Daniel Kriesten

Systementwurf eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem am Beispiel einer modularen Plattform zur Erfassung und Verarbeitung von Sensordaten

Daniel Kriesten

Systementwurf eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem am Beispiel einer modularen Plattform zur Erfassung und Verarbeitung von Sensordaten

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Universitätsverlag Chemnitz

2014

Impressum

Bibliografische Information der Deutschen Nationalbibliothek

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

Diese Arbeit wurde von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität Chemnitz als Dissertation zur Erlangung des akademischen Grades Doktoringenieur (Dr.-Ing.) genehmigt.

Tag der Einreichung: 16.12.2013

Betreuer: Prof. Dr.-Ing. Ulrich Heinkel

1. Gutachter: Prof. Dr.-Ing. Ulrich Heinkel
2. Gutachter: Prof. Dr.-rer. nat. habil. Wolfram Hardt

Tag der Verteidigung: 07.10.2014

Technische Universität Chemnitz/Universitätsbibliothek
Universitätsverlag Chemnitz
09107 Chemnitz
http://www.tu-chemnitz.de/ub/univerlag

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
http://www.mv-verlag.de

ISBN 978-3-944640-34-1

http://nbn-resolving.de/urn:nbn:de:bsz:ch1-qucosa-154966

Bibliographische Angaben

Titel: Systementwurf eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem am Beispiel einer modularen Plattform zur Erfassung und Verarbeitung von Sensordaten

Daniel Kriesten — 2014 — 276 Seiten
66 Abbildungen — 15 Tabellen — 212 Literaturquellen
Technische Universität Chemnitz, Fakultät für Elektrotechnik und Informationstechnik, Dissertationsschrift

Kurzreferat

Ausgehend von einer modularen Plattform zur Erfassung und Verarbeitung von Sensordaten bereichert die vorliegende Dissertationsschrift den Systementwurf eingebetteter Systeme um neue Facetten. Ihr besonderer Fokus liegt dabei auf rekonfigurierbaren Architekturen und Linux-basierten Systemen. Ein wesentlicher Beitrag ist die Darstellung und Diskussion von Konzepten und Architekturen vorgenannter Systeme durch ihre Betrachtung auf einer hohen Abstraktionsebene. Dazu schafft die Arbeit ein umfassendes Verständnis für Kommunikation und Konfiguration in heterogenen rekonfigurierbaren Systemen und überträgt die Erkenntnisse auf das Linux-Betriebssystem. Es erfolgt außerdem eine systematische Darstellung der etablierten Zusammenhänge und Abläufe beim Software-, Paket- und Versionsmanagement im Linux-Umfeld. Zur Verbesserung des Entwurfsflusses werden Konzepte und ein geeignetes Werkzeug zur High-Level Spezifikation von Linux-Systemen dargestellt. Die in der Arbeit gewonnenen wissenschaftlichen Erkenntnisse werden hinsichtlich praktischer Relevanz evaluiert und durch prototypische Implementierungen verifiziert.

Schlagwörter

Systementwurf, Spezifikation, Betriebssystem, Linux, Eingebettete Systeme, Heterogene rekonfigurierbare Systeme, Dynamisch partielle Rekonfiguration

Inhaltsverzeichnis

Glossar **13**

Akronyme **15**

Vorwort **21**

1 Einleitung **23**

1.1 Bedeutung und Entwicklung eingebetteter Systeme 23
1.2 Möglichkeiten rekonfigurierbarer Hardwaresysteme 25
1.3 Das Betriebssystem Linux 27
1.4 Zielsetzung 29
1.5 Verwandte Arbeiten 31
1.6 Struktur dieser Arbeit 32

2 Grundlagen **35**

2.1 Begriffsdefinitionen 35
2.2 Systementwurf und Entwurfsablauf 37
2.3 Eingebettete Systeme 41
2.3.1 Grundsätzliche Systemarchitektur eingebetteter Systeme 43
2.3.2 Hardwareentwurf, -test und -verifikation 47
2.4 Field Programmable Gate Arrays 49
2.4.1 Schnittstellen und Konfigurationsmodi 50
2.4.2 Datenformate 54
2.4.3 Weitere Funktionen und Merkmale 55
2.5 Run-time Reconfiguration 56
2.5.1 Formen der Run-time Reconfiguration 56
2.5.2 Run-time Reconfiguration mit FPGAs 58
2.6 Betriebssysteme 63
2.6.1 Betriebssysteme im Kontext eingebetteter Systeme . . 64

2.6.2 Firmware 65
2.6.3 Anforderungen an den Softwareentwurf 66
2.7 Zusammenfassung und Diskussion 68

3 Systemarchitekturen heterogener rekonfigurierbarer Systeme 71
3.1 Begriffserklärung und Systemarchitekturen 71
3.2 Vertiefung der Systemarchitekturen 75
3.2.1 Kommunikationsarchitekturen 76
3.2.2 Datenaustausch 77
3.2.3 Konfigurationsarchitekturen 80
3.3 Architekturen für eingebettete Systeme 83
3.4 Entwurf eingebetteter heterogener rekonfigurierbarer Systeme 86
3.5 Betriebssystem 87
3.5.1 Konfigurationsstrategien 87
3.5.2 Konfigurationsmanagement 89
3.6 Zusammenfassung 92

4 Das Betriebssystem Linux im Kontext des Systementwurfs 95
4.1 Linux Grundlagen 95
4.1.1 Begriffe 96
4.1.2 Linux-Systeme 96
4.2 Softwaremanagement im Linux-Umfeld 103
4.2.1 Paketmanagement 104
4.2.2 Versionsmanagement 107
4.2.3 Automatisiertes Versionsmanagement 108
4.3 Besonderheiten bei eingebetteten Systemen 109
4.4 Erstellen eines Linux für eingebettete Systeme 112
4.4.1 Werkzeuge 112
4.4.2 Buildsysteme 113
4.4.3 Patchmanagement 117
4.4.4 Versionsmanagement 119
4.5 Spezifika bei heterogenen rekonfigurierbaren Systemen 125
4.5.1 Kommunikation mit dem RM 125
4.5.2 Umsetzung der Konfiguration 127
4.6 Nutzung plattformunabhängiger Sprachen in eingebetteten Systemen 128

4.7 Zusammenfassung und Diskussion 129
4.7.1 Zusammenfassung 130
4.7.2 Diskussion der Erkenntnisse 131

5 Spezifikation, Test und Verifikation von Linux-Systemen 135
5.1 Spezifikation von Linux-Systemen 135
5.2 High-Level Spezifikation einer Linux-Firmware 137
5.3 Test und Verifikation 141
5.4 Zusammenfassung und Diskussion 144

6 Umsetzung der Systemkonzepte 145
6.1 Vorstellung der genutzten Entwicklungsumgebung 145
6.1.1 Entwicklungsumgebung 146
6.1.2 Versionsverwaltung 148
6.1.3 Buildsysteme 150
6.2 Xilinx System „embedded FPGA“ 152
6.2.1 Hardware 152
6.2.2 Software und Entwurfsumgebung 154
6.2.3 Systemkonzept und -entwurf 156
6.2.4 Beispielanwendung 157
6.3 Prototyping-System „AVR32“ 158
6.3.1 Hardware 158
6.3.2 Software und Entwurfsumgebung 159
6.3.3 Systemkonzept und -entwurf 160
6.3.4 Beispielanwendung 161
6.4 PowerPC-System „CarBox“ 167
6.4.1 Hardware 167
6.4.2 Software und Entwurfsumgebung 170
6.4.3 Systemkonzept und -entwurf 172
6.4.4 Beispielanwendungen 177
6.5 Raspberry Pi 181
6.5.1 Hardware 182
6.5.2 Systemkonzept und -entwurf 184
6.5.3 Beispielanwendung 185
6.6 Zusammenfassung 194

7 Konzepte zur plattformunabhängigen Erfassung von Sensordaten 199
7.1 Sensordatenerfassung auf einer zentralen Plattform 199
7.1.1 Datenmodelle zur Sensordatenerfassung 201
7.1.2 Datenübertragung an einen Datenbankserver 207
7.1.3 Implementierung des Systems 208
7.2 Abstraktion von Kommunikationskanälen 210
7.3 Zusammenfassung 212

8 Zusammenfassung und Ausblick 215
8.1 Zusammenfassung 216
8.2 Ausblick 218

Anhang 220

A Programmierbare Logikschaltkreise 223
A.1 Klassifikation Programmierbarer Logikschaltkreise 223

B Der Entwurfsprozess bei FPGA 227

C Versionsmanagement mit Git 231

D Buildsysteme 233

E Linux High-Level Spezifikation 235
E.1 Alternative Konzepte des Synthesesystems 235
E.2 Alternative Implementierungsvariante 236
E.3 Das kconfig-System 238

F Das Projekt „GPSV“ 241

G Details zur Umsetzung der Software auf dem AVR32 243
G.1 Verarbeitung von XSVF-Dateien 243
G.2 Exemplarische Darstellung eines Kernel-Treibers 244

Literatur 247

Abbildungsverzeichnis 267

Tabellenverzeichnis 271

Thesen 273

Glossar

Application
: dt. Anwendung; Schlagwort für eine kleine Anwendung, welches vorrangig im Bereich mobiler, eingebetteter Systeme Verwendung findet.

Application Programming Interface
: dt. Programmierschnittstelle; Das *Application Programming Interface (API)* ist eine definierte Schnittstelle einer Software, über die anderen Programmen der Zugriff auf die zur Verfügung gestellte Funktionalität ermöglicht wird.

ARM
: ARM (Acorn Risc Machine) bezeichnet eine 1983 vom britischen Unternehmen Acorn entwickelte 32-Bit Prozessorarchitektur.

Graphical User Interface
: dt. grafische Benutzerschnittstelle; Als *Graphical User Interface (GUI)* bezeichnet man eine Bedienoberfläche, die dem Benutzer die Interaktion mit einem System über grafische Symbole ermöglicht.

Institute of Electrical and Electronics Engineers
: Das *Institute of Electrical and Electronics Engineers (IEEE)* ist ein Weltweiter Berufsverband von Ingenieuren der Elektro- und Informationstechnik, der Gremien für die Standardisierung bildet, diverse Fachtagungen veranstaltet und Fachzeitschriften herausgibt.

Integrated Circuit
: dt. integrierter Schaltkreis; Als *Integrated Circuit (IC)* bezeichnet man eine auf einem einzelnen Silizium-Die integrierte Schaltung.

Intellectual Property Core	dt. Block geistigen Eigentums; *Intellectual Property Core (IP-Core)* bezeichnet einen Teil einer digitalen Schaltung, der als solcher Eigentum des Erstellers ist, aber von anderen genutzt werden kann.
Interprozesskommunikation	*Interprozesskommunikation (IPC)* – engl. interprocess communication; Im hier verwendeten Sinne die Kommunikation zwischen Prozessen auf einem Computersystem.
Joint Test Action Group	Der Begriff *Joint Test Action Group (JTAG)* bezeichnet die Gruppe von Verfahren zum Testen und Debuggen von elektronischen Schaltungen nach dem IEEE-Standard 1149.1.
PR-Modul	Die dynamischen Teile eines Field Programmable Gate Array-Designs.
PR-Region	Die Bereiche eines Field Programmable Gate Array, in dem Module bei der dynamisch-partiellen Rekonfiguration platziert werden.
PS/2-Schnittstelle	Die PS/2-Schnittstelle ist eine serielle Schnittstelle, die im Personal Computer-Bereich hauptsächlich für Eingabegeräte wie Tastaturen und Mäuse verwendung findet.
Software Programmable Reconfiguration	*Software Programmable Reconfiguration (SPR)* bezeichnet die vom Design vorgesehene Möglichkeit, Abläufe in digitaler Logik durch interne oder externe Softwarekommandos zu modifizieren.
SQL	SQL ist eine Datenbanksprach, mit des sich Daten und Strukturen in einer SQL-fähigen Datenbank manipulieren lassen.
Video Graphics Array	VGA ist ein Standard für Darstellungsmodi (Auflösung, Wiederholfequenz und Farbtiefe) bei Computergrafiksystemen.
Virtex-II Pro Development System	Test- und Evaluationsplattform aus dem Xilinx Universitätsprogramm mit Virtex-II Pro FPGA.

Akronyme

A/D-Wandler . . . . . Analog-Digital-Wandler

ABI . . . . Application Binary Interface

ALU . . . . arithmetisch-logische Einheit

AMBA . . . . Advanced Microcontroller Bus Architecture

AMBA™AXI . . . . . Advanced Microcontroller Bus Architecture™Advanced eXtensible Interface Bus

API . . . . Application Programming Interface

apt . . . . Advanced Package Tool

ASIC . . . . Application Specific Integrated Circuit

ATNGW100 . . . . . AVR32 Network Gateway Kit

AXI . . . . Advanced eXtensible Interface Bus

BPI . . . . Byte Peripheral Interface

BSP . . . . Board Support Package

µC . . . . Mikrocontroller

CAN . . . . Controller Area Network

CASE . . . . Computer-Aided Software Engineering

CB . . . . PR Control Block

CLB . . . . Configurable Logic Block

CPLD . . . . Complex Programmable Logic Device

CPU . . . . Central Processing Unit

D/A-Wandler . . . . . Digital-Analog-Wandler

DBMS . . . . Datenbankmanagementsystem

DHCP . . . . Dynamic Host Configuration Protocol

DKI . . . . Digitales Kombiinstrument

DMA . . . . Direct Memory Access

dpR . . . . dynamisch-partielle Rekonfiguration

dR . . . . dynamische Rekonfiguration

DUT . . . . Device Under Test

DVVS . . . . dezentrales Versionsverwaltungssystem

EABI . . . . Embedded Application Binary Interface

EDA . . . . Electronic Design Automation

EDK . . . . Embedded Development Kit

EEPROM . . . . Electrical Eraseable Programmeable Read Only Memory

ELDK . . . . Embedded Linux Development Kit

EPC . . . . Enhanced Configuration Devices

ERM . . . . Entity-Relationship-Modell

FDT . . . . Flattened Device Tree

FHS . . . . File System Hierarchy Standard

FOS . . . . Free and Open Source

FPGA . . . . Field Programmable Gate Array

FPU . . . . Floating Point Unit

FSF . . . . Free Software Foundation

FSM . . . . Finite State Machine

GCC . . . . GNU Compiler Collection

GNU . . . . GNU's not Unix

GNU GPL . . . . . . GNU General Public License

GPIO . . . . General Purpose Input Output

GPP . . . . General Purpose Processor

GPSV . . . . Generalisierte Plattform zur Sensordatenverarbeitung

GPU . . . . Graphics Processing Unit

GSM . . . . Global System for Mobile Communications

GUI . . . . Graphical User Interface

HPC . . . . High Performance Computing

hrS . . . . heterogene rekonfigurierbare Systeme

HTML . . . . Hypertext Markup Language

HTTP . . . . Hypertext Transfer Protocol

HW . . . . Hardware

I/O . . . . Input and Output

I^2C . . . . Inter-Integrated Circuit

ICAP . . . . Internal Configuration Access Port

IC . . . . Integrated Circuit

IDE . . . . Integrated Design Environment

IEEE . . . . Institute of Electrical and Electronics Engineers

IP . . . . Intellectual Property

IPC . . . . Interprozesskommunikation

IP-Core . . . . Intellectual Property Core

IRC . . . . Internet Relay Chat

ISE . . . . Integrated Software Environment Design Suite

ISP . . . . In System Programmable

JTAG . . . . Joint Test Action Group

KBSt . . . . Koordinierungs- und Beratungsstelle für Informationstechnik in der Bundesverwaltung

KDB . . . . Kernel Debugger

Kfz . . . . Kraftfahrzeug

KMU . . . . kleine und mittelständische Unternehmen

LED . . . . Light Emitting Diode

LSB . . . . Linux Standard Base

LTE . . . . Long Term Evolution

LVDS . . . . Low Voltage Differential Signaling

MIPS . . . . Microprocessor Without Interlocked Pipeline Stages

MMU . . . . Memory Management Unit

NFS . . . . Network File System

OOA . . . . objektorientierte Analyse

OOD . . . . objektorientiertes Designs

OPB . . . . On-chip Peripheral Bus

PC . . . . Personal Computer

PCI . . . . Peripheral Component Interconnect
PCIe . . . . Peripheral Component Interconnect Express
PDA . . . . Personal Digital Assistant
PFP . . . . Platform Flash *PROM*
PLB . . . . Processor Local Bus
PLD . . . . Programmable Logic Device
pR . . . . partielle Rekonfiguration
PROM . . . . Programmeable Read Only Memory
PSoC . . . . Programmable System-on-Chip

RAM . . . . Random Access Memory
RISC . . . . Reduced Instruction Set Computer
RM . . . . rekonfigurierbares Modul
ROM . . . . Read Only Memory
rSoC . . . . rekonfigurierbares System-on-Chip
RT-Ebene . . . . Register-Transfer-Ebene
RTOS . . . . Real-Time Operating System
RTR . . . . run-time Reconfiguration

SMF . . . . System Management Facility
SoC . . . . System-on-Chip
SoM . . . . System-on-Module
SoP . . . . System on Package
SPI . . . . Serial Peripheral Interface
SPLD . . . . Simple Programmable Logic Device
SPR . . . . Software Programmable Reconfiguration

SRAM . . . . Static Random Access Memory
SSC . . . . Synchronous Serial Controller
SSH . . . . Secure Shell
STAPL . . . . Standard Test and Programming Language
SVF . . . . Serial Vector Format
SW . . . . Software

TAP . . . . Test Access Port
TCK . . . . Test Clock
TDI . . . . Test Data In
TDO . . . . Test Data Out
TFTP . . . . Trivial File Transfer Protocol
TMS . . . . Test Mode Select

UART . . . . Universal Asynchronous Receiver Transmitter
UIO . . . . User Space Input and Output
UMTS . . . . Universal Mobile Telecommunications System
USB . . . . Universal Serial Bus

Vivado . . . . Xilinx Vivado™ Design Suite
VVS . . . . Versionsverwaltungssystem

WLAN . . . . Wireless Local Area Network

XAPP . . . . Xilinx Application Note
XML . . . . Extensible Markup Language
XSVF . . . . Xilinx Serial Vector Format
XUP-V2P . . . . Virtex-II Pro Development System

Vorwort

Die Idee für die vorliegende Arbeit entstand während meiner Tätigkeit an der *Professur Schaltkreis- und Systementwurf (SSE)* der Technischen Universität Chemnitz. Wesentliche Impulse lieferten dabei die Themen, die ich im Rahmen der Projekte *Generalisierte Plattform zur Sensordatenverarbeitung (GPSV)* und *Generische Plattform für Systemzuverlässigkeit und Verifikation (GPZV)* bearbeitete, die vom Bundesministerium für Bildung und Forschung innerhalb der Initiative „InnoProfile — Unternehmen Region“ unter den Förderkennzeichen 03IP505 und 03IPT505X gefördert wurden.

Mein besonderer Dank gilt Prof. Ulrich Heinkel, der mir die Arbeit an meiner Dissertation ermöglichte, mir beratend zur Seite stand und mich immer wieder motivierte, meine Untersuchungen voranzutreiben. Prof. Wolfram Hardt danke ich für die Übernahme des Zweitgutachtens.

Eine wichtige Rolle bei der Entstehung der Arbeit spielten meine Kollegen am SSE, die mich kollegial und auch freundschaftlich bei der Bewältigung meiner täglichen Aufgaben unterstützten und mir so den einen oder anderen zusätzlichen Freiraum schafften. Im Besonderen danke ich Wolfgang Kilian, der sich in unzähligen Diskussionen als kreativer, kritischer und geduldiger Partner erwiesen hat und bei dem ich jederzeit ein offenes Ohr fand. Sebastian Kratzert danke ich für die vielen Hinweise und Unterweisungen bei der Überführung einer Idee in realen, wartbaren Quellcode und für die anregenden Gespräche zum Softwareentwurf. Steffen Weichold und Thomas Graichen danke ich für das akribische und kritische Korrekturlesen der Arbeit und die daraus entstandenen Diskussionen. Volker Pankalla und Paul Friedrich für ihr Engagement im Rahmen ihrer Diplomarbeiten, die bei der Validierung einiger grundsätzlicher Ideen und Ansätze meiner Arbeit zum Tragen kamen. Ebenso danke ich Stefan Weisleder, Florian Schlegel und Robert Kiesel, die mich während ihrer studentischen Tätigkeiten bei der Validierung der Ideen im Bereich Linux und Datenbanken unterstützten.

Mithin der größte Dank gilt jedoch meiner Familie und den vielen Freunden, die uns in der arbeitsreichen Zeit unterstützten. Meinen Eltern ebenso wie meinen Schwiegereltern danke ich für aufmunternde und motivierende Gespräche. Meiner wundervollen Tochter Henriette und meiner liebevollen Frau Jenny für ihre unendliche Geduld, ihre Nachsicht und ihre fortwährende Unterstützung. Ihr seid der wahre Grund für den erfolgreichen Abschluss dieser Arbeit.

1 Einleitung

Eingebettete Systeme entstehen in vielen Fällen mit Bezug zu einer bestimmten Anwendung. Bedingt durch ihre Flexibilität und Leistungsfähigkeit, aber auch durch den mit ihrem Entwurf verbundenen Aufwand, werden sie anschließend mit geringen Modifikationen oft in weiteren Situationen verwendet. Daher stellt sich fortlaufend die Frage, ob eine Konvergenz verschiedener Plattformen hin zu einem „generalisierten“ System möglich ist und was dies aus Sicht des Systementwurfs für Auswirkungen hat. Die Auseinandersetzung mit dieser Fragestellung bildet den Grundgedanken für die vorliegende Dissertation.

1.1 Bedeutung und Entwicklung eingebetteter Systeme

Der Begriff „Embedded System“ (dt. eingebettetes System) mit seiner unscharfen Definition (siehe Abschnitt 2.3) findet weitläufig Anwendung. Anfangs verstand man unter diesem Begriff Systeme, die in einen meist technischen Kontext eingebunden sind und vom Nutzer nicht direkt wahrgenommen werden. Diese Plattformen waren meist hoch spezialisierte Geräte einer bestimmten Domäne und geprägt durch spezielle Hardware sowie angepasste Software. Die technischen Möglichkeiten und die wachsende Durchdringung vorhandener sowie die Erschließung neuer Einsatzbereiche verändern die Wahrnehmung der Nutzer für diese Systeme und damit auch das Verständnis für den Begriff des eingebetteten Systems. Neben den domänenspezifischen und kontextbezogenen Systemen, seien es komplexe Steuerungssysteme in der Industrie, sicherheitskritische Komponenten im Fahrzeug oder einfache in Haushaltsgeräten, zählen heute auch „Embedded PCs“ mit grafischer Nutzerschnittstelle, PDAs, Smartphones und Tablet-*PC*s zur Klasse der eingebetteten Systeme –

und werden vom Nutzer nicht mehr direkt als „Rechner“ wahrgenommen. Die zunehmende Verbreitung informationsverarbeitender Systeme bei gleichzeitiger Verringerung der Sensitivität für diese Systeme wird unter den Schlagworten *ubiquitous computing* (engl. allgegenwärtiges Rechnen), *pervasive computing* [Bur+01; Han+03] oder auch *ambient intelligence* zusammengefasst und wissenschaftlich untersucht.

Je weiter man den Kreis der eingebetteten Systeme fasst, umso stärker lässt sich erkennen, wie dieser Markt von seiner eigenen Diversität profitiert. Der schnelllebige (End-)Verbrauchermarkt (*consumer market*) verzeiht Unzulänglichkeiten[1], verlangt aber in zunehmendem Maße nach Energieeffizienz, hoher Rechenleistung und steht unter enormem Preisdruck. Hier fließen viele Neuerungen aus dem klassischen *PC*- und *HPC*-Bereich ein, erfahren eine Anpassung an die Spezifika eingebetteter Systeme und werden diesbezüglich weiterentwickelt. Im weiteren Verlauf fließen Technologien, die sich auf diese Weise bewähren konnten, auch dort ein, wo Robustheit und Sicherheit eingebetteter Systeme stärker im Vordergrund stehen.

Der Markt der eingebetteten Systeme weist momentan in vielen Bereichen eine sehr hohe Dynamik auf. So erscheinen Produkte aus dem Endverbrauchermarkt, wie Mobiltelefone oder Fernseher, mittlerweile im Abstand weniger Monate. Gleichzeitig ist der Preisdruck in diesen Domänen extrem hoch. Hersteller begegnen diesen Herausforderungen unter anderem durch Modularisierung und Mehrfachnutzung („*Re-Use*“), was die Verwendung einmal entworfener und verifizierter Komponenten in mehreren Produktgenerationen erlaubt. Die Granularität der Module, unabhängig davon, ob es sich um *Hardware (HW)* oder *Software (SW)* handelt, ist dabei sehr unterschiedlich und reicht von Teilen einer Schaltung oder eines Programms über Module eines komplexeren Systems bis hin zu kompletten Systemen. Im letztgenannten Fall unterscheiden sich die Produkte zweier Hersteller nur durch die Marke und ein ggf. damit verbundenes spezifisches Aussehen, den Preis, den Service und die Software.

Der Software kommt in Hinblick auf kurze Produktzyklen und die damit verbundenen verkürzten Verifikationszyklen eine weitere wesentliche Bedeutung zu. Einmal an den Kunden gelieferte Hardware kann im Fehlerfall – wenn

[1] Diese können von Schwächen in der Bedienbarkeit bis hin zu gelegentlichen Fehlern reichen.

überhaupt – nur durch erhöhten Aufwand nachgebessert werden. Die Veränderung der Software hingegen, sei es zur Beseitigung von Fehlern oder zur Erweiterung der Funktionalität, ist verhältnismäßig einfach. Dieser als *Update* oder *Firmware-Update*[2] bezeichnete Vorgang ist nicht neu. Durch die starke Verbreitung von netzwerkfähigen Geräten und Breitbandanschlüssen hat er aber eine neue Qualität erreicht. Besonders im Endverbrauchermarkt dienen die Verfügbarkeit und die Frequenz von Soft- und Firmware-Updates sogar als positives Produktmerkmal.

Gerade der Endverbrauchermarkt für Elektronikprodukte adressiert ein riesiges Kundenspektrum. Die an kurze Produktzyklen und regelmäßige, einfach zu handhabende „Updates" gewöhnten Konsumenten übertragen diese Erwartungen auch auf andere Domänen, in denen bisher andere Gesetzmäßigkeiten galten. So fällt es beispielsweise Automobilherstellern, deren Produktzyklen mehrere Jahre betragen, schwer, den geänderten Kundenbedürfnissen Rechnung zu tragen, besonders unter Berücksichtigung der notwendigen Qualitätsaspekte. Doch auch die Hersteller von Werkzeugmaschinen und anderen Großgeräten mit traditionell langen Wartungs- und Verfügbarkeitszyklen stehen dieser aus der veränderten Kundensicht erwachsenden Herausforderung gegenüber.

1.2 Möglichkeiten rekonfigurierbarer Hardwaresysteme

Schier unzählige Bücher und Schriften motivieren die Idee rekonfigurierbarer Hardware seit ihrer Entstehung in den 1960er Jahren mit den verschiedensten Argumenten [BAK96; LSC96; RM98; VM05; BCV06; Bob07; PTW10]. Fest steht, dass die Möglichkeit, die Hardware eines Systems ebenso leicht und elegant verändern zu können wie seine Software, ein faszinierender Gedanke war und ist. Diese Möglichkeit bietet Herstellern einen weiteren Freiheitsgrad bei der Produktentwicklung und -vermarktung, da nun auch technisch gleiche Systeme Unterschiede in Hard- und Software bieten.

[2] Nachbesserung oder Erweiterung der Anwendungssoftware bzw. Firmware, vgl. Abschnitt 2.6.2

Dennoch konnten sich rekonfigurierbare Systeme bisher kaum im Markt etablieren. Dabei finden sich Einsatzmöglichkeiten, wie nachfolgend skizziert, auch in sehr prominenten Beispielen.

Ein greifbares und leicht verständliches – wenn auch hypothetisches – Anwendungsbeispiel für den Einsatz rekonfigurierbarer Hardware sind aktuelle Smartphones und ihre Betriebssysteme [Kri12]. Seit dem Durchbruch dieser Geräteklasse im Jahr 2007[3] wird die Leistungsfähigkeit der Systeme durch die Verwendung von immer mehr und immer komplexerer Hardware gesteigert. Inzwischen werden in High-End Geräten zwei bis vier Prozessorkerne, drei oder vier Grafikprozessoren und mehrere Co-Prozessoren für die Beschleunigung spezieller Aufgaben eingesetzt. Das führt selbstverständlich zu steigendem Platzbedarf, erhöhtem Energiebedarf, steigender Abwärme und nicht zuletzt höheren Kosten. Doch aufgrund der Nutzungsphilosophie, der kleinen Displays und der im Vergleich zu Desktop-*PCs* noch immer leistungsschwachen Hardware wird üblicherweise nur eine Anwendung aktiv genutzt, z. B. der Musikspieler, der Videospieler oder die volle Grafikleistung beim Spielen. Ein Hinweis darauf, dass den Herstellern dieser Umstand bekannt ist, ist die zögerliche Umsetzung des Multitasking – also des Nutzens mehrere Anwendungen parallel – auf Anwendungsebene[4]. Auch die verfügbare Software nutzt in vielen Fällen die parallel verfügbare Hardware nicht oder nur schlecht aus. Selbst Apples aktuelles Desktop-Betriebssystem bietet einen Bedienmodus, bei dem eine Anwendung die komplette Bildschirmfläche füllt, um den Nutzer nicht abzulenken. Wenn also bestimmte Hardwareeinheiten nur gelegentlich und nicht zwingend gleichzeitig benötigt werden, drängt sich die Nutzung rekonfigurierbarer Hardware geradezu auf. Ein geeigneter, rekonfigurierbarer Co-Prozessor im System kann dann die Aufgaben mehrerer dedizierter Beschleuniger übernehmen. Dies spart Chipfläche, Energie und damit auch Kosten.

Augenscheinlich gibt es also auch in Märkten mit hohem Absatzvolumen Anwendungsfälle für rekonfigurierbare Hardwaresysteme und auch die für solche Systeme notwendige Infrastruktur existiert. Es gibt Hersteller geeigneter rekonfigurierbarer Schaltkreise (engl. ICs) und es gibt kommerzielle An-

[3] Die Vorstellung des ersten „iPhones“ durch Apple gilt als Durchbruch für Smartphones und Tablet-*PCs*.

[4] iOS bietet Multitasking erst seit Version 4, Windows Mobile kam ohne Multitasking auf den Markt.

bieter für die benötigten Hardwaremodule. Doch außer speziellen Lösungen im industriellen Bereich und vereinzelten Nutzungsbeispielen im Endkundenmarkt[5] [Alt06; Hip11] gibt es bisher keine nennenswerten Umsetzungen. Gründe dafür liegen nach den Erfahrungen des Autors u. a. in mangelndem Bewusstsein für die vorhandenen Lösungen speziell beim Management, in fehlender Erfahrung bei den Anwendungsingenieuren, in der bisher fehlenden Standardisierung von Funktionen und Schnittstellen und im veränderten Entwurfsfluss.

1.3 Das Betriebssystem Linux

Seinen Ursprung hat das Betriebssystem Linux in den Arbeiten des finnischen Studenten Linus Torvalds, die er im Jahr 1991 erstmals veröffentlichten [TD01]. Er publizierte die Quellen des Kernels unter der *GNU General Public License (GNU GPL)* als *Open Source* und ermöglichte es so anderen, an seiner Arbeit zu partizipieren. Gleichzeitig erreichte er durch die Regeln der GNU GPL den Rückfluss der Entwicklungsleistungen in sein Projekt. Eine wachsende Entwicklergemeinde unterstützte Torvalds bei der weiteren Entwicklung des Linux-Kernels. Doch erst die Kooperation mit anderen freien Projekten, wie die *GNU's not Unix (GNU)* Tools der *Free Software Foundation (FSF)*, ermöglichten das komplexe Unix-System, zu dem Linux inzwischen geworden ist.

Heute ist der Linux-Kernel auf mehr als zwei Dutzend Prozessorarchitekturen portiert. Linux-Systeme finden sich auf Servern mit höchster Verfügbarkeit, Supercomputern, Clustern, Desktop-*PCs*, mobilen Geräten und verschiedensten eingebetteten Systemen. Zu dieser Entwicklung haben verschiedene Faktoren beigetragen. Der freie Zugang zu den Quellen der Kernels und zu vielen der mit einem typischen Linux-System verbundenen Programme, sowie die Möglichkeit, diese nach eigenem Bedarf zu modifizieren[6], zählen ebenso dazu, wie die Skalierbarkeit und Portierbarkeit des Linux-Kernels. Außerdem bietet

[5] Die bekannte AVM FritzBox! (z. B. Fon 7170, auch als Speedport W 701V von der Telekom vermarktet) nutzt je nach Ausführung ein Xilinx Spartan-3/3E *FPGA*.

[6] Die Verfügbarkeit der Quelltexte und die Erlaubnis, diese zu Benutzen, zu Ändern und die Änderungen auch frei verfügbar zu machen, sind die Grundlagen von *Free and Open Source*-Software.[Sta86; Fre01]

Linux einem Hersteller die Möglichkeit, seine Software unabhängig von kommerziellen Anbietern und deren Produkt- und Markenpolitik zu halten.

Diese „weichen Faktoren“ sind ein Teil der Erfolgsgeschichte des Unix-Derivates und werden durch technisch begründete Fakten ergänzt. Die momentane Entwicklung lässt den Schluss zu, dass die positiven Punkte des Betriebssystems und der mit ihm eng verbundenen Software die naturgemäß existierenden Nachteile und Probleme auf technischer, organisatorischer und rechtlicher Seite ausgleichen oder gar überflügeln.

