eine Bestätigung von Paypal, sowie von dpunkt eine E-Mail mit den Downloadlinks für die gekauften Dokumente sowie einem Link zu einer PDF-Rechnung für die Bestellung.

Die Links sind zwei Wochen lang gültig. Die Dokumente selbst sind mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag:
Frau Karin Riedinger (riedinger (at) dpunkt.de bzw. fon 06221-148350).

Jürgen Quade studierte Elektrotechnik an der TU München. Danach arbeitete er dort als Assistent am Lehrstuhl für Prozessrechner (heute Lehrstuhl für Realzeit-Computersysteme), promovierte und wechselte später in die Industrie, wo er im Bereich Prozessautomatisierung bei der Softing AG tätig war. Heute ist Jürgen Quade Professor an der Hochschule Niederrhein, wo er u.a. das Labor für Echtzeitsysteme betreut. Seine Schwerpunkte sind Echtzeitsysteme, Embedded Linux, Rechner- und Netzwerksicherheit sowie Open Source. Als Autor ist er vielen Lesern über das dpunkt-Buch »Linux-Treiber entwickeln« und die regelmäßig erscheinenden Artikel der Serie »Kern-Technik« im Linux-Magazin bekannt.

Jürgen Quade

Embedded Linux lernen mit dem Raspberry Pi

Linux-Systeme selber bauen und programmieren

dpunkt.verlag

Jürgen Quade
quade@hsnr.de

Lektorat: René Schönfeldt
Copy Editing: Ursula Zimpfer, Herrenberg
Satz: data2type GmbH, Heidelberg
Herstellung: Frank Heidt
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN
Buch 978-3-86490-143-0
PDF 978-3-86491-509-3
ePub 978-3-86491-510-9

1., korrigierter Nachdruck

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1

Vorwort

Embedded Systems

Die Zahlen der Marktforscher sind beeindruckend: Bei einer Weltbevölkerung von über 7 Milliarden Menschen werden pro Jahr mehr als 13 Milliarden Prozessoren hergestellt. Ein vergleichsweise kleiner Anteil davon (etwa 350 Millionen) landet in Form eines PCs oder Notebooks auf unserem Schreibtisch. Der erheblich größere Teil wird aber in Waschmaschinen, Autos, TV-Geräten, Digicams, Smartphones oder Automatisierungsanlagen eingebettet.

Embedded Linux

Der Markt dieser eingebetteten Systeme kann grob in zwei Lager eingeteilt werden. Sogenannte Deeply Embedded Systems setzen auf einfache 8- oder 16-Bit-Prozessoren, wie der bekannte Atmega, der im populären Arduino-Board steckt. Anwendungs- und Systemsoftware ist häufig proprietär und die Geräte erledigen vorwiegend einfache Aufgaben, so beispielsweise die Spiegelsteuerung in einem Auto. Werden demgegenüber komplexere Funktionen — allem voran Vernetzung — gefordert, greift der Entwickler für ein Open Embedded System zu 32-Bit-Prozessoren und immer häufiger zu einem Standardbetriebssystem, insbesondere Linux. Linux treibt heute Fernseher, Digicams, Router-Hardware, Uhren und vieles mehr an. Das auch aus gutem Grund, schließlich ist es funktional, bekannt und vor allem Open Source.

Ein eigenes Embedded Linux konfektionieren

Allerdings wird Linux in einem eingebetteten System nur selten über eine Standarddistribution, wie beispielsweise Ubuntu, installiert. Das liegt nicht nur an der meist nicht konformen Hardware. Vielmehr wird das System auf die leistungsschwächere Hardware, auf andere Hardwareplattformen (ARM statt x86) und auf die auszuführende Funktionalität abgestimmt. Auch werden andere Update-Zyklen benötigt als auf dem Desktop üblich. Hinzu kommt die Notwendigkeit, Sensoren und Aktoren anzukoppeln und softwaretechnisch anzusprechen.

Hierfür benötigen Entwickler ein umfangreiches Know-how. Das beginnt beim Entwicklungsprozess, der typischerweise in Form einer Host-/Target- und Cross-Entwicklung abläuft, der Entwicklungsumgebung, die linuxspezifisch kommandozeilenorientiert ist, geht über notwendige Kernelerweiterungen, um damit eigene Hardwareerweiterungen anzusprechen, und endet bei den Applikationen, die nicht zuletzt

aufgrund minimaler Ressourcen nur bedingt auf vorhandene Funktionen aufsetzen können.

Dieses Know-how möchte das vorliegende Buch in praxisorientierter und kompakter Weise vermitteln.

Der Raspberry Pi als Praxisbeispiel

Damit Sie die Inhalte nachvollziehen können, werden viele Techniken mithilfe des Raspberry Pi vorgestellt. Der Raspberry Pi ist ein Anfang 2012 auf den Markt gekommener Kleincomputer, der sich nicht zuletzt wegen seiner leichten Erweiterbarkeit gut als Herzstück eines eingebetteten Systems einsetzen lässt. Mit einem ARM-Prozessor ausgestattet, ist er zudem bei einem Preis von unter 40 € (ohne Speicherkarte, ohne Netzteil) preiswert. Dieser günstige Preis zusammen mit einem »Fun-Faktor« haben für eine hohe Verbreitung gesorgt. Aus diesem Grund eignet er sich besonders gut als Basis für eigene Experimente im Bereich Embedded Linux.

Ziel des Buchs: Ein eigenes Embedded-Linux-System

Methodisch werden Sie an das Thema durch den Aufbau und die Konfektionierung eines komplett eigenen Systems herangeführt. Neben dem Lerneffekt steht am Ende ein eingebettetes System für den Raspberry Pi, das effizienter, schneller und vor allem auch sicherer als eine Standarddistribution ist.

Vom Systemanwender zum Systementwickler: Während die meisten Bücher rund um den Raspberry Pi zeigen, wie Sie — häufig auf Basis der Linux-Variante Raspbian — Systeme unterschiedlicher Funktionalität aufbauen, entwickeln Sie mithilfe des vorliegenden Mitmach-Buches und des Raspberry Pi Ihr eigenes Embedded Linux. Mit der Anwendung im Blick zeigt das Buch, woraus ein Embedded Linux besteht und wie es funktioniert. Es erläutert Hintergründe und zeigt Lösungen, die sich auch in zeitkritischeren Umgebungen einsetzen lassen. Der Raspberry Pi ermöglicht den schnellen und einfachen Einstieg in die Welt eingebetteter Linux Systeme.

Das Thema Embedded Linux lässt sich in einem Buch von rund 300 Seiten auch als Einführung nicht annähernd vollständig abhandeln. Die Auswahl und Tiefe der dargebotenen Aspekte orientieren sich primär an deren Wichtigkeit, an der Aktualität, aber auch an meinen eigenen Fachkenntnissen. Sie erfolgen in eher kompakter Form.

Kempen, im März 2014

Inhaltsverzeichnis

1 Einleitung

Im Bereich eingebetteter Systeme ist Linux weit verbreitet und eine feste Größe. In Kombination mit der preiswerten Embedded-Plattform Raspberry Pi bildet es ein optimales Gespann, um Kenntnisse und Techniken, die für die Entwicklung eingebetteter Systeme notwendig sind, nachvollziehbar und praxisorientiert zu vermitteln.

Das als Einführung in das Thema gedachte Buch beschreibt den Aufbau, die Konzeption und die Realisierung eingebetteter Linux-Systeme auf Basis des Raspberry Pi.

Es demonstriert, wie als Teil einer Host-/Target-Entwicklung auf einem Linux-Hostsystem eine (Cross-)Toolchain installiert wird, um damit lauffähigen Code für die Zielplattform zu erzeugen. Es zeigt, aus welchen Komponenten die Systemsoftware besteht und wie sie für den spezifischen Einsatz konfektioniert und zu einem funktionstüchtigen Embedded System zusammengebaut wird. Dieser Vorgang wird in seinen Einzelschritten (from scratch) ebenso beschrieben wie die Automatisierung mithilfe des Systembuilders »Buildroot«. Das Buch führt außerdem in die softwaretechnische Ankopplung von Hardware ein, stellt dazu aktuelle Applikationsschnittstellen vor und demonstriert die Programmierung einfacher Linux-Treiber, beispielsweise für den Zugriff auf GPIOs. Tipps und Tricks, wie beispielsweise zur Erreichung kurzer Entwicklungszyklen durch ein Booten über ein Netzwerk, runden das Thema ebenso ab wie ein Abschnitt über die Sicherheit (Embedded Security) im Umfeld eingebetteter Systeme.

Ein beispielhaftes Projekt zum Abschluss zeigt die vorgestellten Techniken im Verbund.

Ziel des Buches ist es,

- eine praxisorientierte, kompakte Einführung in Embedded Linux zu geben und die Unterschiede zur Entwicklung bei einem Standardsystem aufzuzeigen,
- anhand eines von Grund auf selbst gebauten Linux-Systems den internen Aufbau und die Abläufe nachvollziehbar vorzustellen,
- exemplarisch eine einfache, auf Linux basierende Cross-Entwicklungsumgebung für eingebettete Systeme vorzustellen und damit

Bootloader, Kernel und Rootfilesystem für den Raspberry Pi zu erstellen,

- Grundkenntnisse für den Hardwarezugriff und die zugehörige Treibererstellung zu vermitteln,
- die Limitierungen und Besonderheiten bei der Erstellung von Applikationen für eingebettete Systeme vorzustellen,
- die Anforderungen an Security zu verdeutlichen und geeignete Techniken zur Realisierung mitzugeben.

Nicht thematisiert werden unter anderem die Portierung des Linux-Kernels auf eine neue Hardwareplattform, grafische Benutzerinterfaces, Realzeiterweiterungen wie der RT-PREEMPT Patch, Adeos/Xenomai oder Rtai und die Verwendung von integrierten Entwicklungsumgebungen (IDE).

Zielgruppen

Entwickler

Das Buch richtet sich damit an Entwickler, die in das Thema Embedded Linux neu einsteigen oder als Umsteiger bisher mit anderen eingebetteten Systemen Erfahrungen gesammelt haben. Nach der Lektüre kennen sie die Vorteile und die Eigenheiten eines Embedded Linux ebenso wie die Problembereiche. Neben dem Gesamtüberblick lernen sie das Treiberinterface und die wichtigsten Anwendungsschnittstellen kennen. Sie sind in der Lage, eine Entwicklungsumgebung aufzusetzen, die notwendigen Komponenten auszuwählen, zu konfigurieren und schließlich automatisiert zu einem funktionierenden Gesamtsystem zusammenzufügen.

Studenten der technischen Informatik

Studenten der technischen Informatik erarbeiten mit diesem Buch praxisorientiert das allgemeine Grundlagenwissen über eingebettete Systeme und deren Entwicklung. Sie lernen die Komponenten und die Zusammenhänge im Detail kennen. Nach der Lektüre können sie eingebettete Systeme mit Linux planen und softwareseitig realisieren. Über vollständige Anleitungen sammeln sie die ersten praktischen Erfahrungen, wobei der Einstieg in die praktische Umsetzung über die im Anhang zu findenden Crashkurse erleichtert wird.

Hobbyisten

Hobbyisten finden ein Mitmach-Buch vor, das ihnen hilft, die Möglichkeiten des Raspberry Pi auszuschöpfen. Die auf dem Webserver zur Verfügung gestellten Komponenten erleichtern dabei den Aufbau und die Fehlersuche und helfen bei der Überbrückung fachlicher Lücken.

Notwendige Voraussetzungen

Für die Lektüre des Buches sind Grundkenntnisse in den folgenden Bereichen sehr nützlich:

- Linux-Systemadministration
- C-Programmierung
- Rechnerhardware
- Ankopplung und Ansteuerung von Hardware
- Raspberry Pi

Begriffe wie Shell, Editor, Compiler, Flash, Interrupt, Ethernet, tcp/ip, DHCP oder Hexziffern sollten Ihnen nicht ganz fremd sein. Sie finden die benötigten Grundkenntnisse übrigens auch in vielen Einsteigerbüchern zum Raspberry Pi.

Die Inhalte werden praxisorientiert vorgestellt, sodass diese mit einem Linux-Rechner als Host und einem Raspberry Pi als Target vom Leser nachgebaut werden können. Das Buch ist als Einführung (appetizer) gedacht und verweist zur vertiefenden Auseinandersetzung mit dem spannenden Thema auf weiterführende Quellen (Literatur).

Scope

Die Entwicklung eingebetteter Systeme findet typischerweise als sogenannte Host-/Target-Entwicklung statt. Als Hostsystem dient für dieses Buch ein Ubuntu 12.04 LTS in der 64-Bit-Variante, das auf einem Kernel in der Version 3.2.0 aufbaut (Abb. 1-1). Als Target wird ein Raspberry Pi Typ B mit 512 MByte Hauptspeicher und einer mindestens 2 GByte großen Flash-Karte eingesetzt. Hier kommt beim selbst gebauten System ein Kernel der Version 3.6.11 und der zur Zeit der Bucherstellung aktuellen Version 3.10.9 zum Einsatz. Ab und zu werden Anleihen auf ein vorgefertigtes System für den Raspberry Pi gemacht, auf Raspbian. Dieses wird in Version `2013-07-26-wheezy-raspbian` verwendet.

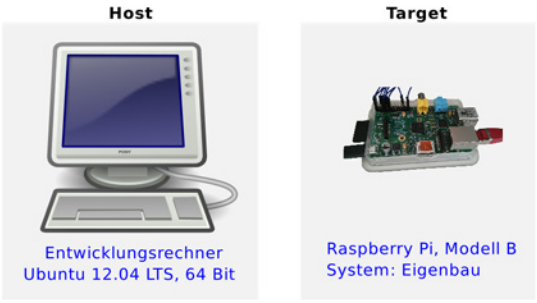


Abb. 1-1
Host-/Target-Entwicklung

Für das selbst gebaute System wird die sogenannten Busybox — ein Multicall-Binary — in der Version 1.21.1 eingesetzt. Der Systembuilder `buildroot` wird in der Version 2013.05 verwendet. Neuere Versionen der

Werkzeuge dürften typischerweise ebenfalls funktionieren. Als Emulator wird Qemu in der Version 1.0 benutzt.

Aufbau des Buches

Kapitel 2 und 3: Grundlagen

Die Einführung in das Thema Embedded Linux in Kapitel 2 beschäftigt sich mit dem grundlegenden Basiswissen. Dazu gehören die Architektur, der Entwicklungsprozess, aber auch Linux und der Raspberry Pi.

Darauf aufbauend wird in Kapitel 3 gezeigt, wie ein Kernel, ein Rootfilesystem und ein Bootloader von Grund auf (from Scratch) manuell generiert und zu einem eingebetteten System zusammengesetzt werden.

Kapitel 4: Systembuilder

Je mehr Funktionalität benötigt wird, desto komplexer wird der manuelle Aufbau eines Embedded System. Systemgeneratoren wie beispielsweise `buildroot`, der in Kapitel 4 vorgestellt wird, automatisieren und erleichtern diesen Vorgang.

Kapitel 5: Anwendungsentwicklung

Ein eingebettetes System erhält seine eigentliche Funktionalität erst durch eine Applikation. Bedingt durch die notwendige Cross-Entwicklung, die limitierten Ressourcen, die häufig vorkommenden Anforderungen an das Realzeitverhalten und die Interaktion mit Hardware, gibt es einige Einschränkungen und Besonderheiten bei der Applikationserstellung, die in Kapitel 5 zusammen mit relevanten Schnittstellen vorgestellt werden.

Kapitel 6: Gerätetreiber

Für die Ankopplung von proprietärer Hardware, zum effizienten und schnellen Einlesen von Sensoren und der Ausgabe von Signalen (Aktoren) werden eigene Gerätetreiber benötigt. Die dafür erforderlichen Grundlagen der Kernel- und Treiberprogrammierung beschreibt Kapitel 6.

Kapitel 7: Sicherheit

Unverzichtbar für jedes vernetzte, technische Gerät, ob als Produkt oder im privaten Bereich eingesetzt, sind grundlegende Mechanismen aus dem Bereich IT-Security. Hierzu gehört beispielsweise die Härtung des Systems durch eine Firewall, das Rechtemanagement oder der Entwurf sicherer Applikationen. Der »Embedded Security« ist Kapitel 7 gewidmet.

Kapitel 8: Ein komplettes Projekt

In Kapitel 8 wird als abgeschlossenes Projekt gezeigt, wie mithilfe der im Buch gewonnenen Erkenntnisse eine simple Messagebox aufgebaut werden kann. Diese zeigt beliebige Nachrichten an, die Sie per Webinterface an das Embedded Linux übermitteln.

Weitere Informationen zu den Themenbereichen Embedded Systems und Embedded Linux

- Das von mir mitverfasste Buch *Linux-Treiber entwickeln* [QuKu2011b] beschreibt systematisch die Gerätetreiber- und Kernelprogrammierung. Im Kontext der eingebetteten Systeme sind, mit

vielen Codebeispielen untermauert, detailliert die wesentlichen kernelspezifischen Aspekte nachzulesen. Das Buch stellt damit eine Abrundung des Themas nach »unten« dar.

- Während das Treiberbuch vor allem die kernelspezifischen Aspekte erörtert, behandelt das ebenfalls von mir mitverfasste Buch *Moderne Realzeitsysteme kompakt — Eine Einführung mit Embedded Linux* [QuMä2012] anwendungs- und architekturspezifische Aspekte. Das Buch stellt damit eine Abrundung des Themas nach »oben« dar.
- Auf der Webseite von Free-Electrons [http://free-electrons.com] finden Sie qualitativ hochwertige Unterlagen und Tutorials rund um das Thema Embedded Linux. Die Macher der Seite bieten auch Schulungen an.
- Thomas Eißenlöffel beschreibt in seinem Buch *Embedded-Software entwickeln* [Eißenlöffel2012] die Grundlagen der Programmierung eingebetteter Systeme mit dem Schwerpunkt auf der Anwendungsentwicklung von Deeply Embedded Systems.
- Die Webseite [elinux.org] widmet sich dem Thema *Embedded Linux*. Hier finden sich neben allgemeinen Informationen auch sehr wertvolle, spezifische Angaben beispielsweise zum Raspberry Pi.

Weitere Informationen zum Raspberry Pi

- The MagPi. Das monatliche Magazin behandelt Themen rund um den Raspberry Pi. Die einzelnen Ausgaben sind kostenlos und lassen sich als PDF von der Webseite herunterladen. Die Artikel sind in englischer Sprache verfasst. Die verständlich geschriebenen Artikel eignen sich sehr gut für Anfänger. Weitere Informationen unter [http://www.themagpi.com/].
- Maik Schmidt zeigt in seinem Buch *Raspberry Pi* [Schmidt2014] alles, was zum Umgang mit der Himbeere notwendig ist. Dazu gehört die Installation eines Systems auf der SD-Karte, die Konfiguration von Standardapplikationen und der einfache Anschluss von Hardware. Im Anhang finden Sie eine Einführung in Linux.
- Zur Hardware des Raspberry Pi gibt es diverse Datenblätter, beispielsweise auch das Datenblatt *BCM2835 ARM Peripherals*, das die Register für den Peripheriezugriff beschreibt [bcm2835].
- Homepage der Standarddistribution für den Raspberry Pi: [http://www.raspberrypi.org]. Download eines Systemimages unter [http://www.raspberrypi.org/downloads].

- Für diejenigen, die sich mehr mit Hardware beschäftigen, ist das Werkzeug Fritzing interessant. Damit lassen sich sehr intuitiv Schaltpläne erstellen. Fritzing unterstützt den Raspberry Pi. Weiterführende Informationen unter [http://fritzing.org].

Verzeichnisbaum

Im Rahmen des Buches werden verschiedene eingebettete Systeme aufgebaut und im Emulator Qemu oder auf dem Raspberry Pi getestet. Die notwendigen Softwarekomponenten sind dabei folgendermaßen organisiert (Abb. 1-2):

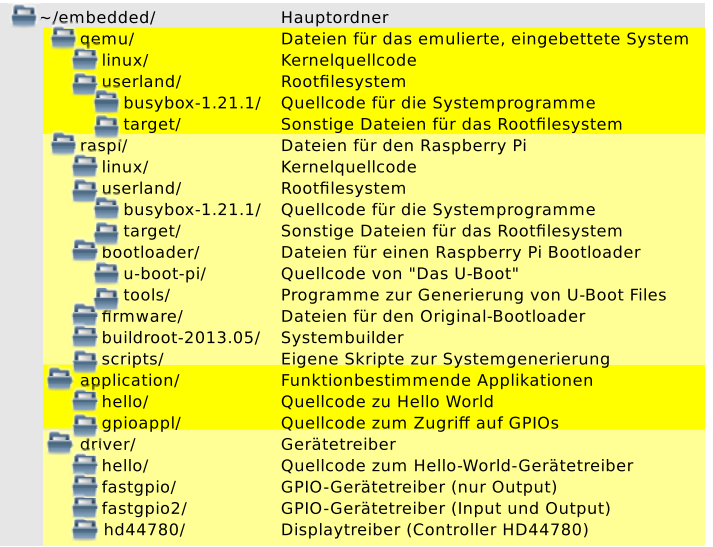


Abb. 1-2 *Ordnerstruktur zur Datenablage*

Im Heimatverzeichnis wird das Verzeichnis `embedded/` angelegt. Darunter gibt es die Verzeichnisse `qemu/`, `raspi/`, `application/` und `driver/`. In `qemu/` wiederum findet sich ein Verzeichnis für den Kernel (`linux/`) und ein Verzeichnis für das Userland (`userland/`).

Das Verzeichnis `raspi/` ist etwas komplexer strukturiert. Hier werden wir zwei unterschiedliche Systeme konstruieren. Ein System — ähnlich dem für den Emulator — ist komplett von Grund auf in allen Komponenten selbst entwickelt. Die Komponenten hierfür finden Sie in den Verzeichnissen `linux/` und `userland/`. Das zweite System wird mithilfe des Systembuilders buildroot aufgesetzt. Alle Komponenten sind im zugehörigen Verzeichnis `buildroot-2013.05/` zu finden. Außerdem werden wir für den Raspberry Pi den Bootloader »Das U-Boot« generieren (Verzeichnis `bootloader/`). Im Ordner `firmware` finden sich die Dateien für den Original-Bootloader des Raspberry Pi. Außerdem gibt es mit

`scripts/` noch ein Verzeichnis, in dem die eigenen Skripte für die Generierung der Komponenten abgelegt werden.

Der Code der im Rahmen des Buches vorgestellten Applikationen findet sich unter `application/`, Code für drei einfache Gerätetreiber unter `driver/`.

Eine genauere Aufschlüsselung der nachfolgenden Verzeichnisse erfolgt in den entsprechenden Kapiteln.

In den Beispielen hat der Entwicklungsrechner den Namen »felicia«, der für die Entwicklung eingesetzte Login lautet »quade«. Der Standardprompt (Eingabezeile) im Terminal des Entwicklungsrechners sieht damit in den Beispielen folgendermaßen aus:

```
quade@felicia:~>
```

Der Prompt auf dem Raspberry Pi besteht aus einem Doppelkreuz (»#«):

```
#
```

Für Ihre Umgebung müssen Sie den Rechnernamen »felicia« und den Loginnamen »quade« durch Ihre eigenen austauschen.

Normalerweise sind Kommandos zeilenorientiert. Passt ein Kommando einmal nicht in eine einzelne Zeile, darf es auch auf mehrere Zeilen verteilt werden. Dazu ist am Ende einer Zeile, zu der es eine Folgezeile gibt, ein umgekehrter Schrägstrich »\« zu setzen:

```
quade@felicia:~/embedded/raspi> mv \
   /media/boot/kernel.img /media/boot/kernel.img.org
```

Onlinematerial

Die Webseite zum Buch

Entwicklungen im Umfeld eingebetteter Systeme sind komplex und deshalb gibt es ganz unterschiedliche Ursachen für auftretende Fehler. Da fast immer mehrere Komponenten beteiligt sind, finden Sie auf der Webseite zum Buch ([https://ezs.kr.hsnr.de/EmbeddedBuch/]) wesentliche Entwicklungsergebnisse der einzelnen Kapitel wie Konfigurationsdateien, Skripte aber auch Imagedateien. Mithilfe dieser Dateien können Sie den Fehler eingrenzen und typischerweise auf die schadhafte Komponente reduzieren.

2 Gut zu wissen

Eine integrierte, mikroelektronische Steuerung wird als eingebettetes System (embedded system) bezeichnet. Es erfüllt meist eine spezifische Aufgabe und hat sehr häufig — man denke beispielsweise an das Antiblockiersystem im Auto — kein ausgeprägtes Benutzerinterface.

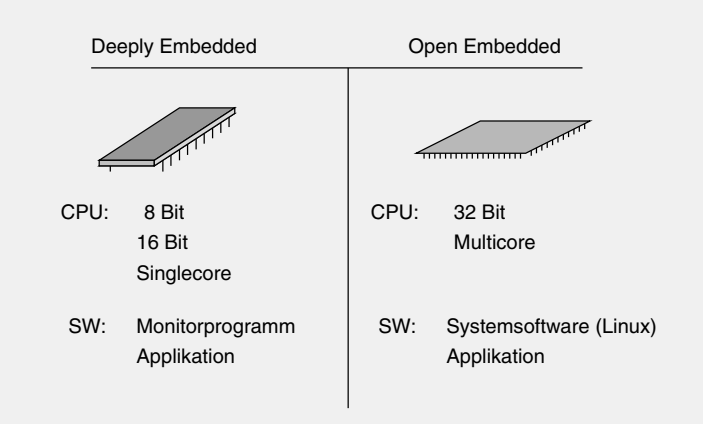


Abb. 2-1
Klassifizierung eingebetteter Systeme

Beispiele für eingebettete Systeme gibt es zuhauf: WLAN-Router, Navi-Systeme, elektronische Steuergeräte im Auto, die Steuerung einer Waschmaschine, Handy und so weiter. Man kann eingebettete Systeme unterklassifizieren in offene (open) und in Deeply Embedded Systems. Deeply Embedded Systems sind typischerweise für genau eine einzelne Aufgabe konstruiert und benötigen dafür einfache Hardware, meist basierend auf einem 8- oder 16-Bit-Mikrocontroller. Sie kommen oftmals ohne spezifische Systemsoftware aus. Offene eingebettete Systeme sind für komplexe Aufgaben gedacht, setzen immer häufiger auf 32-Bit-Prozessoren und auf standardisierte Systemsoftware. Die Grenze zwischen den beiden Kategorien ist fließend.

Eingebettete Systeme lassen sich durch eine Reihe unterschiedlicher Anforderungen und Eigenschaften von Standardsystemen abgrenzen. Tabelle 2-1 zeigt, dass eingebettete Systeme neben den üblichen Anforderungen an Kosten und Funktionalität zusätzliche Kriterien erfüllen müssen. So wird in vielen Einsatzbereichen ein *Instant on* benötigt; das

Gerät muss also unmittelbar nach dem Einschalten betriebsbereit sein. Dazu muss das eingesetzte Betriebssystem kurze Boot-Zeiten garantieren, sodass beispielsweise bei einem Pkw direkt nach dem Einsteigen das (elektronische) Cockpit zur Verfügung steht.

Umgekehrt muss das Betriebssystem aber auch damit zurechtkommen, dass es ohne Vorwarnung stromlos geschaltet wird. Das ist der Fall, wenn der Strom ausfällt, beispielsweise weil der Anwender den Stromstecker zieht.

Wer an Smartphones oder Uhren denkt, dem wird sehr schnell klar, dass auch die räumlichen Ausmaße (Größe) und das Gewicht ein Kriterium sind. Oftmals ist der Einbauplatz begrenzt. Denkt man an elektronische Steuergeräte im Automobil (ECU, Electronic Control Unit), wird klar, dass diese eingebetteten Systeme meist kein Benutzerinterface, keinen Bildschirm und erst recht keine Tastatur haben. Man spricht hier auch vom Headless-Betrieb. Dieser geht häufig einher mit dem Nonstop-Betrieb, bei dem Geräte 24 Stunden am Tag und 365 Tage im Jahr betrieben werden; nicht selten sogar über 20 oder gar 30 Jahre. Außerdem wird Robustheit gefordert, werden die Geräte doch oft im rauen Umfeld eingesetzt.

Aus diesen Anforderungen ergibt sich, dass für ein eingebettetes System typischerweise weniger Ressourcen zur Verfügung stehen. Daher wird oft eine auf die Gerätefunktion angepasste Hardwareplattform entwickelt und die Systemsoftware wiederum auf die Hardware und die gewünschte Funktionalität angepasst. Die Systeme werden diskless aufgebaut, bewegte Teile, wie beispielsweise Festplattenlaufwerke, versucht der Entwickler zu vermeiden.

Tabelle 2-1 *Anforderungen an eingebettete Systeme*

Anforderung
Funktionalität
Preis
Robustheit, funktionstüchtig im rauen Umfeld
Instant on, kurze Boot-Zeiten
Fast poweroff, ohne Vorwarnung stromlos
Räumliche Ausmaße
Kein oder eingeschränktes GUI
Headless-Betrieb: keine Tastatur, kein Bildschirm
Nonstop-Betrieb (Dauerbetrieb)
Lange Lebensdauer (hohe Standzeit)

2.1 Die Architektur eingebetteter Systeme

Ein eingebettetes System besteht aus Hardware (CPU, Speicher, Sensorik, Aktorik) und Software (Firmware, Betriebssystem, Anwendung).

2.1.1 Hardware

Die Hardware eines eingebetteten Systems entscheidet über Leistung, Stromverbrauch, Größe und Robustheit. Ein Embedded System wird unter anderem dadurch robust, dass es ohne bewegte Teile, also ohne Lüfter oder Festplatte, auskommt, möglichst wenig Steckverbindungen aufweist, für eine optimale Temperatur ausgelegt (eventuell klimatisiert) und gekapselt ist, sodass mechanische Beanspruchungen abgewehrt werden.

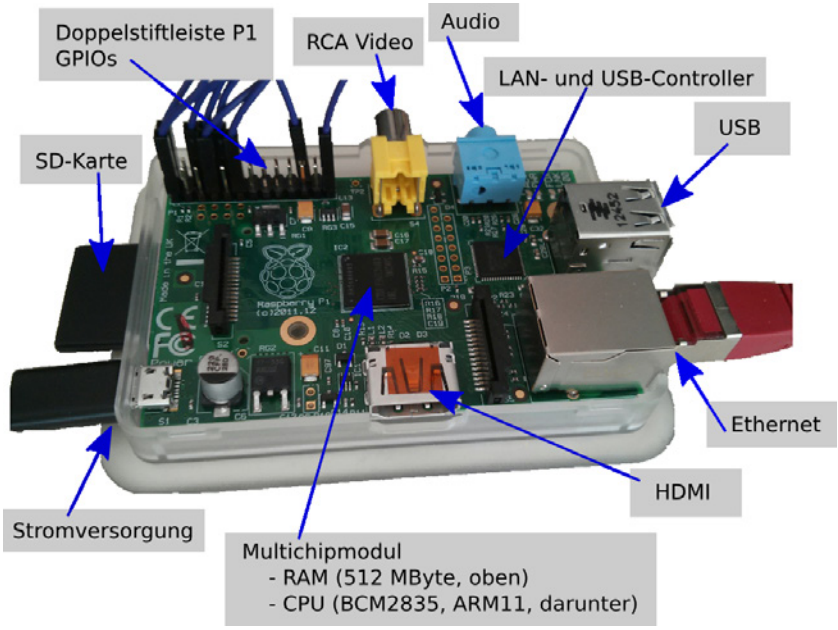


Abb. 2-2
Der Raspberry Pi

Im Kern besteht die Embedded-Hardware aus einem Prozessor (CPU), dem Hauptspeicher (RAM), dem Hintergrundspeicher (Festplatte, Flash, SD-Karte) und diversen Schnittstellen (Abb. 2-2). Hierzu gehört klassischerweise die serielle Schnittstelle, zunehmend auch USB. Darüber hinaus findet sich häufig eine Netzwerkschnittstelle (Ethernet), WLAN, Grafikausgabe (VGA oder HDMI) und eventuell auch Audio. Zur Ankopplung weiterer Peripherie werden einige Leitungen, Pins beziehungsweise Steckverbinder eingesetzt, die digitale Signale übertragen (General Purpose Input Output, GPIO). Da diese Leitungen typischer-

weise eine nur begrenzte Leistung haben, werden sie über Treiber mit beispielsweise LEDs oder Relais (Ein-/Aus-/Umschalter) verbunden. Häufig ist noch eine galvanische Entkopplung (über Optokoppler) notwendig, damit die Embedded-Hardware vor Störungen aus Motoren oder Ähnlichem geschützt sind.

Die Prozessoren in eingebetteten Systemen sind häufig als System on Chip (SoC; monolithische Integration) ausgeprägt. Bei diesen befindet sich nicht nur die eigentliche CPU auf dem Chip, sondern zusätzlich noch die sogenannte Glue-Logic (Interrupt-Controller, Zeitgeber, Watchdog), Grafikcontroller, Audio, digitale Ein-/Ausgabeleitungen (GPIO), Krypto-Module und so weiter. Bei einer CPU plus Peripherie spricht man auch von einem Mikrocontroller.

Als CPU wird zunehmend ein ARM-Core eingesetzt. Alternativ sind noch PowerPC, Mips und x86-Architekturen im Einsatz. Für Deeply Embedded Systems werden gerne ATMEGA- und PIC-Prozessoren ausgewählt, wie sie beispielsweise auf einem Arduino-Board anzutreffen sind. Häufig bieten die eingesetzten Prozessoren keine Gleitkommaeinheit (Floating Point Unit). Zusätzlich werden sie durch digitale Signalprozessoren unterstützt, die beispielsweise Sprachverarbeitung oder Krypto-Funktionen übernehmen.

Hintergrund: ARM

Die zurzeit wohl wichtigste Prozessorarchitektur am Markt ist Advanced Risc Machine, ARM. Pro Jahr werden etwa 6 Milliarden Prozessoren dieses Typs hergestellt. Damit liegt ihr Marktanteil bei fast 50%. Rund 62% der Cores gehen in den Mobilfunkbereich, wobei ein modernes Smartphone mehrere ARM-Prozessoren beherbergt. ARM-Prozessoren finden sich in beinahe sämtlichen Smartphones oder in Navigationsgeräten.

ARM-Prozessoren haben sich in vielen Bereichen durchgesetzt, da sie hohe Leistung bei gleichzeitig niedrigem Energieverbrauch bieten. Da der Prozessor aus vergleichsweise wenig Transistoren aufgebaut ist, wird auch wenig Chipfläche (Silizium) benötigt. Das wirkt sich positiv auf den Herstellungsprozess und auf den Preis aus.

Die Entwicklung begann 1983, der erste Prozessor, ARM2, wurde 1987 fertiggestellt. Das Design, der Schaltplan also, man spricht auch vom *Core*, wird von der Firma ARM Ltd. (Advanced Risc Machine Ldt.) hergestellt. Lizenznehmer übernehmen dann den Core in eigene Designs. Dabei gibt es Objekt-Lizenzen und Source-Lizenzen. Die Lizenzkosten liegen derzeit bei etwa 10 Cent pro Core.

ARM-Prozessoren tauchen im Markt unter verschiedenen Namen auf. Die Prozessoren der Firma Qualcomm beispielsweise firmieren unter dem Namen Snapdragon, Nvidia vertreibt seine Prozessoren unter dem Namen Tegra. Mit der Zeit haben sich unterschiedliche Architekturen entwickelt, die von ARMv1 bis aktuell (Stand 2013) ARMv8 reichen. ARMv8 ist die 64-Bit-Variante.

Jeweils mehrere Prozessorfamilien implementieren die jeweilige Architektur.

Die aktuelle 32-Bit-Architektur ARMv7 wird beispielsweise durch die Designs Cortex A9 oder auch Cortex A15 realisiert. Die Bezeichnung »A« (Cortex A15) steht für Application. Daneben gibt es auch spezielle Designs für Realtime (»R«) und Mikrocontroller (»M«). ARM bietet zwar direkt keine Floating-Point-Kommandos an, hat aber die Möglichkeit, den Befehlssatz über bis zu 16 Coprozessoren hard- oder auch softwaretechnisch zu erweitern.

ARM-Prozessoren sind ursprünglich 32-Bit-Prozessoren, es gibt aber bereits erste 64-Bit-Versionen, wie beispielsweise im iPhone 5s. Auch die 64-Bit-Version hat 32-Bit breite Befehle, die weitgehend mit dem Befehlssatz A32 identisch sind. Die Befehle bekommen 32 oder 64 Bit breite Argumente übergeben. Adressen sind grundsätzlich 64 Bit breit. Der Adressraum ist erweitert.

Die ARM-Architektur beruht auf einer 3-Register-Maschine. Bei dieser wird das Ergebnis einer Operation, beispielsweise der Addition zweier Register (Variablen), einem dritten Register zugewiesen: `add r1, r2, r3; r1=r2+r3`. In der typischen 32-Bit-Variante gibt es 16 Register. 15 davon sind sogenannte General Purpose Register, das 16. Register ist der Program Counter. Die neue 64-Bit-Variante verfügt über 32 Register.

Der ARM-Befehlssatz bietet in Ergänzung zu bedingten Sprüngen Conditional Instructions, also Befehle, die abhängig von einer Bedingung ausgeführt werden. Neben dem Standardbefehlssatz gibt es abhängig von der eingesetzten Architektur noch weitere Befehlssätze, beispielsweise Thumb beziehungsweise Thumb2, die eine besonders kompakte Codierung ermöglichen.

Der Prozessor unterstützt mehrere Betriebsmodi, unter anderem einen User-Modus, einen Supervisor-Modus, einen Interrupt-Modus und einen Fast-Interrupt-Modus.

Als Hintergrundspeicher (Festplatte, nicht flüchtiger Speicher) findet in eingebetteten Systemen Flash-Speicher Verwendung. Das hat mehrere Vorteile: schneller Zugriff und keine bewegten Teile. Andererseits bringen Flash-Speicher auch Nachteile mit sich: Abhängig von der Technologie können nur Blöcke und nicht einzelne Speicherzellen geschrieben werden und die Anzahl der Schreibzyklen ist endlich.

Hintergrund: Flash-Technologien

Flash-Speicher tauchen in zwei Technologien auf, dem NAND-Flash und dem NOR-Flash. Beide haben unterschiedliche Vor- und Nachteile.

NAND-Flash bietet bis zu eine Million Löschzyklen. Die Speicher sind kompakt und benötigen im Vergleich zu NOR-Speichern 40% weniger Chipfläche. Der Zugriff ist allerdings langsam und das Lesen und Schreiben ist nur blockweise möglich. Ein Block ist typischerweise 15 kByte groß (organisiert in 32 Pages à 512 Byte). NAND-Speicher benötigen ein Bad Block Management, das für defekte Blöcke Ersatzblöcke zur Verfügung stellt.

NOR-Flash hat etwa 100.000 Schreibzyklen, einen schnellen Lesezugriff und kleine Datenmengen lassen sich ebenfalls schnell schreiben. NOR-Flash bietet einen wahlfreien Zugriff auf die Speicherzellen, die Adressierung ist byte- oder wortweise möglich.

Während das Setzen von Bits nur in Blöcken möglich ist (Löschen), können Bits häufig byte- oder wortweise zurückgesetzt werden. Die Konsequenz: Für einen Schreibzugriff müssen erst sämtliche Bits eines Blockes gesetzt werden, danach können die Bits zurückgesetzt werden. Schreiboperationen sind also nur durch eine Kommandofolge möglich.

NOR-Flash benötigt mehr Strom als NAND-Flash und bietet kleinere Speicherkapazitäten.

2.1.2 Software

Die Software eines eingebetteten Systems lässt sich in die Bereiche Systemsoftware und funktionsbestimmende Anwendungssoftware unterscheiden.

Während Deeply Embedded Systems keine oder nur eine schwach ausgeprägte Systemsoftware besitzen, ist diese ein Kennzeichen der Open Embedded Systems.

Die Systemsoftware ihrerseits lässt sich in folgende Komponenten einteilen:

- Firmware (BIOS)
- Bootloader
- Kernel
- Userland

Diese werden im Folgenden vorgestellt.

Firmware und Bootloader

Die Firmware hat die Aufgabe, eine Basisinitialisierung der Hardware vorzunehmen, danach den Bootloader (aus dem Flash) zu laden und schließlich zu starten.

Der Bootloader führt die noch fehlenden Hardwareinitialisierungen durch, insbesondere des Memory-, Interrupt- und Ethernet-Controllers und der Ein-/Ausgabe-Bausteine. Anschließend kopiert er den Kernelcode und das Userland (Rootfilesystem), die beispielsweise im Flash abgelegt sind, in den Hauptspeicher. Moderne Bootloader können Kernelcode und Rootfilesystem auch über tcp/ip von einem Server laden. Danach wird die Programmkontrolle inklusive eventuell notwendiger Startparameter an die Systemsoftware übergeben.

Bootloader, wie beispielsweise »Das U-Boot« enthalten häufig auch Monitorfunktionalitäten, also Operationen, um

- Hauptspeicherzellen zu lesen und zu schreiben,
- Bootparameter zu spezifizieren,
- die Bootquelle (Netzwerk, Flash) auszuwählen,
- das zu bootende Image auszuwählen und
- Recovery durchzuführen (neu flashen).

Wollen Sie einen Bootloader für ein Projekt auswählen, berücksichtigen Sie die folgenden Kriterien bei der Auswahl:

- Unterstützung für die eigene Plattform (CPU, Speicherausbau, Peripherie)
- Codeumfang
- Funktionsumfang (tftp-boot, nfs-boot, flash-boot, Monitorfunktionalität, Scripting-Fähigkeit)
- Lebendigkeit (Wartung, Pflege)
- Verbreitung

Der Bootloader befindet sich meist auf einem eigenen Bereich des Flash-Speichers. Eine Aktualisierung ist nur selten notwendig.

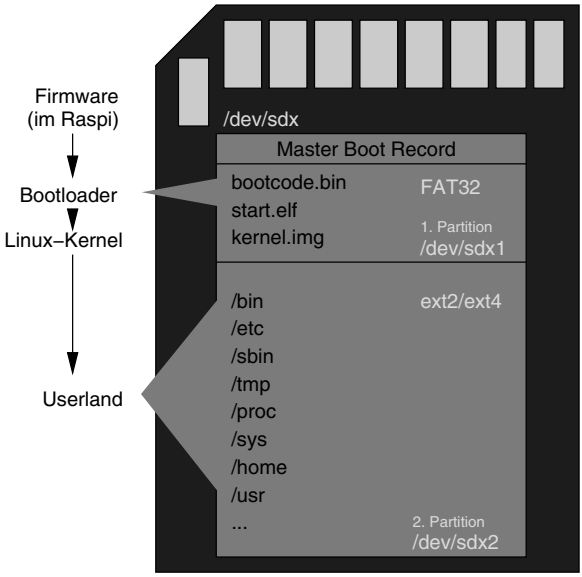


Abb. 2-3
Organisation des Flash-Speichers beim Raspberry Pi

Der Bootloader des Raspberry Pi ist Teil der im Chip integrierten Firmware. Dabei bootet der Raspberry Pi nicht über die normale CPU, sondern über den Grafikcontroller. Die Firmware erwartet einen Flash-Speicher, der im von Microsoft definierten FAT-Fileformat vorbereitet ist. Darauf befindet sich eine Konfiguration in einer Datei mit Namen `config.txt` Außerdem muss dort der Code des eigentlichen Bootloaders abgelegt sein, typischerweise unter dem Namen `bootcode.bin`. Der Bootloader lädt und aktiviert `start.elf`, was den Linux-Kernel lädt (Abb. 2-3).

Kernel

Herzstück der Systemsoftware ist der Betriebssystemkern, auch kurz Kernel genannt. Er stellt über das sogenannte Systemcall-Interface die Dienste, wie beispielsweise das Lesen der Uhrzeit, das Abspeichern oder Einlesen von Daten, das Starten und Beenden von Programmen oder die Übertragung von Daten, zur Verfügung. In Abbildung 2-4 ist zu erkennen, dass der Linux-Kernel vereinfacht aus fünf Blöcken besteht: dem Prozessmanagement, dem Memory Management, dem IO-Management, den Gerätetreibern und dem Systemcall-Interface.

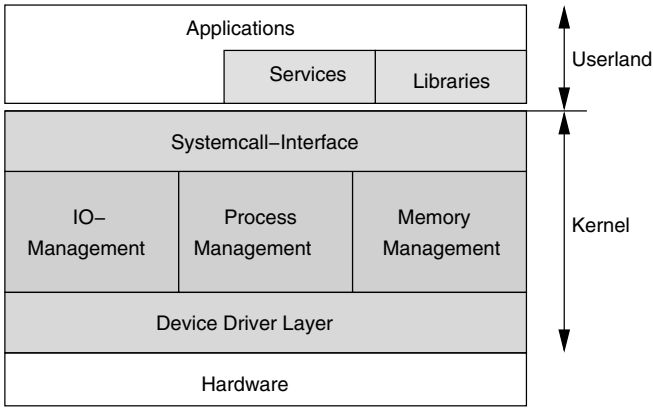


Abb. 2-4
Architektur des Linux-Kernels

Prozessmanagement

Der Linux-Kernel ist für das Multithreading, also für die quasi- und auf Mehrkernmaschinen auch real-parallele Verarbeitung, zuständig, das sogenannte Task-Scheduling. Linux bietet unterschiedliche Scheduling-Verfahren an. Im Bereich eingebetteter Systeme ist dabei insbesondere das prioritätengesteuerte Scheduling relevant, mit dem Realzeiteigenschaften realisiert werden können. Das prioritätengesteuerte Scheduling erlaubt Threads einer Prioritätsebene zuzuweisen. Es ist dann immer derjenige Thread aktiv, der etwas zu arbeiten hat und sich gleichzeitig

auf der höchsten Prioritätsebene befindet. Wie Sie einem Thread eine Prioritätsebene zuweisen, wird in Abschnitt 5.2.2 gezeigt.

Memory Management

Der zweite große Block des Linux-Kernels ist das Memory Management. Auf einem Standardsystem hat es vier Aufgaben:

- Adressumsetzung
- Speicherschutz
- Virtuellen Speicher zur Verfügung stellen
- Erweiterten Speicher zur Verfügung stellen

Für ein eingebettetes System spielt die vierte Aufgabe keine Rolle; typischerweise haben diese Systeme nicht mehr Hauptspeicher, als die CPU adressieren kann. Bei Serversystemen jedoch ermöglicht das Betriebssystem mit diesem Feature die Nutzung von mehr Hauptspeicher als eigentlich aufgrund der Registerbreite von einer Applikation adressiert werden kann.

Grundsätzlich müssen Sie wissen, dass Programme über Adressen auf den Hauptspeicher zugreifen. Diese Adressen werden logische Adressen genannt, der Umfang der logischen Adressen ist der Adressraum. Eine 32-Bit-Linux-Applikation hat dabei normalerweise einen logischen Adressraum von 3 GByte, also von 0x00000000 bis 0xbfffffff. Der Rest (von 0xc0000000 bis 0xffffffff) ist übrigens für den Kernel reserviert.

Dem logischen Adressraum steht ein physischer Adressraum gegenüber. Dem Datenblatt kann man entnehmen, dass der Raspberry Pi mit 512 MByte Hauptspeicher ausgerüstet ist. Der physische Adressraum beginnt damit bei 0x00000000 und reicht bis 0x1fffffff (siehe Abb. 2-5).

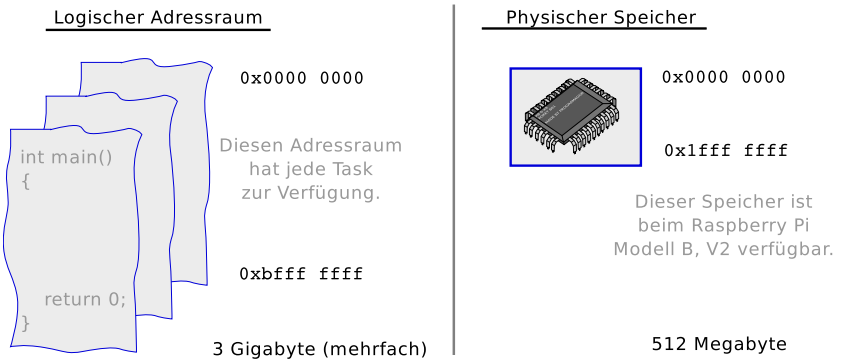


Abb. 2-5 *Unterschied logischer und physischer Adressraum*

Bereits die unterschiedlichen Größen von logischem und physischem Adressraum verdeutlichen, dass beide nicht eins zu eins aufeinander abgebildet werden können. Stattdessen findet eine softwaregesteuerte und durch die Hardware unterstützte Umsetzung der logischen auf die physikalischen Adressen statt, die bei der Kernelprogrammierung (siehe Abschnitt 6.1.2) berücksichtigt werden muss. Das ist die Aufgabe der Adressumsetzung, die zugleich den Speicherschutz realisiert. Durch Letzteren wird verhindert, dass eine Applikation auf Speicherzellen zugreift, die von einer anderen Applikation genutzt werden. Die dritte Aufgabe, virtuellen Speicher zur Verfügung stellen, bedeutet übrigens, dass die Applikation den vollen logischen Adressraum nutzen kann, auch wenn (wie beim Raspberry Pi) physisch deutlich weniger vorhanden ist. Realisiert wird dieses Feature über Swapping, also das Auslagern von Daten vom Hauptspeicher auf den Hintergrundspeicher. Da Swapping aber zeitintensiv ist, wird es im Bereich eingebetteter System nur sehr selten eingesetzt.

IO-Management

Der dritte große Block des Kernels ist das IO-Management. Dieses ist für den Zugriff auf die Peripherie zuständig und damit für ein Embedded System von besonderer Bedeutung. Es realisiert darüber hinaus auch unterschiedliche Dateisysteme. Diese ermöglichen die hierarchische Ablage von Daten in Dateien, die ihrerseits in Verzeichnisse organisiert sind.

Unter Linux ist das klassische Dateisystem ext4. Da Flash-Speicher jedoch besondere Anforderungen beispielsweise bezüglich einer limitierten Anzahl von Schreibzyklen aufweisen, gibt es für den Bereich der eingebetteten Systeme spezielle Filesysteme wie das JFFS2 oder das BTRFS.

Von besonderer Relevanz ist der Zugriff auf die Peripherie. Hier sorgt das IO-Management für ein einheitliches Programmierinterface, unabhängig von der Art der Hardware. Zumindest theoretisch kann damit über die immer gleichen Funktionen (`open()`, `read()`, `write()` und `close()`) auf unterschiedliche Peripherie zugegriffen werden. In der Praxis lässt sich aber immer häufiger komplexe Hardware (zum Beispiel Grafikkarten) nicht mehr auf dieses Interface abbilden.

Außerdem stellt das IO-Subsystem das Treiberinterface zur Verfügung, über das die selbst erstellten Gerätetreiber in den Kernel eingebunden werden (Kapitel 6).

Device-Driver-Layer

Der größte Teil des Kernels sind die Gerätetreiber selbst. Diese lassen sich dynamisch, also während der Kernel bereits aktiv ist, laden. Im Umfeld eingebetteter Systeme, in denen man es weniger häufig mit

wechselnder Hardware zu tun hat, werden die Treiber als Teil des Kernels fest einkompiliert. Auch werden nur die Treiber verwendet, die real zum Einsatz kommen (mehr darüber ebenfalls in Kapitel 6).

Userland

Unter dem Begriff Userland werden die Systemteile zusammengefasst, die für den Betrieb notwendig sind, aber nicht im Kernel liegen. Das sind beispielsweise Bibliotheken oder Programme, mit denen das System (Netzwerk) konfiguriert wird. Hierzu gehören auch die sogenannten Gerätedateien, die die Verbindung zwischen Applikationen und den Treibern herstellen, die die Peripherie ansteuern (siehe Kasten auf Seite 21).

Das Userland befindet sich auf dem Rootfilesystem. Das Rootfilesystem wiederum kann auf dem Flash liegen oder als Image abgelegt werden. Das hat unterschiedliche Vor- und Nachteile, die in Abschnitt 3.5 diskutiert werden.

In Abschnitt 3.2 werden Sie das Userland im Detail kennenlernen, unter anderem indem Sie selbst eines aufbauen.

2.1.3 Auf dem Host für das Target entwickeln

Zwischen der Entwicklung einer Applikation und der Entwicklung eines eingebetteten Systems gibt es elementare Unterschiede. Diese liegen darin begründet, dass das eingebettete System typischerweise weniger Ressourcen zur Verfügung stellt, eine andere Hardware als beispielsweise auf dem PC besitzt und neben der Applikation auch die Systemsoftware eine wesentliche Rolle spielt.

Daher hat man im Embedded-Umfeld typischerweise sowohl eine Host-/Target-Entwicklung als auch eine Cross-Entwicklung.

Host-/Target-Entwicklung

Bei einer Host-/Target-Entwicklung sind zwei Rechner beteiligt. Der Host ist dabei der eigentliche, meist leistungsstarke Entwicklungsrechner. Mit Target wird die Zielhardware, also der Steuerungsrechner des eingebetteten Systems, bezeichnet. Die Host-/Target-Entwicklung ist immer dann notwendig, wenn der Zielrechner leistungsschwach ist. Aber auch fehlende Ein- und Ausgabemöglichkeiten (Tastatur und Bildschirm) oder das Fehlen einer geeigneten Entwicklungsumgebung können eine Host-/Target-Entwicklung erfordern.

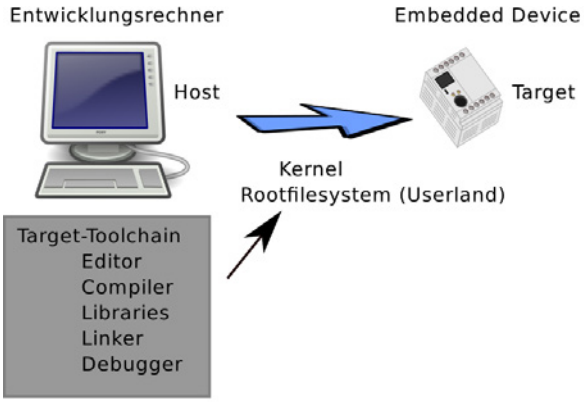


Abb. 2-6 *Host-/Target-Entwicklung*

Ein Problem bei der Host-/Target-Entwicklung ist der Transport der entwickelten Software vom Host zum Target. Im besten Fall existiert eine Netzwerkverbindung. Hier ist insbesondere der Bootloader gefragt. Die zurzeit für den Entwickler beste Lösung stellt `bootp` (respektive `dhcp`) dar. Der Entwickler legt seine Software auf einem Bootp-Server ab, der Bootloader auf dem Target holt sich ohne weitere Interaktion beim Booten die Software per `tftp` ab, legt sie in den Hauptspeicher und aktiviert diese.

Besteht keine Netzwerkverbindung kann alternativ die generierte Software möglicherweise über eine SD-Karte vom Host zum Target gebracht werden. In Deeply Embedded Systems wird hierfür häufig noch die serielle Schnittstelle verwendet, was allerdings nicht nur unhandlich, sondern auch zeitaufwendig ist. Dass zudem auch die eigentliche Systemsoftware auf die Zielhardware gebracht werden muss, macht die Sache nicht einfacher.

Cross-Entwicklung

Eine Cross-Entwicklung wird notwendig, wenn Host und Target unterschiedliche Plattformen repräsentieren. Während ein Hostrechner typischerweise auf einer x86-Architektur basiert, setzen sehr viele eingebettete Systeme auf einen ARM-Prozessor — so auch der Raspberry Pi. Damit läuft der nativ auf dem Entwicklungsrechner erstellte Code nicht auf dem Raspberry Pi. Daher wird auf dem Entwicklungsrechner eine Cross-Development-Toolchain installiert, eine Werkzeugkette also, die ausführbaren Code für die Zielplattform generieren kann.

2.2 Arbeiten mit Linux

Linux — wie andere Betriebssysteme auch — ermöglicht die Abarbeitung mehrerer Tasks auf einem Rechner. Zugleich stellt es den Tasks Dienste wie beispielsweise das Lesen der Uhrzeit, das Schreiben von Daten oder den Zugriff auf die Peripherie zur Verfügung.

Dabei sollten dem Anwender einige grundlegende Konzepte bekannt sein. So lautet eine der fundamentalen Unix-Philosophien: Alles ist eine Datei. Das gilt eben auch für Geräte, also für die Tastatur oder eine USB-Schnittstelle. Die Konsequenz: Anstatt unterschiedliche Schnittstellen kennenzulernen, reicht es für einen Anwender aus, Dateien lesen, schreiben und verbinden zu können. Das Verbinden wird über Pipes realisiert, einem weiteren fundamentalen Konzept. Dabei werden Ausgaben eines Programms über eine Pipe als Eingaben an ein anderes Programm weitergereicht. Wenn Sie beispielsweise wissen wollen, wie viele Dateien sich in einem Verzeichnis befinden, können Sie sich die Dateien mit dem Kommando `ls` anzeigen lassen. Das Kommando `wc` (word count) ist für das Zählen von Zeichen, Wörtern und Absätzen zuständig, die eingegeben werden. Wenn Sie also `ls` mit `wc` über eine Pipe verbinden (`ls | wc`), erhalten Sie das gesuchte Ergebnis.

Linux unterschiedet mehrere Dateitypen, die über Attribute festgelegt werden:

- Normale Dateien (ordinary files)
- Verzeichnisse (Ordner, directories)
- Gerätedateien (Character- und Blockdevices)
- Pipes
- Links (symbolic links, hard links)

Für unser eingebettetes System benötigen wir später unter anderem die Gerätedateien, um die Beziehung zu einem Gerätetreiber, also zur Hardware, herzustellen, und die Links, über die wir auf effiziente Weise unterschiedliche Systemkommandos realisieren.

Gerätedateien

Eine Gerätedatei ist für Anwender das Bindeglied zur Peripherie. Wie auf jede andere Datei kann auf die Gerätedatei mit Kommandos wie `cat` oder `cp` zugegriffen werden. Vom Anwender unbemerkt transferieren die Zugriffsfunktionen aber typischerweise nicht Daten zwischen Applikation und Festplatte, sondern zwischen Applikation und Peripherie, einem Sensor oder Taster beispielsweise.

Der Systemadministrator erkennt eine Gerätedatei anhand des Attributs *Dateityp*. Es werden zwei Typen unterschieden, die Characterdevices und die Blockdevices. Blockdevices werden beim Zugriff auf DVD-Laufwerke, SD-Karten und Ähnlichem benötigt. Intern können Sie immer nur Blöcke von beispielsweise 512 oder 4096 Byte transferieren. Characterdevices transferieren demgegenüber auch einzelne Bytes.

Der Name einer Gerätedatei kann frei gewählt werden. Es ist aber üblich, die Gerätedateien im Verzeichnis `/dev/` anzulegen. Zum Anlegen dient das Kommando `mknod`, das neben dem Namen und dem Typ der Gerätedatei (»b« oder »c«) noch die Nummer des zugehörigen Treibers übergeben bekommt (beispielsweise »mknod /dev/tty1 c 4 1«). Diese Nummer besteht aus zwei Teilen, wobei der erste Major- und der zweite Minornummer genannt wird. Sie dient zur kernelinternen Verwaltung der Gerätedateien.

Das nachfolgende Beispiel für die Ausgabe eines Kommandos »ls -l« zeigt erst eine normale Datei, danach eine weitere, die ein Characterdevice angibt und eine dritte, die ein Blockdevice repräsentiert. Die Art ist direkt am ersten Zeichen der Zeile erkennbar (»-«, »b« oder »c«). Bei den Gerätedateien `tty1` und `sda` sind anstelle der Dateigröße die Major- und Minornummern (4, 1 und 8, 0) gelistet.

```
-rw-r----- 1 syslog adm 260777 Okt  6 19:27 /var/log/syslog
crw-rw---- 1 root   tty   4, 1 Okt  6 09:24 /dev/tty1
brw-rw---- 1 root   disk  8, 0 Okt  6 09:23 /dev/sda
```

In Linux-Systemen gibt es nur einen Verzeichnisbaum, also keine Laufwerke. Filesysteme auf USB-Sticks oder Festplatten werden über Verzeichnisse eingebunden (Abb. 2-7). Auf Desktop-Systemen werden beispielsweise die auf einer separaten Partition abgelegten User-Daten über das Verzeichnis `/home/` zugänglich gemacht, USB-Sticks unter dem Verzeichnis `/media/<name_des_sticks>/`. Das Einhängen eines Filesystems wird als *Mounten* bezeichnet, das Aushängen als *Umounten*.

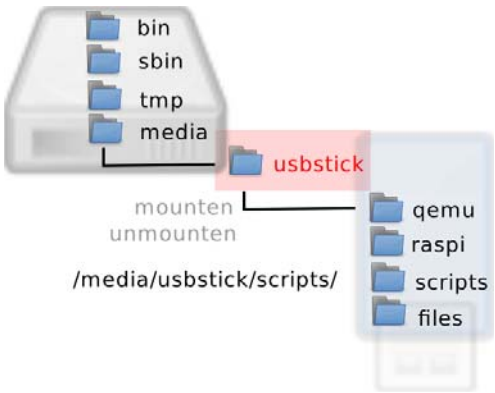


Abb. 2-7 *Externe Speicher werden über Verzeichnisse zugänglich gemacht.*

2.2.1 Die Shell

Bei einer Shell handelt es sich um ein Programm, das Eingaben von der Tastatur entgegennimmt, interpretiert und als Kommandos ausführt. Wir werden die Shell nutzen, um damit Verzeichnisse anzulegen, Dateien anzusehen oder eine Systemgenerierung zu starten. Eine kompakte Einführung in die Linux-Shell finden Sie im Anhang A, *Crashkurs Linux-Shell*.

Für ein eingebettetes System sollten Sie die in Tabelle 2-2 gelisteten, grundlegenden Shellkommandos kennen. Die Systemkommandos aus Tabelle 2-3 repräsentieren typischerweise selbstständige Programme, die wir ebenfalls benötigen. Für vernetzte Systeme schließlich kommen außerdem die in Tabelle 2-4 aufgeführten Kommandos in Betracht.

Tabelle 2-2 *Wichtige Userland-Kommandos*

Kommando	Funktion
ls	Inhalt von Verzeichnissen listen
cp	Dateien kopieren
mv	Dateien umbenennen (verschieben)
echo	Text ausgeben
mkdir	Verzeichnis anlegen
kill	Tasks beenden
ln	Symbolischen Link anlegen
pwd	Name des aktuellen Verzeichnisses ausgeben (print working directory)
rm, rmdir	Dateien bzw. Verzeichnis löschen
chmod	Zugriffsrechte ändern
chown	Besitzverhältnisse ändern
cat	Inhalt einer Datei ausgeben
dd	Dateien blockweise auslesen und abspeichern (disk dump)
ps	Taskliste ausgeben
patch	Quellcode patchen
bunzip2	Dateien dekomprimieren
tar	Daten archivieren
install	Dateien kopieren und mit Rechten ausstatten
reboot	Rechner herunterfahren und neu booten
poweroff	Rechner herunterfahren und stromlos schalten

Tabelle 2-3 *Grundlegende Systemkommandos*

Kommando	Funktion
vi	Editor
tail	Die letzten Zeilen einer Datei ausgeben
find	Filesystem nach verschiedenen Kriterien durchsuchen

➔

Kommando	Funktion
cpio	Dateien archivieren (copy io)
syslog	Prozess zur Protokollierung (Logging)
logrotate	Umfang von Logmeldungen begrenzen
grep	Datei durchsuchen
mount, umount	Filesystem ein- bzw. aushängen
insmod, modprobe	Laden von Treibern und Kernelmodulen
lsmod, rmmod	Listen und Löschen von Treibern und Kernelmodulen
fdisk, gdisk	Anzeigen und modifizieren der Partitionstabelle
mkfs.ext4, mkfs.vfat	Filesysteme anlegen (formatieren)
mknod	Gerätedateien anlegen

Tabelle 2-4
Kommandos für vernetzte Systeme

Kommando	Funktion
ifconfig	Konfiguration des Interface (IP-Adresse usw.)
route, ip	Konfiguration der Routing-Tabelle
boa, lighttpd	Webserver
ftp	FTP-Server
iptables	Firewall konfigurieren
ping	Erreichbarkeit überprüfen
ssh, scp	Remote Shell aufrufen (secure shell)

2.2.2 Die Verzeichnisstruktur

Ein klassisches Linux-Userland hat die in Tabelle 2-5 aufgezeigte Verzeichnisstruktur. Die Verzeichnisse `/proc/`, `/sys/`, `/tmp/` und manchmal auch `/var/` sind virtuelle Verzeichnisse. Ihre Ordner und Dateien werden dynamisch generiert, die Daten liegen real im Hauptspeicher und nicht auf einem Hintergrundspeicher (Festplatte).

Tabelle 2-5
Verzeichnisstruktur eines klassischen Linux-Userlands.

Verzeichnis	Bedeutung
/	Root-Verzeichnis
/bin/	Elementare Systemprogramme
/sbin/	Kommandos für die Systemverwaltung
/boot/	Boot-Dateien, Kernel und initiale RAM-Disk
/dev/	Gerätedateien
/etc/	Konfigurationsdateien
/home/	Persönliche Daten der Nutzer
/root/	Heimatverzeichnis des Superusers (root)
/usr/	Anwendungsprogramme, (Quellcodeverzeichnis /usr/src/)

➔

Verzeichnis	Bedeutung
/lib/	Bibliotheken
/proc/	Informationen über laufende Tasks und über das System
/sys/	Informationen über Geräte und Treiber
/tmp/	Temporäre Daten, werden typischerweise nach dem Reboot gelöscht
/var/	Logdateien, Spooldateien (z.B. E-Mails, Druckjobs), temporäre Daten, Lockdateien
/dev/	Gerätedateien

2.2.3 Editor

Konfigurationen zu Programmen, wie beispielsweise einem Webserver, befinden sich typischerweise in Textdateien, die mit einem Editor bearbeitet werden können. Der Editor wird ebenfalls benötigt, wenn Sie Programmcode oder HTML-Seiten erstellen. Der Editor ist eines der wichtigsten, wenn nicht gar das wichtigste Werkzeug für den Entwickler.

Auf einem Linux-System mit grafischer Oberfläche (Ubuntu) ist zum Editieren von Textdateien das Programm `gedit` empfehlenswert. Es hat viele nützliche Funktionen und die Bedienung ist selbsterklärend. Sie können es auch zur Erstellung der meisten im Rahmen dieses Buches gezeigten Textdateien (Skripte, C-Programme, Konfigurationsdateien, Makefiles und so weiter) einsetzen.

Da es aber auf einem eingebetteten System typischerweise keine grafische Oberfläche gibt, steht der ohnehin ressourcenhungrige `gedit` dort nicht zur Verfügung. Die meisten dieser Systeme liefern eine Variante des Editor-Urgesteins `vi` aus. Das liegt zum einen daran, dass dieser Editor anstelle einer grafischen Oberfläche nur ein Textterminal voraussetzt und zudem überdurchschnittlich leistungsfähig ist. Allerdings ist der vi weniger für Laien als vielmehr für Profis gedacht: Viele unterschiedliche Kommandos ermöglichen das Editieren von Dateien, ohne die Finger von der Tastatur zu nehmen. Selbst Cursortasten werden nicht benötigt. Dafür müssen Sie jedoch viele Kommandos lernen und entsprechend Zeit für die Einarbeitung spendieren; eine Investition, die sich lohnt! Dafür können Sie dann auch spaltenweise editieren und Editierfunktionen direkt auf mehrere Dateien zugleich anwenden. Insbesondere in der auf den Entwicklungsrechnern vorwiegend anzutreffenden Variante vim (vi-Improved) erscheint die Funktionsvielfalt unendlich. Da der vi der einzige Editor ist, der auf allen hier in diesem Buch verwendeten und gebauten Plattformen zur Verfügung steht und auch sonst sehr weit verbreitet ist, sollten Sie zumindest mit den Grundzügen vertraut sein. Eine Einführung in den Editor finden Sie in Anhang B, *Crashkurs vi*.

Wer den Aufwand der Einarbeitung in den vi scheut, sollte sich den Editor `nano` ansehen. Dieser kommt ebenfalls ohne grafische Oberfläche aus und wird direkt auf einem Terminal verwendet. Er hat gegenüber dem vi den Vorteil, dass benötigte Kommandos im unteren Bildschirmbereich eingeblendet sind (siehe Abb. 2-8). Das Hütchen (»^«) steht dabei für die Taste <Strg>. Zum Beenden geben Sie demnach die Tastenkombination <Strg><x> ein. Für das in Kapitel 4 gebaute System können Sie den Editor nano per Konfiguration leicht einbauen.

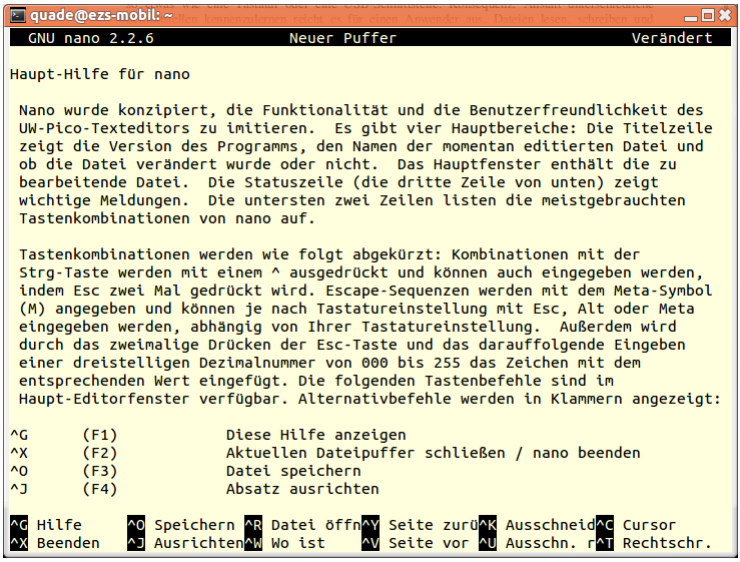


Abb. 2-8
Der konsolenbasierte Editor nano

2.3 Erste Schritte mit dem Raspberry Pi

Der Raspberry Pi ist ein Minicomputer, der mit der Intention entwickelt und hergestellt wurde, die Ausbildung im Bereich Informatik und Mikrocomputertechnik zu fördern. Der kreditkartengroße Computer ist mit 25 $ respektive 35 $ für das Motherboard erstaunlich preiswert, dabei leistungsfähig genug, um als einfacher PC zu fungieren. Außerdem ist er energieeffizient. Last, but not least ist er erfreulich leicht erweiterbar. Vergleichsweise problemlos lassen sich LEDs, Taster, Sensoren oder auch Aktoren ansprechen. Er eignet sich daher sehr gut als Plattform für eingebettete Systeme. Der Raspberry Pi soll — so die Entwickler — Spaß machen. Und das macht er auch.

Den Raspberry Pi gibt es in zwei Varianten. Die mit 25 $ preiswerte Variante A ist mit 256 MByte Hauptspeicher ausgestattet und verzichtet auf die Ethernetschnittstelle. Die Variante B hat neben dem Ethernetport in der Version 2 — die hier eingesetzt wird — 512 MByte Hauptspeicher. Gerade für die Entwicklung ist Variante B attraktiv.

Zum Betrieb eines Raspberry Pi Modell B werden minimal eine Stromversorgung in Form eines Micro-USB-Netzteils und eine SD-Karte mit einer Größe von mindestens zwei GByte benötigt. Damit nutzt man den Rechner über Netzwerk per Remote-Zugang (ssh). Um direkt an dem Gerät zu arbeiten, ist noch eine Tastatur, ein HDMI-fähiger Monitor (HDMI — High Definition Multimedia Interface) und für die grafische Oberfläche eine Maus erforderlich. Tastatur und Maus werden über die beiden USB-Ports angeschlossen. Für ein eingebettetes System werden diese Komponenten aber typischerweise nicht benötigt.

Sinnvoll gerade auch während der Entwicklung ist jedoch eine serielle Schnittstelle, die der Raspberry Pi ebenfalls bietet. Mithilfe eines gut 10 € kostenden USB-Adapters (USB zu seriell) tätigt der Raspberry Pi Ein- und Ausgaben auf jedem PC, der eine USB-Schnittstelle zur Verfügung stellt. Die Auswahl eines Adapters, die Verschaltung auf dem Raspberry Pi und die Konfiguration der benötigten Terminalsoftware ist in Anhang D, *Die serielle Schnittstelle* nachzulesen.

2.3.1 System aufspielen

Um erste Erfahrungen mit dem Raspberry Pi und Linux auf der Konsole zu machen, bietet es sich an, das System mit dem bereits vorkonfigurierten, debianbasierten System Raspbian zu booten. Dazu ist vor der Inbetriebnahme die SD-Karte mit einem Image zu flashen.

Hierzu gibt es im Internet, beispielsweise unter [http://elinux.org/RPi_Easy_SD_Card_Setup] sehr detaillierte Anleitungen für unterschiedliche Betriebssysteme. An dieser Stelle reicht es daher aus, die Vorbereitung der SD-Karte ausgehend von einer Linux-Konsole zu demonstrieren.

1. Laden Sie eine aktuelle Version des Raspbian-Images unter [http://www.raspberrypi.org/downloads] herunter. Auf einer Konsole können Sie dazu `wget` verwenden. Das Image ist ausgepackt typischerweise 2 GByte groß. Anhand des Hashwertes, den Sie auf der Download-Seite finden, stellen Sie sicher, dass der Download erfolgreich war. Nach dem Download sollten Sie das komprimierte Image per `unzip` entpacken.

```
quade@felicia:~> wget \
http://downloads.raspberrypi.org/images/raspbian/
2013-07-26-wheezy-raspbian/2013-07-26-wheezy-raspbian.zip
...
quade@felicia:~> sha1sum 2013-07-26-wheezy-raspbian.zip
f072b87a8a832004973db4f5e1edb863ed27507b
2013-07-26-wheezy-raspbian.zip
quade@felicia:~> unzip 2013-07-26-wheezy-raspbian.zip
...
```

2. Lassen Sie sich die Liste der aktuell verfügbaren Laufwerke durch Eingabe von `lsblk` auf einem Terminal im Entwicklungsrechner ausgeben. Stecken Sie danach die SD-Karte in den Kartenleser des Entwicklungsrechners und rufen Sie erneut `lsblk` auf. In Abbildung 2-9 ist beim zweiten Aufruf des Kommandos, also nach dem Einstecken der SD-Karte, die neue Disk `/dev/sdb` mit den beiden Partitionen `/dev/sdb1` und `/dev/sdb2` aufgetaucht. Wichtig: In der Spalte »RM« muss eine »1« stehen, die signalisiert, dass es sich um eine »removable«, also eine entfernbare und nicht um eine festeingebaute Disk beziehungsweise Partition handelt. Der Laufwerksbuchstabe — um diese Terminologie zu benutzen — ist »b«. Alternativ können Sie die SD-Karte ebenso identifizieren, wenn Sie nach dem Einstecken `dmesg` oder `tail /var/log/syslog` aufrufen. Unmounten Sie alle neu eingehängten Laufwerke wieder, indem Sie mit Root-Rechten versehen das Kommando `umount` mit dem Namen der Gerätedatei als Parameter aufrufen. In unserem Fall wäre das »umount /dev/sdb1 /dev/sdb2«, oder allgemein ausgedrückt (sie müssen das nachfolgende Kommando also anpassen):

```
quade@felicia:~> sudo umount /dev/sdx1
quade@felicia:~> sudo umount /dev/sdx2
```

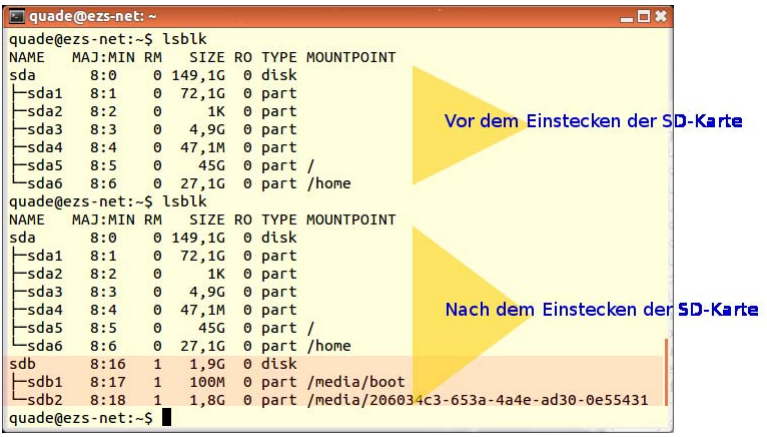


Abb. 2-9
Mithilfe des Kommandos `lsblk` lässt sich die Gerätedatei identifizieren.

3. Das Image wird auf die Karte mithilfe von `dd` geschrieben. `dd` kopiert von der Eingabedatei (infile, if) blockweise die Daten auf die Ausgabedatei (outfile, of). ACHTUNG: Alle auf der Karte bisher abgelegten Daten gehen hierdurch verloren! Ersetzen Sie im folgenden Beispiel »/dev/sdx« durch die im zweiten Schritt identifizierte Gerätedatei. Arbeiten Sie sehr sorgsam, nicht dass Sie versehentlich den Inhalt der Festplatte Ihres Entwicklungsrechners überschreiben!

```
quade@felicia:~> sudo dd if=2013-07-26-wheezy-raspbian.img \
   of=/dev/sdx bs=4k
```

Das Schreiben des 2 GByte großen Images kann eine Weile Zeit in Anspruch nehmen.

4. Die SD-Karte ist jetzt vorbereitet und kann im Raspberry Pi ausprobiert werden.

2.3.2 Startvorgang

Der Raspberry Pi hat keinen Ein-/Ausschalter. Durch Anstecken der Spannungsversorgung bootet der Minicomputer von der SD-Karte das vorbereitete System »Raspbian«. Ist der Rechner mit einem (lokalen) Netzwerk versehen, versucht er per DHCP eine IP-Adresse zu bekommen. Haben Sie den Rechner per HDMI mit einem Monitor verbunden oder über eine serielle Schnittstelle mit einem anderen Rechner (Anhang D, *Die serielle Schnittstelle*), sollten Sie die Bootmeldungen sehen. Sobald die Aufforderung zum Login kommt, können Sie sich mit dem Loginnamen »pi« und dem Passwort »raspberry« einloggen.

Falls sich der Raspberry Pi im lokalen Netz befindet, können Sie sich sogar mit diesem verbinden, wenn kein Monitor und keine serielle Schnittstelle angeschlossen sind. Typischerweise hat er in diesem Fall von einem DHCP-Server eine IP-Adresse bekommen, die Sie nur herausfinden müssen. Hierbei kann Ihnen unter Umständen das Programm `nmap` helfen (siehe Kasten auf Seite 29).

Raspberry Pi per nmap finden

Installieren Sie, falls nicht schon vorhanden, nmap. Auf einem Ubuntu können Sie in einem Terminal hierfür das Kommando `sudo apt-get install nmap` verwenden. Identifizieren Sie Ihre eigene IP-Adresse und Subnetzmaske mithilfe von `ifconfig | grep -i "inet Adr"`. Neben der Localhost-Adresse »127.0.0.1« sollten Sie mindestens eine weitere Adresse inklusive der Subnetzmaske sehen. Im Beispiel ist das die Adresse »192.168.178.48« mit der Maske »255.255.255.0«.

Die Maske muss für `nmap` in die CIDR-Form (Classless Inter Domain Routing), also die Bitanzahl der Netzadresse, umgerechnet werden. Das absolvieren Sie, indem Sie jede der vier Zahlenwerte in ein Bitformat umrechnen und dann die Einsen zählen. Die 255 (Hexadezimal 0xff) entspricht dabei »1111 1111« also acht Einsen. Die Netzmaske »255.255.255.0« wird demnach im CIDR-Format als »/24« dargestellt. Starten Sie `nmap` unter Angabe der Netzadresse mit angehängter Netzmaske.

```
quade@felicia:~$ ifconfig | grep -i "inet Adr"
    inet Adresse:192.168.178.48  Bcast:192.168.178.255
         Maske:255.255.255.0
    inet Adresse:127.0.0.1  Maske:255.0.0.0
quade@felicia:~$ nmap 192.168.178.0/24
```

Das Programm `nmap` listet alle Rechner, die im Netz erreichbar sind, der Raspberry Pi müsste darunter sein. Gehen Sie die Liste durch. Der Raspberry Pi dürfte wie hier zu sehen ist, nur einen ssh-Port offen haben:

```
...
Nmap scan report for 192.168.178.45
Host is up (0.0020s latency).
Not shown: 999 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
MAC Address: B8:27:EB:F6:50:2A (Unknown)
```

Loggen Sie sich mithilfe von `ssh`, dem Usernamen »pi« und dem Passwort »raspberry« auf dem Gerät ein. Gelingt Ihnen das nicht, probieren Sie einen anderen der gelisteten Rechner aus.

2.3.3 Einloggen und Grundkonfiguration

Haben Sie Kontakt mit dem Raspberry Pi per Monitor und Tastatur oder über die serielle Schnittstelle, können Sie sich mit dem Usernamen »pi« und dem Passwort »raspberry« anmelden (einloggen). Falls Sie sich über eine grafische Oberfläche einloggen, sollten Sie zunächst ein Terminalfenster starten.

