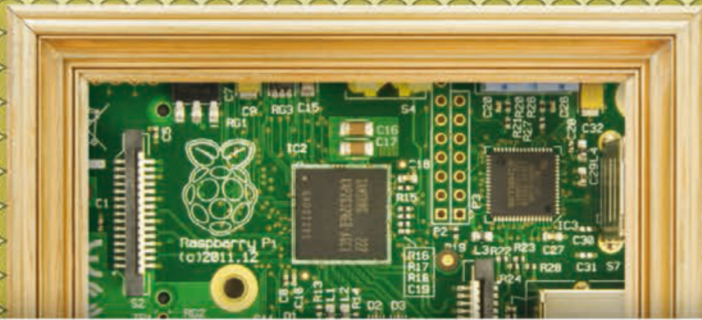


Jürgen Quade

Embedded Linux lernen mit dem Raspberry Pi

Linux-Systeme selber bauen und programmieren

dpunkt.verlag

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.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.de/plus

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
Copyright © 2014 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
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 1

2 Gut zu wissen 9

2.1 Die Architektur eingebetteter Systeme 11
- 2.1.1 Hardware 11
- 2.1.2 Software 14
- 2.1.3 Auf dem Host für das Target entwickeln 19

2.2 Arbeiten mit Linux 21
- 2.2.1 Die Shell 23
- 2.2.2 Die Verzeichnisstruktur 24
- 2.2.3 Editor 25

2.3 Erste Schritte mit dem Raspberry Pi 26
- 2.3.1 System aufspielen 27
- 2.3.2 Startvorgang 29
- 2.3.3 Einloggen und Grundkonfiguration 30
- 2.3.4 Hello World: Entwickeln auf dem Raspberry Pi 30

3 Embedded von Grund auf 33

3.1 Der Linux-Kernel 34

3.2 Das Userland 41
- 3.2.1 Systemebene 43
- 3.2.2 Funktionsbestimmende Applikationen 59

3.3 Cross-Development für den Raspberry Pi 64
- 3.3.1 Cross-Generierung Kernel 64
- 3.3.2 Cross-Generierung Userland 67
- 3.3.3 Installation auf dem Raspberry Pi 71

3.4 Bootloader »Das U-Boot« 76
- 3.4.1 Kernel von der SD-Karte booten 80
- 3.4.2 Netzwerk-Boot 84

3.5 Initramfs: Filesystem im RAM 86

4 Systembuilder Buildroot 95

4.1 Überblick 95
4.2 Buildroot-Praxis 99
4.2.1 Installation auf der SD-Karte 101
4.2.2 Netzwerk-Boot per U-Boot 104
4.3 Systemanpassung 110
4.3.1 Postimage-Skript 111
4.3.2 Postbuild-Skript 113
4.4 Eigene Buildroot-Pakete 131
4.4.1 Grundstruktur 131
4.4.2 Praxis 137
4.5 Hinweise zum Backup 141

5 Anwendungsentwicklung 143

5.1 Cross-Development 144
5.2 Basisfunktionen der eingebetteten Anwendungsprogrammierung 147
5.2.1 Modularisierung 148
5.2.2 Realzeitaspekte 150
5.3 Hardwarezugriffe 155
5.3.1 Systemcalls für den Hardwarezugriff 156
5.3.2 GPIO-Zugriff über das Sys-Filesystem 162

6 Gerätetreiber selbst gemacht 167

6.1 Einführung in die Treiberprogrammierung 168
6.1.1 Grundprinzip 169
6.1.2 Aufbau eines Gerätetreibers 170
6.1.3 Generierung des Gerätetreibers 173
6.2 Schneller GPIO-Treiberzugriff 176
6.2.1 Digitale Ausgabe 177
6.2.2 Digitale Eingabe 185
6.2.3 Programmierhinweise zum Hardwarezugriff 192

7 Embedded Security 197

7.1 Härtung des Systems 199
7.1.1 Firewalling 200
7.1.2 Intrusion Detection and Prevention 212
7.1.3 Rechtevergabe 213
7.1.4 Ressourcenverwaltung 219

7.1.5 Entropie-Management 224
7.1.6 ASLR und Data Execution Prevention 225
7.2 Entwicklungsprozess 226
7.3 Secure-Application-Design 229
7.3.1 Sicherheitsmechanismen in der Applikation 230
7.3.2 Least Privilege 231
7.3.3 Easter Eggs 233
7.3.4 Passwortmanagement 233
7.3.5 Verschlüsselung 235
7.3.6 Randomisiertes Laufzeitverhalten 236

8 Ein komplettes Embedded-Linux-Projekt 237

8.1 Hardware: Anschluss des Displays 238
8.2 Software 240
8.3 Systemintegration 249

Anhänge

A Crashkurs Linux-Shell 259

B Crashkurs vi 269

C Git im Einsatz 273

D Die serielle Schnittstelle 279

Literaturverzeichnis 283

Stichwortverzeichnis 287

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.