Inzwischen hat der Trend zu frei zugänglichen und verwendbaren Informationen auch andere Teilbereiche erfasst und beginnt diese zu revolutionieren. Ein weit bekanntes Beispiel ist die Online-Enzyklopädie „Wikipedia“ [Wik], auf der tausende von Menschen ihr Wissen zusammentragen und unter freien Lizenzen wie der *Creative Commons Attribution-ShareAlike 3.0 Unported License* [Cre] zur Verfügung stellen. Auch im Bereich des Hardwareentwurfs hat das Modell frei verfügbarer Entwürfe inzwischen Fuß gefasst. Der Wunsch nach zugänglicher und anpassbarer Hardware (engl. *Open Hardware*) erreicht auch Hersteller eingebetteter Systeme, deren Kunden häufiger den Wunsch nach anpassbarer Hard- und Software äußern[7].

Wie bereits angedeutet, existieren auch in der Open Source Welt verschiedene Probleme, von denen vier hier explizit aufgegriffen werden.

Komplexität Der frei verfügbare Quellcode ermöglicht es jedem Interessierten, ihn zu verändern und zu erweitern. Dies führt in manchen Fällen zu sehr komplexer Software mit einer großen Anzahl an Funktionsmerkmalen. Die Nutzung einer solchen Software erfordert ein hohes Maß an Einarbeitung und zugleich steigt die Wahrscheinlichkeit für Fehler in der Software.

Vielfalt Ebenso schnell ist durch unterschiedliche Interessen aus einem Projekt ein neues ausgegliedert, was üblicherweise als *Fork* bezeichnet wird. Dadurch existieren gelegentlich mehrere Projekte, die nahezu gleiche Ziele verfolgen. Die an die Projekte gebundenen Entwickler könnten, würden sie gemeinsam an einem Projekt arbeiten, womöglich qualitativ bessere Software erzeugen. Außerdem ist für den Anwender die Wahl

[7] Quelle: Persönliche Gespräche des Autors auf Konferenzen und mit Partnern in der Industrie.

des für ihn passenden Projekts schwerer, wenn mehrere nahezu gleiche Lösungen existieren.

Aktualisierung Freie Projekte unterliegen je nach Organisation und Teamgröße den unterschiedlichsten Entwicklungs- und Aktualisierungszyklen. Das Verfolgen und Bewerten von Aktualisierungen stellt die Anwender vor ernstzunehmende Herausforderungen [Kno13], da fehlende Prozesse und Standards nicht immer eine Unterscheidung von funktionserweiternden und fehlerbereinigenden Updates erlauben.

Lizenz Auch freie Software steht unter rechtlichem Schutz durch Lizenzen wie die GNU GPL. Dies bringt für den Nutzer auch Pflichten mit sich, wie die Nennung verwendeter Projekte oder die Freigabe des eigenen Quellcodes.

Dies sind nur vier Punkte, die vor dem Einsatz freier Software bedacht werden sollten und somit in direktem Zusammenhang mit dem Systementwurf stehen. Einige der Probleme freier Software spiegeln sich auch in anderen Bereichen mit freien Lizenzen wieder. Bisher existieren aber keine allgemeingültigen Lösungsstrategien für diese Probleme.

1.4 Zielsetzung

Die Arbeit vermittelt ein vertieftes Verständnis für die Systemkonzepte von *heterogenen rekonfigurierbaren Systemen (hrS)* und eingebetteten Systemen mit Linux-Betriebssystem. Grundlage dafür sind die vom Autor gewonnenen Erkenntnisse über den Entwurf und die Architektur eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem, deren Umsetzung nur durch das vertiefte Verständnis beider Themenkomplexe möglich ist. Daraus resultiert ein Beitrag zur Verbesserung des Entwurfsflusses vorgenannter Systeme durch ihre Betrachtung auf einer hohen Abstraktionsebene, wohl wissend, dass für die einzelnen Domänen etablierte Vorgehensweisen bei der Umsetzung zum Produkt existieren. Ein expliziter Fokus liegt auf der praktischen Relevanz und Umsetzbarkeit der vorgestellten und diskutierten Architekturen, Methoden und Ansätze.

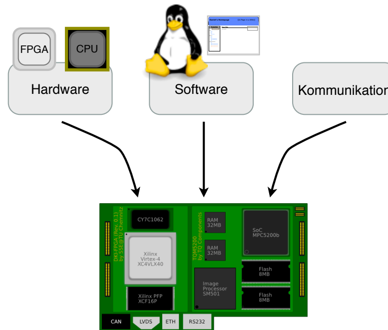


Abbildung 1.1: Teilgebiete beim Entwurf eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem

Abbildung 1.1[8] rekonfigurierbaren Systems und damit auch in den folgenden Kapiteln eine Rolle spielen.

Software ist in vielen Bereichen ein Innovationstreiber und dient immer häufiger als wesentliches Unterscheidungsmerkmal zwischen Konkurrenten. Linux erlaubt durch seine Philosophie, sein Konzept und seine Flexibilität genau diese Differenzierung. Beim Systementwurf stehen Fragen nach den notwendigen Hardwareressourcen wie beispielsweise Unterstützung bestimmter Hardware, *CPU*-Architektur und -voraussetzungen, minimale Anforderungen an *RAM* und Flash-Speicher oder die Nutzbarkeit verschiedener Protokolle und Schnittstellen im Vordergrund. Außerdem spielen die Spezifikation, Softwareentwicklung und das Lifecycle Management von System- und Anwendungssoftware eine Rolle.

Bei der Hardware liegt der Fokus auf dem Einsatz eingebetteter Systeme und reprogrammierbarer Schaltkreise, sowohl zur Erfassung und Auswertung von Sensordaten, als auch zur Nutzung als Beschleuniger. Von Interesse sind im Besonderen Konzepte und Methoden für die Konfiguration und die Rekonfiguration der Hardware, die Systemarchitektur und das Update der Systeme im laufenden Betrieb in Verbindung mit Linux.

[8] Das Original der Abbildung des Linux-Pinguins „Tux“ stammt von Larry Ewing <lewing@isc.tamu.edu> und wurde mit dem freien Bildbearbeitungsprogramm „The GIMP“ erstellt.

Die Kommunikation zwischen den Komponenten eines Systems wird in Zusammenhang mit der Architektur von heterogenen rekonfigurierbaren Systemen betrachtet. Die Kommunikation nach außen rückt die vorliegende Arbeit hingegen nicht in den Vordergrund. Dennoch spielt sie bei der Bearbeitung der Problemstellungen stets eine Rolle. Ein Aspekt bei allen Untersuchungen war die Möglichkeit, die Systeme aus der Ferne zu überwachen, zu konfigurieren und die System- und Anwendungssoftware zu aktualisieren und zu erweitern.

1.5 Verwandte Arbeiten

Die verschiedenen Ausprägungen der *run-time Reconfiguration (RTR)* spielen in einer großen Anzahl wissenschaftlicher Abhandlungen eine Rolle. Aktuelle Arbeiten auf dem Gebiet der heterogenen rekonfigurierbaren Systeme konzentrieren sich auf Architekturen, bei denen ein Prozessorkern in Verbindung mit rekonfigurierbarer Logik auf einem Schaltkreis integriert ist (sog. *rekonfigurierbare System-on-Chip (rSoC)*). Allgemeine Systemarchitekturen finden hingegen nur wenig Beachtung. Im Wesentlichen sind hier die Arbeiten von Compton/Hauck, Todman und Koch [CH02; Tod+05; Koc04] zu nennen (vgl. Kapitel 3). Eine gezielte Betrachtung der Systemarchitekturen aus Sicht der Rekonfiguration erfolgt jedoch nicht.

Das Interesse an Linux als Betriebssystem für eingebettete Systeme ist ungebrochen, auch wenn es hier nur eine begrenzte Menge an Primärliteratur gibt. Die Standardwerke zu diesem Thema [CRK05; Hal06; Yag+08; SGD09] konzentrieren sich weitgehend auf die Beschreibung der Umsetzung an konkreten Beispielen. Eine Verallgemeinerung von Entwurfsabläufen erfolgt kaum oder gar nicht. Ebenso gibt es keine dem Autor bekannten Arbeiten zu allgemeinen Abläufen bei Implementierung und Pflege von Linux-Systemen für eingebettete Systeme.

Bei der Betrachtung von Betriebssystemen für heterogene rekonfigurierbare Systeme spielen meist rSoCs und *Real-Time Operating Systems (RTOSs)* eine Rolle. Im Fokus stehen die Themen HW/SW-Codesign und Platzierung rekonfigurierbarer Module. Andere Architekturen sowie Konfigurationsstrategien für Systeme ohne *dynamisch-partielle Rekonfiguration (dpR)* werden

kaum betrachtet. Ein beachtenswertes System ist im Rahmen des Egret-Projektes entstanden [BW03] und bietet eine rSoC-Plattform mit angepasstem Linux-Betriebssystem. Mit *BORPH*, dem *Berkeley Operating System for ReProgrammable Hardware* [So07] existiert ein Ansatz, bei dem ein Linux-System um Funktionen für rekonfigurierbare Hardware ergänzt wurde. Einen ähnlichen Ansatz bietet das *ACCelerator File System (ACCFS)* [Str05; Str+09]. Die Umsetzung von BORPH greift tief in das Linux-System ein. ACCFS mindert diesen Umstand, erfordert jedoch weiterhin deutliche Anpassungen im Linux und bei der Software, die auf die konfigurierbaren Ressourcen zugreift. Beide Ansätze basieren weiterhin auf zusätzlichen Hardwareressourcen für die Unterstützung der dynamischen (Re-) Konfiguration.

1.6 Struktur dieser Arbeit

Die Arbeit gliedert sich in 8 Kapitel, deren Inhalt nachfolgend kurz zusammengefasst wird.

Kapitel 2 ab Seite 35 definiert wesentliche Begriffe und führt in die allgemeine Theorie des Systementwurfes ein. Weiterhin werden die Grundlagen für das Verständnis eingebetteter und rekonfigurierbarer Systeme dargelegt, sowie allgemeine Aussagen zu Betriebssystemen getroffen. Ergänzend erfolgt die Darstellung der domänenspezifischen Entwurfsabläufe, sowie deren Zusammenfassung in einem angepassten Entwurfsfluss für heterogene rekonfigurierbare Systeme.

Kapitel 3 ab Seite 71 führt in das Gebiet der heterogenen rekonfigurierbaren Systeme ein. Der Begriffserklärung folgt die Darstellung grundsätzlicher Systemarchitekturen. Anschließend werden diese Architekturen in Ergänzung zu bestehenden Ansätzen vertieft diskutiert und ein auf dieser Diskussion aufbauender Bezug zu eingebetteten Systemen und zum Entwurfsfluss hergestellt. Abschließend erfolgt die Verbindung der neu gewonnen Erkenntnisse mit den allgemeinen Abläufen in Betriebssystemen.

Kapitel 4 ab Seite 95 stellt in kompakter Form das Betriebssystem Linux dar und diskutiert dessen Einsatz in eingebetteten Systemen und heterogenen rekonfigurierbaren Systemen. Einigen allgemeinen Grundlagen zu Aufbau und

Ressourcenbedarf des Betriebssystems folgt eine Systematisierung gebräuchlicher Abläufe beim Systemmanagement. Anschließend erfolgt eine ebenfalls systematisierte Darstellung der Entwurfsabläufe beim Einsatz des Betriebssystems in eingebetteten Systemen, unter Beachtung der damit verbundenen Änderungen im Vergleich zu den eingangs dargelegten Grundlagen. Abschließend wird die Verbindung von Linux und heterogenen rekonfigurierbaren Systemen beschrieben.

In Kapitel 5 ab Seite 135 erfolgt die Vorstellung eines neuartigen Spezifikationsansatzes für Linux-Systeme. Damit verbunden werden die Themen Spezifikation, Test und Verifikation von Linux-Systemen betrachtet.

Kapitel 6 ab Seite 145 überführt die allgemeinen Ansätze der Kapitel 3, 4 und 5 in einen konkreten praktischen Kontext. Dabei werden eingangs die verwendeten Werkzeuge und Methoden erläutert. Anschließend folgt die Darstellung konkreter Umsetzungen.

Kapitel 7 ab Seite 199 beleuchtet weitere Konzepte einer generalisierten Plattform zur Erfassung von Sensordaten. Im Speziellen werden eine Umsetzung einer datenbankbasierten Sensordatenerfassung und das Konzept eines Sensornetzwerks vorgestellt.

Kapitel 8 ab Seite 215 fasst die Arbeit zusammen und gibt einen Ausblick auf mögliche weiterführende Arbeiten.

2 Grundlagen

Dieses Kapitel definiert notwendige Grundbegriffe und führt in den Entwurfsablauf für die wesentlichen Teilgebiete der eingebetteten heterogenen rekonfigurierbaren Systeme ein. Es erläutert Hardwaregrundlagen und Systemarchitekturen eingebetteter Systeme und heterogener rekonfigurierbarer Systeme. Weiterhin werden Verfahren der Rekonfiguration von Hardwaresystemen zur Laufzeit und Grundlagen zu Betriebssystemen vorgestellt.

2.1 Begriffsdefinitionen

Aufgrund der Dynamik bei der wissenschaftlichen Untersuchung und technischen Weiterentwicklung eingebetteter Systeme und ihrer weiten Verbreitung in verschiedenen Anwendungsfeldern herrscht ein hoher Grad der Diversifizierung von Begriffen. Daher werden an dieser Stelle, in Anlehnung an die Dissertationsschrift von Erik Markert [Mar09], die für diese Arbeit gültigen Definitionen grundsätzlicher Begriffe dargestellt.

- Der Begriff **System**, abgeleitet vom griechischen *systema* („das Gebilde, Zusammengestellte, Verbundene“) bezeichnet ein Gebilde, dessen wesentliche Teile so aufeinander bezogen sind und in einer Weise wechselwirken, dass sie als aufgaben-, sinn- oder zweckgebundene Einheit (d. h. als Ganzes) angesehen werden und sich in dieser Hinsicht gegenüber der Umwelt abgrenzen. Nach Ropohl [Rop78] sind *technische Systeme* gekennzeichnet durch die drei Ein- und Ausgabegrößen – Energie, Materie und Information – durch die sie mit ihrer Umwelt wechselwirken.

- Die DIN-Norm 44300 [Deu85] definiert **Hardware** als „Gesamtheit oder Teil der apparativen Ausstattung von Rechensystemen“. Damit umfasst sie die materiellen Systemkomponenten.
- **Software**, ebenfalls nach DIN 44300 [Deu85], bezeichnet die „Gesamtheit oder Teil der Programme für Rechnersysteme“, und somit die immateriellen Anteile eines Systems.
- **Simulation** ist die Nachbildung des Ist-Verhaltens von Systemen, Komponenten oder (Teil-) Schaltungen mit softwaretechnischen Mitteln [HM04].
- **Emulation** ist die Nachbildung des Ist-Verhaltens von Systeme, Komponenten oder (Teil-) Schaltungen mit hardwaretechnischer Unterstützung [HM04].
- **Verifikation** ist der Vergleich des geforderten Soll-Verhaltens mit dem im Ergebnis der Entwurfsaktivitäten entstandenen Ist-Verhalten [HM04].
- Bei einer **Abstraktion** werden reale Zusammenhänge vereinfacht dargestellt. Damit einher geht meist ein Verlust an Genauigkeit, wobei die Komplexität im Allgemeinen sinkt. Im Systementwurf unterscheidet man mehrere Abstraktionsebenen.
- Eine **Synthese** im Sinne des Systementwurfs bezeichnet die Transformation einer System- oder Komponentenbeschreibung von einer höheren Entwurfsebene in eine niedrigere.

Entwurfsverfahren Es existieren drei wesentliche Entwurfsverfahren, die in allen Entwurfsabläufen eine zentrale Rolle spielen [SN92; Mar09]. Beim *Top-Down-Entwurf* werden die Entwurfsebenen „von oben“, also beginnend mit der Sicht auf das Gesamtsystem durchlaufen. Durch Verfeinerung, wie beispielsweise die Zerlegung in Komponenten, werden nun die Ebenen mit höherem Detail-, aber geringerem Abstraktionsgrad erreicht. Beim *Bottom-Up-Entwurf* hingegen beginnt der Entwurf auf der Ebene mit dem höchsten Detailgrad und man bewegt sich – im Sinne der Abstraktion – durch Komposition auf höhere Entwurfsebenen. Beide Entwurfsverfahren haben aufgrund ihrer Vor- und Nachteile auch heute noch ihre Daseinsberechtigung und eine

Verknüpfung der Ansätze liegt nahe. Dieses Vorgehen wird – je nach spezieller Ausprägung – in der Literatur als *Meet-in-the-Middle* oder *Jo-Jo-Verfahren* bezeichnet.

2.2 Systementwurf und Entwurfsablauf

Zur Beschreibung des Entwurfsablaufes bzw. der Entwicklung eines Systems existieren verschiedene Modelle, die die Schritte von der Idee zum fertigen Produkt beschreiben. Die Modelle unterscheiden sich in Abhängigkeit der Domäne für die sie entwickelt wurden, bieten jedoch auch immer Parallelen. Allen Modellen ist gemein, dass sie den Prozess mit Hilfe von Ebenen beschreiben, wobei der Abstraktionsgrad von der höheren zur darunterliegenden Ebene sinkt. Die Ergebnisse einer Ebene bilden jeweils den Ausgangspunkt für die Arbeiten in der nächsten Ebene. Nachfolgend werden drei grundlegende Modelle des Systementwurfs erläutert.

In [HT10] wird für die Entwicklung eines eingebetteten Systems das in Abbildung 2.1 auf der nächsten Seite dargestellte, vereinfachte Wasserfall-Modell gezeigt. Über die Anforderungsanalyse kommt man von der Idee zur Spezifikation, die die Grundlage für den Entwurfsprozess bildet. Beim anschließenden Entwurf[1] erfolgt die Umsetzung der Spezifikation in eine Implementierung. Ausgehend von der Implementierung beinhaltet der letzte Schritt die Herstellung des Produktes.

In Deutschland hat sich das V-Modell [IAB06] als Spezialisierung des Wasserfallmodells für die Beschreibung des allgemeinen Entwurfsablaufs etabliert (vgl. [HT10]). Die *Koordinierungs- und Beratungsstelle für Informationstechnik in der Bundesverwaltung (KBSt)* empfiehlt es bei zivilen und militärischen Vorhaben des Bundes und auch bei öffentlichen Aufträgen seitens der Bundesbehörden wird es vorausgesetzt. Durch seine Bedeutung und breite Verwendung entstanden verschiedene abgeleitete Diagramme. Abbildung 2.2 auf der nächsten Seite zeigt das V-Modell wie es beispielsweise bei Heinkel [Hei09] Verwendung findet und das auch als Basis für die folgende Beschreibung dient. Die Betrachtung erfolgt von links oben entlang des „V“ nach rechts oben.

[1] bzw. bei der Synthese

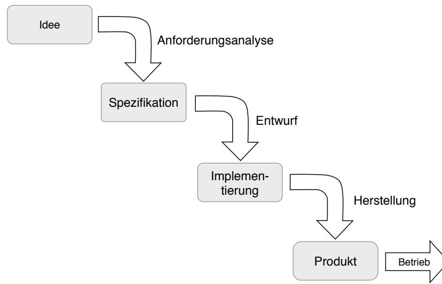


Abbildung 2.1: Entwicklung eines eingebetteten Systems (Quelle: [HT10])

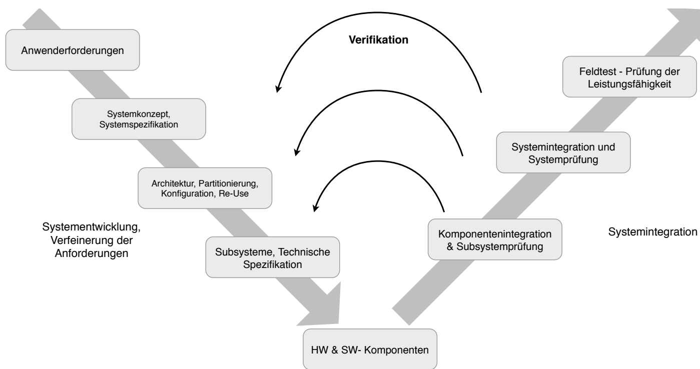


Abbildung 2.2: Angepasstes V-Modell (Quelle: [Hei09])

Diese Form des V-Modells geht davon aus, dass die Anforderungen an das System bereits formuliert sind, z. B. in Form eines Lastenheftes. Daraus erfolgt durch eine Anforderungsanalyse die Erarbeitung eines Systemkonzepts und der funktionellen Systemanforderungen. Im nächsten Schritt folgt die Partitionierung in Hard- und Software sowie die Auswahl der Zielarchitektur. Dabei werden bereits vorhandene und daher getestete Funktionsblöcke im Sinne des „Re-Use“ (*dt. Wiederverwenden*) berücksichtigt, um so die Entwurfszeit zu verringern und potentiell fehlerträchtige Neuentwürfe zu vermeiden. Nun erfolgt die Definition von Subsystemen und damit eine Verfeinerung der eingangs formulierten Anforderungen, bevor es zur eigentlichen Realisierung der

Komponenten kommt. Der rechte Ast umfasst die Phasen der Systemintegration. Die einzelnen Komponenten werden zu Subsystemen integriert und getestet. Anschließend wird das Gesamtsystem aufgebaut und ebenfalls getestet. Zuletzt erfolgen Feld- und Leistungstests. Fehler sollten so früh wie möglich entdeckt werden, da dadurch die Kosten zur Beseitigung geringer ausfallen. Die beiden Äste des „V" sind durch die notwendige Verifikation miteinander verbunden, die sicher stellt, dass das endgültige System den zu Beginn formulierten Anforderungen gerecht wird.

Das Buch von Marwedel [Mar08] orientiert sich am vereinfachten Informationsfluss eingebetteter Systeme, der in Abbildung 2.3 dargestellt ist.

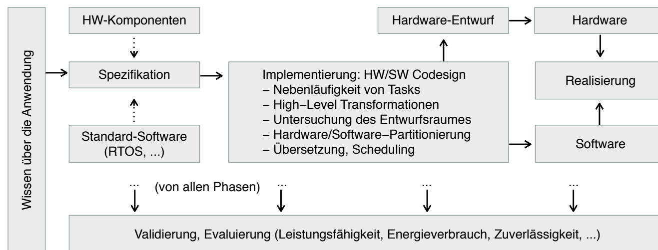


Abbildung 2.3: Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme (Quelle: [Mar08])

Zentraler Punkt ist der Fluss der Informationen über das Entwurfsobjekt, also die Weitergabe von Phasenergebnissen zwischen den Abstraktionsebenen. Dabei wird explizit berücksichtigt, dass der Entwurf eingebetteter Systeme meist Hardware und Software umfasst. Der gesamte Prozess beginnt mit einer Idee, die dann in die Spezifikation überführt werden muss, was den eigentlichen Beginn des Entwurfs markiert. Bei der Spezifikation sollten vorhandene Hard- und Softwarekomponenten berücksichtigt werden. Die Entwurfsaktivitäten, die die zentrale Rolle im Informationsfluss spielen, umfassen dann u. a. das Abbilden von Operationen auf parallele Tasks, High-Level-Transformationen, Entwurfsraumbetrachtungen, das Abbilden von Aufgaben auf Hard- oder Software[2] sowie die Übersetzung der Software und die Ablaufplanung. Der zusätzliche Hardware-Entwurf deutet an, dass eventuell der

[2] HW/SW-Partitionierung

Einsatz spezialisierter Hardware für bestimmte Aufgaben notwendig ist. Für die Realisierung erfolgt abschließend die Zusammenführung von Hard- und Software. Der gesamte Entwurfsablauf erfordert immer wieder die Validierung und Evaluierung der Entwurfsergebnisse hinsichtlich der Festlegungen in der Spezifikation.

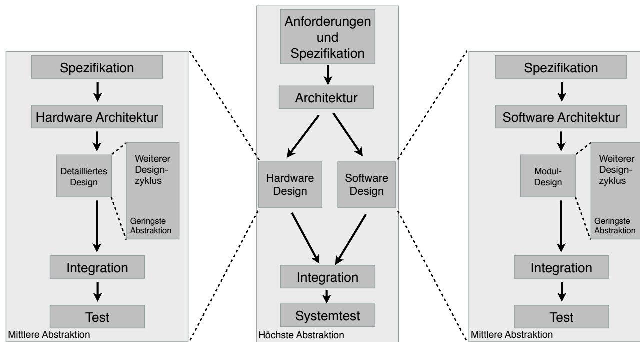


Abbildung 2.4: Hierarchischer Designflow nach [Wol10]

Eine Vielzahl weiterer Entwurfsmethoden findet sich in der Artikelserie von Wayne Wolf [Wol10]. Er stellt klar, dass jeder Designschritt eines abstrakten Entwurfsmodells wiederum einen eigenen Entwurfsfluss beinhaltet, was in Abbildung 2.4 dargestellt ist. Dieser Punkt wird an den entsprechenden Stellen dieser Arbeit wieder aufgegriffen.

Das V-Modell von Heinkel und der von Marwedel genutzte Informationsfluss beim Entwurf eingebetteter Systeme beschreiben das gleiche, schrittweise Vorgehen von der Idee über Spezifikation, Partitionierung und Implementierung hin zu Integration und Realisierung. In beiden Abläufen spielen Test und Verifikation eine wesentliche Rolle. Das V-Modell behält dabei einen höheren Abstraktionsgrad, während sich das Modell von Marwedel deutlicher an eingebetteten Systemen orientiert. Allen Modellen ist gemein, dass sie einen einfachen Weg von der Idee zum Produkt suggerieren, der in der Praxis nur selten existiert. Oftmals ist es notwendig, Entwurfsaktivitäten zu wiederholen, da Test oder Verifikation Fehler oder Diskrepanzen zur Spezifikation aufgedeckt haben. Hier zeigt sich, welches Gewicht die Anforderungsanalyse hat, die nur im Modell von [HT10] explizit benannt wird. Lediglich die korrekte

Überführung der oft informell formulierten Anforderungen in eine geschlossene Spezifikation erlaubt einen bedarfsgerechten Entwurf und damit ein den Anforderungen entsprechendes Produkt. Für die Darstellung der Spezifikation eigenen sich in besonderem Maße formale Sprachen, die eine automatisierte Überprüfung auf Vollständigkeit und Widerspruchsfreiheit erlauben. Eine ausführliche Erläuterung zu Spezifikationssprachen für eingebettete Systeme findet sich in [Mar08, Seite 13 ff.].

2.3 Eingebettete Systeme

In seinem Buch definiert Marwedel [Mar08] eingebettete Systeme in einer sehr ausführlichen und anschaulichen Art, die in ihren wesentlichen Zügen hier noch einmal wiedergegeben wird.

> *Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.*

Basierend auf dieser grundsätzlichen Definition erläutert Marwedel weiterhin wesentliche Eigenschaften eingebetteter Systeme. Diese

- interagieren mit der Umwelt über Sensoren und Aktoren,
- müssen verlässlich sein,
- müssen effizient sein,
- sind in den meisten Fällen applikationsspezifisch,
- besitzen oft eine spezialisierte Nutzerschnittstelle,
- werden in Echtzeit-Anwendungen genutzt,
- sind Systeme aus analogen und digitalen Komponenten (sog. hybride Systeme) und
- sind reaktive[3] Systeme.

[3] Ein reaktives System ist ein System, das in kontinuierlicher Interaktion mit seiner Umgebung steht und mit einer Geschwindigkeit arbeitet, die von seiner Umwelt vorgegeben wird [Ber95].

Marwedels abschließende Definition eingebetteter Systeme, die sich mit der Auffassung des Autors deckt, ist die Folgende:

> *Natürlich hat nicht jedes eingebettete System alle oben genannten Eigenschaften. Wir können „eingebettete Systeme" also wie folgt definieren:* ***Informationsverarbeitende Systeme, welche die meisten der oben aufgezählten Eigenschaften erfüllen, heißen eingebettete Systeme.*** *Diese Definition beinhaltet eine gewisse Unschärfe. Allerdings erscheint es weder notwendig noch möglich, diese Unschärfe zu vermeiden.*

Im Buch von Schröder, Gockel und Dillmann [SGD09] wird im Gegensatz zu Marwedel nicht mehr nur der klassische Definitionsansatz des „unsichtbaren Rechners" genutzt, sondern auch auf die Besonderheiten im Vergleich zur klassischen *Personal Computer (PC)*-Technik hingewiesen. Ausgehend von der nachfolgend wiedergegebenen Grunddefinition werden bestimmte Eigenschaften wie der Formfaktor, die Lokalität, die dedizierte Anpassung an eine Aufgabe, die optionale Echtzeitfähigkeit sowie Robustheit, Wartungsarmut und Langlebigkeit angeführt. Die Grunddefinition nach Schröder, Gockel und Dillmann ist:

> *Der Begriff des eingebetteten Systems bezeichnet einen Rechner (Mikrocontroller, Prozessor, DSP, SPS), welcher dezentral in einem technischen Gerät, in einer technischen Anlage, allgemein in einem technischen Kontext eingebunden (eingebettet) ist. Dieser Rechner hat die Aufgabe, das einbettende System zu steuern, zu regeln, zu überwachen oder auch Benutzerinteraktion zu ermöglichen und ist speziell für die vorliegende Aufgabe angepasst.*

Dieser Ansatz ist unter Berücksichtigung der in Abschnitt 1.1 geschilderten Entwicklung durchaus gerechtfertigt. Vornehmlich (aber nicht ausschließlich) Geräte im Endkundenmarkt – wie Tablet-PCs oder *Personal Digital Assistants (PDAs)* – werden durchaus als Computer wahrgenommen, erfüllen aber alle wesentlichen Kriterien eingebetteter Systeme und sind daher auch als solche zu betrachten. Schröder, Gockel und Dillmann gehen daher auch auf die Punkte Formfaktor, Mechanik, Kühlung, Robustheit, genutzte Speichermedien, gebotene Schnittstellen, Stromversorgung, Chipsätze, Watchdogs und Echtzeitfähigkeit ein.

In der vorliegenden Arbeit werden solche Rechensysteme als *eingebettete Systeme* verstanden,

- die über mindestens einen Prozessor verfügen,
- die durch
  - ihren mechanischen Aufbau,
  - ihre elektrischen und elektronischen Komponenten,
  - ihre Nutzerschnittstelle,
  - ihre Software und
  - die ihnen zur Verfügung stehenden Ressourcen

  an eine spezielle Aufgabenstellung angepasst sind und
- deren grundsätzlicher Softwareumfang nicht notwendigerweise vom Nutzer veränderbar ist.

Diese absichtlich sehr weit gefasste Definition soll dem allgemeinen Verständnis eingebetteter Systeme in Wirtschaft, Wissenschaft und Endverbrauchermarkt gerecht werden und erlaubt eine saubere Abgrenzung zu Computern wie Desktop-PCs, Servern oder Mainframes. Sie berücksichtigt die eingangs dargelegten Auffassungen von Marwedel und Schröder ebenso wie die persönlichen Erfahrungen des Autors. Zusätzlich deckt sich die Definition auch mit der Auffassung weiterer wesentlicher Literaturquellen, wie beispielsweise [Hal06], [Cat05] und [Yag+08].

2.3.1 Grundsätzliche Systemarchitektur eingebetteter Systeme

Computersysteme verbindet eine allgemeine, weithin bekannte Systemarchitektur, wie sie beispielhaft in Abbildung 2.5 auf der nächsten Seite[4] dargestellt ist. Die üblicherweise verwendeten Komponenten und die damit verbundenen Begriffe werden im Folgenden eingeführt und kurz erläutert. Für eine detaillierte Einführung in den Entwurf eingebetteter Systeme sei dem Leser [Cat05] empfohlen.

[4] Die Abbildung des RAM-Moduls stammt vom Nutzer „elkbuntu“ von openclipart [elk07].

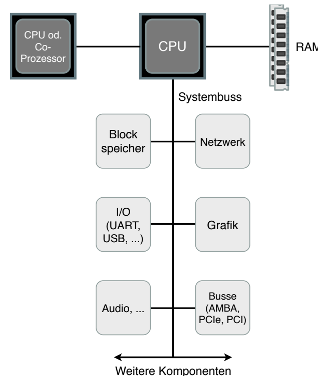


Abbildung 2.5: Generisches Computersystem in Anlehnung an [Cat05]

Prozessor Als *Prozessor* wird in der vorliegenden Arbeit eine elektrische Schaltung verstanden, die einen Algorithmus abarbeitet und gemäß der ihr übergebenen Befehle andere elektrische Schaltungen oder Maschinen steuert. Als *Central Processing Unit (CPU)* wird der Haupt- oder Zentralprozessor eines Computersystems bezeichnet, wie man ihn beispielsweise in typischen PC-Systemen findet. Er ist für die Abarbeitung allgemeiner Aufgaben geeignet. Ein *Co-Prozessor* hingegen ist für die Abarbeitung einer bestimmten Aufgabe oder Aufgabenkategorie spezialisiert. Der Prozessor delegiert üblicherweise bestimmte Aufgaben an verfügbare Co-Prozessoren.

Welche Prozessorarchitektur in einem eingebetteten System zum Einsatz kommt, hängt von Faktoren wie Preis, Leistungsaufnahme und Anwendungsfall ab. Die derzeit besonders beliebte ARM-Architektur kommt u. a. wegen ihrer geringen Leistungsaufnahme in mobilen Endgeräten zum Einsatz. Prozessoren mit MIPS-Architektur finden wegen ihrer Leistungsfähigkeit im TV- und Videobereich Anwendung. PowerPC und SH4 werden oft für den Einsatz unter rauen Umgebungsbedingungen gewählt. In [Yag+08, ab Seite 56] stellt Yaghmour einige verbreitete Prozessorarchitekturen vor.