Ansonsten verbinden Sie sich per `ssh` mit dem Raspberry Pi über dessen IP-Adresse. Das Standardpasswort lautet auch hier »raspberry« und wird beim Eintippen aus Sicherheitsgründen nicht angezeigt:

```
quade@felicia:~$ ssh pi@192.168.178.45
pi@192.168.178.45's password: [raspberry]
...
```

Um die Grundkonfiguration durchzuführen, starten Sie als Superuser das Programm `raspi-config`. Konfigurieren Sie insbesondere Ihre Tastatur (bei diesem Schritt müssen Sie geduldig sein), ändern Sie das Passwort und setzen Sie die Zeitzone.

```
pi@raspberrypi ~ $ sudo raspi-config
```

2.3.4 Hello World: Entwickeln auf dem Raspberry Pi

Mit der Distribution *Raspbian* ist das Entwickeln von Programmen, die beispielsweise auf der Programmiersprache C beruhen, auf dem Rasp-

berry Pi sehr einfach. Die wichtigsten Werkzeuge wie Editor, Compiler, Linker und die Libraries sind bereits installiert.

Probieren Sie jetzt noch abschließend aus, ein erstes Programm auf dem Raspberry Pi zu kompilieren und zu testen. Editieren Sie beispielsweise per `vi hello.c` oder `nano hello.c` ein kurzes C-Programm (Beispiel 2-1). Rufen Sie den Compiler per `make` auf. Beseitigen Sie eventuell gemeldete Fehler und starten Sie das Programm, nachdem dieses vollständig kompiliert werden konnte. Zum Starten ist es aus Sicherheitsgründen notwendig, vor dem Namen des Kommandos »./« einzugeben. Beispiel 2-2 zeigt die Befehle, die auf dem Raspberry Pi über eine Shell einzugeben sind.

```
#include <stdio.h>

int main( int argc, char **argv, char **envp )
{
    printf("Hello World\n");
    return 0;
}
```

Beispiel 2-1
Hello World <hello.c>

```
pi@raspberrypi ~ $ vi hello.c
pi@raspberrypi ~ $ make hello
cc     hello.c   -o hello
pi@raspberrypi ~ $ ./hello
Hello World
pi@raspberrypi ~ $
```

Beispiel 2-2
Befehlssequenz zum Generieren auf dem Raspberry Pi

3 Embedded von Grund auf

Eingebettete Systeme als integrierte, mikroelektronische Steuerungen bestehen neben Hardware aus einer Firmware (BIOS), aus einem Bootloader, einem Kernel, einem Rootfilesystem mit Systemprogrammen und funktionsbestimmenden Applikationen (Abb. 3-1).

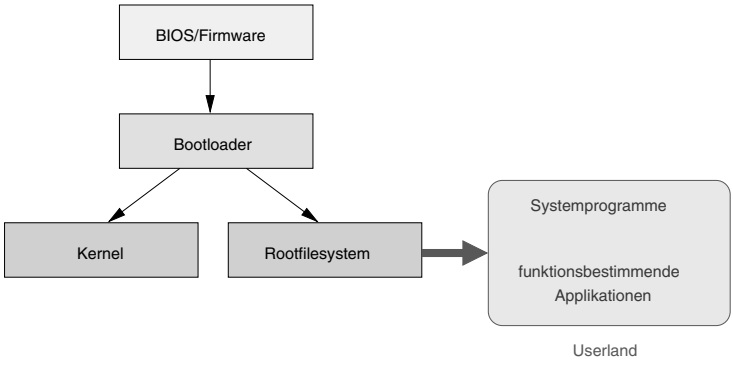


Abb. 3-1
Komponenten des eingebetteten Systems

Um diese Komponenten kennenzulernen, bauen wir Schritt für Schritt von Grund auf ein System zusammen. Wir starten bei den wichtigsten Komponenten, dem Kernel und dem Rootfilesystem mit den Systemprogrammen (Userland). Diese Komponenten werden zunächst im Emulator getestet, was typischerweise ohne Komplikationen verläuft. Im Anschluss soll das System um eine einfache Anwendung, einen Webserver, erweitert werden. Läuft das System, wird es in Abschnitt 3.3 auf die Raspberry Pi-Hardware übertragen.

Emulator

Als Emulator bezeichnet man ein System, das ein anderes System, beispielsweise einen Computer, nachbildet. Der nachgebildete Computer kann dabei eine komplett andere Architektur aufweisen, da typischerweise sowohl die CPU als auch die Peripherie (Tastatur, Grafik, Netzwerk) emuliert werden. Es muss damit nicht zwangsweise die Hardware für ein spezifisches System vorhanden sein, um dennoch dafür Software entwickeln, testen oder einfach nur betreiben zu können.

Läuft der Emulator auf einem leistungsfähigen Hostsystem, ist das emulierte System unter Umständen sogar schneller als das Original; ansonsten leider langsamer. Außerdem unterstützt der Emulator die System- und Softwareentwicklung durch erweiterte Debugmöglichkeiten und den einfachen Transfer von entwickelter Software zwischen Host und Target.

Ein im Umfeld eingebetteter Systeme auf einem PC häufig eingesetzter Emulator ist der auch hier verwendete Qemu, der viele unterschiedliche Architekturen und diverse Peripheriekomponenten emulieren kann.

Öffnen Sie als Vorbereitung für die nachfolgenden Tätigkeiten auf dem Entwicklungsrechner ein Terminalfenster. Auf einem Ubuntu geht das sehr einfach durch gleichzeitiges Drücken der Tasten <Strg><Shift><T>. Legen Sie dann mithilfe des Kommandos `mkdir` das Verzeichnis `embedded` an, unter dem wir sämtliche Arbeiten durchführen werden. Beachten Sie, dass Sie auf Ihrem Heimatverzeichnis rund 15 GByte freien Plattenplatz benötigen. Sie können auch gleich die Unterverzeichnisse `qemu` und `raspi` anlegen. Das erstere ist für die Dateien des emulierten Systems gedacht, `raspi` für alles, was später auf dem Raspberry Pi laufen soll.

```
quade@felicia:~> mkdir embedded
quade@felicia:~> cd embedded
quade@felicia:~/embedded>
quade@felicia:~/embedded> mkdir qemu
quade@felicia:~/embedded> mkdir raspi
```

3.1 Der Linux-Kernel

Um einen eigenen Kernel zu bauen, sind die folgenden Schritte notwendig:

1. Quellcode herunterladen und Integrität überprüfen
2. Eventuell patchen
3. Kernel konfigurieren
4. Kernel generieren

Der Betriebssystemkern, Kernel genannt, stellt den Applikationen Dienste, wie beispielsweise das Speichern von Daten oder das Lesen einer Uhrzeit, zur Verfügung. Intern ist er aus den Blöcken Systemcall-Interface, Task-Management, Memory Management, IO-Management und Gerätetreibern aufgebaut (siehe Abschnitt "Kernel", Seite 16).

Während auf Desktop-Systemen universelle Kernel eingesetzt werden, die mit möglichst jeder Hardware zurechtkommen, werden im Bereich eingebetteter Systeme auf die Hardware angepasste Kernel ver-

wendet. Dazu ist es nach dem Herunterladen des Kernelquellcodes häufig notwendig, den Kernel zunächst noch zu patchen, danach selbst zu konfigurieren und zu kompilieren.

Im Folgenden erstellen wir einen Kernel, der in unserem selbst gebauten System eingesetzt wird.

Quellcode installieren

Der Quellcode zum Linux-Kernel findet sich auf kernel.org ([http://www.kernel.org]) und kann von dort als komprimiertes Archiv heruntergeladen werden. Um den Quellcode einer spezifischen Version zu erhalten, gehen Sie auf den Link [http://www.kernel.org/pub/linux/kernel/]. Hier gibt es für jede Hauptversion ein Verzeichnis, in dem die einzelnen Kernel, Versions-Patches und die zugehörigen, digitalen Unterschriften abgelegt sind. Den Quellcode zum Kernel 3.10.9 beispielsweise finden Sie im Unterverzeichnis `V3.x/` unter dem Namen `linux-3.10.9.tar.gz`, `linux-3.10.9.tar.bz2` oder `linux-3.10.9.tar.xz`. Es handelt sich jedes Mal um das gleiche Archiv (tar), das jedoch mit jeweils einem anderen Verfahren komprimiert ist. Die schlankeste Variante ist die mit der Dateierweiterung »xz«. Die zum Archiv gehörende digitale Unterschrift, mit der die Integrität und die Authentizität des Codes überprüft werden kann, befindet sich in der Datei `linux-3.10.9.tar.sign`. Die Patchdatei, um vom Kernel mit der Version 3.10 auf Version 3.10.9 zu patchen, heißt `patch-3.10.9.tar.gz`, `patch-3.10.9.tar.bz2` beziehungsweise `patch-3.10.9.tar.xz`, die zugehörige Signatur ist `patch-3.10.9.tar.sign`. Wir verwenden in unserem System diesen Kernel, weil er zurzeit der Bucherstellung aktuell ist. Um den Kernelquellcode herunterzuladen, benötigen Sie keinen Browser. Auf der Kommandozeile können Quellcode und eventuell gewünschte Patchdateien auch per `wget` (am besten direkt aus dem Unterverzeichnis `embedded/qemu/` heraus) heruntergeladen werden:

```
cd ~/embedded/qemu
wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.10.tar.xz
wget http://www.kernel.org/pub/linux/kernel/v3.x/linux-3.10.tar.sign
wget http://www.kernel.org/pub/linux/kernel/v3.x/patch-3.10.9.xz
wget http://www.kernel.org/pub/linux/kernel/v3.x/patch-3.10.9.sign
```

Eigentlich könnten Sie direkt die Kernelversion 3.10.9 herunterladen. Um den Vorgang des Patchens zu demonstrieren, wählen wir jedoch den komplizierteren Weg. Bei diesem verwenden wir die Grundversion 3.10 des Kernels und den Patch auf die Version 3.10.9. Für den einfacheren Weg laden Sie direkt per `wget` die Version linux-3.10.9 herunter, packen diese wie hier beschrieben aus und überspringen dann den Abschnitt „Patchen''.

Die digitale Unterschrift zu den Quellcode- beziehungsweise Patchdateien sollte unbedingt mit jedem Download überprüft werden! Dazu ist es in einem ersten Schritt notwendig, die Unterschriftenprobe auf dem Entwicklungsrechner zu installieren. Kernel sind meistens von Linus Torvalds unterschrieben, die Patches von Greg Kroah-Hartman. Zurzeit lautet die Zertifikats-ID von Torvalds `00411886`, die von Kroah-Hartman `6092693E`. Die Unterschriftenprobe, das Zertifikat also, kann per `gpg` mit der Option `--recv-keys` installiert werden:

```
quade@felicia:~> gpg --keyserver hkp://keys.gnupg.net \
   --recv-keys 00411886 6092693E
```

Danach sollten Sie noch mithilfe des Kommandos `gpg --edit-key 00411886` und des Unterkommandos `trust` die Vertrauensstufe einstellen. Ansonsten erfolgt bei den nachfolgenden Kommandos die folgende Warnung, die bei entsprechend konfigurierter Vertrauensstufe (»absolutes Vertrauen«) unterbleibt:

```
gpg: WARNUNG: Dieser Schlüssel trägt keine vertrauenswürdige Signatur!
gpg:          Es gibt keinen Hinweis, dass die Signatur wirklich dem
vorgeblichen Besitzer gehört.
Haupt-Fingerabdruck  = ABAF 11C6 5A29 70B1 30AB  E3C4 79BE 3E43 0041 1886
```

Bevor die Integrität und die Authentizität überprüft werden können, muss der Download dekomprimiert werden. Je nach Kompressionstyp setzen Sie dazu `gzip`, `bunzip2` oder `xz` ein. Zur Überprüfung der Integrität und Authentizität verwenden Sie `gpg` mit der Option `--verify`:

```
quade@felicia:~/embedded/qemu> xz -d linux-3.10.tar.xz
quade@felicia:~/embedded/qemu> gpg --verify linux-3.10.tar.sign
gpg: Unterschrift vom Mo 01 Jul 2013 00:47:38 CEST mittels
RSA-Schlüssel ID 00411886
gpg: Korrekte Unterschrift von »Linus Torvalds
<torvalds@linux-foundation.org>«
quade@felicia:~/embedded/qemu>
quade@felicia:~/embedded/qemu> xz -d patch-3.10.9.xz
quade@felicia:~/embedded/qemu> gpg --verify patch-3.10.9.sign
gpg: Unterschrift vom Mi 21 Aug 2013 00:41:13 CEST mittels
RSA-Schlüssel ID 6092693E
gpg: Korrekte Unterschrift von »Greg Kroah-Hartman (Linux kernel
stable release signing key) <greg@kroah.com>«
quade@felicia:~/embedded/qemu>
```

Meldet `gpg`, dass es den öffentlichen Schlüssel (public key, Unterschriftenprobe) nicht gefunden hat, kann der Schlüssel anhand der in der Ausgabe ersichtlichen Schlüssel-ID, wie bereits gezeigt, gesucht und importiert werden (Option `recv-keys`).

Normalerweise wird der (dekomprimierte) Quellcode des Linux-Kernels im Verzeichnis `/usr/src/` ausgepackt. Da wir den Quellcode

aber nicht für den Entwicklungsrechner, sondern für das eingebettete System (Target) benötigen, packen wir diesen lieber in einem eigenen Verzeichnis wie beispielsweise `embedded/qemu/` unterhalb unseres Heimatverzeichnisses aus.

```
quade@felicia:~/embedded/qemu/> tar xvf linux-3.10.tar
...
```

Patchen

In vielen Fällen ist es notwendig, Quellcode zu patchen. Per Patch lassen sich Funktionen nachrüsten (mit dem PREEMPT_RT-Patch beispielsweise vollständige Realzeitunterstützung) und Fehler bereinigen.

Ein Quellcode-Patch ist im Linux-Umfeld typischerweise eine Datei, die mit dem Programm `diff` erstellt wurde und die Änderungen zwischen zwei Versionen enthält. Im Patch ist beschrieben, an welchen Stellen Zeilen zu löschen und welche Zeilen hinzuzufügen sind. Die relevanten Dateien sind inklusive der relativen Pfade angegeben.

Ist ein Patch komprimiert, wie das für den Linux-Kernel typisch ist, muss er zunächst dekomprimiert werden. Danach muss man in die Verzeichnishierarchie wechseln, von der aus der Patch erstellt wurde. Im Fall des Linux-Kernels wäre das beispielsweise `/usr/src/`. Da sich jedoch die Namen der Verzeichnisse, die den Quellcode enthalten, von Entwickler zu Entwickler unterscheiden und damit die Pfadangaben in der Patchdatei nicht brauchbar sind, wechselt man häufig in das eigentliche Quellcodeverzeichnis und weist das Patchprogramm `patch` per Kommandozeilenoption »-p 1« an, den ersten Teil der Pfadangaben zu überspringen. Da `patch` den Input von der Standardeingabe erwartet, wird per Eingabeumlenkung (»<«) der Inhalt der Patchdatei zugeführt. Um den Linux-Kernel zu patchen, kann man also folgendermaßen vorgehen:

```
quade@felicia:~/embedded/qemu> cd linux-3.10/
quade@felicia:~/embedded/qemu/linux-3.10> patch -p1 <../patch-3.10.9
...
quade@felicia:~/embedded/qemu/linux-3.10> cd ..
quade@felicia:~/embedded/qemu> mv linux-3.10 linux-3.10.9
```

Kernel konfigurieren

Der Kernel eines eingebetteten Systems zeichnet sich dadurch aus, exakt auf die Hardware angepasst zu sein und nur die wirklich notwendigen Komponenten zu beinhalten.

Der Kernel stellt zur Konfiguration und zur Generierung ein auf `make` basierendes Kernel-Build-System (KBS) zur Verfügung. Dieses bietet für die unterschiedlichen Aufgaben diverse Targets. Für die Kernelkonfiguration sind dies die in Tabelle 3-1 angegebenen. Target ist im Kontext des Generierungswerkzeugs `make` der Fachbegriff für »Generie-

rungsziel«. Kann man beispielsweise mit `make` eine Release- und eine Debugversion erzeugen, gibt es zwei Targets (`release` und `debug`). Die Kernelkonfig selbst befindet sich übrigens in der (versteckten) Datei `.config`.

Tabelle 3-1 *Targets für die Kernelkonfiguration*

Target	Bedeutung
config, nconfig, menuconfig, xconfig, gconfig	Aktualisiert die aktuelle Konfiguration interaktiv mit verschiedenen Oberflächen/Interfaces
oldconfig	Aktualisiert die aktuelle Konfiguration
localmodconfig	Erstellt eine Konfiguration, die nur die gerade geladenen Module enthält
localyesconfig	Erstellt eine Konfiguration, die die geladenen Module als festen Kernelbestandteil übernimmt
silentoldconfig	Wie oldconfig, nur ohne Ausgaben
defconfig	Erstellt eine neue Konfiguration gemäß einem architekturspezifischen Template
savedefconfig	Speichert die aktuelle Konfiguration als Default-Config (`defconfig`)
allyesconfig	Neue Konfiguration, alle Optionen sind mit »yes« aktiviert (Maximalkonfiguration)
allnoconfig	Neue Konfiguration, alle Optionen sind mit »no« aktiviert (Minimalkonfiguration)
allmodconfig	Neue Konfiguration, wenn möglich werden Optionen als Module ausgeprägt
alldefconfig	Neue Konfiguration, alle Optionen werden gemäß Defaulteinstellung übernommen
randconfig	Neue Konfiguration, Optionen werden per Zufall aktiviert oder deaktiviert
listnewconfig	Zeigt neue Optionen an
olddefconfig	Wie silentoldconfig, neue Optionen werden gemäß Defaulteinstellung übernommen

Gerade für ein eingebettetes System verwendet man optimal angepasste Softwarekomponenten. Das trifft auch auf den Kernel zu, der nicht mehr Teile enthalten sollte, als tatsächlich benötigt werden.

Für einen solchen minimalen Kernel wird zunächst das Target `allnoconfig` ausgewählt. Anschließend werden die für den Betrieb des Systems notwendigen Komponenten mithilfe des Targets `menuconfig` aktiviert (siehe Abb. 3-2). Damit das Target `menuconfig` funktioniert, ist eventuell zuvor mit Root-Rechten versehen per `apt-get` das Paket `libncurses-dev` zu installieren:

```
quade@felicia:~/embedded/qemu> sudo apt-get install libncurses-dev
[sudo] password for quade:
...
quade@felicia:~/embedded/qemu> cd linux-3.10.9/
```

```
quade@felicia:~/embedded/qemu/linux-3.10.9> make allnoconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
scripts/kconfig/conf --allnoconfig Kconfig
#
# configuration written to .config
#
quade@felicia:~/embedded/qemu/linux-3.10.9> make menuconfig
...
```

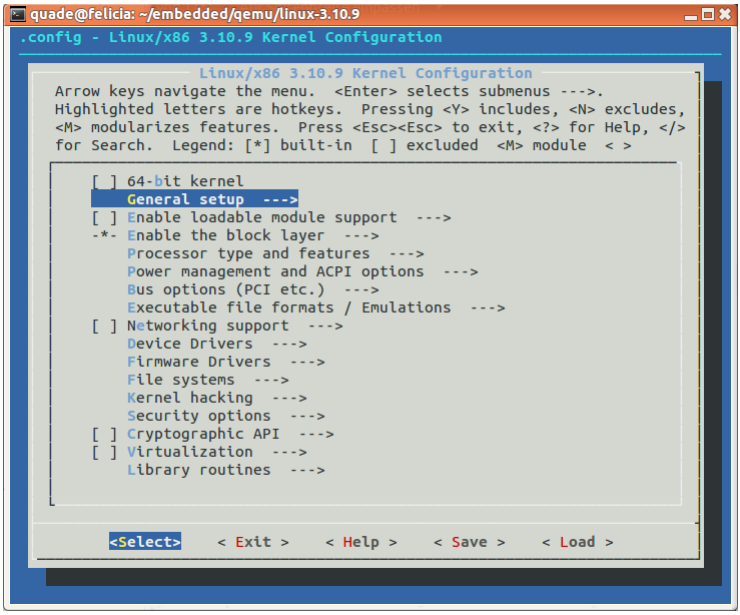


Abb. 3-2
Oberfläche zur Kernelkonfiguration

Jetzt wählen Sie den Prozessor aus, den Systembus, das Executable-Format für Programme, die Netzwerkunterstützung inklusive Treiber für Netzwerkkarte und die Unterstützung für Festplatten inklusive Dateisystem. Alle Komponenten werden fest in den Kernel eingebunden, auf Module verzichten wir, da das Embedded System ansonsten die Programme zum Laden und Entladen der Module benötigen würde.

Setzen Sie also für das erste von uns gebaute System konkret die in Tabelle 3-2 gezeigten Optionen, die für ein 64-Bit-Entwicklungssystem gelten. Sollten Sie eine andere Kernelversion verwenden, können die Optionen variieren, ebenso, wenn Sie den Kernel für eine andere Platt-

form generieren. In diesem Fall müssen Sie mit Geduld die notwendigen Optionen anpassen.

Tabelle 3-2 *Notwendige Kerneloptionen (Kernel 3.10.9, 64 Bit)*

Hauptmenü	1. Untermenü	2. Untermenü
64-bit kernel		
Bus options	PCI	
Executable File Formats	Kernelsupport for Elf-binaries	
Networking Support	Networking Options	TCP/IP networking
		Network packet filtering framework
		802.1d Ethernet bridging
Device Drivers	SCSI device support	SCSI device support
		SCSI disk support
		SCSI generic support
	Serial ATA and Parallel ATA drivers	[ATA SFF support] [ATA BMDMA support] [Intel ESB, ICH, PIIX3, PIIX4 PATA/SATA support]
	Network Device Support	Universal TUN/TAP device driver support
		[Ethernet Driver Support] [National Semiconductor Devices][PCI NE2000 and clones support (see help)]
File systems	Second extended fs support	
	Pseudo Filesystems	tmpfs Virtual memory file system support (former shm fs)

Schauen Sie sich die vielfältigen Konfigurationsmöglichkeiten in Ruhe an. Speichern Sie zum Schluss die Konfiguration über die Option [Exit]. In diesem Fall speichert der Kernel die Daten in der (versteckten) Datei mit dem Namen `.config`, aus der das Kernel-Build-System später für die Generierung die Konfiguration verwendet.

Kernel generieren

Ohne Angabe eines Targets generiert `make` den Kernel. Auf Multicore-Maschinen ist die Option »-j« relevant, der man als Parameter die Anzahl der zur Verfügung stehenden CPU-Cores mitgibt. Durch die damit angestoßene parallele Verarbeitung verkürzt sich die Generierungszeit signifikant.

```
quade@felicia:~/embedded/qemu/linux-3.10.9>make -j 4
...
```

Den fertigen Kernel finden Sie unter `arch/<architektur>/boot/<kernelname>`, also für eine x86-Architektur unter `arch/x86/boot/bzImage`. Wir

könnten ihn zwar bereits mithilfe des Emulators Qemu testen, bauen aber zuvor erst noch ein geeignetes Userland.

Target	Bedeutung
all	Generiert Kernel plus Module
vmlinux	Generiert einen unkomprimierten Kernel
modules	Generiert die Kernelmodule
uimage	Generiert einen Kernel mit Metainformationen für den Bootloader »Das U-Boot«

Tabelle 3-3 *Wichtige Targets für die Kernelgenerierung*

3.2 Das Userland

Die für den Betrieb des eingebetteten Systems notwendigen Dateien, also System- und Applikationsprogramme und Konfigurationsdateien, sind in einer Verzeichnisstruktur auf dem Rootfilesystem abgelegt. Dieses repräsentiert das Userland, das sich im Bereich eingebetteter Systeme deutlich von seinem Pendant der Standardsysteme unterscheidet.

Im Userland ist Software auf Systemebene von funktionsbestimmenden Applikationen zu unterscheiden. Letztere sind zentral, implementieren sie doch die Funktionen, für die das Gerät überhaupt konstruiert und eingesetzt wird. Software auf Systemebene kann einer funktionsbestimmenden Applikation Dienste zur Verfügung stellen, sie kann für die Verwaltung des Systems oder für das Update-Management zuständig sein.

In eingebetteten Systemen stehen den Applikationen typischerweise nur eingeschränkte Umgebungen zur Verfügung. So fehlt zuweilen ein Framework für gemeinsam genutzte Bibliotheken (Shared-Libs, DLLs). Daher werden Programme statisch gelinkt oder es wird ein sogenanntes Multicall-Binary eingesetzt.

Multicall-Binary

Als Multicall-Binary wird eine einzelne Applikation bezeichnet, die funktional mehrere andere Programme ersetzt. Dazu wird das Multicall-Binary mithilfe symbolischer Links unter verschiedenen Programmnamen im Filesystem abgelegt. Das Multicall-Binary erkennt anhand des Namens, unter dem es aufgerufen wurde, welche Funktion es erbringen soll.

Wie in Abbildung 3-3 zu sehen ist, sparen Multicall-Binaries insbesondere dann Ressourcen, wenn keine Shared Libraries installiert sind. Würden die unterschiedlichen Kommandos, wie auf dem Desktop üblich, als eigenständige Programme realisiert, würde jedes die Standardfunktionen wie beispielsweise `printf` dazu binden. Das benötigt sowohl auf der Festplatte als auch nach dem Start im Hauptspeicher unnötig Ressourcen. Beim Multicall-Binary werden diese Funktionen einmal dazu gebunden und stehen dann allen implementierten Kommandos zur Verfügung.

Ein Multicall-Binary ist typischerweise auch schlank und schnell, da nur die wichtigsten Kommandos und diese mit den am häufigsten benutzten Optionen implementiert sind.

Das bekannteste Multicall-Binary ist die Busybox ([http://www.busybox.net]). Sie bietet ein breites Spektrum unterschiedlicher Kommandos, inklusive Netzwerkkonfiguration oder Webserver an. Außerdem ist sie einfach konfigurierbar und hochgradig portierbar. Auch wir werden die Busybox verwenden, um damit das eingebettete System zu bauen.

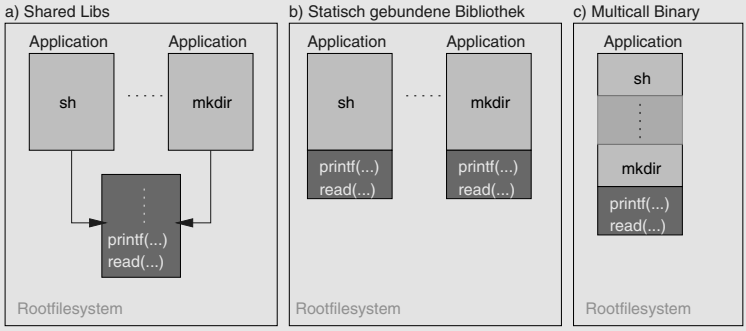


Abb. 3-3
Multicall-Binaries sparen Haupt- und Hintergrundspeicher

Aber auch für das statische Linken werden natürlich Bibliotheken benötigt. In eingebetteten Systemen greifen die Entwickler jedoch auf spezielle Varianten zurück, die besonderes ressourcensparend sind. So bringt beispielsweise der Linux-Kernel eine eigene Bibliothek mit (klibc), die im »early userland« (initramfs, RAMdisk) eingesetzt wird. Wichtig zu wissen: Die klibc ist nicht threadsafe. Werden also mehrere Threads parallel abgearbeitet, ist das Verhalten nicht mehr vorhersehbar.

Häufig ist die »uclibc« ([http://www.uclibc.org]) zu finden. Diese ist auch für Linux-Systeme ohne MMU (Memory Management Unit) geeignet. Android schließlich setzt auf BIONIC, die dank BSD-Lizenz unabhängig von der GPL und nur 200 kByte groß ist. Natürlich kann auch die glibc verwendet werden, die Standardbibliothek auf den Desktop-Systemen. Sie bietet viele Funktionen, ist in der Anwendung einfach, bindet aber sehr viele Ressourcen.

Neben der C-Standardbibliothek gibt es auch andere Libraries, die speziell für eingebettete Systeme zur Verfügung gestellt werden. Mit der »uclibc++« existiert beispielsweise eine C++ Standard Template Library.

Auf Systemebene sind einige Shell-Kommandos sinnvoll, die jedoch auf die notwendige Funktionalität reduziert sind. Tabelle 2-2 listet die wichtigsten auf.

Handelt es sich um ein vernetztes System, werden weitere Programme benötigt. Eine Auswahl zeigt Tabelle 2-4. Außerdem sind Programme und Mechanismen sinnvoll, die die Systemsoftware automatisiert aktualisieren.

3.2.1 Systemebene

Das Userland besteht aus folgenden Komponenten:

1. Hilfsprogramme. Dazu gehören Programme zum Setzen des Systemnamens und der IP-Adresse oder auch die Shell, über die mittels Tastatur Kommandos eingegeben werden können. Wie bereits erwähnt lassen sich die Hilfsprogramme für ein eingebettetes System am sinnvollsten in Form eines Multicall-Binaries realisieren. Obwohl real nur ein einziges Hilfsprogramm vorhanden ist, scheinen auf dem System mithilfe symbolischer Links diverse Hilfsprogramme zu existieren.
2. Konfigurationsdateien. In den Dateien ist beispielsweise der Systemname, die IP-Adresse oder der Name einer beim Hochfahren des Systems zu startenden Applikation abgelegt.
3. Gerätedateien. Diese stellen die Verbindung zwischen den Programmen und den Treibern her. So benötigt die Shell den Zugriff auf die Tastatur und den Bildschirm, die Konsole.
4. Strukturierung. Hilfsprogramme und Konfigurationsdateien werden sinnvollerweise strukturiert, also über eine Ordnerstruktur, den Dateibaum, auf einem Dateisystem abgelegt.

Ein minimaler Dateibaum eines klassischen Linux-Systems besteht aus den in Tabelle 3-4 aufgelisteten Verzeichnissen.

Verzeichnis	Bedeutung
/	Root-Verzeichnis
/bin	User-Kommandos
/sbin	Systemprogramme
/lib	Bibliotheken
/dev	Gerätedateien
/usr	Anwendungsprogramme
/etc	Konfigurationsdateien
/tmp	Temporäre Daten
/var	Log- und Spooldateien
/proc	Einhängepunkt für das virtuelle Proc-Filesystem
/sys	Einhängepunkt für das virtuelle Sys-Filesystem

Tabelle 3-4 *(Minimaler) Dateibaum*

Außerdem werden drei Gerätedateien benötigt:

- /dev/console
- /dev/zero
- /dev/null

Über die Konsole gibt das System Debug- und Logmeldungen aus. Daten, die auf `/dev/null` geschrieben werden, werden weggeworfen und der lesende Zugriff auf `/dev/zero` liefert Nullen zurück.

Je nach unterstützter Hardware gibt es weitere Gerätedateien, die entweder durch den Entwickler angelegt werden oder aber bei etwas komplexeren Systemen auch durch einen Prozess namens `udevd`.

Userland im Selbstbau

Um ein Userland zu bauen, sind die folgenden Schritte notwendig:

1. Imagedatei per `dd` erzeugen und formatieren
2. Busybox herunterladen, konfigurieren und generieren
3. Imagedatei (als Loop-Device) einhängen
4. Busybox installieren und Zugriffsrechte setzen
5. Gerätedateien anlegen
6. Imagedatei aushängen

Im Folgenden bauen wir ein Userland in Form einer Imagedatei, die der Emulation als Festplatte dient. Das Userland wird schlank und benötigt für die gestellte Aufgabe nur 8 Megabyte. Da wir das Userland zunächst im Emulator Qemu testen wollen, gehen wir wieder vom Verzeichnis `~/embedded/qemu/` aus.

Tabelle 3-5 *Basisoptionen für Busybox*

Hauptmenü	1. Untermenü	2. Untermenü
Busybox Settings	Build Options	Build Busybox as a static binary (no shared libs)
Coreutils	cat	
	cp	
	ls	
	mv	
	pwd	
	rm	
	rmdir	
	sleep	

➔

Hauptmenü	1. Untermenü	2. Untermenü
Linux System Utilities	mount	
	umount	
Process Utilities	ps	
Shells	ash	
	Choose which shell is aliased to 'sh' name ()	ash
	Choose which shell is aliased to 'bash' name ()	ash

1. Erstellen Sie für das Rootfilesystem ein eigenes Verzeichnis und wechseln Sie in dieses Verzeichnis:

```
quade@felicia:~/embedded/qemu> mkdir userland
quade@felicia:~/embedded/qemu> cd userland
quade@felicia:~/embedded/qemu/userland>
```

2. Als Erstes wird die Imagedatei erzeugt. Erstellen Sie dazu mithilfe des Kommandos `dd` (Diskdump) eine Datei der Größe 8.0 MByte. Geben Sie der Imagedatei den Namen `rootfs.img`. Die Option »if« (infile) des dd-Kommandos spezifiziert die Quelle, woher die Daten kommen. »of« (outfile) spezifiziert das Ziel der Daten. »bs« (blocksize) ist die Blockgröße, mit der die Daten intern von der Eingabe zur Ausgabe kopiert werden. »count« schließlich gibt die Anzahl der Blöcke an. In unserem Fall werden also von `/dev/zero` Nullen gelesen und in die Datei `rootfs.img` geschrieben, und zwar 8192 Mal 1024 (1K) Nullen.

```
quade@felicia:~/embedded/qemu/userland> dd if=/dev/zero of=rootfs.img \
   bs=1k count=8192
8192+0 Datensätze ein
8192+0 Datensätze aus
8388608 Bytes (8,4 MB) kopiert, 0,0301726 s, 278 MB/s
quade@felicia:~/embedded/qemu/userland>
```

3. Damit Sie auf die Imagedatei `rootfs.img` wie auf eine Festplatte zugreifen können, formatieren Sie diese mit einem `ext2`-Filesystem. Achten Sie darauf, dass genügend Inodes zur Verfügung stehen. Diese werden benötigt, um viele Dateien auf dem Filesystem ablegen zu können. Verwenden Sie hierzu die Option »-i 1024« beim Aufruf.

```
quade@felicia:~/embedded/qemu/userland> mkfs.ext2 -i 1024 -F rootfs.img ...
```

4. Als Standardprogramm (Multicall-Binary) für die Inhalte des Rootfilesystems wird Busybox eingesetzt. Laden Sie den Quellcode in einer aktuellen Version von der Webseite [http://www.busybox.net] herunter (zum Beispiel [http://www.busybox.net/downloads/busy-

box-1.21.1.tar.bz2]). Sie können hierfür wieder das Kommando `wget` verwenden.

```
quade@felicia:~/embedded/qemu/userland> wget \
    http://www.busybox.net/downloads/busybox-1.21.1.tar.bz2
```

5. Packen Sie Busybox aus.

```
quade@felicia:~/embedded/qemu/userland> tar xvf busybox-1.21.1.tar.bz2
...
```

6. Busybox verwendet das gleiche Konfigurationssystem wie der Linux-Kernel (Abb. 3-4), weshalb wir in dem Quellcodeverzeichnis von Busybox »make menuconfig« aufrufen können. Allerdings wollen wir ein minimales System generieren, das Sie komplett selbst zusammengestellt haben und entsprechend kennen. Daher sollten Sie sämtliche Standardeinstellungen durch Aufruf von »make allnoconfig« deaktivieren, bevor Sie die eigentliche Konfiguration respektive Auswahl vornehmen. Wichtig: Beim Build-Prozess (`make`) muss eine *statisch gelinkte Variante* erzeugt werden (Auswahlpunkt [Busybox Settings][Build Options])! Außerdem: Nur die Kommandos, die Sie auch konfigurieren, stehen Ihnen zur Verfügung. Konfigurieren Sie zumindest die Kommandos `ls`, `mount`, `mv`, `ps` und `pwd`. Wählen Sie des Weiteren als Shell die `ash` aus!

   Konfigurieren Sie anfangs nicht zu viele Werkzeuge, sondern vor allem diejenigen, die Sie kennen. Sie sollten die Konfiguration ohnehin in mehreren Iterationen auf Ihre Bedürfnisse anpassen.

   Sollten Sie Busybox auf einem 64-Bit-System generieren, wobei die Zielplattform ein 32-Bit-System ist, installieren Sie die Bibliothek `sudo apt-get install libc6-dev-i386`. Fügen Sie dann vor jedes `make`-Kommando noch die beiden Umgebungsvariablen `EXTRA_CFLAGS="-m32" EXTRA_LDFLAGS="-m32"` ein.

```
quade@felicia:~/embedded/qemu/userland> cd busybox-1.21.1/
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make allnoconfig
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make menuconfig
...
```

7. Um Busybox zu kompilieren, reicht ein einfaches `make` aus. Danach sollten Sie das Kommando `make install` aufrufen. Dieses sorgt dafür, dass im Verzeichnis `_install` die notwendigen symbolischen Links angelegt werden.

```
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make install
...
```

```
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> cd _install/
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1/_install> ls
bin
```

8. In diesem Schritt wird das Userland zusammengebaut. Dazu wird Busybox in das vorbereitete Rootfilesystem `rootfs.img` kopiert. Dabei ist das Programm `rsync` hilfreich, das nur die Dateien kopiert, die unterschiedlich sind. Sie müssen darüber hinaus mithilfe des Kommandos `chown` sicherstellen, dass die neuen Systemdateien dem Superuser gehören.

   Um auf die Imagedatei `rootfs.img` wie auf ein normales Filesystem (beispielsweise wie auf einen USB-Stick) zugreifen zu können, bietet Linux die Möglichkeit des Loop-Mountens an. Um unser Image mounten zu können, benötigen wir allerdings noch ein Verzeichnis, über das wir dann auf die Inhalte der Imagedatei zugreifen. Legen Sie daher unterhalb von `~/embedded/qemu/userland/` ein Verzeichnis `loop/` an.

```
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1/_install>
cd ../..
quade@felicia:~/embedded/qemu/userland> mkdir loop
quade@felicia:~/embedded/qemu/userland> sudo mount rootfs.img loop
loop[sudo] password for quade:
quade@felicia:~/embedded/qemu/userland> ls loop
lost+found
quade@felicia:~/embedded/qemu/userland> rsync -a busybox-1.21.1/_install/
loop
quade@felicia:~/embedded/qemu/userland> ls loop
bin  lost+found
quade@felicia:~/embedded/qemu/userland> sudo chown -R root.root loop
```

9. Damit Sie mit Busybox arbeiten können, benötigen Sie die Gerätedateien `/dev/console` und `/dev/null`. Später ist außerdem noch `/dev/tty1` erforderlich. Legen Sie diese bitte ebenfalls im Rootfilesystem an. Zum Anlegen der Gerätedateien dient das Kommando `mknod`, das neben dem Namen der Gerätedatei den Gerätedateityp und eine Major- und eine Minornummer mitbekommt. Als Typ stehen »b« für Blockdevices und »c« für Characterdevices zur Verfügung. Blockdevices repräsentieren Festplatten, USB-Sticks, SD-Karten oder DVD-Laufwerke. Hier kann man Daten aus technischen Gründen nur in Blöcken lesen. Characterdevices demgegenüber ermöglichen auch das Lesen einzelner Zeichen (Bytes), wie sie beispielsweise die Tastatur oder die serielle Schnittstelle liefert. Die Majornummer und die Minornummer bilden die sogenannte Gerätenummer, die einen Gerätetreiber identifiziert. Im Betriebssystemkern sind die Gerätetreiber nämlich nicht über Namen, sondern über Nummern zu finden. Die Konsole beispielsweise, über die Linux seine Standardein-

gaben bekommt beziehungsweise seine Ausgaben tätigt, ist (etwas verkürzt ausgedrückt) über den Treiber mit der Nummer 5-1 zu erreichen.

```
quade@felicia:~/embedded/qemu/userland> sudo mkdir loop/dev
quade@felicia:~/embedded/qemu/userland> sudo mknod loop/dev/console c 5 1
quade@felicia:~/embedded/qemu/userland> sudo mknod loop/dev/null c 1 3
quade@felicia:~/embedded/qemu/userland> sudo mknod loop/dev/tty1 c 4 1
quade@felicia:~/embedded/qemu/userland>
```

10. Hängen Sie das Image wieder aus (unmounten).

```
quade@felicia:~/embedded/qemu/userland> sudo umount loop
```

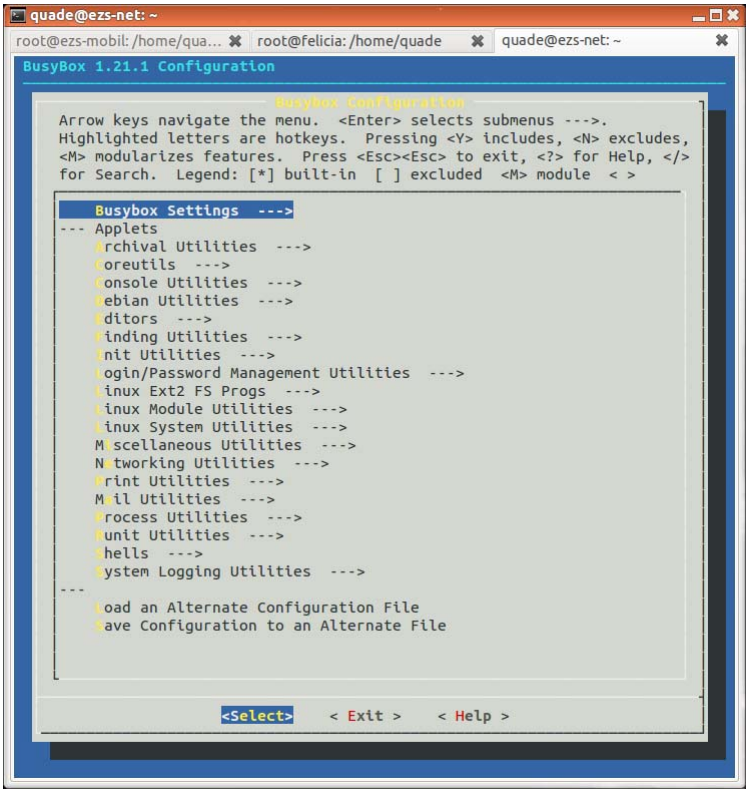


Abb. 3-4 *Oberfläche zur Konfiguration der Busybox*

Jetzt können Sie Ihren Kernel und Ihr Rootfilesystem testen. Installieren Sie — falls nicht bereits geschehen — den Emulator Qemu. Starten Sie den Emulator Qemu aus dem Verzeichnis `~/embedded/qemu/` heraus mit dem Kernel (Option »-kernel«) und dem Rootfilesystem (Option »-hda«) als Parameter (Abb. 3-5). Das unten stehende Kommando bootet mithilfe der über die Option »-append« übergebenen Kernelparameter Ihr System direkt in eine Shell, in der Sie die vorher konfigurierten Kommandos wie beispielsweise `ls` absetzen können. Auch wenn es noch

keine Benutzerverwaltung gibt, sind Sie über die Shell direkt mit Superuser-Rechten ausgestattet. Einige Kommandos, beispielsweise `ps`, führen zu Fehlermeldungen, da das System noch unkonfiguriert ist. Erst durch das Einbinden (Mounten) der virtuellen Filesysteme `/proc/` und `/sys/` erhalten diese Programme Zugriff auf die kernelinternen Informationen. Übrigens geht das von Ihnen erstellte System von einem englischen Tastaturlayout aus.

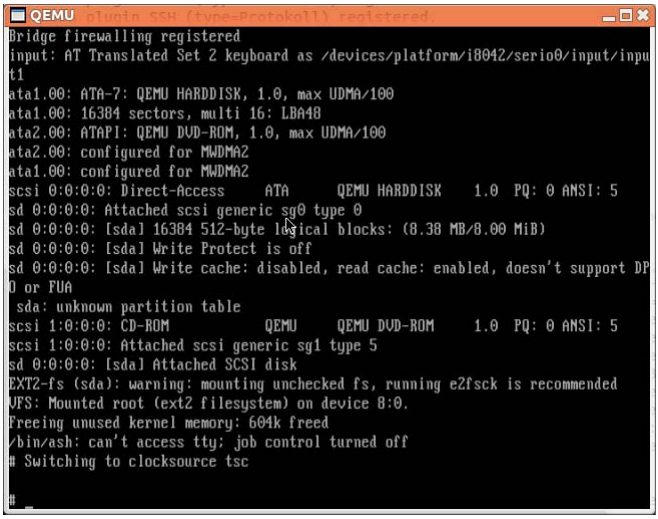


Abb. 3-5
Das selbst gebaute System bootet in eine Shell.

```
quade@felicia:~/embedded/qemu/userland> sudo apt-get install qemu
quade@felicia:~/embedded/qemu/userland> cd ..
quade@felicia:~/embedded/qemu> qemu-system-x86_64 -kernel \
   linux-3.10.9/arch/x86/boot/bzImage -hda userland/rootfs.img \
   -append "root=/dev/sda rw init=/bin/ash"
```

Wenn Sie die Compile-Vorgänge (Kernel, Busybox) auf einem 32-Bit-System durchgeführt haben, muss der Kernel auch für einen 32-Bit-Prozessor generiert worden sein. Qemu wird in diesem Fall mit den folgenden Parametern aufgerufen:

```
quade@felicia:~/embedded/qemu> qemu-system-i386 -kernel \
   linux-3.10.9/arch/x86/boot/bzImage -hda userland/rootfs.img \
   -append "root=/dev/sda rw init=/bin/ash"
```

Wenn's schiefgeht ...

Wenn das System nicht wie erwartet bootet, kommen drei Ursachen infrage: der Emulator Qemu, der Kernel oder das Rootfilesystem.

In manchen Versionen bereitet Qemu mit der grafischen Oberfläche Schwierigkeiten. Versuchen Sie in diesem Fall die Option `-curses`.

Wenn bereits der Kernel nicht bootet, passen möglicherweise Kernel und Qemu nicht zusammen. Versuchen Sie einen Kernel, der für einen x86-Prozessor kompiliert wurde, auf einem Qemu für ARM zu starten? Auf der Webseite zum Buch finden Sie mit der Datei `3.2./bzImage.3.10.9.x86_64` ein Image, das für ein 64-Bit-Hostsystem und den Emulator Qemu `qemu-system-x86_64` übersetzt wurde. Hiermit können Sie die Kombination Qemu und Kernel testen.

Erscheinen zwar viele Kernelmeldungen, aber auch nach Eingabe von Return kein Prompt (Eingabeaufforderung in Form eines »#«), kann es mehrere Ursachen dafür geben.

Überprüfen Sie als Erstes, ob das Rootfilesystem gemountet werden konnte. Sie können dazu die Meldungen auf Ihrem Terminal mit denen in Abbildung 3-5 vergleichen. Ist dies nicht der Fall, fehlen möglicherweise im Kernel Treiber für das Filesystem. Verwenden Sie wie hier vorgeschlagen zunächst ein ext2-Filesystem.

Konnte das Rootfilesystem gemountet, aber keine Shell gestartet werden? Überprüfen Sie, ob Sie als Option beim Aufruf von Qemu `-append "root=/dev/sda rw init=/bin/ash"` angegeben haben. Insbesondere der Teil mit `init` ist wichtig. Eventuell ist Busybox aber auch nicht richtig übersetzt worden. Das liegt häufig daran, dass Sie keine statisch gelinkte Variante von Busybox erzeugt haben (siehe Tabelle 3-5) oder dass Busybox für 32-Bit übersetzt wurde und Sie ein 64-Bit-System emulieren. Umgekehrt ebenso: Busybox ist für ein 64-Bit-System generiert worden, der Kernel aber nur für 32-Bit. Um das herauszufinden, können Sie `rootfs.img` erneut mounten und mit dem Befehl `file` den Typ des Executables `loop/bin/busybox` untersuchen. Hier stellen Sie übrigens auch fest, ob das Programm statisch oder dynamisch gelinkt wurde.

Alternativ finden Sie auf der Webseite zum Buch ein downloadbares Image (`3.2./rootfs.img.x86_64_1`), das Sie testen können. Ebenso finden Sie dort die Konfiguration zum Kernel als auch zur Busybox, die Sie ebenfalls zum Test heranziehen können. Dazu kopieren Sie die Konfigurationsdateien an die entsprechenden Stellen und generieren neu. Hier folgen zunächst die Befehle für den Kernel, dabei wird angenommen, dass die funktionstüchtige Konfigurationsdatei `3.2./config.3.10.9.x86_64` bereits im Verzeichnis des Kernelquellcodes liegt:

```
quade@felicia:~/embedded/qemu/linux-3.10.9> \
    cp ../../files/3.2./config.3.10.9.x86_64 .config
quade@felicia:~/embedded/qemu/linux-3.10.9> make clean
quade@felicia:~/embedded/qemu/linux-3.10.9> make
```

Und für die Busybox entsprechend:

```
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> \
cp ../../files/3.2./config.busybox-1.21.1.qemu .config
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make clean
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make install
```

Sobald Busybox neu generiert und installiert ist, muss natürlich noch das Rootfilesystem aktualisiert werden (ab Schritt 8).

Erste Automatisierung

Um das Target-System über ein einzelnes Kommando (Skript) zu generieren, sind die folgenden Schritte notwendig:

1. Skript zur Generierung des Targets `mkrootfs.sh` erstellen
2. Execute-Attribute setzen, sodass das Skript ausgeführt werden kann

Um Ihr System nutzbar zu machen, sind weitere Systemkommandos zu konfigurieren und zu installieren. Es ist mühsam dabei immer alle Befehle von Hand erneut aufzurufen. Mithilfe eines Skripts automatisieren Sie die Generierung des Systems (siehe Beispiel 3-1).

Beispiel 3-1 *Skript zur Generierung des Rootfilesystems (Version 1) <mkrootfs.sh>*

```
#!/bin/bash

MAINDIR=~/embedded

cd $MAINDIR/qemu/userland
dd if=/dev/zero of=rootfs.img bs=4k count=2k
mke2fs -i 1024 -F rootfs.img
sudo mount -o loop rootfs.img loop
rsync -a busybox-1.21.1/_install/ loop
mkdir loop/dev
mkdir loop/etc
mkdir loop/proc
mkdir loop/sys
sudo mknod loop/dev/console c 5 1
sudo mknod loop/dev/null c 1 3
sudo mknod loop/dev/tty0 c 4 0
sudo mknod loop/dev/tty1 c 4 1
sudo mknod loop/dev/tty2 c 4 2
sudo chown -R root.root loop
# MARK A
# MARK B
# MARK C
# MARK D
# Aushaengen
sudo umount loop
```

Erstellen Sie im Ordner `~/embedded/qemu/` ein solches Skript. Als Editor können Sie dafür beispielsweise `gedit` oder bei entsprechender Erfahrung auch `vim` verwenden. Vergessen Sie nicht, das Skript per `chmod` ausführbar zu machen. Generieren Sie dann das Rootfilesystem neu, indem Sie das erstellte Skript mit Root-Rechten ausführen. Sobald das Userland auf diese Art neu generiert ist, testen Sie es mit Qemu.

```
quade@felicia:~/embedded/qemu> gedit mkrootfs.sh &
quade@felicia:~/embedded/qemu> chmod +x mkrootfs.sh
quade@felicia:~/embedded/qemu> ./mkrootfs.sh
2048+0 Datensätze ein
2048+0 Datensätze aus
8388608 Bytes (8,4 MB) kopiert, 0,00872987 s, 961 MB/s
mke2fs 1.42 (29-Nov-2011)
...
Platz für Gruppentabellen wird angefordert: erledigt
Inode-Tabellen werden geschrieben: erledigt
Schreibe Superblöcke und Dateisystem-Accountinginformationen: erledigt

quade@felicia:~/embedded/qemu>
```

Netzwerkeinbindung

Um das Embedded Linux ins Netzwerk einzubinden, sind die folgenden Schritte notwendig:

1. Netzwerkprogramme (ifconfig, route, ping) einbinden
2. Skript `start_el.sh` zur Netzwerkkonfiguration für Qemu erstellen
3. Embedded Linux per Skript booten und auf dem System per `ifconfig` eine IP-Adresse zuweisen

Um ein vollständiges Beispiel zu haben, soll das selbst gebaute System die Funktion eines Webservers bekommen. Dieser gibt eine HTML-Seite mit einer Liste der im System laufenden Rechenprozesse (Ausgabe des Kommandos `ps`) zurück. Dazu muss das Netzwerk in Kombination mit dem Emulator Qemu aktiviert sein.

Als Erstes benötigen Sie auf dem eingebetteten System Programme zur Netzwerkkonfiguration. Minimal sind das `ifconfig`, `route` und `ping`. Alle drei sind bei der Busybox-Konfiguration unter dem Menü `Networking Utilities` zu finden.

```
quade@felicia:~/embedded/qemu> cd userland/busybox-1.21.1
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make menuconfig
...
# [Networking Utilities] ping, ifconfig und route einbinden
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make install
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> cd ../..
quade@felicia:~/embedded/qemu> ./mkrootfs.sh
```

Installieren Sie auf Ihrem Hostsystem das Paket `uml-utilities`: `sudo apt-get install uml-utilities`. Dieses wird zur Einrichtung des Tap-Devices

benötigt, das Qemu für das Netzwerk braucht und das nach den Aktivitäten wieder gelöscht werden sollte.

Die komplexe Konfiguration, die neben dem Erstellen des Tap-Devices auch noch aus der Konfiguration der Firewall besteht, ist am einfachsten per Skript zu bewerkstelligen (siehe Beispiel 3-2). Achten Sie auch bei diesem Skript darauf, dass das x-Bit gesetzt und es damit ausführbar ist (Kommando `chmod +x`). Das Tap-Device bekommt per Skript die IP-Adresse `10.69.0.1`. Falls Sie selbst diese Netzwerkadresse bereits verwenden, sollten Sie anstelle dieser Adresse beispielsweise die private Netzadresse 192.168.69.1 wählen. Des Weiteren beachten Sie bitte, dass `start_el.sh` Firewallregeln ändert. Sollte auf Ihrem Entwicklungsrechner eine Firewall aktiv sein, müssen Sie das Skript anpassen.

```
quade@felicia:~/embedded/qemu/userland> sudo apt-get install uml-utilities
quade@felicia:~/embedded/qemu/userland> cd ~/embedded/qemu
quade@felicia:~/embedded/qemu> gedit start_el.sh &
quade@felicia:~/embedded/qemu> chmod +x start_el.sh
quade@felicia:~/embedded/qemu> ./start_el.sh # Start des Embedded Linux
```

Beispiel 3-2
Skript zum Starten von Qemu mit Netzwerkunterstützung <start_el.sh>

```
#!/bin/bash

export ARCH=x86_64
#export EXTRA_CFLAGS=-m32 EXTRA_LDFLAGS=-m32
KERNELPARAMS="root=/dev/sda rw init=/bin/ash"
QEMU_COMMAND=qemu-system-$ARCH

function create_tap {
   echo "creating tap device"
   sudo modprobe tun

   USERID=$(whoami)
   iface=$(sudo tunctl -b -u $USERID)
   sudo tunctl -u $USERID -t $iface
   sudo ip a a 10.69.0.1/24 dev $iface
   sudo ifconfig $iface up
}

function internet_tap_on {
   sudo bash -c "echo 1 > /proc/sys/net/ipv4/ip_forward"
   sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
}

function internet_tap_off {
#   sudo bash -c "echo 0 > /proc/sys/net/ipv4/ip_forward"
   sudo iptables -t nat -D POSTROUTING -o eth0 -j MASQUERADE
}
```

```
function delete_tap {
   sudo tunctl -d $1
}

function start_qemu {
   create_tap
   $QEMU_COMMAND -m 128 -hda userland/rootfs.img \
      -kernel linux-*/arch/x86/boot/bzImage -append "$KERNELPARAMS" \
      -net nic,model=ne2k_pci \
      -net tap,script=no,downscript=no,ifname=tap0
   delete_tap tap0
}

if [ $0 != "bash" ]
then
   internet_tap_on
   start_qemu
   internet_tap_off
fi
```

Interessant ist jetzt natürlich vor allem die Aktivierung des Netzwerkes auf dem Target, dem eingebetteten System. Um hier ein Netzwerk zum Laufen zu bekommen, muss es ebenfalls eine IP-Adresse aus dem Netzbereich haben, der dem Tap-Device von Qemu zugeordnet worden ist, beispielsweise 10.69.0.99 beziehungsweise, falls die Adresse bereits vergeben war, die 192.168.69.99

Starten Sie hierzu Ihr System per `start_el.sh`. Auf dem Embedded Linux aktivieren Sie per `ifconfig eth0 10.69.0.99` das Netzwerk. Wenn Sie die Funktionstüchtigkeit des lokalen Netzwerks beispielsweise per `ping` testen, fügen Sie auf dem emulierten System die Option »-c 4« hinzu. Damit testet `ping` die Verbindung genau vier Mal (siehe Abb. 3-6) und beendet sich danach. Ein <Strg><c>, was sonst typischerweise zum Abbruch eines Kommandos führt, funktioniert im selbst gebauten System nämlich noch nicht.

Um mit jedem Reboot die Netzwerkverbindung nicht von Hand auf dem eingebetteten System konfigurieren zu müssen, ist eine Automatisierung des Vorgangs sinnvoll. Dazu muss zunächst das selbst gebaute System dazu gebracht werden, nicht direkt in eine Shell zu booten, sondern automatisiert ein lokales Skript zu starten, das die notwendigen Konfigurationen durchführt. Dieser Systemumbau wiederum soll durch eine Erweiterung des Skripts `mkrootfs.sh` erfolgen.

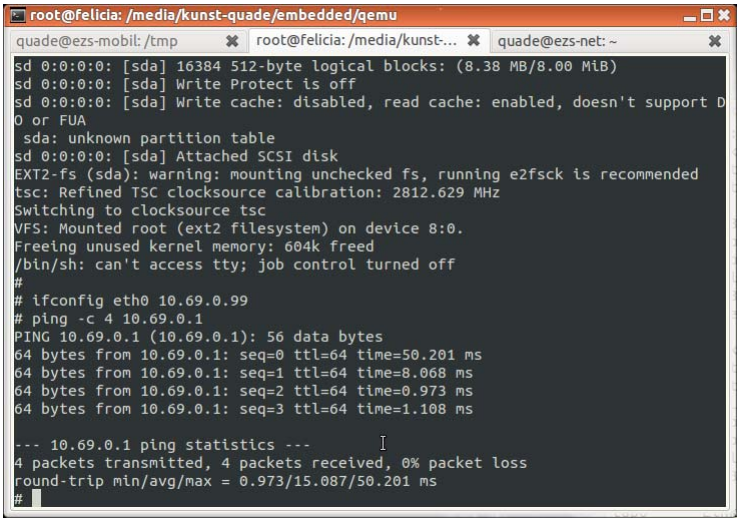


Abb. 3-6
Per ifconfig wird das Netzwerk aktiviert.

Init-System

Um auf dem Embedded Linux das Netzwerk beim Booten mithilfe von `init` automatisiert zu starten, sind folgende Schritte notwendig:

1. Inittab im Verzeichnis `userland/target/` anlegen
2. Embedded-Linux-Startskript `rcS` im Verzeichnis `userland/target/` anlegen
3. Generierungsskript `mkrootfs.sh` anpassen, sodass `inittab` und `rcS` ins Image kopiert werden
4. Skript `start_el.sh` abändern, so dass `init` gestartet wird
5. In Busybox `init` auswählen, Busybox generieren und installieren
6. Target-System per `mkrootfs.sh` generieren

Bisher ist beim Aufruf von Qemu dem Kernel der Parameter »init=/bin/ash« mitgegeben worden. Der hat dafür gesorgt, dass anstelle des Init-Prozesses eine Shell gestartet wurde.

Die erste Task, die in einem Unixsystem gestartet wird, ist die Init-Task. Normalerweise liest diese die Datei `/etc/inittab` ein, die eine Konfiguration bezüglich der nächsten zu startenden Jobs enthält. `init` startet nicht nur diese Jobs, sondern überwacht diese auch. Kasten auf Seite 56 beschreibt den Aufbau der inittab.

Inittab

Die Init-Task entnimmt der Datei `/etc/inittab` die Jobs, die abhängig vom Systemzustand zu starten und zu überwachen sind. Runlevel ermöglichen darüber hinaus unterschiedliche Betriebsmodi, in denen auch unterschiedliche Jobs aktiviert werden. So kann ein Runlevel das System in eine Konsole booten, ein anderer die grafische Benutzeroberfläche starten.

Die Inittab besteht aus einer Reihe von Zeilen, die alle nach dem Schema

```
id:runlevels:action:process
```

aufgebaut sind. Die Felder haben die folgende Bedeutung:

id

Dies ist ein eindeutiger Name aus bis zu vier Buchstaben, der den Eintrag identifiziert. Typischerweise wird die ID nach dem tty benannt, auf dem die spezifizierte Task ihre Ausgaben macht, beispielsweise `tty1`. Bleibt das Feld leer, wird `/dev/console` als Ausgabeziel angenommen.

runlevels

In diesem Feld stehen einer oder mehrere Runlevel (von 0 bis 6), für die der Eintrag gelten soll. Hinweis: Das Init-System von Busybox unterstützt keine Runlevel, hier bleibt der Eintrag leer.

action

Dieses Feld beschreibt die Aktion, die von `init` ausgeführt werden soll. Das Init-System der Busybox kennt die Aktionen »initdefault«, »sysinit«, »respawn«, »askfirst«, »wait«, »once«. Andere Init-Systeme kennen neben anderen noch die Aktionen »boot«, »bootwait«, »off«, »powerwait«, »powerfail« und »ctrlaltdel«. Tabelle 3-6 zeigt die Bedeutung der Aktionen.

process

Gibt an, welche Task mit welchen Parametern gestartet werden soll.

Beispiel 3-3 zeigt eine Inittab für das Init-System der Busybox, die unter anderem eine einfache Systeminitialisierung vornimmt. Die Inittab eines Desktop-Systems kann Busybox übrigens nicht verarbeiten!

Tabelle 3-6
Von init unterstützte Aktionen

Aktion	Bedeutung
initdefault	Dieser Eintrag spezifiziert den Runlevel, den das System nach dem Boot-Vorgang einnehmen soll. Das Feld »process« wird folglich ignoriert.
sysinit	Die Task wird während des Bootens ausgeführt, und zwar vor den Einträgen von »boot» und »bootwait«.
respawn	Die Task wird von Init neu gestartet, sobald sie terminiert (Einsatz beispielsweise bei `getty`).
askfirst	Die Task wird von Init jedes Mal neu gestartet, sobald sie sich beendet hat (wie respawn). Allerdings wird vorher die Meldung »Please press Enter to activate this console.« angezeigt.
wait	Die Task wird einmalig gestartet. Init wartet dann auf das Ende der Task.

➔

Aktion	Bedeutung
once	Die Task wird einmalig gestartet.
boot	Die Task wird während des Boot-Vorgangs ausgeführt.
bootwait	Die Task wird während des Boot-Vorgangs ausgeführt, Init wartet aber auf das Ende (wird beispielsweise für /etc/rc verwendet).
off	Es wird nichts gemacht.
powerwait	Die Task wird ausgeführt, wenn das System ein Problem mit der Stromversorgung (Powerfail) feststellt. Init wartet auf das Ende der Task.
powerfail	Die Task wird ausgeführt, wenn das System ein Problem mit der Stromversorgung (Powerfail) feststellt.
ctrlaltdel	Die Task wird ausgeführt, wenn Init das Signal SIGINT empfängt. Das ist typischerweise der Fall, wenn jemand die Tastenkombination <STRG><ALT><ENTF> betätigt.

Beispiel 3-3
Konfiguration zum Systemstart </etc/inittab>

```
# Startup the system
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -o remount,rw / # REMOUNT_ROOTFS_RW
null::sysinit:/bin/mkdir -p /dev/pts
null::sysinit:/bin/mkdir -p /dev/shm
null::sysinit:/bin/mount -a
null::sysinit:/bin/hostname -F /etc/hostname
# now run any rc scripts
::sysinit:/etc/init.d/rcS

# Put a getty on the serial port
tty1::respawn:/sbin/getty -L tty1 115200 vt100
ttyAMA0::respawn:/sbin/getty -L ttyAMA0 115200 vt100

::ctrlaltdel:/sbin/reboot
```

Init ist damit sehr gut geeignet, um zum einen ein Skript zur Systemkonfiguration (wir nennen dieses rcS) und zum anderen eine Shell (Kommandozeile) zu starten. Dafür wird nicht unbedingt eine komplexe Inittab benötigt, wie in Beispiel 3-3 dargestellt. Vielmehr reicht zunächst eine einfache Variante, bestehend aus nur zwei Zeilen:

```
::sysinit:/bin/ash /etc/rcS
::askfirst:/bin/ash
```

Das von Init über die vorgestellte Inittab aufgerufene Skript /etc/rcS soll dann die Systemverzeichnisse /proc/ und /sys/ einbinden. Damit funktionieren dann auch Kommandos wie ps. Außerdem soll die IP-Adresse konfiguriert werden. Damit hat das Skript den folgenden Inhalt:

```
#!/bin/ash

mount -t proc none /proc
mount -t sysfs none /sys
ifconfig eth0 10.69.0.99
```

Sowohl die Inittab als auch das Skript `rcS` sollen per Skript `mkrootfs.sh` in das Image kopiert werden. Dazu legen wir die beiden Dateien beispielsweise in einem neuen Unterverzeichnis `~/embedded/qemu/userland/target/` ab. Legen Sie ein entsprechendes Verzeichnis an und erstellen Sie in diesem die beiden Dateien.

Das Skript `start_el.sh` startet über den Kernelparameter »init=/bin/ash« das System so, dass es in eine Shell bootet. Da wir jetzt `init` verwenden, muss dieser Teil gelöscht werden, sodass `KERNELPARAMS="root=/dev/sda rw"` übrig bleibt.

Anschließend muss noch `mkrootfs.sh` angepasst werden. Das Skript `mkrootfs.sh` ist dafür insofern vorbereitet, als mit `# MARK A` die Stelle markiert ist, an der die ersten Änderungen einzutragen sind. Ergänzen Sie zwischen MARK A und MARK B die folgenden Zeilen:

```
...
# MARK A
cp target/inittab loop/etc
cp target/rcS loop/etc
chmod +x loop/etc/rcS
# MARK B
...
```

Das Kopieren der Datei per `cp` und das Ändern der Zugriffsrechte per `chmod` lässt sich auch durch das Kommando `install` realisieren. Über die Option »-m« wird oktal der Zugriffsmodus angegeben. Jede Zahl, die Sechs beispielsweise, ist als Bitmuster zu interpretieren (also `110`). Ist dabei das erste der drei Bits »1«, darf die Datei gelesen werden. Ist das zweite gesetzt, darf geschrieben werden, und wenn das dritte gesetzt ist, ist die Datei ein Programm, das ausgeführt werden kann. Damit bedeutet die Sechs Lesen und Schreiben, die Fünf (`101`) Lesen und Ausführen und die Vier (`100`) bedeutet nur Lesen. Alternativ zu der obigen Version können Sie also auch die folgenden Kommandos verwenden:

```
# MARK A
# alternativ zu cp und chmod
sudo install -m 644 target/inittab loop/etc
sudo install -m 755 target/rcS loop/etc
# MARK B
```

Konfigurieren Sie noch Busybox, sodass diese die Funktion `init` zur Verfügung stellt. Danach generieren Sie das System neu:

```
quade@felicia:~/embedded/qemu> cd userland
quade@felicia:~/embedded/qemu/userland> mkdir target
quade@felicia:~/embedded/qemu/userland> cd target
quade@felicia:~/embedded/qemu/userland/target> gedit inittab &
    ::sysinit:/etc/rcS
    ::askfirst:/bin/ash
quade@felicia:~/embedded/qemu/userland/target> gedit rcS &
quade@felicia:~/embedded/qemu/userland/target> cd ../..
quade@felicia:~/embedded/qemu> gedit start_el.sh &
    # "... init=/bin/ash" loeschen
quade@felicia:~/embedded/qemu> gedit mkrootfs.sh &
quade@felicia:~/embedded/qemu>cd userland/busybox-1.21.1/
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make menuconfig
...
# [Init Utilities][init]
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make install
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> cd ../..
quade@felicia:~/embedded/qemu>./mkrootfs.sh
...
```

Wenn Sie jetzt per `start_el.sh` das selbst gebaute System im Emulator booten, sollte es vom Entwicklungsrechner aus über das Netzwerk erreichbar sein.

3.2.2 Funktionsbestimmende Applikationen

Um dem Embedded Linux die Funktion eines Webservers zu geben, sind die folgenden Schritte notwendig:

1. In Busybox die Funktion `httpd` freischalten, Busybox generieren und installieren
2. Die Konfigurationsdatei `httpd.conf` im Verzeichnis `userland/target/` anlegen
3. Die Webseite `index.html` im Verzeichnis `userland/target/` anlegen
4. Das Skript `ps.cgi` im Verzeichnis `userland/target/` anlegen
5. Generierungsskript `mkrootfs.sh` anpassen, sodass `httpd.conf`, `index.html` und `ps.cgi` ins Image kopiert werden
6. Target-System per `mkrootfs.sh` generieren

Für die funktionsbestimmenden Applikationen steht im Umfeld eingebetteter Systeme meist nur eine eingeschränkte Umgebung zur Verfü-

gung. Insbesondere fehlt häufig ein Framework für dynamische Bibliotheken, sodass eigene Programme statisch gelinkt werden müssen.

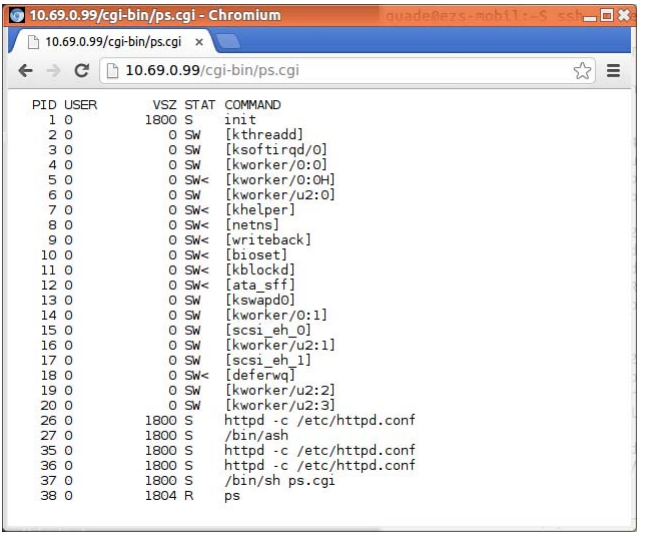


Abb. 3-7
Dynamische Inhalte werden per Skript generiert.

Das bisher von uns aufgebaute System soll die Funktion eines Webservers bekommen, der neben einer statischen Webseite als dynamischen Inhalt die Liste der aktiven Rechenprozesse in einer zweiten Webseite darstellen soll (Abb. 3-7).

Der auf Serversystemen übliche Webserver Apache wird im Bereich eingebetteter Systeme nur in Ausnahmefällen eingesetzt. Er ist komplex, benötigt viele Ressourcen und eben auch eine umfangreiche Unterstützung durch das System. In eingebetteten Systemen wird auf einfachere Varianten, beispielsweise den Webbserver `boa` oder `lighttpd`, zurückgegriffen. Auch Busybox bringt einen Webserver mit, der für unsere Zwecke ausreichend und dabei sehr schlank und einfach ist.

Neben dem Webserver benötigen wir noch die Inhalte, die Webseiten. Webseiten sind technisch nach dem HTML-Format aufgebaut. In unserem Fall soll eine statische und eine dynamische Webseite zur Verfügung gestellt werden. Statische Webseiten sind direkt im Filesystem als HTML-Dateien abgelegt. Demgegenüber werden dynamische Webseiten von Skripten oder Programmen generiert, die der Webserver eventuell mit Parametern versehen startet. Sie generieren dann eine HTML-Seite, die dem Webserver als Ergebnis geliefert wird. Der Webserver reicht diese Webseite an den anfragenden Webbrowser (Client) weiter.

Als Programmiersprachen für die Skripte wird häufig PHP oder Perl eingesetzt, sie lassen sich aber auch als normale Shellskripte erstellen.

```
<html>
<h1>Hi, I am your embedded system!</h1>
</html>
```

Beispiel 3-4 *Statische HTML-Seite für den Webserver <index.html>*

```
#!/bin/sh
echo "Content-type: text/html"
echo ""
echo ""
echo "<html>"
echo "<pre>"
ps
echo "</pre>"
echo "</html>"
echo ""
echo ""
```

Beispiel 3-5 *Skript zur Generierung der dynamischen Inhalte <ps.cgi>*

```
H:/var/www
*.cgi:/bin/sh
```

Beispiel 3-6 *Konfigurationsdatei für den Busybox-Webserver <httpd.conf>*

Die Webseiten des Servers (HTML-Seiten und Skripte) befinden sich typischerweise unterhalb des Verzeichnisses `/var/www/`. Der Webserver selbst benötigt meist noch eine Konfiguration, die im Verzeichnis `/etc/` abgelegt wird.

Um den Webserver in dem eingebetteten System zu installieren, sind damit die folgenden Aufgaben zu bewerkstelligen (siehe Beispiel 3-7):

- In der Busybox-Konfiguration muss unter dem Auswahlpunkt [Networking Utilities] der Webserver `httpd` ausgewählt werden. Danach wird Busybox neu generiert und installiert.
- Sie müssen dafür sorgen, dass der HTTP-Server `httpd` auf Ihrem eingebetteten System automatisch gestartet wird. Das können Sie beispielsweise über eine Modifikation des Skripts `rcS` auf dem Entwicklungsrechner im Verzeichnis `userland/target/` bewerkstelligen. Hier fügen Sie die Zeile »httpd -c /etc/httpd.conf« am Ende der Datei ein.
- Auf dem Target, dem eingebetteten Linux-System, werden drei Dateien benötigt, die Sie auf dem Entwicklungsrechner per Editor unterhalb des Verzeichnisses `userland/target/` erstellen:
  - Der Webserver benötigt eine Konfiguration, die in Beispiel 3-6 zu sehen ist. Insbesondere benötigt der Webserver für die Ausführung von Skripten die Angabe des zugehörigen Interpreters.

Die Konfigurationsdatei `httpd.conf`, die diese Information enthält, muss auf dem eingebetteten System in `/etc/` abgelegt sein, was von `mkrootfs.sh` durchgeführt wird.

- Die statische Webseite muss mit beispielsweise dem Inhalt aus Beispiel 3-4 als Datei mit dem Namen `index.html` erstellt werden.
- Das Skript zur Generierung der dynamischen Webseite soll mit dem Inhalt aus Beispiel 3-5 als normales Shellskript unter dem Namen `ps.cgi` erstellt werden.

- Das Generierungsskript `mkrootfs.sh` muss angepasst werden. Dabei müssen im Rootfilesystem des eingebetteten Systems die Verzeichnisse für die Webserverinhalte angelegt werden (`/var/www/` und `/var/www/cgi-bin/`). Außerdem müssen die drei Dateien in das Rootfilesystem an die richtige Stelle kopiert werden. Im Skript `mkrootfs.sh` ist dafür zwischen den Markierungen B und C der nachfolgende Code einzufügen:

```
# MARK B
sudo mkdir -p loop/var/www/cgi-bin/
sudo install -m 644 target/index.html loop/var/www/
sudo install -m 755 target/ps.cgi loop/var/www/cgi-bin/
sudo install -m 644 target/httpd.conf loop/etc
# MARK C
```

- Das System wird schließlich durch Aufruf von `./mkrootfs.sh` generiert.

Beispiel 3-7 *Kommandos, um dem Embedded Linux eine Aufgabe zu geben*

```
quade@felicia:~/embedded/qemu>cd userland/target/
quade@felicia:~/embedded/qemu/userland/target> gedit rcS &
quade@felicia:~/embedded/qemu/userland/target> gedit index.html &
quade@felicia:~/embedded/qemu/userland/target> gedit ps.cgi &
quade@felicia:~/embedded/qemu/userland/target> gedit httpd.conf &
quade@felicia:~/embedded/qemu/userland/target> cd ../busybox-1.21.1/
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make menuconfig
...
# [Networking Utilities][httpd] Webserver zur Auswahl hinzufügen
...
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> make install
quade@felicia:~/embedded/qemu/userland/busybox-1.21.1> cd ../..
quade@felicia:~/embedded/qemu> gedit mkrootfs.sh &
quade@felicia:~/embedded/qemu> ./mkrootfs.sh # System generieren
quade@felicia:~/embedded/qemu> ./start_el.sh # System testen
```

Wenn Sie nach diesen Vorbereitungen das Userland neu generiert haben, können Sie mit Qemu durch Aufruf von `start_el.sh` testen. Rufen Sie dazu von Ihrem Entwicklungsrechner (nur von diesem aus gibt es über Qemu die Verbindung zum emulierten System) einen Browser auf und geben Sie die URL `http://10.69.0.99/index.html` ein. Nun muss eine HTML-Seite mit dem Text »Hi, I am your embedded system!« erscheinen (Abb. 3-8). Rufen Sie danach die URL `10.69.0.99/cgi-bin/ps.cgi` auf. Jetzt müssten Sie eine Liste der aktiven Rechenprozesse in Ihrem Webbrowser sehen (Abb. 3-7). Ist das der Fall — herzlichen Glückwunsch — haben wir unser Ettapenziel erreicht.

Abb. 3-8
Der Webserver auf dem Embedded System

Wenn's schiefgeht ...

Sollten Sie Schwierigkeiten haben, können Sie die Ursache wieder mithilfe der auf den Webseiten zum Buch abgelegten Dateien eingrenzen. Die Datei `rootfs.img.x86_64_2` ist ein funktionierendes Systemimage, das auf die IP-Adresse `10.69.0.99` konfiguriert ist. Funktioniert das Qemu-Netzwerk, müssten Sie mit diesem auf die Webseiten zugreifen können.

Die zugehörige Busybox-Konfiguration finden Sie in der Datei `config.busybox-1.21.1.qemu_2`. Sie können diese wieder auf die Datei `busybox-1.21.1/.config` kopieren. Die neue Version des Skripts zur Generierung des Rootfilesystems ist unter dem Namen `mkrootfs.sh_B` auf den Webseiten zu finden, das Skript zur Systeminitialisierung unter `rcS_B`.

3.3 Cross-Development für den Raspberry Pi

Um für den Raspberry Pi ein eigenes System zu bauen, sind die folgenden Schritte notwendig:

1. Kernel herunterladen, konfigurieren, generieren
2. Userland generieren
3. Installation auf der SD-Karte

Theoretisch ist es möglich, die System- und Anwendungssoftware für den Raspberry Pi auf dem Raspberry Pi selbst zu erzeugen, also eine Hostentwicklung durchzuführen. Praktisch jedoch dauert allein das Generieren des Kernels mehrere Stunden, sodass eine Host-/Target-Entwicklung effizienter ist. Je nach Leistungsfähigkeit des Hostsystems kann die Generierung des Kernels auf unter zehn Minuten reduziert werden.

Für die Host-/Target-Entwicklung wird eine Cross-Development-Toolchain benötigt, die aus Cross-Compiler, Cross-Linker, Cross-Libraries und Cross-Debugger besteht. Als Cross-Development-Toolchain wird im Allgemeinen gern auf die GNU-Toolchain gesetzt, die parallel zu den Hostwerkzeugen (also Compiler für das Hostsystem selbst) installiert wird. Allerdings ist die Installation häufig nicht trivial, sodass wir zunächst auf vorbereitete Pakete zurückgreifen beziehungsweise später ein Werkzeug zur automatisierten Erstellung der Toolchain einsetzen.

Im Folgenden wollen wir das Selbstbausystem auf den Raspberry Pi portieren. Daran lässt sich sehr gut der Umgang mit Cross-Entwicklungswerkzeugen demonstrieren. Wir fangen wieder mit dem Kernel an, um danach das Userland aufzusetzen. Die Entwicklung selbst soll unterhalb des Verzeichnisses `~/embedded/raspi/` stattfinden, das wir bereits angelegt haben.

3.3.1 Cross-Generierung Kernel

Um einen Kernel für den Raspberry Pi zu bauen, ist Folgendes zu tun (siehe Beispiel 3-8):

1. Toolchain installieren
2. Kernelquellen per git herunterladen
3. Kernel konfigurieren
4. Generieren

Für die Cross-Generierung des Kernels wird als Erstes eine Cross-Entwicklungsumgebung benötigt, die sich auf einem Ubuntu per »apt-get install gcc-arm-linux-gnueabi« leicht installieren lässt. Auch wenn diese (in der unter Ubuntu 12.04 zur Verfügung stehenden Version) weder Userland noch Bootloader generiert, wollen wir sie zunächst für den Kernel einsetzen.

Leider können Sie die bereits installierten Kernelquellen nicht weiter benutzen, da der über [http://www.kernel.org] herunterladbare Kernel in der Version 3.10.9 von Linus Torvalds den Raspberry Pi nicht vollständig unterstützt. Nehmen Sie daher einen bereits vorkonfektionierten Quellcode, der sich per `git` über Github unterhalb von `~/embedded/raspi/` installieren lässt. Der Umgang mit `git` — falls unbekannt — ist übrigens im Anhang C, *Git im Einsatz* in Kurzform beschrieben.

Vorteilhafterweise bringt der vorkonfektionierte Kernel die in Tabelle 3-7 aufgeführten Generierungs-Targets für den Raspberry Pi mit, die die Kernelkonfiguration erheblich vereinfachen. Der mit dem Target `bcmrpi_defconfig` generierte Kernel kann übrigens auch mit einem Raspbian eingesetzt werden, wobei einige nicht elementare Treiber als Module generiert werden.