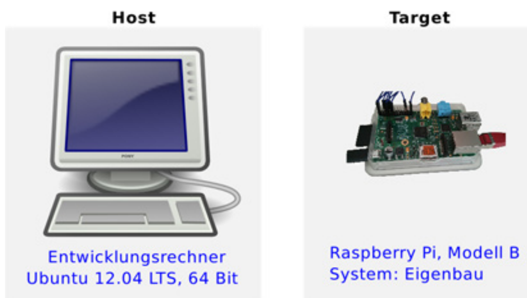


Abb. 1-1 *Host-/Target-Entwicklung*

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

```
~/embedded/                 Hauptordner
  qemu/                     Dateien für das emulierte, eingebettete System
    linux/                  Kernelquellcode
    userland/               Rootfilesystem
      busybox-1.21.1/       Quellcode für die Systemprogramme
      target/               Sonstige Dateien für das Rootfilesystem
  raspi/                    Dateien für den Raspberry Pi
    linux/                  Kernelquellcode
    userland/               Rootfilesystem
      busybox-1.21.1/       Quellcode für die Systemprogramme
      target/               Sonstige Dateien für das Rootfilesystem
    bootloader/             Dateien für einen Raspberry Pi Bootloader
      u-boot-pi/            Quellcode von "Das U-Boot"
      tools/                Programme zur Generierung von U-Boot Files
    firmware/               Dateien für den Original-Bootloader
    buildroot-2013.05/      Systembuilder
    scripts/                Eigene Skripte zur Systemgenerierung
  application/              Funktionbestimmende Applikationen
    hello/                  Quellcode zu Hello World
    gpioappl/               Quellcode zum Zugriff auf GPIOs
  driver/                   Gerätetreiber
    hello/                  Quellcode zum Hello-World-Gerätetreiber
    fastgpio/               GPIO-Gerätetreiber (nur Output)
    fastgpio2/              GPIO-Gerätetreiber (Input und Output)
    hd44780/                Displaytreiber (Controller HD44780)
```

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.

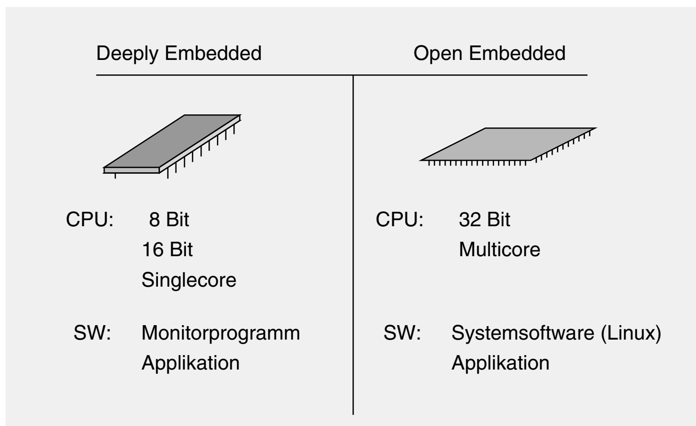


Abb. 2-1 *Klassifizierung eingebetteter Systeme*

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.

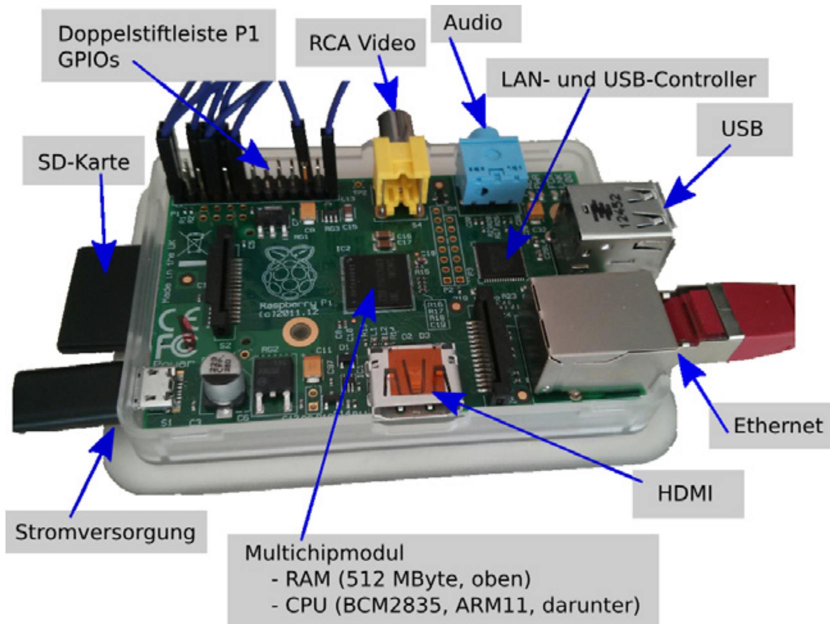


Abb. 2-2
Der Raspberry Pi

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.