Speicher Den Prozessoren eines Systems steht üblicherweise flüchtiger Speicher (*Random Access Memory (RAM)*) zur Verfügung, wobei Art und Menge je nach Geräteklasse und Einsatzzweck schwanken. Auch Art und

Menge des nichtflüchtigen Speichers ergeben sich durch den Einsatzzweck des Systems. Heute dominiert im Bereich der eingebetteten Systeme der Einsatz von Flash-Speichermedien, die entweder fest im System verlötet oder als Wechselmedien z. B. in Form von CompactFlash- oder SD-Karten ausgeführt sind. Nachteile wie die geringere Datentransferrate, die begrenzte Anzahl von Schreibzyklen und den höheren Preis im Vergleich zu modernen Magnetspeichern gleichen sie durch ihre geringe Bauform, ihren meist geringeren Energiebedarf und die Robustheit gegen Erschütterungen aus. Für große Datenmengen oder bei häufigen Schreibvorgängen werden hingegen Magnetspeicher (z. B. Festplatten) bevorzugt. In [Hal06, ab Seite 19] gibt Christopher Hallinan einen sehr ausführlichen Überblick über Speichertechnologien.

I/O und Kommunikation Im Allgemeinen kommunizieren die Komponenten eines Prozessorsystems über Busse. Je nach Anforderung und Komplexität des Prozessorsystems und der Kommunikationsarchitektur kommt nicht nur ein Bus, sondern ein hierarchisches oder segmentiertes Bussystem, z. B. aus Systembus und Peripheriebus, zum Einsatz. Bei modernen CPUs steht oft einer der System- oder Peripheriebusse explizit zur schnellen Kommunikation mit weiteren Prozessoren und dem Chipsatz zur Verfügung. Auch bei *System-on-Chips (SoCs)* wird gelegentlich ein System- oder Peripheriebus zur Erweiterung bereitgestellt.

Ein Prozessorsystem bietet in der Regel drei Möglichkeiten, mit externen Komponenten zu kommunizieren. Eine aus Hardwaresicht einfache, aber sehr flexible Schnittstelle sind General Purpose Input Output (GPIO)-*Pins*. Diese Pins können per Software angesteuert und somit nahezu jedes beliebige Signalspiel erzeugt werden. Limitierende Faktoren bei den GPIOs sind deren Anzahl sowie die maximale Geschwindigkeit, mit der die Pins gelesen bzw. geschrieben werden können. Weiterer Nachteil ist die fehleranfällige und zeitraubende Implementierung des Kommunikationsprotokolls in Software. Allerdings lässt sich der erzeugte Quellcode bei hinreichender Modularisierung und Strukturierung im Sinne des „Re-Use“ sehr gut in anderen Umgebungen wiederverwenden.

Zur Kommunikation mit externen Komponenten bieten *Mikrocontroller (µC)* und SoCs in aller Regel eine Vielzahl an integrierten *Input and Output*

(I/O)-Schnittstellen. Neben standardisierten Bus-Schnittstellen wie z. B. *Inter-Integrated Circuit (I^2C)*, *Serial Peripheral Interface (SPI)*, *Peripheral Component Interconnect (PCI)/Peripheral Component Interconnect Express (PCIe)*, *Controller Area Network (CAN)* u. v. m., finden sich auch programmierbare Schnittstellencontroller, die zur Umsetzung mehrerer, ähnlicher Protokolle genutzt werden können. Außerdem verfügen CPUs und Controller je nach Ausführung über spezielle Schnittstellen zur effizienten Kopplung mit Co-Prozessoren oder weiteren Prozessoren in einem Multi-CPU-System sowie zur Anbindung von externen Speichern.

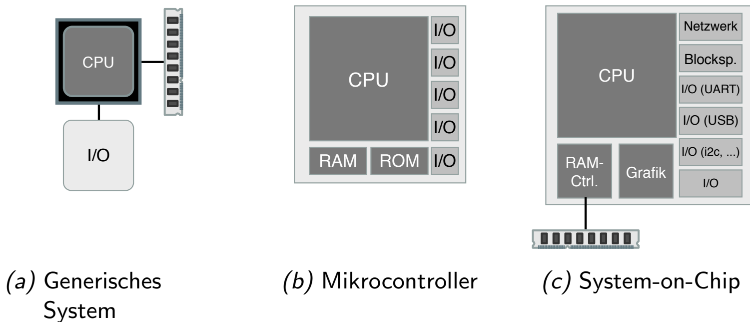
In diesem Zusammenhang sei nochmals auf [Yag+08] verwiesen, wo der Autor ab Seite 64 eine Vielzahl von Bussen und I/O-Schnittstellen kurz erläutert.

Mikrocontroller und System on Chip Der Begriff *Mikrocontroller* bezeichnet in dieser Arbeit einen Prozessor, der zusammen mit einigen I/O-Schnittstellen sowie *ROM*- und RAM-Speicher auf einem Chip integriert ist. Mit steigender Integrationsdichte wurde es möglich, mehr und komplexere Funktionseinheiten zusammen mit dem Prozessor auf einem Chip zu integrieren und so ein komplettes Grundsystem zu implementieren. Für diese Systeme hat sich in der Fachliteratur der Begriff *System-on-Chip* durchgesetzt. Bis in jüngster Vergangenheit verzichtete man auf die Integration von RAM und *Read Only Memory (ROM)*, um die Flexibilität bei der Systemintegration zu erhöhen. Somit waren zur Verwendung eines SoC neben RAM und ROM üblicherweise noch Taktquelle[5], Spannungsversorgung und Pegelanpassung zu implementieren. Inzwischen werden jedoch auch *System on Package (SoP)* angeboten, bei denen SoC und Speicher in einem Gehäuse integriert sind. Abbildung 2.6 auf der nächsten Seite illustriert die drei Grundsysteme CPU, µC und SoC.

In der vorliegenden Arbeit wird der Begriff *Prozessorsystem* als Oberbegriff für die soeben vorgestellten Systeme (CPU basiert, µC oder SoC) verwendet. Eine explizite Unterscheidung erfolgt nur dort, wo es aus Entwurfs- oder Systemsicht notwendig ist.

Ergänzend erfolgt an dieser Stelle noch die Einführung des Begriffs *System-on-Module (SoM)*. Ein SoM ist eine Leiterplatte mit einer definierten Bauform[6],

[5] Oft werden aus Gründen der Genauigkeit externe Taktquellen den Internen vorgezogen.
[6] Für übliche Bauformen siehe [SGD09, Seite 22].



(a) Generisches System

(b) Mikrocontroller

(c) System-on-Chip

Abbildung 2.6: Darstellung der Prozessorsysteme, die in einem eingebetteten System Anwendung finden

die ähnlich einem SoC alle Grundkomponenten eines Computersystems integriert. Im Vergleich zum SoC bietet ein SoM jedoch mehr Flexibilität, da es eine potentiell größere Fläche umfasst und auch Komponenten vereinen kann, die nicht in einem IC zusammengefasst werden können. Meist hat ein Hersteller mehrere SoM mit gleichen Schnittstellen aber verschiedenen Funktionen im Sortiment, die in vielen Fällen durch gleiche oder ähnliche Schnittstellenbelegungen gekennzeichnet sind. Daher ist eine Systemanpassung an wechselnde Spezifikationen durch einfachen Modultausch möglich.

Heterogene rekonfigurierbare Systeme sind unabhängig von der Prozessorarchitektur. Allerdings kann die Wahl des Prozessorsystems aufgrund seiner eigenen Spezifika auch die Systemarchitektur des heterogenen rekonfigurierbaren Systems beeinflussen, was Kapitel 3 ab Seite 71 weiter diskutiert. In der vorliegenden Arbeit wird der Begriff *Hostprozessor* als Bezeichnung für das Prozessorelement in einem heterogenen rekonfigurierbaren System verwendet.

2.3.2 Hardwareentwurf, -test und -verifikation

Der Entwurf eingebetteter Systeme beinhaltet Aufgaben aus den Bereichen des Hardware- und des Softwareentwurfs und erfordert daher insgesamt ein methodisch gestütztes Vorgehen. Der Hardwareentwurf von eingebetteten Sys-

temen umfasst eher selten das Design von digitalen Schaltkreisen, wie es in Anhang B auf Seite 227 beschrieben wird.

Vielmehr geht es um den Entwurf von Leiterplatten und die damit verbundene Auswahl geeigneter Prozessoren, Speicher und Peripherieschaltkreise [Kha05; Cat05]. Unterstützt wird dieser Prozess beispielsweise durch:

- festgelegte Designabläufe (siehe [Wol10]),
- die Erfahrung des Designteams,
- *Electronic Design Automation (EDA)* Werkzeuge mit integrierter Plausibilitätsprüfung [Amm87],
- Simulationswerkzeuge für die Schaltungs- oder Thermosimulation, für die Signalintegrität und für elektromagnetische Interferenzen sowie
- Reviewprozesse im Entwurfsteam.

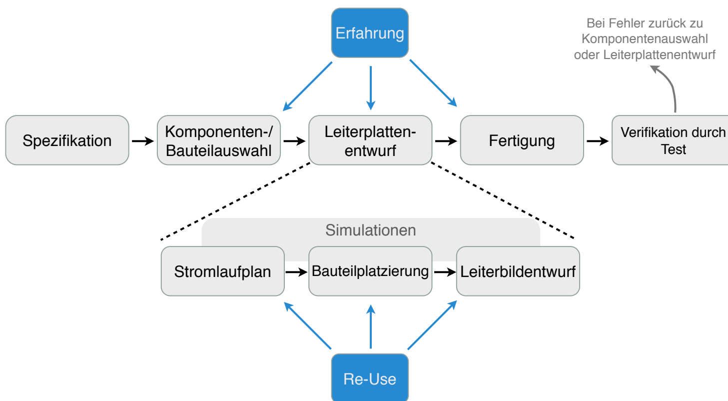


Abbildung 2.7: Vereinfachte Darstellung des Hardware-Entwurfsablaufes

Abbildung 2.7 stellt einen vereinfachten Entwurfsablauf dar und orientiert sich dabei am bereits vorgestellten V-Modell. Die Abbildung zeigt, an welchen Stellen der Leiterplattenentwurf von „Re-Use“, also der Verwendung bereits verifizierter Module profitiert. Außerdem soll sie verdeutlichen, dass der Entwurfs- und Verifikationsprozess derzeit noch stark vom Expertenwissen der Entwerfer dominiert wird, da automatisierte Testverfahren aus verschiedenen, hier

nicht näher erläuterten Gründen an ihre Grenzen stoßen oder nicht genutzt werden. Die EDA-Werkzeuge unterstützen im Entwurfsprozess durch Plausibilitätsprüfungen (engl. *Design-Rule-Check*) sowie durch die automatisierte Platzierung von Bauelementen und Leiterbahnen. Der Stromlaufplan kann durch Simulation (z. B. SPICE [Mül90]) geprüft werden. Bei der Bauteilplatzierung und dem Leiterbildentwurf helfen Werkzeuge zur Plausibilitätsprüfung, zur Signalanalyse, für elektromagnetische Effekte und die Thermoanalyse. Allerdings fehlen häufig Modelle der verwendeten Komponenten und eine HW/SW Co-Simulation, die eine dynamische Analyse der Effekte ermöglichen würde, ist nur unter großen Einschränkungen möglich. Test und Verifikation erfolgen meist an Prototypen oder einer Kleinserie. Dabei kommen bewährte Testverfahren wie die JTAG-Schnittstelle nach IEEE 1149 ebenso zum Einsatz wie der manuelle Test nach einem Testplan. Im Idealfall wurde der Testplan aus der (formalen) Spezifikation des Boards erzeugt und die Tests werden von einer Person durchgeführt, die nicht an Spezifikation und Layout beteiligt war.

2.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) bilden eine Klasse der programmierbaren Logikschaltkreise (engl. *Programmable Logic Device (PLD)*, vgl. Anhang A auf Seite 223). Wesentliches Merkmal der FPGAs ist ihr Aufbau aus konfigurierbaren Logikblöcken und ebenfalls konfigurierbaren Verbindungen zwischen diesen, die eine Programmierung der logischen Funktionen erlauben. Die Konfigurationsdaten entstehen während des Entwurfsprozesses durch Synthese einer Funktionsbeschreibung (vgl. Anhang B auf Seite 227) und werden in Form eines sogenannten *Bitstroms* an das FPGA übertragen. In das FPGA gelangt die Konfiguration über speziell dafür vorgesehene Schnittstellen. Am Markt dominieren Modelle, die zur Speicherung der Konfiguration auf flüchtigen *Static Random Access Memory (SRAM)* setzen, was zusätzlich zum FPGA einen externen Speicher für die Konfigurationsdaten erforderlich macht.

Durch ihre Mächtigkeit haben FPGAs eine hohe Marktdurchdringung erlangt. Aktuelle Generationen der verschiedenen Hersteller vereinen feingranulare Logikblöcke mit dedizierten Funktionseinheiten (Hardmakros) für häufige

Anwendungsfälle. Neuere Modellreihen unterstützen verschiedene Methoden der dynamischen und partiellen Rekonfiguration. Weit entwickelte und gut gepflegte Toolflows unterstützen ihre Nutzer bei der Erstellung von Anwendungen. Selbst Software für den Entwurf von Leiterplatten bietet nun Unterstützung für Designs mit FPGAs verschiedener Hersteller (z. B. [Alto8b]). Trotzdem erfordert die Flexibilität dieser ICs von Designern und Anwendungsentwicklern ein hohes Maß an Verständnis für Technologie und Architektur und ein methodisches Vorgehen beim Entwurf (vgl. Anhang B auf Seite 227).

Der in Abbildung 2.8 dargestellte, allgemeine Entwurfsfluss für rekonfigurierbare Systeme entspricht grundsätzlich wieder dem V-Modell. In Erweiterung zum allgemeinen Entwurfsablauf (vgl. Anhang B auf Seite 227) ist hier der gesamte Lebenszyklus abgebildet, da die Konfiguration reversibler PLDs prinzipiell auch nach der Auslieferung eines Produktes an den Kunden geändert werden kann. Dieser Umstand ist im Bereich des HW-Entwurfs kaum beachtet, während er im SW-Entwurf von jeher integriert ist.

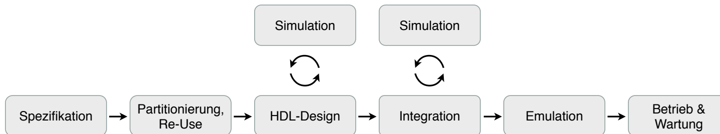


Abbildung 2.8: Entwurfsfluss für rekonfigurierbare Systeme

2.4.1 Schnittstellen und Konfigurationsmodi

In Hinblick auf rekonfigurierbare Hardwaresysteme sind besonders die Mechanismen zur Konfiguration von FPGAs von Bedeutung. SRAM-basierte Modelle müssen im System programmierbar bleiben (*In System Programmable (ISP)*), da sie ihre Konfiguration bei Unterbrechung der Stromzufuhr verlieren. Dieser Umstand ist bereits beim Entwurf des Systems zu berücksichtigen, denn beim Board-Layout sind entsprechende Ressourcen für einen geeigneten Konfigurationsspeicher, notwendige Verbindungsleitungen und die benötigte Pinbelegung einzuplanen.

Einige Schnittstellen und Konfigurationsmodi werden von nahezu allen FPGAs unterstützt, andere sind herstellerspezifisch. Dieser Abschnitt führt Schnittstellen und Datenformate für die Konfiguration ein, auf die im weiteren Verlauf zurückgegriffen wird. Dabei werden im Besonderen Produkte der Marktführer Xilinx und Altera [Xil09a; Alt09a] berücksichtigt. Für eine ausführliche Darstellung zu den Konfigurationsschnittstellen und Konfigurationsmodi wird auf die Literatur der Hersteller verwiesen, u. a. [Alt09b; Alt12b; Xil08e; Xil08g; Xil11c; Xil13; Tap07].

FPGAs unterstützen in den meisten Fällen mehrere Konfigurationsmodi, die sich nach zwei Merkmalen gruppieren lassen. Unterscheidet man nach der Art der Datenübertragung, so lassen sich *serielle* und *parallele* Konfigurationsmodi unterscheiden. Wird der Initiator eines Konfigurationsvorgangs zur Klassifikation verwendet, unterscheidet man, ob das FPGA als *Master* oder *Slave* bei der Datenübertragung agiert[7]. Als Master erzeugt das FPGA notwendige Steuersignale, wie z. B. Takt- oder Resetsignal. In vielen, aber nicht allen Fällen ist die Kombination von Master/Slave und Datenübertragungsart frei.

Die Wahl des Konfigurationsmodus erfolgt meist während des Systementwurfs und wird hauptsächlich durch vier voneinander abhängigen Faktoren bestimmt. Dies sind:

- die Architektur,
- die gewünschte Konfigurationsgeschwindigkeit,
- der auf der Leiterplatte vorhandenen Platz und
- die Anzahl und geplante Nutzung der zur Konfiguration verfügbaren Pins.

Die für eine vollständige Konfiguration benötigte Zeit T_K ergibt sich aus der Bitstromgröße (S_{BS}), der maximalen Konfigurationsfrequenz (f_K) und der Breite des Konfigurationsbusses (B_K).

$$T_K[s] = \frac{S_{BS}[bit]}{f_K[Hz] * B_K[bit]}$$

[7] Altera nennt die Modi *aktiv* und *passiv*.

Die maximale Bitstromgröße ist dabei durch das gewählte FPGA vorgegeben, allerdings variiert sie je nach Design und anderen hier nicht näher betrachteten Einstellungen. Der verwendete Konfigurationsbus hängt vom geplanten Konfigurationsspeicher, dem verfügbaren Platz auf der Leiterplatte und den am FPGA verfügbaren bzw. im Layout erreichbaren Pins ab. Gleichzeitig bestimmt der Konfigurationsbus auch maßgeblich die maximale Konfigurationsfrequenz (siehe Tabelle 2.1). Tabelle 2.2 auf der nächsten Seite gibt einen Überblick über die maximale Bitstromgröße einiger ausgewählter FPGAs. Sie zeigt, dass auch Bitströme für kleine FPGAs leicht eine Größe von mehreren Mebibyte[8] erreichen.

Tabelle 2.1: Geschwindigkeit der Konfigurationsschnittstellen für Xilinx FPGAs

Schnittstelle	Busbreite	max. Geschwindigkeit
Serial	1 bit	100 MHz
JTAG	1 bit	66 MHz
Parallel (extern)	8...32 bit	100 MHz
ICAP (intern)	8...32 bit	100 MHz

Durch die starke Segmentierung des FPGA-Marktes existiert eine große Anzahl von Konfigurationsschnittstellen. Deren Ausprägung und Benennung schwankt nicht nur zwischen den Herstellern, sondern auch zwischen den verschiedenen ICs eines Herstellers. Neben proprietären Schnittstellen bieten neuere FPGAs mehr und mehr standardisierte Schnittstellen, z. B. zur Ansteuerung von Flash-Speichern oder zur Nutzung von Bussystemen.

Eine häufig verwendete Schnittstelle ist der JTAG bzw. Boundary-Scan-Port [BO91; 02], den vermutlich alle FPGAs unterstützen. Mit Hilfe dieser Schnittstelle können einzelne ICs ebenso konfiguriert werden, wie Ketten aus mehreren Schaltkreisen. Außerdem lässt sich der Port für Tests und Fehlersuche verwenden. So lassen sich beispielsweise Virtex-4/5 FPGAs von Xilinx per JTAG initial konfigurieren und auch rekonfigurieren. Weiterhin ist

[8] Der IEC-Norm 60027-2 folgend werden Binärprefixe verwendet.

Tabelle 2.2: Abschätzung der Bitstromgröße für verschiedene FPGA-Varianten in Mebibyte

Hersteller, Serie	Typ	Bitstromgröße (Mebibyte)
Xilinx Virtex		
	Virtex 4	0.57...6.12
	Virtex 5	0.59...9.86
	Virtex 6	2.32...22.02
Xilinx Spartan		
	Spartan 3	0.05...1.58
	Spartan 6	2.57...4.02
Xilinx Series-7		
	Artix-7	3.65...9.28
	Kintex-7	2.87...17.87
	Virtex-7	13.26...53.33
Altera Stratix V	E, GX, GS, GT	11.10...46.18
Altera Stratix IV	E, GX, GS, GT	5.71...28.81
Altera Arria V	GX, GT, SX, ST	8.23...22.09
Altera Cyclone V	GX	2.46...23.67

auch das Auslesen des im FPGA befindlichen Bitstroms und das Verwenden des Ports für Fehlersuche und Tests möglich. Die Xilinx-FPGAs unterstützen die IEEE Standards 1149.1 und 1532. Ausführliche Hinweise zur Nutzung der JTAG/Boundary-Scan Funktionalität finden sich z. B. in [Xilo8e] und [Xilo8g]. Ein Anwendungsbeispiel beschreibt Abschnitt 6.3.4 auf Seite 161.

Als nichtflüchtiger Speicher kommen in Verbindung mit SRAM-basierten FPGAs schon seit langem *Electrical Eraseable Programmeable Read Only Memorys (EEPROMs)* zum Einsatz. Xilinx und Altera bieten speziell auf ihre FPGAs abgestimmte ICs an, die eine geeignete Schnittstelle zur schnellen Übertragung des Bitstroms sowie weitere Besonderheiten bieten. Xilinx nennt

diese ICs *Platform Flash* PROM *(PROM)* [Xil08c], Altera *Enhanced Configuration Devices (EPC)* oder *Altera® Serial Configuration Devices* [Alt09b]. Die grundsätzlichen Funktionen, die sich mit Hilfe der *Programmeable Read Only Memorys (PROMs)* realisieren lassen, ähneln sich bei beiden Herstellern. Je nach PROM- und FPGA-Größe können mehrere Konfigurationen (Bitströme) hinterlegt werden, die entweder ein FPGA nach Bedarf mit verschiedenen Designs konfigurieren oder zur simultanen Konfiguration mehrerer ICs dienen. Zum Befüllen der PROMs greifen beide Hersteller wiederum auf den JTAG-Port zurück und bieten geeignete Programmiergeräte zum Anschluss an einen PC. Die Nutzung eines PROMs beschreibt Abschnitt 6.4.4 auf Seite 177.

Als letzte Möglichkeit zur externen Konfiguration von FPGAs sollen hier Schnittstellen zur Nutzung von Flash-Speichern genannt werden. Xilinx-FPGAs unterstützen sowohl Flash-Speicher mit SPI- als auch solche mit paralleler Schnittstelle, wobei hierfür das *Byte Peripheral Interface (BPI)* zum Einsatz kommt. Altera nennt den Modus *Active Parallel Configuration*. Vor der Nutzung muss mit Hilfe der Datenblätter und Internetseiten der Hersteller geprüft werden, welche Flash-ICs unterstützt werden.

Einige FPGAs können auch von „innen heraus" konfiguriert werden, was nachfolgend in Abschnitt 2.5.2 eingehend erläutert wird. Dafür ist zusätzliche Hardware im FPGA erforderlich. Bei Xilinx heißt der verwendete, fest im FPGA verankerte Block *Internal Configuration Access Port (ICAP)*. Bei Altera wird der Controller als *PR Control Block (CB)* bezeichnet. Die Controller können vom FPGA-Design aus angesprochen werden und veranlassen und überwachen die entsprechenden Konfigurationsvorgänge im FPGA.

2.4.2 Datenformate

Wie bereits erläutert, liegt nach erfolgreicher Synthese ein Bitstrom vor, der die Konfigurationsinformationen für die SRAM-Zellen und Befehle für die Steuerlogik des ICs enthält. Dieses FPGA- und herstellerspezifische Format wird als „RAW-Format" bezeichnet. Weitere Datenformate erleichtern die Arbeit mit den Konfigurationsinformationen. Zwei Formate werden nachfolgend exemplarisch vorgestellt.

Mit dem *Serial Vector Format (SVF)* existiert ein herstellerübergreifendes Dateiformat zur Speicherung von JTAG-Operationsabläufen. Es eignet sich zur

Programmierung und Verifikation von JTAG fähigen ICs. Das von Texas Instruments und Teradyne entwickelte SVF wurde durch Xilinx zum *Xilinx Serial Vector Format (XSVF)* [BC09] weiterentwickelt, einem kompakten, binären Dateiformat zur Speicherung von Xilinx-bezogenen Abläufen für die JTAG-Konfiguration ihrer Schaltkreise. Mit der *Standard Test and Programming Language (STAPL)* hat Altera ein dem SVF ähnliches Dateiformat entwickelt, das seit 1999 durch die *JEDEC Solid State Technology Association* standardisiert ist [Alt09b]. Auch STAPL bietet ein komprimiertes Bytecode-Format ähnlich dem XSVF. Die offenen Formate XSVF und STAPL erlauben eine einfache Interaktion mit FPGAs, um diese beispielsweise durch externe Controller zu konfigurieren. Ein Anwendungsbeispiel findet sich in Abschnitt 6.3.4 auf Seite 161.

2.4.3 Weitere Funktionen und Merkmale

Die Hersteller konfigurierbarer Schaltkreise arbeiten ständig an Verbesserungen und Erweiterungen. Einige Funktionen, die die Konfiguration und die Datenströme betreffen, sind im Folgenden kurz erwähnt.

Fallback-Konfiguration und MultiBoot Einige FPGA-Serien von Xilinx und Altera bieten Funktionen, um je nach Zustand verschiedene Konfigurationen zu laden. Somit können beispielsweise mehrere anwendungsspezifische Konfigurationen in ein FPGA geladen werden (*MultiBoot/page mode feature*). Eine Fallback-Konfiguration (*Rückfallkonfiguration*) hingegen stellt die Funktion des Systems auch im Fehlerfall sicher. Hierzu wird zusätzlich zur anwendungsspezifischen eine (mit Sicherheit funktionsfähige) Konfiguration hinterlegt. Schlägt der anwendungsspezifische Konfigurationsvorgang z. B. nach einem Update fehl, lädt das FPGA die Fallback-Konfiguration, um das System zumindest in einen Minimalbetrieb zu versetzen.

Kompression, Sicherung der Bitströme Um Speicherplatz für die Bitströme zu sparen, können diese komprimiert werden. Während der Konfiguration dekomprimiert das FPGA den Datenstrom [Khu01]. Viele FPGAs und PROMs unterstützen die Verschlüsselung der Bitströme. Somit kann das in einem Design enthaltene geistige Eigentum (*Intellectual Property (IP)*) vor

Fremdzugriffen geschützt werden. Nicht zuletzt enthalten die Bitströme eine Prüfsumme, um Übertragungs- bzw. Entschlüsselungsfehler aufzudecken. Das Berechnen der Prüfsummen ist hinreichend bekannt [Mor05; Xil08d; Alt08a], sodass die Prüfsumme auch anderweitig genutzt werden kann.

2.5 Run-time Reconfiguration

Der englische Begriff run-time Reconfiguration (RTR) (Laufzeit-Rekonfiguration bzw. Rekonfiguration zur Laufzeit) bezeichnet die Möglichkeit, die Funktionalität eines Systems zur Laufzeit zu verändern, ohne dabei ein bestimmtes Konzept zu adressieren. Die Verwendung des Begriffs run-time Reconfiguration hat sich daher sowohl für Lösungen etabliert, die eine Veränderung von Software adressieren, als auch für solche, die auf der Veränderung von Hardwareeinheiten basieren. Im Folgenden stehen Systemkonzepte im Vordergrund, die rekonfigurierbare Hardwareeinheiten umfassen.

2.5.1 Formen der Run-time Reconfiguration

In diesem Abschnitt werden die grundsätzlichen Konzepte der RTR aus Systemsicht vorgestellt. Abschnitt 2.5.2 erläutert anschließend die einzelnen Konzepte mit direktem Bezug zu FPGAs.

Um die grundlegenden Ansätze der RTR zu beschreiben, ist erst die Klärung des Begriffs Laufzeit notwendig. Im Rahmen der Arbeit wird der Begriff wie folgt verstanden:

Laufzeit Der Begriff beschreibt im weitesten Sinne die Zeitspanne, die seit der Aktivierung bzw. Inbetriebnahme eines Systems vergangen ist. In technischen Zusammenhängen kann dies die Zeit seit dem letzten Stillstand oder die Zeit seit der ersten Inbetriebnahme ebenso umfassen, wie die Zeitspanne, die ein Signal vom Eintritt in ein System bis zum Austritt benötigt. In der Informatik beschreibt der Begriff hingegen meist die Zeitdauer zur Ausführung eines Programms. Für die Ausprägungen

der RTR bleibt diese begriffliche Unschärfe bestehen, da je nach Sichtweise auch hier die Veränderung der Systemfunktion

- seit der ersten Inbetriebnahme,
- allgemein während der Benutzung oder
- während der Laufzeit eines Programms/einer Aufgabe

zugrunde gelegt wird.

Basierend auf dem Laufzeitbegriff ergeben sich die folgenden Konzepte für die RTR aus Systemsicht, die anschließend durch einige Beispiele illustriert werden.

Statische Konfiguration Ist eine Veränderung der Konfiguration zur Laufzeit eines Systems nicht als Bestandteil der Systemfunktion vorgesehen, wird das System als *statisch* bezeichnet.

Dynamische (Re-) Konfiguration Im Gegensatz zur statischen Konfiguration ist hier die nachträgliche Veränderung der Systemfunktion als grundsätzliches Konzept vorgesehen. Die einfachste Form der *dynamischen Rekonfiguration (dR)* ist die *vollständige Rekonfiguration* [DH03]. Dabei steht das System für den Zeitraum der Rekonfiguration und einer gegebenenfalls notwendigen Initialisierung (z. B. Booten eines Betriebssystems) nicht zur Verfügung. Bei der *partiellen Rekonfiguration (pR)* werden nur Teile der Hard- oder Software des Systems neu konfiguriert, was u. U. den Konfigurationsaufwand vermindert. Bleibt ein System während der Rekonfiguration verfügbar, auch wenn seine Funktionen ggf. eingeschränkt sind, spricht man von dynamisch-partieller Rekonfiguration.

Die vollständige Rekonfiguration ist ein weit verbreitetes Konzept und fordert oft eine aktive Mitwirkung des Nutzers oder einen manuellen Eingriff in das System. Beispiele hierfür sind die Rekonfiguration eines FPGAs mit Hilfe eines PCs und eines externen Programmiergerätes, die Aktualisierung des Betriebssystemkerns eines PCs oder das Aufspielen einer neuen Firmware auf ein eingebettetes System, wie einen Fernseher oder ein Telefon. In den meisten Fällen kann die Rekonfiguration jedoch „im Feld" – also direkt am Nutzungsort des Systems – erfolgen. Dynamische Rekonfiguration findet ebenfalls im Bereich der Firm- bzw. Software-Updates Anwendung. Anstatt den gesamten

Flash-Speicher eines eingebetteten Systems zu überschreiben, werden nur geänderte Teile ersetzt. So kann Zeit während der Rekonfiguration und der Verteilung von Updates (z. B. per Netzwerk) gespart werden. Im Fall dynamisch-partieller Rekonfiguration kann die Systemfunktion durch einzelne Elemente aufrecht erhalten werden, während andere durch den Aktualisierungsvorgang nicht verfügbar sind. Auf Systemebene findet man dieses Konzept beispielsweise bei der Aktualisierung von Anwendungssoftware, Gerätetreibern oder von Firmware einzelner Systemkomponenten.

2.5.2 Run-time Reconfiguration mit FPGAs

Wie im vorhergehenden Abschnitt bereits angedeutet, ist die RTR nicht auf Software beschränkt. PLDs ermöglichen auch die Veränderung von Hardwarefunktionen zur Laufzeit, wobei sich die soeben eingeführten Konzepte dynamische Rekonfiguration, partielle Rekonfiguration und dynamisch-partielle Rekonfiguration wiederfinden. In Anhang A auf Seite 223 wird in Verbindung mit der Klassifikation von PLDs auf verschiedene Konfigurationsarchitekturen dieser ICs hingewiesen.

Bei Single Context FPGAs [CH99] wird der Schaltkreis bei jedem Konfigurationsvorgang angehalten und der gesamte Konfigurationsspeicher neu geschrieben. Alle internen Zustände gehen verloren. Diese Art der Rekonfiguration wird in der Literatur auch als vollständige Rekonfiguration bezeichnet. Partiell rekonfigurierbare Schaltkreise erlauben die Modifikation nur eines Teils des Konfigurationsspeichers. Dabei variiert die Granularität des veränderbaren Bereichs. So ist es bei manchen FPGAs möglich, einzelne Konfigurationsbits zu manipulieren, während andere nur die Modifikation von Speicherblöcken verschiedener Größe zulassen. Der IC steht hier während der Rekonfiguration ebenfalls nicht zur Verfügung. Der Vorteil partiell rekonfigurierbarer Schaltkreise ist die verringerte Größe des partiellen Bitstroms und die damit einhergehende potentielle Verringerung der Rekonfigurationszeit [Dye12]. Einen zusätzlichen Vorteil bieten FPGAs, die eine partielle Rekonfiguration im laufenden Betrieb ermöglichen. In diesem Fall werden nur die zu rekonfigurierenden Teile des Schaltkreises angehalten, während alle anderen Teile funktionsfähig bleiben. Für diese Art der Rekonfiguration werden in der Literatur verschiedene Begriffe gleichwertig verwendet. Häufig findet man die englischen Bezeichnungen *run-time partial reconfiguration*, *dy-*

namic reconfiguration, *dynamic partial* bzw. *partial dynamic reconfiguration*. Im deutschen Sprachraum wird meist der Begriff *dynamisch-partielle Rekonfiguration (dpR)* genutzt. Eine sehr gute Zusammenfassung der vier Konfigurationsklassen[9] nach Lysaght [LD94] und Wannemacher [Wan98] findet sich auch in [Mei10]. In [KR08] wird eine weitere Form der Rekonfiguration FPGA basierter Systeme eingeführt, die sogenannte *SPR*. Dabei werden alle benötigten Funktionen bereits zur Designzeit implementiert. Software ermöglicht das dynamische Zu- bzw. Abschalten von Funktionsblöcken. In Verbindung mit *Clock-Gating* bzw. *Power-Gating* kann auch dies zur Energieeinsparung führen.