Target	Bedeutung
bcmrpi_defconfig	Konfiguration für ein normales System
bcmrpi_cutdown_defconfig	Auf die wesentliche Funktionen reduziert, z.B. kein Debugging, Tracing oder Firewalling
bcmrpi_quick_defconfig	Schlanker Kernel, kaum Module, wenig Features
bcmrpi_emergency_defconfig	Schlanker Kernel für das Notsystem

Tabelle 3-7 *Vordefinierte Kernelkonfigurationen für den Raspberry Pi*

Für die Konfiguration und die Generierung des Linux-Kernels mit einem Cross-Compiler wird `make` mit dem zusätzlichen Parameter `ARCH=arm` aufgerufen [QuKu2011a]. Zusätzlich muss noch der Parameter `CROSS_COMPILE=arm-linux-gnueabi-` angegeben werden. Nur so werden die richtigen Cross-Compile-Werkzeuge gefunden, deren Namen sich aus den im Parameter stehenden Namensvorsatz »arm-linux-gnueabi-« und dem Werkzeugnamen selbst ergeben. Der Name des Cross-Compilers `gcc` lautet demnach »arm-linux-gnueabi-gcc«. Beispiel 3-8 zeigt im Detail die Kommandos, die in einem Terminal aufzurufen sind, um auf einem Ubuntu einen Raspberry Pi-Kernel zu generieren. Dabei müssen Sie insbesondere für das »Klonen« des Kernels (Herunterladen des gepatchten Kernelquellcodes per git) einige Minuten Zeit einplanen.

Beispiel 3-8 *Kommandos, um den Raspberry-Kernel auf einem Ubuntu zu generieren*

```
quade@felicia:~/embedded>
quade@felicia:~/embedded> cd raspi
quade@felicia:~/embedded/raspi> sudo apt-get \
   install gcc-arm-linux-gnueabi ncurses-dev
...
quade@felicia:~/embedded/raspi> git clone \
   https://github.com/raspberrypi/linux.git
...
quade@felicia:~/embedded/raspi> cd linux
quade@felicia:~/embedded/raspi/linux> git branch -a
* rpi-3.6.y
  remotes/origin/HEAD -> origin/rpi-3.6.y
  remotes/origin/master
  remotes/origin/rpi-3.10.y
  remotes/origin/rpi-3.2.27
  remotes/origin/rpi-3.6.y
  remotes/origin/rpi-3.8.y
  remotes/origin/rpi-3.9.y
  remotes/origin/rpi-patches
quade@felicia:~/embedded/raspi/linux> git checkout rpi-3.10.y
quade@felicia:~/embedded/raspi/linux> make ARCH=arm bcmrpi_defconfig
quade@felicia:~/embedded/raspi/linux> make ARCH=arm \
   CROSS_COMPILE=arm-linux-gnueabi- -j4
quade@felicia:~/embedded/raspi/linux>
```

Der Parameter »-j4« sorgt wieder für die Parallelverarbeitung auf einem Multicore-System. Der Kernel befindet sich nach einer erfolgreich durchgelaufenen Generierung im Linux-Quellcodeverzeichnis unter `~/embedded/raspi/linux/arch/arm/boot/zImage`.

Die Installation des Kernels auf der SD-Karte des Raspberry Pi wird nach der Generierung des Userlands beschrieben.

Kernel auf dem Raspberry Pi selbst bauen

Auch wenn es lange dauert, wenn der Raspberry Pi mit einem Raspbian betrieben wird, lässt sich der Kernel auf diesem nativ kompilieren. Loggen Sie sich dazu auf dem Raspberry Pi mit dem Login »pi« (Passwort »raspberry«) ein und werden Sie Superuser (`sudo su`). Sie können dann die gepatchten Kernelquellen installieren, konfigurieren und genererieren. Beispiel 3-9 zeigt Ihnen die dazu benötigten Befehle und zusätzlich noch die Kommandos, mit denen Kernel und Module installiert werden. Da der Kernel zusätzlich zum Standardkernel installiert wird, wird noch die Datei `/boot/config.txt` modifiziert. Dieser Schritt darf allerdings nur einmalig durchgeführt werden .

```
pi@raspberrypi ~ $ sudo su
root@raspberrypi:/home/pi# cd /usr/src
root@raspberrypi:/usr/src# git clone \
   https://github.com/raspberrypi/linux.git -b rpi-3.10.y
root@raspberrypi:# cd linux
root@raspberrypi:# make bcmrpi_defconfig
root@raspberrypi:# make menuconfig
root@raspberrypi:# make
root@raspberrypi:# make modules_install
root@raspberrypi:# cp arch/arm/boot/zImage /boot/linux-3.10.y
root@raspberrypi:# echo "kernel=linux-3.10.y" >>/boot/config.txt
```

Beispiel 3-9
Kernel auf dem Raspberry Pi selbst bauen

3.3.2 Cross-Generierung Userland

Um das Userland für den Raspberry Pi zu bauen, ist Folgendes zu tun:

1. Das für Qemu erstellte Userland kopieren
2. Das Init-Skript `rcS` anpassen
3. Die `inittab` anpassen
4. Cross-Entwicklungsumgebung erstellen
5. Busybox für die Cross-Entwicklung konfigurieren und Busybox kompilieren
6. Generierungsskript `mkrootfs.sh` anpassen
7. Das Userland per `mkrootfs.sh` zusammenbauen

Um das bereits in Abschnitt 3.2 vorbereitete Userland auf dem Raspberry Pi einsetzen zu können, muss es angepasst werden. Dazu kopieren Sie als Erstes die Quelldateien in das Verzeichnis `raspi`, legen für Skripte ein eigenes Verzeichnis an und kopieren dann noch das Generierungsskript `mkrootfs.sh` in das neue Verzeichnis:

```
quade@felicia:~> cd embedded/raspi/
quade@felicia:~/embedded/raspi> cp -r ../qemu/userland/ .
quade@felicia:~/embedded/raspi> mkdir scripts
quade@felicia:~/embedded/raspi/scripts> cp ../../qemu/mkrootfs.sh .
quade@felicia:~/embedded/raspi/scripts>
```

Die Anpassungen des Userlands betreffen zunächst die Dateien `inittab` (zur Unterstützung der seriellen Schnittstelle) und `rcS` (zur Konfiguration des Netzwerkinterface). Außerdem müssen noch zusätzliche Gerätedateien angelegt und die Busybox cross-kompiliert werden.

Beispiel 3-10 *Einfache Inittab für den Raspberry Pi <inittab>*

```
::sysinit:/etc/rcS
::askfirst:/bin/ash
ttyAMA0::askfirst:/bin/ash
```

Um im Userland nicht nur mit einer über USB angeschlossenen Tastatur und einem per HDMI angeschlossenen Monitor Ein- und Ausgaben tätigen zu können, soll die im Bereich eingebetteter Systeme häufig anzutreffende serielle Schnittstelle unterstützt werden. Dazu wird `init` über die `inittab` so konfiguriert, dass eine zweite Shell auf der seriellen Schnittstelle gestartet wird. Beispiel 3-10 zeigt die modifizierte Konfigurationsdatei, die um eine Zeile ergänzt wurde. Dieser entnehmen Sie, dass die serielle Schnittstelle auf dem Raspberry Pi über den Namen `ttyAMA0` angesprochen wird. Die zugehörige Gerätedatei `/dev/ttyAMA0` wird später über das Skript `mkrootfs.sh` angelegt.

```
quade@felicia:~/embedded/raspi> cd userland/target
quade@felicia:~/embedded/raspi/userland/target> gedit inittab &
```

Die Netzwerkkonfiguration muss angepasst werden, da das Ethernet-Interface auf dem Raspberry Pi eine Initialisierungszeit benötigt. Vorher lässt sich keine IP-Adresse konfigurieren. Die IP-Adresse selbst ist ebenfalls auszutauschen. Anstelle der für Qemu verwendeten IP-Adresse benutzen Sie eine aus Ihrem Netz. Die um die Initialisierungszeit von drei Sekunden (`sleep 3`) erweiterte Datei `rcS` inklusive der Änderung auf eine andere IP-Adresse (`192.168.178.99`) finden Sie unter Beispiel 3-11. Auskommentiert ist übrigens eine Zeile, bei der anstelle einer festen IP-Adresse ein DHCP-Client gestartet wird, der sich von einem heute meist im Netzwerk vorhandenen DHCP-Server die IP-Adresse dynamisch zweisen lässt. Falls Sie die Konfiguration per DHCP wünschen, müssen Sie die Zeile mit `ifconfig` durch Einfügen eines »#« am Zeilenanfang auskommentieren und die nachfolgende Zeile durch Löschen des ersten Zeichens aktivieren.

```
quade@felicia:~/embedded/raspi/userland/target> gedit rcS &
```

Im nächsten Schritt muss Busybox für den Raspberry Pi cross-kompiliert werden. Da die mit `apt-get` installierte Cross-Entwicklungsumgebung kein Userland generieren kann, ist eine eigene Umgebung aufzusetzen. Das ist allerdings sehr komplex und fehlerbehaftet. Die Raspberry Pi-Homepage schlägt hierfür `crosstool-ng` vor, besser und einfacher geht das allerdings über den Systembuilder `buildroot`, der von uns ohnehin in Kapitel 4 eingesetzt wird.

```
#!/bin/ash

mount -t proc none /proc
mount -t sysfs none /sys
sleep 3
ifconfig eth0 192.168.178.99
# udhcpc -s /etc/simple.script
httpd -c /etc/httpd.conf
```

Beispiel 3-11
Einfaches Skript zur Systemkonfiguration <rcS>

Mit den folgenden Kommandos laden Sie sich über das Internet den Buildroot-Quellcode herunter, konfigurieren diesen und lassen über diesen mithilfe des Targets `make toolchain` die Toolchain generieren:

```
quade@felicia:~/embedded/raspi/userland/target> cd ../..
quade@felicia:~/embedded/raspi> wget \
    http://www.buildroot.net/downloads/buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi> tar xvf buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi/buildroot-2013.05> make rpi_defconfig
quade@felicia:~/embedded/raspi/buildroot-2013.05> make toolchain
```

Insbesondere der Download dauert eine geraume Zeit. Wenn die Generierung der Toolchain erfolgreich verläuft, liegen im Verzeichnis `~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/` die notwendigen Programme. Unter `~/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot` befinden sich die zugehörigen Headerdateien und Bibliotheken. Verläuft die Generierung nicht erfolgreich, liegt das meistens daran, dass erforderliche Pakete nicht installiert sind. In diesem Fall ist es notwendig, die Fehlermeldung von Buildroot genau zu studieren und die fehlenden Pakete auf dem Entwicklungsrechner mithilfe von `apt-get` nachzuinstallieren.

Damit Busybox cross-kompiliert wird, rufen Sie im Unterverzeichnis `~/embedded/raspi/userland/busybox-1.21.1/make menuconfig` auf. Wählen Sie `Busybox Settings` und danach `Build Options`. Unter `Cross Compiler Prefix` tragen Sie jetzt `arm-linux-` ein und unter `Path to Sysroot` den Pfad zu den Headerdateien und Bibliotheken der Cross-Toolchain, die mit `sysroot` bezeichnet werden. Wichtig dabei: Sie müssen den Pfad absolut und nicht relativ angeben! In meinem Fall ist das `/home/quade/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/` (Abb. 3-9). Vergessen Sie nicht, meinen Loginnamen im Pfad durch den Ihrigen auszutauschen. Sollte der Pfad zu Sysroot falsch eingegeben sein, wird `make` mit einem Fehler abbrechen und dabei melden, dass es die Datei `limits.h` nicht finden kann.

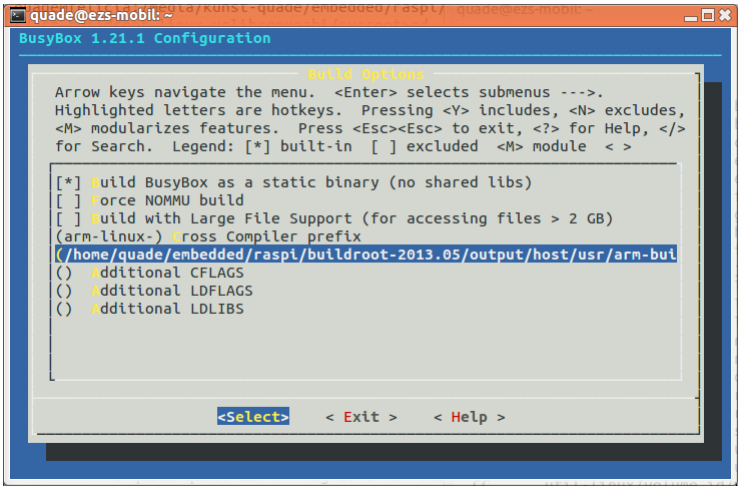


Abb. 3-9
Einstellungen in Busybox zur Cross-Kompilation

Falls Sie anstelle der festen IP-Adresse DHCP verwenden wollen, müssen Sie hierzu bei der Busybox-Konfiguration unter `Networking Utilities` den udhcp-client (`udhcpc`) auswählen. Udhcpc ruft, wenn es eine IP-Adresse zugewiesen bekommen hat, ein Skript auf, das dann für das Setzen der IP-Adresse zuständig ist. Busybox stellt im Verzeichnis `examples/udhcp/` das Skript `simple.script` zur Verfügung, das für unsere Zwecke ausreichend ist.

Außerdem muss vor dem Aufruf von `make` noch die Umgebungsvariable »PATH« so angepasst werden, dass der Pfad, in dem sich Cross-Compiler, Cross-Linker und die übrigen Cross-Entwicklungswerkzeuge befinden (`~/embedded/raspi/buildroot-2013.05/output/`), angehängt wird. Nach diesen Vorbereitungen cross-generiert ein `make` die Busybox und ein `make install` installiert die Systemprogramme im Verzeichnis `_install`.

```
quade@felicia:~/embedded/raspi> cd userland/busybox-1.21.1
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> export \
   PATH=$PATH:~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make menuconfig
...
# [Busybox Settings][Build Options] Cross-Werkzeug-Präfix
# [Busybox Settings][Build Options] Absoluten Pfad zu Sysroot eintragen
# [Networking Utilities][udhcpc] DHCP-Client auswählen
...
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> make install
```

Damit ist Busybox cross-generiert, das Userland kann zusammengebaut werden. Allerdings sind zuvor noch kleine Änderungen in `mkrootfs.sh` notwendig. Es muss noch die zusätzliche Gerätedatei `/dev/ttyAMA0` (Majornummer 204, Minornummer 64) angelegt werden, sodass Aus- und

Eingaben über die serielle Schnittstelle des Raspberry Pi möglich sind. Außerdem erwartet der Kernel das Programm `init` nicht wie bei der emulierten Variante unter `/sbin/init`, sondern im Root-Verzeichnis. Um das Problem zu lösen, legen wir mit `ln -s sbin/init loop/init` einen symbolischen Link an. Als Drittes schließlich muss für `udhcpc` (so er denn von Ihnen verwendet wird) noch das Skript kopiert werden, das aufgerufen wird, sobald der DHCP-Client eine IP-Adresse zugewiesen bekommen hat. Tragen Sie also die nachfolgenden Zeilen hinter `# MARK C` ein:

```
# MARK C
sudo mknod loop/dev/ttyAMA0 c 204 64
sudo ln -s sbin/init loop/init
sudo install -m 755 busybox-1.21.1/examples/udhcp/simple.script loop/etc
```

Last, but not least dürfen Sie nicht vergessen, den im Skript vorkommenden Pfadnamen »qemu« durch »raspi« zu ersetzen! Hier die Kommandos dazu im Überblick:

```
quade@felicia:~/embedded/raspi/userland/busybox-1.21.1> cd ../..
quade@felicia:~/embedded/raspi> cd scripts
quade@felicia:~/embedded/raspi/scripts> gedit mkrootfs.sh & # Änderungen
vornehmen
# ACHTUNG: Pfadnamen "qemu" durch "raspi" ersetzen
#          cd $MAINDIR/raspi/userland
# Ergänzungen vornehmen
quade@felicia:~/embedded/raspi/scripts> ./mkrootfs.sh # Userland generieren
```

3.3.3 Installation auf dem Raspberry Pi

Zur Installation des Selbstbau-Linux auf einer SD-Karte sind die folgenden Schritte notwendig:

1. Bootpartition erstellen, entweder im Selbstbau oder durch Installation von Raspbian
2. Eigenbau-Kernel auf der Bootpartition installieren
3. Userland auf der Systempartition installieren

Sind Kernel und Busybox cross-kompiliert und das Userland zusammengebaut, können die Komponenten auf einer SD-Karte installiert werden. Die SD-Karte bekommt zwei Partitionen: Auf der ersten, der Bootpartition, befinden sich die Bootprogramme und der Kernel, auf der zweiten Partition (Systempartition) befindet sich das Rootfilesystem beziehungsweise das Userland.

Bootpartition

Der Raspberry Pi setzt zwingend eine SD-Karte voraus, deren erste Partition mit einem FAT32-Filesystem (bei dem Partitionierungsprogramm `fdisk` ist die ID »c« einzustellen) formatiert ist. Auf dieser Partition müssen sich die in Tabelle 3-8 gelisteten Dateien befinden. Die erste Partition hat eine Größe von typischerweise 50 bis 100 MBytes.

Tabelle 3-8
Dateien auf der Bootpartition des Raspberry Pi

Name	Bedeutung
bootcode.bin	GPU-Firmware, die für das Booten zuständig ist
start.elf	GPU-Firmware, die für das Laden des Kernels zuständig ist
LICENCE.broadcom	Lizenzbestimmungen für die Nutzung des Bootloader-Codes
kernel.img	Linux-Kernel

Sie können diese Partition vorzugsweise wie in Kasten auf Seite 73 gezeigt von Grund auf selbst erstellen, oder alternativ durch Installation eines Raspbian wie in Abschnitt 2.3.1. Bei beiden Methoden stimmt übrigens die im Bootsektor hinterlegte Größe der zweiten Partition nicht mit der realen Größe überein. Der Funktion tut das aber keinen Abbruch. Die SD-Karte ist partitioniert und für die Aufnahme der eigentlichen Daten eingerichtet. Sie sollten übrigens sicherheitshalber die SD-Karte einmal kurz aus- und danach wieder einstecken. Das stellt sicher, dass die durch Ihre Aktionen geänderte Partitionstabelle neu eingelesen wird. Nach dem Einstecken der SD-Karte sollten sowohl die Boot- als auch die Systempartition eingehängt worden sein, die Bootpartition dabei unter dem Verzeichnis `/media/boot/`. Ist die Bootpartition unter einem anderen Verzeichnis eingehängt, tauschen Sie in den folgenden Erläuterungen die Pfadangabe entsprechend aus.

Sie müssen jetzt den selbstkompilierten Kernel auf die Bootpartition kopieren. Sie können diesen unter dem Standardnamen `kernel.img` ablegen oder aber auch einen individuellen Namen, beispielsweise `zImage`, wählen. Falls Sie den individuellen Namen vorziehen, benötigen Sie zusätzlich eine Datei `config.txt`, in der Sie die Zeile »kernel=zImage« eintragen. Nach dem Kopieren des Kernels kann die Bootpartition wieder ausgehängt werden:

```
quade@felicia:~/embedded/raspi> mv \
   /media/boot/kernel.img /media/boot/kernel.img.org

quade@felicia:~/embedded/raspi> cp \
   linux/arch/arm/boot/zImage /media/boot/kernel.img
quade@felicia:~/embedded/raspi> echo "kernel=zImage" \
   >>/media/boot/config.txt
quade@felicia:~/embedded/raspi> sudo umount /media/boot/
```

SD-Karten-Installation from scratch

Zur Partitionierung der SD-Karte wird das Programm `fdisk` eingesetzt, das als Parameter den Namen der Gerätedatei mitbekommt, über die auf die SD-Karte zugegriffen wird. Typischerweise ist das `/dev/sdc` oder `/dev/sdb`. Aber Vorsicht: Wer hier die falsche Gerätedatei eingibt, zerstört sein Hostsystem. Um die richtige Gerätedatei zu identifizieren, verfahren Sie wieder wie in Abbildung 2-9 gezeigt. Die letzten Einträge der nach Eingabe von `lsblk` erscheinenden Tabelle sollten die beiden Partitionen Ihrer SD-Karte sein. Gleich zu Anfang steht das zugehörige Device, beispielsweise `/dev/sdb1` für die Bootpartition und `/dev/sdb2` für das Rootfilesystem. Bei Ihnen könnte beispielsweise auch `/dev/sdc1` und `/dev/sdc2` ausgegeben sein. In der darüber stehenden Zeile ist die Gerätedatei aufgeführt, die die gesamte SD-Karte bezeichnet, beispielsweise `/dev/sdb`.

Starten Sie `fdisk` unter Angabe der Gerätedatei, die für die SD-Karte steht. Sollte es auf der verwendeten SD-Karte bereits irgendwelche Partitionen geben, löschen Sie diese mit dem Kommando »d«. Falls mehrere Partitionen vorhanden sind, müssen Sie noch die Nummer der zu löschenden Partition angeben. Sind alle Partitionen gelöscht, legen Sie durch Eingabe des Buchstabens »n« eine neue Partition vom Typ »primary« an. Diese Partition sollte mindestens 50 MByte groß sein. Der Partitionstyp ist danach auf »c« zu setzen, das einer FAT32-Partition entspricht. Beispiel 3-12 zeigt die Befehlssequenzen für eine unter `/dev/sdb` erreichbare SD-Karte, bei der per `fdisk` zunächst zwei vorhandene Partitionen gelöscht und danach zwei neue Partitionen angelegt wurden. Durch Eingabe des Buchstabens »p« zeigt Ihnen `fdisk` die jeweilige Partitionstabelle an, die aber erst mit Eingabe eines »w« wirklich geschrieben wird.

Beispiel 3-12
Befehlssequenz zum Anlegen der SD-Karte-Partitionen

```
root@felicia:~# fdisk /dev/sdb

Befehl (m für Hilfe): p

Platte /dev/sdb: 1977 MByte, 1977614336 Byte
61 Köpfe, 62 Sektoren/Spur, 1021 Zylinder, zusammen 3862528 Sektoren
Einheiten = Sektoren von 1 × 512 = 512 Bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Festplattenidentifikation: 0x00047c7a

Gerät     boot.    Anfang       Ende     Blöcke   Id  System
/dev/sdb1            2048     206847     102400    c  W95 FAT32 (LBA)
/dev/sdb2          206848    3862527    1827840   83  Linux

Befehl (m für Hilfe): d
Partitionsnummer (1-4): 1

Befehl (m für Hilfe): d
Partition 2 ausgewählt
```

```
Befehl (m für Hilfe): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partitionsnummer (1-4, Vorgabe: 1): 1
Erster Sektor (2048-3862527, Vorgabe: 2048): 2048
Last Sektor, +Sektoren or +size{K,M,G} (2048-3862527,
Vorgabe: 3862527): +100M

Befehl (m für Hilfe): t
Partition 1 ausgewählt
Hex code (L um eine Liste anzuzeigen): c
Der Dateisystemtyp der Partition 1 ist nun c (W95 FAT32 (LBA))

Befehl (m für Hilfe): n
Partition type:
   p   primary (1 primary, 0 extended, 3 free)
   e   extended
Select (default p): p
Partitionsnummer (1-4, Vorgabe: 2): 2
Erster Sektor (206848-3862527, Vorgabe: 206848): 206848
Last Sektor, +Sektoren or +size{K,M,G} (206848-3862527,
Vorgabe: 3862527): 3862527

Befehl (m für Hilfe): w
Die Partitionstabelle wurde verändert!

Rufe ioctl() um Partitionstabelle neu einzulesen.
```

Sobald die SD-Karte partitioniert ist, muss noch ein Filesystem angelegt werden. Dazu verwenden Sie die Kommandos `mkfs.vfat` für die Bootpartition und `mkfs.ext2` für die Systempartition. Achten Sie wieder darauf, die richtige Gerätedatei anzugeben, indem Sie das »x« im Namen der Gerätedatei durch den zugehörigen Laufwerksbuchstaben ersetzen:

```
quade@felicia:~/embedded> sudo mkfs.vfat -n boot /dev/sdx1
quade@felicia:~/embedded> sudo mkfs.ext2 -L rootfs -i 1024 /dev/sdx2
```

Jetzt können die Partitionen eingehängt werden. Auf die erste Partition müssen dann die Boot-Dateien (Tabelle 3-8) kopiert werden. Dazu laden Sie die Firmware über [https://github.com/raspberrypi/firmware] herunter und packen sie unter `~/embedded/raspi/` aus. Die gesuchten Dateien finden sich im Verzeichnis `firmware-master/boot/`:

```
quade@felicia:~/embedded> sudo mount /dev/sdX1 /mnt
quade@felicia:~/embedded> cd files
quade@felicia:~/embedded/files> \
   wget https://github.com/raspberrypi/firmware/archive/master.zip
quade@felicia:~/embedded> cd ../raspi
```

```
quade@felicia:~/embedded/raspi> unzip ../files/master.zip
...
quade@felicia:~/embedded/raspi> cd firmware-master/boot
quade@felicia:~/embedded/raspi/firmware-master/boot> \
   cp bootcode.bin start.elf LICENCE.broadcom /mnt/
```

Systempartition

Um das Userland zu installieren, kopieren Sie das zugehörige Image per `dd` auf die ausgehängte, zweite Partition:

```
quade@felicia:~/embedded/raspi> sudo umount /dev/sdx2
quade@felicia:~/embedded/raspi> sudo dd if=userland/rootfs.img of=/dev/sdx2
...
```

Damit ist die SD-Karte vorbereitet und kann im Raspberry Pi getestet werden. Stecken Sie die vorbereitete SD-Karte in den Minicomputer und schalten Sie diesen ein. Beobachten Sie die Ausgaben des Raspberry Pi. Wenn alles gut gegangen ist, müsste wieder eine Shell Ihre Kommandoeingabe ermöglichen. Per `ifconfig` können Sie sich die IP-Adresse ausgeben lassen. Mit dieser können Sie dann in einem Browser hoffentlich auf den Webserver zugreifen.

Wenn's schiefgeht ...

Sollte der Zugriff auf den Webserver nicht gelingen, gilt es herauszufinden, ob die Probleme im Kernel oder im Userland liegen. Installieren Sie dazu wieder den ursprünglichen Kernel, indem Sie auf der Bootpartition der SD-Karte den unter dem Namen `kernel.img.org` geretteten Kernel unter `kernel.img` ablegen. Stecken Sie dazu wieder die SD-Karte in den Entwicklungsrechner, wo diese typischerweise automatisch (in meinem Fall unter `/media/boot/`) gemountet wird. Danach können Sie den Kernel über die Konsole per `cp` zurückkopieren.

```
quade@felicia:~> cp /media/boot/kernel.img.org /media/boot/kernel.img
```

Stecken Sie die SD-Karte wieder in den Raspberry Pi. Sollte das System danach wie gewünscht funktionieren, liegen die Probleme wohl im Kernel. Zeigt sich jedoch das gleiche Symptom wie mit dem selbst gebautem Kernel, ist das Problem im Userland zu suchen.

Probleme mit dem Kernel

Bleibt die Meldung »Uncompressing linux« aus, ist der Kernel nicht richtig auf der SD-Karte in der Bootpartition installiert. Überprüfen Sie, ob in der Bootpartition die Datei »kernel.img« existiert und diese auch identisch mit der von Ihnen generierten Version ist.

Haben Sie für den Kernel nicht `kernel.img`, sondern einen individuellen Namen gewählt? Vielleicht fehlt dann der Eintrag in `config.txt`?

Bleibt der Kernel beim Booten stecken, durchforsten Sie die bis dahin ausgegebenen Logmeldungen. Diese geben typischerweise darüber Aufschluss, welcher Treiber beispielsweise fehlt. Ein häufiger Fehler ist das Fehlen des Treibers für das Filesystem (ext2 und ext4) oder für das Executable-Format (elf).

Probleme im Userland

Meldet der Kernel, dass er das Rootfilesystem nicht mounten kann, ist das Userland eventuell mit einem Filesystem erstellt worden, für das im Kernel keine Unterstützung existiert. Sie können das verwendete Filesystem am einfachsten identifizieren, wenn Sie die SD-Karte in den Entwicklungsrechner stecken und dann auf der Konsole den Befehl `mount` eingeben. Damit werden alle eingehängten Filesysteme inklusive ihres Typs ausgegeben. Überprüfen Sie anhand der Kernelkonfiguration, ob das Filesystem auf der SD-Karte vom Kernel unterstützt wird.

Konnte das Rootfilesystem eingehängt werden, aber deuten die nachfolgenden Meldungen auf dem Bildschirm des Raspberry Pi an, dass die in `rcS` spezifizierten Programme nicht gestartet werden konnten, ist das Userland möglicherweise nicht mit dem richtigen Cross-Compiler übersetzt worden. Um dies zu überprüfen, stecken Sie wiederum die SD-Karte in den Host, sodass die Partitionen gemountet werden. Wechseln Sie per `cd` in einem Terminal in das Rootfilesystem und überprüfen Sie dort mithilfe des Kommandos `file` den Typ des ausführbaren Programms `busybox`. Dieses muss ein Elf-Executable für ARM sein.

```
quade@felicia:/media/rootfs/bin$ file busybox
busybox: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically
linked, stripped
```

Bekommen Sie keine Shell, dürften die Einträge in der Inittab nicht korrekt sein. Hier ist möglicherweise das falsche Gerät eingetragen? Typischerweise sollte das für die serielle Schnittstelle des Raspberry Pi das Gerät `ttyAMA0` sein.

3.4 Bootloader »Das U-Boot«

Um das U-Boot auf dem Raspberry Pi zu installieren, sind die folgenden Schritte notwendig:

1. Quellcode herunterladen und Bootloader generieren
2. Programm zur Headergenerierung herunterladen
3. Bootloader mit Header versehen
4. Installation des mit Header versehenen Bootloaders auf die Bootpartition

5. Der Kernel wird mit einem U-Boot-Header versehen
6. Konfiguration erstellen, sodass der Kernel entweder von der SD-Karte oder über das Netzwerk gebootet wird

Im Umfeld eingebetteter (Open Source) Systeme werden häufig »Das U-Boot« und »Barebox« als Bootloader eingesetzt. Beide sind portierbar, skalierbar und bieten umfangreiche Funktionen.

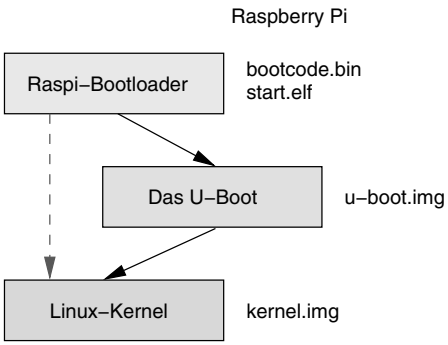


Abb. 3-10
Das dreistufiges Bootverfahren ermöglicht eine Systemauswahl.

Der Raspberry Pi setzt jedoch standardmäßig auf einen eigenen Bootloader, der einen Kernel und falls konfiguriert ein initiales Rootfilesystem (initramfs) in den Hauptspeicher lädt und diesem dann die Programmkontrolle übergibt. Funktionalitäten wie beispielsweise Netzwerk-Boot unterstützt er jedoch nicht. Bei einer Host-/Target-Entwicklung verkürzt das Booten über ein Netzwerk per `tftp` oder `nfs` die Turnaround-Zeit (Zeitdauer zwischen einer Systemänderung und dem Test der Änderung), also den Entwicklungszyklus, erheblich. Im Fall des Raspberry Pi erspart sich der Entwickler dadurch das ständige Wechseln der SD-Karte. Außerdem kann beim Booten zwischen verschiedenen Kerneln und — falls entsprechend gebaut — auch zwischen unterschiedlichen Rootfilesystemen gewählt werden.

Um diese Funktionalitäten nutzen zu können, wird das System so zusammengebaut, dass der originale Bootloader zunächst das U-Boot lädt und anschließend das U-Boot den Kernel und eventuell das Rootfilesystem in den Hauptspeicher transferiert. Damit ergibt sich das in Abbildung 3-10 dargestellte mehrstufige Bootverfahren.

Um den Umgang mit einem Bootloader kennenzulernen und um eine elegante Entwicklungsumgebung zur Verfügung zu stellen, installieren wir den Bootloader »Das U-Boot« auf der SD-Karte des Raspberry Pi.

Das Original lässt sich von der Webseite [http://www.denx.de] per `git` herunterladen.

```
quade@felicia:~/embedded> git clone git://git.denx.de/u-boot.git
```

Das U-Boot unterstützt zwar von Hause aus auch den Raspberry Pi, allerdings ist diese Unterstützung zurzeit unvollständig. Insbesondere funktioniert USB nicht »out of the box«, was wiederum für die Netzwerkverbindung notwendig ist. Daher ist es sinnvoll, auf ein alternatives Repository [https://github.com/gonzoua/u-boot-pi/tree/rpi] zuzugreifen. Auch dieses lässt sich per `git` herunterladen:

```
quade@felicia:~/embedded/raspi> mkdir bootloader
quade@felicia:~/embedded/raspi> cd bootloader
quade@felicia:~/embedded/raspi/bootloader> git clone \
   git://github.com/gonzoua/u-boot-pi
Cloning into 'u-boot-pi'...
remote: Counting objects: 193433, done.
remote: Compressing objects: 100% (40842/40842), done.
remote: Total 193433 (delta 153181), reused 190053 (delta 149922)
Receiving objects: 100% (193433/193433), 55.79 MiB | 1.11 MiB/s, done.
Resolving deltas: 100% (153181/153181), done.
quade@felicia:~/embedded/raspi/bootloader> cd u-boot-pi/
quade@felicia:~/embedded/raspi/u-boot-pi/bootloader> git branch -a
* rpi
  remotes/origin/HEAD -> origin/rpi
  remotes/origin/master
  remotes/origin/rpi
```

Achten Sie darauf, dass der Zweig »rpi« ausgewählt ist; der Branch »master« enthält nämlich die ungepatchten Originalquellen. Zur Generierung des Bootloaders wird ein geeigneter Cross-Compiler benötigt. Wir verwenden wieder die von Buildroot generierte Toolchain und müssen die Pfadvariable (`PATH`) in dem Terminal anpassen, in dem wir auch `make` aufrufen. Die Variable muss das Verzeichnis beinhalten, in dem die Cross-Werkzeuge sich befinden. Außerdem ist das Präfix für die Cross-Werkzeuge über die Umgebungsvariable CROSS_COMPILE zu exportieren. Zum Generieren selbst dient schließlich das Target `rpi_b`. Beispiel 3-13 zeigt die notwendigen Befehle. Nach der erfolgreichen Generierung liegt im Verzeichnis `u-boot-pi/` das Bootloader-Executable in der Datei `u-boot.bin` vor.

Beispiel 3-13 *Befehlssequenz zur Generierung des Bootloaders*

```
quade@felicia:~/embedded/raspi/bootloader/u-boot-pi> \
   export CROSS_COMPILE=arm-linux-
quade@felicia:~/embedded/raspi/bootloader/u-boot-pi> \
   PATH=$PATH:~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/
quade@felicia:~/embedded/raspi/bootloader/u-boot-pi> make rpi_b
```

Damit der originale Bootloader des Raspberry Pi den flexibleren Bootloader »Das U-Boot« laden kann, muss letzterer noch mit einem Header versehen werden. Hierzu laden Sie ein weiteres Repository per `git` von der Adresse [http://github.com/raspberrypi/tools]. In dem neuen Verzeichnis `tools/` (nicht zu verwechseln mit dem Verzeichnis `u-boot-pi/tools/`) befindet sich dann das Skript `imagetool-uncompressed.py`, das die Anpassung vornimmt. Das Skript schreibt das Ergebnis in die Datei `kernel.img`. Die zugehörigen Befehlsfolgen können Sie im Detail Beispiel 3-14 entnehmen. Dabei wird zusätzlich die generierte Datei `kernel.img` in `u-boot.img` umbenannt, um diese nicht mit dem Linux-Kernel zu verwechseln.

Beispiel 3-14
Befehlssequenz zur Umwandlung des Bootloader-Executables in ein Image

```
quade@felicia:~/embedded/raspi/bootloader/u-boot-pi> cd ..
quade@felicia:~/embedded/raspi/bootloader> \
   git clone http://github.com/raspberrypi/tools
quade@felicia:~/embedded/raspi/bootloader> cd tools/mkimage/
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> \
   ./imagetool-uncompressed.py ../../u-boot-pi/u-boot.bin
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> ls -lrt
insgesamt 328
-rwxrwxr-x 1 quade quade    822 Aug 28 18:24 imagetool-uncompressed.py
-rw-rw-r-- 1 quade quade    201 Aug 28 18:24 boot-uncompressed.txt
-rw-rw-r-- 1 quade quade    157 Aug 28 18:24 args-uncompressed.txt
-rw-rw-r-- 1 quade quade  32768 Aug 28 18:25 first32k.bin
-rw-rw-r-- 1 quade quade 288480 Aug 28 18:25 kernel.img
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> mv kernel.img \
   u-boot.img
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage>
```

Um den Bootloader zu verwenden, wird `u-boot.img` auf die SD-Karte in die Bootpartition kopiert. Da wir nicht den Standardnamen `kernel.img` verwenden, muss die Konfiguration `config.txt` des Raspberry Pi-Bootloaders um den Eintrag `kernel=u-boot.img` ergänzt beziehungsweise abgeändert werden. Stecken Sie hierzu wieder die SD-Karte in den Entwicklungsrechner. Dieser wird die Karte automatisch mounten; die für uns relevante Bootpartition erscheint dann voraussichtlich im Verzeichnis `/media/boot/`. Vergessen Sie nicht die Datei `/media/boot/config.txt` zu editieren. Hängen Sie anschließend die Partitionen aus.

```
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> cp u-boot.img \
      /media/boot/
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> gedit \
   /media/boot/config.txt &
# "kernel=u-boot.img" eintragen
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> sudo umount \
   /media/boot
```

Stecken Sie die Karte in den Raspberry Pi und starten Sie diesen. Wenn alles gut gegangen ist, haben Sie ein ähnliches Bild wie in Abbildung 3-11 gezeigt auf Ihrem Monitor. Den kompilierten Bootloader finden Sie übrigens auch unter dem Namen `u-boot.img` auf der Webseite zum Buch. Wenn Sie das Kommando `help` eingeben, zeigt das U-Boot Ihnen die implementierten Befehle. Allerdings geht das nur per serieller Schnittstelle. Die USB-Tastatur wird in der vorliegenden Version durch den Bootloader leider nicht unterstützt.

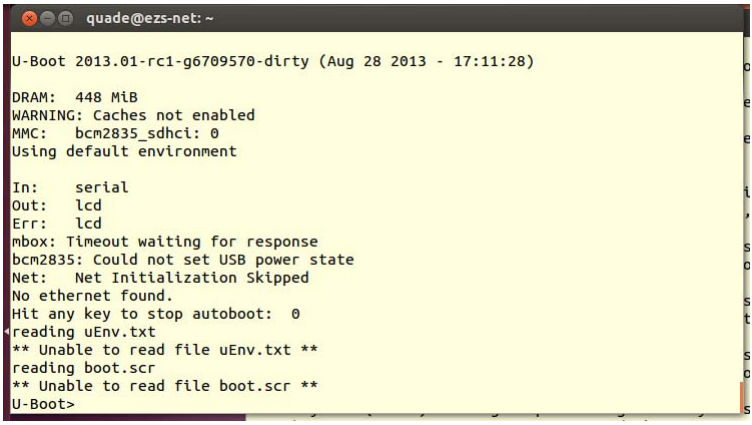


Abb. 3-11
Der Bootloader »Das U-Boot« nach dem Booten

3.4.1 Kernel von der SD-Karte booten

Um den Kernel per U-Boot von der SD-Karte zu booten, sind die folgenden Schritte notwendig:

1. Linux-Kernel mit einem U-Boot-Header generieren
2. Kernel auf die SD-Karte kopieren
3. U-Boot-Skript erstellen und ebenfalls auf die SD-Karte kopieren

Damit das U-Boot einen Linux-Kernel booten kann, benötigt der Kernel einen U-Boot-Header. Dieser wird mithilfe des Programms `mkimage` erstellt, das Teil der U-Boot-Distribution ist. Soll das Kernel-Build-System bei der Kernelgenerierung direkt einen per U-Boot ladbaren Kernel generieren, muss es auf `mkimage` zugreifen können. Dazu erwartet es das Tool als Teil der Cross-Entwicklungsumgebung, sodass wir es mit dem Präfix `arm-linux-` unter dem Namen `arm-linux-mkimage` in das Verzeichnis der Cross-Werkzeuge ablegen müssen:

```
quade@felicia:~/embedded/raspi/bootloader/tools/mkimage> \
   cd ../../u-boot-pi/tools/
quade@felicia:~/embedded/raspi/bootloader/u-boot-pi/tools> \
```

```
cp mkimage ~/embedded/raspi/buildroot-2013.05/output/host/usr/bin \
/arm-linux-mkimage
```

Danach können Sie den Kernel mit dem Target `uimage` neu generieren. Vergessen Sie dabei keinesfalls die Variablen ARCH und CROSS_COMPILE zu setzen und auch die Pfadvariable so konfiguriert zu haben, dass die Cross-Werkzeuge von `make` gefunden werden. Interessant sind übrigens die Ausgaben am Ende des Generierungsprozesses. Hier gibt `mkimage` die Adresse an, ab welcher Stelle der Kernel in den Hauptspeicher gelegt wird. In Beispiel 3-15 ist beispielsweise zu sehen, dass der Kernel an die Adresse (hexadezimal) »`00008000`« gelegt wird.

Beispiel 3-15
Linux-Kernel für den Bootloader generieren

```
quade@felicia:~> cd ~/embedded/raspi/linux/
quade@felicia:~/embedded/raspi/linux> make ARCH=arm \
   CROSS_COMPILE=arm-linux- uImage -j 4
...
  AS      arch/arm/boot/compressed/ashldi3.o
  AS      arch/arm/boot/compressed/piggy.gzip.o
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.10.9+
Created:      Wed Aug 28 20:13:20 2013
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    3225704 Bytes = 3150.10 kB = 3.08 MB
Load Address: 00008000
Entry Point:  00008000
  Image arch/arm/boot/uImage is ready
quade@felicia:~/embedded/raspi/linux>
```

Ist der Kernel für das U-Boot mit dem Namen `uImage` generiert, kann er auf die SD-Karte kopiert werden: SD-Karte also wieder in den Hostrechner stecken und danach den Kernel in die Bootpartition kopieren:

```
quade@felicia:~/embedded/raspi/linux> cp arch/arm/boot/uImage /media/boot/
quade@felicia:~/embedded/raspi/linux> sudo umount /media/boot
```

Nach dem Aushängen (umount) der SD-Karte können wir wieder testen. Dazu wird der Raspberry Pi gebootet. Der originale Bootloader lädt das U-Boot, das seinerseits auf dem Monitor beziehungsweise über die serielle Schnittstelle den Kommandoprompt `U-Boot>` der U-Boot-Shell ausgibt. Über diese Shell nimmt das U-Boot-Kommandos per serieller Schnittstelle entgegen. Ein USB-Keyboard wird in der hier verwendeten Version nicht unterstützt! In diesem Fall erstellen Sie wie weiter unten beschrieben direkt ein U-Boot-Skript.

Die wichtigsten U-Boot-Kommandos finden Sie in Tabelle 3-9. Die SD-Karte wird von der U-Boot-Shell über die Adresse »mmc 0:1« angesprochen. Dabei steht »mmc« für die SD-Karte (Multi Media Card), 0 für die erste SD-Karte (falls es mehrere gibt) und 1 für die erste Partition.

Um sich den Inhalt der SD-Karte anzeigen zu lassen, rufen Sie `fatls mmc 0` auf. Um das Image zu laden, verwenden Sie das U-Boot-Kommando `fatload`, mit dem der Kernel an die richtige Hauptspeicheradresse geladen werden kann. Ein `bootm` startet schließlich das Linux-System. Beispiel 3-16 zeigt den Vorgang im Detail.

Beispiel 3-16 *Kommandos, um den Kernel uImage per U-Boot zu starten*

```
U-Boot> mmc rescan
U-Boot> fatload mmc 0:1 00200000 uimage
reading uimage
3225776 bytes read in 524153 ms (5.9 KiB/s)
U-Boot> bootm 00200000
## Booting kernel from Legacy Image at 00200000 ...
   Image Name:   Linux-3.10.9+
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3225712 Bytes = 3.1 MiB
   Load Address: 00200000
   Entry Point:  00200000
   Verifying Checksum ... OK
   XIP Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
...
```

Automatisierung des Bootloaders

Das U-Boot bietet die Möglichkeit, Kommandos automatisiert abzuarbeiten und damit ohne Eingabe auszukommen. So kann der Bootloader eingesetzt werden, auch wenn keine serielle Schnittstelle zur Verfügung steht.

Die auszuführenden Kommandos müssen dazu in eine Datei (siehe Beispiel 3-17) geschrieben und mit dem Programm `mkimage` zu einem für den Bootloader zu lesenden Skript mit dem Namen `boot.scr` verarbeitet werden. Dieses Skript legen Sie auf die Bootpartition des Raspberry Pi ab (siehe Beispiel 3-18).

```
echo "Activating USB..."
#usb start
#mmc rescan
echo "Loading uImage..."
fatload mmc 0:1 00200000 uImage
bootm 00200000
```

Beispiel 3-17
Skriptdatei mit U-Boot-Kommandos <boot-sdcard.txt>

```
quade@felicia:~/embedded/raspi/bootloader> gedit boot-sdcard.txt &
quade@felicia:~/embedded/raspi/bootloader> arm-linux-mkimage  -A arm \
   -O linux -T script -C none -d boot-sdcard.txt boot.scr
Image Name:
Created:      Wed Aug 28 20:47:04 2013
Image Type:   ARM Linux Script (uncompressed)
Data Size:    126 Bytes = 0.12 kB = 0.00 MB
Load Address: 00000000
Entry Point:  00000000
Contents:
   Image 0: 118 Bytes = 0.12 kB = 0.00 MB
quade@felicia:~/embedded/raspi/bootloader> cp boot.scr /media/boot/
quade@felicia:~/embedded/raspi/bootloader> sudo umount /media/boot
```

Beispiel 3-18
Generieren von Skripten für den Bootloader »Das U-Boot«

Weitere Informationen finden Sie unter [elinux-uboot].

Name	Funktion
usb start	Aktiviert das USB-Subsystem
dhcp	Startet den DHCP-Client, um sich automatisch eine IP-Adresse zuweisen zu lassen
fatls mmc 0	Listet den Inhalt der ersten Partion auf der SD-Karte
fatload mmc 0 ${loadaddr} <filename>	Lädt die Datei »filename« von der ersten Partition der SD-Karte an die in der Variablen ${loadaddr} abgelegten Adresse
help	Listet sämtliche implementierten Kommandos auf
help <command>	Zeigt Hilfestellung zum Kommando »command« an
setenv <variable> <value>	Das Kommando belegt die Variable »variable« mit dem Wert »value«. Interessant sind beispielsweise die Variablen `serverip` und `ipaddr`.
reset	Führt ein reset aus, bootet also neu
bootm <addr_kernel> <addr_rootfs>	Bootet den Kernel, der ab »addr_kernel« im Speicher liegt und übergibt ihm »addr_rootfs« als Adresse für das Rootfilesystem.

Tabelle 3-9
Ausgewählte Kommandos des Bootloaders »Das U-Boot«

3.4.2 Netzwerk-Boot

Um Netzwerk-Boot auf dem Raspberry Pi zu nutzen, sind die folgenden Schritte notwendig:

1. tftp-Server installieren
2. Linux-Kernel auf den tftp-Server kopieren
3. U-Boot-Skript zum Netzwerk-Boot erstellen und mit Header versehen
4. Skript auf der SD-Karte installieren

Für die Entwicklung ist es besonders interessant, ein System über das Netzwerk booten zu können. Damit unterbleibt das zeitaufwendige Handling mit der SD-Karte.

Das Booten über das Netzwerk ist über die Protokolle `bootp` und `tftp` (Trivial File Transfer Protocol) realisiert. Das eingebettete System bekommt über `bootp` die Namen der Dateien mitgeteilt, die es danach von einem tftp-Server in den Hauptspeicher lädt. Anstelle des klassischen `bootp` kann auch ein moderner `dhcp`-Server eingesetzt werden. Allerdings ist Vorsicht angesagt, wenn im lokalen Netz bereits ein dhcp-Server aktiv ist.

In Kombination mit dem Raspberry Pi und dem Bootloader das U-Boot reichen allerdings ein zentraler dhcp-Server und ein auf dem Entwicklungsrechner installierter tftp-Server aus (siehe Abb. 3-12). Unter Ubuntu empfiehlt sich als tftp-Server der `tftpd-hpa`, der schnell per `apt-get install tftpd-hpa` über ein Terminal installiert ist. Nach einem Reboot des Entwicklungsrechners startet der Dienst allerdings nicht automatisch. Dies ist ein bekanntes Problem in Ubuntu 12.04, was mit einem »sudo service tftpd-hpa restart« (bis zum nächsten Reboot) gelöst werden kann.

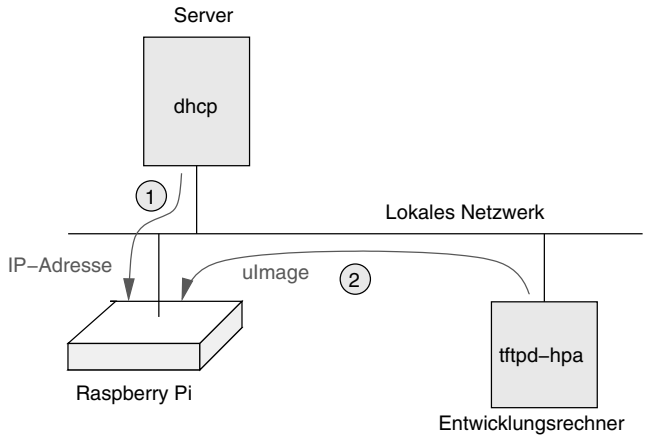


Abb. 3-12
Per tftp holt sich der Raspberry Pi den Kernel.

Eine Konfiguration des tftp-Servers tftpd-hpa ist weiter nicht notwendig. Er verwendet das Verzeichnis `/var/lib/tftpboot/` als Speicherort für die Dateien. Mit Root-Rechten ausgestattet können Sie die benötigten Dateien — bei uns zunächst der Kernel mit dem Namen `uImage` — in den Ordner `/var/lib/tftpboot/` kopieren:

```
quade@felicia:~/embedded/raspi> sudo apt-get install tftpd-hpa
quade@felicia:~/embedded/raspi> sudo cp linux/arch/arm/boot/uImage \
   /var/lib/tftpboot/
```

Sollten Sie per serieller Schnittstelle auf den Raspberry Pi zugreifen, können Sie das Booten des Kernels über das Netzwerk vorab testen. Greifen Sie jedoch nur per USB-Tastatur und Monitor auf den Raspberry Pi zu, müssen Sie wie unten beschrieben direkt ein Bootskript erzeugen.

Für den manuellen Vorabtest verhindern Sie durch Drücken einer beliebigen Taste (Meldung auf dem Bildschirm »Hit any key to stop autoboot«) das automatische Laden eines Kernels. Danach aktivieren Sie USB (Kommando `usb start`), lassen sich mit dem Kommando `dhcp` eine Ethernetadresse zuweisen und gleichzeitig den Kernel an die Adresse `00200000` (hexadezimal) laden. Per `bootm` wird der Kernel aktiviert. Beispiel 3-19 zeigt die Befehlsfolge im Detail, wobei davon ausgegangen wird, dass der tftp-Server die IP-Adresse 192.168.178.25 hat. Diese Adresse müssen Sie gegen die IP-Adresse Ihres Entwicklungsrechners (auf dem der tftp-Server aktiviert wurde) ändern. Theoretisch können Sie die IP-Adresse des tftp-Servers auch weglassen, praktisch gibt es aber unter Umständen weitere tftp-Server im Netz, mit denen der Raspberry Pi Verbindung aufnimmt. Solche Geräte sind auch Router, wie beispielsweise die populäre FritzBox.

```
usb start
dhcp 00200000 192.168.178.25:uImage
bootm 00200000
```

Beispiel 3-19
Befehlsfolge zum Booten des Kernels von einem Server

Falls Sie eine feste IP-Adresse verwenden wollen, ist dies über die folgenden U-Boot-Kommandos möglich:

```
usb start
setenv ipaddr 192.168.178.99
setenv serverip 192.168.178.25
tftpboot uImage
bootm 00200000
```

Bauen Sie jetzt aus diesen Kommandos ein Skript. Schreiben Sie dazu die Kommandos wieder in eine Datei, beispielsweise `boot-net.txt`, und

führen Sie dann das nachfolgende Kommando aus (Voraussetzung dafür ist, dass die Pfadvariable auf die Cross-Werkzeuge zeigt):

```
quade@felicia:~/embedded/raspi/bootloader> \
   arm-linux-mkimage  -A arm -O linux \
   -T script -C none -d boot-net.txt boot-net.scr
quade@felicia:~/embedded/raspi/bootloader> \
   cp boot-net.scr /media/boot/boot.scr
quade@felicia:~/embedded/raspi/bootloader> sudo umount /media/boot
```

3.5 Initramfs: Filesystem im RAM

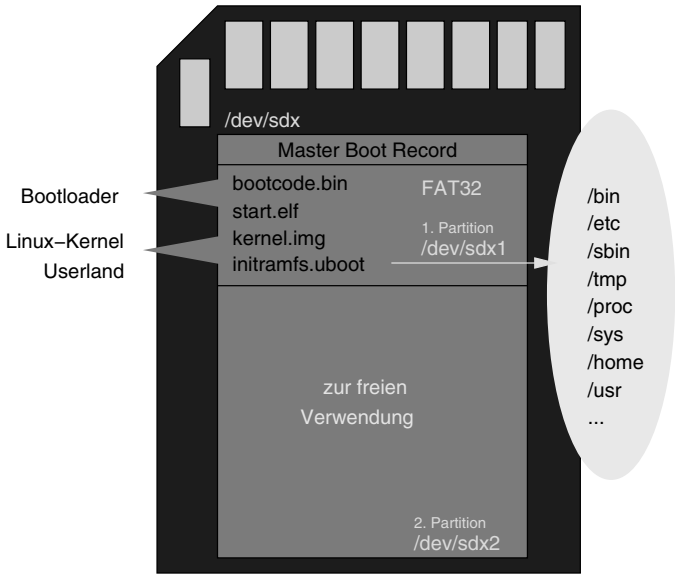


Abb. 3-13
Initramfs: Das Rootfilesystem liegt als Image vor.

Im Embedded-Bereich wird das Rootfilesystem auf zwei unterschiedliche Arten realisiert. In der einfacheren Variante liegt es im Flash. Dazu wird auf dem Flash eine Partition reserviert, auf der dann die Verzeichnisse und Dateien abgelegt werden (Abb. 2-3). Bei der zweiten und für den Bereich der eingebetteten Systeme — falls genügend Hauptspeicher zur Verfügung steht — besseren Variante liegt das Rootfilesystem in Form einer Abbilddatei (Image) als so genanntes Initramfs vor (Abb. 3-13). Dieses Image liegt dann auf dem Flash-Speicher und wird beim Booten in den Hauptspeicher geladen. Das hat mehrere Vorteile:

- Konsistentes Filesystem bei jedem Neustart
- Booten ohne Filesystemcheck
- Einfache Systemaktualisierung durch Austausch des Initramfs-Image

- Schneller Zugriff (da im Hauptspeicher)
- Im RAM keine Limitierung der Schreibzyklen
- Mehrerer Rootfilesysteme zur Auswahl

Von Nachteil ist hingegen, dass Modifikationen am Rootfilesystem mit jedem Neustart verloren gehen, es sei denn, sie sind im Image durchgeführt worden oder werden anderweitig auf dem Flash abgelegt und beim Neustart eingelesen. Außerdem wird für das Rootfilesystem Hauptspeicher benötigt, der den übrigen Komponenten dann nicht mehr zur Verfügung steht.

Das Rootfilesystem im RAM gibt es bei Linux in zwei Ausprägungen: einmal als klassische RAM-Disk und zum anderen als initiales Rootfilesystem (Initramfs). Das Initramfs wird vorzugsweise eingesetzt. Während die klassische RAM-Disk einen reservierten Teil des Hauptspeichers wie eine Festplatte behandelt, die formatiert und falls gewünscht sogar partitioniert werden kann, erspart das Initramfs den Filesystemtreiber und setzt ohne Zwischenschicht direkt auf interne Schnittstellen auf. Das hat zwei Vorteile: Erstens sind die Zugriffe schneller und zweitens gibt es keine feste Größe für das Rootfilesystem. Anders als eine RAM-Disk, die beispielsweise vier, acht oder 16 MByte groß ist, benötigt das RAM-Filesystem immer genau so viel Speicher, wie die darin abgelegten Daten belegen. Ist nur eine Datei mit beispielsweise 12 kByte gespeichert, werden auch nur 12 kByte Memory verwendet. Werden mehr Dateien abgelegt, dann wächst das RAM-Filesystem dem Bedarf entsprechend mit. Natürlich kann dieser Vorteil auch ein Nachteil sein, nämlich dann, wenn amoklaufende oder böswillige Threads ein solches RAM-Filesystem mit Bytes fluten.

Um ein Initramfs (oder auch eine RAM-Disk) nutzen zu können, müssen die Inhalte in den Hauptspeicher geschaufelt werden. Unter Linux werden die Inhalte dazu als CPIO-Archiv (die Imagedatei) dort abgelegt. Der Kernel entpackt beim Booten das Archiv und legt es gemäß seiner internen Datenstrukturen in den Hauptspeicher.

Der Transport des Archivs in den Hauptspeicher wird durch den Bootloader vorgenommen und kann auf zwei Arten erfolgen:

1. separat vom Kernel als eigenständige Imagedatei und
2. hinten am Kernel angehängt (Kernel und Userland bilden eine Einheit).

Bei Version a) ist von Vorteil, dass kein Kernelgenerierungsprozess angestoßen werden muss, wenn nur Änderungen am Filesystem vorgenommen wurden. Außerdem können leichter unterschiedliche Kombinatio-

nen von Kernel und Userland ausprobiert werden. (Generierung aufwendiger, Laden schneller)

Version b) hat den Vorteil, dass nur ein Transfer in den Hauptspeicher notwendig ist und dass der Kernel direkt die Adresse des Initramfs-Archivs kennt. Hier geht also das Laden (etwas) schneller vonstatten (Generierung schneller, Laden länger).

Ob Variante a) oder b) eingesetzt wird, wird im Kernel konfiguriert, und zwar konkret durch den Auswahlpunkt [General setup][Initramfs source file(s)]. Ist die Auswahl leer, erwartet der Kernel ein getrenntes Initramfs. Ist hier der Pfad zu den Dateien beziehungsweise zu einem CPIO-Archiv (Dateierweiterung ».cpio«) angegeben, nimmt der Kernel die Dateien beziehungsweise das Archiv und hängt es an den Kernel an.

Wird die Variante mit einem separaten Userland gewählt, muss der Bootloader nicht nur den Kernel, sondern eben auch das CPIO-Archiv des Userlands in den Hauptspeicher kopieren. Dazu muss der Bootloader geeignet konfiguriert werden.

Der Standard-Bootloader des Raspberry Pi bietet hierfür in der Datei `/boot/config.txt` der Bootpartition auf der SD-Karte die Option »initramfs <rootfs.cpio>« an. Anzumerken ist, dass bei dieser Option das sonst in der Datei übliche Gleichheitszeichen (»=«) nicht verwendet wird.

Seperates Userland

Um mit U-Boot neben dem Kernel ein separates Userland zu booten, sind die folgenden Schritte notwendig:

1. Kernel mit Initramfs-Unterstützung konfigurieren und mit U-Boot-Header (uImage) generieren
2. Generierungsskript `mkrootfs.sh` editieren und unter MARK D Folgendes ergänzen:
   a. Per `find` aus dem Rootfilesystem ein CPIO-Archiv erzeugen
   b. Das CPIO-Archiv mit dem U-Boot-Header versehen und entweder auf den tftp-Server oder auf die SD-Karte kopieren
3. Das Userland per `mkrootfs.sh` generieren und zum Booten bereitstellen.
4. U-Boot-Skript `boot-net.txt` zum Booten von Kernel und Initramfs erstellen
5. U-Boot-Skript mit Raspberry Pi-Bootloader-Header versehen (`boot.scr`)
6. Skript auf die SD-Karte kopieren

Das U-Boot kann ebenfalls für ein separates Userland konfiguriert werden. Wesentlich dabei ist das Kommando `bootm`.

Das U-Boot-Kommando `bootm addr_kernel <addr_rootfs> <addr_devtree>` hat bis zu drei Parameter. Der erste Parameter (`addr_kernel`) gibt hexadezimal die Adresse an, ab der die Datei `uImage` im Hauptspeicher abgelegt wurde. Der zweite Parameter (`addr_rootfs`) wird nur benötigt, wenn auch ein initiales RAM-Filesystem verwendet wird. Damit wird die zugehörige Adresse in hexadezimaler Form abgelegt. Der dritte Parameter schließlich ist für die Adresse des Devicetree vorgesehen, eine Datei, die eine Beschreibung der Hardware für den Linux-Kernel enthält. Devicetrees werden von uns bisher nicht verwendet. Per definitionem werden übrigens die Adressen des Initramfs und des Devicetree auf einer ARM-Plattform (Raspberry Pi) in einer *tagged list* (ATAG Register) übergeben. Das U-Boot belegt die Register und der Linux-Kernel liest sie beim Booten aus und kennt daher die Adresse des CPIO-Archivs.

Die Adressen, an denen Kernel und Initramfs im Speicher abgelegt werden, dürfen sich nicht überschneiden. Den Kernel haben wir bisher bei 0x00200000 abgelegt, das Initramfs legen wir beispielsweise ab 0x00c00000 ab.

Damit muss der Bootloader erst den Kernel an die Adresse 0x00200000 und danach das Initramfs-Archiv an die Adresse 0x00c00000 laden (oder umgekehrt). Liegen Kernel und Archiv auf der SD-Karte, setzen Sie `fatload` ein, für den Netzwerkzugriff verwenden Sie `dhcp` und `tftpboot` (siehe Beispiel 4-2).

```
U-Boot> fatload mmc 0:1 00200000 uimage
U-Boot> fatload mmc 0:1 00c00000 initramfs.uboot
U-Boot> bootm 00200000 00c00000
```

Um ein Initramfs zu nutzen, sind also die folgenden Schritte notwendig:

1. In der Kernelkonfiguration wird Initramfs aktiviert [General setup] [Initial RAM filesystem and RAM disk (initramfs/initrd) support] und der Kernel per `make uImage` neu generiert.

2. Auf dem Entwicklungssystem muss ein Verzeichnis mit der Struktur und den Dateien des Rootfilesystems existieren. In unserem Fall ist nichts weiter zu tun, da die Daten bereits in einer Abbilddatei (rootfs.img) vorliegen. Diese wird während der Generierung im Verzeichnis `loop/` eingehängt.

3. Aus dem Verzeichnisbaum mit den Dateien des Rootfilesystems wird ein komprimiertes CPIO-Archiv erstellt. Wechseln Sie dazu in das Verzeichnis des Initramfs (also in das Verzeichnis `loop/`) und rufen Sie die nachfolgende Kommandofolge auf:

   ```
   find . | cpio -o -H newc | gzip -9 >initramfs.cpio.gz
   ```

4. Das Image benötigt noch einen U-Boot-Header. Dazu wird `arm-linux-mkimage` eingesetzt:

```
arm-linux-mkimage -A arm -O linux -T ramdisk -C none \
   -n "Root Filesystem" -d initramfs.cpio.gz initramfs.uboot
```

5. Kernel `arch/arm/boot/uImage` und Image `initramfs.uboot` werden entweder auf die SD-Karte oder per `cp` auf den tftp-Server abgelegt.

Damit Sie diese Befehle nicht immer von Hand durchführen müssen, tragen Sie sie unter »MARK D« in das Generierungsskript `mkrootfs.sh`, inklusive des Kopierens auf den tftp-Server, ein. Sie finden die Befehle im vollständigen Skript, zu sehen in Beispiel 3-20.

Beispiel 3-20 *Das vollständige Skript zur Generierung des Rootfilesystems <mkrootfs.sh>*

```
#!/bin/bash

MAINDIR=~/embedded

cd $MAINDIR/raspi/userland
dd if=/dev/zero of=rootfs.img bs=4k count=2k
mke2fs -i 1024 -F rootfs.img
sudo mount -o loop rootfs.img loop
sudo rsync -a busybox-1.21.1/_install/ loop
mkdir loop/dev
mkdir loop/etc
mkdir loop/proc
mkdir loop/sys
sudo mknod loop/dev/console c 5 1
sudo mknod loop/dev/null c 1 3
sudo mknod loop/dev/tty0 c 4 0
sudo mknod loop/dev/tty1 c 4 1
sudo mknod loop/dev/tty2 c 4 2
sudo chown -R root.root loop
# MARK A
sudo install -m 644 ./target/inittab loop/etc
sudo install -m 755 ./target/rcS loop/etc
# MARK B
sudo mkdir -p loop/var/www/cgi-bin/
sudo install -m 644 target/index.html loop/var/www/
sudo install -m 755 target/ps.cgi loop/var/www/cgi-bin/
sudo install -m 644 target/httpd.conf loop/etc/
# MARK C
sudo mknod loop/dev/ttyAMA0 c 204 64
sudo ln -s sbin/init loop/init
sudo install -m 755 busybox-1.21.1/examples/udhcp/simple.script loop/etc
# MARK D
(cd loop; sudo find . | cpio -o -H newc | gzip -9) > initramfs.cpio.gz
arm-linux-mkimage -A arm -O linux -T RAMDISK -C none -a 0 \
        -n "Root Filesystem" -d initramfs.cpio.gz initramfs.uboot
sudo cp initramfs.uboot /var/lib/tftpboot/
```

```
# Aushaengen
sudo umount loop
```

Die passenden Kommandos für ein U-Boot-Skript, mit denen Kernel und Rootfilesystem per tftp geladen werden, finden Sie in Beispiel 3-21. Der Kernel wird an die Adresse 0x00200000 und das Initramfs an 0x00f00000 geladen. Sie müssen allerdings wieder die angegebene IP-Adresse gegen diejenige Ihres tftp-Servers austauschen.

Beispiel 3-21
Bootloader-Kommandos zum Laden von Kernel und Rootfilesystem <boot-net.txt>

```
usb start
dhcp 00200000 192.168.178.25:uImage
tftpboot 00f00000 192.168.178.25:initramfs.uboot
bootm 00200000 00f00000
```

Modifizieren Sie also die Datei `~/embedded/raspi/bootloader/boot-net.txt` gemäß Beispiel 3-21, erzeugen Sie daraus wieder ein U-Boot-Skript, kopieren Sie dieses auf die eingehängte SD-Karte und hängen Sie die SD-Karte wieder aus und stecken Sie sie in den Raspberry Pi:

```
quade@felicia:~/embedded/raspi/bootloader> \
   arm-linux-mkimage  -A arm -O linux \
   -T script -C none -d boot-net.txt boot-net.scr
quade@felicia:~/embedded/raspi/bootloader> cp boot-net.scr \
   /media/boot/boot.scr
quade@felicia:~/embedded/raspi/bootloader> sudo umount /media/boot
```

Mit dem nächsten Boot holt sich der Raspberry Pi den Kernel und das Rootfilesystem vom tftp-Server in den Hauptspeicher und übergibt danach dem Kernel die Kontrolle. Sie müssten in etwa die in Beispiel 3-22 dargestellten Ausgaben auf Ihrem Bildschirm verfolgen können.

Beispiel 3-22
Laden von Kernel und Userland per tftp: Ausgaben des Bootloaders

```
U-Boot 2013.01-rc1-g6709570 (Aug 28 2013 - 18:07:08)

DRAM:  448 MiB
WARNING: Caches not enabled
MMC:   bcm2835_sdhci: 0
Using default environment

In:    serial
Out:   lcd
Err:   lcd
mbox: Timeout waiting for response
bcm2835: Could not set USB power state
Net:   Net Initialization Skipped
No ethernet found.
Hit any key to stop autoboot:  3
reading uEnv.txt
```

```
** Unable to read file uEnv.txt **
reading boot.scr
189 bytes read in 13302 ms (0 Bytes/s)
Running bootscript from mmc0 ...
## Executing script at 00200000
(Re)start USB...
USB0:   Core Release: 2.80a
scanning bus 0 for devices... 3 USB Device(s) found
       scanning usb for storage devices... 0 Storage Device(s) found
       scanning usb for ethernet devices... 1 Ethernet Device(s) found
Waiting for Ethernet connection... done.
BOOTP broadcast 1
BOOTP broadcast 2
DHCP client bound to address 192.168.178.147
Using sms0 device
TFTP from server 192.168.178.25; our IP address is 192.168.178.147
Filename 'uImage'.
Load address: 0x200000
Loading: *#################################################################
    #################################################################
    #################################################################
done
Bytes transferred = 2818384 (2b0150 hex)
Waiting for Ethernet connection... done.
Using sms0 device
TFTP from server 192.168.178.25; our IP address is 192.168.178.147
Filename 'initramfs.uboot'.
Load address: 0xf00000
Loading: *################
done
Bytes transferred = 221090 (35fa2 hex)
## Booting kernel from Legacy Image at 00200000 ...
   Image Name:   Linux-3.6.11
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2818320 Bytes = 2.7 MiB
   Load Address: 00008000
   Entry Point:  00008000
   Verifying Checksum ... OK
## Loading init Ramdisk from Legacy Image at 00f00000 ...
   Image Name:   Root Filesystem
   Image Type:   ARM Linux RAMDisk Image (uncompressed)
   Data Size:    221026 Bytes = 215.8 KiB
   Load Address: 00000000
   Entry Point:  00000000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
```

```
[    0.000000] Booting Linux on physical CPU 0
[    0.000000] Initializing cgroup subsys cpu
...
[    1.045044] Trying to unpack rootfs image as initramfs...
[    1.082339] Freeing initrd memory: 212K
...
```

Falls Ihr System doch nicht bootet, finden Sie zur Eingrenzung der fehlerhaften Komponente auf der Webseite zum Buch die in Tabelle 3-10 gelisteten Komponenten, die Sie sukzessive abhängig vom Fehlerbild gegen die von Ihnen erstellten austauschen.

Dateiname	Bedeutung
3.5./linux-3.10.9.config	Konfiguration für den Linux-Kernel
3.5./uImage	Linux-Kernel für das U-Boot
3.5./busybox-1.21.1.config	Busybox-Konfiguration
3.5./mkrootfs.sh_D	Skript zur Generierung des Rootfilesystems
3.5./initramfs.uboot	Initramfs für Netzwerk-Boot per U-Boot
3.5./boot-net.txt	Quelldatei für das U-Boot-Skript
3.5./boot.scr	Skriptdatei für das U-Boot

Tabelle 3-10
Konfigurations- und Ergebnisdateien zum Booten per Netzwerk

Noch ein Hinweis: Das bis hierhin erstellte System ist bei Weitem nicht ausgereift und bedarf noch einiges Feintunings in der Konfiguration und der Auswahl der Befehle.

4 Systembuilder Buildroot

Nachdem wir im ersten Teil ein eingebettetes System von Grund auf zunächst für den Emulator, dann für den Raspberry Pi aufgebaut haben, soll im Folgenden ein Systembuilder zum Einsatz kommen. Dieser übernimmt nicht nur die Aufgabe des Skripts `mkrootfs.sh`, sondern generiert durch Aufruf eines Kommandos sämtliche Systemteile: Firmware, Kernel, Rootfilesystem und die funktionsbestimmende Applikation. Außerdem sorgt ein Systembuilder wie Buildroot selbstständig dafür, dass die benötigten Werkzeuge, insbesondere die Cross-Entwicklungsumgebung, zur Verfügung stehen. Ein Systembuilder vereinfacht damit nicht nur die Generierung des Systems, er bietet zudem erheblich mehr Funktionalität. Auf der einen Seite gibt es mehr Pakete, die ausgewählt und eingebunden werden können (beispielsweise einen SSH- oder VPN-Server), auf der anderen Seite lassen sich auch mit wenig Aufwand komplexere Systemteile, wie beispielsweise ein Multiuser-Management (Login über Passwörter) realisieren.

4.1 Überblick

Die Aufgaben des Systemgenerators bestehen darin,

- Systemkomponenten über Pakete zur Auswahl zu stellen,
- die Konfiguration der Pakete zu ermöglichen,
- die (Cross-)Toolchain (Compiler, Linker, Libs, ...) zu generieren sowie
- das eingebettete System selbst zu realisieren:
  - Generierung der einzelnen Systemteile (Firmware, Kernel, Rootfilesystem, Applikationen)
  - Zusammenbau des Userlands.

Das Ergebnis sind typischerweise Imagedateien, die auf dem Target (Embedded System) eingespielt werden.

Es gibt diverse Systemgeneratoren. Bekannt sind unter anderem:

- Buildroot [http://www.buildroot.net]
- OpenEmbedded [http://www.openembedded.org]
- Yocto [https://www.yoctoproject.org/]

OpenEmbedded und das davon abgeleitete Yocto sind zwei Generatoren, die auf der einen Seite sehr mächtig, auf der anderen Seite aber auch ressourcenhungrig sind und die eine hohen Einarbeitungsaufwand erfordern. Erste Schritte mit Yocto für den Raspberry Pi finden Sie beispielsweise unter [cnxsoft2013]. Buildroot dagegen ist übersichtlich, verbraucht erheblich weniger Ressourcen, generiert ein System deutlich schneller und ist damit für einfache Projekte bestens geeignet. Es bietet sich zum Einstieg in die Technik der Systembuilder an und wird hier vorgestellt.

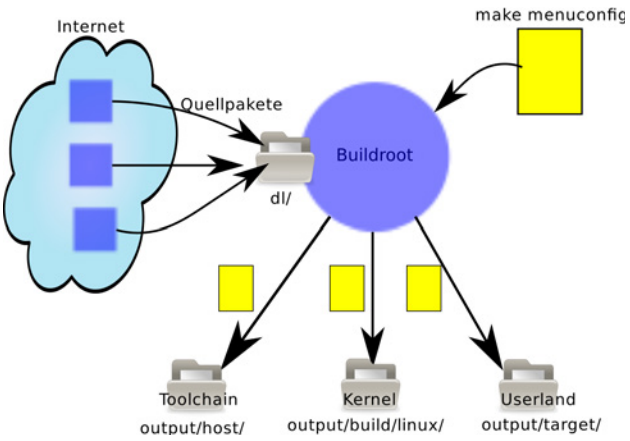


Abb. 4-1 *Grundstruktur des Systembuilders Buildroot*

Buildroot

Bei Buildroot handelt es sich um eine Sammlung von Skripten zur Konfiguration, Generierung und zum Zusammenbau einer Distribution. Dazu lädt Buildroot den Quellcode der ausgewählten Pakete aus dem Internet herunter. Es installiert eine Cross-Development-Toolchain, die zum ausgewählten Target passt und aus Compiler, Binutils, Target-Libraries und Debugger (für Host und Target) besteht. Buildroot ist Open Source und kostenfrei. In den folgenden Abschnitten wird Buildroot vorgestellt und die wichtigsten Funktionen werden demonstriert. Ausführlichere und ergänzende Informationen finden Sie übrigens unter [bruserman] oder auch im Internet, beispielsweise bei [cellux2013].

Buildroot generiert ein System aus unterschiedlichen Systemteilen:

- Busybox
- Netzwerk (inklusive Server)
- Grafik (GUI)
- Audio
- Kernel
- Bootloader
- Eigene Software

Zur Konfiguration greift es auf den gleichen Mechanismus zurück wie das Kernel-Build-System und das Multicall-Binary Busybox. Dabei bietet es unter anderem die folgenden Generierungsziele (Targets) an:

- `make menuconfig`
  Hiermit wird Buildroot selbst konfiguriert und insbesondere die verwendete Hardwareplattform, das Target, ausgewählt.
- `make busybox-menuconfig`
  Startet die Konfiguration von `busybox`. Allerdings sollte Busybox erst nach dem ersten Kompilationslauf konfiguriert werden. Nach der Konfiguration ist ein erneuter Kompilationslauf notwendig, der durch Aufruf von `make` gestartet wird.
- `make uclibc-menuconfig`
  Dieser Aufruf startet die Konfiguration der Lib uclibc. Allerdings dürfte für die meisten Einsatzfälle diese Lib in ihren Defaulteinstellungen geeignet konfiguriert sein. Auch uclibc ist erst nach einem ersten Generierungslauf zu konfigurieren.
- `make linux-menuconfig`
  Startet die Kernelkonfiguration.
- `make toolchain`
  Generiert nur die Cross-Toolchain.
- `make dl`
  Dieses Target lädt alle benötigten Pakete aus dem Internet. Damit kann die Systemgenerierung hinterher offline erfolgen.
- `make clean`
  Clean löscht alle generierten Pakete. Das nachfolgende `make` erzeugt dann alles neu. Vergessen Sie anschließend nicht — falls Sie den Bootloader U-Boot verwenden — das Programm `mkimage`, wie in Abschnitt 3.4.1 beschrieben, nachzuinstallieren. Auch eventuell durch-

geführte Änderungen an der Kernelkonfiguration (zum Beispiel die Aktivierung des Initramfs), der Busybox-Konfiguration und der Konfiguration der uclibc (beispielsweise die Unterstützung starker Hashverfahren wie in Abschnitt 7.1.3 erläutert) sind verloren und müssen erneuert werden. Mithilfe der in Abschnitt 4.5 beschriebenen Methode lassen sie sich aber vorher sichern und nachher wieder einspielen.

- `make help`
  Help zeigt die wichtigsten, von Buildroot unterstützten Targets an.
- `make <paketname>-reconfigure`
  Dieses Target veranlasst Buildroot, das Paket »*paketname*« neu zu konfigurieren.
- `make <paketname>-rebuild`
  Mit Rebuild lassen sich einzelne Pakete neu generieren. »make linux-rebuild« generiert beispielsweise einen neuen Kernel.

Durch `make` aktiviert, generiert Buildroot Images (Abbilddateien), indem die benötigten Quellcodepakete, inklusive des Quellcodes für die Cross-Development-Toolchain, von dem jeweiligen Quellcodeserver geladen werden (Abb. 4-1). Die Toolchain wird konfiguriert, gepatcht, generiert und installiert. Der Quellcode für die Pakete wird — wo notwendig — ebenfalls gepatcht, die Pakete werden automatisch konfiguriert und generiert. Außerdem wird der Kernel kompiliert. Schließlich wird das Rootfilesystem generiert und mit der erstellten Software gefüllt. Zuletzt erstellt Buildroot aus dem Kernel und dem Rootfilesystem die Imagedateien in auswählbaren Formaten.

Tabelle 4-1 zeigt die Verzeichnisstruktur von Buildroot.

Tabelle 4-1 *Verzeichnisstruktur von Buildroot*

Verzeichnis	Bedeutung
configs/	Konfigurationsdateien für die unterstützten Plattformen
output/images/	Enthält die generierten Imagedateien (Kernel, Bootloader, Rootfilesystem)
output/build/	Hier sind die zur Generierung eines Images notwendigen Werkzeuge abgelegt
output/host/usr/	Das Verzeichnis enthält die generierten Cross-Development-Werkzeuge
dl/	Ablage für die Download-Dateien
docs/	Dokumentation zu Buildroot. Der Einstieg erfolgt per Webbrowser über `docs/buildroot.html`.
package/	Enthält zu jedem Paket notwendige Patches oder Zusatzdateien wie beispielsweise Startskripts
toolchain/	Enthält Konfigurationen beziehungsweise Patches für die Entwicklungssoftware selbst (Toolchain)

Änderungen am erzeugten Filesystem können über das Verzeichnis `system/skeleton/` vorgenommen werden. Auf diese Art fügen Sie beispielsweise eine eigene Konfigurationsdatei unter `/etc/` ein.

Zugriffsrechte des zu erzeugenden Filesystems lassen sich über die Datei `system/device_table.txt` einstellen.

Darüber hinaus besteht die Möglichkeit, ein Postbuild-Skript zur Verfügung zu stellen. Dieses wird nach der Generierung der Pakete aufgerufen, aber bevor das Rootfilesystem zusammengebaut wird. Mithilfe des Skripts können beispielsweise Dateien in das Rootfilesystem kopiert werden (siehe Abschnitt 4.3.2).

Buildroot kann natürlich nur für die Plattformen Systeme generieren, die dem Werkzeug bekannt sind. Der Aufruf von `make help` im Buildroot-Ordner zeigt unter anderem die standardmäßig unterstützten Plattformen an. In der Version 2013.05 sind das beispielsweise 55. Darunter sind einige von Qemu unterstützte Plattformen ebenso wie der Raspberry Pi.

4.2 Buildroot-Praxis

Um mit Buildroot ein Embedded Linux für den Raspberry Pi zu bauen, sind die folgenden Schritte notwendig:

1. Download und Installation von Buildroot
2. Defaultkonfiguration von Buildroot (Auswahl der Plattform Raspberry Pi)
3. Konfiguration des Embedded Linux (Auswahl und Konfiguration der zu generierenden Pakete)
4. Generierung der Programme für das Hostsystem und Generierung der Image- und Archivdateien für das Embedded Linux
5. Installation des Embedded Linux auf der SD-Karte oder auf dem tftp-Server

In Abschnitt 3.3.2 haben wir Buildroot eingesetzt, damit es die Cross-Entwicklungsumgebung generiert. Jetzt nutzen wir Buildroot, um damit sämtliche Teile eines eingebetteten Systems, also insbesondere Kernel, Userland und Applikation, zu bauen. Dieses System ist direkt auf dem Raspberry Pi einsatzfähig.