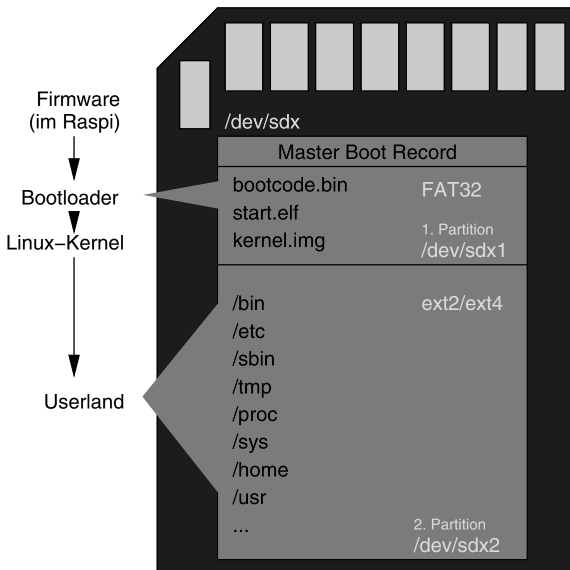


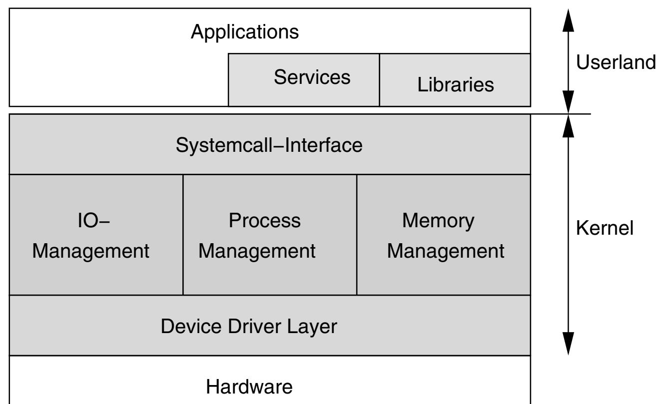
Abb. 2-3
Organisation des Flash-Speichers beim Raspberry Pi

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).

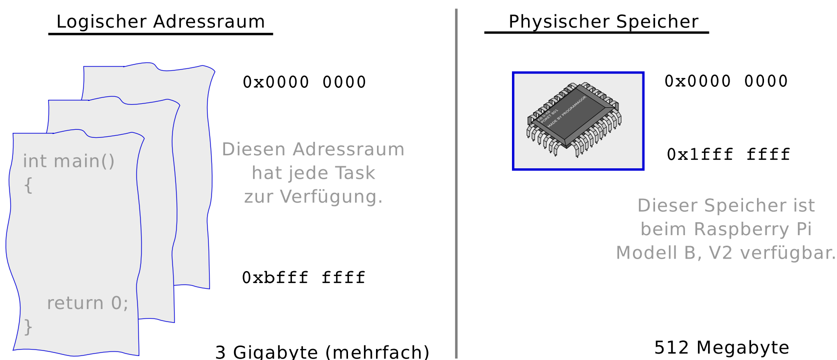


Abb. 2-5
Unterschied logischer und physischer Adressraum

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.

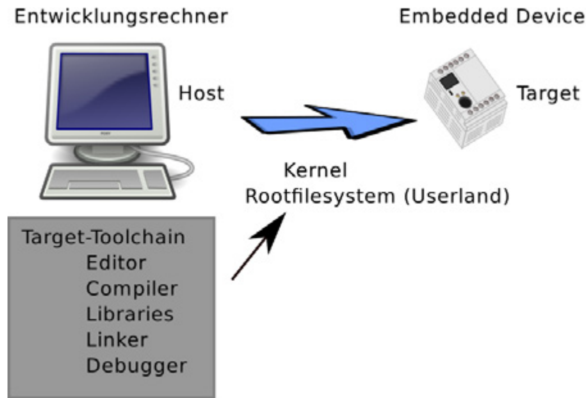


Abb. 2-6
Host-/Target-Entwicklung

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.