Donthi nimmt in [DH03] eine weitere Unterscheidung vor, die in Verbindung mit Multi-Context FPGAs steht. Bezieht das FPGA bei der Rekonfiguration seine Daten über eine externe Schnittstelle, so spricht man von *off-chip* Rekonfiguration. Erfolgt die Rekonfiguration jedoch durch die Benutzung von auf dem IC vorhandenen Speicher (*on-chip* Speicher), bezeichnet man diesen Vorgang als *context* Rekonfiguration.

In Abhängigkeit davon, wer die Rekonfiguration eines FPGAs steuert, unterscheidet man nach extern gesteuerter Rekonfiguration (*exo-reconfiguration*/Exorekonfiguration) und interner Rekonfiguration (*endo-reconfiguration*/Selbstrekonfiguration) (vgl. [WB04]). Bei der Exorekonfiguration wird der Rekonfigurationsprozess durch eine externe Instanz, z. B. einen Prozessor eingeleitet. Bei der Selbstrekonfiguration hingegen leitet das FPGA-System die Rekonfiguration von sich aus ein.

Somit lassen sich zusammenfassend drei Fragen zum Rekonfigurationsvorgang formulieren, die in Tabelle 2.3 auf der nächsten Seite dargestellt sind und eine eindeutige Klassifikation des Systems erlauben.

Schnittstellen für die (Re-) Konfiguration

Nicht alle der in Abschnitt 2.4.1 vorgestellten Schnittstellen sind für jede Form der Rekonfiguration nutzbar. Dieser Umstand muss bereits bei der Systemspezifikation berücksichtigt werden. Ist die Art der geplanten Rekonfiguration

[9] Die vier Klassen sind konfigurierbar, rekonfigurierbar, partiell rekonfigurierbar und dynamisch rekonfigurierbar.

Tabelle 2.3: Zusammenfassende Darstellung der RTR mit FPGAs

Wie erfolgt die Rekonfiguration?	vollständige Rekonfiguration partielle Rekonfiguration dynamisch-partielle Rekonfiguration
Woher werden die Konfigurationsdaten bezogen?	on-chip Rekonfiguration off-chip Rekonfiguration
Wer steuert die Konfiguration?	Exorekonfiguration Selbstrekonfiguration

bekannt, begrenzt dies u. U. den Entwurfsraum des Systems, da nicht mehr alle Schnittstellen in Betracht kommen. Somit ist die frühzeitige Festlegung auf eine Methode ein Vorteil für den Entwurfsvorgang. Sie führt jedoch auch zum Ausschluss anderer Ansätze. Tabelle 2.4 auf der nächsten Seite gibt einen Überblick über die geeigneten Schnittstellen bei FPGAs von Xilinx und Altera.

Bei der Endo-Rekonfiguration gibt es zwei Ansätze. Eine Variante ist die Nutzung der internen Konfigurationsschnittstellen (ICAP bzw. CB), um eine partielle Rekonfiguration anzustoßen. Die zweite Variante heißt bei Altera *page mode feature* und bei Xilinx *MultiBoot* und ist in verschiedenen Serien verfügbar. Bei diesen Varianten werden mehrere Konfigurationen in einem Konfigurationsspeicher hinterlegt. Das FPGA wird initial mit einer „sicheren“ Konfiguration geladen und kann anschließend eine der anderen Konfigurationen aus dem Speicher selektieren und eine dynamische Rekonfiguration anstoßen.

Entwurfsmethoden für partielle und dynamisch-partielle Rekonfiguration

Alle derzeit am Markt erhältlichen FPGAs lassen sich für Systeme mit statischer oder dynamischer Rekonfiguration einsetzen. Immer mehr Hersteller bieten ICs, die auch partielle Rekonfiguration oder dynamisch-partielle Rekonfiguration unterstützen, unter anderem Xilinx, Altera, Atmel und Lattice

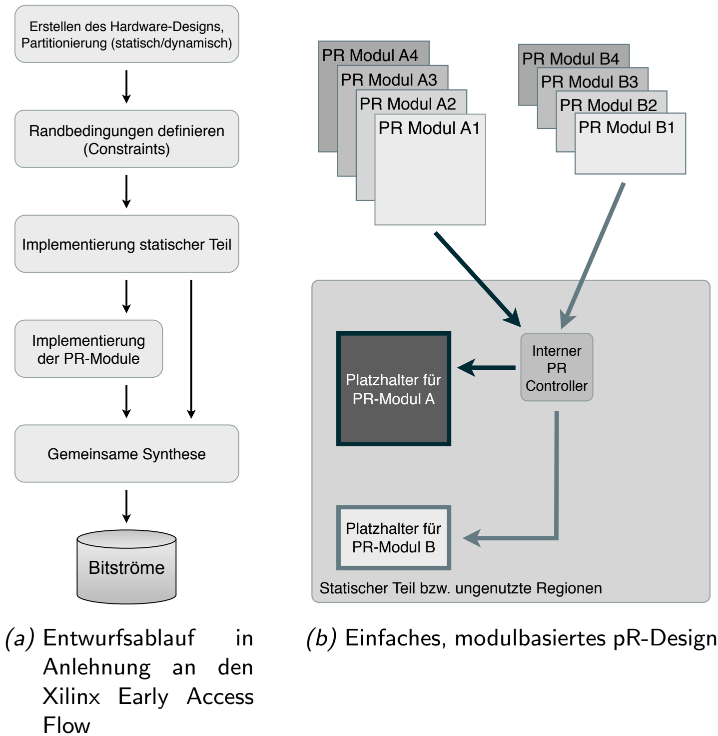
Tabelle 2.4: Schnittstellen für die (Re-) Konfiguration

vollständige Rekonfiguration	Alle Schnittstellen
pR (Xilinx)	JTAG, SelectMAP, ICAP
pR (Altera)	Passiv-Parallele Schnittstelle mit 16 Bit Datenbus oder CB
dR (Xilinx)	JTAG, SelectMAP, ICAP
dR (Altera)	*Dynamic Reconfiguration Controller*
dpR (Xilinx)	JTAG, SelectMAP, ICAP
Exo-Rekonfiguration	Alle Schnittstellen außer ICAP PR Control Block
Endo-Rekonfiguration	ICAP/PR Control Block sowie die Schnittstellen für externe Speicher

[Mes+03; McD08]. Der Funktionsumfang und die daraus erwachsenden Möglichkeiten für pR und dpR unterscheiden sich zwischen den Herstellern jedoch stark. Im wissenschaftlichen Umfeld gibt es nach Auffassung des Autors eine unverkennbare Tendenz zur Nutzung von Xilinx FPGAs.

Zusammen mit der wachsenden Zahl an FPGAs verbessern sich auch die notwendigen Tools und Toolflows, die für die pR/dpR notwendig sind (vgl. [PTW10; LP10; San+11; Dye12]). Deren Bedeutung wird klar, wenn man sich den grundsätzlichen Ablauf für die Erstellung eines Projekts mit partieller Rekonfiguration ansieht, wie es in Abbildung 2.9a auf der nächsten Seite dargestellt ist. Die Betrachtung orientiert sich am derzeitigen Designflow von Xilinx. Die Entwicklung, die die Werkzeuge von Xilinx genommen haben, lässt sich mit Hilfe von [Xil08a; Et007; Xil12d; Xil12c; Xil12b] nachvollziehen, was in den Diplomarbeiten von Orgis [Org05] und Pankalla [Pan10] noch einmal kompakt zusammengefasst wird.

Ausgehend von der Gesamtspezifikation für das rekonfigurierbare Design erfolgt im ersten Schritt die Spezifikation der statischen und dynamischen Teile sowie ihrer Schnittstellen. Der statische Teil beinhaltet alle Komponenten, die



(a) Entwurfsablauf in Anlehnung an den Xilinx Early Access Flow

(b) Einfaches, modulbasiertes pR-Design

Abbildung 2.9: pR-Entwurfsablauf und pR-Design

während der Laufzeit des FPGAs nicht verändert werden. Die dynamischen Bereiche (*PR-Regionen*) sind Platzhalter für die Komponenten, die zur Laufzeit rekonfiguriert werden können. Anschließend erfolgt der Entwurf des statischen Teils sowie der dynamischen Teile (*PR-Module*), was mehrere Synthesevorgänge beinhaltet. Das Vorgehen ähnelt dem Entwurf mehrerer statischer Designs mit gemeinsamen Teilen (im Sinne von Re-Use). Dabei müssen verschiedene Randbedingungen (engl. *Constraints*) beachtet werden, die der statische bzw. der dynamische Teil erfüllen müssen. Die so erstellten (positionierten) PR-Module eignen sich jedoch nur genau für den Platzhalter (PR-Region), für den sie synthetisiert wurden (siehe Abbildung 2.9b), was die Flexibilität der

Methodik noch immer begrenzt. Ist die PR-Region für ein Modul belegt, kann es nicht einfach in eine andere, freie Region geladen werden, auch wenn diese grundsätzlich geeignet wäre[10]. Beim derzeitigen (Xilinx-) Flow müssen Module, die parallel in verschiedenen PR-Regionen Verwendung finden, für jede Region einmal erzeugt werden. Verschiedene Forschungsarbeiten beschäftigen sich daher mit Ansätzen, positionsunabhängige PR-Modul zu erzeugen. Kettelhoit [Ket09] bezeichnet dieses Vorgehen als Modulrelozierung, häufiger findet man den englischen Begriff *Relocation* [WB04; Mei10].

2.6 Betriebssysteme

Nahezu jedes Werk mit Bezug zu eingebetteten Systemen beinhaltet eine mehr oder weniger ausführliche Darstellung zu Betriebssystemen für speziell diese Geräteklasse und deren Besonderheiten. Dem geneigten Leser werden an dieser Stelle die Werke von Marwedel, Färber, Schröder et. al. sowie Wietzke [Mar08; Fär09; SGD09; Wie12] empfohlen, die auch für diesen Abschnitt als Grundlage dienten.

Im Hinblick auf heterogene rekonfigurierbare Systeme ist das Betriebssystem neben Prozessor und programmierbarer Logikeinheit eine weitere wesentliche Komponente. Nach [Tan09] ist der Begriff wie folgt definiert:

> *Ein Betriebssystem ist eine Sammlung von Computerprogrammen, die die Systemressourcen eines Computers wie Arbeitsspeicher, Festplatten, Ein- und Ausgabegeräte verwaltet und diese Anwendungsprogrammen zur Verfügung stellt. Das Betriebssystem bildet dadurch die Schnittstelle zwischen den Hardwarekomponenten und der Anwendungssoftware des Benutzers.*

Im allgemeinen Verständnis nimmt der Begriff jedoch inzwischen eine umfassendere Bedeutung ein. Achilles [Ach05] nutzt daher eine andere Herangehensweise an die Erklärung des Begriffs. Seinem Ansatz liegt ein Schichtensystem zugrunde, dass den Betriebssystemkern, Systemmodule, Dienstprogramme und Anwendungen unterscheidet. Der Betriebssystemkern implementiert

[10] Voraussetzung sind hinreichende Größe und Anzahl Ein- und Ausgangssignale.

zentrale Aufgaben, die nicht weiter unterteilt werden können. Die Systemmodule, die eng mit dem Kernel verbunden sind, ermöglichen der Anwendungssoftware den Zugriff auf weitere Systemressourcen, wie Speicher oder Netzwerk. Somit entsprechen diese beiden Schichten der Definition nach Tanenbaum. Gleichzeitig abstrahieren Betriebssystemkern und Systemmodule spezifische Eigenschaften der Hardware. Die Klasse der Dienstprogramme umfasst die Software, die eng mit Betriebssystemkern und Systemmodulen verzahnt ist und deren Besonderheiten weiter abstrahiert, aber direkt vom Anwender genutzt werden kann.

Für den von Achilles als Dienstprogramme bezeichnete Teil der betriebssystemnahen Software werden im folgenden die Begriffe *Bibliotheksschicht* bzw. *Bibliotheken* genutzt, da es sich häufig um nur wenige ausführbare Programme, jedoch eine größere Menge an Systembibliotheken handelt. Anwendungsprogramme nutzen das durch diese Bibliotheken bereitgestellte API (Anwendungs-Programmier-Schnittstelle) für einen einfacheren Zugriff auf komplexe Funktionen des Betriebssystems. Dadurch können Anwendungsprogramme in höherem Maße unabhängig vom Betriebssystem gestaltet werden, was die Nutzung auf verschiedenen Betriebssystemen vereinfacht.

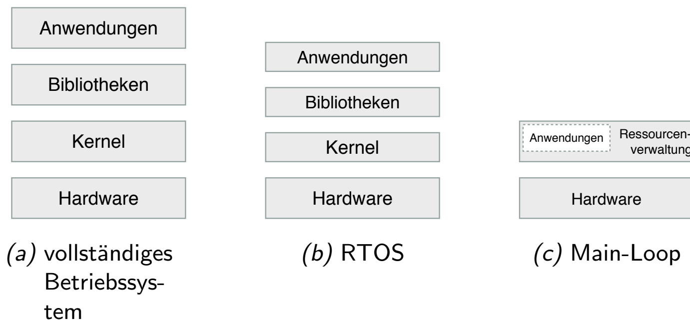
2.6.1 Betriebssysteme im Kontext eingebetteter Systeme

Einfache eingebettete Systeme kommen auch ohne Betriebssystem aus. Hier werden die zu bearbeitenden Aufgaben mit Hilfe einer zentralen Schleife (engl. *Main-Loop*) abgearbeitet, was zu einem streng sequentiellen Ablauf führt. Durch *Interrupts* (dt. Unterbrechungen) kann ein gewisser Grad der Quasi-Parallelität erreicht werden. Erst mit steigenden Ansprüchen an die Ressourcen- und Taskverwaltung, das Antwortverhalten, die Hardwareunterstützung, die Nutzerschnittstelle und die verfügbaren Anwendungen wird ein komplexes Betriebssystem notwendig. Schröder et. al. [SGD09, Seite 27] schreiben dazu:

> *... Oft kommt weiterhin noch eine Benutzerschnittstelle hinzu. Spätestens jetzt ist der Einsatz eines Betriebssystems sinnvoll; es organisiert und priorisiert den quasi-parallelen Ablauf der einzelnen Tasks in Zeitscheiben, es bietet eine Kapselung für Schnittstellen,*

> *verwaltet den Arbeitsspeicher und regelt den Zugriff auf Festspeichermedien über ein Dateisystem.*

Abbildung 2.10 stellt den Unterschied zwischen beiden Ansätzen dar. Aus der Abbildung wird auch ersichtlich, dass ein (vollständiges) Betriebssystem, bedingt durch seinen Funktionsumfang, ungleich höhere Anforderungen an Prozessor und Speicherausbau eines Systems stellt. Eine im Umfeld eingebetteter Systeme oft genutzte Alternative sind RTOSs. Diese Systeme unterliegen einem gewissen Determinismus[11] und erlauben daher die Bearbeitung von Aufgaben innerhalb vorgegebener zeitlicher Schranken. Dieser Umstand trifft auf eine Vielzahl eingebetteter Systeme zu, die auf bestimmte Ereignisse in einer festgelegten Zeitspanne reagieren müssen. Traditionell sind RTOS trotz vergleichbarer Grundarchitektur kompakter als vollständige Betriebssysteme und daher auch für eingebettete Systeme mit eingeschränkten Ressourcen einsetzbar, unabhängig von deren zeitlichen Anforderungen.



(a) vollständiges Betriebssystem *(b)* RTOS *(c)* Main-Loop

Abbildung 2.10: Schichtenmodelle zum Vergleich der Ansätze bei verschiedenen Betriebssystemen

2.6.2 Firmware

Im bisherigen Text wurde bereits mehrfach der Begriff *Firmware* verwendet, ohne dessen Bedeutung zu erläutern. Der IEEE-Standard 1275-1994 [94] definiert Firmware als die ROM-basierte Software, die einen Computer in der

[11] Sogenannte harte und weiche Echtzeitbedingungen, siehe z. B. [Mar08, Kapitel 4 ab Seite 135].

Zeit zwischen dem Einschalten und dem Start des primären Betriebssystems kontrolliert.

Im weiteren Sinne, und besonders in Verbindung mit eingebetteten Systemen, wird die gesamte, im nichtflüchtigen Speicher abgelegte Software als Firmware bezeichnet. In diesem Fall kann es sich also um ein vollständiges Betriebssystem, ein RTOS oder „Main-Loop"-Software handeln. Der Begriff findet auch Anwendung, wenn der vom Endnutzer nicht veränderbare Teil der Systemsoftware adressiert wird – beispielsweise im Kontext von Mobiltelefonen oder Tablet-PCs. Hier kann der Nutzer zusätzliche Software[12] im Flash-Speicher hinterlegen, während die restlichen Softwareteile nicht zugänglich sind.

In dieser Arbeit wir der Begriff *Firmware* bewusst an Stellen genutzt, wo er in seiner unscharfen Bedeutung auch im allgemeinen Sprachgebrauch Anwendung findet und mit seiner Benutzung kein Bedeutungsverlust einhergeht.

2.6.3 Anforderungen an den Softwareentwurf

Software hat inzwischen einen wesentlichen Anteil am Gesamtsystem und dient nicht selten als zusätzliches Funktionsmerkmal. Daher nimmt die Softwaretechnik (engl. *software engineering*) – also das ingenieurmäßige, sorgfältig geplante, gesteuerte Vorgehen beim Softwareentwurf – einen zentralen Platz beim Entwurf eingebetteter Systeme ein. Der Entwurfsprozess von Softwaresystemen lässt sich bei hinreichender Abstraktion wieder im Rahmen des V-Modells abbilden (Abbildung 2.11 auf der nächsten Seite). Die Methoden und Werkzeuge für den Softwareentwurf werden in der Fachliteratur unter dem Begriff *Computer-Aided Software Engineering (CASE)* [SN92; Bal00] zusammengefasst.

Wie bei digitalen Hardwaresystemen sind Test und Verifikation von Softwaresystemen in der Regel nicht vollständig und der Nachweis von Fehlerfreiheit nicht durchführbar [HT10]. Daher sind die korrekte Erfassung und Modellierung der Anforderungen von besonderer Bedeutung. Im Softwareentwurf dominieren objektorientierte Ansätze. In der Spezifikationsphase werden Methoden der objektorientierten Programmierung wie Objekte, Attribute, Operationen, Assoziation und Aggregation für die *objektorientierte Analyse (OOA)*

[12] Meist als App (vom engl. Application) bezeichnet.

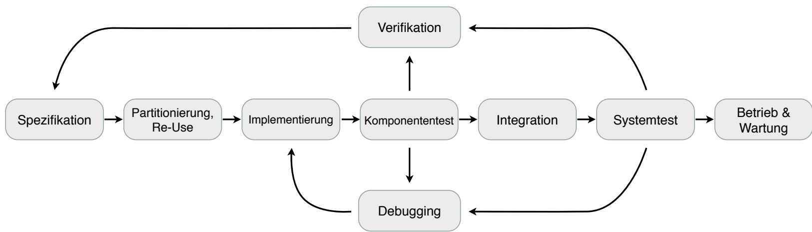


Abbildung 2.11: Entwurfsfluss bei der Softwareentwicklung

genutzt. Mit Hilfe des *objektorientierten Designs (OOD)*, dt. *objektorientierter Entwurf*, lässt sich ein OOA-Modell unter technischen Gesichtspunkten verfeinern, sodass im Anschluss eine Implementierung möglich ist. Beide Konzepte finden bei der Anforderungsanalyse, der Spezifikation und der Partitionierung Anwendung. Weiterhin wird im Softwareentwurf explizit zwischen der Datenhaltung, der (Benutzer-) Schnittstelle und der Verteilung von Anwendungen auf verschiedene Systeme unterschieden. Auch bei Software ist die Wiederverwendung vorhandener Funktionalität („Re-Use") ein bewährtes Konzept mit verschiedenen Ausprägungen von kleineren Quellcodesequenzen bis hin zu komplexen Klassenbibliotheken. Nach der Implementierung der einzelnen Komponenten erfolgt deren Test, wobei das Finden und Beseitigen von Fehlern als *Debuggen* oder *Debugging* bezeichnet wird. Der formale Test von Software ist zwar möglich [HT10], beschränkt sich jedoch auf sehr kleine Codesequenzen. Verbreitete Methoden sind die statische Codeanalyse, bei der Werkzeuge den Quellcode analysieren und nach Fehlern durchsuchen, und sogenannte *Unittests*, bei denen Softwaremodule mit klar definierten Schnittstellen überprüft werden. Dazu erfolgt die Stimulation des Moduls und anschließend der Vergleich der Ergebnisse mit dem erwarteten Verhalten. Einen Überblick über geeignete Werkzeuge gibt beispielsweise Luithardt [LGS13]. Zusammen mit dem Test kann auch die Verifikation der Komponenten erfolgen. Anschließend folgen Integration und Systemtest.

Im Gegensatz zu Leiterplatten und nicht programmierbaren digitalen Schaltkreisen ist die Veränderung von Software auch nach der Auslieferung noch möglich, was man in zwei Kategorien teilen kann (vgl. [SN92]):

- Aktualisierung und
- Reparatur.

Die Aktualisierung umfasst die Erweiterung der Funktionalität. Das geschieht entweder zur Anpassung oder Erweiterung der Systemfunktionen (*adaptive maintenance*) oder zu Verbesserung von Stabilität, Geschwindigkeit und Wartbarkeit (*perfective maintenance*). Bei der Reparatur hingegen werden Fehler in der Implementierung beseitigt (*corrective maintenance*).

2.7 Zusammenfassung und Diskussion

Das Kapitel führt in die für den Aufbau eingebetteter rekonfigurierbarer Systeme relevanten Forschungsfelder – eingebettete Systeme, Betriebssysteme, Programmable Logic Devices, mit Schwerpunkt FPGAs und run-time Reconfiguration – ein. Problematisch bei der zusammenfassenden Betrachtung ist die indifferente Begriffswelt innerhalb wie auch zwischen den einzelnen Domänen, die von unscharfen Begriffsdefinitionen und Mehrdeutigkeiten geprägt ist. In den überlappenden Bereichen der einzelnen Gebiete führt dies auch in Fachartikeln zu bisweilen schwer verständlichen oder gar inkorrekten Interpretationen bestimmter Sachverhalte. Daher erfolgt im gesamten Kapitel die Definition von Begriffen, wie sie im Rahmen dieser Arbeit verstanden und verwendet werden. Weiterhin fasst das Kapitel die verschiedenen Teilgebiete im Hinblick auf den Systementwurf und damit verbundene Entwurfsabläufe zusammen.

Mit dem V-Modell existiert eine allgemein anerkannte Methodik auf höchstem Abstraktionsniveau. Dennoch kommt der Autor nach der Betrachtung bekannter und allgemein anerkannter Vorgehensmodelle für den Systementwurf zu dem Schluss, dass es derzeit kein geschlossenes Modell oder Werkzeug für den durchgängigen Entwurf komplexer, inhomogener Systeme von der Spezifikation zum Produkt gibt. Vielmehr ist es notwendig, für jede Teilkomponente eines Systems einer ihr angemessenen Entwurfsmethodik zu folgen und diese mit dazu geeigneten Werkzeugen umzusetzen.

Daher erörtert der Autor in Anlehnung an die genutzte Fachliteratur und ergänzend zum dort dargestellten Vorgehen für jeden Teilaspekt einen typi-

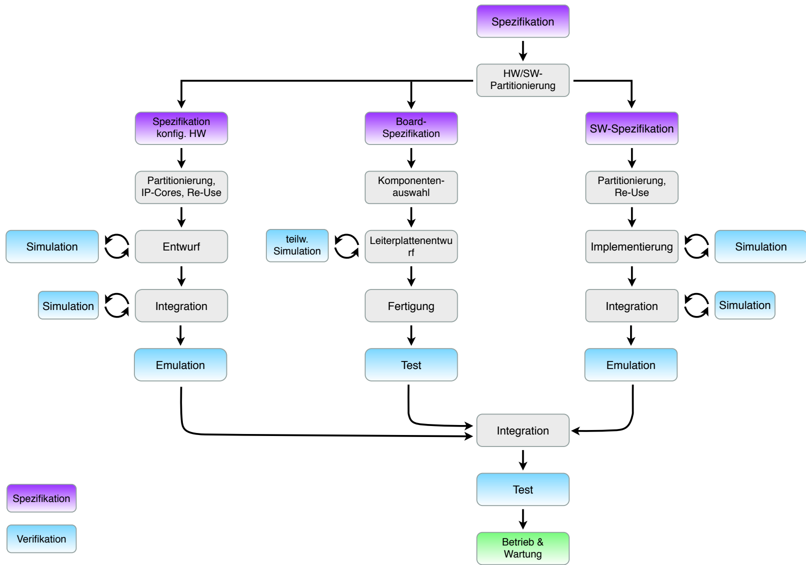


Abbildung 2.12: Zusammenfassung der einzelnen Entwurfsabläufe

schen Entwurfsablauf, der domänenspezifische Aspekte einbezieht. Aus diesem Ansatz heraus ergibt sich die in Abbildung 2.12 dargestellte Vorgehensweise für den Entwurfsfluss bei eingebetteten heterogenen rekonfigurierbaren Systemen. Der erste Schritt besteht dabei in der Überführung der (informellen) Anforderungsbeschreibung in eine konsistente und möglichst formale bzw. formalisierte Spezifikation des Gesamtsystems[13]. Der Begriff *Gesamtsystem* meint hier ein eingebettetes System bestehend aus Hardware (inklusive *rekonfigurierbarem Modul (RM)*), Betriebssystem und Anwendungssoftware. Eine ausführbare Spezifikation als Systemmodell (z. B. mit SystemC) erscheint in dieser Phase nicht realisierbar, da die Ein- und Ausgabeparameter ebenso wie die Übergangsfunktion zu komplex ausfallen. Der nächste Schritt ist daher die Partitionierung des Systems in Hardware und Software. Die Aufteilung erfolgt in den meisten Fällen aufgrund von Erfahrungswerten der beauftragten Entwerfer oder durch Vorgaben, die sich aus der Spezifikation ergeben. Ba-

[13] z. B. in Form eines Pflichtenheftes

sierend auf dieser grundsätzlichen Entwurfsentscheidung können nun die bekannten Abläufe für die drei Domänen Leiterplattenentwurf, Softwareentwurf und Entwurf dedizierter Hardwarekomponenten genutzt werden. Die Bearbeitung erfolgt dabei weitgehend parallel. Software und Module der digitalen Hardware können durch Simulation getestet und verifiziert werden. Bereits vor der Fertigstellung der endgültigen Hardwareplattform (bestückte Leiterplatte) sind mit Hilfe von geeigneten (oft kommerziell verfügbaren) Entwicklungsboards auch Emulationen möglich. Abschließend erfolgen Integration, Test und soweit möglich Verifikation des Gesamtsystems. Diese können sich, aufgrund der Möglichkeit rekonfigurierbare Logik und Software auch nach der Auslieferung des Systems noch zu verändern, über den gesamten Lebenszyklus erstrecken. Dieser (nicht unumstrittene Ansatz) wird besonders bei unkritischen Anwendungen gern genutzt.

Wie für alle anderen Entwurfsmodelle gilt auch hier, dass der dargestellte direkte Ablauf (Top-Down) idealisiert ist. Wie sich in den weiteren Kapiteln zeigen wird, erfolgen viele Entwurfsschritte itterativ und der gesamte Entwurfsablauf folgt einem „Top-Down geprägten Jo-Jo-Ansatz“.

3 Vertiefung der Systemarchitekturen heterogener rekonfigurierbarer Systeme

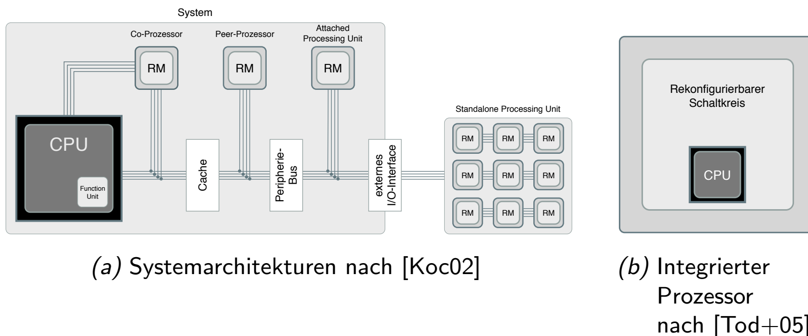
Bereits in Abschnitt 1.2 wurde auf das Potential rekonfigurierbarer Hardwaresysteme hingewiesen. Kapitel 2 erarbeitete darauf aufbauend notwendige Grundlagen für den Entwurf solcher Systeme. In diesem Kapitel werden nun heterogene rekonfigurierbare Systeme eingeführt, erläutert und diskutiert. Schwerpunkt der Einführung sind Systemarchitekturen und Entwurfskonzepte. Dabei werden aufbauend auf dem Stand der Technik neue Betrachtungsweisen eingeführt, die das Systemverständnis weiter vertiefen.

3.1 Begriffserklärung und Systemarchitekturen

Wie bereits mit dem Entwurfsfluss in Abschnitt 2.7 angedeutet, erweitern rekonfigurierbare Schaltkreise die Möglichkeiten für die Architektur und den Einsatz von Rechnersystemen. Die Kombination aus *General Purpose Processor (GPP)* und programmierbarer Logik erlaubt im Idealfall die Nutzung der Vorteile beider Architekturen. Es hat sich jedoch gezeigt, dass rekonfigurierbare Module besonders für die Anwendung vieler Berechnungen auf kleinen Datenmengen oder für einfache Berechnungen auf großen Datenmengen geeignet sind [BAK96]. Für die Umsetzung des Kontrollflusses eines Algorithmus eignen sich konventionelle Prozessoren besser. Daher findet man häufig die Kombination eines Prozessors und eines RMs, beispielsweise eines FPGAs.

Diese Klasse von Systemen wird auch als *hybride Hardwaresysteme* oder *heterogene Hardwaresysteme* bezeichnet. Für diese Arbeit wird der Begriff heterogene rekonfigurierbare Systeme (hrS) genutzt. (siehe [Koc02])

Allgemein besteht ein heterogenes rekonfigurierbares System aus einem oder mehreren Prozessoren, einem oder mehreren RMs[1] sowie allen weiteren benötigten Komponenten zum Aufbau eines Systems (vgl. Abschnitt 2.3.1). Oftmals erfolgt eine Unterscheidung der Systeme nach dem Grad der Kopplung zwischen RM und Prozessor. Die entstehenden Systemarchitekturen ähneln daher solchen aus dem Bereich der parallelen Rechnerarchitekturen (vgl. [Ben+08]). Eine detaillierte Darstellung möglicher Architekturkonzepte erfolgte durch K. Compton und S. Hauck [CH02]. Diese Darstellung wurde durch T. J. Todman erweitert [Tod+05]. Im deutschsprachigen Raum hat sich A. Koch mit dem Gebiet auseinandergesetzt [Koc04]. Den Ausführungen ist gemein, dass sie sich weitgehend an der Erweiterung von Workstations oder PCs orientieren und Werkzeuge zur Erstellung von Bitströmen für FPGAs diskutieren. Betrachtungen mit Bezug zu eingebetteten Systemen erfolgen daher in Abschnitt 3.3. Zur Einführung in die Thematik werden die in Abbildung 3.1 dargestellten Architekturen nach [CH02; Tod+05; Koc04] im Folgenden erläutert und diskutiert.



(a) Systemarchitekturen nach [Koc02]

(b) Integrierter Prozessor nach [Tod+05]

Abbildung 3.1: Systemarchitekturen für heterogene rekonfigurierbare Systeme

[1] Geeignet ist jedes reversibel programmierbare PLD. Häufig kommen aber FPGAs zum Einsatz.

Architektur 1 Die loseste Form der Kopplung ist ein extern an das System angebundenes (re-) konfigurierbares Modul. Diese Module sind in ihrer Ausführung meist eigenständige Systeme mit eigener Peripherie (RAM, Display, I/O-Ports), Stromversorgung und Schnittstellen zur Umwelt und können nahezu beliebige Ausmaße (mehrere FPGAs) haben. Ihr Einsatz umfasst beispielsweise Prototyping- und Emulationsaufgaben. Für die direkte Unterstützung von Prozessoren bei Berechnungen sind sie hingegen nicht konzipiert. Entsprechend gering ist die Leistungsfähigkeit der Kommunikationsverbindung mit dem Hostsystem (z. B. JTAG, RS232, Ethernet). Eine häufige Rekonfiguration durch die auf dem Prozessor ausgeführte Software ist nicht vorgesehen bzw. sinnvoll. Ein typisches Beispiel für diese Form der Architektur sind die Referenzboards der verschiedenen FPGA-Hersteller.