Die Schritte dazu sind in Abbildung 4-2 dargestellt. Als Erstes wird Buildroot installiert, anschließend konfiguriert und Werkzeuge und System werden generiert. Die Installation des Kernels und die Installation des generierten Userlands muss dann vom Anwender selbst vorgenommen werden. Der erste Generierungsvorgang dauert länger, da hier zu-

nächst die auf dem Entwicklungsrechner laufenden Werkzeuge gebaut werden müssen.

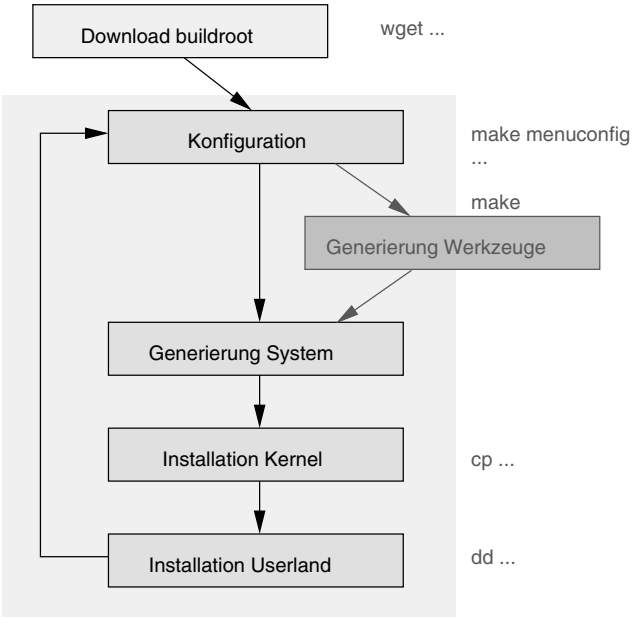


Abb. 4-2 *Vorgehen beim Einsatz von Buildroot*

Falls noch nicht geschehen, laden Sie als Erstes Buildroot aus dem Internet. Das Target `rpi_defconfig` konfiguriert das System für den Raspberry Pi. Sie sollten danach noch einmal `make menuconfig` aufrufen (Abb. 4-3) und die einzelnen Auswahlpunkte durchgehen.

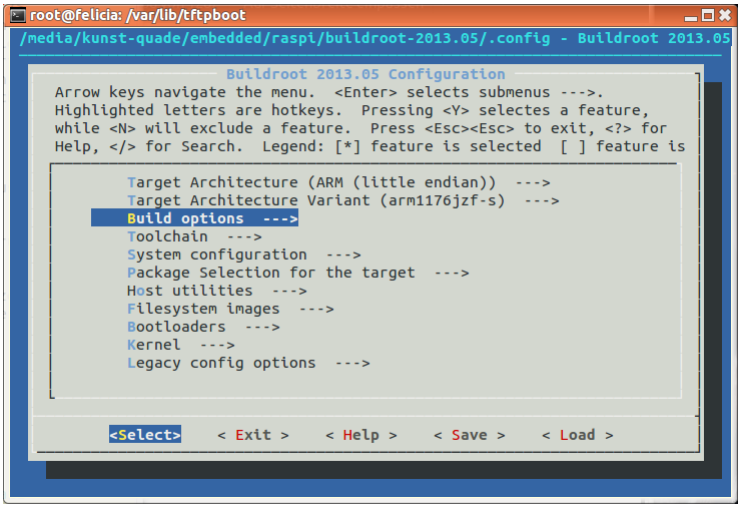


Abb. 4-3 *Konfigurationsoberfläche von Buildroot*

Sollten Sie den Raspberry Pi über eine serielle Schnittstelle kontrollieren, tauschen Sie unter [System configuration][(tty1) Port to run a getty (login prompt) on] »tty1« gegen »ttyAMA0« aus. Außerdem müssen Sie die Pakete »rpi-firmware« [Package Selection for the target][Hardware handling][Misc devices firmwares][rpi-firmware] und »rpi-userland« [Package Selection for the target][Hardware handling] [rpi-userland] abwählen, da sich diese in der gewählten Version von Buildroot nicht übersetzen lassen. Eventuell ist das Problem in neueren Versionen behoben. Installieren Sie — falls nicht bereits geschehen — Git. Sollte es bei der Generierung sonst noch Probleme geben, probieren Sie zunächst ein `make clean`, bevor Sie nochmals mit `make` den Prozess anstoßen.

```
quade@felicia:~> cd ~/embedded/raspi
quade@felicia:~/embedded/raspi> sudo apt-get install git
quade@felicia:~/embedded/raspi> wget http://www.buildroot.net/downloads/
buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi> tar xvf buildroot-2013.05.tar.bz2
quade@felicia:~/embedded/raspi/buildroot-2013.05> make rpi_defconfig
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
# Für die serielle Schnittstelle wählen Sie ttyAMA0 an Stelle von tty1 aus
#     [System configuration][(ttyAMA0) Port to run a getty (login prompt) on]
# Abwählen des Paketes rpi_firmware
#     [Package Selection for the target][Hardware handling]
#     [Misc devices firmwares][rpi-firmware]
# Abwählen des Paketes rpi_userland
#     [Package Selection for the target][Hardware handling] [rpi-userland]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Nach erfolgreicher Generierung finden sich die Ergebnisse im Verzeichnis `embedded/raspi/buildroot-2013.05/output/images`.

4.2.1 Installation auf der SD-Karte

Um das Embedded Linux auf der SD-Karte zu installieren, sind die folgenden Schritte notwendig:

1. Vorbereiten der SD-Karte (partitionieren und formatieren)
2. Installation des Kernels auf der Bootpartition und eventuell Anpassen der Datei `config.txt`
3. Installation des Userlands auf der zweiten Partition durch Formatieren (falls noch nicht geschehen), Mounten und Auspacken des Rootfilesystem-Archivs

Voraussetzung zur Installation des Embedded Linux ist eine wie in Kasten auf Seite 73 oder alternativ wie in Abschnitt 2.3.1 gezeigte partitionierte und formatierte SD-Karte.

Der von Buildroot generierte Kernel ist dann leicht per `cp` zu installieren:

```
quade@felicia:~> cd ~/embedded/raspi/buildroot-2013.05>
quade@felicia:~/embedded/raspi/buildroot-2013.05> \
   sudo cp output/images/zImage /media/boot/kernel.img
```

Eventuell benennen Sie den Kernel jedoch um und tragen in `/media/boot/config.txt` den Namen des neuen Kernels ein.

Da Buildroot vom Userland kein Image, sondern ein Archiv erzeugt hat, wird es etwas anders auf der zweiten Partition der SD-Karte installiert. Am einfachsten ist es, diese Partition durch Anlegen eines neuen Filesystems zu überschreiben. Bei der Gelegenheit geben wir dem Filesystem einen einprägsamen Namen, unter dem es dann automatisch beim Einführen der SD-Karte unter dem Ordner `/media/` eingehängt werden wird.

Nach dem Anlegen des neuen Filesystems können Sie die Partition entweder selbst mounten oder durch erneutes Ein- und Ausstecken automatisch mounten lassen. Daraufhin packen Sie das von Buildroot erzeugte Archiv in der zweiten Partition per `tar` aus. Beispiel 4-1 zeigt den Vorgang, wobei Sie wieder das »x« durch den Buchstaben des Laufwerks ersetzen müssen, das die SD-Karte repräsentiert.

Beispiel 4-1
Installation des Buildroot-Userlands auf der SD-Karte

```
quade@felicia:~/embedded/raspi> sudo umount /dev/sdx2
quade@felicia:~/embedded/raspi> sudo mkfs.ext4 -i 1024 -L rootfs /dev/sdx2
# SD-Karte einmal entfernen und wieder einstecken oder
# alternativ selber mounten
# mkdir /media/rootfs
# mount /dev/sdx2 /media/rootfs
quade@felicia:~/embedded/raspi> cd /media/rootfs
quade@felicia:~/media/rootfs> sudo \
   tar xvf ~/embedded/raspi/buildroot-2013.05/output/images/rootfs.tar
quade@felicia:~/media/rootfs> cd ..
quade@felicia:~/media/> sudo umount /dev/sdx2
quade@felicia:~/media/> sudo umount /media/boot
```

Falls die Installation erfolgreich war, meldet sich das System wie in Abbildung 4-4 dargestellt. Buildroot hat übrigens standardmäßig zwei Logins angelegt: den User »root« und den User »default«.

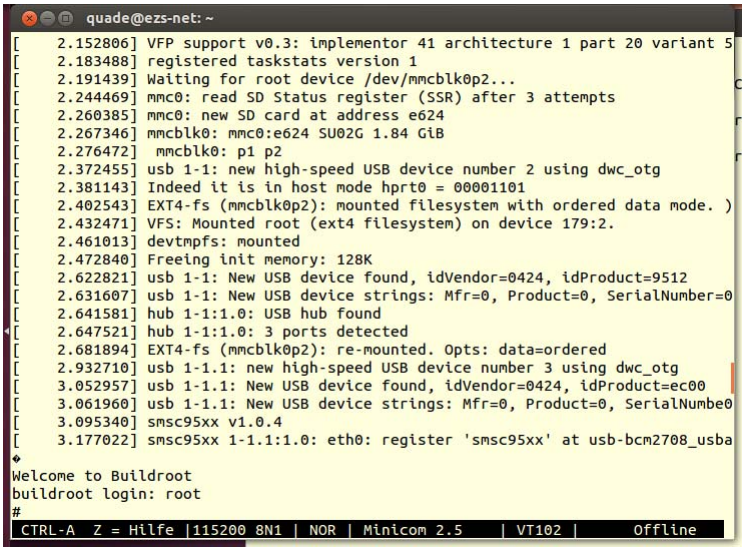


Abb. 4-4
Der Raspberry Pi bootet das Buildroot-System.

Wenn's schiefgeht ...

Sollte das per Systembuilder generierte System nicht booten, können die auf den Webseiten zum Buch abgelegten Dateien bei der Fehlereingrenzung helfen. Um den Kernel als Ursache auszuschließen, installieren Sie als Erstes den Kernel `4.2.1./zImage` unter dem Namen `kernel.img` auf der ersten Partition der SD-Karte. Sollte dieser Kernel nicht booten, dürfte die Bootloader-Konfiguration nicht stimmen. Stellen Sie sicher, dass in der Datei `config.txt` auf der SD-Karte entweder keine Zeile mit dem Namen `kernel` auftaucht oder dieser Eintrag folgendermaßen aussieht: »kernel=kernel.img«.

Bootet der Kernel, aber Sie bekommen keinen Prompt, können Sie zunächst das vorbereitete Userland `4.2.1./rootfs.img.buildroot_1` ausprobieren (falls dieses die Endung ».gz« trägt, müssen Sie es zuvor mit `gunzip` entpacken). Das Userland-Image können Sie mit dem Kommando `dd` auf die Linux-Partition der SD-Karte kopieren:

```
quade@felicia:~/embedded/raspi> gunzip rootfs.img.buildroot_1.gz
quade@felicia:~/embedded/raspi> sudo dd if=rootfs.img.buildroot_1 \
   of=/dev/sdx2
```

Ohne `dd` können Sie alternativ die Systemdateien auch per `tar` auf die vorbereitete Partition des Rootfilesystems kopieren. Diese finden Sie in der Datei `4.2.1./rootfs.tar.buildroot_1`. Als Letztes schließlich finden Sie noch die Datei `4.2.1./config.buildroot_1`, die die Buildroot-Konfiguration enthält.

Sie können die Konfiguration mit der Ihrigen vergleichen oder diese anstelle der Datei `.config` in das Hauptverzeichnis von Buildroot kopieren. Ein nachfolgendes `make` generiert dann das System neu.

4.2.2 Netzwerk-Boot per U-Boot

Einer der großen Vorteile des Bootloaders U-Boot ist die Möglichkeit, über das LAN (Local Area Network) zu booten. Das erleichtert die Entwicklung eines eingebetteten Systems signifikant. Beim Netzwerk-Boot wird das Userland (Rootfilesystem) in Form eines Initramfs zur Verfügung gestellt. Dieses kann entweder zusammen mit dem Kernel als eine Datei oder aber auch separat (als zweite Datei) neben dem Kernel zur Verfügung gestellt werden. Während die erste Variante einfacher in der Handhabung und schneller beim Booten ist, bietet die zweite Variante mehr Flexibilität. Änderungen am Userland sind nämlich völlig unabhängig vom Kernel.

Grundsätzlich müssen für beide Varianten folgende Schritte vorgenommen werden:

1. Der Kernel muss so generiert werden, dass er das Feature Initramfs unterstützt und im Fall der ersten Variante ein Initramfs erzeugt und an den Kernel anhängt.
2. Kernel und eventuell das separate Initramfs müssen mit einem U-Boot-Header versehen werden.
3. Die mit Header versehenen Dateien werden auf dem tftp-Server abgelegt.
4. U-Boot wird installiert und konfiguriert.

Rootfs als Anhängsel des Kernels

Um insbesondere in der Entwicklungsphase das Embedded Linux über Netz booten zu können, sind für den Kernel inklusive Initramfs die folgenden Schritte notwendig:

1. In der Buildroot-Konfiguration die Feature »initramfs« und »Kernel-Binary-Format `uImage`« auswählen
2. Per Buildroot das Embedded Linux generieren
3. Das generierte System (Datei `uImage`) auf den tftp-Server kopieren
4. U-Boot auf die SD-Karte installieren
5. U-Boot-Skript `boot-uimage.txt` erstellen
6. Das Skript `boot-uimage.txt` mit einem U-Boot-Header versehen
7. Das Skript unter dem Namen `boot.scr` auf die SD-Karte kopieren

Als Erstes wird Buildroot so konfiguriert, dass es ein Initramfs erzeugt. In der Buildroot-Konfiguration ist dazu die Auswahl [Filesystem ima-

ges][cpio the root filesystem (for use as an initial RAM filesystem)] zu treffen. Als Kompressionsmethode wählen Sie `gzip` aus.

Standardmäßig versieht Buildroot nicht die Ergebnisdateien mit den von U-Boot benötigten Metainformationen. Sobald Sie aber in der Konfiguration unter [Kernel][Kernel binary format (uImage)] das Format `uImage` auswählen, erstellt Buildroot mit dem nächsten `make` den Kernel und hängt dort das Rootfilesystem als CPIO-Archiv an. Außerdem ergänzt Buildroot durch Aufruf von `mkimage` noch den U-Boot-Header. Das Ergebnis wird unter dem Namen `uImage` in das Verzeichnis `buildroot-2013.05/output/images/` abgelegt. Die Ergebnisdatei `uImage` wird danach auf dem tftp-Server abgelegt.

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
#    [Filesystem images][cpio the root filesystem (for use as an initial
RAM filesystem)]
#    [Filesystem images][Compression method (gzip)]
#    [Filesystem images][initial RAM filesystem linked into linux kernel]
#    [Kernel][Kernel binary format (uImage)]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make linux-reconfigure
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
quade@felicia:~/embedded/raspi/buildroot-2013.05> sudo cp \
   output/images/uImage /var/lib/tftpboot/
```

Um diese Datei automatisiert beim Booten des Raspberry Pi von U-Boot über das Netzwerk in den Hauptspeicher zu laden und dann den Kernel zu starten, installieren Sie als Erstes das U-Boot wie in Abschnitt 3.4 beschrieben. Schreiben Sie danach ein kurzes U-Boot-Skript zum Laden über LAN. Allerdings müssen Sie die IP-Adresse (hier 192.168.178.25) gegen die IP-Adresse von Ihrem tftp-Server austauschen! Hier das Skript:

```
usb start
dhcp 00200000 192.168.178.25:uImage
bootm 00200000
```

Das Skript (`boot-uimage.txt`) selbst muss ebenfalls mit dem U-Boot-Header versehen werden. `arm-linux-mkimage` hilft dabei, wobei es zuvor wie in Abschnitt 3.4.1 beschrieben in das Verzeichnis der Toolchain installiert worden sein muss. Nach dieser Bearbeitung wird das Image in die Bootpartition der SD-Karte unter dem Namen `boot.scr` abgelegt. Dort wird es dann mit dem nächsten Reboot ausgeführt:

```
# SD-Karte muss im Rechner eingesteckt, Bootpartition gemountet sein
quade@felicia:~/embedded/raspi> cd bootloader
quade@felicia:~/embedded/raspi/bootloader> \
    cp tools/mkimage/u-boot.img /media/boot/u-boot.img
quade@felicia:~/embedded/raspi/bootloader> gedit /media/boot/config.txt &
    # In die Datei "kernel=u-boot.img" eintragen
    ...
```

```
quade@felicia:~/embedded/raspi/bootloader> gedit boot-uimage.txt &
    # Skriptdatei mit obigem Inhalt erstellen
    ...
quade@felicia:~/embedded/raspi/bootloader> arm-linux-mkimage  -A arm \
   -O linux -T script -C none -d boot-uimage.txt boot-uimage.scr
quade@felicia:~/embedded/raspi/bootloader> cp boot-uimage.scr \
   /media/boot/boot.scr
quade@felicia:~/embedded/raspi/bootloader> sudo umount /media/boot
```

Wenn's schiefgeht ...

Um das Kommando `arm-linux-mkimage` aufrufen zu können, muss die Umgebungsvariable PATH die Pfade der Cross-Toolchain kennen:

```
PATH=$PATH:~/embedded/raspi/buildroot-2013.05/output/host/usr/bin/
```

Bootet das System nicht, ist als Erstes anhand der Ausgaben des Raspberry Pi beim Booten festzustellen, ob es am Bootloader, dem Kernel oder dem Initramfs liegt. Die Boot-Nachrichten lassen sich am einfachsten über die serielle Schnittstelle verfolgen.

Sie können die Komponenten jeweils wieder gegen die auf der Webseite zum Buch zur Verfügung gestellten Komponenten testen.

Um auszuschließen, dass es sich um Probleme mit U-Boot handelt, rufen Sie die folgenden Befehle auf (unter der Voraussetzung, die Dateien vom Webserver zum Buch sind bereits in das Verzeichnis `files` kopiert worden):

```
quade@felicia:~embedded> cp ~/embedded/files/4.2.2. \
   /boot-uimage.scr_1 /media/boot/boot.scr
quade@felicia:~embedded> cp ~/embedded/files/4.2.2./config.txt \
   /media/boot/config.txt
```

Meldet der Bootloader, dass er den Kernel inklusive Initramfs nicht laden kann, ist eventuell der tftp-Server nicht betriebsbereit. Starten Sie diesen auf dem Entwicklungsrechner auf einer Konsole (Terminal) mit dem folgenden Kommando:

```
quade@felicia:~/embedded> sudo service tftpd-hpa restart
```

Gibt es Probleme mit dem Laden des Kernels, überprüfen Sie noch einmal das U-Boot-Skript `boot-uimage.txt`. Hier findet sich möglicherweise nicht die korrekte IP-Adresse von Ihrem tftp-Server. Editieren Sie die Datei und versehen Sie sie mit einem U-Boot-Header. Danach muss sie auf die Bootpartition der SD-Karte kopiert werden.

Gibt es Probleme mit dem Kernel? Dann verwenden Sie zum Test den Kernel mit Initramfs von der Webseite. Kopieren Sie diesen auf den tftp-Server:

```
quade@felicia:~/embedded> sudo cp files/4.2.2./uImage_Initramfs \
   /var/lib/tftpboot/uImage
```

Kernel und Rootfilesystem separat

Um insbesondere in der Entwicklungsphase das Embedded Linux, bestehend aus Kernel und separatem Initramfs, über das Netz booten zu können, sind die folgenden Schritte notwendig:

1. In der Buildroot-Konfiguration die Feature »initramfs« und »Kernel-Binary-Format `uImage`« auswählen
2. Embedded Linux durch Buildroot per Target `linux-reconfigure` generieren (Ergebnis `uImage` und `rootfs.cpio.gz`)
3. Das generierte Initramfs mit einem U-Boot-Header versehen (`initramfs.uboot`)
4. Das generierte System (`uImage` und `initramfs.uboot`) auf den tftp-Server kopieren
5. U-Boot-Skript `boot-buildroot.txt` erstellen
6. Das Skript `boot-buildroot.txt` mit einem U-Boot-Header versehen
7. Das Skript unter dem Namen `boot.scr` auf die SD-Karte kopieren
8. Falls noch nicht geschehen: U-Boot (das Programm selbst) auf der SD-Karte installieren

Die zweite Variante, Kernel und Rootfilesystem separat zu generieren und zu laden, bietet dem Entwickler mehr Flexibilität, wenngleich sie vom Raspberry Pi wegen des zweifachen Netzzugriffes langsamer gebootet wird. Da sie nicht direkt (out of the box) von Buildroot unterstützt wird, ist sie auch aufwendiger.

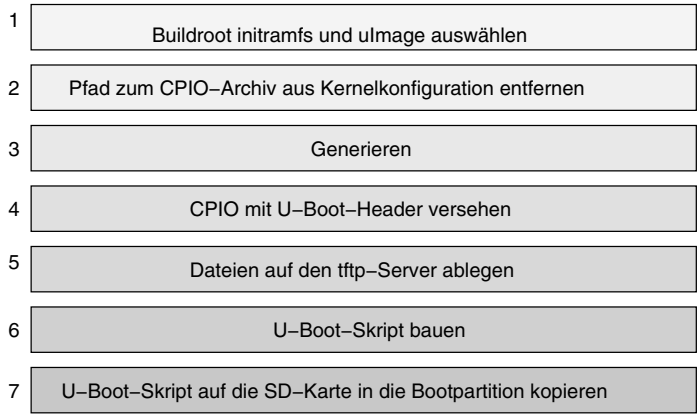


Abb. 4-5
Schritte zum Netzwerk-Boot von Kernel und Initramfs

Als Erstes wird Buildroot wieder so konfiguriert, dass es ein Initramfs erzeugt (Abb. 4-5). In der Buildroot-Konfiguration ist dazu die Auswahl [Filesystem images][cpio the root filesystem (for use as an initial RAM

filesystem)] zu treffen. Die Kompressionsmethode sollte `gzip` sein. Den Auswahlpunkt [Filesystem images][initial RAM filesystem linked into linux kernel] wählen Sie ebenfalls an; Buildroot unterstützt ein separates Initramfs nicht per se. Daher müssen Sie die Kernelkonfiguration noch selbst anpassen. Rufen Sie hierzu aus dem Buildroot-Verzeichnis das Target `make linux-menuconfig` auf. Die Auswahl [General setup][Initramfs source file(s)] ist zu löschen. Dieser Auswahlpunkt findet sich direkt hinter [Initramfs source file(s)] und dürfte mit einem Pfad belegt sein.

```
quade@felicia:~/embedded/raspi> cd buildroot-2013.05/
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # [Filesystem images][cpio the root filesystem (for use as an initial
RAM filesystem)]
    # [Filesystem images][Compression method (gzip)]
    # [Filesystem images][initial RAM filesystem linked into linux kernel]
    # [Kernel][Kernel binary format (uImage)]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make linux-menuconfig
    # Pfad löschen: [General setup][Initramfs source file(s)]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Nach der Generierung stehen im Verzeichnis `output/images/` die Dateien `rootfs.cpio.gz` und `uImage`. Wenngleich der Kernel bereits in einer U-Boot fähigen Variante vorliegt, muss das Userland noch per `arm-linux-mkimage` angepasst werden. Danach wird das System, bestehend aus `uImage` und `initramfs.uboot`, auf den tftp-Server kopiert. Achten Sie darauf, dass `uImage` dabei in `uimage.uboot` umbenannt wird.

Beispiel 4-2 *U-Boot-Skript zum Netzwerk-Boot <boot-buildroot.txt>*

```
usb start
dhcp 00200000 192.168.178.25:uimage.uboot
tftpboot 00c00000 192.168.178.25:initramfs.uboot
bootm 00200000 00c00000
```

Im nächsten Schritt muss U-Boot konfiguriert und auf die SD-Karte installiert werden. Dazu erstellen Sie ein U-Boot-Skript unter dem Namen `boot-buildroot.txt`. Die Adressen, an denen Kernel und Initramfs im Speicher abgelegt werden, dürfen sich nicht überschneiden. Den Kernel haben wir bisher bei 0x00200000 abgelegt, das Initramfs legen wir bei 0x00c00000 ab. Damit ergibt sich für ein U-Boot-Skript die in Beispiel 4-2 dargestellte Befehlsfolge. Tauschen Sie dabei wieder die verwendete IP-Adresse gegen diejenige von Ihrem tftp-Server aus. Das Skript wird mit einem U-Boot-Header versehen und steht dann zur Installation auf der SD-Karte bereit. Nachfolgend sind die Kommandos zur Generierung und Installation von Kernel, Userland und U-Boot-Skript auf der Konsole angegeben. Das U-Boot-Skript erhält dabei wieder den Namen `boot.scr`.

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> cd output/images>
quade@felicia:~/embedded/raspi/buildroot-2013.05/output/images> \
   arm-linux-mkimage -A arm \
    -O linux -T ramdisk -C none -d rootfs.cpio.gz initramfs.uboot
quade@felicia:~/embedded/raspi/buildroot-2013.05/output/images> \
   sudo cp uImage /var/lib/tftpboot/uimage.uboot
quade@felicia:~/embedded/raspi/buildroot-2013.05/output/images> \
   sudo cp initramfs.uboot /var/lib/tftpboot/
quade@felicia:~/embedded/raspi/buildroot-2013.05/output/images> \
   cd ../../../bootloader
quade@felicia:~/embedded/raspi/bootloader> gedit boot-buildroot.txt &
...
quade@felicia:~/embedded/raspi/bootloader> arm-linux-mkimage  -A arm \
   -O linux -T script -C none -d boot-buildroot.txt boot-buildroot.scr
#
# SD-Karte muss im Entwicklungsrechner unter /media/boot/ eingehängt sein
quade@felicia:~/embedded/raspi/bootloader> cp boot-buildroot.scr \
   /media/boot/boot.scr
```

Befindet sich der Bootloader `u-boot.img` noch nicht auf der SD-Karte, ist dieser ebenfalls auf der Bootpartition zu installieren. Da wir das vom Standard-Bootloader zu ladende Programm nicht `kernel.img`, sondern eben `u-boot.img` genannt haben, müssen Sie sicherstellen, dass sich in der Datei `config.txt` der Bootpartition (also bei eingehängter SD-Karte `/media/boot/config.txt`) der Eintrag »kernel=u-boot.img« befindet.

```
# Die nachfolgenden Schritte sind nur notwendig, wenn U-Boot
# nicht bereits auf der SD-Karte installiert wurde.
quade@felicia:~/embedded/raspi/bootloader> \
    cp tools/mkimage/u-boot.img /media/boot/u-boot.img
quade@felicia:~/embedded/raspi/bootloader> \
    # Datei /media/boot/config.txt editieren und kernel=u-boot.img
    # eintragen
quade@felicia:~/embedded/raspi/bootloader> sudo umount /media/boot
```

Damit ist der nur einmalig für die SD-Karte durchzuführende Schritt erledigt. Der per LAN-Kabel vernetzte Raspberry Pi sollte jetzt sowohl den Buildroot-Kernel als auch das Buildroot-Userland vom tftp-Server mit der IP, die in der Datei `boot-buildroot.txt` angegeben war, booten.

Wenn's schiefgeht ...

Sollte das System nicht wie erwartet bis zum Login booten, kann der Bootloader, der Boot-Vorgang selbst, der Kernel oder das Userland (Initramfs) Grund dafür sein.

Die Funktionstüchtigkeit des Bootloaders lässt sich am einfachsten durch Beobachten der Ausgaben zum Beispiel über die serielle Schnittstelle feststellen.

Sollte sich der Bootloader nicht melden, fehlt vielleicht (bei im Entwicklungsrechner eingehängter SD-Karte) in der Datei `/media/boot/config.txt` der Eintrag »kernel=u-boot.img«. Falls das nicht die Ursache ist, installieren Sie vom Webserver die Datei `4.2.2./u-boot.img` durch Kopieren der Datei auf die SD-Karte.

Meldet sich U-Boot, aber bootet weder Kernel noch Initramfs, ist vielleicht die Skriptdatei nicht korrekt. Auf dem Webserver finden Sie das Skript unter dem Namen `4.2.2./boot-buildroot.txt` (beziehungsweise `4.2.2./boot-buildroot.scr`). Allerdings verwendet es die für Sie mit großer Wahrscheinlichkeit ungeeignete IP-Adresse 192.168.178.25 für den tftp-Server. Tauschen Sie diese aus und versehen Sie das Skript mit einem U-Boot-Header. Danach können Sie es auf die Bootpartition der SD-Karte kopieren.

Gibt es weiterhin Probleme? Gleichen Sie die Namen der per Skript zu ladenden Dateien vom Kernel und vom Initramfs mit den Dateien auf dem tftp-Server ab. Stellen Sie sicher, dass der tftp-Server läuft.

Falls der Kernel Schwierigkeiten macht, finden Sie auf dem Webserver unter dem Namen `4.2.2./uimage.uboot` einen funktionstüchtigen Kernel. Dieser wird mit gleichem Namen auf das Download-Verzeichnis des tftp-Servers kopiert.

Startet der Kernel zwar, verwendet aber nicht das Initramfs (sondern beispielsweise die vorhandene zweite Partition auf der SD-Karte), dann liegt das meistens am Kernel, der Initramfs nicht unterstützt. Also auch in diesem Fall sollten Sie den Kernel vom Webserver austesten.

Probleme mit dem Userland identifizieren Sie durch Austausch der Datei `4.2.2./initramfs.uboot`. Auch diese wird per `cp` auf das Download-Verzeichnis des tftp-Servers kopiert.

4.3 Systemanpassung

Das bisher generierte System ist zwar funktionstüchtig, übernimmt aber bislang noch keine besondere Aufgabe, wie beispielsweise die eines Webservers. Darüber hinaus weist es diverse Schwächen auf, so startet das Netzwerk nicht vollständig. Auch müssen die generierten Dateien zurzeit noch von Hand mit einem U-Boot-Header versehen werden, wenn sie beispielsweise über den Bootloader »Das U-Boot« gebootet werden sollen.

Buildroot bietet eine Reihe von Möglichkeiten an, die Systemkonfiguration so automatisiert anzupassen, dass ein voll funktionsfähiges und direkt bootbares System herauskommt. Eine gute Dokumentation findet sich unter [bruserman]. Einige der Möglichkeiten sollen im Folgenden vorgestellt werden.

Mit Buildroot lassen sich Postbuild- und Postimage-Skripte ausführen. Die Postbuild-Skripte werden nach der Generierung des Systems, aber bevor das Image des Rootfilesystems gebildet wird, in der konfigurierten Reihenfolge aufgerufen. Das Postimage-Skript wird aufgerufen, wenn das Rootfilesystem als Image beziehungsweise CPIO-Archiv vorliegt.

Ein Postbuild-Skript bekommt als ersten Parameter den Ordner übergeben, in dem das Rootfilesystem abgelegt wurde. Das Skript selbst wird aus dem Hauptverzeichnis von Buildroot (`buildroot-2013.05`) aufgerufen. Außerdem hat es Zugriff auf eine Reihe von Umgebungsvariablen, wie beispielsweise `BASE_DIR`, das Verzeichnis, in das Buildroot die Ergebnisdateien ablegt.

Ein Postimage-Skript bekommt als ersten Parameter den Pfad zu dem Verzeichnis übergeben, in das das Rootfilesystem-Image (oder CPIO-Archiv) abgelegt wird. Es wird ebenfalls aus dem Hauptverzeichnis von Buildroot heraus aufgerufen. Das Skript wird mit den Rechten des Users gestartet, der auch den Generierungsprozess als solchen gestartet hat. Benötigt es besondere Rechte, kann `sudo` eingesetzt werden. Auch diese Skripte haben Zugriff auf diverse Umgebungsvariablen.

4.3.1 Postimage-Skript

Um direkt kopier- und installationsfähige Dateien mithilfe eines Postimage-Skripts zu erzeugen, sind die folgenden Schritte notwendig:

1. Postimage-Skript `networkboot.sh` erstellen und per `chmod` ausführbar machen
2. In der Buildroot-Konfiguration Pfad und Name des Postimage-Skripts (`../scripts/networkboot.sh`) unter [System configuration][Custom scripts to run after creating filesystem images] eintragen
3. System per `make` generieren

Ein Postimage-Skript kann verwendet werden, um sowohl den U-Boot-Header an das komprimierte CPIO-Archiv des Rootfilesystems zu hängen als auch den Kernel `uImage` und das Rootfilesystem auf den tftp-Server zu kopieren. Das Skript legen Sie am besten im neu zu erstellenden Verzeichnis `~/embedded/raspi/scripts/` mit dem Namen `networkboot.sh` ab.

Damit Buildroot das Skript findet, konfigurieren Sie den Pfad und den Namen unter [System configuration][Custom scripts to run after creating filesystem images]. Hier tragen Sie »../scripts/networkboot.sh« ein (siehe Abb. 4-6).

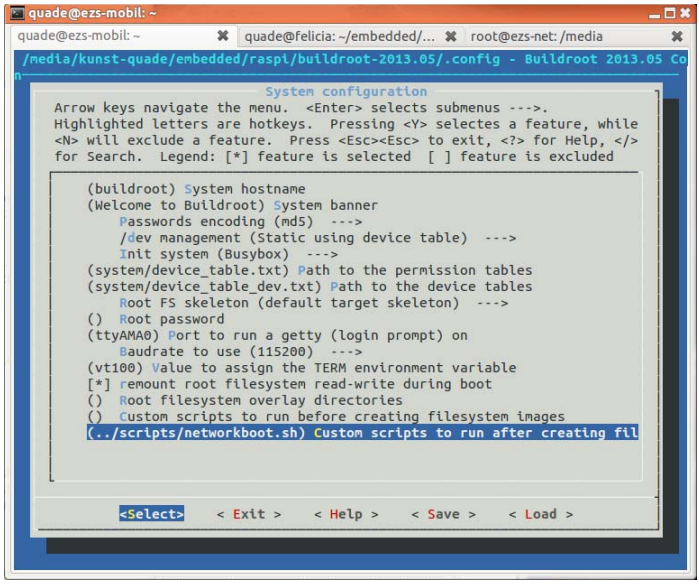


Abb. 4-6
Konfiguration eines Postimage-Skripts

Erzeugen Sie im Verzeichnis `~/embedded/raspi/` ein neues Verzeichnis `scripts`, unter dem die Postimage- und Postbuild-Skripte abgelegt werden. Erstellen Sie dort mithilfe eines Editors das Postimage-Skript `networkboot.sh` (siehe Beispiel 4-3). Vergessen Sie keinesfalls, das Skript durch `chmod +x` ausführbar zu machen:

Beispiel 4-3
Postimage-Skript für Buildroot

```
#!/bin/bash

cd $1/../images/
arm-linux-mkimage  -A arm -O linux \
        -T ramdisk -C none -d rootfs.cpio.gz initramfs.uboot

sudo cp uImage /var/lib/tftpboot/uimage.uboot
sudo cp initramfs.uboot /var/lib/tftpboot/
```

```
quade@felicia:~/embedded> mkdir raspi/scripts
quade@felicia:~/embedded> cd raspi/scripts
quade@felicia:~/embedded/raspi/scripts> gedit networkboot.sh &
...
quade@felicia:~/embedded/raspi/scripts> chmod +x networkboot.sh
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # [System configuration][Custom scripts to run after creating
filesystem images]
    # Pfad zum Script eintragen: ../scripts/networkboot.sh
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Denken Sie daran, dass für das Generieren die Pfadvariable `PATH` korrekt gesetzt ist. Außerdem müssen Sie dank `sudo` im Postimage-Skript eventuell Ihr Passwort eingeben, um kurzfristig Superuser-Rechte zu erlangen. Ist Ihr Raspberry Pi auf Netzwerk-Boot konfiguriert, können sie ihn direkt mit dem soeben generierten System booten.

4.3.2 Postbuild-Skript

Um das System für den Betrieb und das Wahrnehmen einer Aufgabe geeignet zu konfigurieren, sind die folgenden Schritte notwendig:

1. Allgemeine Vorbereitungen treffen
2. Netzwerk starten
3. Zeit setzen
4. SSH-Server starten
5. Einfachen Zugriff auf die SD-Karte ermöglichen

Während das Postimage-Skript den Entwicklungsprozess vereinfacht, kann mit dem Postbuild-Skript das eingebettete Linux-System selbst angepasst werden. So stört an dem bisherigen System, dass das Netzwerk nicht startet, die Uhrzeit nicht stimmt, kein direkter Zugriff auf die SD-Karte möglich ist und der Webserver nicht aktiv ist. Praktisch wäre für eine Fernwartung zudem ein SSH-Login.

Das Postbuild-Skript, das ein System ohne die genannten Defizite generiert, wird im Folgenden Schritt für Schritt entwickelt. Es bekommt übrigens als Parameter den Pfad zum Verzeichnis übergeben, in dem sich das generierte Target-System befindet. Es kann innerhalb des Skripts über »$1« referenziert werden. Das Gesamtergebnis ist unter Beispiel 4-11 zu finden. Die Befehle für die einzelnen Schritte werden am Ende der Abschnitte aufgeführt und sind in der Gesamtübersicht auch noch einmal unter Beispiel 4-12 nachzulesen.

Allgemeine Vorbereitungen treffen

Zu den allgemeinen Vorbereitungen gehören die folgenden Punkte:

1. In der Buildroot-Konfiguration sind Pfad und Name des Postbuild-Skripts (`../scripts/postbuild.sh`) unter [System configuration][Custom scripts to run before creating filesystem images] einzutragen.
2. Template für das Postbuild-Skript `postbuild.sh` im Verzeichnis `~/embedded/raspi/userland/target/` anlegen und per `chmod` ausführbar machen.

Beispiel 4-4
Template für das Postbuild-Skript postbuild.sh

```
#!/bin/sh

# MARK A: Netzwerk starten
#
# MARK B: Zeit setzen
#
# MARK C: SD-Karte einhaengen
#
# MARK D: Webserver
#
# MARK E: Firewall
#
# MARK F: Usermanagement
```

Bevor die einzelnen Verbesserungen bearbeitet werden können, müssen Sie in Form des Postbuild-Skripts eine Umgebung dazu in Buildroot integrieren. Legen Sie zunächst `postbuild.sh` gemäß Beispiel 4-4 im Verzeichnis `~/embedded/raspi/scripts/` als ausführbares Skript an. In Buildroot ist der Name des Skripts inklusive Pfad unter dem Auswahlpunkt [System configuration][Custom scripts to run before creating filesystem images] mit dem Inhalt »../scripts/postbuild.sh« einzutragen.

```
quade@felicia:~/embedded> cd ~/embedded/raspi/scripts/
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh &
# Template anlegen
quade@felicia:~/embedded/raspi/scripts> chmod +x postbuild.sh
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05/
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
# [System configuration][Custom scripts to run after creating
# filesystem images]
# ../scripts/postbuild.sh
```

Netzwerk starten

Um das Netzwerk auf dem System zu starten, sind die folgenden Schritte notwendig:

1. Init-Skript `S41udhcpc` im Verzeichnis `~/embedded/raspi/userland/target/` erstellen.
2. Postbuild-Skript anpassen, sodass `S41udhcpc` in das Verzeichnis `/etc/init.d/` auf dem Target installiert wird.
3. Zum Test System per `make` generieren und den mit dem LAN verbundenen Raspberry starten.

Voraussetzung für viele Funktionen, beispielsweise das Setzen der aktuellen Uhrzeit oder der Remotezugang per SSH, ist ein funktionierendes

Netzwerk, das hier am Beispiel des Ethernetports (LAN, Kabelverbindung) eingerichtet wird. Die Netzwerkadresse soll in meinem Fall per dhcp von einem dhcp-Server zugeteilt werden.

Das Hauptproblem bei der Herstellung einer Netzwerkverbindung auf dem Raspberry Pi besteht darin, dass zunächst die über USB angeschlossenen Ethernet-Devices zur Verfügung stehen müssen, bevor das benötigte Gerät `eth0` überhaupt verwendet und eine Netzwerkadresse per dhcp bezogen werden kann. Das erfordert etwas Zeit, sodass die Konfiguration der Netzwerkdienste selbst etwas verzögert wird. Auch wenn es für dieses Problem mit dem `ifplugd` eine professionelle Lösung gibt, verwenden wir hier eine Quick-and-dirty-Lösung über eine simple Zeitverzögerung.

Typischerweise werden in eingebetteten Systemen als dhcp-Client die Programme `udhcpc` oder `dhclient` eingesetzt. Wir verwenden hier den von Busybox zur Verfügung gestellten `udhcpc`, der über das Init-System (Kasten auf Seite 56) mithilfe des Init-Skripts Beispiel 4-5 gestartet werden soll. In diesem Skript ist mittels »sleep« die zeitliche Verzögerung integriert. Typischerweise ist `udhcpc` in der Standardkonfiguration von Buildroot bereits ausgewählt. Sie können das durch Aufruf von `make busybox-menuconfig` im Buildroot-Verzeichnis überprüfen und gegebenenfalls ändern.

Init-Skripte

Init ist der erste Prozess, der in einem Linux-System gestartet wird. Er hat grundsätzlich die Id »1«. Init ist für die Initialisierung und das Starten der einzelnen Dienste zuständig. Dabei setzt `init` auf die Inittab, die bereits in Kasten auf Seite 56 beschrieben ist. Für etwas komplexere Systeme wird die Konfiguration über die Inittab schnell unübersichtlich. Außerdem lassen sich (zeitliche) Abhängigkeiten zwischen vorzunehmenden Aktionen nicht richtig ausdrücken.

Daher ist es üblich, in der Inittab einen Job `rcS` zu starten, der seinerseits Skripte, die im Verzeichnis `/etc/init.d/` abgelegt sind, startet. Es hat sich die Konvention eingebürgert, dass zum Starten eines Dienstes ein Skript zur Verfügung gestellt wird, dessen Namen mit einem »S« (S für starten) beginnt, gefolgt von einer zweistelligen Nummer und dem eigentlichen Namen. Die Nummer gibt Aufschluss über die Reihenfolge, in der die Dienste gestartet werden. Das Netzwerk beispielsweise wird im Buildroot-System über das Skript `S40network` hochgefahren. Der dhcp-Client muss später aktiviert werden. Daher bekommt das zugehörige Skript im Dateinamen eine Nummer, die größer als 40 ist. Hier bietet es sich also an, ein Skript mit dem Namen `S41udhcpc` anzulegen. Erstellen Sie das Skript und legen Sie es im Verzeichnis `~/embedded/raspi/userland/target` ab. Buildroot bringt übrigens ein passendes Skript `rcS` mit, sodass wir uns nicht mehr darum kümmern müssen.

Skripte, die nach dem gleichen Schema mit einem »K« anstelle des »S« beginnen, werden beim Herunterfahren des Systems aufgerufen.

Je nach Ausprägung von `rcS` werden noch Verzeichnisse für jeden Runlevel, beispielsweise `rc2.d`, angelegt, in denen per symbolischem Link die für den Runlevel relevanten Skripte referenziert werden. Im Bereich der eingebetteten Systeme ist diese auch unter dem Namen System-V-Init bekannte Variante nur für komplexere Systeme sinnvoll. Sie wird hier nicht weiter betrachtet.

Eine weitere Konvention besteht darin, dass die Init-Skripte als ersten Parameter ein Kommando entgegennehmen, das beispielsweise »start«, »stop« oder »restart« lautet. Ein Beispielskript für `udhcpc` finden Sie in Beispiel 4-5.

Während in eingebetteten Systemen weiterhin `init` zum Einsatz kommt, wechseln Desktop- und Serversystem, wie beispielsweise auch Ubuntu, zu komplexeren Alternativen wie `upstart` oder `systemd`. Letzteres wird übrigens ebenfalls von Buildroot unterstützt.

Beispiel 4-5 *Init-Skript S41udhcpc zum Starten des udhcpc*

```
#!/bin/sh
# Based on: http://code.google.com/p/panda-buildroot/source/\
# browse/buildroot/package/busybox/S41udhcpc?name=panda-buildroot_V0.2
# /etc/init.d/udhcpc: start or stop udhcpc client

set -e

PATH=/bin:/usr/bin:/sbin:/usr/sbin
DAEMON=/sbin/udhcpc

test -x $DAEMON || exit 0

case "$1" in
  start)
    echo -n "Starting DHCP client: udhcpc"
    sleep 2 # wait for network comming up
    start-stop-daemon --start -b --quiet --exec $DAEMON \
      -- --script=/usr/share/udhcpc/default.script || \
    echo -n " already running"
    echo "."
  ;;

  restart)
    /etc/init.d/S41udhcpc stop
    /etc/init.d/S41udhcpc start
  ;;

  reload)
  ;;

  force-reload)
  ;;
```

```
  stop)
    echo -n "Stopping DHCP client: udhcpc"
    start-stop-daemon --stop --quiet --exec $DAEMON || \
    echo -n " not running"
    echo "."
  ;;

  renew)
    start-stop-daemon --signal USR1 --stop --quiet --exec $DAEMON || \
    echo -n " not running"
  ;;

  release)
    start-stop-daemon --signal USR2 --stop --quiet --exec $DAEMON || \
    echo -n " not running"
  ;;

  *)
    echo "Usage: /etc/init.d/udhcpc \
    {start|stop|restart|reload|force-reload}"
    exit 1
    ;;
esac

exit 0
```

Im Postbuild-Skript soll das vorbereitete Startskript `S41udhcpc` auf dem Target in das Verzeichnis `/etc/init.d/` installiert werden. Ergänzen Sie dazu in `postbuild.sh` hinter der Zeile mit `MARK A` den in Beispiel 4-6 gezeigten Inhalt.

Beispiel 4-6
postbuild.sh: Installation des dhcp-Startskripts

```
...
# MARK A: Netzwerk starten
echo "Netzwerk starten..."
install -m 755 ../userland/target/S41udhcpc $1/etc/init.d/

# MARK B: Zeit setzen
...
```

Gehen Sie danach in das Verzeichnis `~/embedded/raspi/buildroot-2013.05/` und rufen Sie `make` auf. Da Pfad und Name des Postbuild-Skripts von uns bereits konfiguriert wurden, generiert Buildroot jetzt das neue System inklusive der Datei `S41uhdcpc` und kopiert diese in das tftp-Verzeichnis. Mit dem nächsten Reboot — dank Netzwerk-Boot — verwendet der Raspberry Pi das neu generierte System.

Im Folgenden sind die Kommandozeilenbefehle im Terminal aufgeführt:

```
quade@felicia:~/embedded> cd raspi/userland/target/
quade@felicia:~/embedded/raspi/userland/target> gedit S41udhcpc
    # ...
quade@felicia:~/embedded/raspi/userland/target> cd ../../scripts
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh
    # install -m 755 ../userland/target/S41dhcpc $1/etc/init.d/
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05
quade@felicia:~/embedded/buildroot-2013.05> make
```

Zeit setzen

Um die Zeit auf dem System zu setzen, sind die folgenden Schritte notwendig:

1. In der Busybox-Konfiguration `ntp` auswählen
2. Init-Skript `S52ntp` im Verzeichnis `~/embedded/raspi/userland/target/` erstellen
3. Postbuild-Skript anpassen, sodass `S52ntp` in das Verzeichnis `/etc/init.d/` auf dem Target installiert wird
4. Zum Test System per `make` generieren und den mit dem LAN verbundenen Raspberry starten

Der Raspberry Pi hat, anders als ein PC, keinen Uhrenbaustein. Um eine aktuelle Uhrzeit zu erhalten, ist er daher auf eine Netzwerkverbindung angewiesen. Dazu holt er sich über das *Network Time Protocol* ntp die Zeit von einem Zeitserver. Per ntp wird die lokale Zeit dann peu à peu an die ausgelesene Zeit angepasst. Mehr Informationen zu ntp und dem Anpassen der Zeit (anstelle des Setzens einer Uhrzeit) finden Sie in [QuMä2012].

Sowohl Busybox als auch Buildroot bieten einen NTP-Server beziehungsweise Client an. Der in Busybox eingebaute ntp-Dienst ist einfacher und benötigt weniger Ressourcen. Für unseren Einsatz reicht diese Variante aus. Dabei gilt es nur zu beachten, dass der Busybox-Dienst aufgrund einer Eigenheit nicht zu früh gestartet wird, sonst beendet er sich gleich wieder. Daher findet sich auch hier im Skript per `sleep` eine Zeitverzögerung.

Beispiel 4-7 *Init-Skript zum Starten von ntp*

```
#! /bin/sh
#

PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
DESC="network time protocol daemon"
NAME=ntpd
DAEMON=/usr/sbin/$NAME
OPTION="-d -p 0.de.pool.ntp.org"
```

```
# Gracefully exit if the package has been removed.
test -x $DAEMON || exit 0

echo $1 >>/tmp/output
case "$1" in
  start)
   echo "Starting $DESC: $NAME"
   sleep 1
   start-stop-daemon -S -x $DAEMON -- $OPTION
   echo "."
   ;;
  stop) echo -n "Stopping $DESC: $NAME"
   start-stop-daemon -K -q -n $NAME
   echo "."
   ;;
  reload|force-reload) echo -n "Reloading $DESC configuration..."
   start-stop-daemon -K -q -n $NAME -s 1
   echo "done."
  ;;
  restart) echo "Restarting $DESC: $NAME"
   $0 stop
   sleep 1
   $0 start
   ;;
  *) echo "Usage: $SCRIPTNAME {start|stop|restart|reload|force-reload}" >&2
   exit 1
   ;;
esac

exit 0
```

Folglich ist als Erstes in der Busybox-Konfiguration `ntp` auszuwählen. Als Zweites ist ein Init-Skript (`S52ntp`) zum Starten von `ntp` im Verzeichnis `~/embedded/raspi/userland/target/` zu erstellen (siehe Beispiel 4-7). Im dritten Schritt schließlich wird das Postbuild-Skript `postbuild.sh` an der Stelle `MARK B` angepasst, sodass das Init-Skript `S52ntp` in das Target-System kopiert wird:

```
...
# MARK B: Zeit setzen
install -m 755 ../userland/target/S52ntp $1/etc/init.d/

# MARK C: ...
```

Die Befehle, die sich damit auf der Kommandozeile ergeben, sind die Folgenden:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> make busybox-menuconfig
# [Networking Utilities][ntpd]
```

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> gedit \
   ../userland/target/S52ntp &
...
quade@felicia:~/embedded/raspi/buildroot-2013.05> gedit \
   ../scripts/postbuild.sh &
...
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Nach dem Generieren des Systems sollte sich (eine Netzwerkverbindung vorausgesetzt) der Raspberry Pi mit dem nächsten Reboot die aktuelle Uhrzeit holen.

SSH-Server

Um das Embedded Linux mit einem SSH-Server zu versehen, sind die folgenden Schritte notwendig:

1. In der Buildroot-Konfiguration SSH-Server `dropbear` auswählen
2. In der Buildroot-Konfiguration für den User »root« ein Passwort setzen
3. Zum Test System per `make` generieren, den mit dem LAN verbundenen Raspberry starten, die IP-Adresse identifizieren und per `ssh` einloggen

Sehr praktisch und eines der ersten Dinge, die jeder Administrator auf einem Linux-System einrichtet, ist ein SSH-Server. Per Secure Shell (SSH) kann man sich von einem anderen Rechner aus auf das System einloggen und es über diesen Remote-Zugriff verwalten. SSH bietet sowohl eine strenge Authentifizierung als auch einen verschlüsselten Datentransport und ist damit eine sichere Alternative zum alten `telnet`, bei dem Passwörter im Klartext über das Netz geschickt werden.

Buildroot bietet mit `openssh` und `dropbear` gleich zwei SSH-Server zur Auswahl. `Openssh` ist dabei der große Bruder, während für diejenigen, die auf SSH1 (aktuell ist SSH2) und SFTP (Secure File Transfer Protocol) verzichten können, der aktiv gepflegte `dropbear` ausreichend ist. Wir werden im Folgenden ebenfalls die schlanke Alternative `dropbear` einsetzen.

Charmant an dieser Lösung ist, dass wir in diesem Fall nur in der Buildroot-Konfiguration einmal Dropbear auswählen [Package Selection for the target][Networking applications][dropbear] und zum anderen für den User Root unter [System configuration][Root password] ein Passwort setzen. Das Passwort wird benötigt, weil SSH es zum Remote-Login verlangt. Da das Passwort aber real das System nicht absichert, sollten Sie später noch wie in Kapitel 7 beschrieben verfahren.

Bereits die Hilfestellung zum Auswahlpunkt »Root password« warnt zu Recht vor zwei Dingen: Erstens wird das Root-Passwort mit dem Hashverfahren MD5 gehasht, das nicht mehr als sicher gilt. Zweitens wird das Passwort im Klartext sowohl in der Konfigurationsdatei als auch in Generierungslogs abgelegt. Damit ist das Passwort so gut wie öffentlich (und kein wirkliches Passwort mehr). Verwenden Sie also auf keinen Fall ein Passwort, das Sie noch an anderer Stelle einsetzen.

Nach der Konfiguration können Sie im Buildroot-Verzeichnis wieder `make` aufrufen und den Raspberry Pi rebooten.

Nach dem Reboot steht der SSH-Server zur Verfügung und Sie können sich beispielsweise vom Entwicklungsrechner aus auf das eingebettete System einloggen. Dazu benötigen Sie allerdings die IP-Adresse. Fahren Sie Ihr Embedded System mit einer festen IP-Adresse, können Sie diese direkt verwenden. Setzen Sie jedoch — wovon wir die ganze Zeit ausgehen — einen dhcp-Server ein, der dynamisch die IP-Adressen verteilt, gibt es mehrere Möglichkeiten, die Adresse zu erfahren. Sie können erstens — falls Sie darauf Zugriff haben — in der Logdatei des dhcp-Servers nachsehen. Sie können sich zweitens auf dem System über den normalen Tastaturlogin einloggen und das Kommando `ifconfig` eingeben. Unter dem Gerät `eth0` (bei einer kabelgebundenen Anbindung) finden Sie dann die Adresse. Als Letztes schließlich können Sie per `nmap` das System nach Geräten scannen, die einen SSH-Server aktiviert haben.

So loggen Sie sich von einem Terminal auf dem eingebetteten System ein, wenn die IP-Adresse 192.168.178.31 lautet:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> ssh root@192.168.178.31
The authenticity of host '192.168.178.31 (192.168.178.31)' can't be established.
RSA key fingerprint is ec:77:54:4e:78:92:0f:a0:17:e2:8d:88:9c:40:85:0a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.178.31' (RSA) to the list of known hosts.
root@192.168.178.31's password:
# cd /
# ls
bin    data    etc     init    linuxrc  mnt     proc    run sys     usr
boot   dev     home    lib     media    opt     root    sbin tmp    var
#
```

Übrigens: Mithilfe des Programms `scp` lassen sich sehr einfach und verschlüsselt Daten zwischen dem eingebetteten System und jedem anderen Rechner austauschen.

Hier noch einmal die Kommandozeilenbefehle:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
# [Package Selection for the target][Networking applications][dropbear]
# [System configuration][Root password]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Zugriff auf die SD-Karte

Um den einfachen Zugriff auf die SD-Karte zu ermöglichen, sind die folgenden Schritte notwendig:

1. Im Verzeichnis `embedded/raspi/userland/target/` eine Datei `mmc.txt` erstellen, die die Informationen zum Anlegen der Gerätedateien enthält.
2. In der Buildroot-Konfiguration sicherstellen, dass »/dev management« auf statische Konfigurationsdateien eingestellt ist.
3. In der Buildroot-Konfiguration den Pfad auf die Gerätedateitabellen (»Path to the device tables«) um die neue Konfigurationsdatei `../userland/target/mmc.txt` ergänzen.
4. Falls die Partitionen der SD-Karte direkt eingehängt werden sollen, ist zusätzlich noch `postbuild.sh` so anzupassen, dass zwei Verzeichnisse als Einhängepunkte und die Datei `/etc/fstab` erzeugt werden.
5. Zum Test System per `make` generieren, den mit dem LAN verbundenen Raspberry starten. Falls die Partitionen direkt eingehängt wurden, sollten diese unter `/boot/` und `/data/` erreichbar sein.

Hat man einmal einen Remote-Zugang zu dem System, gibt es kaum noch Gründe, die SD-Karte aus dem Raspberry Pi zu entfernen. Wir können nun Daten — beispielsweise neue Boot-Dateien vom U-Boot — per `scp` vom Host auf das eingebettete System verschieben. Um die Daten dann auf die SD-Karte zu schreiben, müssen allerdings noch die notwendigen Gerätedateien `/dev/mmcblk0`, `/dev/mmcblk0p1` und `/dev/mmcblk0p2` erstellt werden.

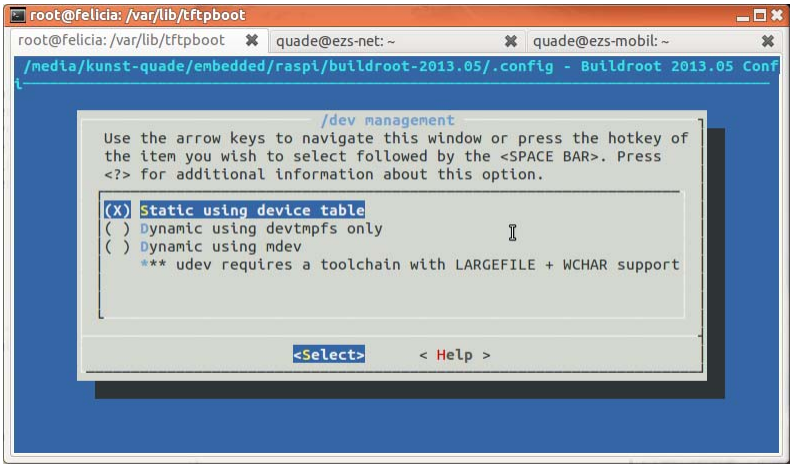


Abb. 4-7 Buildroot-Gerätemanagement

Zum Anlegen von Gerätedateien bietet Buildroot unter dem Punkt Gerätemanagement wie in Abbildung 4-7 zu sehen ist, vier verschiedene

Auswahlmöglichkeiten an. Während die letzte Variante `udev` die auf einem Desktop- oder Serversystem favorisierte, aber letztlich auch kompliziert zu konfigurierende Variante ist, stellt die erste Variante (statische Einrichtung über Konfigurationsdateien) den einfachsten Weg dar. Dieser ist zudem unabhängig von der Kernelversion. Die übrigen drei Varianten verlangen nämlich alle `devtmpfs`, was erst ab Kernel 2.6.32 unterstützt wird. Auch wenn das automatische Anlegen der Gerätedateien im Betrieb sehr attraktiv ist, bleiben wir für unser eingebettetes System bei der einfachen und übersichtlichen Variante mittels statischer Konfigurationsdateien.

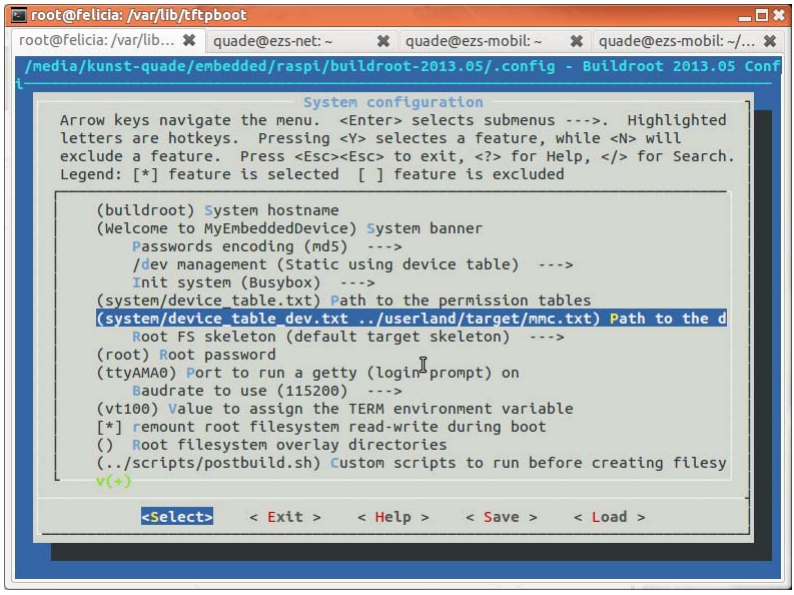


Abb. 4-8
Buildroot-Gerätemanagement (Fortsetzung)

```
# See package/makedevs/README for details
#
# This device table is used only to create device files when a static
# device configuration is used (entries in /dev are static).
#
# <name>   <type>   <mode>   <uid>   <gid>   <major>   <minor>
# <start>   <inc>   <count>

# MMC devices
/dev/mmcblk0   b   660   0   0   179   0   0   0   -
/dev/mmcblk0p1   b   660   0   0   179   1   0   0   -
/dev/mmcblk0p2   b   660   0   0   179   2   0   0   -
```

Beispiel 4-8
Konfiguration fehlender Gerätedateien für den Zugriff auf die SD-Karte <mmc.txt>

Dazu legen Sie als Erstes nach dem Vorbild Beispiel 4-8 im Verzeichnis ~/embedded/raspi/userland/target/ eine Datei mmc.txt mit den zusätzlich zu erstellenden Gerätedateien an. Anschließend konfigurieren Sie Buildroot und ergänzen wie in Abbildung 4-8 zu sehen unter [System configuration][Path to the device tables] den Pfad und Dateinamen der soeben erstellten Konfigurationsdatei »../userland/target/mmc.txt«. Mit dem nächsten System-Build werden dann die Gerätedateien angelegt und der Zugriff ist möglich.

Wollen Sie die Partitionen der SD-Karte beim Hochfahren direkt mit einhängen? In dem Fall ist noch postbuild.sh anzupassen.

Per Konvention sind die einzuhängenden Filesysteme in der Datei /etc/fstab/ abgelegt. Wir müssen diese Datei auf dem Target also um zwei Einträge erweitern. Dazu benötigen wir noch zwei Einhängepunkte (Mountpoints), also Verzeichnisse, unter denen die Daten später erscheinen sollen. Als Einhängepunkte verwenden wir beispielsweise /boot/ für die erste Partition der SD-Karte, die sämtliche Boot-Dateien enthält, und /data/ für die zweite Partition, die klassischerweise das System bevorratet. In unserem Fall — wir haben ja Initramfs — könnte die Partition auch leer sein. Wir benötigen auf der SD-Karte nur die Bootpartition.

In postbuild.sh legen wir damit zwei neue Verzeichnisse an und ergänzen die Datei /etc/fstab. So werden beim nächsten Rebooot des Raspberry Pi beide Partitionen ein- und beim ordentlichen Herunterfahren auch wieder ausgehängt. Hier die Ergänzung für postbuild.sh:

```
# MARK C: SD-Karte einhaengen
mkdir $1/boot
mkdir $1/data
echo "/dev/mmcblk0p1    /boot   vfat   defaults   0   0" >>$1/etc/fstab
echo "/dev/mmcblk0p2    /data   ext4   defaults   0   0" >>$1/etc/fstab
```

Und hier noch die Kommandozeilenbefehle:

```
quade@felicia:~/embedded> cd ~/embedded/raspi/userland/target/
quade@felicia:~/embedded/raspi/userland/target> gedit mmc.txt &
...
quade@felicia:~/embedded/raspi/userland/target> \
   cd ../../buildroot-2013.05/
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
# [System configuration][/dev management (Static using device table)]
# [System configuration][(system/device_table_dev.txt
# ../userland/target/mmc.txt) Path to the device tables]
quade@felicia:~/embedded/raspi/buildroot-2013.05> \
   cd ~/embedded/raspi/scripts/
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh &
...
```

```
quade@felicia:~/embedded/raspi/userland/target> \
   cd ../../buildroot-2013.05/
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Webserver (httpd)

Um einen Webserver zu installieren, sind die folgenden Schritte notwendig:

1. Webserverinhalte in unsere Umgebung unter `~/embedded/raspi/userland/target/` kopieren
2. In Busybox den Webserver auswählen
3. Die Webserverkonfiguration in unsere Umgebung unter `~/embedded/raspi/userland/target/` kopieren
4. Init-Skript `S51httpd` zum Starten des Webservers erstellen
5. Das Generierungsskript `postbuild.sh` um die notwendigen Aktionen ergänzen
6. Zum Test System per `make` generieren, den mit dem LAN verbundenen Raspberry starten und per Webbrowser auf den Webserver zugreifen

Eine der häufigsten Aufgaben (und manchmal auch die einzige) besteht für ein eingebettetes System darin, einen Webserver zur Verfügung zu stellen. Mit Buildroot existieren hierfür viele Alternativen, beispielsweise `boa`, `lighttpd`, `mongoose`, `thttpd` oder `tinyhttpd`. Darüber hinaus kann weiterhin der http-Server von Busybox genutzt werden. Da wir für diesen bereits sämtliche Konfigurationen und Dateien vorbereitet haben und weil er eine der schlankesten Möglichkeiten für einen Webserver darstellt, installieren wir ihn auf dem Target.

Für den Webserver brauchen wir:

1. Webserverinhalte

   Die Inhalte — in unserem Fall eine statische und eine dynamische Webseite — haben wir bereits unter `~/embedded/qemu/userland/target/` erstellt. Die zugehörigen Dateien `index.html` und `ps.cgi` liegen im Verzeichnis `~/embedded/raspi/userland/target/`. Das noch anzupassende Skript `postbuild.sh` wird sie später in das noch anzulegende Target-Verzeichnis `/var/www/` respektive `/var/www/cgi-bin/` verschieben.

2. Webserverauswahl in Busybox

   In der Busybox- (nicht in der Buildroot-)Konfiguration muss der Webserver ausgewählt werden. Rufen Sie dazu im Buildroot-Verzeichnis `make busybox-menuconfig` auf. Sie finden den Auswahlpunkt unter [Networking Utilities][httpd].

3. Webserverkonfiguration

   Hierfür können wir wieder auf die bereits im Verzeichnis `~/embedded/qemu/userland/target/` erstellte Konfiguration `httpd.conf` zurückgreifen. Diese befindet sich im Verzeichnis `~/embedded/raspi/userland/target/` und wird später per `postbuild.sh` in das Target-Verzeichnis `/etc/init.d/` kopiert.

4. Startskript für das Target

   Als Nächstes benötigen wir noch ein Init-Skript zum Starten des Webservers während des Boot-Vorgangs. Bezüglich der Startreihenfolge sollte erst das Netzwerk aktiv sein. Daher bekommt das Skript eine Nummer oberhalb von »S41«, beispielsweise »S51«. Damit wird das Skript auch nach dem SSH-Server gestartet. Der Aufbau des Skripts ist in Beispiel 4-10 dargestellt. Im Wesentlichen entspricht es dem Skript für ntp, nur dass im Kopf die beiden Variablen `$NAME` und `$OPTION` angepasst sind.

5. Ergänzungen zum Generierungsskript

   Das Anlegen der benötigten Verzeichnisse, das Kopieren der Konfigurations- und Startskriptdateien in das Rootfilesystem des Targets und das Setzen der zugehörigen Zugriffsrechte (`install`) während der Systemgenerierung übernimmt das Skript `postbuild.sh`. Die dort anzuhängenden Befehle finden Sie unter Beispiel 4-9.

Beispiel 4-9 *Webserverkonfiguration für das Postbuild-Skript*

```
# Configuring webserver
mkdir -p $1/var/www/cgi-bin/
install -m 755 ../userland/target/S51httpd $1/etc/init.d/
install ../userland/target/httpd.conf $1/etc/
install -D ../userland/target/index.html $1/var/www/
install -D -m 755 ../userland/target/ps.cgi $1/var/www/cgi-bin/
```

Beispiel 4-10 *Init-Skript S51httpd zum Starten des Webservers*

```
#! /bin/sh
#

PATH=/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
DESC="httpd-server"
NAME=httpd
DAEMON=/usr/sbin/$NAME
OPTION="-c /etc/httpd.conf -h /var/www"

# Gracefully exit if the package has been removed.
test -x $DAEMON || exit 0

echo $1 >>/tmp/output
case "$1" in
  start)
```

```
   echo "Starting $DESC: $NAME"
   start-stop-daemon -S -x $DAEMON -- $OPTION
   echo "."
   ;;
  stop) echo -n "Stopping $DESC: $NAME"
   start-stop-daemon -K -q -n $NAME
   echo "."
   ;;
  reload|force-reload) echo -n "Reloading $DESC configuration..."
   start-stop-daemon -K -q -n $NAME -s 1
   echo "done."
  ;;
  restart) echo "Restarting $DESC: $NAME"
   $0 stop
   sleep 1
   $0 start
   ;;
  *) echo "Usage: $SCRIPTNAME {start|stop|restart|reload|force-reload}" >&2
   exit 1
   ;;
esac

exit 0
```

Die Kommandozeilenbefehle für die Installation eines Webservers noch einmal im Überblick:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> cd ..
quade@felicia:~/embedded/raspi> cp ~/embedded/qemu/userland/target/index.html \
   ~/embedded/raspi/userland/target
quade@felicia:~/embedded/raspi> cp ~/embedded/qemu/userland/target/ps.cgi \
   ~/embedded/raspi/userland/target
quade@felicia:~/embedded/raspi> cp ~/embedded/qemu/userland/target/httpd.conf \
   ~/embedded/raspi/userland/target
quade@felicia:~/embedded/raspi> cd buildroot-2013.05/
quade@felicia:~/embedded/raspi/buildroot-2013.05> make busybox-menuconfig
# [Networking Utilities][httpd]
quade@felicia:~/embedded/raspi/buildroot-2013.05> cd ../userland/target/
quade@felicia:~/embedded/raspi/userland/target> gedit S51httpd &
...
quade@felicia:~/embedded/raspi/userland/target> cd ../../scripts
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh &
# MARK D: Webserver
...
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Zusammengefasst

Das gebaute System ist weit davon entfernt, perfekt zu sein, und kann beileibe noch viele Optimierungen erfahren. Nehmen Sie dies als Gelegenheit, sich selbst an Optimierungen zu versuchen. Eine ausreichende Grundlage ist dafür jetzt gelegt. Sowohl in der Buildroot-Dokumentation [bruserman] als auch im Internet gibt es viele Quellen für spezifische Informationen.

Beispiel 4-11
Das Postbuild-Skript mit Feintuning-Aktionen

```
#!/bin/bash

# MARK A: Netzwerk starten
echo "Netzwerk starten..."
install -m 755 ../userland/target/S41udhcpc $1/etc/init.d/

# MARK B: Zeit setzen
echo "Zeit setzen..."
install -m 755 ../userland/target/S52ntp $1/etc/init.d/

# MARK C: SD-Karte einhaengen
echo "SD-Karte einhaengen..."
mkdir $1/boot
mkdir $1/data
echo "/dev/mmcblk0p1   /boot   vfat   defaults    0    0">>$1/etc/fstab
echo "/dev/mmcblk0p2   /data   ext4   defaults    0    0">>$1/etc/fstab

# MARK D: Webserver
echo "Webserver..."
mkdir -p $1/var/www/cgi-bin
install -m 755 ../userland/target/S51httpd $1/etc/init.d/
install ../userland/target/httpd.conf $1/etc/
install -D ../userland/target/index.html $1/var/www/
install -D -m 755 ../userland/target/ps.cgi $1/var/www/cgi-bin/
```

Beispiel 4-12
Kommandozeilenbefehle zur Optimierung des Buildroot-Systems

```
############################
# Allgemeine Vorbereitungen #
############################
cd ~/embedded/raspi/scripts/
gedit postbuild.sh
# Template anlegen
#
chmod +x postbuild.sh
cd ~/embedded/raspi/buildroot-2013.05/
make menuconfig
# [System configuration]
#     [Custom scripts to run after creating filesystem images]
# ../scripts/postbuild.sh
```

```
####################
# Netzwerk starten #
####################
cd ~/embedded/raspi/userland/target/
gedit S41udhcp
# Init-Skript zum Starten von udhcpc
cd ~/embedded/raspi/scripts/
gedit postbuild.sh
# MARK A: Befehle zum Netzwerkstart einfügen
#
cd ~/embedded/raspi/buildroot-2013.05/
make
# Wird der am LAN hängende Raspberry Pi neu gebootet,
# sollte er per DHCP eine IP-Adresse bekommen.

###############
# Zeit setzen #
###############
cd ~/embedded/raspi/buildroot-2013.05/
make busybox-menuconfig
# [Networking Utilities][ntpd]

# Init-Skript zum Setzen der Zeit erstellen
cd ~/embedded/raspi/userland/target/
gedit S52ntp
cd ~/embedded/raspi/scripts/
gedit postbuild.sh
# MARK B: Befehle zum Setzen der Zeit einfügen
#
cd ~/embedded/raspi/buildroot-2013.05/
make
# Wird der am LAN hängende Raspberry Pi neu gebootet,
# sollte er zusätzlich die aktuelle Uhrzeit haben.

##############
# SSH-Server #
##############
cd ~/embedded/raspi/buildroot-2013.05/
make menuconfig
# [Package Selection for the target][Networking applications]
#     [dropbear]
# [System configuration][Root password]

make

############################
# Zugriff auf die SD-Karte #
############################
cd ~/embedded/raspi/userland/target/
gedit mmc.txt
cd ~/embedded/raspi/buildroot-2013.05/
```

```
make menuconfig
# [System configuration][/dev management (Static using device table)]
# [System configuration][(system/device_table_dev.txt
#     ../userland/target/mmc.txt) Path to the device tables]

cd ~/embedded/raspi/scripts/
gedit postbuild.sh
# MARK C: SD-Karte

cd ~/embedded/raspi/buildroot-2013.05/
make

#############
# Webserver #
#############
cp ~/embedded/qemu/userland/target/index.html \
        ~/embedded/raspi/userland/target
cp ~/embedded/qemu/userland/target/ps.cgi \
        ~/embedded/raspi/userland/target
cd ~/embedded/raspi/buildroot-2013.05/
make busybox-menuconfig
# [Networking Utilities][httpd]
cp ~/embedded/qemu/userland/target/httpd.conf \
        ~/embedded/raspi/userland/httpd.conf
cd ~/embedded/raspi/userland/target/
gedit S51httpd
cd ~/embedded/raspi/scripts/
gedit postbuild.sh
# MARK D: Webserver

cd ~/embedded/raspi/buildroot-2013.05/
make
```

In Beispiel 4-11 ist das komplette Postbuild-Skript abgedruckt, eine Zusammenfassung der Kommandozeilenbefehle finden Sie in Beispiel 4-12.

Wenn's schiefgeht ...

Führen Sie die Optimierung des Systems Schritt für Schritt durch, also erst Netzwerk aktivieren, danach Zeit setzen, dann den SSH-Server einrichten, den Zugriff auf die SD-Karte und schließlich die Installation des Webservers. Überprüfen Sie nach jedem Schritt wie angegeben den Erfolg.

Sollten einzelne Schritte nicht erfolgreich sein, überprüfen Sie auf dem gestarteten Target-System, ob dort die zugehörigen Dateien im richtigen Verzeichnis (zum Beispiel S51ntp in `/etc/init.d/`) abgelegt sind. Überprüfen Sie durch Aufruf von `ps` auf dem eingebetteten System, ob die zugehörigen Dienste laufen.

Außerdem finden Sie auf der Webseite zum Buch unter dem Verzeichnis `4.3.2.` sowohl die einzelnen Init-Skripte, Generierungsskripte, Webserverdateien und Konfigurationsdateien, als auch die Imagedateien von Kernel und Initramfs (siehe Tabelle 4-2).

Tabelle 4-2 *Dateien für das optimierte Buildroot-System*

Dateiname	Verzeichnis auf dem Entwicklungssystem	Verzeichnis auf dem Zielsystem	Bedeutung
mmc.txt	userland/target/		Spezifikation Gerätedateien
networkboot.sh	scripts/		Postimage-Skript
postbuild.sh	scripts/		Postbuild-Skript
S41udhcpc	userland/target/	/etc/init.d/	Startskript DHCP
S51httpd	userland/target/	/etc/init.d/	Startskript HTTPD
S52ntp	userland/target/	/etc/init.d/	Startskript NTP
httpd.conf	userland/target/	/etc/	Webserver-konfiguration
index.html	userland/target/	/var/www/	Webserver-Startseite
ps.cgi	userland/target/	/var/www/cgi-bin/	CGI-Skript Webserver
uimage.uboot	/var/lib/tftpboot/		Linux-Kernel
initramfs.uboot	/var/lib/tftpboot/		Initramfs (Userland)

4.4 Eigene Buildroot-Pakete

Typischerweise werden in einem eingebetteten System eine Reihe von Eigenentwicklungen, beispielsweise Applikationen oder Treiber, eingesetzt. Diese sollten natürlich ebenfalls von dem Systembuilder mit eingebaut werden. Dazu sind aus den einzelnen Komponenten Pakete zu bilden, die dann vom Systembuilder, hier Buildroot, integriert werden.

4.4.1 Grundstruktur

Zur Integration benötigt Buildroot eine umfangreiche Paketbeschreibung. Sämtliche Paketbeschreibungen befinden sich unterhalb des Buildroot-Hauptverzeichnisses im Unterverzeichnis `package` in einem Ordner, das den Namen des Paketes trägt (siehe Abb. 4-9). Pakete können übri-

gens durchaus Unterpakete anbieten, die ihrerseits dann in einem Verzeichnis darunter abgelegt werden.

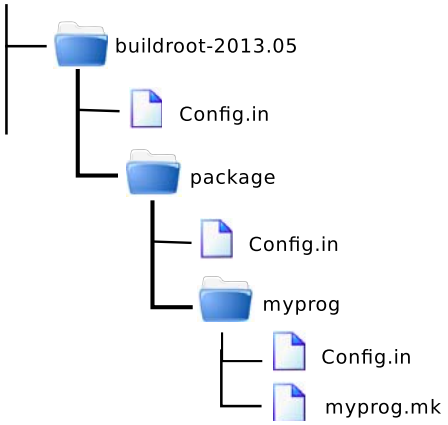


Abb. 4-9 *Dateiorganisation der Buildroot-Beschreibung für das Paket »myprog«*

Die Paketbeschreibung besteht aus mindestens zwei Dateien: erstens der Auswahlbeschreibung `Config.in` und zweitens einer Generierungsbeschreibung (Dateierweiterung `.mk`).

Die Auswahlbeschreibung (`Config.in`) lässt den Nutzer bei der Buildroot-Konfiguration (`make menuconfig`) das Paket aus- oder abwählen. Beispiel 4-13 zeigt den Aufbau der Datei.

Beispiel 4-13 *Config.in-Datei in Buildroot*

```
config BR2_PACKAGE_MYPROG
    bool "Myprog"
    select BR2_PACKAGE_ASWELL
    depends on BR2_PACKAGE_NEEDED
    help
      Application with a wonderful function.

      http://www.my-prog.org/
```

Wesentliches Element ist das Label, hier `BR2_PACKAGE_MYPROG`. Das Schlüsselwort »bool« gibt an, dass der User das Paket an- oder abwählen kann. Das Schlüsselwort »select« ist optional und bedeutet, dass das dort angegebene Paket (hier `BR2_PACKAGE_ASWELL`) bei Auswahl des Paketes ebenfalls mit ausgewählt wird. »depends« spiegelt die Abhängigkeit wider. Nur wenn das dort angegebene Paket mit dem Namen `BR2_PACKAGE_NEEDED` bereits ausgewählt ist, wird unser Paket überhaupt zur Auswahl angeboten und kann danach auch ausgewählt werden. Alle Schlüsselwörter sind übrigens per Tabulator (TAB) eingerückt; der dem Schlüsselwort »help« folgende Hilfetext zusätzlich noch um zwei Leerzeichen (Space).

Damit die `Config.in`-Datei später beim Aufruf von `make menuconfig` erscheint, muss sie noch in das Build-System aufgenommen werden. Das Build-System startet mit der Datei `Config.in`, die sich im Hauptverzeichnis (also buildroot-2013.05) von Buildroot befindet. Hier werden die übrigen Dateien per Kommando `source <pfad>/Config.in` eingebunden, so zum Beispiel auch die Datei `Config.in`, die sich im Unterverzeichnis `packages` befindet. Die Datei `buildroot-2013.05/package/Config.in` ist daher zu editieren, um das Paket in das Build-System mit aufzunehmen.

Die zweite Datei enthält die Bauvorschrift für das ausgewählte Paket. Sie wird von Buildroot bearbeitet, wenn das System generiert wird. Der Name der Datei ist der Paketname plus der Dateierweiterung ».mk«. Heißt das Paket beispielsweise »myprog«, lautet der Name der Datei »myprog.mk«. Beispiel 4-14 zeigt eine Beispiel-Bauvorschrift, die der Buildroot-Dokumentation entnommen ist.