Architektur 2 Eine weit verbreitete Form der Kopplung ist die Nutzung von Peripheriebussen für die Anbindung von RMs. Ihre weite Verbreitung verdankt diese Architektur der guten Balance von mechanischen und technischen Vor- und Nachteilen. Vorteile sind die technischen Parameter der Bussysteme (Bandbreite, Latenz), die weitgehende Standardisierung der Schnittstellen und der damit verbundenen Bauteile sowie die Verfügbarkeit von IP-Cores und Software für die Kommunikation. Moderne FPGAs bieten zusätzlich Unterstützung für gängige Peripheriebusse (z. B. PCI, PCIe) in Form von dedizierten Hardwareblöcken. Die Kapazität der RMs ist im wesentlichen abhängig von mechanischen und thermischen Gegebenheiten wie Platzangebot und Abwärmetransport. Wesentlicher Nachteil dieser Form der Kopplung sind die im Vergleich zu den nachfolgend beschriebenen Formen begrenzten Datenraten und die Latenz der Bussysteme. Daher eignen sich *Attached Processing Units* für Aufgaben, die der Datenrate des verwendeten Peripheriebusses und der Latenz angepasst sind. Am Markt gibt es verschiedene Lösungen in Form von Steckkarten für PCI, PCIe oder auch *Universal Serial Bus (USB)*.

Architektur 3 Bei der nächsten Architekturstufe wird das RM als weiterer Prozessor betrachtet. Das entstehende System gleicht in seiner Architektur einem Multiprozessorsystem, wobei Prozessor und RM parallel an möglichst unabhängigen Aufgaben arbeiten. Die Kommunikation mit weiteren Prozessoren oder RMs erfolgt über Systembusse, deren Datenrate meist höher ist als

die der Peripheriebusse, bei gleichzeitig geringerer Latenz. Die aktuelle Entwicklung auf dem Gebiet der Multiprozessorsysteme begünstigt diese Architektur weiter. Prozessoren verfügen über dedizierte Schnittstellen zu Nachbarprozessoren[2], die eine schnelle und effiziente Kommunikation sowie hohe Datenraten in Multiprozessorsystemen erlauben. Aktuelle FPGAs unterstützen einige dieser dedizierten Peripheriebusse in Hardware [Mac07] oder als IP-Core[3]. Somit kann diese Architektur bereits als eng gekoppelt betrachtet werden. Die Kapazität des RMs ist üblicherweise auf einen Schaltkreis beschränkt, obwohl auch komplexere Architekturen realisierbar sind.

Architektur 4 Eine weitere Annäherung des RMs an die CPU wird erreicht, wenn das RM als Co-Prozessor ausgeführt ist, wobei Prozessorkern und RM über einen dedizierten Kanal (z. B. einen Prozessorbus) kommunizieren. In diesem Fall bietet sich die Integration auf einem Modul oder in einem SoC an [Put09]. Der Co-Prozessor übernimmt komplexe Aufgaben, die er aufgrund von Spezialisierung effizienter abarbeiten kann als die CPU. Durch diese Selbstständigkeit verringert sich der notwendige Kommunikationsaufwand, da der Prozessor Daten an das RM übergibt und erst nach Ende der Bearbeitung über das Ergebnis informiert wird. Nach [Koc02] kommunizieren CPU und Co-Prozessor über einen gemeinsamen Speicher. Um Inkonsistenzen ohne aufwändige Synchronisation zu vermeiden, nutzen beide Einheiten den Speicher quasi-exklusiv.

Architektur 5 Die engste Form der Kopplung stellen Systeme dar, bei denen das RM direkt in die CPU integriert ist. Anwendung findet diese Architektur, wenn beispielsweise einzelne Befehle zur schnelleren Abarbeitung in Hardware ausgeführt werden und diese auch im Nachhinein veränderbar bleiben sollen. Das RM ist als weitere *arithmetisch-logische Einheit (ALU)* direkt in den Datenpfad des Prozessors integriert. Der Datenaustausch erfolgt über interne Register. Daher ist die Kommunikation zwischen Prozessor und RM sehr schnell, aber auf wenige Byte begrenzt. Die physische Größe des RMs unterliegt ebenfalls starken Beschränkungen.

[2] z. B. AMD HyperTransport und Intel QuickPath Interconnect

[3] z. B. Xilinx QuickPath Interconnect IP

Architektur 6 Durch die steigende Integrationsdichte und die daraus folgende Verfügbarkeit von mehr Funktionalität pro Chip-Fläche gewann in den letzten Jahren eine weitere Architektur enorm an Bedeutung. Hier ist der Prozessorkern direkt in die rekonfigurierbare Logik des RMs integriert, entweder als Hardmakro oder als Softcore. Die Verbindung mit der ihn umgebenden rekonfigurierbaren Logik erfolgt über Prozessorbusse mit geringer Latenz. Je nach Ausführung wird ein Teil des FPGAs für den Prozessor und seine Peripherie verwendet, was die nutzbare Menge an Logikgattern beeinflusst. Diese Architektur wird in der Literatur auch als *Programmable System-on-Chip (PSoC)* bezeichnet. PSoCs ähneln der 4. Architektur, da sich innerhalb des RM weitere rekonfigurierbare Funktionseinheiten umsetzen lassen. Bekannte Architekturen sind die Virtex-FPGAs (II Pro, 4 und 5) von Xilinx [Xil08i; Xil08f; Xil08h] mit integrierten *PowerPC*-Kernen, die Zynq-Serie von Xilinx mit ARM-Kernen [Xil12f] sowie Umsetzungen mit den Soft-Cores *MicroBlaze* [Xil08b], *NIOS II* [Alt10] oder *Leon* [Pro].

Verteilte Architekturen Diese Art der Architektur findet in der klassischen Betrachtung nach [CH02; Tod+05; Koc04] keine Berücksichtigung. Ausgehend von der Ähnlichkeit der soeben vorgestellten Systemarchitekturen mit den Konzepten bei parallelen Rechnersystemen ist es naheliegend, auch Ansätzen verteilter Rechnersysteme zu betrachten. In der vorliegenden Arbeit fanden diese Konzepte jedoch keine Anwendung, da sie für eingebettete Systeme derzeit noch nicht von Interesse sind. Daher werden sie an dieser Stelle nicht weiter betrachtet und lediglich der Vollständigkeit halber genannt.

3.2 Vertiefung der Systemarchitekturen

Bei den Systemlevel-Architekturen nach [CH02; Tod+05; Koc04] wird keine explizite Unterscheidung nach Konfiguration, Kommunikation oder Datenfluss vorgenommen. Kommunikationskonzepte und deren architektonische Umsetzung verschwinden durch die Abstraktion komplett. Auch eine Trennung zwischen Kommunikation und Konfiguration erfolgt nicht. Aus Sicht des Entwurfsflusses (vgl. Abschnitt 2.7) sind diese Fragen jedoch bereits in der Spezifikationsphase zu berücksichtigen, da sie Einfluss auf verschiedene nachfolgende Entwurfsphasen in den drei Domänen haben. Daher erfolgt an dieser

Stelle eine vertiefende Diskussion der Architekturen in Hinblick auf Kommunikation, Datenaustausch und Konfiguration.

3.2.1 Kommunikationsarchitekturen

Position und Aufgabe des RM im Datenpfad der Zielapplikation beeinflussen direkt die Kommunikationsarchitektur zwischen RM und Prozessor. Dabei ist in erster Linie entscheidend, dass der notwendige Datenaustausch in der dafür vorgesehenen Zeit erfolgen kann. Maßgeblich dafür sind die zu übertragende Datenmenge und die verfügbare Bandbreite zwischen Hostprozessor und RM. Müssen beide lediglich einfache Statusinformationen und Konfigurationsdaten austauschen, ist eine schmalbandige Verbindung meist ausreichend, was nach Abschnitt 3.1 Architekturen mit loser Kopplung entspricht. Eng gekoppelte Systeme verfügen hingegen über Verbindungen mit höherer Bandbreite und geringer Latenz, was eine schnellere Datenübertragung ermöglicht.

Abbildung 3.2 auf der nächsten Seite stellt die möglichen Kommunikationsarchitekturen dar. Zum einen kann das RM einen Datenstrom verarbeiten, während es mit dem Hostprozessor nur Status- und Konfigurationsdaten austauscht (Abbildung 3.2, links). Dies entspricht der expliziten Aufteilung von Daten- und Kontrollfluss nach [BAK96]. Es erlaubt eine minimale Auslegung der Verbindung (Anzahl der Leitungen), wenn diese für Konfigurations- und Statusinformationen verwendet werden kann. In allen anderen Fällen tauschen RM und Hostprozessor Daten mit bestimmten Anforderungen an die Datenrate oder die Latenz aus. Wie aus dem rechten Teil von Abbildung 3.2 auf der nächsten Seite ersichtlich, ergeben sich dann fünf verschiedene Architekturen:

- Datenein- und -ausgang über den Hostprozessor,
- Datenein- und -ausgang über das RM,
- Dateneingang über RM und Datenausgang über den Hostprozessor,
- Dateneingang über Hostprozessor und Datenausgang über das RM sowie
- Datenein- und -ausgang über Hostprozessor und RM.

Auch in diesen Fällen ist die parallele Nutzung der Kommunikationsverbindung für Status- und Konfigurationsdaten möglich (vgl. Umsetzung in Abschnitt 6.4). In der Mehrzahl der Fälle wird es jedoch dedizierte Verbindungen für Status- und Konfigurationsdaten geben, da die kombinierte Nutzung einen zusätzlichen Grad der Komplexität bei Leiterplatten-, Hardware- und Softwareentwurf darstellt.

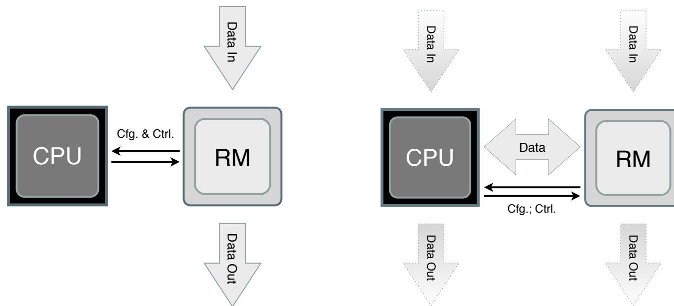
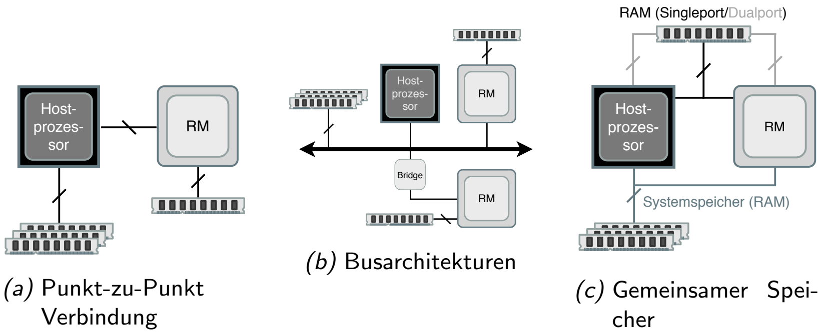


Abbildung 3.2: Position und Aufgabe des RM im Datenpfad

3.2.2 Datenaustausch

Ausgehend von den Systemarchitekturen und den soeben aufgezeigten Kommunikationsarchitekturen erfolgt nun die Diskussion einiger Konzepte zum Datenaustausch zwischen Hostprozessor und RM. Aufgrund der Verwandtschaft der heterogenen rekonfigurierbaren Systeme mit parallelen und verteilten Rechnersystemen lassen sich die Ansätze aus diesem Bereich (siehe z. B. [Ben+08]) auch auf hrS übertragen. Die möglichen Architekturen werden durch den jeweiligen Grad der Kopplung beeinflusst. Grundsätzlich sind jedoch die drei in Abbildung 3.3 auf der nächsten Seite dargestellten Konzepte von Bedeutung:

- der Datenaustausch über eine Punkt-zu-Punkt Verbindung,
- der Datenaustausch über eine Busarchitektur und
- der Datenaustausch über gemeinsamen Speicher.



(a) Punkt-zu-Punkt Verbindung

(b) Busarchitekturen

(c) Gemeinsamer Speicher

Abbildung 3.3: Konzepte zum Datenaustausch zwischen Hostprozessor und RM

Jedes Konzept erfordert die Synchronisation von Hostprozessor und RM. Dafür werden in [BB04] drei Varianten vorgeschlagen. Zum einen kann die Bearbeitung im RM für die Software auf dem Hostprozessor als blockierendes Lesen (blocking read) erscheinen. Dann ist der Prozess jedoch für die Ausführungszeit des RMs angehalten. Die zweite Möglichkeit sind Soft- oder Hardwarepolling, wobei letzteres nicht immer zur Verfügung steht. Beim Softwarepolling, z. B. realisiert durch Semaphoren oder Locks, werden zusätzliche Bearbeitungszyklen für das Prüfen der Sperren verwendet. Bei der dritten Variante löst das RM nach Abschluss der Bearbeitung einen Interrupt aus. Das kann bei sehr kurzen Bearbeitungszeiten im RM zu einer hohen Interrupt-Last auf dem Hostprozessor führen.

Punkt-zu-Punkt Verbindung Bei einer direkten Verbindung von Hostprozessor und RM (siehe Abbildung 3.3a) können die Art der Verbindung sowie das Kommunikationsprotokoll und damit der benötigte Aufwand beliebig variiert werden. Dem Vorteil der hohen Flexibilität stehen der gegebenenfalls notwendige Aufwand bei der Implementierung und hardwareseitigen Umsetzung gegenüber. Es muss ein für den Hostprozessor (z. B. in Software) ebenso wie für dass RM (z. B. als IP-Core) geeignetes Kommunikationsprotokoll implementiert werden. Zudem ist auf der Leiterplatte der Platzbedarf für die notwendigen Verbindungen zu berücksichtigen. Der Transfer von Daten aus dem

Systemspeicher führt immer über den Hostprozessor. Dies muss nicht unbedingt von Nachteil sein, da der Hostprozessor wesentlich flexibler auf verteilte oder irreguläre Datenstrukturen im Speicher zugreifen kann, als die programmierbare Logikeinheit. Allerdings sollte das RM für speicherintensive Berechnungen über einen eigenen RAM verfügen.

Busarchitekturen Die Verwendung einer Busarchitektur zur direkten Kommunikation zwischen Hostprozessor und RM ist sehr häufig anzutreffen (vgl. Abbildung 3.3b auf der vorherigen Seite). Sie betrifft lose gekoppelte Systeme, bei denen ein Peripheriebus verwendet wird, ebenso wie eng gekoppelte und integrierte Architekturen, bei denen der Datenaustausch über einen Prozessorbus erfolgt. In jedem Fall muss im RM die benötigte Logik zur Umsetzung des auf dem Bus verwendeten Kommunikationsprotokolls implementiert sein, was zusätzliche Ressourcen erfordert. In vielen Fällen sind diese Logikteile als IP-Core oder Hardmakro verfügbar, so das der zusätzliche Aufwand bei der Implementierung überschaubar ist. Bei lose gekoppelten Architekturen hat das RM keinen Zugang zum Systemspeicher. Außerdem ist die Kommunikation zwischen Hostprozessor und RM aufgrund geringer Datenraten und hoher Latenz teuer. Daher sollte für datenintensive Anwendungen zusätzlicher, direkt durch das RM erreichbarer Speicher zur Verfügung stehen. Kommunizieren RM und Hostprozessor über einen Systembus direkt miteinander, empfiehlt sich auch hier ein zusätzlicher Speicher, wenn das RM größere Datenmengen verarbeiten muss.

Gemeinsamer Speicher Das dritte Konzept sieht den Austausch von Daten über einen gemeinsam genutzten Speicher vor (Abbildung 3.3c auf der vorherigen Seite). Dieser Speicher kann ebenso der Systemspeicher wie ein exklusiver Speicher sein. In jedem Fall haben RM und Hostprozessor Zugriff auf den gemeinsamen Speicher und tauschen Daten über diesen Weg aus. Greifen die Kommunikationspartner über einen Bus auf den Speicher zu, entspricht dies[4] grundsätzlich der soeben erläuterten Busarchitektur. Als Alternative bietet sich ein Speicher mit getrennten Ports für Hostprozessor und RM an, was allerdings aus Sicht der Leiterplatte einen erheblichen Mehraufwand bedeutet. Gemeinsamer Speicher erlaubt auf effiziente Weise den Austausch auch

[4] Ausgehend von den Anforderungen an die Hardware

großer Datenmengen. Diesem augenscheinlichen Vorteil stehen jedoch diverse Nachteile entgegen. Der Speicherinhalt sollte eine reguläre Struktur aufweisen, die sich aus dem RM heraus einfach nutzen lässt. Außerdem muss die Kohärenz und Konsistenz des Speicherinhaltes sichergestellt werden, was einen zusätzlichen Aufwand bei der Implementierung nach sich zieht. Das Kohärenzproblem bei der gemeinsamen Nutzung des Systemspeichers entsteht aus der Tatsache heraus, dass moderne Prozessoren meist über integrierte Caches verfügen, die Teile des RAMs aus Geschwindigkeitsgründen zwischenspeichern. Wie bei Multiprozessorarchitekturen erfordert die Synchronisation der Caches mit dem Speicherinhalt zusätzlichen Aufwand [Ben+08]. Ein exklusiv zum Datenaustausch genutzter Speicher wird meist ohne zusätzlichen Cache angebunden. Trotzdem muss eine Zugriffssicherung erfolgen, um gleichzeitiges Lesen/Schreiben zu verhindern. Dies kann in Hardware erfolgen, etwa bei Dualport-RAM. Alternativen sind bekannte Synchronisationsverfahren wie Locks, Semaphoren oder Monitore.

Eine qualitative Analyse einiger der vorgestellten Architekturen präsentieren Kalte et. al. [KPR04]. Die Autoren versuchen, die Auswirkungen der Kopplung zwischen Prozessor und RM auf die Leistungsfähigkeit des Gesamtsystems zu analysieren. Dazu modellieren sie die Kommunikation für verschiedene Einbettungsvarianten. Im Speziellen werden Modelle für

- die Datenpfad-Kommunikationskosten,
- die CPU-Bus-Kommunikationskosten und
- die PCI-Bus-Kommunikationskosten

vorgestellt. Die Ergebnisse der Analyse von Kalte et. al. entsprechen den Erwartungen aus den Architekturbeschreibungen und untermauern somit die dargelegten Varianten.

3.2.3 Konfigurationsarchitekturen

Wie die Kommunikation in Abschnitt 3.2.2 erfordern auch die Konfigurationsstrategien bereits in frühen Entwurfsphasen Berücksichtigung, da sie durchaus einen signifikanten Anteil an den Ressourcen eines Systems benötigen. Dabei sind vier Punkte von Bedeutung:

- der Speicherort der Konfigurationen,
- die Erreichbarkeit der Konfigurationsspeicher,
- die Art der (Re-) Konfiguration und
- die gewünschte Konfigurationsgeschwindigkeit.

Alle der in Abschnitt 3.1 beschriebenen Architekturen können die Rekonfiguration des RMs unterstützen. Ausgehend von der Begriffsdefinition in Abschnitt 2.5.1 fallen die Architekturvarianten eins bis fünf in die Klasse der dynamisch-partiell rekonfigurierbaren Systeme, da der Hostprozessor (und damit das System) auch während der Rekonfiguration verfügbar ist. Bei Variante sechs kann das gesamte RM durch vollständige Rekonfiguration angepasst werden, während durch dynamisch-partielle Rekonfiguration auch Teile des RMs im Betrieb veränderbar sind. Aufgrund dessen werden die Formen der RTR bei hrS immer aus Sicht des RMs benannt.[5]

Abbildung 3.4 auf der nächsten Seite stellt einige grundsätzliche Beziehungen zwischen Hostprozessor, RM und Konfigurationsspeicher dar und illustriert die zwei Kernpunkte für die Konfigurationsarchitektur:

- Woher bezieht das RM seine Folgekonfigurationen und
- wie wird eine neue Konfiguration zum System übertragen?

Im Bild wird zwischen Konfigurationen unterschieden, die von außen in das System gebracht werden (*neue Konfiguration*) und solchen, die bereits im System gespeichert sind. Ein Sonderfall, der jedoch ebenfalls mit abgedeckt ist, ergibt sich dann, wenn neue Konfigurationen durch Software erzeugt werden, die auf dem Hostprozessor ausgeführt wird [Ket09]. Diese kommen nicht von außen in das System, müssen aber ggf. gespeichert werden.

Ein System kann entweder über einen separaten Speicher für Konfigurationen verfügen, oder aber ein anderes, verfügbares Speichermedium nutzen. Ersteres bedeutet einen zusätzlichen Ressourcenbedarf (Speicher-IC, Leiterbahnen, Pins an den ICs), bietet aber eventuell aus strategischer oder architektonischer Sicht einen Vorteil oder ist aufgrund der Systemarchitektur unumgänglich. Die parallele Verwendung vorhandener Ressourcen erscheint als effizienteste

[5] Somit steht beispielsweise vollständige Rekonfiguration dafür, dass das RM gestoppt, rekonfiguriert und gestartet wird.

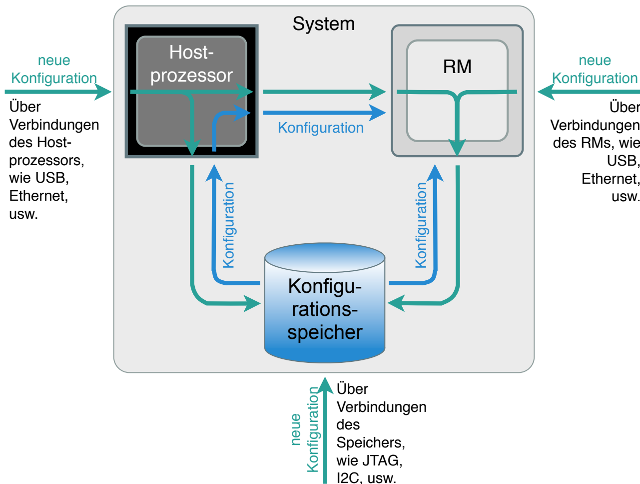


Abbildung 3.4: Grundsätzliche Beziehung zwischen Hostprozessor, RM und Konfigurationsspeicher

Lösung für die permanente Speicherung der Konfiguration im System, muss aber durch die Hard- und Softwarearchitektur unterstützt werden. Ist eine permanente Speicherung nicht notwendig, stellen externe Medien, die beispielsweise über eine Netzwerkverbindung oder Peripheriebusse erreichbar sind, eine Alternative dar. In diesem Fall erscheint eine Konfiguration aus Sicht des Systems immer als neue Konfiguration.

In Abbildung 3.4 ist zu erkennen, dass das RM entweder direkten Zugriff auf den Konfigurationsspeicher hat oder die Konfiguration über den Hostprozessor an das RM übertragen wird (blaue Verbindungen). Der Hostprozessor kann die Konfiguration aus einem lokalen Speicher beziehen oder sie generieren. Außerdem ist es möglich, neue Konfigurationen von außerhalb des Systems direkt oder unter Verwendung des Hostprozessors in das RM zu spielen.

Im Bild türkis dargestellt sind die Verbindungen, über die eine neue Konfiguration in den lokalen Konfigurationsspeicher gelangen kann. Über eine geeignete Schnittstelle (z. B. JTAG) ist ein direkter Zugriff von außen auf den

Konfigurationsspeicher möglich. Alternativ bietet sich die Nutzung der von außen erreichbaren Schnittstellen des Hostprozessors oder des RMs an, um neue Daten im Konfigurationsspeicher abzulegen. Dazu müssen Hostprozessor bzw. RM schreibenden Zugriff auf den Konfigurationsspeicher haben. Darüber hinaus muss das RM über die für das Schreiben notwendige Logik verfügen. Aufgrund dieser Randbedingungen ergeben sich vier Kombinationen, um eine Konfiguration über Hostprozessor und/oder RM in den Speicher zu schreiben:

- vom Hostprozessor direkt in den Speicher,
- vom RM direkt in den Speicher,
- vom Hostprozessor über das RM in den Speicher und
- vom RM über den Hostprozessor in den Speicher.

3.3 Architekturen für eingebettete Systeme

Die in Abschnitt 3.1 beschriebenen Architekturen sind eher auf die Erweiterung von Workstations oder PCs ausgelegt. Einige von ihnen finden sich im Bereich der eingebetteten Systeme wieder. Durch die Fortschritte bei Energieverbrauch, Verlustleistungsreduktion und die sinkenden Preise haben sich im kommerziellen Umfeld FPGAs als Umsetzung für das rekonfigurierbare Modul etabliert – und damit auch die mit FPGAs realisierbaren Architekturen.

Die wohl verbreitetste Architektur bei eingebetteten Systemen ist vermutlich die Kopplung von Prozessor und externem RM über Peripheriebusse oder mit direkter Anbindung an den Hostprozessor (vgl. Architekturen 2 und 4). Hier können die preiswerten und ausgereiften Prozessoren für eingebettete Systeme durch kleinere FPGAs mit anwendungsspezifischen Funktionen unterstützt werden. Außerdem lässt der Ansatz viele Freiheitsgrade bei der Umsetzung von Kommunikations- und Konfigurationsarchitektur. Somit finden sich diverse kommerzielle Produkte, wie die AVM Fritz! Box oder die Studio MovieBox Deluxe von Pinnacle Systems. Eine Realisierung als SoM erhöht die Flexibilität zusätzlich, da bei Pin-Kompatibilität zu weiteren Modulen ein zusätzlicher Freiheitsgrad bei der Systemumsetzung entsteht.

Die Integration eines Prozessorkerns direkt in die rekonfigurierbare Logik ist ebenfalls bei verschiedenen Produkten anzutreffen. Die bekannten Softcore-Implementierungen [TAK06] wie NIOS II, MicroBlaze oder Leon sind nach Bedarf skalierbar, bieten eine gute Dokumentation und ebenso guten Support. Allerdings sind sie im Vergleich zu dedizierten Prozessoren für eingebettete Systeme nur wenig leistungsfähig und verhältnismäßig teuer, da sie wertvolle Logikressourcen im FPGA belegen. Sie ermöglichen jedoch die Nutzung der Vorteile von Prozessoren bei der Umsetzung von Kontrollfluss und Speicherzugriffen und vereinfachen das Design der Leiterplatten dahingehend, dass nur die wohlbekannte Kombination von FPGA und Konfigurationsspeicher notwendig ist. Da die Softcores meist als abgeschlossene IP-Cores verwendet werden, ähneln die Systeme der vierten Architekturvariante, bei der Prozessorkern und Funktionsblöcke im RM über einen internen Systembus kommunizieren.

Alternativ sind auch ICs mit als Hardmakro integriertem Prozessor verfügbar. Nachdem Xilinx mit der Virtex5 Serie die letzten FPGAs mit PowerPC-Kern angeboten hatte, kommt gerade die Zynq-Architektur [Xil12f] auf, bei der zwei ARM-Kerne zusammen mit einem rekonfigurierbaren Teil auf einem Die[6] integriert sind. Hier erfolgt die Kopplung der Prozessoren und der konfigurierbaren Logik über den *AMBA™AXI*[7].

Als sehr gute Alternative erscheinen je nach Anforderung an die Größe des RMs die beiden Architekturen mit enger Kopplung. Jedoch gibt es bisher keine nennenswerten kommerziellen Umsetzungen von Prozessoren für eingebettete Systeme mit eingebettetem RM. Dabei erscheint die Beschleunigung einzelner Befehle oder Befehlssequenzen als geeignetes Instrument für eingebettete Systeme, und Projekte wie MORPHEUS [VRH09; Put09] zeigen das Potential solcher Lösungen auf.

Ausgehend von der Kombination von Prozessor und RM ergibt sich als nächstes die Frage nach den weiteren architektonischen Merkmalen. Über die Position im Datenpfad entscheidet letztlich die Anwendung. Ziel sollte hier aus Gründen der Ressourceneffizienz eine Minimierung redundanter sowie überdimensionierter Systemkomponenten sein. Auch die Anforderungen an

[6] Ungehäuster Halbleiter-Chip.
[7] *Advanced Microcontroller Bus Architecture™Advanced eXtensible Interface Bus (AMBA™AXI)*

die Kommunikationsarchitektur werden maßgeblich durch den Anwendungsbereich vorgegeben. Die Verwendung etablierter Busarchitekturen und der damit verbundenen Kommunikationsprotokolle (z. B. *Direct Memory Access (DMA)*) dürfte jedoch den Vorteil des geringeren Implementierungsaufwandes haben. Leistungsabschätzungen für konkrete Umsetzungen sind aufgrund von Standards und Spezifikationen mit mittlerem Aufwand realisierbar und natürlich sinnvoll.

Signifikante Freiheitsgrade ergeben sich damit durch die Wahl des Konfigurationsansatzes. Bei der Umsetzung eines statischen Systems (vgl. Abschnitt 2.5.1) hat das FPGA eine Kommunikationsverbindung zum Hostprozessor und einen PROM zur Speicherung der Konfigurationen.

Dieser etablierte Ansatz wird auch gern als Grundlage für vollständig rekonfigurierbare Systeme verwendet. Dabei schreiben Hostprozessor oder FPGA[8] eine neue Konfiguration in den PROM. Anschließend erfolgt ein Neustart des FPGAs zur Aktivierung der neuen Konfiguration. Zusammen mit einer Fallback-Konfiguration (vgl. Abschnitt 2.4.3) ist so auf einfache Weise ein robustes System umsetzbar. Hinweise zu derartigen Systemen finden sich auch direkt bei den FPGA Herstellern [Alt12a; HP08]. Diese Lösungen sind allerdings nicht besonders ressourcensparend, da ein PROM und Verbindungen zu dessen Programmierschnittstelle benötigt werden.

Als Alternative bietet sich für vollständig rekonfigurierbare ebenso wie für dynamisch rekonfigurierbare Systeme das Programmieren des FPGAs durch den Hostprozessor an. Die Konfigurationen werden in einem durch den Hostprozessor erreichbaren Speicher hinterlegt und es ist nur eine Verbindung zur Programmierschnittstelle des RMs notwendig. Damit ist diese Variante mit deutlich weniger Ressourcenaufwand verbunden. Auch hier finden sich Implementierungsvorschläge direkt bei den Herstellern [Xil07b; DF00; Car99; PK06]. Allerdings ist die Umsetzung mit mehr Aufwand verbunden, da die Konfiguration vom Hostprozessor aus bedient werden muss. Die für die notwendige Software vorhandenen Beispiele decken nicht alle Schnittstellen ab und machen daher in vielen Fällen eine entsprechende Softwareimplementierung notwendig.

[8] Die Nutzung des RMs für Konfigurationsaufgaben bindet weitere der meist begrenzten Logikressourcen (vgl. Abschnitt 6.4.3).

Mit einem Flash-Speicher und einem geeigneten FPGA ist eine ähnliche Lösung möglich, bei der sich das RM seine Konfiguration direkt aus dem Flash-Speicher lädt. Wird ein RM mit integriertem Prozessor genutzt, ist immer ein externer Speicher notwendig, der wenigstens eine initiale Konfiguration vorhält.

3.4 Entwurf eingebetteter heterogener rekonfigurierbarer Systeme

Beim Entwurf von hrS kommt dem HW/SW-Codesign eine besondere Rolle zu, denn für die dynamischen Ansätze ist die Verfügbarkeit von Logikressourcen und passenden Konfigurationen essentiell. Nach der hinreichenden Modularisierung des Gesamtsystems werden die in rekonfigurierbarer Logik umzusetzenden Teile festgelegt. Die Aufteilung in Hard- und Software sowie das Festlegen von Konfigurationszeitpunkten und das Zuweisen von Konfigurationsressourcen kann statisch zur Designzeit des Systems erfolgen. Dieser Ansatz ist im Besonderen für Echtzeitbetriebssysteme geeignet, da hier die Einhaltung von Zeitschranken und die Verfügbarkeit von Ressourcen im Vordergrund stehen und daher die Ablaufplanung oft fest zur Designzeit erfolgt. Im Betrieb sind somit der Systeminstanz, die die Rekonfiguration steuert und überwacht, Reihenfolge und Zeitpunkt der Konfigurationsvorgänge bekannt.

Ist eine statische Zuteilung der Konfigurationsressourcen nicht möglich, sind weitere Entwurfsschritte und Maßnahmen notwendig. Denn nun sind der kontrollierenden Systeminstanz Reihenfolge und Zeitpunkt der Rekonfiguration nicht bekannt. Somit ist auch eine statische Zuteilung der Konfigurationsressourcen nicht mehr möglich. Diesem Problem kann auf verschiedene Weise begegnet werden (vgl. Abschnitt 2.5.2). Ein Ansatz ist die Bereitstellung mehrerer PR-Module, die in verschiedene PR-Regionen passen. Dies bedeutet jedoch einen erhöhten Ressourcenbedarf, da jedes Modul mehrfach vorhanden ist. Weitere Ansätze sind die Platzierung der rekonfigurierbaren Module zur Laufzeit (Relokation) oder die voll-dynamische Erzeugung der Module. Ausführliche Beschreibungen verschiedener Ansätze, die noch immer Ge-

genstand wissenschaftlicher Untersuchungen sind, finden sich beispielsweise in [WB04; Wal05; PTW10; Mei10].

3.5 Betriebssystem

Heterogene rekonfigurierbare Systeme stellen nicht nur eine eigene Klasse der Hardwarearchitektur für eingebettete Systeme dar und erfordern erweiterte Entwurfskonzepte, sie eröffnen auch neue architektonische Ansätze bei Betriebssystemen. Besonderes Augenmerk gilt in der Literatur dabei dem effizienten Nutzen der rekonfigurierbaren Logik bei Systemen mit dynamischer (dR) und im besonderen dynamisch-partieller Rekonfiguration (dpR) (u. a. [WB04; Wal05; S007; PTW10]). Aus Anwendungssicht ist es von besonderem Interesse, die Komplexität bei der Verwaltung der Konfigurationen und beim Konfigurationsvorgang an sich gering zu halten. Daher finden sich in der Literatur verschiedene Ansätze zur Bereitstellung geeigneter Methoden zur Verwaltung der Konfigurationen sowie für einfach nutzbare Konfigurationsschnittstellen (u. a. [WB04; Wal05; Str05; S007; Str+09]). Weiterhin ist das Scheduling der Konfigurationsvorgänge von Interesse, speziell im Bereich der RTOS, aber auch bei der Nutzung klassischer Betriebssysteme (u. a. [BW03; Wal05; PTW10]).

3.5.1 Konfigurationsstrategien

In Ergänzung zu den vorgenannten Quellen erfolgt in diesem Abschnitt eine Betrachtung von Konfigurationsstrategien aus Sicht des Betriebssystems, wobei das Hauptaugenmerk auf der Frage:

- Zu welchem Zeitpunkt kann oder muss die Konfiguration erfolgen und welche Vor- und Nachteile birgt dies?

liegt. Eine Diskussion im Kontext eines konkreten Betriebssystems erfolgt in Abschnitt 4.5.

Die Frage nach dem Zeitpunkt ergibt sich aus Hard- und Softwarearchitektur. Stark verallgemeinert gilt jedoch, dass die Konfiguration immer dann erfolgt

sein muss, wenn eine der Systemschichten (vgl. Abschnitt 2.6.1) die Ressource benötigt. Ist die Ressource nicht bereit oder nicht verfügbar, muss entsprechend gewartet werden. Durch die Rekonfiguration ändern sich ggf. die Funktionen und Schnittstellen des RMs. Alle involvierten Teile des System müssen mit dieser Dynamik umgehen können, da es sonst zu Inkonsistenzen und Fehlverhalten kommt. Im Sinne der Stabilität des Systems sollte auch die erfolgreiche (Re-) Konfiguration überprüft werden. Verfeinert man die Betrachtung möglicher Zeitpunkte mit Bezug zur System- und Softwarearchitektur, ergeben sich vier wesentliche Konfigurationszeitpunkte.

Initiale Konfiguration (initial configuration) Ist die erfolgreiche Konfiguration des RMs Voraussetzung für den Beginn des Bootvorgangs, z. B. bei Systemen mit im RM integriertem Prozessor, muss sie direkt beim Einschalten des Systems (*Power On*) geladen werden. Daher benötigt das RM Zugriff auf die Konfiguration. Eine Überwachung des Konfigurationsvorgangs hat entweder durch zusätzliche Mechanismen im System (z. B. Hardware-Watchdog) oder durch das RM zu erfolgen. Für den Fehlerfall können Rettungsansätze mit Hilfe von MultiBoot oder Fallback-Konfiguration verfolgt werden, um trotzdem Zugang zum System zu erlangen.

Frühe Konfiguration (early access) Werden Funktionen des RMs bereits in einer frühen Phase des Systemstarts (z. B. in der Initialisierungsphase des Kernels) benötigt, bietet sich alternativ zur initialen Konfiguration eine Konfiguration über den Hostprozessor mit Hilfe des Bootloaders an. Dieser kann bereits Teile der Systemhardware (Netzwerkschnittstellen, Massenspeicher) in Betrieb nehmen und so Konfigurationen verfügbar machen, auf die das RM nicht direkt zugreifen kann. Die Überwachung der Konfiguration kann ebenso wie die Reaktion im Fehlerfall in Software erfolgen, was die Flexibilität erhöht.

Konfiguration per Kernel (early OS access) Die Konfiguration des RM kann auch während der Initialisierungsphase des Kernels erfolgen. Entweder in einer sehr frühen Startphase (early init), falls das RM eine wesentliche Systemkomponente ist, oder erst später im Bootprozess. Die sehr frühe Initialisierung führt im Fehlerfall sehr wahrscheinlich zum Fehlschlagen des Bootprozesses und damit zu einem nicht funktionsfähigen System. In diesem Fall sind Fallback-Lösungen in Software nur

schwer zu implementieren. Der Ansatz sollte entsprechend kritisch betrachtet werden, kann sich aber trotzdem als notwendig und sinnvoll erweisen, beispielsweise wenn der Bootloader nicht für eine Konfiguration genutzt werden kann und auch die initiale Konfiguration nicht verfügbar ist. Erfolgt die Konfiguration erst zusammen mit der Inbetriebnahme weiterer Peripherie, können die gleichen Fehlerbehandlungsroutinen greifen, die auch bei anderen Hardwarefehlern genutzt werden. Hier sollte das Betriebssystem entsprechende Fallback-Lösungen bereithalten.

Späte Konfiguration (late access) Die Möglichkeit der Rekonfiguration aus dem laufenden Betriebssystem heraus dürfte neben der initialen Konfiguration die wohl häufigste Ausprägung sein. Der Übergang vom early OS access zum late access auf Kernel-Ebene entspricht der Nutzung eines im Betrieb ladbaren Treibers/Kernel-Moduls. Alternativ kann eine beliebige Anwendungssoftware einen Konfigurationsprozess anstoßen, wenn entsprechende Schnittstellen zur Verfügung stehen. Die späte Konfiguration bietet die größtmögliche Vielfalt an Konfigurations-, Rekonfigurations- und Fehlerbehandlungsmöglichkeiten. Nachteil ist die späte Verfügbarkeit des konfigurierten RMs.

3.5.2 Konfigurationsmanagement

Durch ihre Flexibilität und die vielfältigen Architekturen können heterogene rekonfigurierbare Systeme nahezu beliebig eingesetzt werden. Allerdings hängt die effiziente Nutzung der Vorteile rekonfigurierbarer Systeme direkt an der nahtlosen Integration ins Betriebssystem. Das umfasst die Kommunikation mit dem RM ebenso wie die Konfiguration. Wird sehr häufig rekonfiguriert oder existieren sehr viele rekonfigurierbare Ressourcen, ist eine Managementfunktion erforderlich, die den Zustand der RMs überwacht.

Durch die Möglichkeit zur Rekonfiguration entsteht eine zusätzliche Dynamik im System – das RM wird ggf. neu gestartet und ändert u. U. seine Funktion und/oder Schnittstellen. Diesem Umstand müssen alle beteiligten Komponenten Rechnung tragen. Für den Zeitraum der Konfiguration sind Zugriffe auf das RM zu verhindern. Das verwendete Kommunikationssystem muss das kurzzeitige Verschwinden eines Knotens unterstützen oder das RM verhindert

die Rückwirkung der Konfiguration auf das Kommunikationssystem. Nach der Konfiguration sind eventuell Initialisierungen des RMs notwendig. Das erinnert an den im Serverbereich etablierten HotPlug-Mechanismus [KL01], der das Wechseln von Hardware im laufenden Betrieb unterstützt. Allerdings sind die dort verwendeten Mechanismen im Embedded-Bereich nicht immer verfügbar, da neben dem Betriebssystem auch alle beteiligten Hardwarekomponenten für HotPlug geeignet sein müssen.

Bei der Rekonfiguration müssen die Systemarchitektur und die damit verbundenen Randbedingungen beachtet werden. Je nach System beinhaltet dies den korrekten Umgang mit Neustarts des RMs oder des Systems, das Laden bzw. Entladen von Treibern/Modulen und das Bereitstellen der Konfigurationen. Entsprechend der Feststellung, dass alle hrS Architekturen für RTR geeignet sind (vgl. Abschnitt 3.2.3), ist eine Betrachtung von partieller und vollständiger Konfiguration notwendig. Systemarchitekturen, die eine Rekonfiguration ohne Neustart des Systems zulassen, bedürfen eines komplexeren Konfigurationsmanagements als solche, die nach einer Konfiguration neu gestartet werden müssen. Dabei ist es kaum von Belang, ob das angeschlossene RM vollständig, dynamisch oder dynamisch-partiell rekonfiguriert wird. Sobald sich der Zustand des RMs aus Sicht des Betriebssystems durch die Konfiguration verändert, wird ein entsprechendes Konfigurationsmanagement notwendig. Die grundsätzlichen Managementabläufe stehen in enger Verbindung mit der gewählten Konfigurationsarchitektur.

Abbildung 3.5 auf der nächsten Seite stellt einen vereinfachten Ablauf der dynamischen Rekonfiguration aus Betriebssystemsicht dar. Um undefiniertes Systemverhalten zu vermeiden, müssen alle Anwendungen, die auf den zu rekonfigurierenden Teil des RMs zugreifen, über den bevorstehenden Konfigurationsvorgang informiert werden. Anschließend gilt es, alle Treiber/Kernel-Module mit Verbindung zum RM zu deaktivieren. Erst jetzt kann eine Rekonfiguration nahezu ohne Rückwirkung auf das System erfolgen. Anschließend ist eine Reinitialisierung aller betroffenen Systemteile notwendig, bevor das RM mit dem neuen Systemverhalten zur Verfügung steht.

Bei Systemen mit Endo-Rekonfiguration ist das RM nicht unmittelbar von der Veränderung betroffen, da das Übertragen der veränderten Konfiguration in den externen Speicher noch kein neues Systemverhalten bewirkt. Die Übernahme der Änderungen (Konfigurationsvorgang) muss explizit ausge-

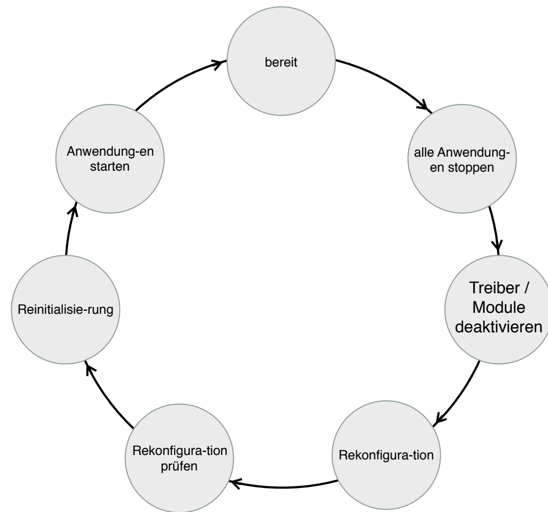


Abbildung 3.5: Schematische Darstellung der allgemeinen Aufgaben bei der dynamischen Rekonfiguration

löst werden. Bis dahin steht das RM mit dem „alten" Verhalten zur Verfügung. Die fehlerfreie Übertragung der Daten in den Konfigurationsspeicher sollte durch die Software überwacht werden. Zur Sicherheit ist die zusätzliche Umsetzung einer Fallback-Lösung empfehlenswert. Die Daten werden bei dieser Lösung wenigstens zwei mal im System übertragen, einmal in den Speicher und von dort zum RM. Allerdings erlaubt die Architektur eine intensivere Sicherung der Datenübertragung, wenn es die sonstigen zeitlichen Vorgaben für den Konfigurationsvorgang nicht verletzt. So ist beispielsweise ein Rücklesen der Konfigurationsdaten denkbar, um die fehlerfreie Übertragung in den Speicher sicherzustellen.

Systeme mit Exo-Rekonfiguration stehen während des Konfigurationsvorgangs gar nicht[9] oder nur eingeschränkt[10] zur Verfügung. Schlägt die Übertragung der Konfigurationsdaten fehl, verlängert sich die Ausfallzeit entsprechend. Im Gegenzug werden die Daten direkt an das RM übertragen, was

[9] bei dynamischer Rekonfiguration
[10] bei dynamisch-partieller Rekonfiguration

die Konfigurationszeit im Vergleich zu Systemen mit zusätzlichem Konfigurationsspeicher verringern kann.

In einem System mit *initial configuration*, dass ebenfalls mit Exo-Rekonfiguration arbeitet, muss zur Aktivierung des geänderten Verhaltens ein Neustart inklusive Konfigurationsvorgang erfolgen, was den Ablauf im Vergleich zu Abbildung 3.5 auf der vorherigen Seite vereinfacht. Eine neue Konfiguration wird unter Beachtung der notwendigen Datenintegrität in den Speicher übertragen. Anschließend erfolgt der Neustart des System in einer Weise, die auch die Konfiguration des RM veranlasst. Nun steht das neue Systemverhalten zur Verfügung und durch den Neustart ist auch eine komplette Reinitialisierung erfolgt.

Durch Multi-Context FPGAs und Konfigurationsspeicher mit mehreren Konfigurationen (vgl. Abschnitt 2.4.3) sind weitere Lösungen möglich, bei denen zwischen Konfigurationen umgeschaltet wird, die sich bereits im Speicher befinden. Das entspricht zeitlich der direkten Übertragung zum RM und ermöglicht Übertragungsstrategien wie bei Systemen mit zusätzlichem Konfigurationsspeicher und darüber hinaus. Neue Konfigurationen werden in inaktive Speicherbereiche geschrieben, die erst nach der Verifikation freigegeben werden.

Durch den geschickten Einsatz von dpR sind auch Systemkonfigurationen umsetzbar, bei denen der Konfigurationsvorgang für Betriebssystem und beteiligte Kommunikationskomponenten transparent ist. Damit entfällt der Ablauf nach Abbildung 3.5. Zusammen mit einer geeigneten Warteschlangenstrategie im Kommunikationstreiber (z. B. blocking read/write) bleibt der Prozess an sich für die Anwendungen unbemerkt.

3.6 Zusammenfassung

Dieses Kapitel führt detailliert in die Architektur von heterogenen rekonfigurierbaren Systemen ein. Es definiert den Begriff und fasst die Architekturdarstellung nach Koch [Koc02], Compton/Hauck [CH02] und Todman [Tod+05] zusammen. Danach folgt eine detaillierte Diskussion von Kommunikationsarchitekturen, Möglichkeiten zum Datenaustausch zwischen RM und Hostprozessor sowie Konfigurationsarchitekturen. Nach der Diskussion zur Verwen-

dung von hrS in eingebetteten Systemen werden die Besonderheiten dieser Systeme aus Sicht der Betriebssystems dargestellt und diskutiert.

Die durch den Autor geführte vertiefte Diskussion der Systemarchitekturen erlaubt die Berücksichtigung der Besonderheiten dieser Systemklasse bereits während der Spezifikationsphase, was aufgrund der Relevanz einzelner Entscheidungen für die nachfolgenden Entwurfsschritte in allen Domänen des Systementwurfs zu signifikanten Verbesserungen führt.

Der Einsatz eines hrS bringt gegenüber dedizierten Hardwaresystemen Vorteile bei der Umsetzung des Kontrollflusses sowie beim Umgang mit irregulären Datenstrukturen. Im Vergleich zu Systemen ohne RM bieten sie einen sehr flexiblen Co-Prozessor. In Abschnitt 3.2.1 wird deutlich, dass die Position des RMs im Kommunikationspfad maßgeblich von der Zielanwendung bestimmt wird. Die Übertragung der Ansätze zum Datenaustausch in parallelen und verteilten Rechnersystemen ergänzen die Ausführungen unter Berücksichtigung der Anforderungen aus Anwendungssicht.

Die umfassende Darstellung der verschiedenen Konfigurationslösungen ist eine Verallgemeinerung von Ansätzen aus Literatur, eigenen Implementierungen und gängigen Vorschlägen der Hersteller von FPGAs. Sie ergänzt die Beschreibungen der Systemarchitekturen aus Abschnitt 3.1, die die Konfiguration nicht mit einbeziehen. Somit geben die Abschnitte 3.1 und 3.2 einen detaillierten und umfassenden Überblick über die Systemarchitekturen von hrS und verstehen sich als Beitrag für die optimale Auslegung des Systems und dadurch eine deutliche Verringerung der Komplexität bei Leiterplatten-, Hardware- und Softwareentwurf.

Die anschließende Übertragung der Erkenntnisse auf eingebettete Systeme geht mit einer Einschränkung der zuvor eingeführten Architekturvarianten einher. Die grundlegende Systemarchitektur ergibt sich demnach aus den Anforderungen der Anwendung, lässt sich allerdings durch die Berücksichtigung der einzelnen Architekturaspekte vereinfachen. Speziell die Konfigurationsarchitektur eröffnet ein erhöhtes Maß an Flexibilität. Hier können Leistungsfähigkeit, Ressourcenbedarf und Implementierungsaufwand mit mehr Spielraum gegeneinander abgewogen werden. Dabei wird deutlich, dass ein direkter Zusammenhang zwischen Konfigurationsstrategie und Systemarchitektur besteht.

Diese Erkenntnis spiegelt sich auch in der Betrachtung zu Konfigurationsstrategien und -management aus Betriebssystemsicht wieder. In Ergänzung zur Literatur erfolgt eine Betrachtung zu Konfigurationszeitpunkten, die unabhängig vom speziellen Betriebssystem und auch unabhängig von der Art des hrS (statisch, dynamisch oder dynamisch-partiell) ist. Dazu werden vier Konfigurationszeitpunkte eingeführt und diskutiert. Ebenso erfolgt eine Diskussion grundsätzlicher Managementabläufe aus Betriebssystemsicht unter Beachtung der vorgestellten Konfigurationsarchitekturen und -strategien.

4 Das Betriebssystem Linux im Kontext des Systementwurfs

Dieses Kapitel befasst sich mit dem Betriebssystem Linux. Dabei erfolgt die technische Betrachtung der Funktionen und Zusammenhänge gerade in dem Maße, wie es für das Verständnis des vorliegenden sowie der folgenden Kapitel notwendig ist. Intention der Ausführungen ist viel mehr, die Faktoren bei der Auswahl und Anpassung von Linux als Betriebssystem für eingebettete Systeme zu beleuchten, die den Systementwurf sowie Betrieb und Wartung beeinflussen. Dazu erfolgt im Anschluss an die Vorstellung allgemeiner Grundlagen eine Systematisierung von Themen und Abläufen mit Bezug zum Entwurf von Linux-Systemen. Anschließend werden diese auf eingebettete und heterogene rekonfigurierbare Systeme übertragen.

4.1 Linux Grundlagen

Wie in Abschnitt 1.3 bereits erläutert, hat das Betriebssystem Linux seinen Ursprung in den Arbeiten des finnischen Studenten Linus Torvalds. Doch erst die kostenfreie Bereitstellung des Quelltextes (*Open Source*) und die Kooperation mit anderen freien Projekten, wie den GNU-Tools der FSF, ermöglichten das komplexe Unix-System, zu dem Linux inzwischen geworden ist.

Neben der Biografie von Linus Torwalds [TD02] sei dem geneigten Leser das Buch von Karim Yaghmour [Yag+08] empfohlen, dass einen sehr ausführlichen Einstieg in die Thematik Linux für eingebettete Systeme bietet. Ebenso lesenswert sind die Bücher von Christopher Hallinan [Hal06] und Schröder et. al.[SGD09]. Allen Veröffentlichungen zum Thema Linux ist gemein, dass

die immense Dynamik in der Weiterentwicklung des Systems viele Detailbeschreibungen schon während der Ausarbeitung in Teilen oder vollständig obsolet macht. Daher beschränkt sich die folgende Darstellung von wesentlichen Grundlagen zum Linux-Betriebssystem auf gefestigte Begriffe und Fakten mit geringerer Dynamik.

4.1.1 Begriffe

Der Begriff *Linux* stand ursprünglich für den Betriebssystem-Kern (*Kernel*). Dieser bietet die Basisfunktionen eines Betriebssystems (vgl. Abschnitt 2.6) sowie die speziell dem Linux eigenen Funktionen, welche eine für Linux entwickelte Software als gegeben ansehen kann. Aus hier nicht näher diskutierten Gründen wird der Begriff *Linux* heute gleichwertig für den Systemkern, für auf den Linux-Kernel aufbauende Systeme und für sogenannte *Distributionen*, also Zusammenstellungen von Software basierend auf dem Linux-Ökosystem, genutzt. Aufgrund dieser begrifflichen Unschärfe gibt es verschiedene Diskussionen zur Schärfung des Begriffsraumes. Der Entwicklung in anderen Fachartikeln und auch der persönlichen Vorliebe des Autors folgend, werden daher im weiteren Text dem Kontext entsprechend die Begriffe *Linux-Kernel* für den Kernel, *Linux-System* für auf dem Kernel basierende Systeme und *Distribution* für vorgefertigte, auf dem Linux-Ökosystem basierende Zusammenstellungen genutzt.

4.1.2 Linux-Systeme

Abbildung 4.1 auf der nächsten Seite zeigt ausgehend von Abbildung 2.10 auf Seite 65 ein an die Systemarchitektur von Linux-Systemen angepasstes, verfeinertes Schichtenmodell. Der Kernel, dessen grobes Schichtenmodell sich ebenfalls in der Abbildung findet, gehört zu den monolithischen Kernen. Er vereint, trotz modularem Aufbau, neben den Basisfunktionen eines Betriebssystems auch Kernel-Erweiterungen (engl.*kernel extensions*) und Treiber in einer funktionalen Einheit. Kernel, Kernel-Erweiterungen und (nahezu) alle Treiber werden im *Kernel Space* ausgeführt, was die Nutzung des privile-

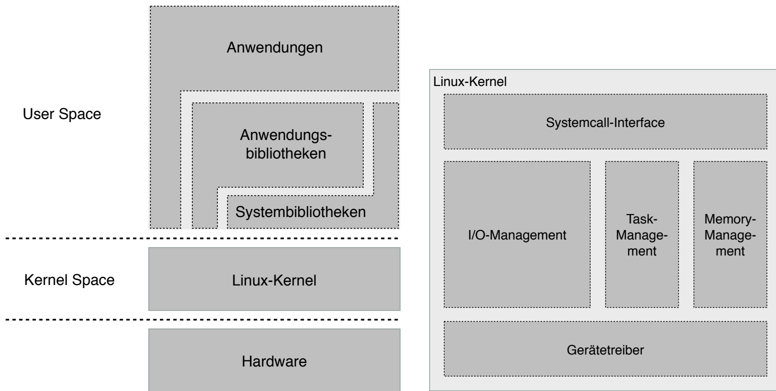


Abbildung 4.1: Das Linux-Schichtenmodell von Kernel und System

gierten *Ring 0*[1] der CPU und die Nutzung eines abgegrenzten RAM-Bereichs bezeichnet. Der Zugriff auf Funktionen des Kernels aus dem User Space erfolgt über Systemaufrufe (auch *Systemcalls* genannt). Das API zwischen Kernel und User Space wird daher auch als Systemcall-Interface[2] bezeichnet. Die ihm bekannten Systemaufrufe speichert der Kernel in der *System Call Table*. Der Kernel ist weiter in Subsysteme untergliedert, die je nach Funktion den Blöcken I/O-Management (z. B. Dateisysteme, Netzwerk), Task-Management (z. B. Scheduling), Memory-Management und Gerätetreiber (z. B. PCI, USB) zuzuordnen sind.

Im User Space (oder *User Land*) werden alle Anwendungsprogramme ausgeführt. Auch wenn die Möglichkeit besteht, die Abhängigkeiten einer Anwendung zu anderen Systemteilen durch statisches Binden (statisches Linken) aufzulösen, dominiert bei Linux aus Ressourcen- und Effizienzgründen das Konzept der geteilten Bibliotheken (engl.*shared libraries*). Diese *Laufzeitbibliotheken* können nach verschiedenen Gesichtspunkten klassifiziert werden. Für die

[1] Viele Architekturen (z. B. x86, ARM, PowerPC) unterstützen mehrere Sicherheitsstufen, die die Rechte des ausgeführten Prozesses beeinflussen.

[2] Das Systemcall-Interface definiert die Schnittstelle auf Quellcode-Ebene. Auf Binärcode-Ebene wird die Schnittstelle durch das *Application Binary Interface (ABI)* bzw. das *Embedded Application Binary Interface (EABI)* definiert. Speziell bei Recherchen im Internet erfolgt nicht immer eine saubere Trennung dieser Ebenen.

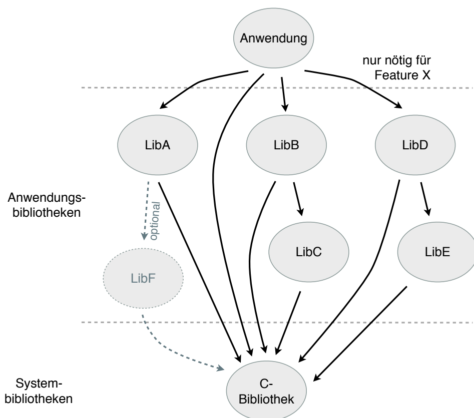


Abbildung 4.2: Fiktiver Abhängigkeitsbaum einer Anwendung

folgenden Abschnitte reicht eine grobe Einteilung in Systembibliotheken und Anwendungsbibliotheken. Systembibliotheken sind die Bibliotheken, die den Zugriff auf das Systemcall-Interface des Kernels abstrahieren und vereinfachen, wie z. B. die C-Bibliothek. Anwendungsbibliotheken hingegen kapseln typische Funktionen höherer Ebenen, die dann durch Anwendungen genutzt werden, beispielsweise komplexe mathematische Funktionen oder grafische Elemente. Wie in Abbildung 4.1 auf der vorherigen Seite dargestellt, ergibt sich ein hierarchischer Aufbau. Ein Anwendungsprogramm greift meist auf Funktionen von Anwendungs- und Systembibliotheken zu. Anwendungsbibliotheken nutzen üblicherweise auch die Systembibliotheken. Alle Schichten können auf das Kernel-API zugreifen. Abbildung 4.2 stellt die entstehende Hierarchie an einem fiktiven Beispiel dar. Das gezeigte Anwendungsprogramm hat aufgrund seiner Funktion *direkte Abhängigkeiten* zu zwei Anwendungsbibliotheken sowie zur C-Bibliothek. Für die Nutzung einer optionalen Funktion[3] ist eine weitere Bibliothek (als direkte Abhängigkeit) notwendig. Die von der Anwendung verwendeten Bibliotheken haben ebenfalls direkte statische oder dynamische Abhängigkeiten, was dazu führt, dass das Anwendungsprogramm

[3] Die Verfügbarkeit optionaler Funktionen hängt von der Implementierung der Software ab, und wird entweder statisch bei der Erstellung oder dynamisch durch die Verfügbarkeit der entsprechenden Bibliotheken gewählt.

indirekt auch von diesen abhängt (*indirekte Abhängigkeit*). Somit kann ein Abhängigkeitsbaum beliebig tief sein.

Zugriff auf Hardware

Dem Motto *„Everything is a File“*[4] folgend wird auch die Kommunikation mit Hardware über sogenannte Gerätedateien abstrahiert. Dabei unterscheidet man zwischen:

- zeichenorientierten Geräten (*character devices*),
- blockorientierten Geräten (*block devices*) und
- socketorientierten Geräten (*sockets* bzw. *socket devices*).

Über diese Gerätedateien können Programme aus dem User Space beliebige Daten mit dem Kernel über die klassischen Dateibefehle `open( )`, `read( )`, `write( )`, `seek( )`, `close( )` austauschen. Die Gerätedateien finden sich unter Linux im Verzeichnis `/dev`.

Gerätedateien bilden die Schnittstelle zu den entsprechenden Treibern im Linux-Kernel. Die Treiber müssen ihrerseits den Vorgaben der Kernel-APIs folgen. Bis zur Aufnahme eines Treibers in die offiziellen Kernel-Quellen ist der jeweilige Entwickler für die Behebung von Fehlern und die Einhaltung der API-Vorgaben zuständig. Der damit verbundene Aufwand kann aufgrund der hohen Geschwindigkeit, mit der sich der Linux-Kernel entwickelt, schnell einen signifikanten Anteil der gesamten Entwicklungsleistung umfassen. An die Qualität von Kernel-Modulen werden besondere Ansprüche gestellt, da Fehler zur Instabilität des gesamten Systems führen können. Die Fehlersuche ist jedoch schwieriger als bei User Space Programmen, obwohl der Kernel inzwischen mehrere Vereinfachungen wie den *Kernel Debugger (KDB)* und das *debugfs* bietet. Einen Ausweg bieten Treiberimplementierungen im User Space, wie sie die *User Space Input and Output (UIO)* Treiber (siehe [Koc09]) bieten. Hier laufen nur zeitkritische Abschnitte im Kernel Space, während der Rest des Treibers im User Space implementiert wird. Detaillierte Ausführungen zur Entwicklung von Kernel-Modulen bieten [CRK05] und [KQ06].

---

[4] dt. Übersetzung: „Alles ist eine Datei.“

Neben den Gerätedateien existieren weitere Mechanismen für den Austausch von kleinen Datenmengen und Statusinformationen zwischen User und Kernel Space[5]. Dazu zählen eine Reihe durch Software generierte (virtuelle) Dateisysteme, die aus dem User Space über Datei- und Verzeichnishierarchie erreichbar sind. Das *proc*-Dateisystem (i. Allg. zu finden unter `/proc`) dient dem Kernel zum Bereitstellen von prozessbezogenen Informationen. Die Einträge im proc-Dateisystem werden durch Objekte des Kernels angelegt. Zugriffe leitet der Kernel an die jeweiligen Ersteller weiter, die den Inhalt dann dynamisch erzeugen. Das *Sysfs* (i. Allg. zu finden unter `/sys`) repräsentiert die Hierarchie der Subsysteme des Kernels und erlaubt den Zugriff auf Attribute der im System vorhandenen Kernel-Objekte (hauptsächlich Geräte und Treiber). Es entstand zusammen mit dem aktuellen Gerätetreibermodell, soll das proc-Dateisystem entlasten und eine klar definierte Schnittstelle zu Kernel-Objekten bieten. Während das sysfs Zugriff auf bestehende Objekte bietet, ist das *configfs* für die Erstellung und Manipulation von Kernel-Objekten gedacht. Um das proc-Dateisystem auch von Debug-Meldungen verschiedener Kernel-Module zu bereinigen, wurde mit dem *debugfs* eine Schnittstelle zur Bereitstellung genau dieser Funktionalität eingeführt.

Einen weiteren Weg zur Interaktion mit einem Treiber bietet die *I/O-Control* Schnittstelle, die über den Funktionsaufruf `ioctl()` erreichbar ist. Aufgrund ihrer nicht festgelegten Struktur ist sie jedoch nicht die erste Wahl für die Umsetzung komplexer Schnittstellen zum User Space.

Tabelle 4.1 auf der nächsten Seite fasst die Kommunikationsschnittstellen noch einmal zusammen.

Toolchain

Ein Linux-System besteht aus dem Kernel und einer Auswahl von Bibliotheken und Anwendungsprogrammen, die aufeinander abgestimmt sind. Weiteres wesentliches Element eines Linux-Systems sind die Werkzeuge zum Erstellen von Kernel, Bibliotheken und Anwendungen – im Allgemeinen als *Toolchain* (Werkzeugkette) bezeichnet. Die Toolchain umfasst:

[5] Für einen Überblick zu diesen und weiteren Schnittstellen siehe z. B. [WWX10].

Tabelle 4.1: Schnittstellen zur Hardware

Zugriff auf	Zugriff über	Beispiel
zeichenorientierte Geräte	`/dev`	Serielle Schnittstelle über `/dev/ttyS0`
blockorientierte Geräte	`/dev`	Festplatte über `/dev/sda1`
socketorientierte Geräte	`/dev` oder Netzwerkstack	Systemmeldungen über `/dev/log`
prozessbezogenen Informationen	`/proc`	Informationen zur RAM-Auslastung über `/proc/meminfo`
Status von Kernel-Objekten	`/sys`	Aktuelle CPU-Frequenz über `/sys/devices/... /cpuinfo_cur_freq`
Kernel-Objekte zur Modifikation	`/config`	Anlegen eines PWM-Objekts über `/config/gpio_pwm`
Debug-Informationen	`/sys/kernel/ debug/`	Informationen zum gpio System über `/sys/kernel/debug/gpio`

- den Compiler (meist *gcc* aus der *GNU Compiler Collection (GCC)*),
- die *binutils* (u. a. bestehend aus Assembler (*gas*) und Linker (*ld*)),
- eine C-Bibliothek (z. B. *glibc* [Fre], *eglibc* [Lin], *uclibc* [And99]) und
- optional einen *Debugger* (meist *gdb*).

Sie steht am Anfang eines jeden Linux-Systems, denn erst nach dem Erstellen einer funktionierenden Toolchain können Kernel und Systemsoftware kompiliert werden. Somit ist das Erstellen und Pflegen der Werkzeugkette eine der Hauptaufgaben für Anbieter von Linux-Systemen. Gleichzeitig ist die Qualität

und Aktualität der Toolchain ein Wettbewerbsmerkmal – speziell im kommerziellen Bereich. Das *Linux From Scratch!* Projekt [Bee98] vermittelt einen sehr guten Eindruck davon, wie man eine Werkzeugkette und mit ihrer Hilfe ein Linux-System erstellt.

Systemstart und -stop

Der Start eines Linux-Systems folgt dem in Abbildung 4.3 dargestellten Ablauf. Nach der minimalen Initialisierung der Hardware durch den/die Bootloader wird der Kernel gestartet. Er vervollständigt die Initialisierung der Hardware und macht sie den Anwendungen im User Space zugänglich. Unter anderem bindet der Kernel ein Dateisystem mit den Programmen für den weiteren Systemstart ein. Am Ende seiner Initialisierungsphase startet der Kernel den ersten Anwendungsprozess – den *Init*-Prozess. Der Init-Prozess initialisiert die Systemumgebung und startet weitere Systemdienste. Daher ist jeder Prozess im User Space ein Kind des Init-Prozesses.

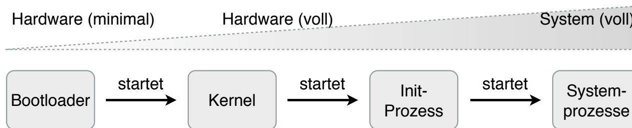


Abbildung 4.3: Vereinfachter Bootprozess

Das vom Kernel eingebundene Dateisystem, welches alle Elemente zum Hochfahren des Betriebssystems und zum Einbinden weiterer Partitionen beinhaltet, wird als *root file system* (dt. Wurzeldateisystem) bezeichnet. Dabei kann es sich sowohl um ein echtes Dateisystem auf einem Datenträger handeln, als auch um ein temporäres, speziell für den Bootvorgang. In letzterem Fall kommt heute meist ein sogenanntes *Initiales RAM Dateisystem (initramfs)* [Yag+08] zum Einsatz, das im weiteren Verlauf des Bootvorgangs durch ein reales Dateisystem ersetzt wird. Der Verzeichnisbaum im realen Wurzelverzeichis folgt in den meisten Fällen den Vorgaben des *File System Hierarchy Standard (FHS)*.