Beispiel 4-14
Bauvorschrift für ein Buildroot-Paket [bruserman]

```
01: ################################################################################
02: #
03: # libfoo
04: #
05: ################################################################################
06:
07: LIBFOO_VERSION = 1.0
08: LIBFOO_SOURCE = libfoo-$(LIBFOO_VERSION).tar.gz
09: LIBFOO_SITE = http://www.foosoftware.org/download
10: LIBFOO_LICENSE = GPLv3+
11: LIBFOO_LICENSE_FILES = COPYING
12: LIBFOO_INSTALL_STAGING = YES
13: LIBFOO_CONFIG_SCRIPTS = libfoo-config
14: LIBFOO_DEPENDENCIES = host-libaaa libbbb
15:
16: define LIBFOO_BUILD_CMDS
17:     $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
18: endef
19:
20: define LIBFOO_INSTALL_STAGING_CMDS
21:     $(INSTALL) -D -m 0755 $(@D)/libfoo.a $(STAGING_DIR)/usr/lib/libfoo.a
22:     $(INSTALL) -D -m 0644 $(@D)/foo.h $(STAGING_DIR)/usr/include/foo.h
23:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(STAGING_DIR)/usr/lib
24: endef
25:
26: define LIBFOO_INSTALL_TARGET_CMDS
27:     $(INSTALL) -D -m 0755 $(@D)/libfoo.so* $(TARGET_DIR)/usr/lib
28:     $(INSTALL) -d -m 0755 $(TARGET_DIR)/etc/foo.d
29: endef
30:
31: define LIBFOO_DEVICES
32:     /dev/foo  c  666  0  0  42  0  -  -  -
33: endef
```

```
34:
35: define LIBFOO_PERMISSIONS
36:     /bin/foo  f  4755  0  0  -  -  -  -  -
37: endef
38:
39: define LIBFOO_USERS
40:     foo -1 libfoo -1 * - - - LibFoo daemon
41: endef
42:
43: $(eval $(generic-package))
```

Jede dieser Dateien ist identisch aufgebaut. Im oberen Teil wird über eine Reihe von Make-Variablen der Generierungsvorgang konfiguriert. Die Namen der Make-Variablen beginnen alle mit dem Paketnamen in Großbuchstaben, wobei eventuell vorkommende Punkte oder Bindestriche in Unterstriche umgewandelt werden. Die ersten Variablen enthalten Metainformationen bezüglich des Paketnamens, der Version oder des Download-Servers.

Im mittleren Teil enthält diese Datei Kommandos für die verschiedenen Stufen des Build-Vorgangs und im unteren Teil — also am Ende der Datei — wird ein Makro aufgerufen (»$(eval $(generic-package))«), das den eigentlichen Build-Prozess durchführt. Hierfür stehen drei Makros zur Auswahl:

1. autotools-package: Das Paket wird mit Werkzeugen wie `autoconf` oder `automake` generiert.
2. cmake-package: Das Paket wird mit `cmake` generiert.
3. generic-package: Dieses Makro wird für alle übrigen Pakete verwendet, beispielsweise für diejenigen, die auf klassischen Makefiles oder einfachen Skripten beruhen. Für den Fall, dass Software für den Host generiert werden soll (beispielsweise der Cross-Compiler), gibt es die Variante `host-generic-package`. Bei der Variante `generic-package` wird das Paket für die Target-Plattform cross-kompiliert. Es ist erlaubt, beide Varianten gleichzeitig einzusetzen.

Grundsätzlich besteht der Build-Vorgang aus den Schritten: downloaden, auspacken, patchen, konfigurieren, generieren, installieren, aufräumen und deinstallieren. Bei jedem Schritt werden Kommandos ausgeführt, die über die im ersten Teil der Datei angegebenen Make-Variablen konfiguriert sind.

Im Rahmen der Kommandos können die folgenden Variablen eingesetzt werden:

$(@D)

Diese Variable speichert das Verzeichnis, in dem die ausgepackten Quelldateien des Paketes liegen.

TARGET_CC, TARGET_LD, ...

Über diese Variablen werden die Cross-Werkzeuge referenziert.

TARGET_CROSS

Diese Variable enthält das Präfix der Cross-Toolchain.

HOST_DIR, TARGET_DIR, STAGING_DIR

Diese Variablen beinhalten Pfad und Name der jeweiligen Verzeichnisse.

Download

Zunächst benötigt Buildroot die Quelldateien des Paketes. Hierfür gibt es verschiedene Methoden, unter anderem git, svn, scp, wget, file und local. Die von Buildroot zu verwendende Methode ist entweder in der Make-Variablen »X_SITE« oder in der optionalen Variablen »X_SITE_METHOD« hinterlegt, wobei das »X« für den Paketnamen steht. Konkret: Bei »MYPROG_SITE=http://www.my-prog.org/myprog« würde Buildroot versuchen, das in der Variablen »MYPROG_SOURCE« angegebene Paket per http herunterzuladen und in das Unterverzeichnis »dl« (ausgehend vom Buildroot-Hauptverzeichnis) abzulegen. In der Variante »MYPROG_SITE=/opt/myprog/myprog.tar.gz« kopiert Buildroot das Tar-File vom Verzeichnis »/opt/myprog/« in das Download-Verzeichnis »dl«. Wird nur ein Verzeichnis angegeben, wählt Buildroot die Methode »local« und kopiert die Dateien direkt aus dem Direktory heraus.

Auspacken

Buildroot unterscheidet zwei Typen von Paketen: die Pakete für den Host und die Pakete für das Target. Die Target-Pakete werden im Verzeichnis `output/build/X-Version/` ausgepackt, wobei X wieder für den Paketnamen steht und »Version« in der Variablen »X_VERSION« (beispielsweise in LIBFOO_VERSION oder MYPROG_VERSION) zu finden ist. Handelt es sich um ein Hostpaket, lautet das Verzeichnis `output/build/host-X-Version/`.

Patchen

Eventuell notwendige Patches werden ebenfalls in das Verzeichnis unterhalb von `output/build/X-Version/` abgelegt. Sie können sich auch bereits im Paket befunden haben. Wichtig ist, dass Patchdateien der folgenden Konvention genügen: Der Name besteht aus drei Teilen. Der

erste Teil ist der Paketname (»libfoo« oder »myprog«), der zweite Teil, vom ersten durch einen Bindestrich (Minus) getrennt, ist frei wählbar und der dritte Teil, vom zweiten durch einen Punkt getrennt, besteht aus der Dateierweiterung »patch«. Beispiel: libfoo-a20130901.patch.

Gibt es mehrere Patchdateien, werden diese in alphabetischer Reihenfolge angewendet.

Konfiguration

Für die Konfiguration greift Buildroot im Fall eines »generic package« auf die Kommandos zurück, die in der Make-Variablen »X_CONFIGURE_CMDS« abgelegt sind. In Beispiel 4-14 ist die Anwendung in Zeile 16 bis 18 dargestellt.

Für die übrigen Paketarten wird diese Variable nicht benötigt. Bei diesen ruft Buildroot das Skript `./configure` mit den Parametern auf, die in der Variablen »X_CONF_OPT« abgelegt sind.

Generierung

In diesem Schritt werden die Kommandos ausgeführt, die unter der Make-Variablen »X_BUILD_CMDS« abgelegt sind (generic package). Sind hier keine Kommandos spezifiziert, ruft Buildroot nur `make` auf.

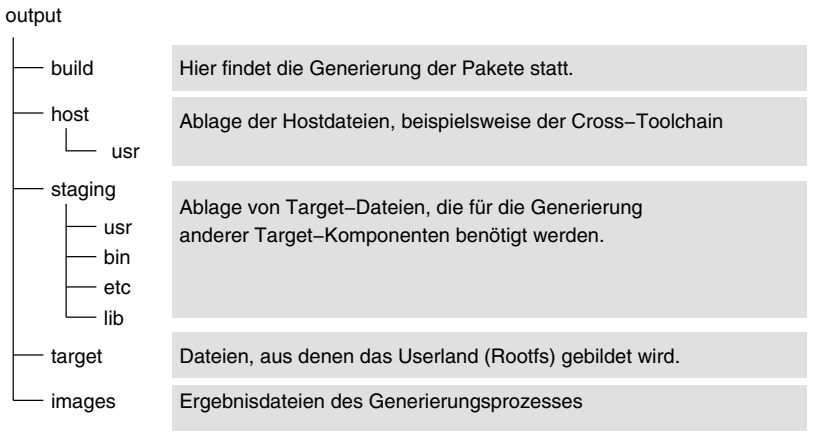


Abb. 4-10 *Buildroot-Verzeichnisstruktur für Ergebnisdateien*

Installation

Buildroot klassifiziert verschiedene Ergebnisse des Generierungsprozesses und erwartet, dass diese in den zugeordneten Verzeichnissen abgelegt werden (Abb. 4-10). Die Werkzeuge für den Host beispielsweise sind unter `output/host` abzulegen. Das Staging-Verzeichnis ist für Zwischenergebnisse, die für die Generierung von Target-Paketen notwendig sind. Das sind beispielsweise statische Target-Bibliotheken, die nur von

einzelnen Programmen benötigt werden. Im Verzeichns `output/target/` baut Buildroot das Rootfilesystem zusammen und legt das Ergebnis, Kernel und Rootfilesystem schließlich in `output/images/` ab.

Über die Variablen »X_INSTALL_CMDS«, »X_INSTALL_STAGING_CMDS«, »X_INSTALL_IMAGES_CMDS« und »X_INSTALL_TARGET_CMDS« lassen sich die Ergebnisse in den Verzeichnissen ablegen.

Darüber hinaus gibt es noch Variablen wie »X_DEVICES«, »X_PERMISSIONS« oder »X_USERS«, über die Gerätedateien, Zugriffsrechte für einzelne Dateien und Einträge für Benutzer konfiguriert werden können (siehe Beispiel 4-14).

Clean und uninstall

Über die Variablen »X_CLEAN_CMDS« und »X_UNINSTALL_TARGETS_CMDS« beziehungsweise »X_UNINSTALL_HOST_CMDS« lassen sich Aktionen angeben, die beim Aufruf von `make clean` beziehungsweise `uninstall` durch Buildroot abgearbeitet werden.

4.4.2 Praxis

Um ein eigenes Paket in die Systemgenerierung von Buildroot zu integrieren, sind die folgenden Schritte notwendig:

1. Erstellen des Verzeichnisses, dessen Name aus dem Paketnamen und einer Versionsnummer besteht. In dem Verzeichnis befinden sich die Quellcodedateien des zu integrierenden Paketes.
2. Erstellen eines Buildroot-Verzeichnisses, in dem Bauvorschrift und Auswahldatei abgelegt wird.
3. Erstellen der Buildroot-Bauvorschrift `hello.mk` im neu angelegten Buildroot-Verzeichnis
4. Erstellen der Paket-Auswahldatei `Config.in` im neu angelegten Buildroot-Verzeichnis
5. Einfügen des Pfades zur Paket-Auswahldatei in der Datei `Config.in` im Hauptverzeichnis von Buildroot
6. Im Buildroot-Verzeichnis `make menuconfig` aufrufen und das neue Paket auswählen
7. Das System per `make` neu generieren

Im Folgenden soll eine Hello-World-Applikation in die Systemgenerierung von Buildroot integriert werden. Die Ingegration von Gerätetreibern ist in Abschnitt 8.3 exemplarisch dargestellt.

Applikation

Zunächst benötigen wir das Paket, das integriert werden soll. Als Beispiel dient das Hello-World-Programm aus Beispiel 4-15. Dieses gibt den Text »Hello World« nicht nur einmal aus, sondern im Sekundenabstand. Für das Schlafenlegen zwischen zwei Ausgaben verwenden wir wieder die moderne Funktion `clock_nanosleep()`, um damit das Linken einer zusätzlichen Bibliothek zu demonstrieren [QuMä2012]. Den Quellcode des Programms legen Sie im Verzeichnis `~/embedded/application/hello-20130901/` unter dem Namen `hello.c` ab.

Beispiel 4-15 *Hello World <hello.c>*

```
#include <stdio.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
    struct timespec sleeptime;

    sleeptime.tv_sec = 1;
    sleeptime.tv_nsec= 0;
    while ( 1 ) {
        printf("Hello World\n");
        clock_nanosleep( CLOCK_MONOTONIC, 0, &sleeptime, NULL );
    }
    return 0;
}
```

Außerdem wird noch ein einfaches Makefile benötigt, das Sie in Beispiel 4-16 sehen. Dieses wird ebenfalls im Verzeichnis `~/embedded/application/hello-20130901/`, dieses Mal unter dem Namen `Makefile`, abgelegt.

Beispiel 4-16 *Makefile zum Hello-World-Programm <Makefile>*

```
CFLAGS=-g -Wall
LDLIBS=-lrt

all: hello

clean:
    rm -rf *.o hello
```

Wenn Sie auf dem Entwicklungsrechner im Verzeichnis `hello-20130901make` eingeben, müsste der Quellcode kompiliert werden. Der nachfolgende Aufruf von `./hello` gibt den bekannten String »Hello World« auf dem Bildschirm aus.

Buildroot-Integration

Für die Buildroot-Integration werden, wie bereits beschrieben, zwei Dateien benötigt, die in ein neues Verzeichnis unterhalb von `package/` gelegt werden.

```
#############################################################
#
# hello
#
#############################################################

HELLO_VERSION = 20130901
HELLO_SITE_METHOD = local
HELLO_SITE = ../../application/hello-$(HELLO_VERSION)

define HELLO_BUILD_CMDS
   $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D)
endef

define HELLO_INSTALL_TARGET_CMDS
   $(INSTALL) -m 0755 -D $(@D)/hello $(TARGET_DIR)/usr/bin/hello
endef

define HELLO_CLEAN_CMDS
   $(MAKE) -C $(@D) clean
endef

define HELLO_UNINSTALL_TARGET_CMDS
   rm -f $(TARGET_DIR)/usr/bin/hello
endef

$(eval $(generic-package))
```

Beispiel 4-17
Buildroot-Bauvorschrift für das Hello-World-Programm <hello.mk>

```
config BR2_PACKAGE_HELLO
   bool "Hello Programm"
   help
     Programm to test the integration of own
     packages in buildroot.
```

Beispiel 4-18
Paket-Auswahldatei <Config.in>

Beispiel 4-18 zeigt die Konfigurationsdatei, um das Paket auswählbar zu machen. Der Bauvorschrift in Beispiel 4-17 ist zu entnehmen, dass Buildroot die Dateien direkt von einem Verzeichnis in das andere kopieren kann (Methode »local«). Wir benötigen ein Build-Kommando. Damit das Cross-Kompilieren funktioniert, ist es wichtig, die Variable $(CC) auf den Target-Compiler zu setzen. Damit findet `make` auch die zugehöri-

ge Bibliothek. Die Option »-C $(@D)« bedeutet, dass als Makefile dasjenige verwendet werden soll, das dem Paket beiliegt.

Sinnvoll ist es auch, ein Clean-Kommando anzugeben, um das Neugenerieren des Paketes zu vereinfachen.

Die beiden Dateien müssen in einem neuen Verzeichnis `~/embedded/buildroot-2013.05/package/hello/` abgelegt werden. Als letzten Schritt schließlich muss die Datei `~/embedded/buildroot-2013.05/package/Config.in` um den Eintrag »source "package/hello/Config.in"« ergänzt werden. Dieser Eintrag kann beispielsweise unterhalb von »menu "Miscellaneous"« gemacht werden.

Mit diesen Modifikationen versehen kann im Buildroot-Hauptverzeichnis `make menuconfig` aufgerufen und das neue Paket ausgewählt werden.

Hier noch einmal die notwendigen Befehle in Kurzform:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> cd
../..
quade@felicia:~/embedded> mkdir application
quade@felicia:~/embedded> cd application
quade@felicia:~/embedded/application> mkdir hello-20130901
quade@felicia:~/embedded/application> cd hello-20130901
quade@felicia:~/embedded/application/hello-20130901> gedit hello.c &
# Quellcodedatei anlegen
quade@felicia:~/embedded/application/hello-20130901> gedit Makefile &
# Makefile anlegen
quade@felicia:~/embedded/application/hello-20130901> cd ../../raspi/
buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> mkdir package/hello
quade@felicia:~/embedded/raspi/buildroot-2013.05> gedit \
   package/hello/Config.in &
# Buildroot-Konfigurationsdatei anlegen
quade@felicia:~/embedded/raspi/buildroot-2013.05> gedit \
   package/hello/hello.mk &
# Buildroot-Bauvorschrift anlegen
quade@felicia:~/embedded/raspi/buildroot-2013.05> gedit \
   package/Config.in &
# Unter Miscellaneous "source package/hello/Config.in" einfügen
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
# Unter [Package Selection for the target][Miscellaneous]
# [Hello Programm] auswählen
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Wenn Sie Änderungen am Quellcode `hello.c` vornehmen, erkennt Buildroot diese nicht. Buildroot hat ja seine Kopie unter `output/build/` abgelegt. Daher wird Buildroot beim nächsten `make` das Programm nicht aktualisieren. Sie können dem aber nachhelfen, indem Sie im Buildroot-Hauptverzeichnis ein `make hello-clean` oder `make hello-dirclean` aufrufen. Im ersten Fall ruft Buildroot das in der Bauvorschrift abgelegte Clean-

Kommando auf, im letzten Fall löscht Buildroot gleich seine komplette Kopie des Paketes unter `output/build/`. Verwenden Sie im Zweifelsfall das zweite Kommando. Mit dem nächsten `make` generiert Buildroot das Paket dann neu.

```
quade@felicia:~/embedded/raspi/buildroot-2013.05> make hello-dirclean
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
```

Booten Sie den Raspberry Pi, um die Integration des Programms `hello` zu testen. Die Bauvorschrift (`hello.mk`) hat `hello` in das Verzeichnis `/usr/bin/` verschoben, sodass es nach dem Booten und nach dem Einloggen durch Eingabe von `hello` gestartet werden kann.

4.5 Hinweise zum Backup

Bei jeder Entwicklung ist es erforderlich, regelmäßig Backups durchzuführen. Im einfachsten Fall würde ein Komplett-Backup des Buildroot-Verzeichnisses durchgeführt werden. Das benötigt aber sehr viel Speicher. Ein vorheriges `make clean` löscht unglücklicherweise nicht nur die generierten Dateien, sondern teilweise auch die Konfigurationen.

Geschickter ist es, nur die Konfigurationen abzuspeichern. Typischerweise sind das die Konfigurationen für

- Buildroot selbst,
- den Kernel,
- Busybox und
- die Standard-Library uclibc.

Die Konfigurationsdateien sind in den zugehörigen Verzeichnissen unter dem Namen ».config« zu finden (siehe Tabelle 4-3). Das Abspeichern und auch das Zurückspeichern (restore) automatisieren Sie über ein Skript.

Komponente	Pfad und Name der Konfigurationsdatei
Buildroot	`buildroot-2013.05/.config`
Busybox	`buildroot-2013.05/output/build/busybox-1.21.0/.config`
Kernel	`buildroot-2013.05/output/build/linux-e959a8e/.config`
uclibc	`buildroot-2013.05/output/toolchain/uClibc-0.9.33.2/.config`

Tabelle 4-3 *Pfade zu den Konfigurationsdateien*

Um die Buildroot-Konfiguration zu sichern, bietet Buildroot für `make` das Target »savedefconfig« an. Über die Variable `BR2_DEFCONFIG` kann ein

Pfad plus Dateiname angegeben werden. Der Vorteil dieses Verfahrens besteht darin, dass Buildroot nur die Unterschiede zur Standardkonfiguration ablegt.

Um die Konfiguration wieder einzuspielen, rufen Sie `make defconfig BR2_DEFCONFIG=...` auf. Auch hier geben Sie mit der Variablen `BR2_DEFCONFIG` Pfad plus Name der Konfiguration an. Die folgende Befehlssequenz zeigt die Anwendung:

```
quade@felicia:~/embedded/raspi/buildroot-2013.05>make savedefconfig \
   BR2_DEFCONFIG=~/myconfig.buildroot
quade@felicia:~/embedded/raspi/buildroot-2013.05>make defconfig \
   BR2_DEFCONFIG=~/myconfig.buildroot
#
# configuration written to
# /home/quade/embedded/raspi/buildroot-2013.05/.config
#
quade@felicia:~/embedded/raspi/buildroot-2013.05>
```

5 Anwendungsentwicklung

Kennzeichen eines eingebetteten Systems ist die Fokussierung auf eine eingeschränkte Zahl von Aufgaben. Diese Aufgaben werden durch die »funktionsbestimmende Applikation« realisiert. Gerade im Umfeld von Linux kann die funktionsbestimmende Applikation auf unterschiedlichste Arten und Weisen umgesetzt werden. Möglicherweise ist es eine Webapplikation, die aus statischen Webseiten und PHP-Code besteht, vielleicht die Ansteuerung einer Hardware, die in C realisiert wird, eventuell auch die Kombination aus beidem.

Werden für die funktionsbestimmende Applikation Skriptsprachen eingesetzt, müssen die zugehörigen Interpreter und Bibliotheken auf dem Embedded System installiert sein. Die Installation der Interpreter stellt bei Verwendung von Buildroot kein besonderes Problem dar. Es bietet beispielsweise PHP, Perl oder Python an.

Ein wesentlicher Vorteil von Embedded Linux besteht darin, dass die Entwicklung einer Anwendung für ein eingebettetes System in weiten Teilen identisch mit der Entwicklung einer Applikation für das Hostsystem ist. Unterschiede gibt es in drei Bereichen:

1. Für Programme, die mit einer Programmiersprache wie C oder C++ erstellt werden, die also kompiliert und nicht interpretiert werden, wird eine Cross-Entwicklungsumgebung benötigt.
2. Das eingebettete System stellt typischerweise weniger Ressourcen (Hauptspeicher, Hintergrundspeicher, Rechenleistung) zur Verfügung als ein Hostsystem.
3. Die Ausstattung mit Systemkomponenten wie Bibliotheken ist auf einem eingebetteten System eingeschränkt.

Häufig hat man darüber hinaus vom Typ her andere Applikationen auf einem eingebetteten System zu erstellen als auf einem Hostsystem. Anforderungen an das Zeitverhalten (Realzeitsysteme) spielen ebenso eine Rolle wie der Hardwarezugriff. Da dies meistens in C oder C++ programmiert wird, konzentriert sich der Abschnitt auf diese Sprachfamilie.

5.1 Cross-Development

Anwendungen für eingebettete Systeme werden typischerweise in einer Cross-Entwicklungsumgebung erstellt. Diese haben Sie bereits bei der Erstellung des Systems kennen gelernt. Unter Linux stehen integrierte Entwicklungsumgebungen (IDE) zur Verfügung, häufig wird aber auch der klassische Weg über `make` mit Editor, Compiler und Linker eingeschlagen. Zentral ist dabei ein Makefile, das anstelle der normalen Werkzeuge die Cross-Pendants aufruft. Dazu muss `make` den Ort kennen, an dem die Werkzeuge liegen. Außerdem muss `make` die zugehörigen Bibliotheken verwenden. Das erfolgt über die Option »--sysroot«. In der von Buildroot generierten Toolchain beispielsweise findet sich das Sysroot-Verzeichnis ausgehend vom Buildroot-Hauptverzeichnis unter `/output/host/usr/arm-buildroot-linux-uclibcgnueabi/sysroot/`. Beispiel 5-1 zeigt ein einfaches Makefile für das bekannte Hello-World-Programm, das auf die in `make` eingebauten Regeln setzt, aber dabei für eine Cross-Generierung ausgelegt ist. Allerdings ist vor dem Aufruf von `make` noch die Pfadvariable `PATH` so anzupassen, dass die Programme der Toolchain, zum Beispiel `arm-linux-gcc`, auch gefunden werden.

Beispiel 5-1 *Makefile-Header für Programme, die cross-kompiliert werden <Makefile>*

```
SYSROOT:=~/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot
-linux-uclibcgnueabi/sysroot
CROSS_COMPILE:=arm-linux-

CFLAGS=--sysroot $(SYSROOT) -g -Wall -static
LDLIBS=-lrt
CC=$(CROSS_COMPILE)gcc
LD=$(CROSS_COMPILE)ld
AR=$(CROSS_COMPILE)ar
AS=$(CROSS_COMPILE)as
CXX=$(CROSS_COMPILE)c++

all: hello

clean:
   rm -f *.o hello
```

Auf einem Hostsystem werden Bibliotheken dynamisch zum eigentlichen Programm gebunden. Das hat den Vorteil, dass der Programmcode auf dem Hintergrundspeicher (Festplatte) weniger Platz benötigt. Andererseits funktioniert die Applikation nur dann, wenn auf dem Zielsystem die entsprechende Bibliothek installiert ist. Unterstützt ein eingebettetes System keine dynamischen Bibliotheken oder ist dort eine erforderliche Bibliothek nicht vorhanden, können Sie das Programm statisch linken.

Um ein Programm statisch zu linken, geben Sie dem Compiler das Flag »-static« mit (siehe Beispiel 5-1, Variable `CFLAGS`). Fehlt dieses Flag, wird das Programm dynamisch gelinkt. In der gleichen Variable taucht übrigens noch die Option »-Wall« auf. Das bedeutet, dass alle Compiler-Warnungen eingeschaltet sind. Diese Option sollte grundsätzlich gesetzt sein.

Soll Ihr Programm auf unterschiedlichen Plattformen generiert werden, lassen Sie die ersten beiden Zeilen des Makefiles weg und übergeben die Variablen über den Aufruf von `make`. Das könnte beispielsweise wie folgt aussehen:

```
quade@felicia:/tmp/hello>make \
   CROSS_COMPILE=arm-linux- \
   SYSROOT=~/embedded/raspi/buildroot-2013.05/output/host/usr/arm-buildroot-
linux-uclibcgnueabi/sysroot
```

Transfer zum Target

Nachdem Sie Ihr Programm durch Aufruf von `make` generiert haben, müssen Sie es auf das Zielsystem, das Target, transferieren. Das kann beim Raspberry Pi dadurch geschehen, dass Sie die SD-Karte in den Host einstecken, das Rootfilesystem mounten und die Applikation an die entsprechende Stelle kopieren. Danach wird das Rootfilesystem wieder ausgehängt und die Karte in den Raspberry Pi gesteckt. Dieser bootet und Sie können die Applikation testen.

Sie werden sehr bald feststellen, dass der beschriebene Weg mühsam ist. Erheblich einfacher und schneller geht es, wenn Ihr eingebettetes System über das Netzwerk erreichbar ist. Sie können dann die Applikation per `netcat` (auch `nc` abgekürzt) vom Host auf das Target transportieren. Netcat wird häufig als *Schweizer Taschenmesser* für das Netzwerk bezeichnet. Über die Option »-l« wird ein Server gestartet, ohne diese Option übernimmt das Programm die Funktion des Clients. Außerdem muss noch per Option »-p <portnummer>» eine auf Server und Client identische Portnummer, beispielsweise 4321, mit angegeben werden.

Der Server benötigt zwei Argumente: die IP-Adresse beziehungsweise den Rechnernamen des Gerätes, für den die zu sendenden Daten bestimmt sind, und als Zweites die Portnummer, beispielsweise die erwähnte 4321. Ist der Server gestartet, werden nach Start des Client-Netcats sämtliche Eingaben zum Server übertragen.

Um nicht nur den Inhalt einer Datei, sondern zugleich den Dateinamen, die Zugriffsrechte und Zugriffsmodi mit zu übertragen, können wir die Unix-Funktionalität des »Pipens« einsetzen. Dabei ist die Ausgabe des einen Programms die Eingabe des nächsten. In unserem Fall starten wir auf dem Target — dem Raspberry Pi — den Netcat-Server, der

die übertragenen Daten als ein Tar-File interpretiert und direkt verarbeitet (»nc -l -p 4321 |tar xfv -«). Auf dem Client — dem Entwicklungsrechner — nutzen wir Tar, um die zu übertragenden Daten in ein Archiv zu packen und das Archiv dann per `netcat` an das Target zu übertragen: »tar cfv - <files or dir> | nc embedded.system.ip 4321« (hierbei ist »embedded.system.ip« durch die IP-Adresse des Raspberry Pi zu ersetzen). Nach der Übertragung beendet sich die Serverinstanz von Netcat automatisch.

Um beispielsweise das komplette Verzeichnis `~/embedded/application/` auf das Target in das dortige Heimatverzeichnis zu übertragen, verfahren Sie so, wie in Abbildung 5-1 dargestellt. Wichtig ist allerdings, dass Sie auf dem Target das Programm `netcat` (`nc`) installiert haben. In der Busybox-Konfiguration finden Sie dieses unter dem Menüpunkt [Networking Utilities][nc]. Achten Sie darauf, dass auch der nachfolgende Menüpunkt [Netcat server options (-l)] aktiviert ist.

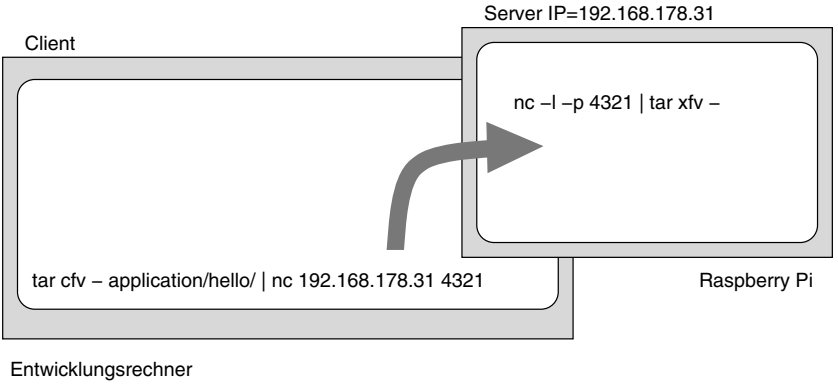


Abb. 5-1
Datentransfer per netcat

Die Übertragung per `netcat` ist sehr effizient, findet allerdings unverschlüsselt statt.

Alternativ installieren Sie auf dem Raspberry Pi einen SSH-Server, den Buildroot zur Auswahl anbietet (`dropbear` oder `openssh`). Dann können Sie die Applikation per `scp` auf den Raspberry Pi kopieren. Sie sparen sich nicht nur das Ein- und Aushängen des Rootfilesystems, sondern auch das durch Entfernen der SD-Karte notwendige, gesonderte Herunter- und Hochfahren des Raspberry Pi.

Der Befehl `scp` funktioniert prinzipiell genau wie `cp`. Allerdings können als Quell- oder Zielangaben dieses Kopierbefehls Loginnamen und Rechnernamen den eigentlichen Pfadinformationen vorangestellt werden. Der notwendige SSH-Server ist immer aktiv, muss also — anders als bei `netcat` — nicht gesondert gestartet werden. Um vom Entwick-

lungsrechner das Verzeichnis `application/hello/` an das eingebettete System mit der IP-Adresse 192.168.178.31 in den Ordner `/tmp/` zu übertragen, kann das folgende Kommando verwendet werden:

```
scp -r application/hello/ root@192.168.178.31:/tmp/
```

Die Option »-r« sorgt übrigens dafür, dass rekursiv sämtliche Dateien und Verzeichnisse übertragen werden. SSH verschlüsselt sämtliche Daten, ist also sicherer als `netcat`, dafür allerdings auch etwas langsamer.

Eine weitere Verkürzung des Entwicklungszyklus erreichen Sie, wenn der Raspberry Pi ein Netzwerkfilesystem wie NFS unterstützt. In diesem Fall muss das Hostsystem per NFS den Ordner exportieren, in dem die Applikation abgelegt wird. Der Raspberry Pi seinerseits hängt den Ordner per NFS in den eigenen Dateibaum mit ein. Dadurch lässt sich der Vorgang des Kopierens sparen.

5.2 Basisfunktionen der eingebetteten Anwendungsprogrammierung

Entwickler der Anwendungssoftware für eingebettete Systeme müssen

- mit wenig Speicherressourcen (sowohl Haupt- als auch Hintergrundspeicher) zurechtkommen,
- davon ausgehen, dass nur eingeschränkt CPU-Leistung zur Verfügung steht,
- so programmieren, dass der Stromverbrauch gering ist,
- sich bewusst sein, dass komplexe Middleware und ausgefuchste Softwarebibliotheken nicht zur Verfügung stehen,
- Anforderungen an das Zeitverhalten gewährleisten (Realtimebetrieb) und
- auf niedriger Ebene Hardware ansprechen.

Dazu modularisiert der Entwickler seine Software, denn das reduziert insbesondere auf Multicore-Maschinen den Stromverbrauch, vereinfacht die Realisierung von Sicherheitsfunktionen auf Basis des Prinzips *Least Privilege* (siehe Abschnitt 7.3.2) und ermöglicht eine bessere Kontrolle des Realzeitverhaltens durch Priorisierung einzelner Threads. Anstelle von ausgewachsenen Datenbanken speichert der Entwickler seine Daten wieder selbst in klassischen Dateien oder verwendet einfache Varianten wie NoSQL. Das spart nicht nur Speicher, sondern benötigt auch weniger Rechenleistung. Das Gleiche gilt für die Hardwarezugriffe. Erfolgen diese direkt, gibt es signifikante Geschwindigkeitsvorteile. Seine Applikation regelmäßig schlafen zu legen, macht sie zwar nicht

schneller, verbraucht aber weniger Rechenzeit und damit weniger Strom.

Im Folgenden sollen daher elementare Funktionen zur Modularisierung auf Basis von Threads vorgestellt werden (Erzeugen und Beenden von Threads). Außerdem werden die Prioritätenvergabe zur Erreichung eines Realzeitverhaltens sowie das Schlafenlegen von Threads demonstriert. Darüber hinaus wird gezeigt, wie aus dem Userland heraus und programmgesteuert auf Hardware allgemein und insbesondere beim Raspberry Pi zugegriffen werden kann.

5.2.1 Modularisierung

Auch wenn der Raspberry Pi selbst nur ein Singlecore-System ist, werden in eingebetteten Systemen zunehmend Multicore-Prozessoren eingesetzt. Das hängt mit der stetig steigenden Anforderung an Rechenleistung bei gleichzeitig niedrigem Stromverbrauch zusammen. Wird die Rechenleistung durch Höhertaktung verdoppelt, vervierfacht sich der Stromverbrauch. Bei Multicore-Systemen hängen Leistung und Stromverbrauch nicht quadratisch, sondern linear voneinander ab. Eine Verdoppelung der Leistung durch einen zweiten Prozessorkern verdoppelt den Stromverbrauch. Diese Variante hat jedoch einen Nachteil: Die erhöhte Rechenleistung kann nur genutzt werden, wenn die Software entsprechend modular konzipiert wird.

Folglich ist die Applikation vom Entwickler weitestgehend parallel aufzubauen. Dazu bieten sich Threads an, deren Programmierinterface über Posix standardisiert ist. Die folgenden Informationen sind [QuMä2012] entnommen.

Threads werden über die Funktion `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)` erzeugt. Intern verwendet `pthread_create()` den Systemcall `clone()`. Um die Funktion nutzen zu können, muss die Bibliothek `libpthread` zum Programm gebunden werden (im Makefile über die Variable `LDLIBS=-lpthread`). Anders als bei `fork()` startet der neue Thread die Verarbeitung in einer separaten Funktion, deren Adresse über den Parameter `start_routine` übergeben wird. Dieser Funktion wird beim Aufruf das Argument `arg` übergeben. Der Parameter `attr` steuert die Erzeugung, `thread` enthält nach dem Aufruf die Kennung des neuen Threads. `pthread_create()` gibt im Erfolgsfall `0` zurück, ansonsten einen Fehlercode.

Beispiel 5-2
Thread-Erzeugung

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *child( void *args )
{
    printf("The child thread has ID %d\n", getpid() );
    // You can either call pthread_exit or "return".
    // pthread_exit( NULL );
    return NULL;
}

int main( int argc, char **argv, char **envp )
{
    pthread_t child_thread;

    if( pthread_create( &child_thread, NULL, child, NULL )!= 0 ) {
        fprintf(stderr,"creation of thread failed\n");
        return -1;
    }
    /* Now the thread is running concurrent to the main task        */
    /* But it won't return to this point (aulthough it might        */
    /* execute a  return statement at the end of the Child routine).*/

    /* waiting for the termination of child_thread */
    pthread_join( child_thread, NULL );
    printf("end of main thread\n");
    return 0;
}
```

Ein Thread beendet sich selbst, indem er `void pthread_exit(void *retval)` aufruft. Der Eltern-Thread verwendet `int pthread_join(pthread_t thread, void **retval)`, um auf den Exitcode des Threads `thread` zu warten (siehe Beispiel 5-2).

Per `int kill(pid_t pid, int sig)` schickt ein Prozess einem anderen Job, der die Prozessidentifikationsnummer `pid` besitzt, das Signal `sig`. Typischerweise führt dies dazu, dass sich der Job daraufhin beendet. Allerdings können die Signale durch die Programme — mit Ausnahme von `SIGKILL` (Signalnummer 9) — ignoriert oder abgefangen und damit das Beenden verhindert oder auch nur verzögert werden.

```
if (kill(child_pid,SIGINT)==-1) {
    perror("kill failed");
}
```

Die eigene Prozessidentifikationsnummer (PID) wird durch `pid_t getpid(void)` abgefragt, die Thread-Identifikationsnummer (TID) per `pid_t gettid(void)`. Allerdings gibt es für diesen Linux-spezifischen Systemcall keinen direkten Funktionsaufruf in der Standard-C-Bibliothek. Das ist beispielsweise eine Funktion, die auf dem eingebetteten System nicht zur Verfügung steht.

Eltern-Threads können Kind-Threads durch Aufruf von `int pthread_kill(pthread_t thread, int sig);` dazu bewegen, sich zu beenden (Beispiel 5-3).

Beispiel 5-3 *Threads killen*

```
pthread_t child_thread;

if( pthread_create( &child_thread, NULL, child, NULL )!= 0 ) {
    fprintf(stderr,"creation of thread failed\n");
    return -1;
}
...
pthread_kill( child_thread, SIGINT );
pthread_join( child_thread, NULL );
...
```

5.2.2 Realzeitaspekte

Hat das eingebettete System Anforderungen zeitlicher Natur zu erfüllen, ist eine angepasste Programmierung erforderlich. Dazu gehört vor allem die Vergabe von Realzeitprioritäten, die die bevorzugte Abarbeitung von Threads kontrolliert. Außerdem müssen Threads häufig Zeitmessungen durchführen und für eine definierte Zeit schlafen gelegt werden (um beispielsweise eine Totzeit abzuwarten). Einige hierfür geeignete Funktionen sollen kurz vorgestellt werden, detailliertere und erweiterte Informationen finden Sie in [QuMä2012].

Realzeitprioritäten vergeben

Realzeitprioritäten können über das Kommandozeilenwerkzeug `chrt` beim Start der Applikation oder sogar für eine laufende Applikation vergeben werden. Ein Programm kann jedoch auch selbst durch den Systemaufruf `sched_setscheduler()` die Realzeitpriorität einstellen.

Das Kommando `chrt -m` zeigt die minimal und maximal zu vergebende Priorität an:

```
quade@felicia:~/embedded>chrt -m
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority    : 1/99
```

```
SCHED_RR min/max priority    : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority    : 0/0
```

Um ein Programm mit einer bestimmten Priorität zu starten, geben Sie `chrt`, gefolgt von der neuen Priorität, und dann den Programmnamen inklusive der für das Programm erforderlichen Argumente an. Die Option »-r« stellt im Fall mehrerer Jobs mit gleicher Priorität ein Round Robin Scheduling (RR) ein, »-f« First Come First Serve (FIFO).

```
quade@felicia:~/embedded>sudo chrt 50 application/hello-20130901/hello
```

Um eine Realzeitpriorität zu vergeben, müssen entsprechende Rechte, typischerweise Root-Rechte vorliegen. Per `ps -ce` geben Sie eine Liste der Rechenprozesse inklusive der zugehörigen Priorität aus. Beachten Sie dabei aber, dass alle Realzeitprioritäten mit einem Offset von 40 angezeigt werden. Eine Realzeitpriorität von 50 steht daher mit 90 in der Liste. Genauere Informationen hierzu finden Sie in [QuMä2012].

```
quade@felicia:~/embedded>ps -ce
...
4896 TS   19 ?        00:00:00 kworker/1:1
4897 TS   19 pts/3    00:00:00 sudo
4898 RR   90 pts/3    00:00:00 hello
4899 TS   19 pts/3    00:00:00 ps
```

Programmgesteuert verwenden Sie zur Vergabe einer Realzeitpriorität die Funktion `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param)`, die für den Job mit der Priorität `pid` das Scheduling-Verfahren `policy` für die in der Datenstruktur `param` definierte Prioritätsebene einstellt. Ist der Parameter `pid0`, gelten die Einstellungen für den aufrufenden Prozess. Folgende Scheduling-Verfahren stehen zur Verfügung:

- SCHED_RR: Round Robin
- SCHED_FIFO: First Come First Serve
- SCHED_OTHER: Defaultverfahren für Jobs ohne (Realzeit-)Priorität

Die gerade aktiven Scheduling-Parameter werden durch Aufruf von `int sched_getparam(pid_t pid, struct sched_param *param)` ausgelesen. `pid` referenziert wieder den Job. An der mit `param` übergebenen Speicheradresse legt die Funktion später die Parameter (Priorität, Prioritätsbereich etc.) ab.

Um das Programm aus Beispiel 5-4 zu generieren, verwenden Sie das Kommando `make LDLIBS=-lrt`. Damit wird die für die Funktion `clock_nanosleep()` benötigte Realzeitbibliothek `librt` dazugebunden.

Beispiel 5-4
Tasks parametrieren

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <time.h>

#define PRIORITY_OF_THIS_TASK     15

char *Policies[] = {
   "SCHED_OTHER",
   "SCHED_FIFO",
   "SCHED_RR"
};

static void print_scheduling_parameter()
{
   struct timespec rr_time;

   printf("Priority-Range SCHED_FF: %d - %d\n",
      sched_get_priority_min(SCHED_FIFO),
      sched_get_priority_max( SCHED_FIFO ) );
   printf("Priority-Range SCHED_RR: %d - %d\n",
      sched_get_priority_min(SCHED_RR),
      sched_get_priority_max( SCHED_RR));
   printf("Current Scheduling Policy: %s\n",
      Policies[sched_getscheduler(0)] );
   sched_rr_get_interval( 0, &rr_time );
   printf("Intervall for Policy RR: %ld [s] %ld [nanosec]\n",
      rr_time.tv_sec, rr_time.tv_nsec );
}

int main( int argc, char **argv )
{
   struct sched_param scheduling_parameter;
   struct timespec t;

   print_scheduling_parameter();
   sched_getparam( 0, &scheduling_parameter );
   printf("Priority: %d\n", scheduling_parameter.sched_priority );

   // only superuser:
   // change scheduling policy and priority to realtime priority
   scheduling_parameter.sched_priority = PRIORITY_OF_THIS_TASK;
   if( sched_setscheduler( 0, SCHED_RR, &scheduling_parameter )!= 0 ) {
      perror( "Set Scheduling Priority" );
      exit( -1 );
   }
   sched_getparam( 0, &scheduling_parameter );
   printf("Priority: %d\n", scheduling_parameter.sched_priority );
```

```
    t.tv_sec = 10;          // sleep
    t.tv_nsec = 1000000000;
    clock_nanosleep( CLOCK_MONOTONIC, 0, &t, NULL );

    print_scheduling_parameter();
    return 0;
}
```

Zeiten messen

Wer auf moderne Art die aktuelle Zeit mit hoher Genauigkeit lesen will, ruft die Funktion `clock_gettime()` auf. Diese bekommt zwei Parameter übergeben. Der erste Parameter spezifiziert die sogenannte Zeitquelle, die Clock. Dabei stehen unter anderem die Zeitquellen `CLOCK_MONOTONIC` und `CLOCK_REALTIME` zur Verfügung. Der Aufruf von `clock_gettime()` mit `CLOCK_REALTIME` liefert die Zeit, die seit dem 1.1.1970 vergangen ist, in Sekunden und die Restzeit in Nanosekunden zurück. Die Zeitquelle `CLOCK_MONOTONIC` liefert die Zeit in Sekunden und die Restzeit in Nanosekunden seit einem im System festgelegten Zeitpunkt, beispielsweise Start des Systems oder aber auch Start des Threads, zurück. Die Zeitquelle eignet sich daher gut für eine Differenzzeitmessung, bei der die Dauer einer Aktion ausgemessen werden soll. Die Zeitquelle `CLOCK_MONOTONIC` zählt immer weiter. Wird an der Systemuhr gedreht (beispielsweise bei der Sommerzeitumstellung), bleibt `CLOCK_MONOTONIC` davon unberührt. `CLOCK_REALTIME` jedoch reagiert darauf. Daher ist die Wahl der geeigneten Zeitquelle durch den Programmierer wichtig.

Der zweite Parameter ist die Adresse einer Datenstruktur vom Typ `struct timespec`. In diese Datenstruktur legt der Kernel das Ergebnis, also die angeforderte Zeit, ab. Die Datenstruktur hat die beiden Elemente `tv_sec` und `tv_nsec`. Die erste Variable (`tv_sec`) enthält die Sekunden, die zweite (`tv_nsec`) die zusätzlich seit der letzten vollen Sekunde vergangenen Nanosekunden (Restzeit). Damit sind die Zeitangaben im Rechner so aufgebaut, wie unsere Zeitangaben, beispielsweise wie Stunde und Minute.

```
struct timepspec timestamp;
...
if (clock_gettime(CLOCK_MONOTONIC,&timestamp)) {
    perror("timestamp");
    return -1;
}
printf("seconds: %ld, nanoseconds: %ld\n",
    timestamp.tv_sec, timestamp.tv_nsec);
```

Die Funktion `clock_gettime()` steht nur über die Realzeitbibliothek `librt` zur Verfügung. Beim Linken ist daher die Option `-lrt` mit anzugeben.

Schlafen legen

Auch zum Schlafenlegen wird eine Funktion mit dem Präfix `clock_` gewählt, nämlich `int clock_nanosleep(clockid_t clock_id, int flags, const struct timespec *request, struct timespec *remain);`. Über den Parameter `clock_id` wird die Zeitquelle (`CLOCK_REALTIME`, `CLOCK_MONOTONIC`) eingestellt. Der Parameter `flags` erlaubt die Angabe, ob die Zeitangabe `request` relativ (`flags==0`) oder absolut (`flags==TIMER_ABSTIME`) zu interpretieren ist. Um die Restzeit bei einem durch ein Signal provozierten vorzeitigen Abbruch aufzunehmen, kann per `remain` eine Speicheradresse dafür übergeben werden. `request` schließlich enthält die Zeitangabe, bis zu der (`TIMER_ABSTIME`) oder die der Job schlafen soll. Bei Angabe der absoluten Zeitangabe ist auf die Normierung zu achten: Wird der Nanosekundenanteil größer oder gleich eine Milliarde, repräsentiert dieser Anteil also eine Sekunde oder mehr, ist mithilfe der Division und der Modulo-Operation eine Anpassung notwendig:

```
if (sleeptime.tv_nsec>999999999) {
    sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
    sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
}
```

Die professionelle Programmierung mit Posix-Funktionen zum Schlafenlegen eines Jobs demonstriert Beispiel 5-5. Beispiel 5-6 zeigt die einfachere Variante des Schlafens mit relativer Zeitangabe.

Beispiel 5-5
Schlafenlegen einer Task mit absoluter Zeitangabe

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

void sigint_handler(int signum)
{
   printf("SIGINT (%d)\n", signum);
}

int main( int argc, char **argv, char **envp )
{
   struct timespec sleeptime;
   struct sigaction sa;

   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
   sa.sa_handler = sigint_handler;
```

```
   if (sigaction(SIGINT, &sa, NULL) == -1) {
      perror( "sigaction" );
      return -1;
   }
   printf("%d sleeps for 10 seconds and 1 millisecond...\n", getpid());
   clock_gettime( CLOCK_MONOTONIC, &sleeptime );
   sleeptime.tv_sec  += 10;
   sleeptime.tv_nsec += 1000000;
   if (sleeptime.tv_nsec>999999999) {
      sleeptime.tv_sec += sleeptime.tv_nsec/1000000000;
      sleeptime.tv_nsec = sleeptime.tv_nsec%1000000000;
   }
   while (clock_nanosleep(CLOCK_MONOTONIC,TIMER_ABSTIME,
      &sleeptime,NULL)==EINTR) {
      printf("interrupted...\n");
   }
   printf("woke up...\n");
   return 0;
}
```

```
struct timespec sleeptime;
...
sleeptime.tv_sec  = 0;
sleeptime.tv_nsec = 250000000; /* 250 Millisekunden */
if ((error=clock_nanosleep(CLOCK_MONOTONIC,0,&sleeptime,NULL))!=0 ) {
    printf("clock_nanosleep reporting error %d\n", error);
}
```

Beispiel 5-6
Schlafenlegen einer Task mit relativer Zeitangabe

Die Funktion `clock_nanosleep()` steht nur über die Realzeitbibliothek `librt` zur Verfügung. Beim Linken ist daher die Option `-lrt` mit anzugeben.

5.3 Hardwarezugriffe

In einer Vielzahl von Fällen sind eingebettete Systeme direkt mit der Umwelt über Sensoren und Aktoren verbunden, die über die Applikation angesprochen werden. Dazu gibt es bereits vorgefertigte Interfaces beziehungsweise Funktionen, die die Applikation verwendet. Diese Funktionen und Interfaces sind typischerweise einfach, teilweise sogar ohne explizite Programmierung zu verwenden. Andererseits bieten sie nicht immer alle Möglichkeiten der angesprochenen Hardware, sind häufig sicherheitstechnisch betrachtet kritisch (wenn beispielsweise Hardwareregister per `mmap` in den Adressraum der Applikation eingeblendet werden) und bringen häufig einen nicht zu vernachlässigenden, zeitlichen Overhead mit sich.

Sind höhere zeitliche Anforderungen zu erfüllen, kann der Hardwarezugriff durch Kernelprogrammierung unterstützt werden. Hierfür ist ein Gerätetreiber zu implementieren. Der Gerätetreiber ist ohnehin notwendig, wenn eine neuartige Hardware angesprochen werden soll, für die bisher kein Treiber existiert (dazu mehr in Kapitel 6).

Im Folgenden wird der Zugriff auf Hardware für Applikationen aus dem Userland vorgestellt.

5.3.1 Systemcalls für den Hardwarezugriff

Linux stellt für den Zugriff auf Hardware im Wesentlichen zwei Schnittstellen zur Verfügung: Gerätedateien und das Sys-Filesystem. Gerätedateien stellen dabei die klassische und vor allem auch direkte Zugriffsvariante auf Hardware dar. Allerdings muss für den Zugriff in den meisten Fällen Programmcode geschrieben werden. Auf das Sys-Filesystem greifen Entwickler dagegen per Skripte zu, nehmen dafür aber langsamere Zugriffe in Kauf.

Gerätedateien sehen aus Sicht eines Programmierers aus wie andere Dateien auch, auf die er mit den Funktionen `open()`, `read()`, `write()` und `close()` zugreift. Daten, die er lesen oder schreiben will, kommen und gehen allerdings nicht zur Festplatte, sondern werden vom Kernel zu und von einem an die Gerätedatei angekoppelten Gerätetreiber weitergeleitet. Der Vorteil liegt auf der Hand: Der Programmierer muss für den Hardwarezugriff kein spezielles Interface erlernen.

Mit den Standardfunktionen `open()`, `read()`, `write()` und `close()` erfolgt der Zugriff in mehreren Schritten. Im ersten Schritt teilt die Applikation durch Angabe der Gerätedatei dem Betriebssystem mit, auf welche Peripherie sie in welcher Art (zum Beispiel lesend oder schreibend) zugreifen möchte. Das Betriebssystem prüft anhand der mit der Gerätedatei verbundenen Zugriffsrechte, ob der Zugriff möglich und erlaubt ist. Ist dies der Fall, bekommt die Applikation die Zugriffsberechtigung in Form eines Handles beziehungsweise Deskriptors mitgeteilt (Rückgabewert von `open()`). Im zweiten Schritt greift die Applikation mithilfe des Deskriptors auf die Peripherie so oft und so lange wie notwendig zu (`read()` und `write()`). Erst wenn keine Zugriffe mehr notwendig sind, wird das Handle beziehungsweise der Deskriptor wieder freigegeben (Schritt 3).

Der Funktion `int open(const char *pathname, int flags)` wird neben dem Namen der Gerätedatei »pathname« mit »flags« ein Parameter zur Angabe der Zugriffsart (lesend, schreibend, nicht blockierend) übergeben. Die Zugriffsarten sind in der Headerdatei `<fcntl.h>` wie folgt definiert:

- `O_RDONLY`: Lesender Zugriff
- `O_WRONLY`: Schreibender Zugriff
- `O_RDWR`: Lesender und schreibender Zugriff
- `O_NONBLOCK`: Nicht blockierender Zugriff

Ein negativer Rückgabewert der Funktion `open` bedeutet, dass der Zugriff nicht möglich ist. Anhand der (Thread-)globalen Variablen `errno` kann die Applikation die Ursache abfragen. Ein positiver Rückgabewert repräsentiert das Handle beziehungsweise den Deskriptor.

ssize_t und size_t

Viele Standardfunktionen verwenden den Datentyp `size_t` (size type, vorzeichenlos) oder `ssize_t` (signed size type, vorzeichenbehaftet). Typischerweise repräsentiert er den originären Typ `unsigned long` beziehungsweise `long`.

Applikationen greifen auf die Peripherie dann über die Funktionen `ssize_t read(int fd, void *buf, size_t count)` und `ssize_t write(int fd, const void *buf, size_t count)` zu. Der Parameter `fd` ist der Filedeskriptor (Handle), der von `open` zurückgegeben wird. Die Adresse `buf` enthält den Speicherbereich, in den `read` die Daten ablegt, und `count` gibt an, welche Größe dieser Speicherbereich hat.

`read` kopiert von der Peripherie mindestens ein Byte und maximal `count` Bytes an die Speicheradresse `buf` und gibt — solange kein Fehler aufgetreten ist — die Anzahl der kopierten Bytes zurück. Gibt es zum Zeitpunkt des Aufrufes der Funktion `read` keine Daten, die gelesen werden können, legt die Funktion im blockierenden Modus den aufrufenden Thread schlafen und weckt ihn wieder auf, wenn mindestens ein Byte in den Speicherbereich `buf` abgelegt wurde. Werden jedoch wie bei einem Dateizugriff am Ende der Datei keine Daten mehr erwartet, gibt die Funktion direkt 0 zurück. Im nicht blockierenden Modus (`O_NONBLOCK`) quittiert die Funktion den Umstand, dass keine Daten zur Verfügung stehen, mit einem negativen Rückgabewert. Die (Thread-)globale Variable `errno` hat dann den in der Headerdatei <errno.h> definierten Wert `EAGAIN`.

Auch im Fehlerfall, wenn beispielsweise der Aufruf durch ein Signal unterbrochen wurde, gibt `read` einen negativen Wert zurück und `errno` enthält den zugehörigen Fehlercode.

Die Funktion `write` arbeitet ähnlich. Die Funktion schreibt mindestens ein Byte, maximal aber `count` Bytes auf das über den Filedeskriptor `fd` spezifizierte Gerät. Falls nicht geschrieben werden kann, weil beispielsweise das Sende-Fifo einer seriellen Schnittstelle bereits voll ist,

wird der zugreifende Thread in den Zustand *schlafend* versetzt, bis der Zugriff möglich ist. Beim nicht blockierenden Zugriff gibt `write` in diesem Fall — schreiben ist zurzeit nicht möglich — einen negativen Wert zurück und `errno` enthält wieder den Wert `EAGAIN`.

Direct-IO versus Buffered-IO

Wenn Ein- und Ausgabebefehle wie die Systemcalls `read()` und `write()` direkt, ohne Verzögerung, umgesetzt werden, spricht man von *Direct-IO*.

Funktionen wie beispielsweise `fprintf()`, `fread()`, `fwrite()`, `fscanf()` und auch `printf()` gehören zur *Buffered-IO*. Bei dieser puffert das Betriebssystem (genauer die Bibliothek) die Daten aus Gründen der Effizienz zwischen. Erst wenn es sinnvoll erscheint, werden die zwischengespeicherten Daten per Systemcall `read()` oder `write()` transferiert. Dies ist beispielsweise der Fall, wenn ein »\n« ausgegeben wird oder wenn 512 Byte Daten gepuffert sind.

Da nur die Direct-IO-Funktionen die volle Kontrolle über die Ein- und Ausgabe geben, werden in Realzeitapplikationen nur diese für den Datentransfer mit der Peripherie eingesetzt.

Beispiel 5-7
Der Gerätezugriff über read und write

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

static char *hello = "Hello World";

int main( int argc, char **argv, char **envp )
{
   int dd; /* device descriptor */
   ssize_t bytes_written, bytes_to_write;
   char *ptr;

   dd = open( "/dev/ttyS0", O_RDWR ); /* blocking mode */
   if (dd<0) {
      perror( "/dev/ttyS0" );
      return -1;
   }

   bytes_to_write = strlen( hello );
   ptr = hello;
   while (bytes_to_write) {
      bytes_written = write( dd, hello, bytes_to_write );
      if (bytes_written<0) {
         perror( "write" );
         close( dd );
```

```
            return -1;
        }

        bytes_to_write -= bytes_written;
        ptr += bytes_written;
    }
    close( dd );
    return 0;
}
```

Der Zugriffsmodus (blockierend oder nicht blockierend) kann im übrigen per `int fcntl(int fd, int cmd, ... /* arg */)` auch nach dem Öffnen noch umgeschaltet werden (Beispiel 5-8). Dazu werden zunächst mit dem Kommando (`cmd`) `F_GETFL` die aktuellen Flags gelesen. Dann wird entweder per Oder-Verknüpfung der nicht blockierende Modus gesetzt beziehungsweise per XOR-Verknüpfung der blockierende Modus wieder aktiviert.

Beispiel 5-8
Zugriffsmodus umschalten

```
int fd, fd_flags, ret;
...
fd_flags = fcntl( fd, F_GETFL );
if (fd_flags<0) {
    return -1; /* Fehler */
}
fd_flags|= O_NONBLOCK; /* nicht blockierenden Modus einschalten */
if (fcntl( fd, F_SETFL, (long)fd_flags )<0 ) {
    return -1; /* Fehler */
}
...
fd_flags = fcntl( fd, F_GETFL );
if (fd_flags<0) {
    return -1; /* Fehler */
}
fd_flags~= O_NONBLOCK; /* nicht blockierenden Modus einschalten */
if (fcntl( fd, F_SETFL, (long)fd_flags )<0 ) {
    return -1; /* Fehler */
}
...
```

Werden regelmäßig (im nicht blockierenden Modus) Peripheriegeräte daraufhin abgeprüft, ob Daten zum Lesen vorliegen oder ob Daten geschrieben werden können, spricht man von Polling. Polling ist insofern ungünstig, als ein Rechner auch dann aktiv wird, wenn eigentlich nichts zu tun ist. Besser ist der ereignisgesteuerte Zugriff (blockierender Modus), bei dem die Applikation nur dann aktiv wird, wenn Daten gelesen oder geschrieben werden können. Allerdings bringt der blockierende

Modus in unixartigen Systemen den Nachteil mit sich, dass per `read` oder `write` immer nur eine Quelle abgefragt werden kann. Oftmals sind aber mehrere Datenquellen oder Datensenken abzufragen.

Die moderne Variante, mehrere Ein- oder Ausgabekanäle zu überwachen, basiert auf der Thread-Programmierung (Abschnitt 5.2.1). Für jeden Kanal (Filedeskriptor, Handle) wird ein eigener Thread aufgezogen, der dann blockierend per `read` oder `write` zugreift. Neben der einfacheren und übersichtlicheren Programmierung hat das auch den Vorteil, dass damit direkt das nebenläufige Programmieren (Stichwort Multicore-Architektur) unterstützt wird (Beispiel 5-9).

Beispiel 5-9
Asynchroner Zugriff über Threads

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>

static void *async_thread( void *arg )
{
   int fd=*((int*)(&arg)); // casting to avoid compiler warning
   char buf[64];

   read( fd, buf, sizeof(buf));
   pthread_exit( NULL );
}

int main( int argc, char **argv, char **envp )
{
   pthread_t async_thread_id;
   int fd;

   fd = open( "device.io", O_RDONLY );
   if( fd<0 ) {
      perror( "device.io" );
      return -1;
   }

   if( pthread_create(&async_thread_id,NULL,async_thread,
      (void*)(long)fd)!= 0 ) { // double cast to avoid warning
      fprintf(stderr,"creation of thread failed\n");
      return -1;
   }
   // perform other tasks
   // ...

   // synchronise with file I/O
   pthread_join( async_thread_id, NULL );
   // ...
```

```
   return 0;
}
```

Sind keine Zugriffe auf die Peripherie über den Deskriptor mehr notwendig, kann dieser per `int close(int fd)` wieder freigegeben werden. In vielen Anwendungen fehlt jedoch zu einem `open` das korrespondierende `close`. Das ist insofern nicht dramatisch, als Linux beim Ende einer Applikation alle noch offenen Deskriptoren von sich aus wieder freigibt.

Manche Peripheriegeräte, wie beispielsweise eine Grafikkarte, transferieren nicht nur einzelne Bits oder Bytes, sondern umfangreiche Speicherbereiche. Diese Daten über einzelne `read`- und `write`-Aufrufe zu verschieben, wäre sehr ineffizient: Mit jedem Zugriff ist ein Kontextwechsel notwendig und außerdem wird zumindest in der Applikation ein Speicherbereich benötigt, in den beziehungsweise von dem die Daten zwischen Applikation und Hardware transferiert werden. Daher bieten Betriebssysteme und die zu den Geräten gehörenden Treiber oft die Möglichkeit, Speicherbereiche der Hardware (oder aber auch des Kernels) in den Adressraum einer Applikation einzublenden. Die Applikation greift dann direkt auf diese Speicherbereiche zu.

Dazu öffnet die Applikation als Erstes die Gerätedatei (Funktion `open`), die den Zugriff auf die gewünschte Peripherie ermöglicht. Der zurückgelieferte Gerätedeskriptor `dd` (Device Descriptor) wird dann der Funktion `void *mmap(void *addr, size_t length, int prot, int flags, int dd, off_t offset)` übergeben. Ist der Parameter `addr` mit einer Adresse vorbelegt, versucht das Betriebssystem, den Speicher an diese Adresse einzublenden. Typischerweise ist `addr` aber auf null gesetzt und das System selbst sucht eine günstige, freie Adresse aus. Der Parameter `length` gibt die Länge des Speicherbereichs an, der aus Sicht der Peripherie ab dem Offset `offset` eingeblendet werden soll, `prot` spezifiziert den gewünschten Speicherschutz und ist entweder `PROT_NONE`, `PROT_EXEC`, `PROT_READ` oder `PROT_WRITE` (bitweise verknüpft). Das Argument `flags` legt fest, ob Änderungen an dem eingeblendeten Speicherbereich für andere Threads sichtbar sind, die den Bereich ebenfalls eingeblendet haben. Hierzu stehen die vordefinierten Werte `MAP_SHARED` und `MAP_PRIVATE` zur Verfügung.

Beispiel 5-10
Speicherbereiche in den Adressraum einblenden

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

int main( int argc, char **argv, char **envp )
{
   int dd;
   void *pageaddr;
```

```
   int *ptr;

   dd = open( "/dev/mmap_dev", O_RDWR );
   if (dd<0) {
      perror( "/dev/mmap_dev" );
      return -1;
   }
   pageaddr = mmap( NULL, 4096, PROT_READ|PROT_WRITE, MAP_SHARED, dd, 0 );

   printf("pageaddr mmap: %p\n", pageaddr );
   if (pageaddr) {
      ptr = (int *)pageaddr;
      printf("*pageaddr: %d\n", *ptr );
      *ptr = 55; /* access to the mapped area */
   }
   munmap( pageaddr, 4096 );

   return 0;
}
```

Um einen eingeblendeten Speicherbereich wieder freizugeben, ruft die Applikation `int munmap(void *addr, size_t length)` auf.

5.3.2 GPIO-Zugriff über das Sys-Filesystem

Einige Gerätetreiber ermöglichen den Zugriff auf die Hardware über das Sys-Filesystem. Das Sys-Filesystem (sysfs) ist eine virtuelle Verzeichnis- und Dateistruktur. Virtuell bedeutet, dass Verzeichnisse und vor allem der Inhalt von Dateien nicht auf einer Festplatte (beziehungsweise allgemein gesprochen auf einem Hintergrundspeicher) liegen, sondern vom Betriebssystemkern dynamisch erst beim Zugriff erzeugt werden.

Insbesondere das GPIO-Subsystem des Linux-Kernels unterstützt standardmäßig das Sys-Filesystem. Daher wird dieses im Folgenden herangezogen, um Interfaces und Zugriffe vorzustellen.

GPIOs (General Purpose Input Output) sind Ein-/Ausgabeleitungen an denen Hardwarekomponenten, die im einfachsten Fall über Pins mit Schalter, Taster und Leuchtdioden verbunden werden. Diese GPIO-Pins sind häufig flexibel konfigurierbar: Als Ausgabeleitung eingesetzt, liegt programmgesteuert Spannung, typischerweise 3,3 V, oder 5 V, oder eben keine Spannung, also 0 V, an. Als Eingabe konfiguriert, übernimmt der Prozessor nach Aufforderung, falls Spannung anliegt, eine »1«, ansonsten eine »0«.

Auch der Raspberry Pi hat derartige GPIOs. Die 26-polige Steckerleiste des preiswerten Kleincomputers stellt dazu insgesamt 17 von 54 Ein-/Ausgabeleitungen, die sogenannten GPIOs (General Purpose Input Output), zur Verfügung (siehe Abb. 5-2).

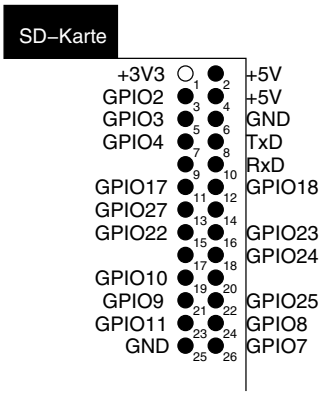


Abb. 5-2
Belegung der Steckerleiste P1 des Raspberry Pi

Falls Sie eine gelbe, grüne oder rote LED, einen Widerstand (um die 300) und Kabel parat haben, können Sie den Zugriff auf GPIOs leicht testen. Dazu bauen Sie wie in Abbildung 5-3 dargestellt eine Schaltung aus der LED und dem Widerstand auf.

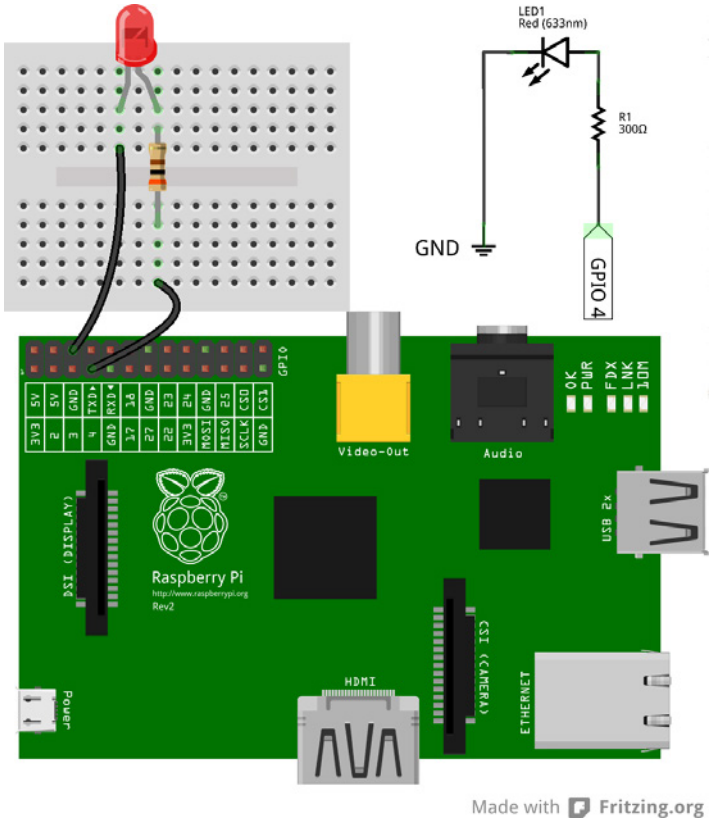


Abb. 5-3
Schaltung zur Ansteuerung einer LED

Beide werden in Reihe geschaltet und mit der Steckerleiste auf dem Raspberry Pi verbunden. Die Seite der Reihenschaltung, an der die Kathode der LED (die Seite mit dem kürzeren Beinchen beziehungsweise mit der abgeflachten Unterkante) liegt, wird mit GND (Pin 6) verbunden; die andere Seite mit GPIO4 (Pin 7). Hilfreich für den Aufbau ist ein Breadboard, in das die Bauteile gesteckt werden können. Bei einem Breadboard sind die Löcher vertikal miteinander jeweils einmal oberhalb und einmal unterhalb verbunden. Hintergrundinformationen zu den Bauteilen und der Schaltung finden Sie übrigens in [Grant2012].

Über das Sys-Filesystem erfolgt anschließend der Test der Schaltung. Um auf eine GPIO-Leitung zugreifen zu können, muss diese im ersten Schritt reserviert und im zweiten Schritt konfiguriert werden. Danach erst können die eigentlichen Zugriffe — entweder ausgeben oder einlesen — durchgeführt werden. Sollte die GPIO-Leitung nicht mehr benötigt werden, wird sie wieder freigegeben (siehe Abb. 6-2).

Die GPIO-Konfiguration findet sich im Verzeichnisbaum unter `/sys/class/gpio/`. Um jetzt eine Leitung, beispielsweise die Leitung GPIO4, benutzen zu können, muss sie reserviert werden. Zur Reservierung wird die Nummer der GPIO-Leitung (aus Sicht der CPU, also nicht die Nummer des Pins an der Doppelstiftleiste) per `echo` in die Datei `/sys/class/gpio/export` geschrieben. Dazu sind allerdings Root-Rechte notwendig. Daraufhin legt der Kernel ein neues Verzeichnis für GPIO4 (`gpio4/`) an. Beispiel 5-11 zeigt die dazu notwendigen Befehle.

Beispiel 5-11 *Kommandos auf dem Raspberry Pi zur Reservierung von GPIO4*

```
# cd /sys/class/gpio/
# ls
export     gpiochip0  unexport
# echo "4" >export
# ls
export     gpio4      gpiochip0  unexport
# cd gpio4
# ls
active_low  direction   edge        power       subsystem   uevent value
```

In dem neuen Verzeichnis `gpio4/` gibt es eine Reihe von Dateien, unter anderem auch `direction`, über die mit den Schlüsselwörtern »out« und »in« konfiguriert wird, ob es sich um eine Ausgabe- oder eine Eingabeleitung handelt. Mit einem `echo "1" >/sys/class/gpio/gpio4/value` wird schließlich die Spannung auf 3,3 V gesetzt. Damit leuchtet die LED. Eine »0« in die gleiche Datei geschrieben, stellt eine Spannung von 0 V ein und schaltet die LED wieder aus. In Beispiel 5-12 finden Sie ein Skript, das aus der LED ein Blinklicht macht. Dazu wird in einer Schleife die LED eingeschaltet, eine Sekunde geschlafen, ausgeschaltet und nochmals eine Sekunde geschlafen. Der Befehl `trap` dient übrigens dazu, beim

Abbrechen des Skripts beispielsweise per [Strg][c] den Pin wieder freizugeben.

Beispiel 5-12
Skriptgesteuerter Zugriff auf GPIOs

```
#!/bin/bash

trap "echo \"4\" >/sys/class/gpio/unexport" EXIT

echo "4" >/sys/class/gpio/export
echo "out" >/sys/class/gpio/gpio4/direction

while true
do
    echo "1" >/sys/class/gpio/gpio4/value
    sleep 1
    echo "0" >/sys/class/gpio/gpio4/value
    sleep 1
done
```

6 Gerätetreiber selbst gemacht

Im Bereich eingebetteter Systeme ist es üblich, Sensoren und Aktoren mit dem eigentlichen Rechnerboard, beispielsweise dem Raspberry Pi, über GPIOs, I²C, SPI oder USB zu verbinden. Der Zugriff auf diese Hardware kann zwar wie beschrieben aus dem Userland erfolgen, was aber zwei Nachteile mit sich bringt. Erstens ist der Zugriff sehr langsam und zweitens beeinträchtigt die Lösung die Betriebssicherheit (Safety). Schließlich ist eine der Aufgaben und Philosophien eines Betriebssystems, Hardwarezugriffe nur im Kernel zuzulassen. In den meisten Fällen ist es also sinnvoll, einen Gerätetreiber als Erweiterung des Linux-Kernels zu schreiben.

Ein Grund, warum häufig die Userland-Lösung einem Gerätetreiber vorgezogen wird, besteht in fehlendem Know-how bezüglich Kernelprogrammierung und dem dafür erwarteten Aufwand. Dabei sind einfache Treiber, die zunächst keine besonderen Funktionalitäten aufweisen, leicht zu erstellen. Allerdings zeigt die Erfahrung, dass die Hardwarezugriffe immer komplexer werden und damit auch der Code des Gerätetreibers. Für eine professionelle Entwicklung eines zuverlässigen und sicheren Systems geht aber letztlich kein Weg an einem Gerätetreiber vorbei. Im Folgenden finden Sie eine Einführung in die Gerätetreiberprogrammierung, mehr Details und Hintergrundinfos finden Sie in [QuKu2011b].

6.1 Einführung in die Treiberprogrammierung

Um einen einfachen Gerätetreiber zu schreiben, der ein virtuelles Gerät bedient, sind die folgenden Schritte notwendig:

1. Sicherstellen, dass der passende Kernelquellcode installiert und konfiguriert ist.
2. Verzeichnis für den Treiberquellcode anlegen (hier `~/embedded/driver/hello/`).
3. In das Verzeichnis wechseln und den Quellcode erstellen. Hierzu gehören insbesondere die wesentlichen Treiberfunktionen wie `mod_init()`, `mod_exit()`, `driver_read()` und `driver_write()`. Außerdem ist die Datenstruktur `struct file_operations` wichtig, die die Adressen der Treiberfunktionen aufnimmt.
4. Im Quellcodeverzeichnis des Gerätetreibers ein Makefile erstellen.
5. Den Treiber durch Aufruf von `make` generieren.
   Soll der Treiber auf dem Entwicklungsrechner für das Target (Raspberry Pi) generiert werden, ist Make mit gesetzten Umgebungsvariablen `KDIR`, `CROSS_COMPILE` und `ARCH` zu starten.
6. Falls es sich um eine Cross-Entwicklung handelt, wird der generierte Treiber auf das Target übertragen (beispielsweise per `ssh`).
7. Der Treiber wird per `insmod` in den Kernel geladen. Das per Buildroot generierte System sollte die Programme `insmod`, `rmmod`, `lsmod` und `modprobe` generiert haben. Ansonsten sind diese in der Busybox-Konfiguration unter [Linux Module Utilities] auszuwählen und das System neu zu generieren.
8. Auf dem Target kann es notwendig sein, per `mknod` eine Gerätedatei anzulegen.
9. Der Treiber kann getestet werden. Vielfach ist das per `cat` und `echo` möglich.

Ein Gerätetreiber ist ein Satz von Funktionen, der den Zugriff auf die Hardware steuert. Die Hardware muss dabei nicht immer physisch existent sein, ein Treiber kann dem Anwender durchaus auch eine virtuelle Hardware bereitstellen. Gerätetreiber sind für den eigentlichen Zugriff auf Hardware notwendig.

Um einen Gerätetreiber schreiben zu können, werden vorkonfigurierte beziehungsweise kompilierte Kernelquellen benötigt. Dieser Vorgang ist in Abschnitt 3.3.1 beschrieben.

6.1.1 Grundprinzip

Ein Gerätetreiber funktioniert vereinfacht folgendermaßen: Die Applikation greift über die Systemcalls `open()`, `close()`, `read()` und `write()` auf Hardware zu (Abschnitt 5.3.1). Werden diese Funktionen aufgerufen, wird der Kernel aktiv. Er sorgt dafür, dass zum Systemcall eine korrespondierende Funktion im Kernel aufgerufen wird, die vom Treiber zur Verfügung gestellt wird und die Parameter aus der Applikation übergeben bekommt. Damit benötigt ein Gerätetreiber diese korrespondierenden Funktionen, die im Folgenden `driver_open()`, `driver_close()`, `driver_read()` und `driver_write()` genannt werden.

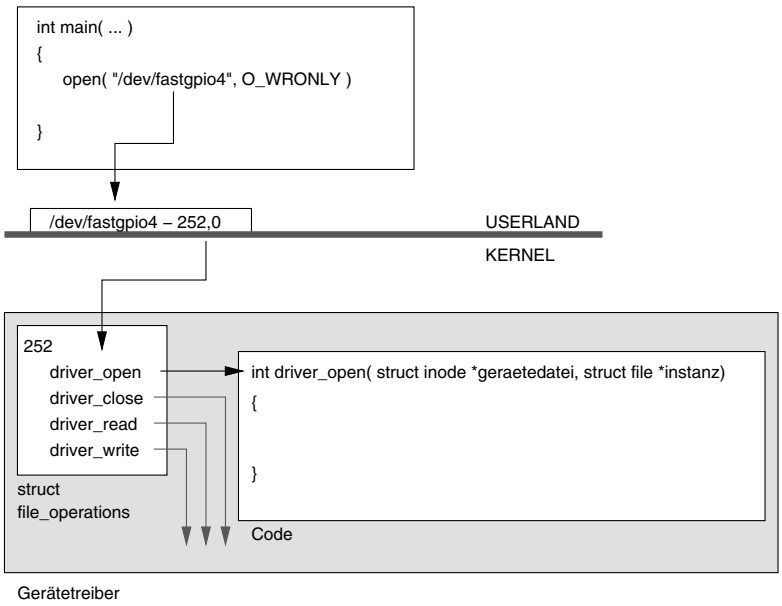


Abb. 6-1
Die Gerätedatei verbindet die Applikation mit dem Treiber.

Da jeder Gerätetreiber diese Funktionen, oder zumindest einen Teil davon, implementiert, muss der Kernel wissen, unter welchen Umständen genau dieser Satz an Funktionen verwendet werden soll. Das ist über die Gerätedatei realisiert, die über eine Gerätenummer die Beziehung zu dem Treiber herstellt. Jeder Treiber bekommt nämlich vom Kernel für jedes Gerät, auf das er zugreift, eine Nummer zugewiesen, die Gerätenummer. Die Gerätenummer wiederum ist mit einer Gerätedatei verbunden, die im Dateisystem unter einem beliebigen Namen angelegt werden kann. Diese Gerätedatei schließlich muss vom Applikationsprogrammierer beim Systemcall `open()` als erster Parameter mit angegeben werden (siehe Abb. 6-1). Mit einem erfolgreichem `open()` legt der Kernel ein internes Objekt an, das wiederum mit der Gerätenummer verbunden ist und durch die Applikation über den Filedeskriptor referenziert wird.

Damit ist über die Gerätedatei und die Gerätenummer die Verbindung zwischen dem `open()` und `driver_open()` geschlossen und über den daraus resultierenden Filedeskriptor auch die Verbindung zwischen den übrigen Funktionen.

Jeder Treiber implementiert noch zwei weitere Funktionen. Die eine (`mod_init()`) wird vom Kernel beim Laden des Treibers aufgerufen, die andere (`mod_exit()`) beim Entladen. `mod_init()` ist notwendig, um sich beim Kernel als Treiber anzumelden. Typischerweise übergibt er dabei dem Kernel Kennungen, die die Hardware kennzeichnen, für die der Treiber verantwortlich ist. Im Fall eines USB-Gerätes sind das beispielsweise die USB-Vendor- und die USB-Geräte-ID. Zusammen mit der Kennung wird eine Funktion übergeben (`driver_probe()`), die der Kernel aufruft, sobald er ein vom Treiber unterstütztes Gerät identifiziert. Innerhalb der Funktion `driver_probe()` lässt sich der Treiber eine Gerätenummer zuweisen und die Gerätedatei anlegen.

Manche Geräte besitzen allerdings keine eindeutige Kennung, so beispielsweise GPIO-Leitungen oder darüber angeschlossene I^2C-Bausteine. In diesem Fall lässt sich der Treiber innerhalb der Funktion `mod_init()` Gerätenummern zuweisen, verknüpft damit die Zugriffsfunktionen `driver_open()`, `driver_close()`, `driver_read()` und `driver_write()` und lässt die zugehörigen Gerätedateien anlegen.

6.1.2 Aufbau eines Gerätetreibers

Nachdem geklärt ist, wie ein Gerätetreiber grundsätzlich arbeitet und dass er im einfachen Fall aus rund sechs Funktionen besteht (wobei nicht jede der genannten Funktionen unbedingt benötigt wird), soll es an die Realisierung eines einfachen Hello-World-Treibers gehen. Dieser Treiber bedient ein virtuelles Gerät, das beim lesenden Zugriff den String »Hello World« zurückgibt. Die zugehörige Gerätedatei soll »/dev/hello« heißen.

Von den vier genannten Zugriffsfunktionen implementiert Beispiel 6-1 nur `driver_read()`. Dies ist möglich, weil das Schreiben auf den Treiber (`driver_write()`) zunächst nicht unterstützt wird, das Öffnen (`driver_open()`) und das Schließen (`driver_close()`) keine weitere Funktionalität benötigen. In einem solchen Fall müssen die Funktionen nicht geschrieben werden.