Die Abläufe des Linux-Init-Prozesses sind nahezu frei konfigurierbar und umfassen unter anderem das Einbinden weiterer Dateisysteme, das Setzen der Systemzeit, die Initialisierung der Netzwerkverbindungen, sowie das Starten von Systemdiensten, Anmeldeprozessen (login) und, falls vorhanden, der grafischen Oberfläche. Verbreitete Init-Systeme sind *SysVinit*, *systemd*, *upstart*, *launchd* und System Management Facility (SMF). Einen einheitlichen Standard gibt es jedoch nicht. Bis etwa 2010 war SysVinit das vorherrschende Linux-Init-System. Es arbeitet alle Startaufgaben in einer vorgegebenen, sequentiellen Folge ab, wodurch sich der Startprozess mit jeder Aufgabe verlängert. Daher gab es verschiedene distributionsspezifische Erweiterungen, die Teile des Systemstarts parallelisieren. Mit upstart kam 2006 ein alternativer Ansatz, der das parallele und ereignisorientierte Ausführen von Initialisierungsaufgaben beherrscht. Einen ähnlichen Ansatz verfolgt das seit 2010 verfügbare systemd. Die Parallelisierung des initialen Startvorgangs führt zur Verkürzung der Zeit bis zur Bereitschaft des Systems, erschwert jedoch das Auffinden von Fehlern im Startprozess. Das ereignisorientierte Ausführen von Initialisierungsaufgaben während der Laufzeit des Systems trägt ebenfalls zur Verringerung der Startzeit bei.

Beim Herunterfahren des Systems kommen wiederum die Funktionen des Init-Systems zum Einsatz. Die von ihm gestarteten Prozesse werden beendet, Daten auf nichtflüchtige Speicher gesichert und Dateisysteme freigegeben. Ziel ist ein Systemzustand, der das Unterbrechen der Stromzufuhr ohne Datenverlust zulässt.

Eine ausführliche Darstellung zum Bootvorgang auf x86-Systemen bietet [Jon06]. Eine detaillierte Erläuterung findet sich in [Hal06, Kapitel 5 und 6].

4.2 Softwaremanagement im Linux-Umfeld

Distributionen sind umfangreiche und zentral gesteuerte Zusammenstellungen von Linux-Kernel, Toolchain, Laufzeitbibliotheken und Anwendungsprogrammen. Sie haben den Anspruch, ein dem vorgesehenen Verwendungszweck entsprechendes Gesamtsystem bereitzustellen. Die meist modular gehaltenen Distributionen stellen ihren Softwareumfang üblicherweise in Form

von sogenannten Paketen (engl. *Packages*) bereit. Außerdem bieten sie eigene Programme für:

- die Installation der Distribution,
- die Installation und Aktualisierung einzelner/aller Pakete und
- die Konfiguration des Linux-Systems.

Auch die Konfiguration der Pakete wird von den *Distributoren* übernommen bzw. vorbereitet. Außerdem unterscheiden sich die Distributionen in ihren Startkonzepten und -systemen wesentlich. Und nicht zuletzt liefern die Distributionen auch Grafiken und Töne für ein Corporate Design. Bekannte Distributionen sind u. a. OpenSUSE, Fedora, Debian GNU/Linux, Ubuntu (vgl. [Uns01]).

Die Distributoren sind aus Marktsicht Konkurrenten und versuchen ihre Kunden durch bestimmte Funktionen oder Konzepte an sich zu binden. Daher sind die verschiedenen Distributionen, auch wenn sie sich in weiten Teilen an Standards wie den FHS oder die *Linux Standard Base (LSB)* halten, meist nicht kompatibel zueinander. Dies liegt natürlich an der Zusammenstellung von Kernel, Bibliotheken und Anwendungen, die sich in Version und Funktionsumfang zwischen den Anbietern unterscheiden. Außerdem sind die verschiedenen Ansätze zur Paketverwaltung (Distribution als Binär- oder Quellpaket, Paketformat) nur bedingt kombinierbar. Auch die Alleinstellungsmerkmale wie Konfigurationsverwaltung und Startprozess oder die abweichende Nutzung von Speicherorten für relevante Software führen zu Inkompatibilität.

4.2.1 Paketmanagement

Wie die Ausführungen in Abschnitt 4.4 noch zeigen werden, ist das Erstellen von Softwarepaketen ein aufwändiger Prozess. Die Verfügbarkeit und Qualität des Quellcodes der Software hat hier einen entscheidenden Einfluss. Daher wird an dieser Stelle der Erstellungs- und Qualitätssicherungsprozess für Softwarepakete im Zusammenspiel von Entwicklern, Betreuern und Anwendern eingehender betrachtet. Abbildung 4.4 auf der nächsten Seite stellt die idealisierte und leicht vereinfachte Produktionskette einer Open Source Software unter Verwendung der üblichen, teilweise englischen Begriffe dar.

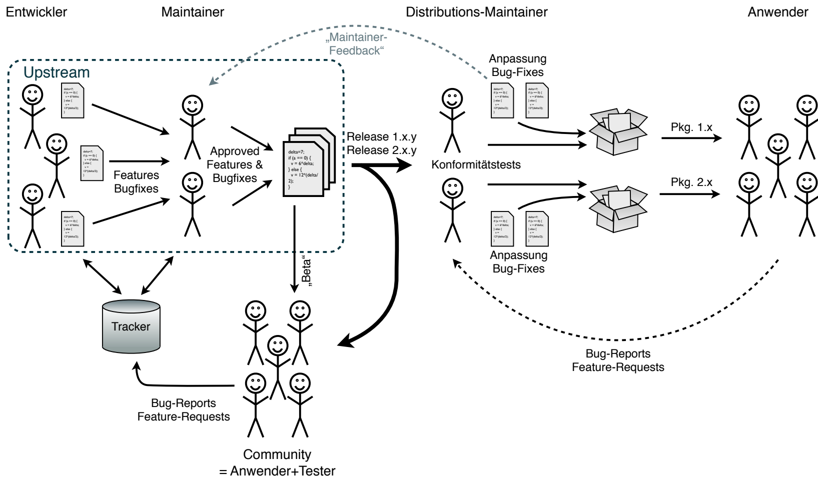


Abbildung 4.4: Softwarezyklus im Linux-Umfeld

Der Quellcode eines Projektes wird von seinen Entwicklern erstellt, gewartet und weiterentwickelt. Die *Maintainer* (dt. Betreuer) eines Projekts übernehmen die Rolle von Moderatoren. Sie sichten die Fehlerkorrekturen (engl. *Bugfixes*) und Funktionserweiterungen (engl. *Features*) der Entwickler, stellen deren Funktion sicher (Qualitätsmanagement) und überführen sie in geordneter Form in die verschiedenen Entwicklungszweige und -stände der Software. Entwickler und Betreuer werden zusammen als *Upstream* bezeichnet. In kleineren Projekten übernehmen ausgewählte Entwickler die Rolle der Betreuer parallel zu ihren Entwicklungsaufgaben. Viele Projekte werden aktiv durch Nutzer unterstützt (*Community*), die die Software bereits in der Entwicklungsphase testen (*Beta-Tests*) und mit Fehlerberichten und Vorschlägen für neue Funktionen zur Weiterentwicklung beitragen. Idealerweise erfolgt der Austausch über möglichst formalisierte Wege. Der direkte Kontakt von Entwicklern und Betreuern mit der Community erfolgt meist über E-Mail-Listen oder per *Internet Relay Chat (IRC)*. Auf diesem Weg werden Fragen beantwortet, mögliche Funktionserweiterungen diskutiert und auch Fehlerberichte entgegengenommen. Allerdings ist die Art der Kommunikation nicht bzw. nur teilweise formalisiert (semiformal). Eine Möglichkeit der

weiteren Formalisierung von Fehlermeldungen und Funktionserweiterungen bieten spezielle Softwarewerkzeuge wie Bug-Tracker oder mit Hilfe von *Versionsverwaltungssystemen* erstellte Patches.

Eine als stabil und für den produktiven Einsatz geeignete Version der Software (*Release*) wird allen potentiellen Anwendern zur Verfügung gestellt. Dies erfolgt meist durch die Bereitstellung von angepassten, durch eine Versionsnummer gekennzeichneten Quellcode-Paketen im Internet, oftmals ergänzt durch (Binär-) Pakete für verbreitete Distributionen.

Gelegentlich existieren mehrere stabile Releases parallel, beispielsweise weil eine neue Funktion oder eine größere Anpassung der Softwarearchitektur zur Inkompatibilität mit den Vorgängerversionen geführt hat. Um Anwendern und Programmierern hinreichend Zeit für die Anpassung ihrer Arbeitsabläufe und Produkte einzuräumen, werden alte und neue Version parallel gepflegt. Die Kennzeichnung stabiler Releases erfolgt mit Hilfe von Versionsnummern, für die sich zwei leicht unterschiedliche Nomenklaturen etabliert haben:

- `<NAME>-<Major>.<Minor>.<Patch>-<Indikator>`
- `<NAME>-<Minor>.<Patch>-<Indikator>`

Bei der ersten, häufigeren Variante bleibt der Paketname über die Releases hin gleich. Die *Major-Nummer* unterscheidet (ggf. inkompatible) Release-Zweige voneinander. Die *Minor-Nummer* weist auf größere Änderungen (z. B. neue Funktionen) hin, die jedoch nicht zur Inkompatibilität führen. *Patch* bzw. *Patchlevel* kennzeichnet Fehlerkorrekturen innerhalb einer Minor-Release. Alle drei Teile der Versionsnummer sind numerisch. Der Indikator weist auf Betaversionen hin und ist oftmals nicht rein numerisch (z. B. `-rc1`). Bei der zweiten Variante entfällt die Major-Nummer. Major-Releases unterscheiden sich durch eine Änderung im Namen. Dies vereinfacht die parallele Existenz mehrerer Releases in einem Linux-System. Obwohl diese Systeme weitgehend Anerkennung finden, gibt es kein standardisiertes Vorgehen bei der Nummerierung oder der Bedeutung der einzelnen Stellen, sodass die Aussage der Versionsnummer im Einzelfall immer zu prüfen ist. Im Internet finden sich Ansätze zur weiteren Formalisierung und Standardisierung der Versionsnummerierung (vgl. [Pre]), die zu vergleichbaren Erkenntnissen wie dieser Abschnitt gelangen.

Eine durch einige Projekte aufgegriffene Alternative zu expliziten Feature- und Bugfix-Releases ist ein "Rapid Release Process" (dt. beschleunigter Veröffentlichungsprozess), der eher an das Modell der „Rolling Releases" (vgl. Abschnitt 4.2.2) erinnert. In kurzen Zeitabständen (beispielsweise wöchentlich oder monatlich) werden Versionen veröffentlicht, die sowohl Fehler beseitigen als auch neue Funktionen bieten. Endanwender kommen so schneller in den Genuss neuer Funktionen. Die Versionsnummern bei dieser Art der Veröffentlichung ist entweder kontinuierlich fortlaufend oder bezieht den Release-Zeitpunkt ein (z. B. `<NAME>.<Jahr>-<Monat>-<Indikator>`).

Auch die Distributoren verfolgen die Weiterentwicklung der von ihnen verwendeten Softwarepakete und reagieren mit unterschiedlichen Strategien auf neue Versionen (vgl. [FHK13]). Die Pakete einer Distribution werden durch eigene Betreuer erstellt, getestet und gewartet. Sie unterziehen die Upstream-Version verschiedenen Tests, um Stabilität und Kompatibilität zur sonstigen Software der Distribution zu gewährleisten. Außerdem nehmen sie notwendige Anpassungen an die Spezifika der Distribution vor. Häufig beheben sie auch Fehler, die während der Tests auftreten, direkt. Nach der Freigabe durch die mit der Qualitätssicherung beauftragen Mitarbeiter der Distribution wird das Paket über die Kanäle der Distribution an ihre Nutzer verteilt. Die Auslieferung der Pakete erfolgt in der Regel in Form von kompilierten Binärpaketen. Der Anwender hat daher nahezu keinen Einfluss auf die Abhängigkeiten der Pakete. Auch der Umfang optionaler Funktionen einer Software wird allein durch den Betreuer und/oder den Distributor festgelegt.

4.2.2 Versionsmanagement

Differenzierungsmerkmale zwischen den Distributionen sind unter anderem auch die Aktualität der Pakete, die Häufigkeit und Art von Paketaktualisierungen (*Updates*) und die Behandlung von bekannt gewordenen Fehlern.

Bei den Aktualisierungsstrategien für die Distribution unterscheidet man zwischen Distributionen mit festen Versionen und solchen mit *Rolling Releases* (kontinuierliche Veröffentlichungen). Distributionen mit festen Versionen veröffentlichen von Zeit zu Zeit eine Version mit einer meist großen Anzahl von (signifikanten) Versionsänderungen. Ab dem Moment der Veröffentlichung folgt eine Phase der Stabilität aller Versionen, in der nur noch

Bugfix-Releases ausgeliefert werden, die Fehler und Schwachstellen beseitigen. Major-Updates gibt es in dieser Zeit nur in seltenen Fällen. Damit werden eine gewisse Kontinuität gesichert und inkompatible Versionssprünge vermieden. Zu dieser Art von Distributionen gehören beispielsweise Debian GNU/Linux, Ubuntu, RedHat (und Fedora) sowie SuSE (und OpenSuse). Distributionen mit Rolling Releases hingegen aktualisieren ihre Pakete kontinuierlich, auch wenn damit inkompatible Änderungen einhergehen. Auch hier gibt es Versionen, also Punkte innerhalb der Entwicklungsgeschichte, an denen der momentane Stand aller Pakete „notiert" wird. Doch es gibt keine längeren oder gar garantierten Phasen der Stabilität einzelner Versionen. Bekannte Vertreter sind Gentoo Linux, Arch Linux und PCLinuxOS.

Distributionen mit Rolling Releases versprechen die höhere Aktualität der ausgelieferten Pakete, während Distributionen mit Release-Zyklen Kontinuität bieten und es Herstellern von Fremdsoftware ermöglichen, ihre Produkte auch über einen längeren Zeitraum nicht an neue Softwareversionen anpassen zu müssen.

4.2.3 Automatisiertes Versionsmanagement

Wie bereits beschrieben, umfassen Distributionen auch Werkzeuge für die Installation und Aktualisierung der verfügbaren Pakete. Das zugrundeliegende Konzept gleicht sich bei der Mehrzahl der Distributionen. Die Software auf den Endgeräten pflegt eine Liste der verfügbaren und installierten Pakete und Paketversionen. Sie verwaltet die Installation und Deinstallation einzelner Pakete und der mit ihnen verbundenen Abhängigkeiten und erlaubt bei Distributionen mit festen Versionen die Aktualisierung der Distributionsversion.

Für die Synchronisation der Liste der verfügbaren Pakete/Versionen stellt der Distributor eine entsprechende Infrastruktur über das Internet bereit. Der Anwender wird durch die lokalen Werkzeuge über neue Versionen der von ihm installierten Pakete informiert und kann den Aktualisierungsvorgang anstoßen. Die Bereitstellung der Pakete erfolgt meist ebenfalls durch eine vom Distributor bereitgestellte Infrastruktur. Zusammen mit der Aktualisierung einzelner Pakete geht auch die Pflege der damit in Verbindung stehenden Abhängigkeiten einher. Die Installation neuer Software greift auf die gleiche Infrastruktur und Werkzeuge zurück.

Der Vorgang aus:

- Paketliste aktualisieren,
- auf neue Versionen installierter Pakete prüfen und
- Aktualisierung der Pakete und ihrer Abhängigkeiten

lässt sich augenscheinlich gut automatisieren, was unter Begriffen wie *AutoUpdate* oder *unattended Update* firmiert [AF13]. Automatisierte Aktualisierungen sind jedoch immer mit einem erhöhten Risiko verbunden, da ein Fehler nicht notwendigerweise direkt bemerkt wird. Diesem Risiko, welches besonders bei der automatisierten Aktualisierung wesentlicher Systemkomponenten (z. B. Kernel, Init-System) immer besteht, lässt sich durch verschiedene Maßnahmen begegnen. Durch das explizite Ein- bzw. Ausschließen von Paketen beim automatisierten Vorgehen können sensible Bereiche geschützt werden. Ebenso besteht die Möglichkeit, automatisierte Aktualisierungen auf bestimmte Paketquellen zu beschränken, sodass ein eigener Server nur Pakete bereitstellt, die vorher hinreichend durch Administratoren getestet wurden.

4.3 Besonderheiten bei eingebetteten Systemen

Auch wenn es Anfangs nicht darauf ausgerichtet war, eignet sich Linux durch seine Modularität ausgesprochen gut für eingebettete Systeme. In [Yag+08] erfolgt in Ergänzung zu Abschnitt 2.3 die Definition eingebetteter Systeme aus Sicht des Linux-Betriebssystems. Dabei werden im Gegensatz zur klassischen Definition die Größe, zeitliche Aspekte, Netzwerkfähigkeit und der Grad der Nutzerinteraktion zur Definition genutzt.

Größe Die physische Größe eines Systems limitiert natürlich auch die Komponenten. ICs, Speicher, Schnittstellen und eventuell notwendige Kühlsysteme sind durch das verfügbare Volumen begrenzt. Dies wiederum beeinflusst auch die Möglichkeiten beim Aufbau eines eingebetteten Systems. Doch im Linux-Sprachgebrauch spielt der Begriff „Größe“ eine weitere Rolle und bezeichnet dabei die Leistungsparameter der Hardware. *Kleine Systeme* verfügen nur über

ein schwaches Prozessorsystem, wenige Megabyte RAM (ab etwa 4 MB) und ebenso wenig ROM bzw. Flash Speicher. Systeme unterhalb dieser Leistungsklasse können zwar auch genutzt werden, in Anbetracht der Entwicklung auf dem Hardware-Sektor erscheint der dafür zu zahlende Preis jedoch inadäquat hoch. *Mittlere Systeme* bieten sowohl ein leistungsfähigeres Prozessorsystem als auch mehr RAM (typisch 64 MB und mehr) und ROM (meist einige Gigabyte Flash Speicher). Geräte mit leistungsstarken (Mehrkern-)SoCs oder sogar Prozessoren aus dem Desktop-PC-Bereich, mehreren Gigabyte RAM und nichtflüchtigem Speicher bilden die Klasse der *großen Systeme*.

Zeitliche Aspekte Oft werden eingebettete Systeme für Aufgaben eingesetzt, die harte oder weiche Echtzeitanforderungen stellen. Das Spektrum reicht von Überwachungs- oder Regelungsaufgaben, über Anforderungen im Mobilfunk- oder Telekommunikationsbereich bis zur flüssig laufenden, grafischen Bedienoberfläche. Diese Systeme lassen sich auch auf Basis des Linux-Kernels realisieren, wobei die Strategien von hinreichend starker Hardware, über optimierte Software bis hin zu Erweiterungen des Linux-Kernels reichen.

Netzwerkfähigkeit Die Möglichkeit zur Anbindung an Kommunikationsnetze gilt bei heutigen eingebetteten Systemen quasi als Standard. In diesem Bereich können Linux-Systeme ihre Stärken ausspielen, da die gesamte Entwicklung von Linux eng mit der Nutzung in Kommunikationsnetzen verzahnt war und ist. Der Linux-Kernel bietet dementsprechend hervorragend entwickelte Netzwerkfunktionen und es gibt eine breite Softwarebasis für alle Aufgaben in Verbindung mit Netzwerken.

Nutzerinteraktion Ausgehend vom Anwendungsgebiet kann die Nutzerschnittstelle lediglich rudimentär entwickelt sein (z. B. einige Leuchtdioden zur Statusanzeige) oder eine komplexe Interaktion des Nutzers ermöglichen (z. B. bei Smartphones oder Tablet-PCs).

In Zusammenhang mit Linux für eingebettete Systeme spielen weitere Punkte eine Rolle. Zum einen ist da die Frage nach geeigneter Hardware. In [Hal06,

Seite 38 ff.] sowie in Kapitel 6 ab Seite 145 finden sich verschiedene Beispiele für Prozessorsysteme und Architekturen, die vom Linux-Kernel unterstützt werden. Die Anzahl unterstützter Architekturen und Prozessorsysteme wächst jedoch kontinuierlich. Dabei sind weder eine *Memory Management Unit (MMU)* noch eine *Floating Point Unit (FPU)* Voraussetzung, auch wenn Architekturen mit MMU empfohlen werden. Bezüglich der minimal erforderlichen Menge an RAM zeigt sich der Linux-Kernel mit 2 MiB sehr bescheiden. Für ein minimales Linux-System werden 4 MiB empfohlen. Aufgrund der vielen Möglichkeiten, ein Linux-System zu booten, ist die Angabe eines minimalen ROM-Ausbaus nicht sinnvoll.

Die Unterstützung einer Prozessorarchitektur durch den Linux-Kernel und daraus abgeleitet die Unterstützung eines auf der Architektur basierenden Prozessorsystems oder SoCs ist noch kein Hinweis auf die Vollständigkeit der Unterstützung. Gleiches gilt für alle Gerätetreiber. Daher sollte die Auswahl einer Hardware immer unter Berücksichtigung der durch das bevorzugte Linux-System gebotenen Unterstützung erfolgen.

Durch die Flexibilität eines Linux-Systems und die Menge an verfügbarer Software, die in Verbindung mit dem Linux-Kernel arbeitet, sind die Freiheiten bei der Software weit größer. Zuerst stellt sich die Frage, ob das System mit einer Distribution betrieben werden soll, oder mit einem speziell erstellten Linux-System. Letzteres erhöht die Flexibilität wesentlich und erlaubt den Einsatz nahezu beliebiger Hardware, erfordert jedoch zusätzlichen Aufwand bei Erstellung, Installation und Test (vgl. Abschnitt 4.4). Um von den Vorteilen einer Distribution zu profitieren (vgl. Abschnitt 4.2), muss man sich an die Vorgaben bezüglich unterstützter Hardware und Paketauswahl halten. In jedem Fall muss aus der Menge an vorhandener Software eine Auswahl getroffen werden, die alle Anforderungen an Funktionalität, Größe, zeitliche Aspekte, Netzwerk- und Nutzerinteraktion erfüllt. Wie sich in den folgenden Abschnitten zeigen wird, ist dies einer der wesentlichen Aspekte beim Entwurf eines Linux-Systems. Alternativ besteht die Möglichkeit, einen Dienstleister in Anspruch zu nehmen, der sich um die Umsetzung der Anforderungen kümmert und im Rahmen der Produkthaftung auch für mögliche Fehler einsteht.

4.4 Erstellen eines Linux für eingebettete Systeme

Mit der wachsenden Marktdurchdringung von Linux wächst auch die Unterstützung der etablierten Distributionen für Architekturen mit beschränkten Ressourcen. Ein aktuelles Beispiel ist der „Einplatinencomputer“ Raspberry Pi (siehe Abschnitt 6.5), der mit seinem geringen Preis von nur 35 € innerhalb kürzester Zeit eine sehr große Nutzergemeinschaft ansprach. Waren anfangs nur wenige Distributionen für die verwendete ARMv6-Architektur verfügbar, änderte sich dies mit dem Erfolg des Raspberry Pi in kurzer Zeit. Die Umsetzung eines eingebetteten Systems ist jedoch in den meisten Fällen eine kunden- oder anwendungsspezifische Aufgabe und macht damit auch die Erstellung eines angepassten Linux-Systems notwendig. Wie bereits erläutert, werden dazu ausgehend von einer funktionierenden Toolchain der Linux-Kernel sowie die benötigten Bibliotheken und Anwendungsprogramme aus ihrem jeweiligen Quellcode erstellt. Das Kompilieren der einzelnen Teile des zukünftigen Linux-Systems, das Auflösen der Abhängigkeiten zu anderen Programmen und Bibliotheken ebenso wie die korrekte Installation auf dem Zielsystem sind aufwändige und fehleranfällige Tätigkeiten. Hinzu kommt der Umstand, dass viele Softwarepakete die Verwendung in eingebetteten Systemen nur unzureichend berücksichtigen, und so immer wieder Anpassungen und Fehlerkorrekturen notwendig sind. Glücklicherweise lassen sich die einzelnen Schritte zur Erstellung eines funktionsfähigen Linux-Systems weitgehend automatisieren, was im Folgenden näher betrachtet wird.

4.4.1 Werkzeuge

Ausgehend von Abschnitt 4.1.2 ist bereits bekannt, dass zur Erstellung eines Linux-Systems eine Toolchain, minimal bestehend aus Compiler, Binutils und C-Bibliothek, notwendig ist. Bei der Erstellung von Software für ein eingebettetes System verfügt das Zielsystem (engl. *Target*) aber häufig entweder nicht über hinreichende Ressourcen für die Kompilierung von Software, oder aber die Kompilation würde aufgrund der begrenzten Ressourcen zu viel Zeit in Anspruch nehmen. Daher hat sich eine als *Cross-Development* bezeichnete Arbeitsmethode etabliert, bei der die Software für das Zielsystem auf einem

leistungsstarken Hostrechner erstellt wird. Voraussetzung dafür ist die sogenannte *Cross-Toolchain*, bei der die beteiligten Werkzeuge zwar auf der Architektur des Hostrechners laufen, aber Binärcode für das Zielsystem erstellen (sog. *Cross-Kompilieren*).

In nahezu allen Fällen wird die GCC in Kombination mit den GNU binutils und einer C-Bibliothek[6] als Grundlage für die Cross-Toolchain genutzt. Das Erstellen einer funktionsfähigen Cross-Toolchain ist ein sehr komplexer und fehleranfälliger Vorgang. Die einzelnen Komponenten müssen aufeinander abgestimmt sein, da nicht jede Kombination der Werkzeuge auch funktionsfähigen Binärcode erzeugt. Hinzu kommt, dass die Zielarchitektur korrekt durch jedes der Werkzeuge unterstützt werden muss, was ebenfalls nicht immer der Fall ist. Daher gibt es Software, die das Erstellen der benötigten Cross-Toolchain vereinfachen. Sie bieten üblicherweise für die von ihnen unterstützten Architekturen eine korrekt funktionierende Kombination von GCC, GNU binutils und C-Bibliothek sowie eventuell noch notwendige Patches. Gleichzeitig automatisieren sie den Erstellungsprozess. Vertreter dieser Gruppe sind *crossdev* [ohn11], *crosstool* [Keg06], *crosstool-ng* [Mor11] und OSELAS.Toolchain( ) [Pen11]. Viele Buildsysteme liefern jedoch bereits eine geeignete Cross-Toolchain mit. Außerdem bieten immer mehr Hersteller Toolchains und Entwicklungsumgebungen für ihre Produkte an. Der schnellen Verfügbarkeit steht hier wiederum die Abhängigkeit vom Anbieter gegenüber.

4.4.2 Buildsysteme

Ausgehend von einer funktionierenden Cross-Toolchain können der Kernel sowie gewünschte Bibliotheken und Anwendungsprogramme aus ihren Quellpaketen erstellt werden. Auch wenn sich dieser Vorgang bei einzelnen Softwarepaketen unterscheidet, gibt es doch viele Projekte, die etablierte Systeme wie das GNU Build System [Stö07], `cmake` oder `qmake` nutzen. Daher bietet sich eine Automatisierung der immer wiederkehrenden Schritte während der Softwareerstellung an. Die dazu verwendeten Systeme werden in dieser Arbeit als *Buildsysteme* bezeichnet. Sie können als eine Art *Metadistribution* betrachtet werden, da sie alle Aufgaben zur Erstellung einer Binärdistribution für genau

[6] Meist glibc, eglibc, uClibc oder newlib

ein eingebettetes System übernehmen. Abbildung 4.5 stellt im linken Teil die grundsätzliche Funktionsweise eines Buildsystems und die dazu benötigten Informationen dar. Im rechten Teil ist das Zusammenspiel der einzelnen Teile eines Buildsystems veranschaulicht. Beides wird nachfolgend erläutert.

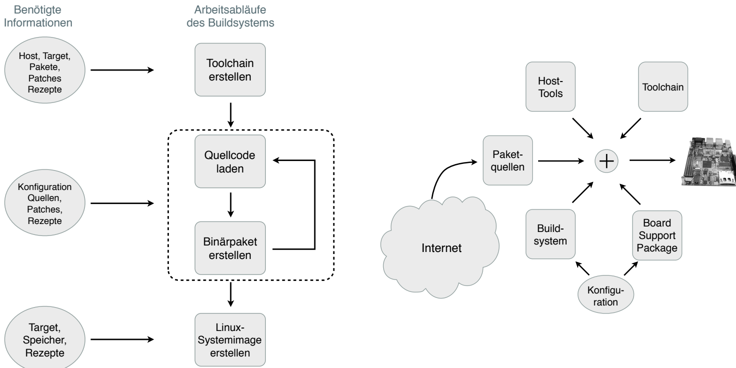


Abbildung 4.5: Grundsätzliche Funktionsweise von Buildsystemen

Ein Buildsystem erstellt aus Quellpaketen eine (Cross-) Toolchain, ggf. benötigte Anwendungen auf dem Host sowie Binärpakete von Kernel, Bibliotheken und Anwendungen für ein bestimmtes Zielsystem. Bei Bedarf führt es die Binärpakete anschließend in einem Systemimage zusammen. Außerdem sollte es die Softwarepakete sowie ihre Abhängigkeiten untereinander verwalten und eine Möglichkeit zur Konfiguration[7] des entstehenden Linux-Systems bieten. Das Buildsystem benötigt für jedes der unterstützten Softwarepakete eine „Bauanleitung", die die einzelnen Schritte vom Herunterladen des Quellcodes über das Entpacken und ggf. notwendige Patchen bis hin zum Kompilieren und zur Installation beschreibt. Diese „Bauanleitungen" werden gemeinhin als *Recipes* (dt. Rezepte) oder *Rule Files* bezeichnet. Neben der Toolchain und den Rezepten werden weitere Werkzeuge auf dem Hostsystem benötigt, die *Host-Tools*. Spezifische Software und Informationen zum Zielsystem sind im sogenannten *Board Support Package (BSP)* zusammengefasst. Buildsysteme, die einige oder alle dieser Anforderungen erfüllen, gibt es in großer Zahl.

[7] Konfiguration bezieht sich hier auf die Auswahl von Kernel, Bibliotheken und Anwendungen.

Die im Rahmen dieser Arbeit eingehender betrachteten sind *Buildroot* [Viz13], *Embedded Linux Development Kit* (*ELDK*), *Gentoo Linux* [Wro08], *OpenWrt* [Fai08], *OpenEmbedded* [STM10] und *PTXdist* [Pen12]. Für die Vorstellung einzelner Buildsysteme wird auf die jeweiligen Internetpräsenzen [Ope09; And09; Gen01] sowie auf vorhandene Literatur [Reb08; GR09; Kam09; KH09; SH10] verwiesen. Die dort getroffenen Feststellungen decken sich mit den Erfahrungen des Autors.

Ebenso wie Distributionen stehen auch Buildsysteme in einer gewissen – wenn auch meist kooperativen – Konkurrenz zueinander. Wie bereits in [KKH11] festgestellt, gibt es dabei klassische Unterscheidungsmerkmale. Offensichtliche Attribute sind die Art und Anzahl der unterstützten Hardware und die Menge der angebotenen Pakete. Ein weiteres Merkmal ist die Ausrichtung des Buildsystems. Adressiert es allgemein die Erstellung eines individualisierten Linux-Systems, oder ist es, wie beispielsweise OpenWrt und DD-WRT, auf Systeme für einen bestimmten Einsatzzweck ausgerichtet, oder werden spezielle Eigenschaften wie eine besonders geringe Größe angestrebt? Auch das in Abschnitt 4.4.3 näher beschriebene Patchmanagement stellt ein Funktions- bzw. Unterscheidungsmerkmal dar. Dokumentation, Support und Aktivität der Community sind drei weitere Punkte. Eine gute und aktuelle Dokumentation erleichtert den Einstieg in die Arbeit mit dem Buildsystem. Dem einfachen Einstieg trägt auch die Komplexität des Buildsystems Rechnung. Allerdings zeigt sich, dass Komplexität und Flexibilität in direktem Zusammenhang stehen. Je einfacher sich ein Buildsystem gibt, desto schwieriger ist die Erweiterung um eigene Pakete und um eigene Systeme. Dabei stecken die Probleme oft in Details wie der Patchverwaltung, der Verzeichnisstruktur oder der für eine einfache Erweiterung notwendigen Modularisierung.