Die Lesefunktion ist aus didaktischen Gründen auf ein Minimum reduziert. Aufgabe der Funktion ist es, die Daten aus der (virtuellen) Hardware auszulesen und an die Adresse zu kopieren, die die Applikation beim Aufruf des Systemcalls `write()` bereitgestellt hat. Diese Adresse wird der Funktion `static ssize_t driver_write(struct file *instanz, char __user *user, size_t count, loff_t *offset)` als zweiter Parameter (`user`)

übergeben. Der dritte Parameter (`count`) enthält die maximale Größe des Speicherbereichs. Damit nicht mehr Daten an die übergebene Adresse geschrieben werden, als dort maximal zur Verfügung stehen, und nicht mehr, als im Kernel überhaupt vorhanden sind, wird als Erstes über die Minimum-Funktion (`min()`) die maximal zu transferierende Anzahl Bytes festgelegt.

Der Schritt, die Daten aus der Hardware auszulesen, kann bei diesem Treiber entfallen, da dies ein virtueller Treiber (ohne physische Hardware) ist. Der Datentransfer der Daten in den Speicherbereich der Applikation, den Userspace, muss über die Funktion `copy_to_user()` abgewickelt werden. Das direkte Kopieren ist nämlich nicht möglich, da die Speicherbereiche über die Memory Management Unit verwaltet werden (siehe Abschnitt "Kernel", Seite 16). Die Funktion `copy_to_user()` und ihr Pendant für die Gegenrichtung `copy_from_user()` haben die Eigenart, die Anzahl Bytes zurückzuliefern, die *nicht* kopiert werden konnten. Das kommt vor, wenn eine Applikation eine ungültige Adressangabe macht, beispielsweise als Adresse null übergibt. Die Funktion `driver_read()` gibt schließlich die Anzahl Bytes zurück, die tatsächlich in den Userspace kopiert wurden. Falls ein Fehler aufgetreten ist, gibt die Funktion »0« zurück.

Für die Verknüpfung zwischen Applikation und Treiber werden noch die Gerätedatei und die Gerätenummer benötigt. Diese Managementfunktionalität ist in Beispiel 6-1 in der Funktion `mod_init()` untergebracht.

Dabei wird per `alloc_chrdev_region()` vom Kernel eine einzelne Gerätenummer angefordert. Diese wird zusammen mit einem Treiberobjekt, das die Liste der Adressen der Treiberfunktionen (`struct file_operations`) `driver_open()`, `driver_read()`, `driver_write()` und `driver_close()` enthält, dem Kernel übergeben.

Jetzt fehlt noch die Gerätedatei selbst. Auf Desktop- und Serversystemen gibt es dazu einen Mechanismus: Falls der Treiber im Sys-Filesystem einen bestimmten Eintrag erzeugt, legt eine Task im Userland (`udevd`) die Gerätedatei an. Um diesen Eintrag im Sys-Filesystem zu erzeugen, sind zwei Schritte notwendig. Als Erstes wird ein Verzeichnis durch das Anlegen eines Klassenobjektes über `class_create()` erstellt und als Zweites wird unterhalb des Verzeichnisses eine Datei mit der Gerätenummer als Inhalt durch Aufruf von `device_create()` erzeugt.

Im Bereich eingebetteter Systeme wird häufig kein `udevd` eingesetzt. Hier werden die Gerätedateien statisch angelegt. Dazu ist es sinnvoll, die Gerätenummer nicht zuweisen zu lassen, sondern auszuwählen, damit sie im Vorhinein (also beim Anlegen der Gerätedatei) bereits bekannt ist. Hierzu dient die Funktion `register_chrdev_region( dev_t from, unsigned count, char *name)`. Soll beispielsweise die Gerätenummer, die

aus der Majornummer 248 und der Minornummer 0 gebildet wird, verwendet werden, beginnt in diesem Fall die Funktion `mod_init()` folgendermaßen:

```
static int __init mod_init( void )
{
   if ( register_chrdev_region(MKDEV(248,0),1,"Hello")<0 )
      return -EIO;
   driver_object = cdev_alloc(); /* Anmeldeobjekt reserv. */
   ...
```

Beispiel 6-1 *Hello-World-Treiber <hellodriver.c>*

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <asm/uaccess.h>

static char hello_world[]="Hello World\n";

static dev_t hello_dev_number;
static struct cdev *driver_object;
static struct class *hello_class;
static struct device *hello_dev;

static ssize_t driver_read(struct file *instanz,char __user *user,
    size_t count, loff_t *offset )
{
    unsigned long not_copied, to_copy;

    to_copy = min( count, strlen(hello_world)+1 );
    not_copied=copy_to_user(user,hello_world,to_copy);
    return to_copy-not_copied;
}

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

static int __init mod_init( void )
{
    if( alloc_chrdev_region(&hello_dev_number,0,1,"Hello")<0 )          ❶
        return -EIO;
    driver_object = cdev_alloc(); /* Anmeldeobjekt reserv. */
    if( driver_object==NULL )
        goto free_device_number;
    driver_object->owner = THIS_MODULE;
    driver_object->ops = &fops;                                        ❷
    if( cdev_add(driver_object,hello_dev_number,1) )                   ❸
        goto free_cdev;
```

```
    /* Eintrag im Sysfs, damit Udev die Geraetedatei erzeugt */
    hello_class = class_create( THIS_MODULE, "Hello" );                ❹
    if( IS_ERR( hello_class ) ) {
        pr_err( "hello: no udev support\n");
        goto free_cdev;
    }
    hello_dev = device_create(hello_class,NULL,hello_dev_number,       ❺
        NULL, "%s", "hello" );
    dev_info( hello_dev, "mod_init called\n" );
    return 0;
free_cdev:
    kobject_put( &driver_object->kobj );
free_device_number:
    unregister_chrdev_region( hello_dev_number, 1 );
    return -EIO;
}

static void __exit mod_exit( void )
{
    dev_info( hello_dev, "mod_exit called\n" );
    /* Loeschen des Sysfs-Eintrags und damit der Geraetedatei */
    device_destroy( hello_class, hello_dev_number );
    class_destroy( hello_class );
    /* Abmelden des Treibers */
    cdev_del( driver_object );
    unregister_chrdev_region( hello_dev_number, 1 );
    return;
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

❶ Die Gerätenummer wird mit den Zugriffsfunktionen des Treibers verknüpft.
❷ Der Treiber lässt sich vom Kernel eine Gerätenummer zuweisen.
❸ Der Treiber wird mit dieser Funktion beim Kernel angemeldet.
❹ Die Funktion erzeugt einen Eintrag im Sys-Filesystem.
❺ Über diesen Aufruf wird ein Eintrag im Sys-Filesystem generiert, über den daraufhin die Gerätedatei erzeugt wird.

6.1.3 Generierung des Gerätetreibers

Das Generieren eines Gerätetreibers erfolgt durch das Kernel-Build-System, das auf `make` beruht. Darin klinkt man sich mithilfe eines speziellen Makefiles ein. Der Treiber wird dabei passend zum Kernel generiert, sodass eine Kernelkonfiguration und der (vorkompilierte) Kernelquellcode notwendig sind.

Beispiel 6-2 *Zum Hello-World-Treiber gehörendes Makefile <Makefile>*

```
ifneq ($(KERNELRELEASE),)
obj-m   := hellodriver.o

else
KDIR   := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)

default:
   $(MAKE)   -C $(KDIR) M=$(PWD) modules
endif

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

Sie sollten den Hello-World-Treiber zunächst auf Ihrem Hostsystem generieren und testen. Erstellen Sie dazu ein neues Verzeichnis mit dem Namen `~/embedded/driver/hello/` und kopieren Sie dorthin sowohl den Quellcode (Beispiel 6-1) als auch das zugehörige Makefile (Beispiel 6-2). Rufen Sie danach `make` auf. Zum Laden des Treibers gibt es das Kommando `insmod`. Konnte der Treiber erfolgreich geladen werden, sollte auf einem Hostsystem die Gerätedatei `/dev/hello` erzeugt worden sein, auf die allerdings nur der Superuser Zugriff hat. Sie können also entweder die Rechte auf die Gerätedatei ändern (`quade@felicia:~$ sudo chmod 666 /dev/hello`) oder als Superuser darauf zugreifen. Die Befehle finden Sie unter Beispiel 6-3.

Beispiel 6-3 *Generieren und Laden des Hello-World-Treibers (Hostsystem)*

```
quade@felicia:~/embedded> mkdir -p driver/hello
quade@felicia:~/embedded> cd driver/hello
quade@felicia:~/embedded/driver/hello> gedit hellodriver.c  &
quade@felicia:~/embedded/driver/hello> gedit Makefile &
quade@felicia:~/embedded/driver/hello> make
make   -C /lib/modules/3.2.0-52-generic/build M=/home/quade/embedded/
driver/hello modules
make[1]: Betrete Verzeichnis '/usr/src/linux-headers-3.2.0-52-generic'
  CC [M]  /home/quade/embedded/driver/hello/hellodriver.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/quade/embedded/driver/hello/hellodriver.mod.o
  LD [M]  /home/quade/embedded/driver/hello/hellodriver.ko
make[1]: Verlasse Verzeichnis '/usr/src/linux-headers-3.2.0-52-generic'
quade@felicia:~/embedded/driver/hello>sudo insmod hellodriver.ko
quade@felicia:~/embedded/driver/hello>sudo cat /dev/hello
```

Gerätetreiber für das eingebettete System

Damit der Treiber auf dem eingebetteten System funktioniert, muss er wieder cross-kompiliert werden. Dazu ist die Pfadvariable so zu setzen, dass der Cross-Compiler vom Kernel-Build-System gefunden werden kann. Daneben müssen noch drei weitere Variablen beim Aufruf von `make` gesetzt werden:

1. `ARCH`
2. `CROSS_COMPILE`
3. `KDIR`

```
quade@felicia:~/embedded/driver/hello> \
   make ARCH=arm \
   CROSS_COMPILE=arm-linux- \
   KDIR=~/embedded/raspi/linux/
```

Die Variable `KDIR` zeigt dabei auf das Quellcodeverzeichnis des Linux-Kernels. Im obigen Beispiel wird der Treiber für das Selbstbausystem aus Kapitel 3 generiert. In Beispiel 6-4 können Sie den Aufruf für das Buildroot-Target-System aus Kapitel 4 sehen.

Ist der Gerätetreiber generiert, wird er auf das Target per `scp`, `netcat`, `nfs` oder SD-Karte kopiert (siehe Abschnitt 5.1) und dort per `insmod` geladen. Zum Entladen eines Treibers dient das Kommando `rmmod`. Um die geladenen Gerätetreiber, die technisch jeweils ein Kernelmodul darstellen, anzuzeigen, geben Sie im Terminal `lsmod` ein. Diese Kommandos müssen natürlich im Selbstbausystem über die Busybox vorhanden sein.

Beispiel 6-4
Generieren und Laden des Hello-World-Treibers (Buildroot-Target)

Die folgenden Kommandos sind auf dem Entwicklungsrechner einzugeben. Anmerkung: Der Name des Kernelquellcode-Verzeichnisses (`linux-e959a8e`) lautet in Ihrem Fall möglicherweise anders. Passen Sie ihn entsprechend an.

```
quade@felicia:~/embedded/driver/hello> make ARCH=arm \
    CROSS_COMPILE=arm-linux- \
    KDIR=~/embedded/raspi/buildroot-2013.05/output/build/linux-e959a8e/
...
quade@felicia:~/embedded/driver/hello> scp hellodriver.ko \
    root@<ipadresse-target>:
quade@felicia:~/embedded/driver/hello> ssh root@<ipadresse-target>
...
```

Wenn Sie wie beschrieben z.B. per `ssh` oder über die serielle Schnittstelle auf dem Target angemeldet sind, wechseln Sie ins Heimatverzeichnis, laden den Treiber, identifizieren die Gerätenummer (Aufruf von `cat /proc/devices`), legen per `mknod` die Gerätedatei (erfolgt auf dem Buildroot-System nicht automatisch) an und können danach auf das neue Gerät zugreifen:

```
# cd
# insmod hellodriver.ko
# cat /proc/devices | grep Hello
248 Hello
# mknod /dev/hello c 248 0
# cat /dev/hello
Hello World
Hello World
...
```

Falls Sie ein Raspbian einsetzen, können Sie den Treiber direkt auf dem Raspberry Pi entwickeln. Allerdings benötigen Sie dazu auf dem Raspbian vorkompilierte Kernelquellen. Beispiel 3-9 zeigt, wie Sie den Raspberry Pi unter Raspbian vorbereiten. Ist der Kernel kompiliert, können Sie den Quellcode und das Makefile auf den Raspberry Pi kopieren und `make` aufrufen. Das Setzen der Umgebungsvariablen ist nicht notwendig.

6.2 Schneller GPIO-Treiberzugriff

Ein Treiber für den Zugriff auf die GPIO-Leitungen des Raspberry Pi soll als Beispiel für Hardwarezugriffe im Kernel dienen. Dieser GPIO-Treiber ist erheblich performanter als der Zugriff über das Sys-Filesystem (Abschnitt 5.3.2).

Dabei muss als Erstes geklärt werden, wie der Anwender auf den Treiber beziehungsweise auf die GPIOs zugreift. Um das Beispiel einfach zu halten, soll der Treiber für die Ausgabe nur GPIO4 bedienen: Wird eine »0« auf die zugehörige Gerätedatei mit dem Namen `/dev/fastgpio` geschrieben, soll am Ausgang »0« anliegen, wird etwas anderes auf die Gerätedatei geschrieben, eine »1«. Für die Eingabe wird GPIO7 verwendet. Das Lesen über die Gerätedatei `/dev/fastgpio` liefert je nach anliegender Spannung eine »0« oder eine »1«.

6.2.1 Digitale Ausgabe

Um einen Gerätetreiber für die digitale Ausgabe per GPIO auf dem Raspberry Pi zu implementieren, sind die folgenden Schritte notwendig:

1. Auf dem Entwicklungsrechner ein neues Verzeichnis (`driver/fastgpio/`) anlegen und einen Treiberquellcode als Template reinkopieren
2. Den Treiberquellcode anpassen
3. Makefile kopieren und anpassen oder neu erstellen
4. Den Treiber mithilfe der Cross-Entwicklungswerkzeuge generieren
5. Eine Applikation vorbereiten und ebenfalls mithilfe der Cross-Entwicklungswerkzeuge generieren
6. Treiber und Applikation auf den Raspberry Pi kopieren
7. Auf dem Raspberry Pi den Treiber laden
8. Auf dem Raspberry Pi eine Gerätedatei anlegen
9. Auf dem Raspberry Pi die Testapplikation starten

Als Ausgangsbasis für den neuen Treiber kann sehr gut der vorgestellte Hello-World-Treiber dienen. Dazu wird der vorhandene Quellcode in ein neu anzulegendes Verzeichnis unter dem neuen Namen `fastgpio.c` kopiert. Im Quellcode selbst müssen dann die folgenden Änderungen beziehungsweise Ergänzungen vorgenommen werden:

- Namen und Variablen von `hello` in `fastgpio` ändern.
- Funktion `driver_open()` implementieren, in der der GPIO4 reserviert und konfiguriert wird.
- Funktion `driver_close()` implementieren, in der der GPIO4 wieder freigegeben wird.
- Zugriffe auf Hardware implementieren (`driver_write()`).
- Funktion `driver_read()` löschen.
- Funktion `mod_init()` so anpassen, dass die feste Gerätenummer MKDEV(248,0) verwendet wird.

In der Datei `fastgpio.c` selbst wird also als Erstes der String »hello« durch »fastgpio« ersetzt. Da die globale Variable `hello_world[]` nicht benötigt wird, kann diese gelöscht werden. Außerdem ist es sinnvoll, den Treiber auf eine feste Gerätenummer, hier über das Makro `MKDEV()` aus der Majornummer 248 und der Minornummer 0 gebildet, einzustellen. Schließlich wird der Treiber im eingebetteten System eingesetzt, das das dynamische Anlegen von Gerätedateien per udevd nicht unterstützt.

Hardwaremanagement

Bevor Ausgaben auf GPIO4 getätigt werden können, muss die Leitung reserviert und konfiguriert werden. Diese Aufgaben werden sinnvollerweise in der Treiberfunktion `driver_open()` implementiert. Glücklicherweise stellt Linux für das Reservieren und Konfigurieren von GPIOs eine Reihe von Funktionen zur Verfügung, die neben anderen in Tabelle 6-1 aufgeführt sind. Ein Vorteil dieser Funktionen besteht darin, dass sie dem Programmierer ansonsten notwendige Bitschiebereien abnehmen.

Tabelle 6-1 *Kernelinterne Funktionen für den GPIO-Zugriff (Auswahl)*

Prototyp	Bedeutung
Managementfunktionen	
int gpio_request(unsigned gpio, const char *label);	GPIO mit der Nummer »gpio« wird unter dem Namen »label« reserviert.
void gpio_free(unsigned gpio);	Der reservierte GPIO-Pin »gpio« wird freigegeben.
int gpio_direction_input(unsigned gpio);	Der Pin »gpio« wird auf Eingabe konfiguriert.
int gpio_direction_output(unsigned gpio, int value);	Der Pin »gpio« wird auf Ausgabe konfiguriert.
int gpio_set_debounce(unsigned gpio, unsigned debounce);	Setzt die Entprell-Zeit (in Mikrosekunden), Zeit, in der der Pin nach einem Zustandswechsel den gleichen Wert haben muss.
int gpio_cansleep(unsigned gpio);	Die Funktion gibt einen Wert ungleich null zurück, falls beim Zugriff eventuell geschlafen werden könnte (GPIO-Zugriff über Bussysteme). In diesem Fall darf kein Zugriff aus einem Interrupt-Kontext heraus erfolgen.
Zugriffsfunktionen	
inline bool gpio_is_valid(int number);	Überprüft, ob GPIO »number« existiert.
int gpio_get_value(unsigned gpio);	GPIO-Pin »gpio« wird eingelesen.
void gpio_set_value(unsigned gpio, int value);	Falls value = 0 ist, wird »gpio« auf 0 V gesetzt, andernfalls auf 3,3 V
int gpio_get_value_cansleep(unsigned gpio);	GPIO-Pin wird aus einem Nicht-Interrupt-Kontext heraus eingelesen.
void gpio_set_value_cansleep(unsigned gpio, int value);	GPIO-Pin wird aus einem Nicht-Interrupt-Kontext heraus gesetzt.
Integrationsfunktionen	
int gpio_export(unsigned gpio, bool direction_may_change);	Reservierung des GPIO »gpio« wird für den Zugriff über das Sys-Filesystem freigegeben.
void gpio_unexport(unsigned gpio);	Zugriff über das Sys-Filesystem wird aufgehoben.

➔

Prototyp	Bedeutung
extern int gpio_export_link(struct device *dev, const char *name, unsigned gpio);	Funktion dient dazu, einen symbolischen Link zwischen Treiber und einem im Sys-Filesystem exportierten Link herzustellen.
extern int gpio_sysfs_set_active_low(unsigned gpio, int value);	Markiert GPIO »gpio« als Active-Low-Signal.

Von diesen Funktionen wird in `driver_open()` die Routine `gpio_request()` benötigt. Diese reserviert die GPIO-Leitung. Mittels `gpio_direction_output()` wird sie auf Ausgabe geschaltet. Damit der Compiler die Funktionen kennt, muss noch die Headerdatei `linux/gpio.h` inkludiert werden. Nicht vergessen: Die Adresse der Funktion muss in der Datenstruktur `struct file_operations` durch Ergänzung der Zeile ».open=driver_open,« eingetragen werden.

Grundsätzlich muss alles, was irgendwann reserviert wird, auch wieder freigegeben werden. Das geschieht in der Funktion `driver_close`, deren Adresse durch Eintrag von ».release=driver_close,« in der `struct file_operations` dem Kernel später ebenfalls bekannt gegeben wird. Die Funktion `driver_close()` ruft dann ihrerseits `gpio_free()` auf.

Hardwarezugriff

Damit sind die Managementaufgaben abgeschlossen und es muss nur noch der eigentliche Hardwarezugriff implementiert werden. Um eine GPIO-Leitung zu setzen, wird `gpio_set_value()` verwendet. Da wir eine Schreib- und keine Lesefunktion (wie im Hello-World-Treiber) benötigen, muss `driver_read()` zuvor in `driver_write()` umbenannt werden, ebenso der Eintrag in der `struct file_operations`. Innerhalb der Funktion wird dann überprüft, ob die Leitung auf »1« oder »0« gesetzt werden soll. Dazu wird der von der Applikation der Funktion `write()` übergebene Parameter per `copy_from_user()` ausgelesen und ausgewertet. Das Auswerten und Setzen kann in einer Zeile Code codiert werden (`gpio_set_value( 4, value?1:0)`), die allerdings etwas schwerer zu interpretieren ist als ein vollständiges if-Konstrukt. Den kompletten Quellcode finden Sie in Beispiel 6-6.

Generierung und Test

Mit diesen Änderungen kann der fastgpio-Gerätetreiber generiert und getestet werden. Als Erstes ist dazu das Makefile anzupassen und in der mit »obj-m« beginnenden Zeile »fastgpio.ko« einzutragen. Um das Modul zu testen, wird es entweder auf einem Raspbian wie in Abschnitt 6.1.3 beschrieben kompiliert oder alternativ auf dem Entwicklungsrechner cross-kompiliert. Beispiel 6-5 zeigt die Befehlssequenz, um den Ge-

rätetreiber für den GPIO-Zugriff zu generieren und eine Testapplikation zu erstellen, auf das Target zu kopieren und auf dem Target schließlich auszutesten. Dabei wird angenommen, dass der Raspberry Pi die IP-Adresse 192.168.178.31 besitzt. Denken Sie daran, dass die Umgebungsvariable PATH den Pfad zu den Cross-Entwicklungswerkzeugen beinhalten muss.

Beispiel 6-5 *Befehle zum Erstellen des GPIO-Treibers für das Buildroot-System*

```
quade@felicia:~/embedded> mkdir -p driver/fastgpio
quade@felicia:~/embedded> cd driver/fastgpio
quade@felicia:~/embedded/driver/fastgpio> cp ../hello/hellodriver.c \
   fastgpio.c
quade@felicia:~/embedded/driver/fastgpio> cp ../hello/Makefile .
quade@felicia:~/embedded/driver/fastgpio> gedit fastgpio.c &
# Änderungen und Erweiterungen vornehmen
quade@felicia:~/embedded/driver/fastgpio> gedit Makefile &
# hellodriver gegen fastgpio austauschen
quade@felicia:~/embedded/driver/fastgpio> gedit gpioappl.c &
# Quellcode für die Testapplikation erstellen
quade@felicia:~/embedded/driver/fastgpio> make gpioappl ARCH=arm \
   CC=arm-linux-cc LDLIBS=-lrt CFLAGS=-Wall
quade@felicia:~/embedded/driver/fastgpio> make ARCH=arm \
   CROSS_COMPILE=arm-linux- \
   KDIR=~/embedded/raspi/buildroot-2013.05/output/build/linux-e959a8e/
make  -C /home/quade/embedded/raspi/buildroot-2013.05/output/build/linux-
e959a8e/ M=/home/quade/embedded/driver/fastgpio modules
...
quade@felicia:~/embedded/driver/fastgpio>scp fastgpio.ko \
   root@192.168.178.31:
   root@192.168.178.31's password:
   fastgpio.ko                                   100% 6861     6.7KB/s   00:00
```

Auf dem Raspberry Pi sind die folgenden Befehle auszuführen:

```
# cd
# insmod fastgpio.ko
# mknod /dev/fastgpio c 248 0
```

Beispiel 6-6 *Kerneltreiber für den Zugriff auf GPIOs*

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/gpio.h>
#include <asm/uaccess.h>

static dev_t fastgpio_dev_number = MKDEV(248,0);
static struct cdev *driver_object;
static struct class *fastgpio_class;
static struct device *fastgpio_dev;
```

```
static int driver_open( struct inode *geraete_datei, struct file *instanz )
{
   int err;

   printk("driver_open\n");
   err = gpio_request( 4, "rpi-gpio-4" );
   if (err) {
      printk("gpio_request failed %d\n", err);
      return -1;
   }
   err = gpio_direction_output( 4, 0 );
   if (err) {
      printk("gpio_direction_output failed %d\n", err);
      gpio_free( 4 );
      return -1;
   }
   printk("gpio 4 successfull configured\n");
   return 0;
}


static int driver_close( struct inode *geraete_datei, struct file *instanz )
{
   printk( "driver_close called\n" );
   gpio_free( 4 );
   return 0;
}

static ssize_t driver_write(struct file *instanz,const char __user *user,
   size_t count, loff_t *offset )
{
   unsigned long not_copied, to_copy;
   u32 value=0;

   to_copy = min( count, sizeof(value) );
   not_copied=copy_from_user(&value, user, to_copy);
   printk("driver_write( %d )\n", value );
   if (value==0)
      gpio_set_value(4,0);
   else
      gpio_set_value(4,1);
   //gpio_set_value( 4, value?1:0 );
   return to_copy-not_copied;
}

static struct file_operations fops = {
    .owner  = THIS_MODULE,
    .open   = driver_open,
    .release= driver_close,
    .write   = driver_write,
};
```

```
static int __init mod_init( void )
{
   if( register_chrdev_region(fastgpio_dev_number,1,"fastgpio")<0 ) {
      printk("devicenumber (248,0) in use!\n");
      return -EIO;
   }
   driver_object = cdev_alloc(); /* Anmeldeobjekt reserv. */
   if( driver_object==NULL )
      goto free_device_number;
   driver_object->owner = THIS_MODULE;
   driver_object->ops = &fops;
   if( cdev_add(driver_object,fastgpio_dev_number,1) )
      goto free_cdev;
   fastgpio_class = class_create( THIS_MODULE, "fastgpio" );
   if( IS_ERR( fastgpio_class ) ) {
      pr_err( "fastgpio: no udev support\n");
      goto free_cdev;
   }
   fastgpio_dev = device_create(fastgpio_class,NULL,fastgpio_dev_number,
      NULL, "%s", "fastgpio" );
   dev_info( fastgpio_dev, "mod_init called\n" );
   return 0;
free_cdev:
   kobject_put( &driver_object->kobj );
free_device_number:
   unregister_chrdev_region( fastgpio_dev_number, 1 );
   printk("mod_init failed\n");
   return -EIO;
}

static void __exit mod_exit( void )
{
    dev_info( fastgpio_dev, "mod_exit called\n" );
    device_destroy( fastgpio_class, fastgpio_dev_number );
    class_destroy( fastgpio_class );
    cdev_del( driver_object );
    unregister_chrdev_region( fastgpio_dev_number, 1 );
    return;
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

Sollten Sie auf dem Raspberry Pi das Betriebssystem Raspbian verwenden, gibt es nach dem Laden automatisch die Gerätedatei `/dev/fastgpio`. Für unsere Selbstbausysteme muss die Gerätedatei aber mithilfe des Kommandos `mknod` erst angelegt werden.

Können Sie den Treiber nicht laden, ist möglicherweise die Gerätenummer bereits vergeben. Prüfen Sie durch Eingabe von `cat /proc/devices`, ob die Nummer 248 noch frei ist. Wenn diese bereits von einem anderen Treiber verwendet wird, wählen Sie für den Treiber und entsprechend für das Kommando `mknod` eine freie Nummer.

Für einen effizienten Zugriff bietet sich eine eigene Applikation an, wie sie in Beispiel 6-7 zu sehen ist.

Beispiel 6-7 *Blinklicht-Applikation für den Raspberry Pi <gpioappl.c>*

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>

int main( int argc, char **argv, char **envp )
{
   int fd_gpio;
   int value;
   struct timespec sleeptime;

   fd_gpio=open("/dev/fastgpio4", O_WRONLY);
   if (fd_gpio<0) {
      printf("kann /dev/fastgpio4 nicht oeffnen.\n");
      return -1;
   }

   sleeptime.tv_sec = 0;
   sleeptime.tv_nsec= 500000000;
   while (1) {
      value = 1;
      write( fd_gpio, &value, sizeof(value) );
      clock_nanosleep( CLOCK_MONOTONIC, 0, &sleeptime, NULL );
      value = 0;
      write( fd_gpio, &value, sizeof(value) );
      clock_nanosleep( CLOCK_MONOTONIC, 0, &sleeptime, NULL );
   }
   return 0;
}
```

Die Applikation öffnet als Erstes die Gerätedatei `/dev/fastgpio`. Damit wird im Treiber über `driver_open()` der GPIO-Pin reserviert und als Ausgang konfiguriert. Falls das erfolgreich war, gibt `open()` einen positiven Filedeskriptor zurück. Danach schreibt die Applikation in einer Schleife abwechselnd »0« und »1« auf die Gerätedatei. Mit jedem Schreibzugriff wird `driver_write()` aufgerufen, das abhängig vom Wert die Leitung auf »1« oder eben auf »0« setzt.

Beispiel 6-8 *Makefile zur Blinklicht-Applikation <Makefile>*

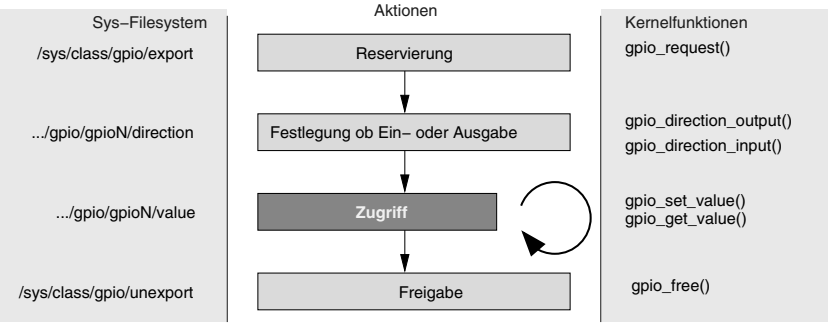
```
CFLAGS=-g -Wall
LDLIBS=-lrt

all: gpioappl

clean:
 rm -f gpioappl *.o
```

Der Quellcode der Applikation wird im Verzeichnis `~/embedded/application/gpioappl/` unter dem Namen `gpioappl.c` zusammen mit einem einfachen Makefile (Beispiel 6-8) abgelegt und durch Aufruf von `make ARCH=arm CC=arm-linux-cc` cross-kompiliert. Danach kann das Executable ebenfalls, beispielsweise per `scp`, auf den Raspberry Pi übertragen werden. Zum Test wird es unter dem Namen `gpioappl` gestartet:

```
# ./gpioappl
writing 1
writing 0
```

Ist Ihre Schaltung (siehe Abb. 5-3) korrekt aufgebaut, blinkt die LED. Übrigens ist diese Variante gut 250 Mal schneller als der Zugriff über das Sys-Filesystem [QuKu2013]. Unter Auslassung der Funktion `clock_nanosleep()` lassen sich abhängig von der Kernelversion und Kernelkonfiguration etwa 380 kHz für die Ein/Aus-Sequenz erreichen. Allerdings muss dabei die Treiberfunktion `driver_write()` auf das Minimum reduziert bleiben. Bereits eine Debugmeldung per `printk` verschlechtert das Zeitverhalten deutlich. Und noch ein Hinweis: Sobald Sie harte Zeitanforderungen haben, sollten Sie dem Programm mithilfe von `chrt` eine Realzeitpriorität geben. Aber Achtung: Läuft das Programm mit Realzeitpriorität, können Sie es nur abbrechen, wenn Sie entweder die Funktion `clock_nanosleep()` eingebaut lassen oder Ihrer Shell eine noch höhere Priorität zuweisen.

Abb. 6-2 *Aktionen und Zugriffsmöglichkeiten für GPIOs*



6.2.2 Digitale Eingabe

Um den Gerätetreiber um die digitale Eingabe per GPIO auf dem Raspberry Pi zu erweitern, sind die folgenden Schritte notwendig:

1. Auf dem Entwicklungsrechner ein neues Verzeichnis (`driver/fastgpio2/`) anlegen und den Treiberquellcode der digitalen Ausgabe kopieren
2. Den Treiberquellcode um die Funktion `driver_read()` erweitern und die `struct file_operations` anpassen
3. Im Treiberquellcode die Funktionen `driver_open()` und `driver_close()` um das Reservieren und Freigeben der GPIO-Leitung erweitern
4. Makefile kopieren und anpassen
5. Den Treiber mithilfe der Cross-Entwicklungswerkzeuge generieren
6. Eine Applikation vorbereiten und ebenfalls mithilfe der Cross-Entwicklungswerkzeuge generieren
7. Treiber und Applikation auf den Raspberry Pi kopieren
8. Auf dem Raspberry Pi den Treiber laden
9. Auf dem Raspberry Pi eine Gerätedatei anlegen
10. Auf dem Raspberry Pi die Funktionalität (zum Beispiel per `hexdump`) testen

Schaltungsaufbau

Die digitale Eingabe wird ähnlich angesteuert wie die Ausgabe. Zum Test ist eine Schaltung hilfreich, wie in Abbildung 6-3 dargestellt. Sie benötigen für diese zwei Widerstände, einer mit etwa 1 k, den zweiten mit etwa 10 k einen Taster und drei Leitungen. Der 10 k Widerstand wird auf der einen Seite mit 3,3 V und auf der anderen Seite mit dem Eingang des Tasters (oder Schalters) verbunden. Die andere Seite des Tasters wird an GND angeschlossen. Der Eingang des Tasters, der bereits mit dem Widerstand verbunden ist, wird jetzt mit GPIO17 verbunden. Aus Sicherheitsgründen stellen wir diese Verbindung über den 1k Widerstand her.

Wird der Taster nicht gedrückt, liegt am Eingang des Tasters und damit auch an GPIO17 eine Spannung von 3,3 V an. Der Raspberry Pi erkennt dieses als eine logische Eins. Wird der Taster aber gedrückt, liegt am Eingang 0 V, was der Raspberry Pi als logische Null interpretiert.

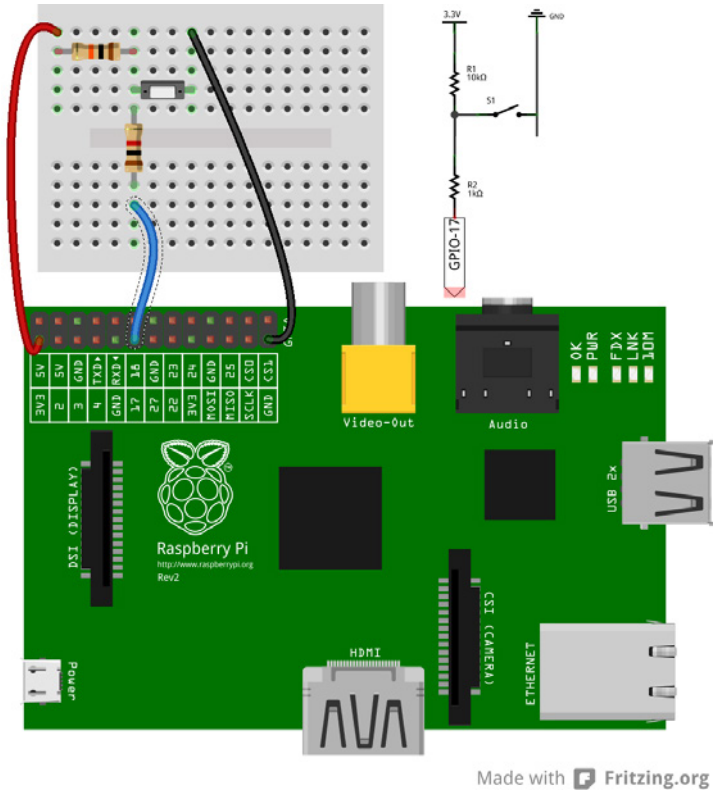


Abb. 6-3
Schaltung zum Anschluss eines Tasters an den Raspberry Pi

Zugriff über das Sys-Filesystem

Der erste Test der Schaltung erfolgt über das Sys-Filesystem. Dazu wird die GPIO-Leitung 17 (an der Doppelstiftleiste Pin 11) freigeschaltet, danach als Eingang konfiguriert und schließlich mit einem kleinen Shellskript der Wert an GPIO17 regelmäßig eingelesen:

```
# cd /sys/class/gpio
# echo 17 >export
# cd gpio17/
# echo "in" >direction
# while true
> do
> cat value
> sleep 1
> done
1
0
0
...
```

Normalerweise wird im Sekundenabstand auf dem Bildschirm eine »1« ausgegeben, sobald Sie den Taster drücken (hier nach einer Sekunde),

eine »0«. Durch <Strg><c> brechen Sie die Ausgabe ab. Vergessen Sie nicht nach Abschluss des Tests `cd ..; echo "17" >unexport` einzugeben.

Gerätetreiber

Für das Einlesen per GPIO soll der Treiber aus Abschnitt 6.2.1 erweitert werden. Für den Anwender ergibt sich damit das Interface, dass beim schreibenden Zugriff auf das Gerät `/dev/fastgpio` wie gehabt eine »0« oder eine »1« auf GPIO4 ausgegeben und zusätzlich beim lesenden Zugriff der Zustand von GPIO17 eingelesen wird.

Der Quellcode zum Gerätetreiber soll im Verzeichnis `~/embedded/driver/fastgpio2/` abgelegt werden. Am einfachsten ist es, wenn Sie das Verzeichnis anlegen und den Quellcode des bisherigen Treibers unter dem neuen Namen `fastgpio2.c` hineinkopieren. Im Makefile tauschen Sie dann ebenfalls den Namen `fastgpio` gegen `fastgpio2` aus.

Der bisherige Treiber hat bereits eine Gerätenummer beim Kernel angefordert und die zugehörigen Einträge im Sys-Filesystem erstellt. Da wir die gleiche Gerätenummer (248,0) mitverwenden, sind in der Funktion `mod_init()` keine weiteren Modifikationen notwendig.

Beispiel 6-9
Zugriffsfunktion auf GPIO17 im Treiber

```
static ssize_t driver_read(struct file *instanz,
    char __user *user, size_t count, loff_t *offset )
{
    unsigned long not_copied, to_copy;
    u32 value = 0;

    value = gpio_get_value( 17 );
    to_copy = min( count, sizeof(value) );
    not_copied=copy_to_user(user,&value,to_copy);
    return to_copy-not_copied;
}
```

Allerdings benötigen wir noch eine Funktion `driver_read()` für den lesenden Zugriff (Beispiel 6-9). Diese Funktion liest mit Hilfe von `gpio_get_value()` (siehe Tabelle 6-1) den GPIO17 ein und kopiert ihn danach per `copy_to_user()` an die von der Applikation zur Verfügung gestellte Adresse im Hauptspeicher. Das in der Zeile vorher aufgerufene Makro `min()` stellt erneut sicher, dass nicht mehr Daten kopiert werden, als vorhanden sind, beziehungsweise dass Speicher zur Verfügung gestellt wurde. Die Funktion gibt schließlich die Anzahl der erfolgreich kopierten Bytes zurück. Vergessen Sie nicht, in der `struct file_operations` die Funktion `driver_read` einzutragen.

Damit auf GPIO17 zugegriffen werden kann, muss die Leitung zunächst reserviert und dann konfiguriert werden (Abb. 6-2). Dazu sind in `driver_open()` die folgenden Zeilen vor das `return 0` einzufügen:

```
err = gpio_request( 17, "rpi-gpio-17" );
if (err) {
    printk("gpio_request failed %d\n", err);
    return -1;
}
err = gpio_direction_input( 17 );
if (err) {
    printk("gpio_direction_input failed %d\n", err);
    gpio_free( 17 );
    return -1;
}
printk("gpio 17 successfull configured\n");
```

Außerdem muss bei einem `close()` beziehungsweise Programmende der GPIO wieder freigegeben werden. Daher ist noch an das Ende der Funktion `driver_close()` die Freigabe einzubauen:

```
static int driver_close( struct inode *geraete_datei,
    struct file *instanz )
{
    gpio_free( 4 );
    gpio_free( 17 );
    return 0;
}
```

Flags auswerten

Öffnet eine Applikation das Gerät `/dev/fastgpio`, werden mit dem gegenwärtigen Code GPIO4 und GPIO17 reserviert, auch wenn die Applikation vielleicht nur Daten ausgeben möchte und damit GPIO17 gar nicht benötigt. Der Treiber soll so modifiziert werden, dass nur die GPIOs reserviert werden, die auch erforderlich sind. Werden für einen Lese- und Schreibzugriff sowohl GPIO4 als auch GPIO17 benötigt, müssen sich auch beide reservieren lassen.

Beispiel 6-10 *Lesender- und schreibender Zugriff im Treiber*

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/gpio.h>
#include <asm/uaccess.h>

static dev_t fastgpio_dev_number = MKDEV(248,0);
static struct cdev *driver_object;
static struct class *fastgpio_class;
static struct device *fastgpio_dev;

static int driver_open( struct inode *geraete_datei, struct file *instanz )
```

```
{
   int err;
   printk("driver_open\n");

   if (instanz->f_flags&O_RDWR || instanz->f_flags&O_WRONLY) {
      err = gpio_request( 4, "rpi-gpio-4" );
      if (err) {
         printk("gpio_request failed %d\n", err);
         return -1;
      }
      err = gpio_direction_output( 4, 0 );
      if (err) {
         printk("gpio_direction_output failed %d\n", err);
         gpio_free( 4 );
         return -1;
      }
      printk("gpio 4 successfull configured\n");
   }

   if (instanz->f_flags&O_RDWR || instanz->f_flags&O_RDONLY) {
      err = gpio_request( 17, "rpi-gpio-17" );
      if (err) {
         printk("gpio_request failed %d\n", err);
         if (instanz->f_flags&O_RDWR)
            gpio_free( 4 );
         return -1;
      }
      err = gpio_direction_input( 17 );
      if (err) {
         printk("gpio_direction_input failed %d\n", err);
         gpio_free( 17 );
         if (instanz->f_flags&O_RDWR)
            gpio_free( 4 );
         return -1;
      }
      printk("gpio 17 successfull configured\n");
   }

   return 0;
}


static int driver_close( struct inode *geraete_datei, struct file *instanz )
{
   printk( "driver_close called\n" );
   if (instanz->f_flags&O_WRONLY || instanz->f_flags&O_RDWR)
      gpio_free( 4 );
   if (instanz->f_flags&O_RDONLY || instanz->f_flags&O_RDWR)
      gpio_free( 17 );
   return 0;
}
```

```
static ssize_t driver_write(struct file *instanz, const char __user *user,
   size_t count, loff_t *offset )
{
   unsigned long not_copied, to_copy;
   u32 value=0;

   to_copy = min( count, sizeof(value) );
   not_copied=copy_from_user(&value, user, to_copy);
   printk("driver_read( %d )\n", value );
   if (value==0)
      gpio_set_value(4,0);
   else
      gpio_set_value(4,1);
   //gpio_set_value( 4, value?1:0 );
   return to_copy-not_copied;
}

static ssize_t driver_read(struct file *instanz,char __user *user,
    size_t count, loff_t *offset )
{
   unsigned long not_copied, to_copy;
   u32 value = 0;

   value = gpio_get_value( 17 );
   to_copy = min( count, sizeof(value) );
   not_copied=copy_to_user(user,&value,to_copy);
   return to_copy-not_copied;
}


static struct file_operations fops = {
    .owner  = THIS_MODULE,
    .open   = driver_open,
    .release= driver_close,
    .read   = driver_read,
    .write  = driver_write,
};

static int __init mod_init( void )
{
   if( register_chrdev_region(fastgpio_dev_number,1,"fastgpio")<0 ) {
      printk("devicenumber (248,0) in use!\n");
      return -EIO;
   }
   driver_object = cdev_alloc(); /* Anmeldeobjekt reserv. */
   if( driver_object==NULL )
      goto free_device_number;
   driver_object->owner = THIS_MODULE;
   driver_object->ops = &fops;
   if( cdev_add(driver_object,fastgpio_dev_number,1) )
      goto free_cdev;
```

```
   fastgpio_class = class_create( THIS_MODULE, "fastgpio" );
   if( IS_ERR( fastgpio_class ) ) {
      pr_err( "fastgpio: no udev support\n");
      goto free_cdev;
   }
   fastgpio_dev = device_create(fastgpio_class,NULL,fastgpio_dev_number,
      NULL, "%s", "fastgpio" );
   dev_info( fastgpio_dev, "mod_init called\n" );
   return 0;
free_cdev:
   kobject_put( &driver_object->kobj );
free_device_number:
   unregister_chrdev_region( fastgpio_dev_number, 1 );
   printk("mod_init failed\n");
   return -EIO;
}

static void __exit mod_exit( void )
{
    dev_info( fastgpio_dev, "mod_exit called\n" );
    device_destroy( fastgpio_class, fastgpio_dev_number );
    class_destroy( fastgpio_class );
    cdev_del( driver_object );
    unregister_chrdev_region( fastgpio_dev_number, 1 );
    return;
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

Eine Applikation gibt beim Öffnen bereits die Art an, wie sie auf Daten zugreifen möchte, nämlich nur lesend (`O_RDONLY`), nur schreibend (`O_WRONLY`) oder lesend und schreibend (`O_RDWR`). Wenn wir diese Angabe in der Funktion `driver_open()` auswerten, können wir abhängig von der Zugriffsart GPIO4 und GPIO17 reservieren und in der Funktion `driver_close()` wieder freigeben. Der vollständige Code ist in Beispiel 6-10 zu sehen.

Beispiel 6-11 zeigt die Befehlssequenz, um den Quellcode zu kopieren und um die digitale Eingabe zu erweitern. Weiterhin wird gezeigt, wie die Cross-Generierung gestartet und das Ergebnis per `scp` auf das Selbstbausystem kopiert wird. Dort laden Sie den Gerätetreiber per `insmod`. Als Nächstes legen Sie noch die Gerätedatei an. Um den Treiber zu testen, können Sie das Programm `hexdump` verwenden (soweit es von Ihnen über Busybox konfiguriert und generiert wurde).

Da wir im Treiber immer 32 Bit kopieren, ist der über Hexdump gelieferte Wert auch als 32 Bit zu interpretieren. Anfangs werden lauter

Einsen gelesen. Dann wurde der Taster gedrückt (nach dem ersten Stern) und `hexdump` liefert die Nullen. Der Taster wird wieder losgelassen und es folgen Einsen. Durch Drücken der Tastenkombination <Strg><c> brechen Sie jederzeit `hexdump` und damit die Ausgabe ab.

Beispiel 6-11
Befehle für die Treiberergänzung zur digitalen Eingabe

Befehlssequenz auf dem Entwicklungsrechner:

```
quade@felicia:~/embedded> mkdir -p driver/fastgpio2
quade@felicia:~/embedded> cd driver/fastgpio2
quade@felicia:~/embedded/driver/fastgpio2> cp ../fastgpio/fastgpio.c \
   fastgpio2.c
quade@felicia:~/embedded/driver/fastgpio2> cp ../fastgpio/Makefile .
quade@felicia:~/embedded/driver/fastgpio2> gedit Makefile &
    # fastgpio durch fastgpio2 ersetzen
quade@felicia:~/embedded/driver/fastgpio2> gedit fastgpio2.c &
    # driver_open und driver_close anpassen
    # driver_read ergänzen und in struct file_operations eintragen
quade@felicia:~/embedded/driver/fastgpio2> make ARCH=arm \
    CROSS_COMPILE=arm-linux- \
    KDIR=~/embedded/raspi/buildroot-2013.05/output/build/linux-e959a8e/
...
quade@felicia:~/embedded/driver/fastgpio2> scp fastgpio2.ko \
   root@192.168.178.31:
...
```

Befehlssequenz auf dem Target:

```
# insmod fastgpio2.ko
# mknod /dev/fastgpio c 248 0
# hexdump /dev/fastgpio
0000000 0001 0000 0001 0000 0001 0000 0001 0000
*
0815f90 0000 0000 0000 0000 0000 0000 0000 0000
*
1004830 0000 0000 0000 0000 0001 0000 0001 0000
1004840 0001 0000 0001 0000 0001 0000 0001 0000
*
1814670 0001 0000 0001 0000 0001 0000 0000 0000
1814680 0000 0000 0000 0000 0000 0000 0000 0000
*
```

6.2.3 Programmierhinweise zum Hardwarezugriff

Bei der Programmierung von Hardwarezugriffen gibt es einige Fallstricke. Das betrifft die Wortbreite, Byte-Ablagefolge und das Alignment.

Wortbreite

Die Wortbreite definiert die Anzahl Bits, die in ein CPU-Register, und damit auch in eine Speicherzelle passen. Der Raspberry Pi beispielsweise verwendet normalerweise eine Wortbreite von 32 Bit, moderne Desktop- und vor allem Serverprozessoren setzen dagegen auf 64 Bit. Deeply Embedded Systems arbeiten häufig mit 16 oder gar 8 Bit. Werden Systeme unterschiedlicher Wortbreite miteinander gekoppelt und wird dieses programmtechnisch nicht berücksichtigt, kommt es zwangsläufig zu Fehlern, beispielsweise wenn ein 32-Bit-Wert auf ein 16-Bit-Register geschrieben wird.

Wenn Sie die Headerdatei `<linux/types.h>` inkludieren, gibt es an der Applikationsschnittstelle die Datentypen:

- __u8, __u16, __u32, __u64 für vorzeichenlose (unsigned) Datentypen und
- __s8, __s16, __s32, __s64 für vorzeichenbehaftete (signed) Datentypen.

Im Treiber stehen die gleichen Datentypen, jedoch ohne »__«, zur Verfügung:

- u8, u16, u32, u64 für vorzeichenlose (unsigned) Datentypen und
- s8, s16, s32, s64 für vorzeichenbehaftete (signed) Datentypen.

Byte-Ablagefolge

Es gibt zwei Methoden, 16-, 32- oder 64-Bit-Worte im Speicher abzulegen. Wird das niederwertigste Byte (beim 32-Bit-Wort `0xDDCCBBAA` also `0xAA`) auch an der niederwertigsten Speicherstelle abgelegt, spricht man vom Little-Endian-Format. Wird es jedoch an der höchstwertigen Adresse abgelegt, ist Big Endian gemeint (siehe Abb. 6-4).

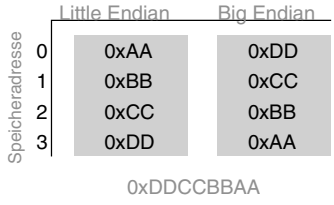


Abb. 6-4
Byte-Reihenfolge des 32-Bit-Wortes 0xDDCCBBAA

Die Ablagefolge muss immer dann programmtechnisch berücksichtigt werden, wenn Daten zwischen Rechnern unterschiedlicher Ablagefolgen ausgetauscht werden. Ein solcher Austausch findet nicht nur bei der Pe-

ripherieanbindung statt, sondern auch bei einer Rechner-/Rechner-Kommunikation (beispielsweise über tcp/udp/ip). Liegt ein solcher Fall vor, wird der periphere Datentyp gemäß Byte-Ablagefolge als Little Endian oder als Big Endian gekennzeichnet:

- le8, le16, le32, le64 für Worte im Little-Endian-Format und
- be8, be16, be32, be64 für Worte im Big-Endian-Format.

Nach dem Einlesen eines solchen Wortes wird es in den Hosttyp umgewandelt beziehungsweise andersherum: Vor dem Schreiben eines solchen Wortes wird es vom Hostformat in das jeweilige von der spezifischen Hardware verwendete Ablageformat (Little oder Big Endian) umgewandelt. Dazu stehen die in Tabelle 6-2 dargestellten Kernelmakros zur Verfügung. Die Makros `htons`, `htonl`, `ntohs` und `ntohl` dienen der Umwandlung von Datentypen, die bei der Netzwerkprogrammierung verwendet werden.

Tabelle 6-2 *Makros zur Konvertierung zwischen unterschiedlichen Wortablageformaten*

Funktion	Kurzbeschreibung
htons, htonl	»host to net short«: Hostformat (__u16, __u32) in das Netzwerkformat (__be16, __be21) umwandeln
ntohs, ntohl	»net to host short«: Netzwerkformat (__be16, __be32) in das Hostformat (__u16, __u32) umwandeln
cpu_to_le16, cpu_to_le32, cpu_to_le64	»cpu to little endian«: Wort im Hostformat in das Format »Little Endian« umwandeln
cpu_to_be16, cpu_to_be32, cpu_to_be64	»cpu to big endian«: Wort im Hostformat in das Format »Big Endian« umwandeln
le_to_cpu16, le_to_cpu32, le_to_cpu64	»little endian to cpu«: Wort im Format »Little Endian« in das Hostformat umwandeln
be_to_cpu16, be_to_cpu32, be_to_cpu64	»big endian to cpu«: Wort im Format »Big Endian« in das Hostformat umwandeln

Alignment

Werden nicht nur einzelne Worte, sondern ganze Datenstrukturen zwischen der Peripherie und dem Kernel oder einer Applikation ausgetauscht, ist zudem auf die Datenablage zu achten. Die Zugriffe einer CPU sind nämlich schneller, wenn Daten auf die natürliche Wortgrenze der CPU ausgerichtet (aligned) sind. Da der Compiler dies berücksichtigt, führt das häufig dazu, dass zwischen den einzelnen Elementen einer Datenstruktur Lücken bleiben, wie in Abbildung 6-5 zu sehen ist.

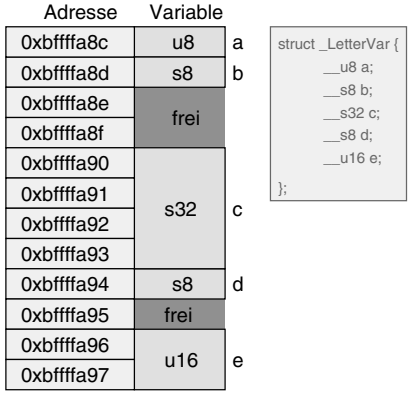


Abb. 6-5
Auf die Wortbreite ausgerichtete Datenstruktur [QuKu2011b]

Wird die gleiche Datenstruktur für eine Architektur mit anderer Wortbreite übersetzt, ergibt sich häufig eine andere Datenablage. Gerade im Bereich eingebetteter Systeme gibt es dabei keine Lücken. Um den Compiler anzuweisen, die Daten ohne Lücken abzulegen, wird beim Compiler (GCC) das Schlüsselwort `__attribute__ (packed)` eingesetzt. Damit ergibt sich die in Abbildung 6-6 dargestellte Datenablage.

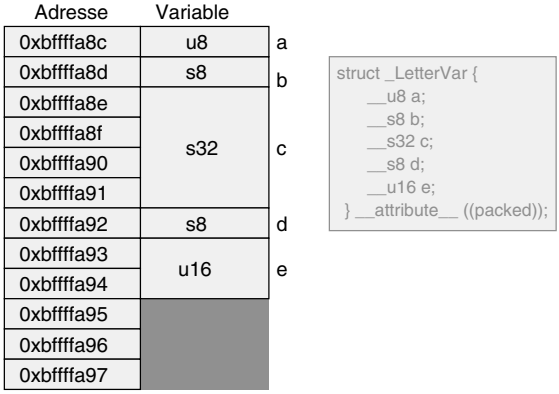


Abb. 6-6
Datenstruktur in gepackter Form [QuKu2011b]

7 Embedded Security

Sicherheit hat nichts mit Vertrauen,
aber viel mit Misstrauen zu tun.

IT-Security ist ein komplexes Thema, das sich sehr aktiv weiterentwickelt. Für Techniken, die heute als sicher gelten, gibt es morgen möglicherweise bereits effektive Angriffsvektoren. Dieser Abschnitt ist eine Einführung und entbindet den Entwickler nicht davon, sich regelmäßig über Schwachstellen in Werkzeugen und Systemen auf dem Laufenden zu halten.

Während im deutschen Sprachgebrauch nur das Wort »Sicherheit« verwendet wird, unterscheidet man im englischsprachigen Raum zwischen »Safety« und »Security«. Safety lässt sich ins Deutsche gut mit Betriebssicherheit und Security mit Angriffssicherheit übersetzen. Bei der Betriebssicherheit geht es darum, die Umwelt vor dem betrachteten System zu schützen, bei der Angriffssicherheit geht es um den Schutz des betrachteten Systems vor der Umwelt.

Dass vernetzte und nicht vernetzte Geräte Angriffen ausgesetzt sind, ist nicht erst seit dem Computerwurm Stuxnet [stuxnet] bekannt. PRISM [prism] zeigt außerdem eindrucksvoll, dass keinem vertraut werden kann, wenn es um Sicherheit und Datenschutz geht. Hier kann Linux einen Trumpf ausspielen, da der Entwickler und Anwender dank Open Source die Möglichkeit hat, das System auf Schwachstellen und versteckte Hintertüren zu untersuchen. Ein System, das Closed Source Software verwendet, kann nicht sicher sein.

Die Berücksichtigung des Themas IT-Security bei der Entwicklung eines eingebetteten Systems ist ein Muss! Dabei gilt es, das System zu härten (also abzusichern), die Applikation so zu gestalten, dass Sicherheit gewährleistet werden kann, und die Entwicklung unter Sicherheitsaspekten durchzuführen.

Grundlagen der IP-Kommunikation

Eingebettete Systeme werden zunehmend vernetzt, um damit beispielsweise die Möglichkeit zur Fernwartung zu erschließen. Über das Internet sind damit Statusinformationen abrufbar und Steuerparameter setzbar.

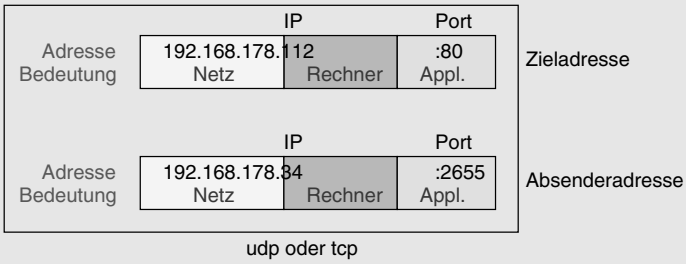


Abb. 7-1
Adressierungsinformationen von Internetprotokollen

Rechner oder allgemein Netzkomponenten benötigen zur Erreichbarkeit im Internet eine IP-Adresse. IP steht für das Internetprotokoll. Zurzeit wird zumeist die Protokollversion IPv4 verwendet, ein Wechsel auf IPv6 zeichnet sich ab.

Eine IPv4-Adresse besteht aus 32 Bit, wobei diese typischerweise durch vier durch Punkte abgetrennte Dezimalzahlen dargestellt werden: 91.201.120.87. Jede Zahl repräsentiert 8 Bit der Adresse, hat also damit einen Wertebereich von hexadezimal 0x00 bis 0xff respektive dezimal von 0 bis 255. Wichtig ist, dass diese Adresse zwei Adresssierungsinformationen enthält: erstens eine Netzwerkadresse und zweitens eine Rechneradresse. Die Grenze zwischen Netzadresse und Rechneradresse ist flexibel. Die zu einer Adresse gehörende, bitorientierte Netzmaske (netmask) legt fest, welche Adressteile das Netz und welche die eigentliche Komponente adressieren. Eine Netzmaske von 255.255.255.0 beispielsweise zeigt, dass die ersten drei Zahlen (24 Bit) der IP-Adresse das Netz und die letzte Zahl (8 Bit) den Rechner adressieren. Häufig wird die Netzadresse auch durch Angabe der Bits, die die Netzadresse repräsentieren, per Schrägstrich einer IP-Adresse angefügt: 92.201.120.87/27. In diesem Fall sind 27 Bits der Adresse für die Netz- und die restlichen 5 für die Rechnerwahl vorgesehen.

Rechnernetze sind über Router, auch Gateways genannt, untereinander verbunden. Nur Router können Daten zwischen den Netzen weiterleiten. Verschickt ein Rechner Daten an einen Rechner in einem anderen Netz, muss er diese Daten dem Gateway übergeben. Damit benötigt der Rechner in seiner Konfiguration eine Gateway-Adresse, die im gleichen Netz liegt wie der sendende Rechner selbst.

Da eine auf Zahlen basierende Adressierung für uns Menschen schwer zu handhaben ist, gibt man den Rechnern beziehungsweise Netzkomponenten zusätzlich Namen, beispielsweise www.dpunkt.de. In hierarchisch organisierten Datenbanken ist die Zuordnung zwischen einem Rechnernamen und der jeweiligen IP-Adresse abgelegt. Die IP-Adresse eines Rechners, der eine solche Datenbank beherbergt (Domain Name Service, DNS), muss ebenfalls konfiguriert sein, damit ein Rechner eine Namensauflösung durchführen kann.

Damit man einem Rechner unabhängig von seinem Standort Daten schicken kann, muss seine IP-Adresse weltweit eindeutig sein. Diese IP-Adressen, die von einer zentralen Instanz zugeteilt werden, nennt man *öffentliche IP-Adressen*. Daneben gibt es noch *private IP-Adressen*. Diese kann jeder verwenden, sie sind aber nur im lokalen Netz gültig. Öffentliche Gateways dürfen keine privaten IP-Adressen routen, also Daten an Rechner mit privaten Netzadressen weiterleiten. Netzbereiche mit privaten Adressen sind in Tabelle 7-1 gelistet.

IP Adressbereich
10.0.0.0 bis 10.255.255.255
172.16.0.0 bis 172.31.255.255
192.168.0.0 bis 192.168.255.255

Tabelle 7-1
Private IP-Adressbereiche

Mit einer IP-Adresse kann zwar ein Rechner erreicht werden, aber nicht eine Applikation, wie beispielsweise ein WWW-Server oder ein SSH-Server. Daher hat man die IP-Adresse um einen zusätzlichen Teil erweitert, die Portadresse. Die Portadresse liegt im Bereich von 0 bis 65535 (16 Bits). Es ist Konvention, dass definierte Applikationen immer die gleichen, festgelegten Portadressen verwenden. Ein Webserver beispielsweise ist über die Portadresse 80 erreichbar, der SSH-Server über 22 und der Domain Name Service (DNS) über 53. Das ist aber kein Muss, der Webserver könnte genauso Port 333 verwenden. Wenn der Client die Portadresse kennt, kann er den Webserver erreichen. Die bekanntesten Protokolle, die nicht nur eine IP-Adresse, sondern zugleich eine Portadresse verwenden, sind TCP und UDP.

Damit zwei Applikationen Daten austauschen können, besteht eine vollständige Adressangabe aus der Empfänger- und der Absenderadresse (IP plus Port), also aus vier Adressinformationen.

7.1 Härtung des Systems

Eine wesentliche Aufgabe bei der Entwicklung eines eingebetteten Systems ist die Absicherung vor Hackerangriffen, man spricht auch von »Härtung«.

Linux bietet diverse Eigenschaften, die zur Erhöhung der Angriffssicherheit beitragen:

- Firewall
- Intrusion Detection and Prevention
- Rechtevergabe
- Ressourcenverwaltung
- Entropie-Management

- ASLR
- DEP
- Systemcall-Supervising

Über die Firewall wird der Netzverkehr überwacht und gefiltert. Die Aktivierung der Firewall stellt damit eine der wichtigsten Maßnahmen zur Härtung eines vernetzten Systems dar. Die Firewall lässt sich auch sinnvoll mit einem Intrusion Detection System (IDS) beziehungsweise noch besser mit einem Intrusion Prevention System (IPS) kombinieren. Daneben bildet das Rechtemanagement und hier wiederum die Einschränkung der Rechte auf ein Minimum eine wesentliche Komponente zur Absicherung. Die Ressourcenverwaltung über beispielsweise Quotas und Cgroups verhindert, dass wichtigen Systemteilen, beispielsweise der funktionsbestimmenden Applikation, die Betriebsmittel wie Speicher oder Rechenzeit ausgehen. Das Entropie-Management schließlich muss aktiv unterstützt werden, damit ausreichend qualitativ hochwertige Zufallszahlen zur Verfügung stehen. Diese werden für die Verschlüsselung benötigt. ASLR (Address Space Layout Randomization) und DEP (Data Execution Prevention) sind zwei Techniken, die standardmäßig aktiviert sind und die wesentlichen Angriffe auf Applikationen, sogenannte Buffer-Overflows, erschweren. Als Letztes schließlich bietet Linux über die Linux Security Moduls (LSM) die Möglichkeit, Systemcalls gezielt zu überwachen und über eine Rollenvergabe dediziert Zugriffe zu filtern. Mit Ausnahme dieses Systemcall-Supervising sollen die übrigen Maßnahmen im Folgenden etwas detaillierter vorgestellt werden.

Sie werden in Ihrem System nicht zwangsläufig alle Maßnahmen umsetzen, sondern diese bezüglich Anforderungen, Risiko und Aufwand abwägen.

7.1.1 Firewalling

Das müssen Sie wissen

Als Firewall wird eine Softwarekomponente oder auch ein komplettes Gerät bezeichnet, das den Datenverkehr filtert und nur als unbedenklich eingestufte Daten an Applikationen oder andere Rechner weiterleitet. Das Filtern wird typischerweise über Filterregeln konfiguriert, die aus einem Bedingungs- und einem Aktionsteil bestehen. Falls eine Bedingung zutrifft, wird die zugehörige Aktion durchgeführt. Typische Aktionen sind beispielsweise:

- Das Zurückweisen (Ausfiltern) eines Paketes
- Das Weiterleiten eines Paketes
- Das Mitprotokollieren eines Paketes
- Die gezielte Manipulation von Teilen des Paketes (Masquerading)
- Das (direkte) Beantworten eines Paketes (ohne dass dieses den wirklichen Empfänger erreicht hat)

Firewalls helfen, Dienste abzusichern, die ungewollt/unbewusst aktiv sind. Firewalls helfen, die Last auf einem System gering zu halten (Schutz vor Überlast), und schützen dadurch, dass kein unnötiger Code durchlaufen wird, der möglicherweise Fehler enthält.

Realisierungsform einer Firewall:

- Netzwerk-Firewall: Router zwischen zwei Netzen
- Personal-Firewall: Filtern des ankommenden und abgehenden Datenverkehrs
- Transparente Firewall: Firewall in einem Leitungsstrang, Bridgebetrieb, gleiches Subnetz rechts und links der Firewall

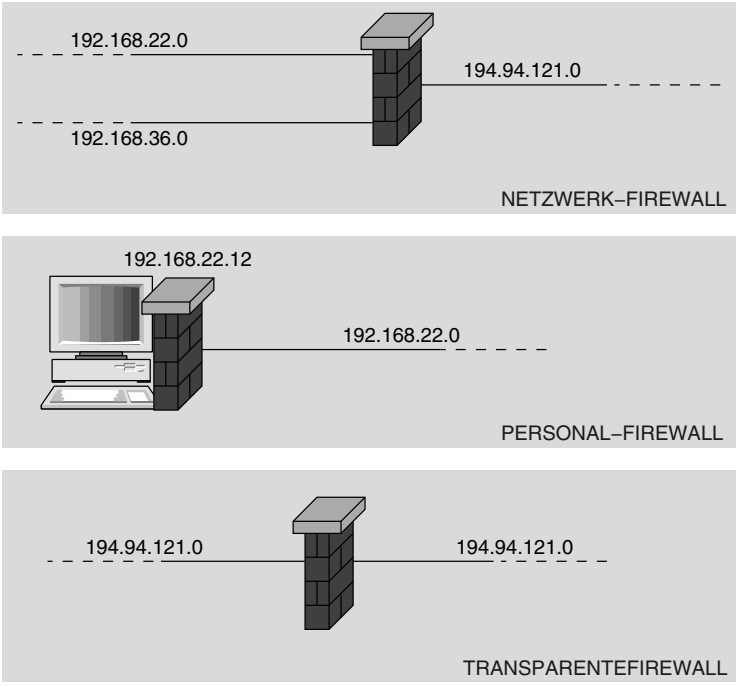


Abb. 7-2
Unterschiedliche Realisierungsformen für Firewalls

Eine Personal-Firewall hat typischerweise genau ein Netzwerkinterface, eine transparente Firewall verfügt über zwei Netzwerkinterfaces und eine Netzwerk-Firewall hat zwei oder mehr Netzwerkinterfaces.

Vorteil der transparenten Firewall: Diese benötigt keine IP-Adresse, sie kann daher auch nicht adressiert werden und ist damit schwer angreifbar.

Prinzipiell werden die beiden folgenden Arten von Firewalls unterschieden:

- Paketfilter
- Application-Level-Firewall

Paketfilter

Wie der Name andeutet, filtert der Paketfilter die Datenpakete typischerweise auf Basis der Absender- und der Zieladresse und des jeweiligen Absenderports und/oder Empfängerports. Auch lässt sich die Protokollart als solche (zum Beispiel UDP oder TCP) berücksichtigen.

Schwierig wird das Filtern, wenn Portadressen nicht bekannt sind, sondern wie beispielsweise bei FTP (File Transfer Protocol) dynamisch ausgehandelt werden. In diesem Fall muss die Firewall den Datenverkehr interpretieren und selbstständig Regeln aufbauen beziehungsweise aktivieren. Dieser Vorgang wird mit »stateful inspection« (auch mit »connection tracking«) bezeichnet. Bei der statischen Filterung werden mit jedem ankommenden Paket die vier Adressinformationen (Quell-IP, Ziel-IP, Quellport, Zielport) untersucht. Bei der »stateful inspection« hingegen werden Informationen zu einzelnen Verbindungen abgespeichert. Die Firewall verfolgt beispielsweise beim FTP die Kommandoübertragung. Stellt die Firewall dabei fest, dass ein neuer Port für die Datenübertragung festgelegt wird, gibt die Firewall diesen neuen Port nur für die bestehende Verbindung (identifizierbar an den IP- und Portadressen) frei. Ist die Datenübertragung abgeschlossen, wird der Port durch das Löschen der dynamisch erstellten Regel wieder gesperrt.

Diese Technik ermöglicht auch die begrenzte Freigabe. Ein Port für einen Rückkanal wird erst dann in der Firewall freigegeben, wenn zuvor eine Anfrage hereingekommen ist. Beispiel: Gemäß Firewallkonfiguration werden ankommende Pakete für Port 80 (http) zugelassen, abgehende Pakete sind aber zunächst gesperrt. Erst wenn eine Anfrage auf Port 80 eingeht, wird für genau die IP-Adresse des anfragenden Rechners eine abgehende Verbindung freigeschaltet. Diese Technik verhindert, dass eine unerkannt installierte Malware (bösartige Software) von sich aus eine Verbindung zu einem externen Rechner aufbaut.

Typischerweise besitzt eine Firewall mehrere Regelsätze:

- Einen Satz für ankommende Pakete (Input-Rules)
- Einen Satz für abgehende Pakete (Output-Rules)
- Einen Satz für durchgehende Pakete (Forwarding-Rules)

Prinzipiell gibt es zwei Betriebsarten für die Firewall:

1. White-Listing: Alles, was nicht explizit erlaubt ist, ist verboten (default DENY).
2. Black-Listing: Alles, was nicht explizit verboten ist, ist erlaubt (default ACCEPT).

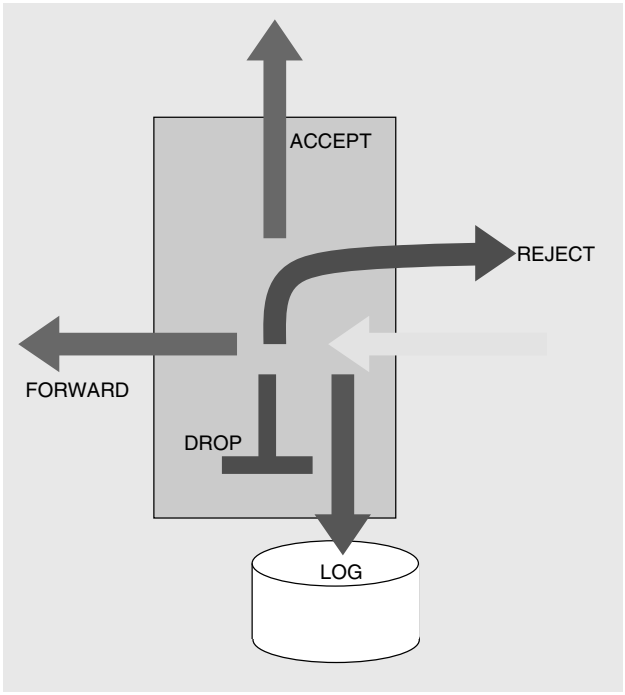


Abb. 7-3
Auswahl der möglichen Paketfilter-Aktionen

Application-Level-Firewall

Die Application-Level-Firewall filtert aufgrund der im Paket enthaltenen Nutzdaten. Sie achtet also beispielsweise darauf, dass auf Ebene des HTTP (Port 80) nicht irgendwelche Daten, sondern (syntaktisch) korrekte HTTP-Pakete ausgetauscht werden (Überwachung des Paketaufbaus).

Meist ist die Application-Level-Firewall als HTTP-Proxy realisiert. Hierbei schickt der Browser seine Anfrage nicht direkt ins Internet, sondern übergibt diese dem Proxy (Stellvertreter). Der Proxy überprüft die

Anfrage auf Korrektheit und leitet sie dann an die Zieladresse weiter. Er nimmt die Antwort entgegen, überprüft diese wiederum und gibt sie dem anfragenden Browser zurück.

Da bei der Application-Level-Firewall eine semantische Überprüfung der Datenpakete stattfindet, kann hiermit eine höhere Sicherheit erreicht werden.

Firewall auf dem Raspberry Pi

Um eine Firewall auf dem Raspberry Pi für das Buildroot-System zu aktivieren, sind die folgenden Schritte notwendig:

1. Skript zur Deaktivierung der Firewall im Verzeichnis `~/embedded/raspi/userland/target/` erstellen.
2. Skript zur Aktivierung der Firewall erstellen. Beispielsweise die Dienste DNS, ICMP, SSH, NTP, HTTP und HTTPS freigeben. Das Skript vom Entwicklungsrechner aus per `nmap` testen, nachdem es auf das Target transferiert und dort ausgeführt wurde.
3. Init-Skript `S45firewall` zum automatischen Aktivieren der Firewall beim Hochfahren des Targets erstellen.
4. Die Datei `~/embedded/raspi/scripts/postbuild.sh` um Befehle ergänzen, die die Skripte auf das Target kopieren.
5. `iptables` in das Buildroot-System konfigurieren.
6. Das System durch Aufruf von `make` aus dem Buildroot-Verzeichnis heraus generieren. Zum Test auf dem Target einloggen und `iptables -L` aufrufen respektive wieder `nmap` verwenden.

Die Komponente, die den Paketfilter im Linux-Kernel realisiert, wird Netfilter genannt. Die Applikation (Userland), mit der der Netfilter konfiguriert wird, heißt `iptables`.

Während die Distribution Raspbian bereits das Kommando `iptables` mitbringt, muss es für die Buildroot-Variante erst ausgewählt werden. Rufen Sie daher im Hauptverzeichnis von Buildroot `make menuconfig` auf und wählen Sie unter [Package Selection for the target][Networking applications] iptables aus. Danach generieren Sie das System neu und installieren es.

Die Linux-Firewall filtert den eingehenden (INPUT), den ausgehenden (OUTPUT) und den durchgereichten (FORWARD) Datenverkehr. Eingehend bedeutet hierbei, dass ein Paket für den Rechner, auf dem die Firewall läuft, bestimmt ist (siehe Abb. 7-4).

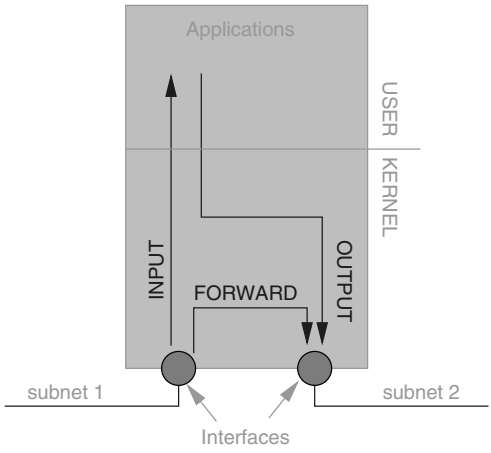


Abb. 7-4
Quelle und Ziel eines Paketes bestimmen die Filterkette.

Die Filterung wird über ein Regelwerk gesteuert. Die Regeln werden dabei in Listen abgelegt und das Netfilter-Framework arbeitet jede Regel in der Liste von oben nach unten ab, so dass die Position einer Regel innerhalb der Liste durchaus relevant ist. Eine Regel besteht aus einem Bedingungs- und einem Aktionsteil. Falls die Bedingungen in der Regel auf ein Paket zutreffen, wird die zugehörige Aktion durchgeführt. So wird das Paket beispielsweise akzeptiert und an eine Applikation durchgereicht. Trifft keine Regel in der Liste zu, wird die Default-Policy — quasi die letzte Regel in der Liste — angewendet.

Von den beiden relevanten Möglichkeiten, als Default-Policiy alle Pakete zu akzeptieren und Pakete, die den Filterkriterien entsprechen, zu verwerfen (Black-Listing) oder aber alle bisher nicht gefilterten Pakete zu verwerfen und nur gefilterte Pakete durchzulassen (White-Listing), wird typischerweise die zweite Methode, das White-Listing, angewendet.

Die Linux-Firewall stellt dabei eine ganze Reihe unterschiedlicher Filterkriterien zur Verfügung:

- Interface (Device, Schnittstelle), Option `-i`
- Protokoll (tcp, udp, icmp), Option `-p`
- Empfänger-IP-Adresse, Option `-d`
- Absender-IP-Adresse, Option `-s`
- Empfängerport, Option `--dport`
- Absenderport, Option `--sport`
- Zustandsbit im Protokoll (`SYN`, ...)

Falls ein Paket den konfigurierten Filterbedingungen entspricht, können die in Tabelle 7-2 aufgeführten Aktionen ausgeführt werden.

Tabelle 7-2 *Aktionen bei Firewallregeln*

Iptables-Option	Bedeutung
-j ACCEPT	Paket annehmen
-j DROP	Paket verwerfen
-j REJECT	Paket zurückweisen
-j LOG	Headerdaten mitprotokollieren
-j MASQUERADE	Abgehende Pakete modifizieren
-j DNAT	Ankommende Pakete modifizieren

Um sich auf einem Linux-System die Filterregeln anzeigen zu lassen, rufen Sie das Kommando `iptables -L` auf.

```
# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Um die Default-Policy zu ändern, wird `iptables` mit der Option `-p` aufgerufen. Mit den folgenden drei Kommandos wird die Firewall aktiviert, sie verbietet jede externe Kommunikation:

```
# iptables -P INPUT DROP
# iptables -P OUTPUT DROP
# iptables -P FORWARD DROP
```

Mit der Option `-A` hängen Sie Regeln der angegebenen Liste an.

Um beispielsweise den Webserver von außen zugänglich zu machen, benötigen Sie zunächst zwei Regeln: Akzeptieren Sie ankommende (`INPUT`) Pakete, die an den Webserver (an Zielport 80, iptables-Option `--dport 80`) per Protokoll `tcp` gerichtet sind. Geben Sie außerdem die Antworten des Webservers frei, also ausgehende (`OUTPUT`) Pakete, die von dem Webserver (von Quellport 80, iptables-Option `--sport 80`) stammen:

```
# iptables -A INPUT -p tcp --dport 80 -j ACCEPT
# iptables -A OUTPUT -p tcp --sport 80 -j ACCEPT
```

Wollen Sie vom Raspberry Pi aus einen Webserver kontaktieren, benötigen Sie außerdem zwei weitere Regeln, bei denen im Gegensatz zu vorher Quell- und Zielports vertauscht sind. Unser Webbrowser sendet Pakete an Port 80 und empfängt Pakete von Port 80.

```
# iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
# iptables -A INPUT -p tcp --sport 80 -j ACCEPT
```

Damit bei der Nutzung als Client auch Webpräsenzen über den Namen und nicht nur über die IP-Adresse angesprochen werden können, muss außerdem der Domain Name Service freigegeben werden. Dieser wird über Port 53 abgewickelt (Protokoll sowohl tcp als auch udp):

```
# iptables -A OUTPUT --dport 53 -j ACCEPT
# iptables -A INPUT --sport 53 -j ACCEPT
```

Mit der Option `-D` können übrigens Regeln wieder aus der Regelliste gelöscht werden:

```
# iptables -D INPUT -p udp --sport 53 -j ACCEPT
```

Mit der Aktion `LOG` werden Pakete typischerweise in der Datei `/var/log/kern.log` mitprotokolliert. Die folgende Regel protokolliert die Verbindungsparameter sämtlicher ankommender TCP-Pakete:

```
# iptables -D INPUT -p tcp -j LOG
```

Bei der Verwendung der obigen Regel auf einem eingebetteten System sollten Sie doppelt vorsichtig sein. Zum einen wird für das Abspeichern viel Speicherplatz benötigt, zum anderen entsprechende Rechenzeit.

Es ist vorteilhaft, die Kommandos zur Firewallkonfiguration in ein Skript zu schreiben, das beim Booten automatisch gestartet wird. Ein solches Skript beginnt üblicherweise mit dem Rücksetzen der Firewall. Hierzu gehört auch das Rücksetzen der bisher unerwähnt gebliebenen Mangle- und Nat-Listen, die für Veränderungen an den Paketen genutzt werden können. Für den Test einer Firewallkonfiguration ist es günstig, zusätzlich ein Skript zu haben, das die Firewall durch Zurücksetzen ausschaltet. Dieses könnte beispielsweise `fw_open.sh` heißen (Beispiel 7-1).

Beispiel 7-1
Skript zur Deaktivierung der Firewall auf dem Raspberry Pi <fw_open.sh>

```
#!/bin/sh

iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT
iptables -P FORWARD ACCEPT
```

Das Skript zur Aktivierung der Firewall nennen wir `fw_up.sh`. Neben dem Rücksetzen enthält es die bereits oben vorgestellten Regeln. Zusätzlich erlauben wir zu Fernwartungszwecken noch den SSH-Zugriff und für das Setzen der Uhrzeit NTP. Damit die Erreichbarkeit getestet werden kann, soll außerdem `ping` unterstützt werden, wozu das Protokoll `icmp` freigegeben wird. Zusammen ergibt sich das in Beispiel 7-2 gezeigte Skript.

Beispiel 7-2 *Aktivierung der Firewall auf dem Raspberry Pi <fw_up.sh>*

```
#!/bin/sh

iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT DROP
iptables -P OUTPUT DROP
iptables -P FORWARD DROP

# HTTP Server
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
iptables -A OUTPUT -p tcp --sport 80 -j ACCEPT

# HTTP Client
iptables -A OUTPUT -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp --sport 80 -j ACCEPT

# DNS
iptables -A OUTPUT --dport 53 -j ACCEPT
iptables -A INPUT --sport 53 -j ACCEPT

# SSH Server
iptables -A INPUT -p tcp --dport 22 -j ACCEPT
iptables -A OUTPUT -p tcp --sport 22 -j ACCEPT

# SSH Client
iptables -A OUTPUT -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -p tcp --sport 22 -j ACCEPT

# NTP
iptables -A OUTPUT -p udp --dport 123 -j ACCEPT
iptables -A INPUT -p udp --sport 123 -j ACCEPT
```

Test der Firewall

Bevor die Skripte in das System integriert werden, sollten Sie diese testen. Transferieren Sie sie auf den Raspberry Pi und machen Sie sie mit dem Kommando `chmod +x` ausführbar.

Deaktivieren Sie die Firewall zunächst durch Aufruf von `fw_open.sh`. Zum Test der Funktionsfähigkeit einer Firewall kann man das Kommando `nmap` auf dem Hostsystem aufrufen, das per `sudo apt-get install nmap` auf einem Ubuntu schnell installiert ist. In Beispiel 7-3 sehen Sie den Aufruf und die Ausgabe, wenn auf dem Raspberry Pi die Firewall deaktiviert ist.

```
quade@felicia:~$ nmap -p 1-65535 192.168.178.31

Starting Nmap 5.21 ( http://nmap.org ) at 2013-09-06 14:31 CEST
Nmap scan report for 192.168.178.31
Host is up (0.030s latency).
Not shown: 65533 closed ports
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 7.91 seconds
```

Beispiel 7-3
Nmap zeigt neben den beiden aktiven Diensten geschlossene Ports.

Aktivieren Sie jetzt die Firewall durch Aufruf von `./fw_up.sh`. Per `iptables -L` überprüfen Sie noch einmal den Erfolg:

```
# ./fw_up.sh
# iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere            tcp dpt:www
ACCEPT     tcp  --  anywhere             anywhere            tcp spt:www
ACCEPT     udp  --  anywhere             anywhere            udp spt:domain
ACCEPT     tcp  --  anywhere             anywhere            tcp dpt:ssh
ACCEPT     tcp  --  anywhere             anywhere            tcp spt:ssh
ACCEPT     udp  --  anywhere             anywhere            udp spt:ntp
ACCEPT     udp  --  anywhere             anywhere            udp dpt:ntp
ACCEPT     icmp --  anywhere             anywhere

Chain FORWARD (policy DROP)
target     prot opt source               destination

Chain OUTPUT (policy DROP)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere            tcp spt:www
ACCEPT     tcp  --  anywhere             anywhere            tcp dpt:www
ACCEPT     udp  --  anywhere             anywhere            udp dpt:domain
```

```
ACCEPT     tcp  --  anywhere            anywhere            tcp spt:ssh
ACCEPT     tcp  --  anywhere            anywhere            tcp dpt:ssh
ACCEPT     udp  --  anywhere            anywhere            udp dpt:ntp
ACCEPT     udp  --  anywhere            anywhere            udp spt:ntp
ACCEPT     icmp --  anywhere            anywhere
#
```

Wenn Sie jetzt noch einmal `nmap` aufrufen, zeigt es anstelle der geschlossenen Ports *gefilterte* Ports an. Die Firewall ist tatsächlich aktiv (Beispiel 7-4).

Beispiel 7-4 *Nmap zeigt neben den beiden aktiven Diensten gefilterte Ports.*

```
quade@felicia:~$ nmap -p 1-65535 192.168.178.31

Starting Nmap 5.21 ( http://nmap.org ) at 2013-09-06 14:33 CEST
Nmap scan report for 192.168.178.31
Host is up (0.00096s latency).
Not shown: 65533 filtered ports
PORT   STATE SERVICE
22/tcp open  ssh
80/tcp open  http

Nmap done: 1 IP address (1 host up) scanned in 104.81 seconds
```

Systemintegration der Firewall

Wenn die Firewall wie gewünscht funktioniert, muss sie noch in das eingebettete System integriert werden, sodass sie beim Booten aktiviert wird. Das wird hier für das System aus Kapitel 4 gezeigt. Dazu legen wir die Dateien (`S45firewall`, `fw_open.sh`, `fw_up.sh`) in das Verzeichnis `~/embedded/raspi/userland/target/`. Die Datei `~/embedded/raspi/scripts/postbuild.sh` muss nun noch um die folgenden Zeilen ergänzt werden:

```
# MARK E: Firewall
echo "Firewall..."
install -m 755 ../userland/target/S45firewall $1/etc/init.d/
install -m 755 ../userland/target/fw_open.sh $1/usr/sbin/
install -m 755 ../userland/target/fw_up.sh $1/usr/sbin/
```

Generieren Sie das System neu. War alles richtig, ist die (private) Firewall aktiv. In diesem Abschnitt haben Sie nur die rudimentären Funktionen der Linux-Firewall kennengelernt. Weitergehende Eigenschaften wie NAT (network address translation) oder »stateful inspection« finden Sie beispielsweise unter [linuxfw] beschrieben.

Wenn Sie Forwarding-Rules benötigen, muss im Linux-Kernel das Forwarding erst noch aktiviert werden. Dazu muss das Firewallskript um die folgende Zeile ergänzt werden:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

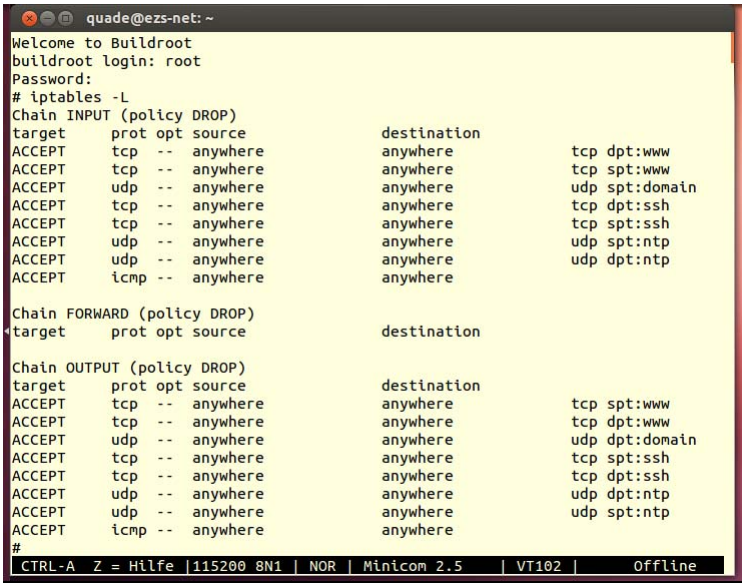


Abb. 7-5
Aktivierte Firewall auf dem Raspberry Pi

Beispiel 7-5 zeigt noch einmal die Befehlssequenz, mit der auf dem Buildroot-System eine Firewall installiert wird. Die notwendigen Dateien finden Sie auch wieder auf der Webseite zum Buch (Tabelle 7-3).

```
quade@felicia:~/embedded> cd raspi/userland/target/
quade@felicia:~/embedded/raspi/userland/target> gedit fw_open.sh
quade@felicia:~/embedded/raspi/userland/target> gedit fw_up.sh
quade@felicia:~/embedded/raspi/userland/target> gedit S45firewall
quade@felicia:~/embedded/raspi/userland/target> cd ../../scripts
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh
    # Installation der Target-Dateien
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> # Pfadvariable setzen
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # [Package Selection for the target][Networking applications][iptables]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
# Raspberry Pi booten
```

Beispiel 7-5
Befehlssequenz zur Installation der Firewall

Tabelle 7-3 Dateien für die Firewallfunktionalität

Dateiname	Verzeichnis auf dem Entwicklungssystem	Verzeichnis auf dem Zielsystem	Bedeutung
S45firewall	userland/target/	/etc/init.d/	Startskript Firewall
fw_open.sh	userland/target/	/usr/sbin/	Skript zur Deaktivierung der Firewall
fw_up.sh	userland/target/	/usr/sbin/	Skript zur Aktivierung der Firewall
postbuild.sh	scripts/		Postbuild-Skript

7.1.2 Intrusion Detection and Prevention

Bei der automatisierten Angriffs- und Einbruchserkennung (Intrusion Detection) werden zwei grundsätzliche Varianten unterschieden: eine hostbasierte und eine netzwerkbasierte. Während erstere den einzelnen Rechner überwacht und entsprechend auf dem zu überwachenden Rechner in Form eines Rechenprozesses aktiv ist, wird die netzwerkbasierte Variante wie eine Firewall vor die zu überwachenden Netzkomponenten gesetzt. Dort hört sie den Netzverkehr mit und sucht nach Anomalien. Heute übernimmt zunehmend die Firewall selbst die Aufgabe, über das Netzwerk Einbrüche zu identifizieren. Das bietet nämlich zugleich die Möglichkeit, bei einem identifizierten Angriff Gegenmaßnahmen durchzuführen. Dazu wird beispielsweise automatisch eine Firewallregel generiert, die den Datenverkehr von und zum angreifenden System oder aber auch nur den betroffenen Dienst (Port) sperrt (IPS, Intrusion-Prevention-System).

Bei einem hostbasierten Intrusion-Detection-System werden zur Angriffs- oder Einbruchserkennung regelmäßig Logdateien und aktive Rechenprozesse überwacht. Darüber hinaus werden wichtige System- und Anwenderdateien über Hashwerte gesichert. In regelmäßigen Abständen werden die Hashwerte neu berechnet und mit den Hashwerten abgeglichen, die zu einem unkompromitierten Zeitpunkt erfasst wurden. Zwar bietet gerade diese Überwachung eine sehr gute Erkennungsrate, aber leider ist sie in der Handhabung kompliziert. Zum einen werden einige Systemdateien regelmäßig durch das System selbst aktualisiert, zum anderen müssen die Vergleichs-Hashwerte nach jedem Update und/oder jeder Softwareinstallation neu erfasst werden.

Für unser eingebettetes System gibt es keine vorkonfektionierten IDS- oder IPS-Pakete.

7.1.3 Rechtevergabe

Um auf dem Embedded Linux eine Userverwaltung mit Username/Passwort-Authentifizierung zu etablieren, sind die folgenden Schritte notwendig:

1. Im Verzeichnis `~/embedded/raspi/scripts/` ein Skript `modifyshadow.sh` erstellen, das im Terminal ein Passwort abfragt und selbiges mit einem zufälligen »Salz« hasht und in die Target-Datei `shadow` ablegt.
2. Das Skript `postbuild.sh` so modifizieren, dass `modifyshadow.sh` für die installierten User aufgerufen wird.
3. Das Hashverfahren SHA-512 zur Codierung des Passwortes in der Buildroot-Konfiguration aktivieren
4. Das Hashverfahren SHA-512 in der Busybox-Konfiguration aktivieren
5. Das Hashverfahren SHA-512 in der uclibc-Konfiguration aktivieren
6. Das Buildroot-System durch Eingabe von `make` neu generieren. Darauf achten, dass dazu die Umgebungsvariable `PATH` richtig gesetzt ist, und nach Aufforderung die Passwörter für die User »root« und »default« eingeben.

Linux-Systeme verwenden klassischerweise ein einfaches Rechtemodell zur Zugriffskontrolle, das als »Discretionary Access Control (DAC)« bezeichnet wird. In diesem Modell gibt es drei Benutzerklassen, nämlich den Besitzer (owner), die Gruppe (group) und alle zusammen, die Welt (world). Jede Ressource, beispielsweise eine Datei oder ein Verzeichnis, ist genau einem Besitzer und einer Gruppe zugeordnet. Unter Linux lassen sich für diese Ressourcen Zugriffsrechte in Form von Attributen für die drei Benutzerklassen getrennt festlegen. Die Rechte selbst sind:

- Lesen
- Schreiben
- Ausführen respektive bei Verzeichnissen das Wechseln in ein Verzeichnis

Zusätzlich gibt es noch die Attribute »S-Bit« (für den Besitzer und für die Gruppe einer Datei) und »T-Bit« (Sticky Bit). Ist das Sticky Bit für ein Verzeichnis gesetzt, kann nur der Eigentümer eine in diesem Verzeichnis befindliche Datei löschen oder umbenennen.

Jedem Rechenprozess sind unter Linux drei User-IDs und drei Gruppen-IDs zugeordnet:

- Die »real user id«, kurz *uid* oder *ruid*, und die »real group id« (*gid*) .
- Die »effective user id«, kurz *euid*, und die »effective group id« (*egid*). Die *euid* wird für die eigentliche Autorisierung, also für die Entscheidung, ob der Zugriff auf eine Ressource (Datei) erlaubt ist oder nicht, herangezogen. Für die *egid* gilt Entsprechendes.
- Die »saved user id«, kurz *suid*, und die »saved group id« (*sgid*). In der *suid* wird die ID abgelegt, die der Prozess beim Starten mitbekommen hat. Dank dieser Sicherung kann bei gesetztem S-Bit programmgesteuert zwischen zwei IDs gewechselt werden. Für die *sgid* gilt Entsprechendes.

Wenn ein Rechenprozess per `fork()` erzeugt wird, erbt er sowohl die *ruid*, *euid* und *suid* als auch die *gid*, *egid* und *sgid* vom Elternprozess. Sobald der Rechenprozess den Systemcall `exec()` aufruft, um neuen Code auszuführen und damit eine andere Funktionalität zu haben, behält er die bisherigen IDs, es sei denn, das S-Bit ist gesetzt. In diesem Fall bekommt der Rechenprozess als *euid* und *suid* diejenige vom Besitzer (owner) des Codes beziehungsweise der Datei.

Da die Zugriffskontrolle auf der *euid* beziehungsweise *egid* basiert, kann ein Rechenprozess über die Modifikation dieser IDs die Zugriffe im Sinne des Entwurfsziels »Least Privilege« steuern. Dabei werden nie mehr Privilegien (Rechte) aktiviert als unbedingt notwendig. Zur Modifikation der IDs stehen unter Linux die Systemcalls `setresuid(uid_t ruid,uid_t euid,uid_t suid)` und `setresgid(gid_t rgid,gid_t egid,gid_t sgid)` zur Verfügung.

Ein Rechenprozess erhöht seine Privilegien, indem er die *euid* auf den Wert des privilegierten Users setzt. Er reduziert seine Privilegien durch Austausch der privilegierten *euid* und *egid* gegen diejenigen unprivilegierter User. Bei der Reduktion der Privilegien sind zwei Arten zu unterscheiden:

1. Um die Privilegien temporär zu reduzieren, speichert die Applikation eine Kopie der *euid* beziehungsweise *egid* in die *suid* beziehungsweise *sgid* ab und überschreibt die effective id mit derjenigen eines unprivilegierten Benutzers. Damit kann er später die effective id aus der saved id restaurieren.
2. Um die Privilegien dauerhaft zu reduzieren, überschreibt die Applikation die *ruid*, *euid* und *suid* mit einer der IDs eines unprivilegierten Users.

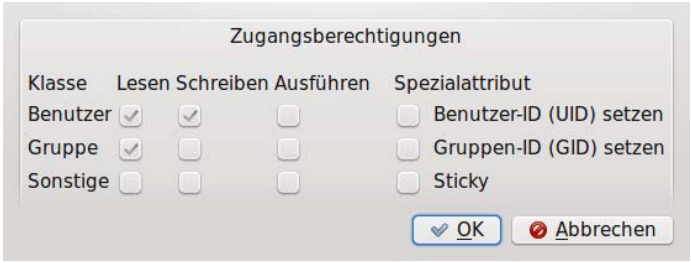


Abb. 7-6
Organisation der Standardzugriffsrechte

Anwendung im Embedded Device

In dem von Grund auf aufgebauten System ist bisher kein Usermanagement implementiert. Damit laufen sämtlichen Tasks mit Root-Rechten. Um das zu ändern, werden auf dem System die Dateien `/etc/passwd` und `/etc/shadow` benötigt. Erstere nimmt die Namen der User, die zugehörigen IDs, den Namen des Heimatverzeichnisses und das beim Einloggen zu startende Programm (typischerweise eine Shell) auf. Letztere speichert zum Usernamen den Passwort-Hash.

Neben dem obligatorischen User »root« wird ein User »nobody« benötigt, der beispielsweise wie unter Ubuntu die ID 65534 oder wie im Buildroot-System die ID 99 bekommt. Bei Applikationen, die beim Systemstart gestartet werden sollen und keine besonderen Rechte benötigen, sollte dann per `chown` als Owner der User »nobody« eingetragen und mit `chmod +s` das S-Bit gesetzt werden. Damit läuft die Task nach dem Starten unter der Kennung »nobody«.

Unter Buildroot ist ein rudimentäres Usermanagement aktiviert. Allerdings ist die Konfiguration sehr schwach ausgelegt. So können sich die beiden in der Datei eingetragenen User »root« und »default« ohne Passwort anmelden. Haben Sie in der Buildroot-Konfiguration ein Passwort für den User »root« vergeben, wird dieses zum einen auf dem Entwicklungsrechner in der Datei `.config` im Klartext abgespeichert, zum anderen wird es mit dem als nicht mehr zeitgemäß geltenden MD5-Verfahren gehasht. Es ist also erforderlich, auf ein sichereres Hashverfahren wie beispielsweise SHA-512 umzusteigen.

Um SHA-512 für das Hashen der Passwörter zu verwenden ist Folgendes zu tun:

1. Wechseln Sie in das Buildroot-Hauptverzeichnis und rufen Sie `make menuconfig` auf. Unter [System configuration][Passwords encoding ()] wählen Sie SHA-512 aus. Damit wird das unter Umständen konfigurierte Defaultpasswort mit diesem Verfahren gehasht. Speichern Sie die Konfiguration beim Verlassen ab.

2. Mit einem mit SHA-512 gehashten Passwort können Sie sich nur einloggen, wenn auch Busybox darauf eingerichtet ist. Rufen Sie al-

so aus dem Buildroot-Hauptverzeichnis `make busybox-config` auf. Wählen Sie hier [Login/Password Management Utilities][Enable SHA/512 crypt functions] aus.

3. Damit Programme wie beispielsweise der SSH-Daemon `dropbear` ebenfalls mit dem sichereren Hashverfahren arbeiten können, muss es auch von `uClibc` unterstützt werden. Das ist standardmäßig nicht der Fall. Rufen Sie daher aus dem Buildroot-Hauptverzeichnis `make uclibc-menuconfig` auf. Unter [Advanced Library Settings][libcrypt support] finden Sie die beiden Menüpunkte [libcrypt SHA256 support] und [libcrypt SHA512 support], die beide aktiviert werden.
4. Nach der Konfiguration rufen Sie aus dem Buildroot-Hauptverzeichnis `make` auf, um damit das neue System zu generieren.

Damit ist zwar das als zurzeit sicher geltende Hashverfahren SHA-512 aktiviert, der User »default« verfügt jedoch immer noch nicht über ein Passwort und das Rootpasswort ist weiterhin im Klartext in der Konfiguration hinterlegt. Dies soll noch geändert werden. Dazu müssen automatisiert in der Datei `buildroot-2013.05/target/etc/shadow` Änderungen vorgenommen werden.

Klartext-Passwörter werden, wie auch in Abschnitt 7.3.4 beschrieben, über ein Hashverfahren in einen Hashwert umgewandelt. Der Hashwert hat die Eigenschaft, dass er keine Rückschlüsse auf den Passwort-Klartext zulässt, aber aus diesem eindeutig generiert wird. Kommt ein Angreifer in den Besitz des Hashwertes, bleibt ihm nur übrig, alle möglichen Klartext-Passwörter zu hashen und das Ergebnis mit seiner Beute zu vergleichen. Bei Übereinstimmung kennt er das Klartext-Passwort. Natürlich gibt es längst umfangreiche Datenbanken, die diverse Klartext-Passwörter gehasht und dann die Hashwerte abgespeichert haben. In einem solchen Fall braucht der Angreifer nur in der Datenbank den Hashwert suchen und, falls vorhanden, das zugehörige Klartext-Passwort auslesen.

Um diesen Brute-Force-Angriff zu erschweren, wird die Wahrscheinlichkeit, dass sich das Passwort in einer derartigen Datenbank befindet, gesenkt. Das ist möglich, indem dafür gesorgt wird, dass das Passwort sehr viele Zeichen umfasst und diese zusätzlichen Zeichen typischerweise auch noch zufällig sind. Der Hashwert eines Passwortes wie »hallo« ist sicher in einer Internet-Datenbank vorhanden, der Hashwert von »hallo12zji,;#9za?« mit hoher Wahrscheinlichkeit nicht. Der Teil, der das Wörterbuch-Passwort verlängert, wird als »Salz« bezeichnet. Damit das Verfahren wirklich funktioniert, muss jeder User sein eigenes Salz haben, was folglich irgendwo abgespeichert sein muss.

In Linux wird das Salz zusammen mit dem Passwort-Hash in der Datei `/etc/shadow` abgespeichert. Kennt ein Angreifer den Passwort-

Hash, kennt er damit mit hoher Wahrscheinlichkeit auch das Salz. Aber er kann nicht auf die Datenbanken zurückgreifen, sondern muss sämtliche Kombinationen durchprobieren. Ist das Klartext-Passwort gut genug gewählt, wird er das in absehbarer Zeit nicht schaffen.

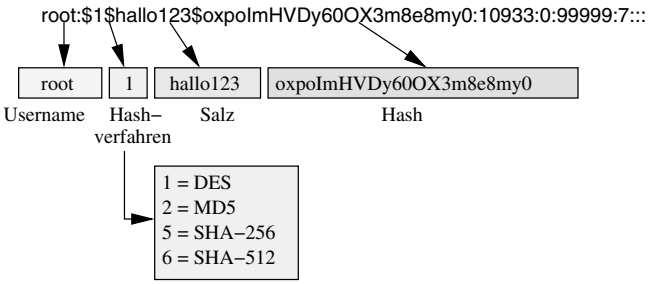


Abb. 7-7
Aufbau eines Eintrags in der Datei shadow

Abbildung 7-7 zeigt den Aufbau eines Eintrags in der Datei `/etc/shadow/`. Die einzelnen Felder sind per Doppelpunkt voneinander getrennt. Das zweite Feld enthält alle Informationen zum Passwort-Hash. Dazu gehört

- das Hashverfahren, beispielsweise MD5, SHA-256 oder SHA-512,
- das Salz sowie
- der sich aus Klartext-Passwort und Salz ergebende Hashwert.

Die einzelnen Informationen sind durch ein Dollarzeichen (»$«) getrennt, wobei das verwendete Hashverfahren durch einen Zahlenwert codiert ist. Eine sechs beispielsweise steht für SHA-512.

Auf einem Ubuntu lässt sich die Information zum Passwort-Hash mithilfe des Kommandos `mkpasswd` für die Nutzung der der Datei `shadow` generieren. `Mkpaswd` bekommt mit der Option »-m« das Hashverfahren übergeben. Zur Auswahl stehen »des«, »md5«, »sha-256« und »sha-512«. Daneben geben Sie beim Aufruf von `mkpasswd` das Salz und das Klartext-Passwort an. Beispiel 7-6 zeigt den Aufruf mit dem Passwort »hallo« und dem Salz »hallo123«.

Beispiel 7-6
Mit mkpasswd lassen sich geeignete Passwort-Hashes generieren

```
quade@felicia:~/embedded> mkpasswd -m help
Verfügbare Methoden:
des   standard 56 bit DES-based crypt(3)
md5   MD5
sha-256   SHA-256
sha-512   SHA-512
quade@felicia:~/embedded> mkpasswd -m sha-512 hallo hallo123
$6$hallo123$1nkKdod4RRdJRO.SI.fbwoKXVR6IJUq9IsdsLlJev.jKj6vxQIMjVAMtZLyt1qQCGlTQ
YGIKnCgrRKGahrumM.
```

Beispiel 7-7 *Skript zur Generierung von Passwort-Hashes <modifyshadow.sh>*

```
#!/bin/bash
# usage: modifyshadow.sh username targetdir
# $1 = username
# $2 = targetdir

USERNAME=$1
TARGETDIR=$2

# read in password; don't show it on the commandline
stty -echo
read -p "Password for $1: " passw; echo
stty echo

salt=`< /dev/urandom tr -dc A-Za-z0-9 | head -c8;echo;`
passwordhash=`mkpasswd -m sha-512 $passw $salt`

sed -i -e "s#^$USERNAME:[^:]*:#$USERNAME:$passwordhash:#" \
   $TARGETDIR/etc/shadow
```

Um unser Buildroot-Image mit einem eigenen Passwort für »root« und »default« zu versehen, schreiben wir ein Skript, das für den als Parameter übergebenen User einen Passwort-Hash generiert (siehe Beispiel 7-7). Das Skript legen wir unter dem Namen `modifyshadow.sh` ins Verzeichnis `scripts/` ab. Beim Aufruf fragt es das Klartext-Passwort ab, das quasi blind eingegeben werden muss, damit es von keinem mitgelesen werden kann. Dazu dient `stty -echo`. Das Skript generiert danach ein zufälliges Salz und modifiziert die Einträge in der Shadow-Datei des Targets. Das Skript selbst rufen wir aus dem Skript `postbuild.sh` heraus auf. Dazu erweitern wir dieses um die folgenden Zeilen:

```
# MARK F: Usermanagement
echo "Usermanagement..." # generiert gute Passwort-Hashes für /etc/shadow
../scripts/modifyshadow.sh root $1
../scripts/modifyshadow.sh default $1
```

Beispiel 7-8 zeigt die Befehlssequenz, um das Buildroot-System um ein Usermanagement zu erweitern.

Beispiel 7-8 *Befehlssequenz zur Etablierung eines Usermanagements*

```
quade@felicia:~/embedded> cd raspi/scripts
quade@felicia:~/embedded/raspi/scripts> gedit modifyshadow.sh
...
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh
    # ../scripts/modifyshadow.sh root $1
    # ../scripts/modifyshadow.sh default $1
quade@felicia:~/embedded/raspi/scripts> cd ../buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # [System configuration][Passwords encoding()] sha-512
quade@felicia:~/embedded/raspi/buildroot-2013.05> make busybox-menuconfig
```

```
    # [Login/Password Management Utilities][Enable SHA/512 crypt functions]
quade@felicia:~/embedded/raspi/buildroot-2013.05> make uclibc-menuconfig
    # [Advanced Library Settings][libcrypt support]
    # [Advanced Library Settings][libcrypt SHA256 support]
    # [Advanced Library Settings][libcrypt SHA512 support]
quade@felicia:~/embedded/raspi/buildroot-2013.05> # PATH=$PATH:...
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
    # Passwort für den Root-Login eingeben
    # Passwort für den User Default eingeben
# Raspberry Pi booten
```

7.1.4 Ressourcenverwaltung

Da es problematisch werden kann, wenn eine wichtige Task notwendige Ressourcen nicht erhält, weil diese anderweitig verwendet werden, ist ein Ressourcenmanagement notwendig. Ressourcen sind auf einem eingebetteten System bekanntlich knapp, andererseits gibt es dort den Vorteil, dass das System relativ abgeschlossen ist und damit auch leichter abgesichert werden kann.

Grundsätzlich sind die folgenden Ressourcen zu verwalten:

- Hauptspeicher
- Hintergrundspeicher (Flash, Festplatte)
- Rechenleistung
- Netzwerk
- Sonstige Ressourcen wie Filedeskriptoren oder Zufallszahlen

Linux bietet eine Reihe unterschiedlicher Möglichkeiten, diese Ressourcen zu verwalten.

Hauptspeicher

Zusammen mit der Rechenleistung ist der Hauptspeicher eine der Ressourcen, die sehr leicht zum Fehlverhalten einer Applikation und damit des eingebetteten Systems führen können. Das hängt nicht zuletzt mit den Programmierern zusammen, die den Erfolg von dynamisch reserviertem Speicher per `malloc()` nicht überwachen. Problematisch ist darüber hinaus, dass eingebettete Systeme aus Ressourcen- und Performance-Gründen typischerweise kein Swap-Space verwenden; die Ressource Hauptspeicher ist damit noch eingeschränkter.

Der Programmierer einer Applikation analysiert in jedem Fall die Rückgabewerte einer Funktion `malloc()`. Darüber hinaus kann er eine als Prefault bezeichnete Technik einsetzen, bei der der benötigte Haupt-

speicher bereits zum Start der Applikation vorreserviert wird (siehe hierzu [QuMä2012, S. 153]).

Wird ein Teil des Hauptspeichers in Form eines RAM-Filesystems zur Datenablage verwendet, ist besondere Vorsicht angeraten. Falls die maximale Größe des Dateisystems nicht festgelegt ist, kann durch Ablage großer Daten der Hauptspeicher schnell geflutet werden.

Hintergrundspeicher

Vorsicht ist auch beim Umgang mit Hintergrundspeicher beziehungsweise Flash-Speicher geboten. Viele Dienste wie beispielsweise Webserver protokollieren Aktivitäten in Logdateien mit, die sich meist im Verzeichnis `/var/log/` befinden. Werden die Logdateien nicht regelmäßig gelöscht, ist zuweilen kein Platz mehr auf dem Flash-Speicher vorhanden. Das Überwachen und automatische Aufräumen der Logdateien kann man übrigens dem Programm `logrotate` überlassen. `logrotate` kann die Logdateien komprimieren, löschen oder aber auch per Mail verschicken. Buildroot bietet das Kommando zur Auswahl an.

Um einer Applikation Speicherplatz zu reservieren, gibt es noch weitere Techniken. Zum einen kann die Applikation den Speicherplatz im Vorhinein, also beim erstmaligen Start, reservieren. Zum anderen könnte auf dem Hintergrundspeicher eine eigene Partition angelegt werden, auf der die kritischen Daten unabhängig (und unbemerkt) von den anderen Programmen abgelegt werden.

Neben `logrotate` und der eigenen Partition beziehungsweise der Vorreservierung gibt es als Drittes noch die Möglichkeit, Quotas einzusetzen. Quotas müssen im Kernel aktiviert werden und beschränken den Hintergrundspeicher, den einzelne User für sich verwenden dürfen.

Rechenleistung

Auch bei der Rechenleistung gibt es mehrere Methoden, Missbrauch einzuschränken:

- Prioritätenvergabe
- Firewalling
- Überwachung der Rechenleistung mit `cpuload`
- Watchdog

Grundsätzlich sollten wichtige Tasks mit einer Realzeitpriorität versehen werden. Neben dem Kommandozeilenwerkzeug `chrt` ist der Programmcode hierfür in Abschnitt 5.2.2 vorgestellt worden.

Eine bekannte Angriffsart aus dem Bereich Sabotage ist der Distributed-Denial-of-Service-Angriff (DDoS-Attacke). Hierbei wird eine

Komponente so sehr mit Anfragen überschüttet, dass diese nur noch mit den Anfragen beschäftigt ist und den normalen Aufgaben nicht mehr nachkommt. Ein DDoS-Angriff ist sehr schwer abzuwehren. Eine in weiten Bereichen wirksame Methode besteht darin, die Anfragen möglichst frühzeitig, bevor sie intensiv durch das System bearbeitet werden, abzufangen. Diese Aufgabe kommt der Firewall zu (siehe Abschnitt 7.1.2).

Generell ist es eine gute Idee, die Last auf dem Rechner in regelmäßigen Abständen zu untersuchen. Hierzu kann das Programm `cpuload` helfen. Wird eine hohe Lastsituation erkannt, wird der verursachende Thread ausfindig gemacht und beendet (`kill`).

Sinnvoll ist es auch, einen Watchdog zu aktivieren. Ein Watchdog ist ein häufig in Hardware realisierter Rückwärtszähler. Wird der Zählerwert null erreicht, löst der Watchdog einen Reset aus, das System bootet neu.

Auch der Raspberry Pi besitzt einen Hardware-Watchdog. Dieser muss im Kernel aktiviert werden (siehe Abb. 7-8). Außerdem muss ein Dienst aktiv sein, der den Watchdog regelmäßig auf den Ursprungswert zurücksetzt, sodass dieser im Normalfall keinen Reset auslöst.

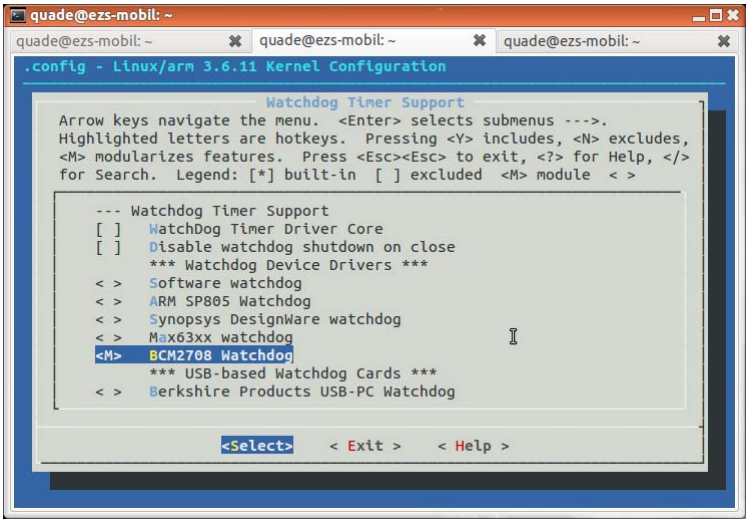


Abb. 7-8
Auswahl des Watchdog-Treibers bei der Kernelkonfiguration

Der Watchdog-Treiber wird durch folgendes Kommando geladen:

```
pi@raspberry:~$ sudo modprobe bcm2709_wdog
```

War dies erfolgreich, stellt der Treiber die Gerätedatei `/dev/watchdog` zur Verfügung. Wird auf diese Gerätedatei per `open()` zugegriffen, ist der Watchdog *scharf*. Jetzt muss in regelmäßigen Abständen eine Null auf die Gerätedatei geschrieben werden, um ein Reboot zu verhindern. Ty-

pischerweise gibt es hierfür 30 Sekunden Zeit. Busybox bringt mit dem Kommando `watchdog` die hierfür benötigte Funktionalität mit. Um beispielsweise die Timeout-Zeit auf 30 Sekunden und das Zurücksetzen des Zählers auf alle 10 Sekunden einzustellen, geben Sie das folgende Kommando ein:

```
# watchdog -T 30 -t 10 /dev/watchdog
```

Mit diesem Kommando muss also spätestens nach 30 Sekunden ein Lebenszeichen erfolgen, um den Reset zu verhindern. Das Lebenszeichen selbst erfolgt alle 10 Sekunden.

Buildroot bietet unter [Package Selection for the target] den Auswahlpunkt [Install the watchdog deamon startup script] mit dem Unterpunkt [Delay between reset] an. Hier kann man direkt beim Booten den Watchdog aktivieren und die Timeout-Zeit konfigurieren.

Alternativ zu dem Busybox-Programm `watchdog` zeigt Beispiel 7-9 ein C-Programm, das den Watchdog aktiviert und regelmäßig zurücksetzt. Der Code lässt sich leicht in die eigene, kritische Applikation übernehmen, sodass eine Fehlfunktion (Absturz) dieser kritischen Applikation direkt in einen System-Reset mündet. In Zeile 17 ist die Konfiguration des Zeitintervalls über ein IO-Control demonstriert. Sie können das Programm per »make CC=arm-linux-cc watchdog_service« cross-kompilieren, dann auf das Target kopieren und dort per `./watchdog_service &` starten.

Beispiel 7-9 *Ein Watchdog-Server überwacht die Vitalität des Systems <watchdog_service.c>.*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <linux/watchdog.h>

int main(int argc, char **argv, char **envp)
{
   int fd = open("/dev/watchdog", O_WRONLY);
   int ret = 0;
   int timeout = 45;

   if (fd == -1) {
      perror("watchdog");
      exit(EXIT_FAILURE);
   }
   ret=ioctl(fd, WDIOC_SETTIMEOUT, &timeout);
   printf("The timeout was set to %d seconds.\n", timeout);
   while (1) {
      ret = write(fd, "\0", 1);
      if (ret != 1) {
         ret = -1;
```

```
            break;
        }
        sleep(10);
    }
    close(fd);
    return ret;
}
```

Weitere Informationen zu der Watchdog-Funktionalität und insbesondere dazu, wie ein eigener Watchdog-Treiber erstellt wird, finden Sie unter [QuKu2009].

Netzwerk

Das Sicherstellen von ausreichender Netzwerkbandbreite wird im Wesentlichen durch die Firewall gewährleistet (Abschnitt 7.1.1).

Darüber hinaus ist aber auch in diesem Bereich ein Monitoring möglich. Buildroot bietet hierzu beispielsweise unter [Package Selection for the target][System tools] das Paket `bwm_ng` an. Dieses kann nicht nur den Netzverkehr, sondern zusätzlich auch die Zugriffe auf den Hintergrundspeicher überwachen.

Sonstige Ressourcen

Ein Linux-System stellt eine ganze Reihe weiterer Ressourcen zur Verfügung, beispielsweise die Anzahl offener Dateien oder die Anzahl der Tasks, die ein einzelner User starten darf. Das Kommando `ulimit` zeigt die diesbezüglichen Einstellungen an beziehungsweise ermöglicht auch die Konfiguration. Die Ausgabe auf dem Raspberry Pi zeigt, dass es kaum Beschränkungen bei den Ressourcen gibt:

```
# ulimit -a
-f: file size (blocks)             unlimited
-t: cpu time (seconds)             unlimited
-d: data seg size (kb)             unlimited
-s: stack size (kb)                8192
-c: core file size (blocks)        0
-m: resident set size (kb)         unlimited
-l: locked memory (kb)             64
-p: processes                      3376
-n: file descriptors               1024
-v: address space (kb)             unlimited
-w: locks                          unlimited
-e: scheduling priority            0
-r: real-time priority             0
```

Mit dem Kommando `ulimit` lassen sich die Ressourcen für den jeweils aktuellen Prozess, also die gerade aktive Shell, beschränken oder auch freigeben. Auf Desktop- oder Serversystemen findet sich unter `/etc/security/` eine Datei `limits.conf`, über die die Beschränkungen systemweit für alle Benutzer oder auch für einzelne Gruppen oder User konfiguriert werden. Dieser Mechanismus wird aber weder von Busybox noch von Buildroot unterstützt.

Allerdings gilt `ulimit` ohnehin als veraltet. Linux bietet alternativ den Systemcall `prlimit()` an, mit dem sich auch die Limits anderer Prozesse modifizieren lassen. Dieses wird zwar von uClibc nicht direkt unterstützt, aber das Paket `util-linux` (gehostet auf [http://www.kernel.org]) bringt ein Kommando namens `prlimit` mit. Das Paket `util-linux` wiederum kann über Buildroot unter [Package Selection for the target][System tools][util-linux] ausgewählt und auf dem eingebetteten System installiert werden. Voraussetzung hierfür ist wiederum, dass die Toolchain mit WCHAR- und Largefile-Support generiert wird ([Toolchain][Enable large file (files > 2 GB) support] und [Toolchain][Enable WCHAR support]).

7.1.5 Entropie-Management

Gerade für den Bereich Verschlüsselung sind gute Zufallszahlen unabdingbar. Diese dürfen keinesfalls berechnet geschweige denn berechenbar sein, sondern müssen richtig zufällig sein. Linux unterstützt die Erzeugung von Zufallszahlen im Kernel. Dazu werden verschiedene Zufallsquellen, vor allem Interrupts von der Tastatur oder der Festplatte, ausgewertet und in einen Pool von Zufallszahlen über Hashfunktionen eingepflegt. Über Statistiken stellt Linux fest, ob ein Zufallswert wirklich zufällig ist oder eben doch nicht. Ist der eingepflegte Wert ausreichend zufällig, wird die Entropie hochgesetzt. Die Entropie spiegelt damit den Gehalt an echtem Zufall im Pool wider.

Für den Pool der Zufallszahlen liefert Linux mit den Gerätedateien `/dev/random` und `/dev/urandom` zwei Interfaces. Jedes Mal, wenn eine Zufallszahl angefordert wird, reduziert Linux die Entropie. Sollte keine Entropie mehr im Pool vorhanden sein, eine Applikation aber eine Zufallszahl über `/dev/random` anfordern, legt Linux die Applikation schlafen. Ist wieder ausreichend Entropie vorhanden, wird die Applikation wieder aufgeweckt. Demgegenüber liefert `/dev/urandom` auch dann eine Zufallszahl zurück, wenn keine Entropie mehr im Pool vorhanden ist. Die Applikation läuft also in jedem Fall weiter.

Für den Anwender hat das die folgenden Konsequenzen:

- Lesen Sie nicht mehr Zufallszahlen von `/dev/random` als unbedingt notwendig. Auch die über `/dev/urandom` gewonnenen Zufallszahlen dürften den meisten Anforderungen genügen.
- Wenn es in Ihrem System beziehungsweise in Ihrer Anwendung Zufallselemente gibt, können Sie mit diesen den Zufallszahlengenerator füttern und so für Entropie sorgen. Wenn Sie einen Treiber schreiben, der Interrupts verwendet, die zu einem zufälligen Zeitpunkt ausgelöst werden, koppeln Sie den Interrupthandler ebenfalls mit dem Zufallszahlengenerator.

7.1.6 ASLR und Data Execution Prevention

Address Space Layout Randomization (ASLR)

Im Adressraum einer Applikation bringt der Linker die einzelnen Segmente wie Code, Daten, Heap oder Stack unter. Legt er dabei die Segmente immer an die gleichen Adressen, setzt er also auf eine starre Aufteilung des Adressraums, ermöglicht das Angreifern, Rücksprungadressen auf dem Stack vorherzusehen und für die eigenen Zwecke »sinnvoll« zu überschreiben (Buffer-Overflow-Angriff). Aus diesem Grund bringt Linux mehr Variation in die Lage von Stack, Heap und der Region, in der sich typischerweise gemeinsam genutzte Bibliotheken befinden. Bei einer zufallsbedingten Adressvergabe kommen Angreifer nur noch mit »Ausprobieren« weiter. Raten sie daneben, wird die angegriffene Applikation mit hoher Wahrscheinlichkeit abstürzen.

Sobald ASLR für eine Task aktiviert ist, verschiebt der Betriebssystemkern die Position des Stacks und der Mmap-Region durch Einbau einer Zufallslücke, in der sich insbesondere die gemeinsam genutzten Bibliotheken (DSO) befinden. Wichtig zur Beurteilung des damit erreichten Sicherheitsgewinns ist die Analyse der Entropie. Die Entropie wird in Bits angegeben und ist ein Maß für Variationsbreite (Anzahl der unterschiedlichen Adresslagen).

ASLR kann Buffer-Overflow-Angriffe zwar nicht gänzlich verhindern, aber doch deutlich erschweren. ASLR ist unter Linux typischerweise aktiviert.

Data Execution Prevention (DEP)

Unter Data Execution Prevention versteht man eine Technologie, bei der das Ausführen von Code (Programmen), der auf dem Stack liegt, unterbunden wird. Der Stack, der Speicherbereich einer Applikation, der für lokale Variablen, Aufruf- und Returnparameter vorgesehen ist, wird von bösartiger Software bei einem Buffer-Overflow-Angriff dazu

genutzt, um dort anstelle normaler Daten ausführbaren Code abzulegen und ausführen zu lassen. DEP erschwert das — typischerweise hardwareunterstützt — signifikant und sollte daher immer aktiviert sein.

ARM-Prozessoren, wie beispielsweise der Prozessor auf dem Raspberry Pi, bringen ab ARMv6 eine Unterstützung in Form des *eXecute Never Bits* (XN-Bit) mit, die von Linux auch genutzt wird. x86-Prozessoren demgegenüber stellen DEP standardmäßig nur in der 64-Bit-Version zur Verfügung. Intel nennt das zugehörige Flag No eXecute (NX-Bit), AMD execution disable (xD-Bit). Auf 32-Bit-x86-Prozessoren ist das NX- oder xD-Bit nur dann nutzbar, wenn PAE (Physical Address Extension) aktiviert wird. PAE erweitert den physischen Adressraum auf 64 GByte. Um also DEP auf einer 32-Bit-x86-Plattform zu nutzen, ist der Kernel wie in Beispiel 7-10 gezeigt zu konfigurieren.

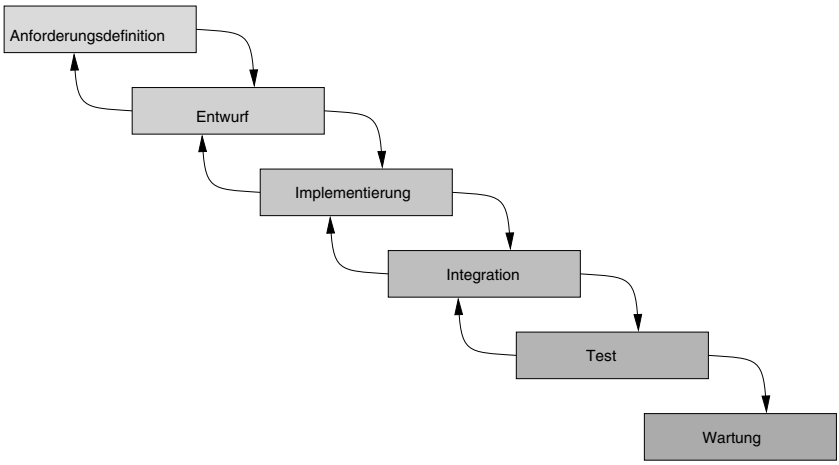
Beispiel 7-10 *Kernelkonfiguration zur Aktivierung von DEP*

```
quade@felicia:/embedded/qemu/linux-3.10.9$ make menuconfig
[Processor type and features]
[High Memory Support]
[64 GByte]
```

7.2 Entwicklungsprozess

Die Entwicklung eines jeden Systems und damit auch eines eingebetteten Systems läuft in Phasen ab, die abhängig vom Entwicklungsmodell mehrfach durchlaufen werden (Abb. 7-9, [wasserfall]). In allen Phasen sind dabei Aspekte der Angriffssicherheit zu beachten.

Abb. 7-9 *Entwicklungsphasen*



Definition der Anforderungen (Requirement Specification)

Bei den Anforderungen sind Maßnahmen zur Abwehr von Angriffen zu berücksichtigen. Zu den festzulegenden Anforderungen gehören unter anderem:

- Sämtliche Komponenten müssen Open Source sein
- Unterstützte Systemkommandos
- Aufbau der Rechteverwaltung
- Anzahl und Namen der Logins
- Überwachung des Netzverkehrs
- Verwendete Netzwerkdienste (und deren Ports)
- Eingesetzte Verschlüsselungsverfahren
- ...

Insbesondere die Forderung, dass nur Open Source Software eingesetzt wird, erscheint radikal. IT-Security hat aber nichts mit Vertrauen zu tun, viel mit Misstrauen. Der PRISM-Skandal hat schließlich deutlich gezeigt, dass nicht nur die Hacker, sondern Regierungen an der Kontrolle von Daten und Systemen größtes Interesse zeigen. Closed Software kann daher nicht sicher sein. Open Source Software ist zwar nicht per se sicherer, hat aber immerhin das Potenzial dazu.

Ist die Forderung nach Open Source Software aus welchen Gründen auch immer nicht für alle Systemteile einzuhalten, sollte darauf beim Entwurf der funktionsbestimmenden Applikation Rücksicht genommen werden und sämtliche Sicherheitsfunktionen, beispielsweise die verschlüsselte Datenablage oder verschlüsselte Kommunikation, müssen in der Applikation selbst implementiert werden.

Entwurf (Design)

Wer ein eingebettetes System baut, entwirft nicht nur die funktionsbestimmende Applikation, sondern auch das System selbst. Damit muss der Entwurf die bereits diskutierten Themen wie Firewalling, Rechtevergabe, Ressourcenverwaltung oder Entropie-Management berücksichtigen. Die jeweiligen Festlegungen sind zu dokumentieren und Testfälle aufzustellen.

In Abschnitt 7.3 wird der sicherheitsgerichtete Entwurf der Applikation diskutiert.

Implementierung

Für die Implementierung gelten unter anderem die folgenden Grundsätze:

- Fehlersituationen (Festplatte voll, Verbindungsabbruch bei tcp/ip,... müssen vollständig behandelt werden.)
- Sämtliche Returncodes von Funktionen sind zu überprüfen.
- Abfangen sämtlicher Signale bei Skripten und Programmen.
- »Vorzeichenlose« und »vorzeichenbehaftete« Datentypen sind bewusst zu verwenden.
- Bei der Programmierung ist grundsätzlich Vorsicht beim Type-Casting geboten.
- Berechnen Sie niemals selbst Zufallszahlen, sondern entnehmen Sie diese aus `/dev/random` oder `/dev/urandom`.
- Sämtliche Eingaben sind auf Länge (zu kurz und zu lang) zu überprüfen.
- Sämtliche Eingaben sind auf nicht darstellbare/unbrauchbare Zeichen hin zu überprüfen und gegebenenfalls anzupassen (Maskieren von Steuerzeichen, die ansonsten eine Code-Injection ermöglichen könnten).

Berücksichtigen Sie außerdem sämtliche Warnungen, die der Compiler während des Übersetzungsvorgangs ausgibt. Der Compiler ist per Option `-Wall` auf höchste Warnstufe zu setzen und sämtliche dabei ausgegebenen Warnungen sind aufzulösen.

Unabhängig davon ist eine dynamische Codeanalyse durchzuführen. Bei dieser wird beispielsweise das Memory Management mit Werkzeugen wie `electric fence`, `DUMA` oder `Dmalloc` überwacht. Setzen Sie außerdem die Compiler-Option `-fstack-protector-all` oder zumindest `-fstack-protector`. Diese Option erschwert das Überschreiben des Stacks.

Neben der dynamischen Codeanalyse ist eine statische Codeanalyse sinnvoll. Hier helfen die beiden leicht anzuwendenden Programme `rats` und `flawfinder`. Diese untersuchen den Code unter anderem auf Verwendung von Funktionen, die bekanntermaßen sicherheitskritisch sind. Tabelle 7-4 zeigt eine Auswahl kritischer Funktionen und ein mögliches, sichereres Pendant dazu.

Tabelle 7-4 *Unsichere versus sichere Funktionen*

Kritische Funktionen	Sichere Alternativen
sprintf	snprintf
gets	fgets
strcpy	strlcpy, strncpy

➔

Kritische Funktionen	Sichere Alternativen
strcat	strlcat, strncat
[vs]scanf	strtoul

Integration

Bei der Integration werden die separat implementierten Systemteile zum Gesamtsystem zusammengesetzt. Dabei ist darauf zu achten, eine vollständige Integration durchzuführen. Viele Maßnahmen aus dem Bereich IT-Security werden nämlich nicht für die eigentliche Funktion des eingebetteten Systems benötigt und das Fehlen fällt erst dann auf, wenn das System erfolgreich angegriffen wurde.

Test

Der Funktionstest des eingebetteten Systems ist durch einen Security-Test zu ergänzen. Hierbei können sehr gut Hackerwerkzeuge, wie beispielsweise `nmap` und `metasploit`, eingesetzt werden.

Außerdem sollte der Code einem Codereview unterzogen werden. Hierbei untersucht ein anderer Entwickler den Code. Wichtig: Die Algorithmen müssen übersichtlich und nachvollziehbar implementiert worden sein.

Wartung (Maintenance)

In der Phase des Betriebs beziehungsweise aus Sicht des Erstellers der Wartung (Maintenance) sind Security-Meldungen über sämtliche im System verwendete Komponenten aufmerksam zu verfolgen. Sind einzelne Komponenten betroffen, ist ein Update entweder von der betroffenen Komponente oder von dem Gesamtsystem zur Verfügung zu stellen. Am günstigsten ist es, wenn das Update automatisch eingespielt werden kann.

In jedem Fall muss jedes Update mit einer digitalen Unterschrift versehen werden, die bei der Installation vom Betreiber ebenso wie vom System geprüft wird. Es dürfen nur autorisierte Updates eingespielt werden.

7.3 Secure-Application-Design

Beim Design der funktionsbestimmenden Applikation sind einige grundsätzliche, sicherheitsrelevante Aspekte zu beachten (hier eine Auswahl):

- Sicherheitsmechanismen werden direkt in der Applikation verankert.
- Least Privilege: Es werden nur die benötigten Rechte eingesetzt.

- Funktionalität: Es werden nur die spezifizierten Funktionalitäten implementiert (nicht mehr Code als notwendig, keine Easter Eggs).
- Keine Defaultpasswörter verwenden.
- Zugang zu sicherheitsrelevanten Eigenschaften wird nur nach einer erfolgreichen Authentifizierung gewährt.
- Passwörter werden *niemals* (Wiederholung: Niemals!) im Klartext, sondern nur als gesalzene Passwort-Hashes abgelegt.
- Sämtliche Kommunikation wird verschlüsselt durchgeführt, auch wenn das zunächst nicht notwendig erscheint.
- Daten werden verschlüsselt abgelegt.
- Das Laufzeitverhalten wird randomisiert.
- Trennung von Code und Daten. Über Skripte und Makros lassen sich Applikationen häufig aushebeln.
- Keine externen Programme aufrufen. Falls dies unbedingt erforderlich ist, externe Programme nur isoliert in einer Sandbox ablaufen lassen.
- Programme möglichst einfach und übersichtlich halten.
- Für alle Konfigurationen von vornherein Defaultwerte wählen, die einen sicheren Betrieb gewährleisten.

7.3.1 Sicherheitsmechanismen in der Applikation

Betriebssysteme wie Linux bieten eine ganze Reihe ausgefeilter Sicherheitsmechanismen, wie Firewalling oder aber auch verschlüsselte Dateisysteme, um dort Daten abzulegen.

Beim Systementwurf könnten Sie also direkt auf diese Mechanismen setzen. Andererseits müssen Sie in diesem Fall den jeweiligen Implementierungen trauen. Besser, allerdings auch mit deutlich mehr Aufwand verbunden, ist es, die Sicherheitsfunktionen direkt in der Applikation zu implementieren. Die Applikation verschlüsselt in diesem Fall selbst die Daten und legt den verschlüsselten Bytestrom auf die Festplatte ab. Das hat mehrere Vorteile: Sie kennen die Verschlüsselung, Sie kontrollieren die (Qualität der) Verschlüsselung, die Implementierung ist einmalig und den meisten Angreifern zumindest anfänglich unbekannt. Ein Angreifer muss also gezielt Ihre Applikation »aufs Korn nehmen« und Ihr System wird eben nicht nebenbei mit gehackt, nur weil in einer Standardimplementierung ein Sicherheitsproblem aufgetaucht ist.

Das Gleiche gilt für die Kommunikation. Wenn Sie die kommunizierenden Systemkomponenten selbst implementieren, können Sie hier

auch ein eigenes, verschlüsseltes Protokoll entwerfen. Der Aufwand ist natürlich wie bei der verschlüsselten Datenablage nicht unerheblich, der Gewinn an Sicherheit aber eben auch.

7.3.2 Least Privilege

Das Prinzip »Least Privilege« verlangt, Applikationen in Module zu unterteilen, die jeweils nur die Rechte erhalten, die für die Funktionalität unbedingt benötigt werden (siehe Abschnitt 7.1.3).

Sollte Ihre Applikation besondere Rechte benötigen, ist diese nach dem Least-Privilege-Prinzip zu programmieren. Um solche besonderen Rechte zu erhalten, wird die Applikation mit Root-Rechten gestartet. Das geschieht entweder, indem der Superuser diese startet, oder, indem als Besitzer »root« eingetragen und das S-Bit gesetzt wird. Nach dem Start muss die Applikation sofort die Rechte reduzieren. Hier sind zwei Szenarien zu unterscheiden:

1. Die besonderen Rechte sind nur zeitlich bedingt, beispielsweise direkt nach Start der Applikation notwendig (Beispiel 7-12).
2. Die besonderen Privilegien werden immer mal wieder benötigt (Beispiel 7-11).

Im ersten Fall können die privilegierten Zugriffe direkt nach Programmstart erfolgen, danach werden diese permanent abgegeben. Im zweiten Fall können die Rechte nur temporär reduziert werden.

Werden immer wieder privilegierte Zugriffe benötigt, sollten Sie darüber nachdenken, diese in einen eigenen Thread auszulagern. In diesem Fall kann der ausgelagerte Thread die erhöhten Privilegien behalten, der ursprüngliche Thread jedoch reduziert seine Privilegien dauerhaft.

Beispiel 7-11 *Code zum temporären Reduzieren der Zugriffsrechte <drop_temporaer.c>*

```
#include <stdio.h>
#include <unistd.h>

#define UID_NOBODY 65534

int main( int argc, char **argv, char **envp )
{
   uid_t ruid, euid, suid;
   int res;

   getresuid(&ruid,&euid,&suid);
   res = setresuid(-1, UID_NOBODY, euid);
   if (res != 0) {
      perror( "setresuid" );
      return -1;
   }
```

```
   getresuid(&ruid,&euid,&suid);
   if (euid != UID_NOBODY) { // paranoid-check
      perror( "setresuid failed" );
      return -1;
   }

   // ab hier sind Privilegien temporaer herabgestuft
   printf("ruid: %d euid: %d suid: %d\n", ruid,euid,suid);

   // Privilegien wieder erhoehen
   euid = suid;
   setresuid(-1,euid,suid);
   getresuid(&ruid,&euid,&suid);
   printf("ruid: %d euid: %d suid: %d\n", ruid,euid,suid);

   return 0;
}
```

Beispiel 7-12 *Code zum dauerhaften Reduzieren der Zugriffsrechte <drop_perm.c>*

```
#include <stdio.h>
#include <unistd.h>

#define UID_NOBODY 65534

int main( int argc, char **argv, char **envp )
{
   uid_t ruid, euid, suid;
   int res;

   getresuid(&ruid,&euid,&suid);
   res = setresuid(UID_NOBODY, UID_NOBODY, UID_NOBODY);
   // bis hierher sind unter Umstaenden Privilegien vorhanden
   if (res != 0) {
      perror( "setresuid" );
      return -1;
   }
   getresuid(&ruid,&euid,&suid);
   if (euid != UID_NOBODY) { // paranoid-check
      perror( "setresuid failed" );
      return -1;
   }
   // ab hier sind Privilegien dauerhaft herabgestuft

   printf("ruid: %d euid: %d suid: %d\n", ruid,euid,suid);

   return 0;
}
```

Weitere Informationen, insbesondere ein portables Interface zum dauerhaften oder temporären Reduzieren der Zugriffsrechte, finden Sie unter [ChWaDe2002].

7.3.3 Easter Eggs

Als Easter Eggs werden Codefragmente bezeichnet, die nicht Teil der Anforderungen sind. Eines der bekanntesten Easter Eggs ist der Flugsimulator in Microsoft Excel. So spannend Easter Eggs im Prinzip sind, aus Sicht der IT-Security sind sie ein absolutes »No-Go«. Easter Eggs bedeuten, dass die Applikation mehr Code enthält. Mehr Code wiederum bedeutet mehr Fehler (keine Software ist fehlerfrei). Jeder Fehler aber ist ein potenzielles Einfallstor für einen Hacker. Mehr noch: Sie können davon ausgehen, dass die Easter-Egg-Funktionalität nicht in den Anforderungen steht und daher nur eingeschränkt, wenn überhaupt, getestet wird. Die Wahrscheinlichkeit für fehlerhaften Code ist entsprechend größer.

Sie sollten daher keinesfalls Funktionalitäten implementieren, die nicht gefordert, nicht dokumentiert und vor allem nicht gründlich und ausreichend getestet sind.

7.3.4 Passwortmanagement

Grundsätzlich darf Zugriff auf kritische Ressourcen nur nach einer Authentifizierung stattfinden. Hierbei wird häufig auf ein Passwort zurückgegriffen. Der sicherheitstechnisch korrekte Umgang mit Passwörtern ist nicht trivial.

Passwörter dürfen in keinem Fall im Klartext abgelegt werden, sondern immer nur als gesalzene Passwort-Hashes. Im Fall einer Server/Client-Anwendung sollte darüber hinaus ein Challenge/Response-Verfahren angewendet werden. Die Passwort-Hashes sind so im System abgelegt, dass nur mit entsprechenden Rechten ausgestattete Programme diese lesen können.

Ein geeignetes Hashverfahren vorausgesetzt, können über Hashwerte gesicherte Passwörter nur über Brute-Force-Attacken geknackt werden. Der Angreifer benötigt den Hashwert und muss alle möglichen Passwörter durchprobieren, bis der Hashwert gefunden wurde. Um diesen Vorgang zu vereinfachen, legen Cracker einmal gebildete Hashwerte in Tabellen ab (insbesondere in den speicherplatzsparenden Rainbow-Tabellen). Dieser Angriff wird den Hackern jedoch erschwert, wenn die Passwörter vor dem Hashen noch »gesalzen« werden.

Die Authentifizierung über Passwörter läuft damit so ab, dass der User aufgefordert wird, neben dem Benutzernamen sein Passwort einzu-

geben. Über den Benutzernamen wird das Salz ausgelesen und mit dem Passwort verknüpft. Für den Zugriff auf das Salz müssen die Privilegien angepasst (erhöht) und später wieder temporär reduziert werden. Das gesalzene Passwort wird in einen Hashwert, den gesalzenen Passwort-Hash, umgewandelt. Danach wird das Klartext-Passwort als Erstes aus dem Speicher (durch Überschreiben) gelöscht. Der gesalzene Passwort-Hash kann nun mit dem abgelegten Passwort-Hash verglichen werden. Bei Übereinstimmung ist der Benutzer authentifiziert. Für den Zugriff auf den abgespeicherten Hashwert müssen ebenfalls die Privilegien kurzfristig erhöht werden. Danach können sie wieder reduziert werden, im günstigsten Fall dauerhaft (permanent).

```
...
   pass_max = sysconf(_SC_PASS_MAX);
   username = getusername();
   passwd = getpass( "Passwort:" );
   if (restore_priv()) {
      return NOT_AUTHORIZED;
   }
   salt = getsalt( username );
   drop_priv_temp();
   salted_passwd = strncat( passwd, salt, pass_max );
   SHA1( (unsigned char*)salted_passwd, strlen(salted_passwd), hash );
   clear_passwd_mem( passwd ); // Klartext loeschen
   if (restore_priv()) {
      return NOT_AUTHORIZED;
   }
   stored_hash = find_hash( username );
   drop_priv_perm();
   if (strncmp((char *)hash, stored_hash, SHA_DIGEST_LENGTH)==0) {
      return AUTHORIZED;
   }
   return NOT_AUTHORIZED;
   ...
```

Bei Client/Server-Applikationen werden die Passwörter ebenfalls auf Basis von Passwort-Hashes verglichen. Dabei sollte der Passwort-Hash direkt auf dem Client angelegt werden. Damit ein Angreifer, der von der Authentifizierung einen Mitschnitt anfertigt, nicht an den Passwort-Hash kommt, wird typischerweise ein Challenge/Response-Verfahren verwendet. Der Server schickt dazu dem Client eine Zufallszahl. Der Client hasht das eingegebene Passwort zusammen mit einem Salz und verknüpft diesen Hashwert dann mit der Zufallszahl (zum Beispiel durch einfaches Aneinanderhängen). Vom Ergebnis wird wiederum ein Hashwert gebildet, der schließlich über das Netz zurück an den Server übertragen wird. Die ursprüngliche Zufallszahl wirkt bei diesem Verfahren wie ein zusätzliches »Salz«. Es verhindert, dass ein Angreifer in endlicher Zeit per Brute-Force-Attacke das Passwort erraten kann. Man

spricht daher auch von Salted-SHA1-Passwörtern oder auch von Digest-Passwörtern.

7.3.5 Verschlüsselung

Verschlüsselte Kommunikation

Die Kommunikation sollte grundsätzlich verschlüsselt stattfinden. Bei einem Webserver beispielsweise sollte statt `http` auf das Protokoll `https` umgestellt werden. Sonstige Kommunikation kann über TSL abgewickelt werden. Hierfür stehen mit der »libssl« geeignete Funktionen beziehungsweise Programme zur Verfügung.

https statt http

Mit dem Protokoll https werden nicht nur Daten verschlüsselt, es findet zudem auch eine Authentifizierung der Kommunikationspartner statt. Dazu sendet der Webserver seinen digitalen Ausweis an den Webbrowser. Der Ausweis wiederum trägt eine digitale Unterschrift, die vom Webbrowser überprüft wird. Dazu sind in den modernen Browsern eine große Anzahl von Unterschriftenproben der Certification Authorities hinterlegt. Wird der Ausweis als gültig erkannt, werden die Daten ausgetauscht. Findet sich aber keine gültige Unterschrift, weil beispielsweise keine Unterschriftenprobe vorliegt, gibt es eine Warnmeldung. Der Anwender muss aktiv erklären, dass er die »unsichere« Webseite wirklich sehen möchte.

Eine digitale Unterschrift für einen Webserver ist zwar mit erträglichem Aufwand zu realisieren, kostet aber regelmäßig Geld. Hinzu kommt, dass mehrere Vorfälle im Bereich der Certification Authorities (zum Beispiel DigiNotar, [heise2012]) zeigen, dass der Certification Authority nicht zu trauen ist. Mehrfach ist es gelungen, gefälschte Zertifikate zu bekommen. Das ist leicht nachvollziehbar, wenn Sie sich die umfangreiche Liste voller unbekannter Namen in den Browsern anzeigen lassen. Eine Authentifizierung mit https ist wertlos! Daher verzichten viele Webserverbetreiber auf die Authentifizierung und nutzen nur die Verschlüsselung. Zu diesem Zweck erstellen sie selbst mit OpenSSL ein Zertifikat, das sie dann mit der eigenen digitalen Unterschrift unterschreiben (self signed certificate).

Alternativ kann auch ein Virtual Private Network eingesetzt werden. Bei diesem werden die Daten verschlüsselt und über zumeist öffentliche Netze getunnelt. Zur Auswahl stehen in diesem Bereich unter anderem IPSec und OpenVPN. IPSec ist zwar Industriestandard, aber in der Konfiguration außerordentlich komplex. Da die Verschlüsselung im Kernel stattfindet, ist der Overhead vergleichsweise gering. OpenVPN dagegen ist in der Anwendung sehr einfach. Die Verschlüsselung findet jedoch im Userland (als Service) statt, wodurch mehr Ressourcen benötigt wer-

den. Allerdings ist dadurch auch die Realisierung einfacher, da im Kernel nur ein überschaubar kleiner Treiber benötigt wird.

Verschlüsselte Datenablage

Daten sollten nur mit wenigen Ausnahmen verschlüsselt abgelegt werden. Am besten ist es dabei, wenn die Applikation selbst die Verschlüsselung vornimmt. In diesem Fall ist es dann auch sicherheitstechnisch egal, ob die Daten lokal oder irgendwo anders (remote, in der Cloud) abgelegt werden.

Wer keine eigene Verschlüsselung implementieren möchte, kann auf verschlüsselte Dateisysteme oder auf vorhandene Werkzeuge zurückgreifen. Eine attraktive Variante zur Verschlüsselung stellt `gpg` dar. Aufwendig ist initial die Schlüsselgenerierung, der Einsatz über Kommandozeilenwerkzeuge gestaltet sich dann jedoch einfach. Buildroot bietet `gpg` in der Konfiguration unter [Package Selection for the target][Shell and utilities][gnupg] an.

7.3.6 Randomisiertes Laufzeitverhalten

Ein bisher noch wenig beachteter Punkt betrifft das deterministische Verhalten des eingebetteten Systems bezüglich Laufzeit und Stromverbrauch. Hacker sind in der Lage, auf Basis des Stromverbrauchs Rückschlüsse auf die verarbeiteten Daten zu ziehen. Auf diese Weise sind bereits Passwörter geknackt worden. Um Angriffe dieser Art zu erschweren, müssen Laufzeitverhalten und auch der Stromverbrauch randomisiert, also zufällig werden.

Eine einfache Möglichkeit der Realisierung besteht darin, einen Dummy-Thread zu starten, der eine zufällig bestimmte Zeit rechnet und sich dann für eine ebenfalls zufällig bestimmte Zeit schlafen legt. Dieser Ansatz hat aber zwei Nachteile: Auf einer Multicore-Maschine könnten der eigentliche Thread und der Dummy-Thread auf unterschiedlichen Cores laufen. Durch Beobachtung der einzelnen Threads könnten weiterhin Rückschlüsse auf die verarbeiteten Daten gezogen werden. Läuft der Dummy-Thread quasi gleichzeitig auf einer CPU, könnte eventuell der Threadwechsel charakteristisch sein und damit identifiziert werden.

Besser ist es also, in der Applikation Veränderungen vorzunehmen und eine Funktion `random_work(int min_ns, int max_ns)` einzuführen und aufzurufen, die im angegebenen Intervall per Zufall rechnet und eventuell sogar Plattenzugriffe durchführt.

8 Ein komplettes Embedded-Linux-Projekt

Mithilfe des Raspberry Pi und einem zweizeiligen Display soll eine Messagebox aufgebaut werden. Diese Messagebox stellt standardmäßig abwechselnd die IP-Adresse des Embedded Linux und die aktuelle Uhrzeit dar. Insbesondere wenn die IP-Adresse per DHCP verteilt wird, ist die Ausgabe sehr nützlich, da man sich das Suchen (zum Beispiel durch Einloggen und Aufruf von `ifconfig`) danach erspart. Wird darüber hinaus an einem Webinterface eine Nachricht eingegeben, erscheint diese auf dem Display, bis sie entweder am Webinterface wieder gelöscht wird oder aber eine neue Nachricht dargestellt werden soll.

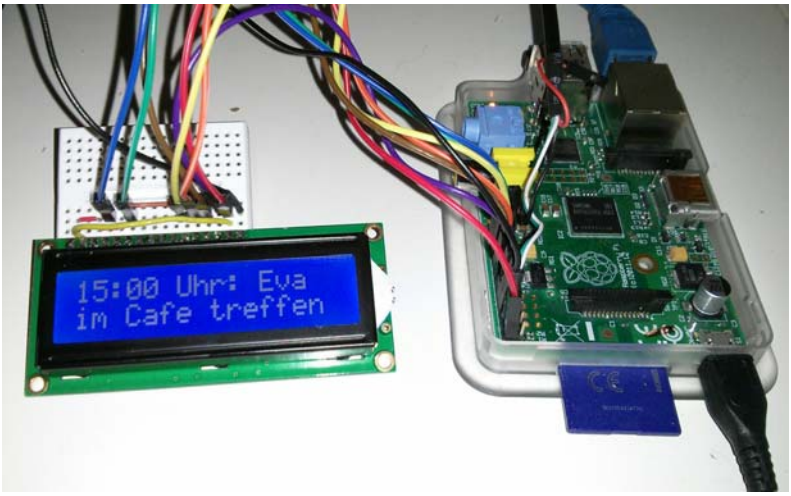


Abb. 8-1
Messagebox

Beim Entwurf und der Realisierung der Messagebox stand weder eine ausgefeilte Funktionalität noch Robustheit im Vordergrund. Vielmehr geht es darum, anhand eines vollständigen Beispiels die Entwicklungsaspekte eines Embedded Linux in seiner Gesamtheit vorzustellen. Das betrifft die Hardware, die System- und natürlich die Anwendungssoftware. Das Embedded Linux wird mithilfe von Buildroot aufgebaut. Die im Buch vorgestellten grundlegenden Aspekte der IT-Sicherheit, insbesondere Firewalling und Rechtemanagement, werden berücksichtigt.

Abb. 8-2 *Messagebox-Webinterface*

8.1 Hardware: Anschluss des Displays

Zwei- oder auch vierzeilige Displays sind bereits ab 3 € zu bekommen. Vorteilhafterweise verwenden die meisten einen Controller vom Typ HD44780, der über ein paralleles Interface angesprochen wird. Die Displaymodule besitzen 16 Lötkontakte, in die man eine einreihige Stiftleiste einlöten kann. Die Pinbelegung dieser Lötkontakte zeigt Tabelle 8-1. Obwohl die Displaymodule für eine Spannung von 5 V ausgelegt sind, kann der mit 3,3 V arbeitende Raspberry Pi selbige über sechs GPIOs ansteuern.

Tabelle 8-1 *Pinbelegung für Display HD44780*

Pin	Bezeichnung	Bedeutung
1	Vss	Masse GND
2	Vdd	Betriebsspannung 5 V
3	Vo	Displayspannung (Kontrast)
4	RS	Register Select (0=Command,1=Data)
5	R/W	Lese-/Schreibrichtung
6	E	Enable
7	D0	Datenleitung 0 (LSB)
8	D1	Datenleitung 1
9	D2	Datenleitung 2
10	D3	Datenleitung 3
11	D4	Datenleitung 4
12	D5	Datenleitung 5
13	D6	Datenleitung 6

➔

Pin	Bezeichnung	Bedeutung
14	D7	Datenleitung 7 (MSB)
15	LED+	Pluspol LED-Beleuchtung
16	LED-	Minuspol LED-Beleuchtung

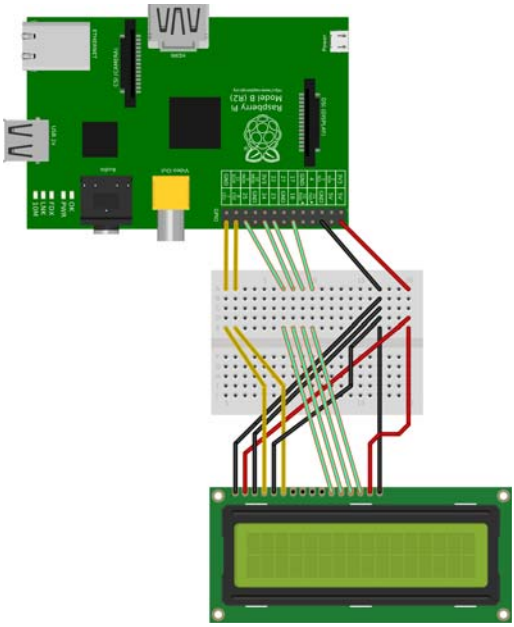


Abb. 8-3
Verschaltung zwischen Raspberry Pi und Display

Das parallele Interface des Controllers besteht aus den Datenleitungen D0-D7, dem Signal R/S (Register Select), dem Signal R/W für die Schreib-/Leserichtung und dem Signal E (Enable). Um dem Controller einen Wert zu übertragen, wird der Wert auf die Datenleitungen gelegt, per R/S entweder das Kommando- oder das Datenregister ausgewählt, die Transferrichtung ausgesucht und dann über eine logische Eins auf der Leitung E die Übernahme der Daten in den Controller eingeleitet. Nachdem die Daten im Controller sind, wird die Leitung E wieder auf Null gelegt.

Um GPIOs zu sparen, lässt sich der Controller in einem 4-Bit-Modus betreiben, bei dem die Datenleitungen D0 bis D3 nicht verwendet werden. Da der lesende Zugriff für den Betrieb nicht zwingend notwendig ist, kann noch eine weitere Leitung eingespart werden: R/W wird mit Masse (GND) verbunden. Damit ergibt sich die in Tabelle 8-2 und Abbildung 8-3 dargestellte Verschaltung zwischen Display und Raspberry Pi.

Neben den GPIO-Leitungen müssen noch die Versorgungsleitungen (Vcc, Vss) verbunden werden. Stellen Sie sicher, dass das von Ihnen verwendete Modul die identische Steckerbelegung hat, bevor Sie selbst die Verdrahtung vornehmen. Auch gibt es anscheinend Module im Markt, die eine andere elektronische Charakteristik aufweisen. Für Beschädigungen, die mit der angegebenen Schaltung auftreten, sind Sie letztlich selbst verantwortlich. Die Verschaltung mitsamt einiger sinnvoller Erläuterungen dazu sind auch in [http://www.schnatterente.net/technik/raspberry-pi-32-zeichen-hitachi-hd44780-display] zu finden.

Tabelle 8-2 *Verbindung zwischen Display und Raspberry Pi*

Display	Steckerleiste Raspberry Pi
1 (Vss,GND)	6 (GND)
2 (Vcc,+5 V)	2 (+5 V
3 (Vo,Kontrast)	6 (GND)
4 (RS,Register Select)	26 (GPIO7)
5 (R/W)	6 (GND, nur schreiben)
6 (E,Enable)	24 (GPIO8)
11 (D4)	22 (GPIO25)
12 (D5)	18 (GPIO24)
13 (D6)	16 (GPIO23)
14 (D7)	12 (GPIO18)
15 (LED+)	2 (+5 V)
16 (LED-)	6 (GND)

8.2 Software

Gerätetreiber

Für das auf dem Controller HD44780 basierende Display ist als Erstes ein Treiber zu schreiben. Neben einer didaktisch nachvollziehbar einfachen Struktur soll er ein intuitiv zu bedienendes Interface haben. Dieses sieht nur einen Schreib- und keinen Lesezugriff vor. ASCII-Zeichen, die auf die Gerätedatei des Treibers geschrieben werden, werden am Display ausgegeben. Die ersten 16 Zeichen erscheinen in der ersten Zeile, die nächsten Zeichen werden in die zweite Zeile geschrieben. Jeder Schreibaufruf überschreibt die vorherigen Daten.

Um den Gerätetreiber erstellen zu können, muss noch einiges über die Ansteuerung des Displays bekannt sein. Die Grundlagen finden Sie in Kasten auf Seite 241.

Softwaretechnische Ansteuerung des Displays

Angesteuert wird das Display über zwei Register, ein Kommandoregister, mit dem beispielsweise ein Cursor an- oder ausgeschaltet werden kann, und ein Datenregister. Dabei werden grundsätzlich acht Bit geschrieben. Im 4-Bit-Modus wird zunächst das obere Nibble (die oberen vier Bit), danach das untere Nibble geschrieben.

Um ein Nibble, also um vier Bit, auf das Interface zu schreiben, werden zunächst die GPIOs 25, 24, 23 und 18 auf den auszugebenden Wert gesetzt. Über GPIO7 wird das Register ausgewählt. Ist GPIO7 null, wird in das Kommandoregister geschrieben, ist es eins, in das Datenregister. Haben die GPIOs den gewünschten Wert, wird über GPIO8 Enable auf eins gesetzt. Damit übernimmt das Display die Daten. Das kann bis zu 40 Mikrosekunden dauern. Danach wird Enable wieder auf null gesetzt.

Um das Display nutzen zu können, muss es in einem ersten Schritt über eine Folge von Kommandos initialisiert werden. Tabelle 8-3 zeigt die möglichen Kommandos. Das Wichtigste ist dabei, das Display zunächst auf den 4-Bit-Betrieb einzustellen. Dabei ist zu beachten, dem Controller zur Verarbeitung der Kommanos Zeit zu geben. Zum Einstellen des 4-Bit-Betriebs ist in das Kommandoregister eine hexadezimale 0x20 zu schreiben. Da man sich nie sicher sein kann, ob das Interface nicht bereits auf 4 Bit eingestellt ist, wird das Interface per 0x30 zunächst auf 8 Bit ein- und dann auf 4 Bit umgestellt. Ist es im 4-Bit-Modus definiert (nach dem Schreiben von dreimal 0x3 und einmal 0x2 ist das der Fall), wird noch die zweistellige Anzeige und die 5x8-Darstellung aktiviert (Befehl hexadezimal 0x28). Der Befehl 0x01 löscht das Display und 0x0c schaltet es ein, den Cursor und das Blinken aus.

Befehl	D7	D6	D5	D4	D3	D2	D1	D0	Beschreibung
Display löschen	0	0	0	0	0	0	0	1	Löscht das Display
Cursor auf Anfang	0	0	0	0	0	0	1	*	Setzt den Cursor auf Adresse 0
Entry Mode Set	0	0	0	0	0	1	I/D	S	Increment/ Decrement=1: Cursor bewegt sich nach rechts; Shift=0: Cursor wandert nach links, Shift=1: Text verschiebt sich auf dem Display
Display ein/aus	0	0	0	0	1	D	C	B	D=0: Display aus, D=1: Display ein; C=0: Cursor aus, C=1: Cursor ein; B=0: Cursor ohne Blinken, B=1: Cursor mit Blinken

➔

Tabelle 8-3 *Befehlscodierung des Display-Controllers HD44780*

Befehl	D7	D6	D5	D4	D3	D2	D1	D0	Beschreibung
Cursor/Display Shift	0	0	0	1	S/C	R/L	*	*	S/C=0: Auswahl Display, S/C=1: Auswahl Cursor; R/L=0: schieben nach links, R/L=1: schieben nach rechts
Function Set	0	0	1	DL	N	F	*	*	DL=0: 4-Bit-Interface, DL=1: 8-Bit-Interface; N=0: Display einzeilig, N=1: Display zweizeilig; F=0: 5x8-Darstellung, F=1: 5x11-Darstellung
DD-RAM-Adresse	1	x	x	x	x	x	x	x	Ausgabeposition einstellen

Der Gerätetreiber für das Display wird aus didaktischen Gründen sehr einfach gehalten. Wird der Treiber geladen, werden die benötigten GPIO-Leitungen reserviert und auf Ausgabe (Output) geschaltet. Ist das erfolgreich, wird das Display initialisiert (4-Bit-Betrieb, zweizeilig, kein Cursor). Als zentrale Funktion wird eine Funktion `driver_write()` benötigt. Diese kopiert den auszugebenden String vom Speicher der Applikation (Userland) in den Kernelspeicher und gibt die ersten 16 Zeichen aus. Danach wird auf die zweite Zeile umgestellt und die nachfolgenden Zeichen werden ausgegeben. Auf `driver_open()` und `driver_close()` verzichten wir, wenngleich hier eine sinnvolle Überprüfung, ob die Applikation auch tatsächlich nur schreibend zugreifen möchte, implementiert werden könnte.

Für das einfache Handling mit der Hardware gibt es im Treiber die Hilfsfunktionen `nibble_write()` und `lcd_write()`. Erstere schreibt ein Nibble in den Controller je nach gewünschtem Register (Kommando oder Daten) und die Funktion `lcd_write()` schreibt ein Byte, indem sie zweimal `nibble_write()` aufruft.

Beispiel 8-1 *Display-Gerätetreiber*

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/gpio.h>
#include <linux/delay.h>
#include <linux/ctype.h>
#include <asm/uaccess.h>

static dev_t hd44780_dev_number = MKDEV(248,0);
static struct cdev *driver_object;
```

```
static struct class *hd44780_class;
static struct device *hd44780_dev;
static char textbuffer[1024];

/*
http://www.sprut.de/electronic/lcd/
https://www.mikrocontroller.net/articles/AVR-Tutorial:_LCD
*/

static void nibble_write( int reg, int value )
{
   gpio_set_value(7, reg); // RS
   gpio_set_value(25, value & 0x1 ); // D4
   gpio_set_value(24, value & 0x2 ); // D5
   gpio_set_value(23, value & 0x4 ); // D6
   gpio_set_value(18, value & 0x8 ); // D7
   gpio_set_value( 8, 1 ); // E
   udelay(40);
   gpio_set_value( 8, 0 );
}

static void lcd_write( int reg, int value )
{
   nibble_write( reg, value>>4 ); // High-Nibble
   nibble_write( reg, value&0xf ); // Low-Nibble
}

static int gpio_request_output( int nr )
{
   char gpio_name[12];
   int err;

   snprintf( gpio_name, sizeof(gpio_name), "rpi-gpio-%d", nr );
   err = gpio_request( nr, gpio_name );
   if (err) {
      printk("gpio_request for %s failed with %d\n", gpio_name, err);
      return -1;
   }
   err = gpio_direction_output( nr, 0 );
   if (err) {
      printk("gpio_direction_output failed %d\n", err);
      gpio_free( nr );
      return -1;
   }
   return 0;
}
```

```
static int display_init( void )
{
   printk("display_init\n");
   if (gpio_request_output( 7)==-1) return -EIO;
   if (gpio_request_output( 8)==-1) goto free7;
   if (gpio_request_output(18)==-1) goto free8;
   if (gpio_request_output(23)==-1) goto free18;
   if (gpio_request_output(24)==-1) goto free23;
   if (gpio_request_output(25)==-1) goto free24;
   msleep( 15 );
   nibble_write( 0, 0x3 );
   msleep( 5 );
   nibble_write( 0, 0x3 );
   udelay( 100 );
   nibble_write( 0, 0x3 );
   msleep( 5 );
   nibble_write( 0, 0x2 );
   msleep( 5 );
   lcd_write( 0, 0x28 ); // CMD: 4-Bit Mode, 2 stellige Anzeige, 5x8 Font
   msleep( 2 );
   lcd_write( 0, 0x01 );
   msleep( 2 );
   lcd_write( 0, 0x0c ); // Display ein, Cursor aus, Blinken aus
   lcd_write( 0, 0xc0 );
   lcd_write( 1, 'H' );
   lcd_write( 1, 'i' );
   return 0;
free24: gpio_free( 24 );
free23: gpio_free( 23 );
free18: gpio_free( 18 );
free8:  gpio_free( 8 );
free7:  gpio_free( 7 );
   return -EIO;
}

static int display_exit( void )
{
   printk( "display_exit called\n" );
   gpio_free( 25 );
   gpio_free( 24 );
   gpio_free( 23 );
   gpio_free( 18 );
   gpio_free( 8 );
   gpio_free( 7 );
   return 0;
}

static ssize_t driver_write(struct file *instanz,const char __user *user,
   size_t count, loff_t *offset )
```

```
{
   unsigned long not_copied, to_copy;
   int i;

   to_copy = min( count, sizeof(textbuffer) );
   not_copied=copy_from_user(textbuffer, user, to_copy);
   //printk("driver_write( %s )\n", textbuffer );

   lcd_write( 0, 0x80 );
   for (i=0; i<to_copy && textbuffer[i]; i++) {
      if (isprint(textbuffer[i]))
         lcd_write( 1, textbuffer[i] );
      if (i==15)
         lcd_write( 0, 0xc0 );
   }

   return to_copy-not_copied;
}

static struct file_operations fops = {
    .owner  = THIS_MODULE,
    .write   = driver_write,
};

static int __init mod_init( void )
{
   if( register_chrdev_region(hd44780_dev_number,1,"hd44780")<0 ) {
      printk("devicenumber (248,0) in use!\n");
      return -EIO;
   }
   driver_object = cdev_alloc(); /* Anmeldeobjekt reserv. */
   if( driver_object==NULL )
      goto free_device_number;
   driver_object->owner = THIS_MODULE;
   driver_object->ops = &fops;
   if( cdev_add(driver_object,hd44780_dev_number,1) )
      goto free_cdev;
   hd44780_class = class_create( THIS_MODULE, "hd44780" );
   if( IS_ERR( hd44780_class ) ) {
      pr_err( "hd44780: no udev support\n");
      goto free_cdev;
   }
   hd44780_dev = device_create(hd44780_class,NULL,hd44780_dev_number,
      NULL, "%s", "hd44780" );
   dev_info( hd44780_dev, "mod_init called\n" );
   if (display_init()==0)
      return 0;
```

```
free_cdev:
   kobject_put( &driver_object->kobj );
free_device_number:
   unregister_chrdev_region( hd44780_dev_number, 1 );
   printk("mod_init failed\n");
   return -EIO;
}

static void __exit mod_exit( void )
{
   dev_info( hd44780_dev, "mod_exit called\n" );
   display_exit();
   device_destroy( hd44780_class, hd44780_dev_number );
   class_destroy( hd44780_class );
   cdev_del( driver_object );
   unregister_chrdev_region( hd44780_dev_number, 1 );
   return;
}

module_init( mod_init );
module_exit( mod_exit );
MODULE_LICENSE("GPL");
```

Displayapplikation

Die Applikation, die dem Gerätetreiber die anzuzeigenden Daten übergibt, soll beim Booten gestartet werden und dann aktiv bleiben. Sie gibt den Inhalt der Datei `/tmp/message` aus. Ist die Datei nicht vorhanden, zeigt sie im Sekundentakt wechselnd die IP-Adresse und die aktuelle Uhrzeit an.

Diese Applikation lässt sich gut als Shellskript realisieren (siehe Beispiel 8-2). In einer Endlosschleife überprüft das Skript, ob eine Datei `/tmp/message` existiert. Ist dies der Fall, wird getestet, ob sich die Datei seit der letzten Überprüfung geändert hat. Dann nämlich muss die Anzeige zunächst gelöscht und schließlich mit dem Inhalt der Datei überschrieben werden. Dazu wird die Ausgabe von `cat` auf die Gerätedatei `/dev/hd44780` umgeleitet. In jedem Fall legt sich das Skript dann für eine Sekunde schlafen.

Etwas verzwickt ist der Mechanismus, mit dem nur dann neue Daten angezeigt werden, wenn es auch eine Änderung gegeben hat. Das wird über eine Hashbildung per `md5sum` realisiert. In jedem Durchlauf wird die md5sum über die Datei gebildet und mit dem vorherigen Wert verglichen. Sind die Werte gleich, gab es keine Änderung.

Existiert jedoch keine Datei /tmp/message, gibt das Skript zunächst das aktuelle Datum aus, schläft eine Sekunde und extrahiert dann mit einer Verkettung der Kommandos ifconfig, grep und cut die IP-Adresse. Diese wird dann mithilfe von echo auf die Gerätedatei umgeleitet, wodurch sie auf dem Display erscheint. Danach schläft das Skript für eine Sekunde.

Das Skript gibt übrigens die Uhrzeit als UTC, als koordinierte Weltzeit, aus. Um beispielsweise die lokale Zeit für Deutschland angezeigt zu bekommen, legen Sie eine Datei /etc/TZ mit dem folgenden Inhalt an:

```
CET-1CEST-2,M3.5.0/02:00:00,M10.5.0/03:00:00
```

Beispiel 8-2
Skript zur Ausgabe von Daten auf das Display

```
#!/bin/sh

while true
do
   if [ -f /tmp/message ]; then
      textid=`md5sum /tmp/message | cut -d" " -f1`
      if [ "$textid" != "$oldtextid" ]; then
         # Anzeige loeschen
         echo "                                ">/dev/hd44780
         # Nachricht ausgeben
         cat /tmp/message >/dev/hd44780
         oldtextid=$textid # Text nicht mehrfach ausgeben
      fi
      sleep 1
   else
      date >/dev/hd44780
      sleep 1
      ip=$(/sbin/ifconfig eth0 | grep inet | cut -d: -f2 | cut -d" " -f1)
      echo "IP:"$ip"                       " >/dev/hd44780
      sleep 1
      textid=0
   fi
done
```

Webseite

Per Webinterface soll die auf der Messagebox angezeigte Nachricht eingegeben beziehungsweise die Standardnachricht, bestehend aus dem Wechsel zwischen IP-Adresse und Uhrzeit, eingestellt werden.

Kern des Webinterface ist ein HTML-Formular, das ein Eingabefeld für die darzustellende Nachricht und zwei Buttons enthält (siehe Abb. 8-2).

Der erste Button aktiviert die eingegebene Nachricht, der zweite schaltet wieder die Standardnachricht ein. Wird einer der beiden Buttons gedrückt, wird auf dem Webserver das Skript `showmsg.php` aktiviert, das die an den Browser zu übertragende HTML-Seite mit den HTML-Formularen generiert.

Wurde der Button für die Standardnachricht gedrückt, löscht das Skript `showmsg.php/tmp/message`. Ansonsten schreibt es den Inhalt des Texteingabefeldes in die Datei. `showtime.sh` erledigt dann den Rest.

Das mit `showmsg.php` realisierte Webinterface ist absolut schnörkelfrei und lenkt damit nicht von der eigentlichen Funktionalität ab.

Beispiel 8-3 *Webinterface der Messagebox*

```
<html>
<h1>Messagebox</h1>

<?php
   $message=htmlspecialchars($_REQUEST["message"]);
   $clear = htmlspecialchars($_REQUEST["clear"]);

   #echo "<br>clear: ".$clear;
   #echo "<br>message: ".$message;

   if ($clear=="Standardmessage") {
      unlink("/tmp/message");
      echo "<h2>Standardnachricht wird angezeigt.</h2><br>\n";
   } else if ($message!="") {
      $datei=fopen( "/tmp/message","w" );
      fwrite($datei,$message,32);
      echo "<h2>Letzte Nachricht: ".$message."</h2><br>\n";
      fclose($datei);
   }
?>

<form action="showmsg.php" method="post">Neue Message:<br>
<input type="Text" name="message"><br>
<input type="Submit" value="Message anzeigen">
</form>

<form action="showmsg.php" method="post">
<input type="Submit" name="clear" value="Standardmessage">
</form>

</html>
```

8.3 Systemintegration

Um die Messagebox zu realisieren, sind die folgenden Schritte notwendig:

1. Bisheriges Buildroot-System als Ausgangsbasis verwenden
2. PHP installieren
   a. In der Buildroot-Konfiguration PHP auswählen
   b. Die Datei `postbuild.sh` um die Modifikation von `php.ini` erweitern
3. Webinterface integrieren
   a. PHP-Skript `showmsg.php` im Verzeichnis `userland/target/` erstellen
   b. Konfigurationsdatei `httpd.conf` erweitern, sodass PHP-Skripte ausgeführt werden dürfen
   c. `postbuild.sh` erweitern, sodass die neuen Dateien in das Target-System kopiert werden
4. Applikation integrieren, die die IP-Adresse, das Datum oder die Nachricht ausgibt
   a. Applikation `showtime.sh` unter `userland/target/` erstellen
   b. Init-Skript `S53showmsg` unter `userland/target/` erstellen
   c. `postbuild.sh` erweitern, sodass die neuen Dateien in das Target-System kopiert werden
5. Integration des Treibers in Buildroot
   a. Treiberquellcode und Makefile im Verzeichnis `driver/hd44780/` anlegen
   b. Neues Verzeichnis `buildroot-2013.05/package/hd44780/` erstellen
   c. Die Bauvorschrift `hd44780.mk` und die Auswahldatei `Config.in` im Verzeichnis erstellen
   d. Die Auswahldatei `buildroot-2013.05/package/Config.in` um die Verbindung auf den neuen Treiber erweitern
   e. Im Buildroot-Verzeichnis `make menuconfig` aufrufen und den Displaytreiber auswählen
6. System durch Aufruf von `make` generieren.

Die Messagebox soll als Embedded Linux auf Basis von Buildroot aufgebaut werden. Als Grundlage dient das System aus Abschnitt 7.1.3, das neben der Basisfunktionalität erste Sicherheitsmechanismen integriert. Sollten Sie das System noch nicht so weit aufgebaut haben, finden Sie die Konfigurationstdateien für Buildroot, Busybox und der uclibc auf dem Webserver unter dem Verzeichnis `7.1.3./`. Zusätzlich sind

noch die in Tabelle 8-4 aufgelisteten Dateien erforderlich. Die für die Anwendung Messagebox zusätzlich benötigten Dateien sind in Tabelle 8-5 gelistet.

Ausgehend von dem System aus Abschnitt 7.1.3 müssen Sie als Erstes PHP installieren. Dazu ist Buildroot entsprechend zu konfigurieren. Die Datei `/etc/php.ini` benötigt zwei kleine Anpassungen, damit PHP mit dem Webserver der Busybox zusammenspielt. So müssen die Zeilen »cgi.force_redirect = 0« und »cgi.redirect_status_env = "yes"« eingefügt werden.

Im zweiten Schritt wird die Webseite in das System integriert. Dazu muss zum einen das Skript `showmsg.php` auf dem Entwicklungsrechner im Verzeichnis `raspi/userland/target/` abgelegt und zum anderen die Konfiguration des Webservers `httpd.conf` angepasst werden. Das Skript `postbuild.sh` wird modifiziert, sodass die Dateien in die richtige Stelle auf dem Target-System kopiert werden.

Beispiel 8-4 *Konfigurationsdatei zur Erstellung von Gerätedateien*

```
# See package/makedevs/README for details
#
# This device table is used only to create device files when a static
# device configuration is used (entries in /dev are static).
#
# <name>   <type>   <mode>   <uid>   <gid>   <major>   <minor>   <start>
<inc>   <count>

# MMC devices
/dev/mmcblk0   b   660   0   0   179   0   0   0   -
/dev/mmcblk0p1   b   660   0   0   179   1   0   0   -
/dev/mmcblk0p2   b   660   0   0   179   2   0   0   -
# Display HD44780
/dev/hd44780   c   222   0   0   248   0   0   0   -
```

Für die Applikation `showtime.sh`, die das Display ansteuert, wird zunächst die Gerätedatei `/dev/hdd44780` benötigt. Dazu erweitern Sie die Datei `raspi/userland/target/mmc.txt` um die notwendigen Informationen bezüglich Gerätedateityp, Zugriffsrechte und der aus Major- und Minornummer bestehenden Gerätenummer. Die Applikation `showtime.sh` und das Init-Skript `S53showip` werden beide unter `raspi/userland/target/` abgelegt, `postbuild.sh` wird erweitert, damit die Dateien auf dem Target-System installiert werden. Das vollständige Skript `postbuild.sh` finden Sie in Beispiel 8-8.

Beispiel 8-5
Init-Skript zum Starten der Displayapplikation <S53showip>

```
#!/bin/sh
#
# Start display hd44780
#

NAME=showtime
DAEMON=/usr/bin/$NAME

case "$1" in
  start)
   echo -n "Show IP: "
   modprobe hd44780
   start-stop-daemon -S -b -x $DAEMON
   ;;
  stop)
   start-stop-daemon -K -q -n $NAME
   echo "Shutting down                  " >/dev/hd44780
   ;;
  restart|reload)
   $0 stop
   sleep 1
   $0 start
   ;;
  *)
   echo "Usage: $0 {start|stop|restart}"
   exit 1
esac

exit $?
```

Als Nächstes muss noch der Treiber integriert werden. Der Quellcode ist bereits vorgestellt worden, für die Integration in Buildroot fehlen aber noch die Bauvorschrift (Beispiel 8-6) und die Konfigurationsauswahldatei (Beispiel 8-7). Beide müssen in einem neuen Verzeichnis `buildroot-2013.05/package/hd44780/` untergebracht werden. Außerdem muss die Datei `buildroot-2013.05/package/Config.in` um den Verweis auf die Auswahldatei (»source "package/hd44780/Config.in"«) für den Displaytreiber unterhalb der Kategorie »Miscellaneous« erweitert werden.

Beispiel 8-6
Buildroot-Bauvorschrift für den Displaytreiber

```
############################################################
#
# hd44780 - simple display driver
#
############################################################

HD44780_VERSION = 20131231
HD44780_SITE_METHOD = local
```

```
#HD44780_SITE = ../../application/hd44780-$(HD44780_VERSION)
HD44780_SITE = ../../driver/hd44780

HD44780_DEPENDENCIES = linux

define HD44780_BUILD_CMDS
   $(MAKE) $(LINUX_MAKE_FLAGS) -C $(LINUX_DIR) M=$(@D) modules
endef

define HD44780_INSTALL_TARGET_CMDS
   $(MAKE) $(LINUX_MAKE_FLAGS) -C $(LINUX_DIR) M=$(@D) modules_install
endef

define HD44780_CLEAN_CMDS
   $(MAKE) -C $(@D) clean
endef

define HD44780_UNINSTALL_TARGET_CMDS
   rm -f $(TARGET_DIR)/usr/bin/hd44780
endef

$(eval $(generic-package))
```

Beispiel 8-7 *Buildroot-Auswahldatei für den Displaytreiber*

```
config BR2_PACKAGE_HD44780
   bool "simple drive for a HD44780 display"
   depends on BR2_LINUX_KERNEL
   help
     Simple driver to control a HD44780 display
```

Damit sind die Vorbereitungen getroffen und im Buildroot-Verzeichnis kann zunächst `make menuconfig` aufgerufen werden. Neben PHP ist der Displaytreiber zu aktivieren. Anschließend wird das System per `make` generiert. Vergessen Sie nicht, dass die Umgebungsvariable PATH den Pfad zu den Cross-Entwicklungswerkzeugen enthält, ansonsten bricht `make` mit einer Fehlermeldung ab. Die komplette Befehlssequenz finden Sie zur Kontrolle in Beispiel 8-9.

Nach der erfolgreichen Systemgenerierung, bei der Sie noch die Passwörter für die User »root« und »default« eingeben, werden Kernel und Initramfs (Rootfilesystem) auf dem TFTP-Server abgelegt. Mit dem nächsten Reboot des per LAN vernetzten Raspberry Pi bootet dieser das System. Ist die Hardware korrekt angeschlossen, zeigt der Raspberry Pi nach dem Booten abwechselnd die IP-Adresse und die aktuelle Uhrzeit an.

Um eine eigene Nachricht anzeigen zu lassen, starten Sie auf Ihrem Entwicklungsrechner einen Browser und geben in der Adresszeile die (angezeigte) IP-Adresse der Messagebox, gefolgt von `/cgi-bin/`

showmsg.php, ein. Auf der erscheinenden Webseite kann die auszugebende Textnachricht eingegeben und an das Embedded Linux System geschickt werden.

Ist das System funktionstüchtig, kann es auch auf die SD-Karte transferiert werden. Das Vorgehen hierzu ist in Abschnitt 4.2.1 bereits beschrieben worden.

Verwenden Sie das System aber keinesfalls für andere Aufgaben als die Ansteuerung des Displays. Das gilt zumindest für den Fall, dass Sie mit den GPIOs andere Hardware betreiben!

Wenn's schiefgeht ...

Die Messagebox ist bereits ein komplexeres Beispiel und entsprechend viele Fallstricke kann es beim Bauen des Systems geben. Falls das System nicht funktioniert, grenzen Sie als Erstes die Ursache auf die Komponenten »Booten«, »Kernel«, »Userland (Initramfs, Rootfilesystem)« oder »Hardware« ein.

Bootprobleme: Überprüfen Sie als Erstes, ob der tftp-Server aktiv ist; eventuell ist er noch einmal neu zu starten. Als Zweites sollten Sie sicherstellen, dass die zu bootenden Dateien `initramfs.uboot` und `uimage.uboot` auf dem tftp-Server abgelegt sind. Schließen Sie eine serielle Schnittstelle an und lassen Sie sich die Boot-Meldungen des Raspberry Pi anzeigen.

Kernel: Sollte der Kernel nicht ordnungsgemäß booten, verwenden Sie den Kernel von der Webseite, den Sie im Ordner `8./` unter dem Namen `uimage.uboot` finden. Funktioniert dieser Kernel, gehen Sie noch einmal die Konfiguration durch. Achten Sie auf die Meldungen, die der Kernel beim Booten (über die serielle Schnittstelle) ausgibt.

Userland: Testen Sie eine Fehlfunktion des Userlands mit dem Initramfs von der Webseite. Schauen Sie sich dann die Ausgaben Ihres Systems an und versuchen Sie, die Probleme auf eine Komponente innerhalb des Userlands einzugrenzen. Wenn Sie sich einloggen können, sehen Sie sich per `ps` die aktiven Rechenprozesse an. Mit `lsmod` können Sie feststellen, ob der Treiber für das Display erfolgreich geladen wurde.

Hardware: Scheinen Kernel und Userland in Ordnung zu sein, doch ist auf dem Display keine Ausgabe zu sehen, ist das Display möglicherweise falsch verdrahtet. Stellen Sie sicher, dass die Pinbelegung Ihres Displays identisch ist mit der hier vorgestellten Pinbelegung. Überprüfen Sie noch einmal die komplette Verdrahtung. Haben Sie die Hintergrundbeleuchtung vielleicht an 3.3 V und nicht an 5 V angeschlossen?

Beispiel 8-8 *Vollständiges Skript zur Systemgenerierung <postbuild.sh>*

```
#!/bin/sh

# MARK A: Netzwerk starten
echo "Netzwerk starten..."
install -m 755 ../userland/target/S41udhcpc $1/etc/init.d/

# MARK B: Zeit setzen
echo "Zeit setzen..."
```

```
install -m 755 ../userland/target/S52ntp $1/etc/init.d/

# MARK C: SD-Karte einhaengen
echo "SD-Karte einhaengen..."
mkdir $1/boot
mkdir $1/data
echo "/dev/mmcblk0p1   /boot   vfat   defaults   0   0">>$1/etc/fstab
echo "/dev/mmcblk0p2   /data   ext4   defaults   0   0">>$1/etc/fstab

# MARK D: Webserver
echo "Webserver..."
mkdir -p $1/var/www/cgi-bin
install -m 755 ../userland/target/S51httpd $1/etc/init.d/
install ../userland/target/httpd.conf $1/etc/
install -D ../userland/target/index.html $1/var/www/
install -D -m 755 ../userland/target/ps.cgi $1/var/www/cgi-bin/

# MARK E: Firewall
echo "Firewall..."
install -m 755 ../userland/target/S45firewall $1/etc/init.d/
install -m 755 ../userland/target/fw_open.sh $1/usr/sbin/
install -m 755 ../userland/target/fw_up.sh $1/usr/sbin/

# MARK F: Usermanagement
echo "Usermanagement..."
../scripts/modifyshadow.sh root $1
../scripts/modifyshadow.sh default $1

# MARK G: Messagebox
echo "Display..."
install -m 755 ../userland/target/S53showip $1/etc/init.d/
install -m 755 ../userland/target/showtime $1/usr/bin/
install -D -m 755 ../userland/target/showmsg.php $1/var/www/cgi-bin/
echo "Modifying php.ini..."
sed -i "s/^;cgi.redirect_status_env =$/cgi.redirect_status_env = \"yes\"/g" \
    output/target/etc/php.ini
sed -i "s/^;cgi.force_redirect =/cgi.force_redirect = 0/g"\
    output/target/etc/php.ini
```

Tabelle 8-4 *Dateien, die für das System benötigt werden*

Dateiname	Verzeichnis auf dem Entwicklungssystem	Verzeichnis auf dem Zielsystem	Bedeutung
S41udhcpc	userland/target/	/etc/init.d/	Startskript DHCP-Client
S45firewall	userland/target/	/etc/init.d/	Startskript Firewall
S50dropbear	userland/target/	/etc/init.d/	Startskript SSH-Server
S51httpd	userland/target/	/etc/init.d/	Startskript Webserver
S52ntp	userland/target/	/etc/init.d/	Startskript Zeitserver

➔

Dateiname	Verzeichnis auf dem Entwicklungssystem	Verzeichnis auf dem Zielsystem	Bedeutung
fw_open.sh	userland/target/	/usr/sbin/	Skript zur Deaktivierung der Firewall
fw_up.sh	userland/target/	/usr/sbin/	Skript zur Aktivierung der Firewall
modifyshadow.sh	scripts/		Generieren von Passwort-Hashes
networkboot.sh	scripts/		Postimage-Skript

Dateiname	Verzeichnis auf dem Entwicklungssystem	Verzeichnis auf dem Zielsystem	Bedeutung
S53showip	userland/target/	/etc/init.d/	Startskript Displayapplikation
showtime.sh	userland/target/	/usr/bin/	Displayapplikation
showmsg.php	userland/target/	/var/www/cgi-bin/	Webapplikation
httpd.conf	userland/target/	/etc/	Webserverkonfiguration
mmc.txt	userland/target/		Konfiguration Gerätedateien
hd44780.c	driver/hd44780/	/lib/modules/	Display-Gerätetreiber
Makefile.c	driver/hd44780/		Gerätetreiber-Makefile
hd44780.mk	buildroot-2013.05/package/hd44780/		Bauvorschrift Gerätetreiber
Config.in	buildroot-2013.05/package/hd44780/		Konfiguration Gerätetreiber
postbuild.sh	scripts/		Systemgenerierung

Tabelle 8-5
Zusätzliche oder geänderte Dateien für die Messagebox

Beispiel 8-9
Befehlssequenz zum Systemaufbau der Messagebox

```
#####################################################
# PHP installieren
#
quade@felicia:~/embedded/raspi> cd buildroot-2013.05
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # [Package Selection for the target][Interpreter languages and scripting]
[php]
quade@felicia:~/embedded/raspi/buildroot-2013.05> cd ../scripts
quade@felicia:~/embedded/raspi/scripts> vi postbuild.sh
#sed -i "s/^;cgi.redirect_status_env =$/cgi.redirect_status_env = \"yes\"/g" \
#    output/target/etc/php.ini
#sed -i "s/^;cgi.force_redirect =/cgi.force_redirect = 0/g"\
#    output/target/etc/php.ini

#####################################################
```

```
# Webseite integrieren
quade@felicia:~/embedded/raspi/scripts> cd \
   ~/embedded/raspi/userland/target
quade@felicia:~/embedded/raspi/userland/target> gedit httpd.conf
    # *.php:/usr/bin/php-cgi ergänzen
quade@felicia:~/embedded/raspi/userland/target> gedit showmsg.php
    # Webinterface erstellen
quade@felicia:~/embedded/raspi/userland/target> cd ../../scripts
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh
    # install showmsg.php /var/www/cgi-bin/

#####################################################
# Applikation, die die IP-Adresse beim Booten ausgibt
#
quade@felicia:~/embedded/raspi/scripts> cd \
   ~/embedded/raspi/userland/target
quade@felicia:~/embedded/raspi/userland/target> gedit mmc.txt
    # Characterdevice /dev/hd44780 eintragen
    # /dev/hd44780    c       222     0     0     248     0     0     0
quade@felicia:~/embedded/raspi/userland/target> gedit showtime.sh
    # Skript erstellen
quade@felicia:~/embedded/raspi/userland/target> gedit S53showip
    # Skript erstellen, das den Treiber lädt,
      die IP extrahiert und ausgibt
quade@felicia:~/embedded/raspi/userland/target> cd ../../scripts
quade@felicia:~/embedded/raspi/scripts> gedit postbuild.sh
    # install showtime.sh /usr/bin
    # install S53showip /etc/init.d

#####################################################
# Integration des Treibers in Buildroot
#
# quade@felicia:~/embedded/raspi/scripts> PATH=$PATH:...
quade@felicia:~/embedded/raspi/scripts> cd \
   ~/embedded/raspi/buildroot-2013.05/package
quade@felicia:~/embedded/raspi/buildroot-2013.05/package> mkdir hd44780
quade@felicia:~/embedded/raspi/buildroot-2013.05/package> cd hd44780
quade@felicia:~/embedded/raspi/buildroot-2013.05/package/hd44780>
   \gedit Config.in
quade@felicia:~/embedded/raspi/buildroot-2013.05/package/hd44780>
   \gedit hd44780.mk
quade@felicia:~/embedded/raspi/buildroot-2013.05/package/hd44780> cd ..
quade@felicia:~/embedded/raspi/buildroot-2013.05/package> gedit Config.in
quade@felicia:~/embedded/raspi/buildroot-2013.05/package> cd ..
quade@felicia:~/embedded/raspi/buildroot-2013.05> make menuconfig
    # hd44780 aktivieren
quade@felicia:~/embedded/raspi/buildroot-2013.05> make
quade@felicia:~/embedded/raspi/buildroot-2013.05>
# Raspberry Pi rebooten und testen
```

Anhänge

A Crashkurs Linux-Shell

Die Shell ist ein interaktives Programm, mit dem Kommandos an den Kernel weitergereicht und die Ergebnisse in einem Terminalfenster dargestellt werden (Abb. A-1). Die Shell arbeitet zeilenorientiert. Dabei gibt die Shell einen Text, den sogenannten Prompt, aus, hinter dem der Anwender sein Kommando und zugehörige Parameter eintippt. Drückt der Anwender die Enter-Taste, wertet die Shell das Kommando aus, arbeitet es ab und gibt die Ergebnisse aus.

Einfache Shells verwenden als Prompt häufig das Dollarzeichen »$«, wenn die Shell die Rechte eines normalen Benutzers besitzt, und ein Hashmark »#«, falls die Shell Root-Rechte hat. Häufig wird im Prompt auch der Username und das aktuelle Verzeichnis angegeben. Das ist typischerweise konfigurierbar.

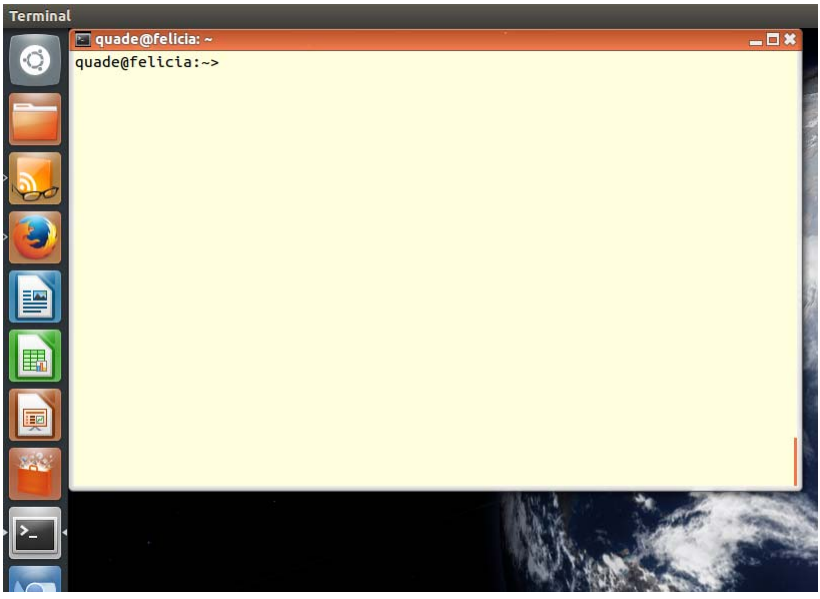


Abb. A-1
Terminalfenster mit Shellprompt

Die Ausgaben der Shell werden, wie man sagt, auf »stdout« (standard out) ausgegeben. In der Praxis ist dies das Terminalfenster. Fehlerausgaben erscheinen ebenfalls im Terminalfenster, auch wenn die Applikation

selbst die Daten auf »stderr« schreibt. Die Eingaben kommen von »stdin«, was typischerweise die Tastatur ist.

Jedes Linux-Programm und damit auch die Shell verfügt über einen Kontext, der auch als Umgebung beziehungsweise Environment bezeichnet wird. Dort ist in Umgebungsvariablen festgelegt, in welchen Verzeichnissen die Shell nach Programmen sucht, deren Namen der Nutzer nach dem Prompt eingegeben hat (Variable PATH), oder auf welches Verzeichnis sich Dateibefehle beziehen. Dieses Verzeichnis ist elementar und wird als »Current Working Directory« bezeichnet. Das Kommando `ls` gibt beispielsweise die Namen der Verzeichnisse und Dateien im »Current Working Directory« aus.

Häufig verarbeiten Shellkommandos Dateien, die durch Pfad und Dateiname referenziert werden. Eine Pfadangabe kann relativ oder absolut sein. Ein absoluter Pfad liegt vor, falls der Pfad mit »/« beginnt, ansonsten handelt es sich um einen relativen Pfad. Bezugspunkt des relativen Pfads ist immer das aktuelle Verzeichnis. Zwei Punkte (»..«) in der Pfadangabe verweisen auf das darüber liegende Verzeichnis.

Datei- oder Verzeichnisnamen müssen nicht immer vollständig angegeben werden. Viele Shells verarbeiten reguläre Ausdrücke, bei denen Metazeichen wie »*« oder »?« durch beliebig viele beziehungsweise ein beliebiges Zeichen ersetzt werden.

Neben dem Namen besitzen Dateien noch einige weitere Attribute, darunter die Zugriffsrechte (siehe Abschnitt 7.1.3). Zugriffsrechte werden für den Besitzer (owner), eine Gruppe (group) und alle übrigen (world) angegeben. Linux kennt standardmäßig die Rechte »lesen«, »schreiben« und »ausführen«. Diese Rechte werden unter anderem für den Besitzer, die Gruppe und alle übrigen mit jeweils einer Dreiergruppe der Zeichen »r«, »w« und »x« repräsentiert. Ist ein Recht nicht gesetzt, wird »-« verwendet. Die unten stehende Ausgabe des Kommandos `ls` ist damit folgendermaßen zu interpretieren: Der Besitzer »syslog« darf auf die Datei `/var/log/syslog` lesend und schreibend zugreifen. Ausführen darf er die Datei nicht. Die Gruppe »adm« darf nur lesend zugreifen und alle übrigen dürfen weder lesend noch schreibend die Datei verwenden.

```
-rw-r----- 1 syslog adm 264631 Okt  6 21:18 /var/log/syslog
```

Der sogenannte Superuser beziehungsweise ein User, der mit Root-Rechten ausgestattet ist, darf die Datei aber in jedem Fall lesen. Während es auf einem Desktop-System den klassischen Superuser aus Sicherheitsgründen nicht mehr gibt, wird dieser auf einem eingebetteten System häufig noch eingerichtet.

Auf Entwicklungssystemen (Ubuntu) kommt als Shell sehr häufig die sehr leistungsstarke `bash` (Bourne again shell) zum Einsatz, im Embedded Device schlankere Varianten wie beispielsweise die `ash`.

A.1 Elementare Kommandos zur Dateiverwaltung

`pwd`

Das Kommando »print working directory« gibt den Namen des aktuellen Verzeichnisses aus.

`cd <Pfad>`

Mit `cd` wird von dem aktuellen Verzeichnis in das mit »Pfad« angegebene Verzeichnis gewechselt. Ohne Parameter wechselt das Kommando in das sogenannte Heimatverzeichnis, das auch über die Tilde »~« erreicht werden kann. Damit haben die Kommandos »cd« und »cd ~« die gleiche Wirkung. Um in der Verzeichnishierarchie eine Ebene höher zu kommen, wird »cd ..« verwendet. Ein einzelner Punkt steht für das aktuelle Verzeichnis.

Kommando	Current Working Directory (CWD)
cd	/home/quade/
cd embedded	/home/quade/embedded/
cd .	/home/quade/embedded/
cd ..	/home/quade/
cd ../../tmp	/tmp/
cd ~	/home/quade/
cd /tmp	/tmp/
cd /home/quade	/home/quade/
cd -	/tmp/

Tabelle A-1 *CWD bei Navigation durch das Dateisystem*

`ls`

Mit diesem Kommando werden die Namen von Verzeichnissen und Dateien ausgegeben. Die Optionen sind vielfältig. »-l« (long) gibt Detailinformationen aus, »-t« (time) sortiert die Ausgabe anhand der Zugriffszeit und »-r« (reverse) in umgekehrter Reihenfolge. Sehr praktisch ist das Kommando »ls -lrt«. Dieses listet die Dateien so auf, dass ganz unten die jeweils als Letztes bearbeitete Datei auftaucht.

mkdir <Pfad>

Falls »Pfad« mit einem Slash (»/«) beginnt, legt das Kommando ein Verzeichnis ausgehend vom Root-Verzeichnis an, ansonsten vom aktuellen Verzeichnis.

```
quade@felicia:~/embedded> mkdir sample_dir
quade@felicia:~/embedded> ls
application/  driver/  files/  qemu/  raspi/  sample_dir/
```

rmdir <verzeichnis>

Das Kommando löscht das Verzeichnis mit dem Namen »verzeichnis«. Damit das Verzeichnis wirklich gelöscht wird, darf es keine (auch keine versteckten) Dateien enthalten. Alternativ kann das Kommando »rm -rf <verzeichnis>« eingesetzt werden, das sowohl den Inhalt als auch das Verzeichnis »verzeichnis« selbst löscht.

```
cyrielles@felicia:~/embedded> rmdir sample_dir
```

cat <dateiname>

Der Befehl gibt den Inhalt der Datei »dateiname« auf »stdout« aus.

cp <quelle> <ziel>

Der Befehl kopiert die Datei »quelle« auf die Datei »ziel«. Quelle und Ziel können neben dem Dateinamen auch eine Pfadangabe enthalten.

mv <quelle> <ziel>

Das Kommando benennt die Datei »quelle« in »ziel« um.

```
eva@felicia:~/embedded/raspi/userland/tmp> ls
beispiel.datei
eva@felicia:~/embedded/raspi/userland/tmp> mv
beispiel.datei neuer_name.datei
eva@felicia:~/embedded/raspi/userland/tmp> ls
neuer_name.datei
```

rm <datei>

Das Kommando löscht die Datei mit dem Namen »datei«. Wird die Option »-r« (recursive) angegeben, lassen sich auch Verzeichnisse löschen. Die Option »-f« unterdrückt eine Fehlermeldung, falls die zum Namen »datei« gehörende Datei nicht existiert.

```
eva@felicia:~/embedded/raspi/userland/tmp> rm neuer_name.datei
eva@felicia:~/embedded/raspi/userland/tmp> ls
eva@felicia:~/embedded/raspi/userland/tmp>
```

```
chmod <rechte> <name>
```

Das Kommando `chmod` ändert die Zugriffsrechte für das Verzeichnis oder die Datei »name«. Die neuen Zugriffsrechte können auf zwei Arten angegeben werden: als (oktal dargestellte) Bitfolge oder symbolisch. Eindeutiger ist dabei die Angabe als Bitfolge. Dabei werden drei Oktalziffern (von denen jede einen möglichen Wert von 0 bis 7 hat) verwendet. Jede Oktalziffer repräsentiert drei Bit, jeweils eines für Lesen, eines für Schreiben und eines für Ausführen. Die Bitkombination `101` repräsentiert demnach: Das Lesen und Ausführen ist erlaubt, das Schreiben aber nicht. Oktal wird dieses Bitmuster durch eine »5« repräsentiert. Das Kommando »chmod 777 /tmp/foo« ermöglicht beispielsweise jedem (owner, group, world) den beliebigen Zugriff (lesen, schreiben, ausführen) auf die Datei `/tmp/foo`.

```
chown <owner.group> <name>
```

Der Befehl setzt den Namen des Besitzers und der Gruppe für die Datei oder das Verzeichnis »name«. Beispiel: »chown nobody.nobody /tmp/foo«.

```
install <optionen> <quelle> <ziel>
```

Der Befehl ist eine Kombination aus »cp«, »chown« und »chmod«. Er kopiert die Datei »quelle« an die Stelle »ziel« und setzt dabei die über die Option »-m« angegebenen Zugriffsrechte (als dreistelligen Oktalwert). Über die Option »-o« lässt sich der Besitzer festlegen, mit »-g« die Gruppe.

```
ln <file> <link>
```

Ein Link ist eine Referenz auf eine bestehende Datei. Durch den Link ist die Datei über zwei Namen erreichbar. Es werden Hardlinks von den Softlinks (auch symbolische Links genannt) unterschieden. Bei Hardlinks erhält die Datei tatsächlich einen zweiten, unabhängigen Namen, bei Softlinks (Option »-s«) wird eine Datei erstellt, die auf die eigentliche Datei verweist. Damit sind Hardlinks nur innerhalb eines Dateisystems anwendbar, während Softlinks dateisystemübergreifend eingesetzt werden können.

```
dd if=<infile> of=<outfile> bs=<blocksize> count=<count>
```

Diskdump kopiert die Datei »infile« blockweise auf die Datei »outfile«, ähnlich dem Kommando `cp`. Allerdings lässt sich hier über die Parameter »blocksize« und »count« die Menge der zu kopierenden Daten ange-

ben. Verwendet man als »infile« /dev/zero, lassen sich mit dem Kommando auch Dateien einer bestimmten Größe erstellen, die nur aus Nullen bestehen.

```
man <kapitel> <befehl>
```

Unixsysteme bieten in den Handbuchseiten Informationen zu den wichtigsten Befehlen. man (manual page) zeigt die Manual-Seite zum angegebenen Befehl »befehl« in einem Terminal an. Um die Anzeige zu beenden, geben Sie »q« ein. Ansonsten ist eine Steuerung der Anzeige mit den klassichen vi-Befehlen möglich. Wenn Sie beispielsweise ein »G« eingeben, wird das Ende der Seite angezeigt, »g« springt zum Anfang.
Die Handbuchseiten sind in mehreren Kapiteln organisiert. Kapitel 1 enthält die Shellbefehle, Kapitel 2 Systemcalls und Kapitel 3 Bibliotheksfunktionen. Da manchmal Shellbefehle den gleichen Namen haben wie Systemcalls, ist es in solchen Fällen notwendig, das jeweilige Manual-Kapitel als Option mit anzugeben: »man 1 write« zeigt also das Kommando write an, während »man 2 write« den gleichnamigen Systemcall anzeigt.

A.2 Systemkommandos

```
ps
```

Das Kommando gibt die Liste der Prozesse aus.

```
lilith@felicia:~/embedded$ ps
  PID TTY          TIME CMD
 3592 pts/2    00:00:00 bash
 3757 pts/2    00:00:17 vi
15628 pts/2    00:00:00 bash
15629 pts/2    00:00:00 ps
```

Die Spalte PID repräsentiert die Identifikationsnummer eines Jobs, über den man dem Job auch per kill die Aufforderung senden kann, sich zu beenden. TTY gibt das so genannte Controlling Terminal an und CMD den Namen und die Aufrufparameter der Task.
Der Befehl ps ist auf die unterschiedlichste Weise parametrierbar. Häufig eingesetzt wird »ps -axwu«. Damit werden sämtliche Prozesse ausgegeben, nicht nur die zum Aufrufer gehörenden. Die Option »ps -ce« listet die Prioritäten auf.

```
kill <pid>
```

Mit kill schickt man der Task mit der PID »pid« ein sogenanntes Signal. Falls die Task das Signal nicht abfängt, wird sie sich beenden. Da es

unterschiedliche Signale gibt, ist es auch möglich das gewünschte Signal per Nummer oder Name anzugeben. Bis auf das Signal mit der Nummer 9 (SIGKILL) kann die Task sämtliche Signale abfangen. Daher eignet sich das Signal 9 (SIGKILL), um eine Task sicher zu beenden. Die Task mit der PID 1234 beendet man also mit dem Kommando »kill -9 1234«.

reboot

Dieses Kommando führt einen Neustart des Systems durch.

poweroff

Das Kommando fährt den Rechner herunter und schaltet — falls technisch möglich — die Stromversorgung ab.

lsblk

Dieses auf dem Entwicklungsrechner vorhandene Kommando listet die blockorientierten Geräte (Blockdevices) auf, also Festplatten, SD-Karten, USB-Sticks, DVD-Laufwerke und deren jeweilige Partitionen. Zu jedem Eintrag wird die Gerätenummer (Major- und Minornummer) und die Größe angegeben. Eine »0« in der Spalte »RM« (removable) zeigt an, dass die Disk (TYPE »disk«) beziehungsweise Partition (TYPE »part«) fest eingebaut ist. Falls die Geräte gemountet (eingehängt) sind, wird auch das Verzeichnis angegeben, über das man sich die Inhalte der Blockgeräte ansehen kann.

```
jonathan@felicia:~$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 149,1G  0 disk
├─sda1   8:1    0  72,1G  0 part /
├─sda2   8:2    0  72,1G  0 part /home
├─sda3   8:3    0   4,9G  0 part
├─sda4   8:4    0  47,1M  0 part
sdb      8:16   1   1,9G  0 disk
├─sdb1   8:17   1    56M  0 part
└─sdb2   8:18   1   1,8G  0 part
jonathan@felicia:~$
```

lsinitramfspara <initramfsfile>

Das für die Entwicklung eingebetteter Systeme nützliche Kommando `lsinitramfs` gibt den Inhalt einer Initramfs aus.

```
julia@felicia:~/embedded/raspi/userland>lsinitramfs -l initramfs.cpio.gz
drwxrwxr-x  11 root   root          0 Sep 22 16:31 .
```

```
drwx------   2 root   root          0 Sep 22 16:31 lost+found
lrwxrwxrwx   1 root   root         11 Sep 10 18:56 linuxrc -> bin/busybox
drwxrwxr-x   2 root   root          0 Sep 10 18:56 bin
lrwxrwxrwx   1 root   root          7 Sep 10 18:56 bin/ash -> busybox
lrwxrwxrwx   1 root   root          7 Sep 10 18:56 bin/bash -> busybox
[...]
crw-r--r--   1 root   root     5,   1 Sep 22 16:31 dev/console
crw-r--r--   1 root   root     1,   3 Sep 22 16:31 dev/null
crw-r--r--   1 root   root     4,   0 Sep 22 16:31 dev/tty0
crw-r--r--   1 root   root     4,   1 Sep 22 16:31 dev/tty1
crw-r--r--   1 root   root   204,  64 Sep 22 16:31 dev/ttyAMA0
drwxrwxr-x   2 root   root          0 Sep 22 16:31 etc
-rw-rw-r--   1 root   root        114 Sep 22 16:31 etc/inittab
-rwxrwxr-x   1 root   root        121 Sep 22 16:31 etc/rcS
-rw-rw-r--   1 root   root         26 Sep 22 16:31 etc/httpd.conf
drwxrwxr-x   2 root   root          0 Sep 22 16:31 proc
drwxrwxr-x   2 root   root          0 Sep 22 16:31 sys
drwxrwxr-x   3 root   root          0 Sep 22 16:31 var
drwxrwxr-x   3 root   root          0 Sep 22 16:31 var/www
drwxrwxr-x   2 root   root          0 Sep 22 16:31 var/www/cgi-bin
-rwxrwxrwx   1 root   root        132 Sep 22 16:31 var/www/cgi-bin/ps.cgi
-rw-rw-r--   1 root   root         55 Sep 22 16:31 var/www/index.html
lrwxrwxrwx   1 root   root          9 Sep 22 16:31 init -> sbin/init
```

sudo <befehl>

Mit `sudo` kann ein User ein Programm unter der Identität eines anderen Users — meist des Superusers — ausführen lassen. Sudo fragt dabei vor der Ausführung das Passwort ab. In der Kombination mit dem Befehl `su` lässt sich damit auch eine Rootshell öffnen (erkennbar am Prompt), die alle eingegebenen Befehle mit Root-Rechten aufruft.

```
quade@felicia:~/embedded>sudo su
[sudo] password for quade:
root@felicia:/home/quade/embedded#
```

df -h

Das Kommando »diskfree« (df) zeigt den verfügbaren, belegten und freien Speicherplatz angeschlossener Festplatten, SD-Karten und anderer Blockgeräte an. Die Option »-h« bewirkt dabei eine Ausgabe im »human readable« Format. Dabei werden die Zahlenangaben direkt in Mega- oder Gigabyte angezeigt.

free

Mit free wird der verfügbare, belegte und der freie Hauptspeicherplatz ausgegeben.

```
quade@felicia:~> free
             total       used       free     shared    buffers     cached
Mem:       3790224    2868788     921436          0     249804    1210152
-/+ buffers/cache:    1408832    2381392
Swap:      4194300          0    4194300
quade@felicia:~>
```

A.3 Grundlegende Befehle zum Netzwerkmanagement

ifconfig <device> <ipadresse>

Ohne Parameter zeigt ifconfig die gegenwärtige Konfiguration, insbesondere IP-, Netzwerk- und Broadcast-Adresse sämtlicher Netzwerkgeräte an. Hauptsächlich wird es aber genutzt, um damit die IP-Adresse »ipadresse« des Netzwerkgerätes »device« zu setzen. Über die zusätzlichen Schlüsselwörter »netmask=« und »broadcast=« lassen sich auch diese Parameter konfigurieren.

```
# ifconfig eth0 192.168.178.95
# ifconfig
eth0  Link encap:Ethernet  HWaddr B8:27:EB:F6:50:2A
      inet addr:192.168.178.95  Bcast:192.168.178.255  Mask:255.255.255.0
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:11 errors:0 dropped:0 overruns:0 frame:0
      TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:1781 (1.7 KiB)  TX bytes:1331 (1.2 KiB)

lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      UP LOOPBACK RUNNING  MTU:65536  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

route

Mit diesem Kommando lässt sich die Routing-Tabelle des Systems anzeigen. Die Option »-n« (numeric) stellt angegebene Rechner über deren IP-Adresse und nicht deren Name dar. Über das Kommando »route

add« lassen sich Einträge hinzufügen, über »route del« Einträge löschen.

```
# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref Use Iface
0.0.0.0         192.168.178.1   0.0.0.0         UG    0      0     0 eth0
192.168.178.0   0.0.0.0         255.255.255.0   U     0      0     0 eth0
```

`ping <ipadresse>`

Mit `ping` lässt sich die netzwerktechnische Erreichbarkeit des Systems mit der IP-Adresse »ipadresse« überprüfen.

B Crashkurs vi

Der vi ist ein konsolenbasierter Editor, der nicht nur auf Desktop-Systemen, sondern auch in eingebetteten Geräten zu finden ist. Er zeichnet sich durch eine hohe Funktionsvielfalt aus, aber eben auch durch eine sehr gewöhnungsbedürftige, zugleich sehr effiziente Bedienung.

Der Schlüssel zum Verständnis des vi besteht darin, sich der zwei Modi bewusst zu werden. Normale Editoren kennen nur einen Einfügemodus, der vi verfügt zusätzlich jedoch über einen Befehlsmodus. Dieser hat den Vorteil, dass sich der Editor den letzten Befehl — der auch aus einer Eingabe bestehen kann — merkt. Er lässt sich dann mit einer einzigen Taste, dem Punkt, wiederholt anwenden.

Es gibt zwei Gruppen von Kommandos: Tastaturkommandos und komplexe Kommandos.

Im Befehlsmodus hat jede Taste der Tastatur eine Bedeutung (siehe Tabelle B-1). Die Taste »i« beispielsweise schaltet in den Einfügemodus, per »ESC« wird der Einfügemodus verlassen und der Befehlsmodus aktiviert, »h« bewegt den Cursor nach links, »G« springt an das Dateiende und »u« macht den letzten Befehl rückgängig. Vielen Tastaturkommandos lässt sich noch eine Zahl voranstellen, die die Häufigkeit repräsentiert, mit der das Kommando ausgeführt wird. Zusätzlich kann der Bereich angegeben werden, auf den das Kommando wirkt. Wird die Taste beispielsweise zweimal angeschlagen, bezieht sich das Kommando auf die aktuelle Zeile, in der sich der Cursor befindet. Wird der Taste eine schließende, geschweifte Klammer angefügt (»}«), soll der Absatz betrachtet werden.

Kommando	Beschreibung
i,I,a,A,o,O	Wechsel in den Eingabemodus
ESC	Wechsel in den Befehlsmodus
.	Den letzten Befehl wiederholen
:	Eingabe eines komplexen Befehls
h,j,k,l	Cursorbewegungen (links, runter, hoch, rechts)
ZZ	Speichern und verlassen
gg	Cursor auf Dateianfang setzen
G	Cursor auf Dateiende setzen

➔

Tabelle B-1 *Wichtige vi-Tastaturkommandos*

Kommando	Beschreibung
u	Undo (letzten Befehl rückgängig machen)
<Strg>r	Reverse Undo
/	Vorwärtssuche
?	Rückwärtssuche
*	Aktuelles Wort suchen
y	Kopieren
p, P	Einfügen
d	Löschen
x	Einzelnes Zeichen löschen

Wird im Befehlsmodus beispielsweise die Tastenfolge »7yy« eingetippt, werden die nächsten sieben Zeilen in einen internen Puffer kopiert: »7« ist der Wiederholungsfaktor, »y« (yank) ist das Kommando für »kopieren« und das zweite »y« limitiert die Kopieraktion auf die aktuelle Zeile. Zum Einfügen der kopierten sieben Zeilen bewegen Sie den Cursor an die gewünschte Stelle und drücken (im Befehlsmodus) die Taste »p« (paste). Die kopierten Zeilen werden *nach* der aktuellen Zeile eingefügt. Bei einem »P« würden die Zeilen *vor* der aktuellen Zeile eingefügt.

Geben Sie nach diesem Schema »3y}« ein, um drei Absätze zu kopieren.

Weitere nützliche Tastaturkommandos sind »/«, »?«, »*«, »n«, »d«, »x« und »Z«. Der Vorwärts-Slash leitet eine Suche ein. Direkt nach Eingabe des Slash geben Sie den Suchstring ein. Für die Rückwärtssuche verwenden Sie das Fragezeichen (»?«). Mit »n« setzen Sie die Suche fort. Um eine Stelle zu suchen, an der ebenfalls das Wort steht, auf dem sich der Cursor gerade befindet, müssen Sie nur einen Stern (»*«) eingeben. Zum Löschen nehmen Sie die Taste »d« (delete). Zweimal gedrückt, wird die aktuelle Zeile gelöscht, folgt dem »d« eine Leerzeichen (Space), wird das Zeichen unter dem Cursor gelöscht. Hierfür kann aber auch alternativ direkt ein »x« gedrückt werden. Mit der Kombination »xp« werden beispielsweise zwei Buchstaben schnell vertauscht. Wichtig ist auch das Tastaturkommando »Z«. Zweimal gedrückt, wird die aktuell bearbeitete Datei gespeichert und der vi verlassen.

Komplexe Befehle, beispielsweise *suchen und ersetzen*, werden über den Doppelpunkt »:« eingeleitet. Nach Eingabe des Doppelpunktes erscheint unten eine Befehlszeile, in der die Kommandos spezifiziert werden. Die Befehle beginnen typischerweise mit einer optionalen Bereichsangabe. Der Punkt steht dabei für die aktuelle Cursorposition, das Prozentzeichen repräsentiert die komplette Datei. Darüber hinaus lassen sich auch zuvor mit dem Tastaturkommando »m« gesetzte Marken ver-

wenden. Die nachfolgenden Kommandos sind vielfältig. »s« steht für *substitute*, also suchen und ersetzen. So wird beispielsweise per »:%s/gelb/rot/« einmal pro Zeile das Wort »gelb« durch »rot« ersetzt. Damit alle »gelb« ersetzt werden, hängen Sie noch ein »g« (global) an: »:%s/gelb/rot/g«. Per »w« wird die Datei geschrieben und »q« verlässt den Editor. Um den Editor zu verlassen, ohne Änderungen zu schreiben, wird »q!« eingegeben.

Denken Sie daran: Nur wer fleißig ESC und ».« verwendet, kann die Leistungsfähigkeit des vi richtig nutzen. Tabelle B-1 zeigt die wichtigsten Tastenbefehle und Tabelle B-2 zeigt ausgewählte, nützliche Kommandos.

Um den vi zu erlernen, bietet sich das Programm `vimtutor` an, das auf vielen Linux-Systemen bereits vorinstalliert ist und durch Eingabe von `vimtutor` in einem Terminal aufgerufen wird (Abb. B-1).

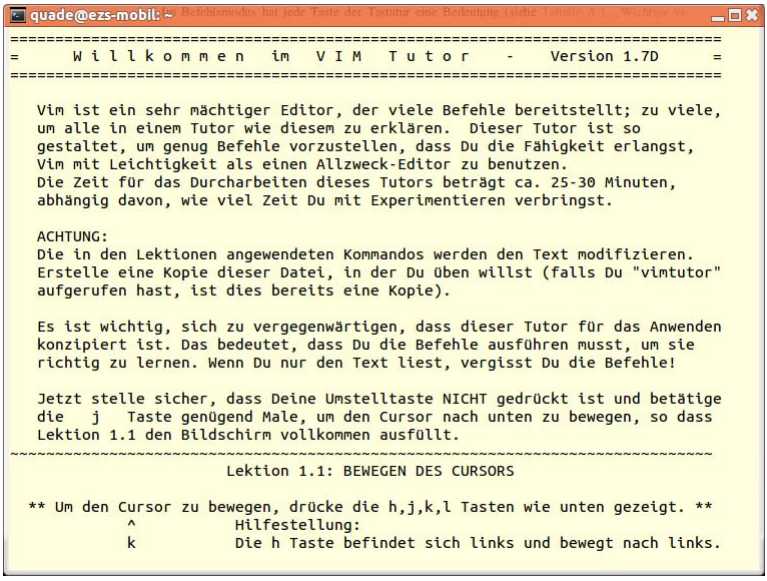


Abb. B-1
Vimtutor zeigt die wichtigsten Editierfunktionen in 30 Minuten.

Kommando	Beschreibung
!!date	Fügt das aktuelle Datum ein
:w	Zwischenspeichern
d/hallo	Löschen bis zum nächsten »hallo«
dd	Aktuelle Zeile löschen
:n <dateiname>	Datei <dateiname> laden
:ls	Im Editor befindliche Dateien listen
:n#	Vorhergehende Datei editieren
:set nu	Zeilennummern einschalten

➔

Tabelle B-2
Nützliche vi-Kommandos

Kommando	Beschreibung
:set nonu	Zeilennummern ausschalten
:set ic aw tw=70 ai	Groß-/Kleinschreibung ignorieren, automatisches abspeichern, Textweite einstellen und automatisch einrücken

C Git im Einsatz

Git ist ein dezentrales Versionskontrollsystem, das breite Anwendung im Open-Source-Umfeld gefunden hat. Es wird zur Quellcodeverwaltung von sehr vielen Open-Source-Projekten eingesetzt, insbesondere im Umfeld eingebetteter Linux-Systeme. Viele dieser Projekte nutzen dabei die Dienste der Social-Coding-Plattform Github ([http://www.github.com]), die ein einfaches und kostenfreies Hosting von mit Git-verwalteten Projekten ermöglicht.

Git hat gegenüber klassischen Versionskontrollsystemen wie SVN den Vorteil, dass es sehr schnell und vor allem dezentral ist. Der Anwender arbeitet lokal und kann selbst entscheiden, wann er Daten austauscht.

C.1 Unterschiedliche Git-Bereiche

Git arbeitet mit drei Bereichen: dem Arbeitsbereich (working area), dem Zwischenbereich (staging area) und dem Repository. Der Arbeitsbereich enthält die Projektverzeichnisse und -dateien. Hier findet die normale Entwicklungsarbeit statt. Ist ein Stand erreicht, der gesichert werden kann, wird er per `git add` in den Staging-Bereich übernommen. Soll er zudem versioniert werden, rufen Sie `git commit` auf. Damit landen die Daten im Repository, das die komplette Projekthistorie abspeichert. Um aus dem Repository einen Stand in den Arbeitsbereich zu extrahieren, dient das Kommando `git checkout`. Staging und Repository befinden sich übrigens nebst Konfiguration im Verzeichnis `.git/`.

Ein neues Git-Repository kann auf zwei Arten angelegt werden: Ein leeres Repository wird per `git init` angelegt. Ein gefülltes Repository kann per `git clone <quelle> <ziel>` als Kopie eines bestehenden Projektes erzeugt werden. Das bestehende Projekt kann dabei auch remote auf einem anderen Rechner, beispielsweise auf Github, liegen.

```
quade@felicia:~/embedded/application/gpioappl> git init
Initialized empty Git repository in /home/quade/embedded/application/
gpioappl/.git/
quade@felicia:~/embedded/application/gpioappl> ls -a
.  ..  .git  gpioappl.c  Makefile
```

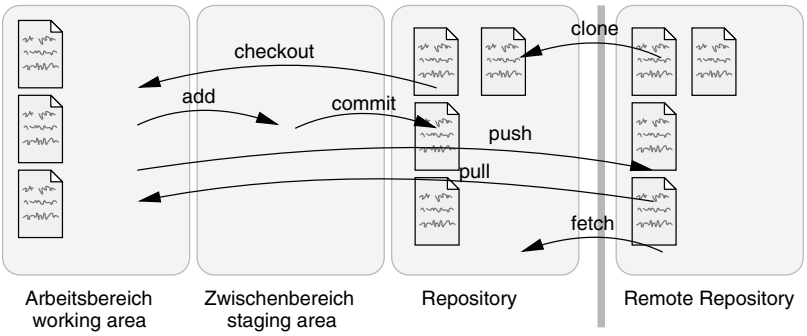


Abb. C-1 *Datentransfer zwischen den Git-Bereichen*

C.2 Dateizustände

Dateien haben für Git prinzipiell zwei Zustände: Entweder sind sie bereits bei Git angemeldet (tracked) oder eben nicht (untracked). Befinden sich Dateien im Zustand »tracked«, können diese unverändert, verändert oder »staged« (im Zwischenbereich, sodass Änderungen mit dem nächsten Commit in das Repository übernommen werden) sein. Entsprechend überführt ein `git add <filename>` eine Datei in den Staging-Bereich, ein `git commit` in das Repository. Alternativ kann auch ein `git commit -a` verwendet werden, das direkt alle Änderungen identifiziert und in das Repository ohne Umweg über den Staging-Bereich übernimmt. Mit dem häufig verwendeten Kommando `git status` lassen Sie sich den Zustand der Dateien im Arbeitsbereich anzeigen.

```
quade@felicia:~/embedded/application/gpioappl> git add \
    Makefile gpioappl.c
quade@felicia:~/embedded/application/gpioappl> git commit
[master (root-commit) 644e836] initial commit
 2 files changed, 40 insertions(+)
 create mode 100644 Makefile
 create mode 100644 gpioappl.c
```

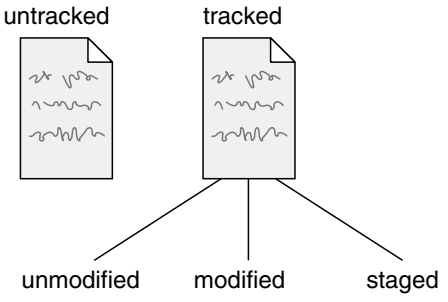


Abb. C-2 *Zustände von Dateien im Git*

C.3 Änderungen anzeigen

Durchgeführte Änderungen lassen sich per `git log -p` anzeigen. Das Kommando kann noch um die Option »-n« ergänzt werden, wobei »n« die Anzahl der anzuzeigenden Änderungen angibt. Ein `git log -p -1` gibt also die letzte Änderung aus.

C.4 Branching und Merging

Ein Git-Repository enthält typischerweise mehrere Entwicklungszweige, sogenannte Branches. Ein Branch kann Version eins und dessen Fehlerkorrekturen und ein anderer Version zwei repräsentieren. Um einen neuen Branch anzulegen, geben Sie übrigens nur `git branch <name>` ein. Per `git branch -v` lassen sich die verschiedenen Entwicklungszweige anzeigen. Der aktive Branch ist dabei durch ein Stern »*« gekennzeichnet. Per `git checkout <branchname>` wird ein Branch aktiviert. Im folgenden Beispiel wird ein neuer Branch »readme« erzeugt und aktiviert:

```
quade@felicia:~/embedded/application/gpioappl> git branch readme
quade@felicia:~/embedded/application/gpioappl> git branch -v
* master 644e836 initial commit
  readme 644e836 initial commit
quade@felicia:~/embedded/application/gpioappl> git checkout readme
Switched to branch 'readme'
quade@felicia:~/embedded/application/gpioappl> git branch -v
  master 644e836 initial commit
* readme 644e836 initial commit
```

In den neuen Branch wird beispielhaft eine neue (leere) Datei per `touch` erzeugt und in das Repository übernommen:

```
quade@felicia:~/embedded/application/gpioappl> touch Readme
quade@felicia:~/embedded/application/gpioappl> git status
# On branch readme
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   Readme
nothing added to commit but untracked files present (use "git add"
to track)
quade@felicia:~/embedded/application/gpioappl> git add Readme
quade@felicia:~/embedded/application/gpioappl> git commit
[readme 9f5c49e] Readme added
 0 files changed
 create mode 100644 Readme
```

Als Nächstes sollen die Änderungen im Zweig »readme« in den Masterzweig übernommen werden. Dazu wird zunächst der Masterzweig aktiviert. Ein `ls` demonstriert, dass dabei die neue Datei »Readme« im Arbeitsbereich (nicht im Repository) gelöscht wurde. Per `git merge <branchname>` können jetzt Änderungen aus dem Zweig »readme« in den Masterzweig übernommen werden. Damit wird auch die Datei »Readme« aus dem Repository in den Arbeitsbereich übernommen, wie ein `ls` demonstriert:

```
quade@felicia:~/embedded/application/gpioappl> git checkout master
Switched to branch 'master'
quade@felicia:~/embedded/application/gpioappl> ls
gpioappl.c  Makefile
quade@felicia:~/embedded/application/gpioappl> git merge readme
Updating 644e836..9f5c49e
Fast-forward
 0 files changed
 create mode 100644 Readme
quade@felicia:~/embedded/application/gpioappl> ls
gpioappl.c  Makefile  Readme
```

C.5 Remote-Repository

Per Git lässt sich auf einfachem Wege Quellcode auf den eigenen Rechner kopieren. Dazu dient das bereits erwähnte Kommando `git clone <quelle> <ziel>`. Der erste Parameter gibt in Form einer URL die Quelle, der zweite das lokale Zielverzeichnis an. Wird »ziel« weggelassen, landet die Kopie in einem neu angelegten Verzeichnis unterhalb des gerade aktiven Ordners. Per `git push` lassen sich lokale Änderungen auch an ein entferntes Repository übertragen, zumindest wenn die Zugriffsrechte dafür gegeben sind. Änderungen, die zwischenzeitlich am entfernten Repository durchgeführt wurden, lassen sich per `git pull` auf den lokalen Arbeitsbereich übertragen. Sollen die Änderung nur in das Repository übernommen werden, kann auch `git fetch` eingesetzt werden. Per `git remote add shortname url` lässt sich auch anstelle einer komplizierten URL ein Kurzname vergeben, sodass die Anwendung von Git etwas bequemer wird.

Tabelle C-1 *Wichtige Git-Kommandos*

Kommando	Funktion
Basisfunktionen	
git init	Neues Git-Archiv wird im lokalen Verzeichnis angelegt.
git add <filename>	Die Datei »filename« wird in den Staging-Bereich übernommen.
git commit	Dateien aus dem Staging-Bereich werden in das Repository übernommen.

Kommando	Funktion
git commit -a	Sämtliche geänderte oder bisher nicht erfasste (untracked) Dateien werden direkt in das Repository übernommen (ohne Umweg über den Staging-Bereich).
Branching und Merging	
git branch <branchname>	Legt einen neuen Entwicklungszweig (Branch) mit dem Namen »branchname« an.
git branch -v	Zeigt die vorhandenen Branches an. Ein »*« in der Ausgabe bezeichnet den HEAD-Bereich (HEAD ist der Zeiger auf den aktuellen Branch).
git checkout <branch>	Verschiebt die Dateien von »branch« aus dem Repository in den Arbeitsbereich (working area). Ohne Angabe von »branch« wird der aktuelle Branch (HEAD) verwendet.
git merge <branch>	Der Zweig mit dem Namen »branch« wird in den Masterzweig übernommen.
History	
git log <-n> <-p>	Gibt die Historie der Änderungen aus. Ein optionales »-p« zeigt dabei die Diffs an, das optionale »-n« limitiert die Anzahl der dargestellten Änderungen.
Datentransfer	
git clone shortname/url	Kopiert das über den Kurznamen »shortname« oder die URL »url« angegebene Repository. Optional kann mittels »-b branch« der Branch ausgewählt werden.
git push	Änderungen in der lokalen Kopie des Repositories werden auf das Remote-Repository geschrieben.
git pull	Änderungen im Remote-Repository werden heruntergeladen und automatisch mit dem aktuellen Branch zusammengeführt (mergen).
git fetch shortname/url	Änderungen im Remote-Repository (angegeben entweder über den Kurznamen oder eine URL) werden in das Repository heruntergeladen.
git remote -v	Zeigt die Kurznamen für hinterlegte Remote-Repositories an.
git remote add shortname url	Legt den Kurznamen »shortname« für das über »url« spezifizierte Remote-Repository an.
Sonstiges (stashing)	
git stash	Änderungen werden auf einer Art Clipboard abgelegt. Das Clipboard ist als Lifo (Last in first out) organisiert.
git stash pop	Änderungen werden in den Arbeitsbereich zurückgeschrieben.
git stash list	Abgelegte Änderungen werden aufgelistet.

D Die serielle Schnittstelle

Der Raspberry Pi nimmt auch per serieller Schnittstelle Kontakt mit seiner Außenwelt auf. Die Sende- und Empfangssignale sind dabei auf dem GPIO-Port (Stecker P1) herausgeführt, allerdings handelt es sich dabei nicht um die bei seriellen Schnittstellen gewohnten 5 V Signale, sondern um 3,3 V Signale. Von daher wird ein geeigneter Pegelwandler benötigt.

Klassischerweise werden hierfür — wie beim Arduino — Chips der Firma FTDI eingesetzt, die zudem auch gleich eine USB-Konvertierung vornehmen können. Fertige Platinen finden sich unter dem Namen »FTDI Basic« bei allen einschlägigen Quellen (z.B. [http://www.reichelt.de], [http://www.conrad.de], [http://www.ebay.de] oder [http://www.amazon.de]) und können für 10 bis 20 € gekauft werden. Alternativ gibt es auch Platinen, die auf einem CP2102 beruhen, manchmal noch preiswerter sind und ebenfalls eingesetzt werden können (Abb. D-1).

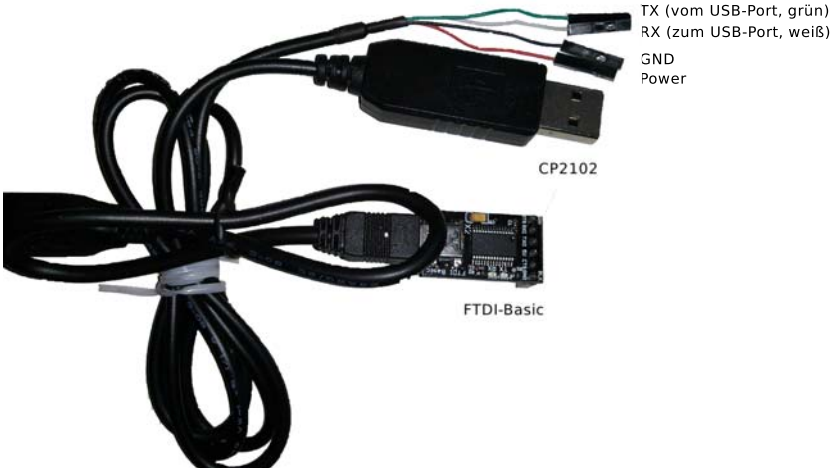


Abb. D-1
Zwei USB-to-TTL-Adapter zur Verbindung von Raspberry Pi und Host

Um die Wandlerplatine mit dem Raspberry Pi zu verbinden, werden außerdem noch drei Anschlussleitungen benötigt: jeweils eine für Senden, Empfangen und für Masse (Ground). Diese werden nach dem in Abbildung D-2 beschriebenen Verfahren angeschlossen.

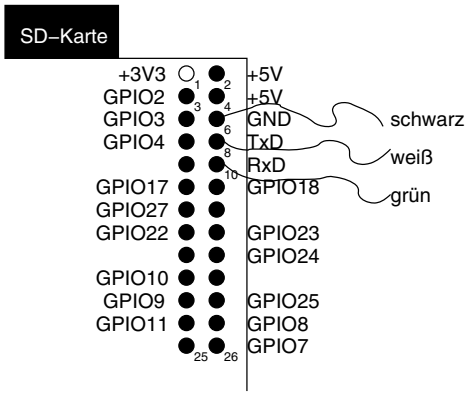


Abb. D-2
Zur seriellen Verbindung werden nur drei Leitungen benötigt.

Aufseiten des Entwicklungsrechners wird das Programm `kermit` oder alternativ `minicom` benötigt. Diese installieren Sie auf der Konsole per

```
quade@felicia:~> sudo apt-get install ckermit minicom
```

Der Raspberry Pi kommuniziert standardmäßig mit 115200 Baud, 8 Daten, keinem Parity-, aber einem Stopbit. Wichtig: Hardware Flow Control ist nicht vorhanden und muss daher ausgeschaltet werden. Um diese Standardkonfigurationen nicht bei jedem Start mit angeben zu müssen, empfiehlt es sich, eine Konfigurationsdatei anzulegen. Die Konfigurationsdatei für Kermit liegt im Heimatverzeichnis und hat den Namen ».kermit«. Bedingt durch den Punkt zu Beginn des Namens handelt es sich um eine versteckte Datei, die mit dem Kommando `ls` ohne weitere Optionen nicht angezeigt wird. Erst `ls -a` zeigt auch versteckte Dateien an. Hier der Inhalt der Datei ».kermit«:

```
set line /dev/ttyUSB0
set flow-control none
set speed 115200
set EIGHTBIT
set parity none
set stopbit 1
```

Wenn Sie den Raspberry Pi richtig mit der Wandlerplatine verbunden haben und den Terminalemulator durch Eingabe von `kermit` starten, können Sie sich direkt durch Eingabe des Kommandos `connect` mit dem Board verbinden. Wird der Raspberry Pi gestartet, sollten sämtliche Boot-Meldungen auf dem Bildschirm ausgegeben werden und anschließend eine Loginmeldung erscheinen. Sollte dieses nicht der Fall sein, überprüfen Sie noch einmal, ob die Sende- und Empfangsleitungen korrekt angeschlossen und nicht vertauscht sind.

Um `kermit` wieder zu verlassen, geben Sie »<Strg><AltGr>\« gefolgt von »c« ein. Dadurch landen Sie wieder auf der Kommandoebene, auf

der Managementkommandos eingegeben werden können. Zum Beenden von kermit geben Sie hier »quit« ein.

Wenn Sie als Terminalemulator minicom bevorzugen, konfigurieren Sie die Terminalemulation (Version 2.5) folgendermaßen:

1. Starten Sie minicom als Superuser und mit der Option »-s«:

   ```
   quade@felicia:~> sudo minicom -s
   ```

2. Unter »Einstellungen zum seriellen Anschluss« konfigurieren Sie:

   - Serieller Anschluss: /dev/ttyUSB0
   - Bps/Par/Bits: 115200 8N1
   - Hardware Flow Control: Nein

3. Verlassen Sie das Untermenü und wählen Sie aus dem Hauptmenü den Punkt »Speichern als dfl«.

4. Wählen Sie »Minicom beenden«.

Hinweis: Die Konfiguration findet sich für den einzelnen Benutzer in der Datei .minirc.dfl oder kann auch global für alle Nutzer in der Datei /etc/minirc.dfl gespeichert werden.

Die Datei .minirc.dfl hat damit den folgenden Inhalt:

```
# Diese Datei ist maschinell erzeugt. Bitte verwenden Sie das
# Einstellungs-Menü im minicom-Programm, um die Einstellungen
# zu ändern.
pu port             /dev/ttyUSB0
pu rtscts           No
```

Sowohl bei kermit als auch bei minicom müssen Sie den Namen der Gerätedatei (/dev/ttyUSB0) an den von Ihrem System verwendeten Namen anpassen.

Nach der Konfiguration geben Sie minicom ein, um sich mit dem Board zu verbinden. Zum Verlassen von minicom, dient die Tastenkombination <Strg><a>, gefolgt von »x«.

Literaturverzeichnis

[bcm2835]
Broadcom Corporation: *BCM2835 ARM Peripherals*. [http://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf]

[bruserman]
The Buildroot user manual. [http://buildroot.net/downloads/manual/manual.pdf]

[cellux2013]
Ruzsa Balázs: *DIY Linux with Buildroot*. [http://cellux.github.io/articles/diy-linux-with-buildroot-part-1/]

[ChWaDe2002]
Hao Chen, David Wagner, Drew Dean: [http://www.usenix.org/event/sec02/full_papers/chen/chen.pdf]

[cnxsoft2013]
CNXSoft: *12MB Minimal Image for Raspberry Pi using the Yocto Project*. [http://www.cnx-software.com/2013/07/05/12mb-minimal-image-for-raspberry-pi-using-the-yocto-project/]

[Eißenlöffel2012]
Thomas Eißenlöffel: *Embedded-Software entwickeln:* Grundlagen der Programmierung eingebetteter Systeme — Eine Einführung in die Anwendungsentwicklung. 1. Auflage, dpunkt.verlag GmbH, Heidelberg, 2012.

[elinux-uboot]
RPi U-Boot. 2. Dezember 2012. [http://elinux.org/RPi_U-Boot]

[foxit2012]
Hans Hoogstraaten, andere: *Report of the investigation into the DigiNotar Certificate Authority breach*. [http://www.rijksoverheid.nl/bestanden/documenten-en-publicaties/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf]

[Grant2012]
Darren Grant: *In Control:* Interfacing Projects for Beginners. Juni 2012, The MagPi Magazine, 2/2012. [http://www.themagpi.com/issue/issue-2/]

[Heise2012]
Protokoll eines Verbrechens: DigiNotar-Einbruch weitgehend aufgeklärt. 2.11.2012, Heise Verlag. [http://www.heise.de/security/meldung/Protokoll-eines-Verbrechens-DigiNotar-Einbruch-weitgehend-aufgeklaert-1741726.html]

[linuxfw]
Wikibooks: *Linux-Kompendium: Linux-Firewall mit IP-Tables*. [https://de.wikibooks.org/wiki/Linux-Kompendium:_Linux-Firewall_mit_IP-Tables]

[prism]
Wikipedia: *PRISM (Überwachungspgrogramm)*. [https://de.wikipedia.org/wiki/PRISM_%28%C3%9Cberwachungsprogramm%29]

[QuKu2009]
Jürgen Quade, Eva-Katharina Kunst: *Kernel- und Treiberprogrammierung mit dem Linux-Kernel:* Kern-Technik 45. Linux-Magazin, 5/2009. [http://www.linux-magazin.de/Ausgaben/2009/05/Kern-Technik]

[QuKu2011a]
Jürgen Quade, Eva-Katharina Kunst: *Kernel- und Treiberprogrammierung mit dem Linux-Kernel:* Kern-Technik 59. Linux-Magazin, 11/2011. [http://www.linux-magazin.de/Ausgaben/2011/11/Kern-Technik]

[QuKu2011b]
Jürgen Quade, Eva-Katharina Kunst: *Linux-Treiber entwickeln:* Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung. 3. Auflage, dpunkt.verlag GmbH, Heidelberg, 2011.

[QuKu2013]
Jürgen Quade, Eva-Katharina Kunst: *Kernel- und Treiberprogrammierung mit dem Linux-Kernel:* Kern-Technik 70. München, 2011, Linux-Magazin, 10/2013. [http://www.linux-magazin.de/Ausgaben/2013/10/Kern-Technik]

[QuMä2012]
Jürgen Quade, Michael Mächtel: *Moderne Realzeitsysteme kompakt:* Eine Einführung mit Embedded Linux. 1. Auflage, dpunkt.verlag GmbH, Heidelberg, 2012.

[Schmidt2014]
Maik Schmidt: *Raspberry Pi:* Einstieg - Optimierung - Projekte. 2. Auflage, dpunkt.verlag GmbH, Heidelberg, 2014.

[stuxnet]
Wikipedia: *Stuxnet — Wikipedia, Die freie Enzyklopädie.* 2013. [https://de.wikipedia.org/w/index.php?title=Stuxnet&oldid=121711061]

[wasserfall]
Wikipedia: *Wasserfallmodell — Wikipedia, Die freie Enzyklopädie.* 2013. [http://de.wikipedia.org/w/index.php?title=Wasserfallmodell&oldid=123333821]

Stichwortverzeichnis

D

E

S