Auch wenn die grundsätzliche Vorgehensweise der Buildsysteme beim Erstellen eines Linux-Systems gleich ist, unterscheiden sich die spezifischen Ansätze sehr stark. Je nach Art der Installation kann man *zentrale* und *dezentrale* Systeme unterscheiden. Als zentrale Systeme werden solche betrachtet, bei der die Nutzer auf eine gemeinsame Installation des Buildsystems und der Toolchain zugreifen. Dezentrale Systeme hingegen werden von jedem Nutzer lokal installiert und verwendet. Das Erstellen der Pakete erfolgt in beiden Fällen in einem Arbeitsverzeichnis lokal beim Anwender. Für diese Art der Nutzung hat sich der Begriff *Out of Tree Builds* etabliert. Zentrale Systeme vereinfachen die Arbeit innerhalb eines Teams, da alle Mitglieder auf einer gemeinsamen

Basis aufsetzen, die sich zentral weiterentwickeln lässt. Die Nutzer sind nur für die Pflege ihrer lokalen Änderungen verantwortlich. Bei dezentralen Systemen ist der jeweilige Nutzer für die Pflege seines Basissystems verantwortlich. Außerdem ist die Art, wie die Buildsysteme mit Erweiterungen der Nutzer umgehen, an das Grundkonzept angepasst. Durch Out of Tree Builds kann auch bei dezentralen Ansätzen das Arbeitsverzeichnis inklusive Toolchain im Dateibaum parallel zum Buildsystem liegen. Ein weiterer Punkt, der von der Architektur des Buildsystems beeinflusst wird, ist die Art der Modularisierung. Auf der einen Seite gibt es Systeme, die die Auswahl von Software (Anwendungen, Bibliotheken) getrennt von den durch die Hardware beeinflussten Teilen (BSP) verwalten. Erst beim Erzeugen einer Systemsoftware erfolgt die Verbindung der Softwareauswahl mit einer spezifischen Hardwarekonfiguration. Dieses Vorgehen ermöglicht die Nutzung der gleichen Software auf verschiedener Hardware ebenso wie die Nutzung der gleichen Hardware in verschiedenen Anwendungen und erleichtert so z. B. den Wechsel der Hardwareplattform. Im Gegensatz dazu stehen die Systeme, die die Kombination von Hard- und Software als ein Ganzes behandeln und damit für jede Hardware-/Softwarekombination eine eigene Konfiguration hinterlegen[8].

Buildsysteme sollten weitere Aufgaben übernehmen, die besonders im Bereich der eingebetteten Systeme von Bedeutung sind. Produkte können hier durchaus über einen längeren Zeitraum am Markt verfügbar sein. So lange sollte auch die Versorgung mit Updates sichergestellt werden. Mit der Markteinführung endet jedoch meist die aktive Entwicklungsphase. Alle Aufzeichnungen und der Quellcode werden archiviert und die Entwickler wenden sich neuen Aufgaben zu oder verlassen gar die Firma. Kommt nun eine Supportanfrage, ist es notwendig, dass Projekt schnell und unkompliziert zu reaktivieren. Hier spielt die *Reproduzierbarkeit* der alten Arbeiten eine wesentliche Rolle und ein Buildsystem, dass diesen Punkt berücksichtigt, kann entscheidend zur Vereinfachung dieses Prozesses beitragen. Gängige Praxis ist die Archivierung kompletter Projektverzeichnisse, inklusive aller Binärpakete und temporären Dateien. Trotzdem kann die Reaktivierung eines so archivierten Projekts aus vielen Gründen fehlschlagen. Eleganter erscheint die Möglichkeit, den Stand des Buildsystems, das BSP sowie die Paketauswahl und ggf. die Toolchain zu sichern, um daraus jederzeit wieder ein Systemimage zu erzeugen. Problematisch sind die kurzen Entwicklungszyklen einiger Pakete, sodass der Quellco-

[8] Die dann oftmals ebenfalls als BSP bezeichnet wird.

de älterer Versionen schon nach kurzer Zeit nicht mehr verfügbar ist. Daher muss auch die Konservierung der Quellcodepakete erfolgen. Das Buildsystem sollte dem beispielsweise durch Unterstützung eines Paketcaches Rechnung tragen.

4.4.3 Patchmanagement

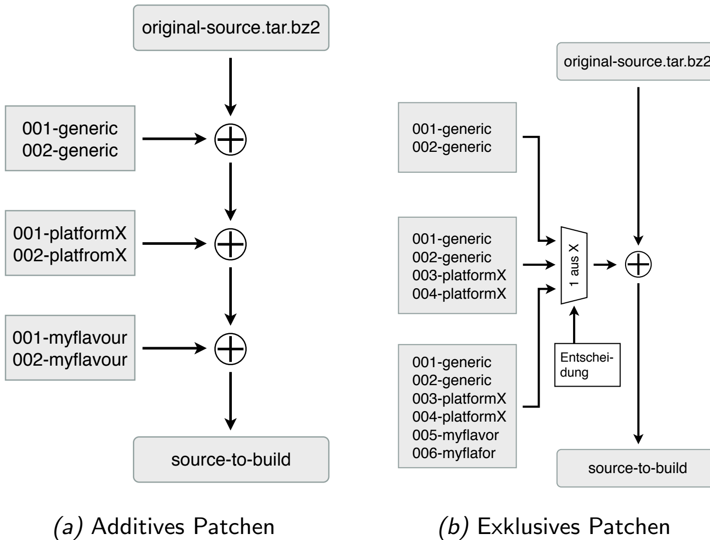
Wie bereits erwähnt, sind die Verfügbarkeit, die Qualität und der Umgang mit Patches für die im Buildsystem enthaltenen Pakete ein wesentliches Qualitätsmerkmal. Viele Softwareprojekte beachten die Besonderheiten des Cross-Kompilierens nicht oder nicht ausreichend und müssen daher vor dem Cross-Kompilieren durch Patches modifiziert werden.

Der Begriff *Patch* bezeichnet im Folgenden die wohl definierte Beschreibung der Änderungen zwischen zwei Versionen einer Datei oder eines Verzeichnisbaums. Üblicherweise wird dabei eine differentielle Form der Darstellung mit Addition für hinzugekommene und Subtraktion für nicht mehr vorhandene Teile genutzt. Weithin bekannte Arten, einen Patch zu erstellen, sind das Unix-Tool `diff` und die jeweiligen Umsetzungen von diff in den diversen *Versionsverwaltungssystemen*, wie `git diff` oder `svn diff`.

Die *Patch-Philosophie*, also Umgang und Herangehensweise an die Problematik der Patches, unterscheidet sich bei den verschiedenen Buildsystemen deutlich. So versucht beispielsweise PTXdist, mit wenigen Patches zu arbeiten und verstärkt die notwendigen Änderungen direkt in die Projekte zu tragen. OpenWrt hingegen ist für seine vielen Patches bekannt.

Beim *Patchmanagement* hingegen lassen sich zwei grundsätzliche Herangehensweisen erkennen, die in Abbildung 4.6 auf der nächsten Seite dargestellt sind. Dabei wird angenommen, dass der gewünschte Zielzustand ausgehend vom Ursprung durch die Anwendung von *N Patchsets* erreicht wird. Ein Patchset besteht dabei aus *K* Patches. Sowohl die Reihenfolge der Patchsets, als auch die Reihenfolge der Patches innerhalb eines Sets ist von Bedeutung und sollte üblicherweise nicht verändert werden.

Beim *additiven* Patchen (Abbildung 4.6a auf der nächsten Seite) werden Patchsets aus verschiedenen Quellen genutzt und nacheinander angewendet, wobei die Quellen und ihre Reihenfolge vom Buildsystem bzw. vom Nutzer festgelegt



(a) Additives Patchen

(b) Exklusives Patchen

Abbildung 4.6: Patchstrategien

werden. Der Vorteil dieser Methode ist die Möglichkeit, die Patchsets unabhängig voneinander zu pflegen und zu verwalten, was beispielsweise die Arbeit im Team erleichtert. Allerdings kann es dazu kommen, dass sich Abhängigkeiten zwischen Patchsets ergeben, die nicht ohne weiteres aufgelöst werden können. Ebenso können Änderungen an einem Patchset zu Fehlern in nachfolgenden Sets führen. Daher ist für die Verwaltung der Patchsets eine eigene Strategie bzw. Vorgehensweise notwendig. Beim *exklusiven* Patchen (Abbildung 4.6b) wird genau ein Patchset genutzt, das sich aus allen benötigten Patches zusammensetzt. Der Vorteil dieser Methode ist die bessere Kontrolle über die anzuwendenden Patches. Konflikte lassen sich einfacher erkennen und Abhängigkeiten können besser aufgelöst werden bzw. entstehen erst gar nicht. Nachteil ist der oft höhere Aufwand bei der Zusammenstellung und Aktualisierung des Patchsets.

4.4.4 Versionsmanagement

Eine konsequente und strukturierte Versionsverwaltung von Buildsystem, Toolchain, BSP und Systemkonfiguration ist für eine erfolgreiche Produktentwicklung und -pflege nahezu unabdingbar. Lange Zeit war Subversion [Nag05] der Quasi-Standard für die Versionsverwaltung. Mittlerweile kommen im Bereich der freien Buildsysteme mehrheitlich *dezentrale Versionsverwaltungssysteme (DVVS)*[9] (meist Git [Cha09] oder Mercurial [OSu09]) zum Einsatz. Ihr Vorteil ist das sehr bequeme und ressourcenschonende Verwalten von Zweigen (engl. *Branches*) sowie die Möglichkeit, auch ohne Verbindung zu einem Server neue Versionen zu erzeugen (*commit*) bzw. auf alte Stände zuzugreifen (*checkout*).

Versionsmanagement des Projekts

Wie bereits erläutert, besteht die Notwendigkeit, den Stand eines Projektes reproduzierbar zu sichern. Zum Zeitpunkt der Veröffentlichung einer Version werden daher die Stände des Buildsystems (X_{BS}), der verwendeten Toolchain (X_T), der Werkzeuge des Host-Betriebssystems (X_{HS}), des Board Support Packages (X_{BSP}) und der Paketauswahl (X_{SW}) festgehalten. Mit dem Tupel (X_{BS}, X_T, X_{HS}, X_{BSP}, X_{SW}) ist dann jederzeit eine Reproduktion der Firmware einer Version möglich. Ein unverändertes Buildsystem kann bereits durch seine Versionsnummer identifiziert werden, ebenso wie Toolchain und Host-Werkzeuge. Ein *Versionsverwaltungssystem (VVS)* scheint für die Konservierung eigener Konfigurationen, Anpassungen und Erweiterungen von Vorteil, was gleichzeitig das Nachvollziehen der Entwicklung der Software ermöglicht. Die jeweiligen Versionen zum Veröffentlichungszeitpunkt werden dann mit Hilfe der Mechanismen des VVS erfasst.

Eine viel beachtete Lösung für die Abdeckung des gesamten Entwicklungszyklus einer Software erläutert Vincent Driessen [Dri10] in seinem Artikel anhand des DVVS `Git`. Für die Verwaltung kleiner Projekte (wie BSP und Konfigurationen) erscheint das System jedoch zu komplex, sodass eine vereinfachte Variante vorgeschlagen wird (vgl. Abbildung 4.7 auf der nächsten Seite).

[9] engl.*distributed revision control system* bzw. *distributed version control system*

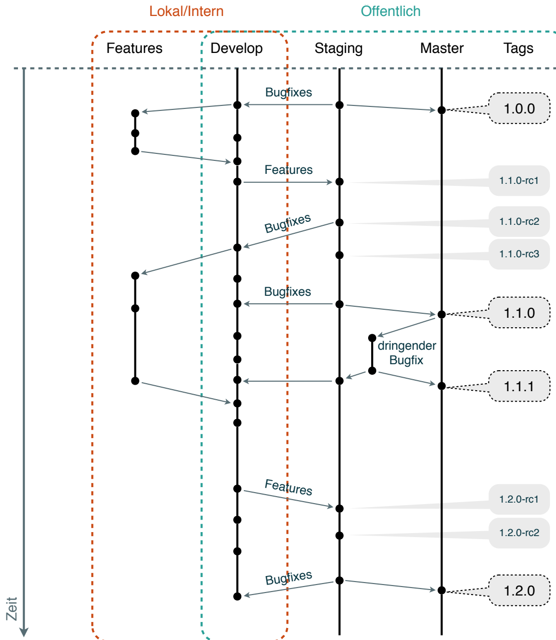


Abbildung 4.7: Versionsmanagement der Entwicklungsumgebung mit Git

Bei Git ist der *Master Branch* der Zweig, den ein Nutzer als erstes nach der Duplizierung der Arbeitskopie (per `git clone`) sieht. Dieser Zweig enthält daher nur die stabilen Versionen der momentanen Major-Release. Die Markierung (*Versionsnummern*) erfolgt mit Hilfe von *Tags*, die eindeutig einen Punkt im Versionsfluss markieren. Ein *Staging Branch* dient im genutzten Arbeitsablauf als Zweig für Betaversionen (*-rcX*). Beide Zweige und die zugehörigen Tags sind öffentlich zugänglich. Für die Weiterentwicklung der Software gibt es einen *Development Branch* (dt. Entwicklerzweig), der hauptsächlich den Entwicklern zur Synchronisation dient und daher nicht zwingend öffentlich zugänglich ist. Die Erarbeitung neuer Funktionen und Erweiterungen nutzt weitere Ebenen der Verzweigung, die jedoch nur lokal oder bilateral zwischen den verantwortlichen Entwicklern genutzt werden. Auch zur Beseitigung kritischer Fehler wird ein lokaler Zweig genutzt, der direkt aus einer

Version auf dem Hauptzweig entsteht und unmittelbar zur nächsten Release führt. *Feature Branches* leiten sich üblicherweise aus einer Entwicklerversion ab und führen immer in den Entwicklerzweig zurück, wo neue Funktionen dann durch andere Entwickler getestet und „stabilisiert“ werden. Aus dem Entwicklerzweig leiten sich auch die „Betaversionen“ für den Staging Branch ab, die nach Abschluss der Testphase zur Release auf dem Hauptzweig führen. Innerhalb der Testphase werden keine neuen Funktionen hinzugefügt, sondern lediglich Fehlerkorrekturen durch die Entwickler. Die Entwicklung in Feature- und Entwicklerzweig kann jedoch weitergehen. Fehlerkorrekturen auf den Zweigen weiter rechts im Bild müssen selbstverständlich auch in alle Zweige zu ihrer Linken übernommen werden.

Das vorgestellte Modell kann in gleicher Weise auch für eigene Erweiterungen am Buildsystem genutzt werden. Hierzu ist es lediglich um eine Ebene ganz links zu erweitern, die die Verbindung zu den externen Versionen darstellt (siehe dazu Anhang C auf Seite 231).

Versionsmanagement der Systemsoftware

Auch für die Verwaltung der Software auf den im Feld befindlichen Systemen ist eine reproduzierbare Versionsverwaltung notwendig. Dabei trifft man gerade im Bereich der eingebetteten Systeme auf eine Vielzahl von Strategien und Anforderungen, deren vollständige Darstellung an dieser Stelle nicht Ziel ist. In erster Linie erfolgt eine Diskussion der Methoden, die sich in Kapitel 6 ab Seite 145 wiederfinden.

Wie sich aus Abschnitt 4.1.2 bereits erkennen lässt, folgt die Verteilung des Festspeichers in einem Linux-System gewissen Grundsätzen. Der erste Bootloader liegt an der Stelle, die vom Prozessorsystem direkt nach dessen Initialisierung angesprungenen wird. Oft bietet bereits das Prozessorsystem einen kleinen, nicht flüchtigen Speicher dafür. Bei einer Implementierung mit mehreren Bootloadern liegen der/die weiteren Bootloader im Festspeicher des Systems. Der Kernel ist meist in einer angrenzenden Region gespeichert, an die sich das Wurzeldateisystem anschließt. Je nach Organisation kann es weitere Partitionen geben, die beispielsweise Nutzerdaten, Konfigurationen oder andere relevante Inhalte enthalten. In vielen eingebetteten Systemen sind dies beispielsweise der *device tree blob* [Lik08; LB08; FDT], die Konfiguration der

Bootloader und das initramfs. Abbildung 4.8 gibt eine Auswahl von häufig zu findenden Aufteilungen des nichtflüchtigen Systemspeichers wieder.

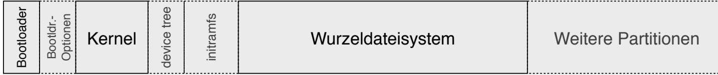
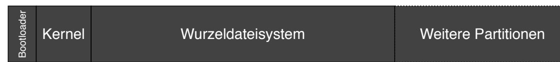


Abbildung 4.8: Typische Aufteilung des ROM-Speichers eines eingebetteten Systems

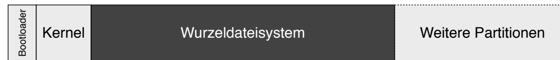
Der Bedarf nach und die Möglichkeit für Firmware-Updates ist bei nahezu jedem eingebetteten System von Interesse. Durch die zunehmende Anbindung der Geräte an Netzwerke besteht verstärkt die Notwendigkeit für sicherheitsrelevante Korrekturen. Gleichzeitig vereinfacht sich damit aber auch der Zugriff auf aktualisierte Softwareversionen.

Updates können *automatisch* ohne Nutzerinteraktion ablaufen, oder aber *interaktiv* unter Mitwirkung eines Nutzers/Administrators. Durch automatisierte Aktualisierungen kann auf einfache Weise sichergestellt werden, dass möglichst viele Geräte mit der gleichen Softwareversion arbeiten. Die Art des Updates kann in ihrer Granularität je nach Notwendigkeit variieren. Ein *vollständiges* Update schreibt eine komplett neue Version der Firmware (Bootloader, Kernel, Systemimage) in den Festspeicher des Systems. *Partielle* Aktualisierungen ersetzen nur einzelne Partitionen im Festspeicher. Das beschleunigt den Vorgang gegenüber einem vollständigen Update, da die zu übertragende Datenmenge geringer ist. Bei einem *block-basierten* Update werden nur einzelne Elemente bzw. Blöcke innerhalb einer Partition ersetzt. Abbildung 4.9 auf der nächsten Seite veranschaulicht diese drei Methoden. Je nach Art und Umfang der Aktualisierung ist im Anschluss ein Neustart des Systems erforderlich, sodass auch automatische Updates nicht immer unbemerkt verlaufen.

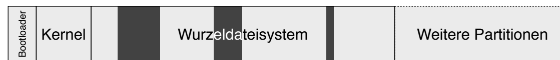
In jedem Fall muss ein System auch nach der Aktualisierung wie erwartet funktionieren. Im Vorfeld ist daher sicherzustellen, dass die neue Software korrekt arbeitet und alle im Feld befindlichen Systeme das Update korrekt durchführen können. Weiterhin müssen Methoden zur Gewährleistung der Datenintegrität sowohl während der Übertragung des Updates zum Zielsystem als auch nach dem Einspielen der neuen Software existieren. Ebenso besteht die



(a) Vollständiges Update



(b) Partielles Update



(c) Block-basiertes Update

Abbildung 4.9: Update-Strategien verschiedener Granularität

Notwendigkeit für eine Fallback-Strategie im Fehlerfall, die bei Systemen ohne Nutzerinteraktion entsprechend automatisiert werden muss. Ein vollständiges Update ist besonders kritisch. Wird nach dem Schreiben der neuen Firmware in den Festspeicher eine Inkonsistenz festgestellt, ist das System als nicht mehr boot- bzw. funktionsfähig zu betrachten. So lange der Update-Mechanismus noch funktionsfähig ist, kann jedoch durch erneutes Schreiben möglicherweise ein funktionsfähiges System hergestellt werden. Speziell für Lösungen ohne Nutzerinteraktion scheint dieses Vorgehen nicht geeignet. Eine Verbesserung erreicht man durch partielle Updates. Ist hinreichend Speicher vorhanden, können bei partiellen Updates freie Speicherbereiche mit der neuen Software beschrieben werden. Als Alternative ist die Nutzung eines kleineren, schreibgeschützten Speicherbereichs für ein Rettungs- oder Servicesystem[10] möglich. Dieses wird vor einem Update aktiviert und installiert die neue Software in die Systempartition. Nach Sicherstellung des erfolgreichen Einspielens der neuen Version in den Speicher, z. B. mit Hilfe von Prüfsummen oder durch Protokollierung eines Neustarts, wird das System angewiesen, die aktualisierten Bereiche zu nutzen. Beide Konzepte stellt Abbildung 4.10 auf der nächsten Seite dar. Partielle Updates sind prinzipiell nicht auf eine Partition beschränkt. Aus Stabilitätsgründen ist ein inkrementelles Vorgehen jedoch zu bevorzugen.

[10] Hierfür eignet sich auch ein entsprechend vorbereitetes initramfs.

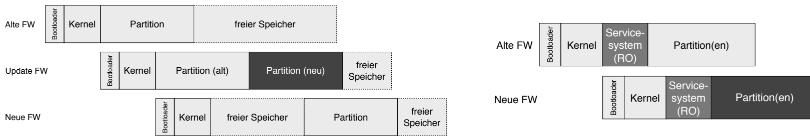


Abbildung 4.10: Konzepte für die partielle Aktualisierung

Vollständige und partielle Updates orientieren sich an der Systemsoftwareentwicklung mit Buildsystemen, da hier meist Bootloader, Kernel und Partitionsabbilder das Ergebnis des Erstellungsvorgangs sind. Blockweise Updates orientieren sich vornehmlich am Vorgehen der Distributionen, die die einzelnen Softwarekomponenten in Paketen verwalten. Aufgrund der strukturellen Vorteile bieten aber auch viele Buildsysteme das Erstellen von Paketen an, die dann mit Werkzeugen wie `ipkg` oder `opkg` auf dem Zielsystem installiert werden. Vorteil der paketorientierten Versionsverwaltung ist die feine Granularität, und damit verbunden das schnelle und zielgerichtete Erneuern veralteter Software. Die kleine zu übertragende Datenmenge stellt nur geringe Ansprüche an die Datenverbindung. Das Risiko für Fehler beim Übertragen der Daten zum System sowie beim Schreiben der Daten in den Festspeicher ist ebenfalls gering. Meist können die Pakete für verschiedene Versionen einer Software im Festspeicher des System verbleiben, sodass im Bedarfsfall auch das Wiederherstellen eines älteren Standes ohne erneute Datenübertragung möglich ist. Schwierig bleibt der Umgang mit den Komponenten, die kritische Bereiche des Systems betreffen, darunter der Kernel, Systemdienste (z. B. udev oder das init-System) und die Paketverwaltung.

Einige Teile des Systems werden nicht durch die Paketverwaltung abgedeckt. Hauptsächlich betrifft dies Konfigurationsdateien der Nutzer[11] und des Systems. Für deren Verwaltung bietet sich ein DVVS an. Es erlaubt nicht nur, wie bereits erläutert, eine strukturierte Versionsverwaltung inklusive Rückkehr zu älteren Ständen, sondern übernimmt auch die gesicherte Datenübertragung zum Zielsystem. Ein entsprechendes Szenario für die Nutzung von DVVS (mit und ohne Nutzerinteraktion) wird in Kapitel 6 ab Seite 145 vorgestellt.

[11] Der Autor geht davon aus, dass, den allgemeinen Standards und Richtlinien folgend, neben dem Administratorkonto auch entsprechende Nutzerkonten für verschiedene Aufgaben eingerichtet werden.

4.5 Spezifika bei heterogenen rekonfigurierbaren Systemen

Die effiziente Nutzung der vielfältigen Architekturen von heterogenen rekonfigurierbaren Systemen (vgl. Kapitel 3 ab Seite 71) hängt direkt an der Verfügbarkeit von APIs und Werkzeugen für den Zugang zu den gebotenen Funktionen. Das umfasst die Konfiguration ebenso wie die Nutzung der Systeme. Verschiedene Arbeiten widmen sich der Integration rekonfigurierbarer Architekturen in Betriebssysteme, sei es Linux (z. B. [So07; Str+09], [NM09, S. 370 ff.]) oder RTOS (z. B. [Wal05]). Allerdings erfordern die Ansätze entweder einen tiefen Eingriff in den Betriebssystemkern, zusätzliche Hardware für die Verwaltung der rekonfigurierbaren Ressourcen oder sind durch ihr Konzept sehr fest an eine bestimmte Architektur oder Kombination aus Hard- und Software gebunden. Viele Arbeiten fokussieren sich außerdem auf die Migration von Prozessen oder Prozessteilen (Threads, Tasks) zwischen RM und Betriebssystem in Szenarien mit dynamisch-partieller Rekonfiguration. (z. B. [LP07]).

Anhand der real verfügbaren Architekturen wird jedoch deutlich, dass nicht dpR, sondern die effiziente Nutzung der Rekonfiguration an sich von praktischem Interesse ist. In vielen Fällen wird ein FPGA als flexibler Beschleuniger für eine oder wenige Aufgaben genutzt und die Möglichkeit der (Re-) Konfiguration dient der (nachträglichen) Beseitigung von Fehlern, der Verbesserung der Funktion oder einer (verhältnismäßig) seltenen Änderung der Funktionalität. Die Kommunikation zwischen Hostprozessor und den RMs erfolgt in diesen Anwendungen häufig über einen Systembus, die Konfiguration über eine dedizierte Verbindung oder ebenfalls über den Bus. Die Softwarearchitektur muss diesem Anwendungsfall entsprechen und die aus dem Linux bekannten Konzepte unterstützen.

4.5.1 Kommunikation mit dem RM

Wie bereits in Abschnitt 4.1.2 dargestellt, gibt es verschiedene Wege zur Kommunikation mit angeschlossener Peripherie. Der zu wählende Weg wird durch die genutzte Kommunikationsarchitektur (vgl. Abschnitt 3.2.2) beeinflusst, die

ggf. einzelne Wege von vornherein ausschließt. Grundsätzlich muss die Software den Anforderungen an Bandbreite und Latenz ebenso genügen wie die Hardware.

Existiert ein dedizierter Treiber, erfolgen Zugriffe je nach dessen Implementierung über eine Gerätedatei in `/dev` oder das Netzwerk-API. Statusinformationen können in diesem Fall über die virtuellen Dateisysteme des Kernels (proc, sysfs, configfs, debugfs) ausgetauscht werden. Gib es keinen passenden Treiber, kann die Kommunikation bei im Adressraum eingeblendeten Schnittstellen auch über die spezielle Gerätedatei `/dev/mem` erfolgen. Allerdings erfordert die Nutzung der Schnittstelle volle Systemrechte[12], was nicht immer mit den Sicherheitsrichtlinien eines Systems vereinbar ist. Weiterhin besteht über diese Schnittstelle unter Umständen Zugriff auf den gesamten Adressbereich des Systems, sodass Programmfehler bei der Nutzung von `/dev/mem` fatale Folgen haben. Für einige standardisierte Schnittstellen wie beispielsweise SPI oder I^2C gibt es bereits Treiber im Kernel und über *libusb* können USB-Geräte aus dem User Space genutzt werden.

Offensichtlich ist die Nutzung von `/dev/mem` mit dem größten Risiko verbunden, bietet aber den schnellsten Weg der Implementierung. Die im User Space laufende Anwendung erlaubt die Nutzung aller Entwicklungswerkzeuge, muss jedoch mit vollen Systemrechten ausgeführt werden. Ein Gerätetreiber im Kernel erscheint als die sauberste Lösung, ist aber mit dem entsprechenden Aufwand verbunden (vgl. Abschnitt 4.1.2). Als Alternative kann ein UIO Treiber genutzt werden. Beim Einsatz standardisierter Schnittstellen wie SPI, I^2C oder USB bietet sich die Nutzung der korrespondierenden Mechanismen an.

Außerdem muss die aus der Rekonfiguration erwachsende Dynamik beachtet werden, was zusätzlichen Aufwand bei der Implementierung und Verwaltung verursacht (vgl. Abschnitt 3.5.2) und bei der Treiberentwicklung berücksichtigt werden muss. Konkrete Umsetzungen für den Umgang mit rekonfigurierbaren Architekturen im Linux präsentiert Kapitel 6 ab Seite 145.

[12] sog. *root*-Rechte, also uneingeschränkten Zugriff auf *alle* Systemressourcen.

4.5.2 Umsetzung der Konfiguration

Wie bereits im vorhergehenden Abschnitt erwähnt, unterstützt Linux einige Basiskonzepte für die Kommunikation mit externen Komponenten. Diese bilden auch die Grundlage für die Konfiguration, die im Grunde auch nur eine Form der Kommunikation von Betriebssystem und RM darstellt. Der zu beschreitende Weg ergibt sich in erster Linie aus der verwendeten Konfigurationsarchitektur (vgl. Abschnitt 3.2.3), die in den meisten Fällen bereits die Nutzung von Schnittstelle und Protokoll für das Auslösen eines Konfigurationsvorganges und die Übertragung der Konfigurationsdaten festlegt. Weiterhin spielt das Konfigurationsmanagement eine Rolle (vgl. Abschnitt 3.5.2).

Ob die Konfigurationsdaten an einen PROM oder direkt an das RM übertragen werden, ist nicht von Belang. Die Nutzung von *Memory Mapped I/O* per `/dev/mem` und auch eine Umsetzung per UIO erscheinen für die Übertragung größerer Datenmengen, wie dies bei der Rekonfiguration meist der Fall ist, nicht zielführend. Aufgrund der umzusetzenden Protokolle ist auch der Rückgriff auf verfügbare Treiber, beispielsweise zur Implementierung eines Konfigurationsprotokolls im User Space mit Hilfe von GPIO-Pins, nicht immer möglich. Somit sind Lösungen auf Basis von Gerätedateien für die Übertragung der Konfigurationsdaten die geeignete Wahl. In vielen Fällen wird es sich dabei um spezielle Implementierungen handeln, die sich direkt an der Konfigurationsarchitektur orientieren. Alternativ ist es möglich, einen Treiber für die Kommunikation mit einem RM auch für die Übertragung der Konfigurationsdaten zu verwenden. Ausschlaggebend ist die Umsetzung der im Linux-Umfeld üblichen APIs.

In der Literatur finden sich nur wenige allgemeine Ansätze für die Konfiguration von hrS. In vielen Fällen erfolgt die Umsetzung system- und anwendungsspezifisch, wobei sich einige Methoden als gängige Praxis etabliert haben.

Eine Variante ist die Integration von Konfigurationsdaten direkt in das Kompilat einer Software. Dieses Vorgehen findet sich sowohl bei der Nutzung des Bootloaders zur Konfiguration (frühe Konfiguration) als auch bei der Umsetzung im Kernel und im User Space. Die Lösungen sind typischerweise direkt auf das System zugeschnitten und kaum portabel. Anwendung findet dies beispielsweise bei Systemen, die eine Kombination aus FPGA und PROM als Alternative zum *Application Specific Integrated Circuit (ASIC)* nutzen und daher

die Konfiguration nur im Zuge der Systempflege erneuern (Bugfixes, kleinere Anpassungen). Der Bitstrom ist direkt in die für das Update genutzte Software integriert. Somit muss nur eine Datei verteilt werden, die dann die Rekonfiguration durchführt. Nach Möglichkeit wird hier auch auf spezielle Treiber verzichtet und stattdessen auf Bordmittel des verwendeten Linux-Kernels zurückgegriffen. Ähnliche Vorteile bietet die Integration des Bitstromes in die Anwendung, solange die Nutzung des RMs exklusiv erfolgt. Die Integration in den Kernel – egal ob permanent oder als ladbares Modul – hingegen muss gut abgewogen werden, da durch die Bitstromgröße auch der Kernel signifikant vergrößert wird. Dem gegenüber steht lediglich der Verzicht auf den Datentransfer vom User Space in den Kernel Space.

Der Aufwand für die Implementierung dedizierter Treiber zur Kommunikation und Konfiguration eines RM rechtfertigt sich trotz der gebotenen Vorteile meist erst ab einer bestimmten Nutzungshäufigkeit oder wenn die gewählte Architektur dies erforderlich macht. Durch die Umsetzung im Kernel Space sind schnell Geschwindigkeitsvorteile zu erzielen. Ein Management ist damit nicht zwingend verbunden, wenngleich Kernel-Module Zugriffe überwachen und durch Funktionen wie blokierendes Lesen/Schreiben steuern können.

Spezielle Zugriffs- und Managementsysteme gehen meist mit hohem Implementierungs- und Wartungsaufwand einher. Daher konnten sich selbst sehr interessante Lösungen wie das von Strunk et. al. vorgestellte ACCFS (ACCelerator File System, vgl. [Str+09]) oder die Umsetzung über den *Device Tree* (vgl. [NM09]) nicht etablieren.

4.6 Nutzung plattformunabhängiger Sprachen in eingebetteten Systemen

Im Bereich der eingebetteten Systeme wird häufig auf Programmiersprachen wie C oder C++ zurückgegriffen, die erst nach einer architekturspezifischen Kompilation auf dem Zielsystem verwendet werden können. Dem Vorteil des meist sehr effizienten Programms stehen einige Nachteile gegenüber. Für die Übersetzung müssen eine geeignete Toolchain (vgl. Abschnitt 4.4.1) bzw. ein hinreichend leistungsfähiges Entwicklungssystem bereitstehen. Beim Cross-Development kommt es aufgrund der verschiedenen Architekturen schnell zu

Fehlern. Bekannte Probleme sind beispielsweise die Byte-Reihenfolge (engl. *Endianness*) und nicht erfüllte Abhängigkeiten auf dem Zielsystem.

Mit der zunehmenden Leistungsfähigkeit der eingebetteten Systeme setzen Hersteller und Entwickler daher verstärkt auf plattformunabhängige Entwicklungssysteme. Dazu zählen Ansätze wie Java [Ull12], bei denen ein plattformunabhängig kompilierter Byte-Code zusammen mit einer plattformabhängigen Laufzeitumgebung genutzt wird. Eine weitere Erleichterung bieten Skriptsprachen, die durch Vereinfachungen, beispielsweise bei der Typisierung, die schnelle Umsetzung von Problemlösungen erlauben. Hierzu zählen direkt interpretierte Kommandosprachen[13] ebenso wie komplexere Umsetzungen, deren Interpretation/Kompilation beim ersten Programmstart erfolgt (z. B. Python [KE12]). Die Plattformunabhängigkeit der Sprachen erlaubt eine sehr komfortable Entwicklung auf leistungsstarken Desktopsystemen und die anschließende Nutzung des gleichen Quellcodes auf der Zielplattform. Somit steht dem Leistungsverlust eine deutlicher Gewinn an Komfort gegenüber. Sowohl Kommandozeileninterpreter als auch einige Skriptsprachen ermöglichen die Nutzung von plattformabhängigem Binärcode, beispielsweise in Form von Bibliotheken, was je nach Anforderung auch eine Anwendung bei zeitkritischen oder hardwarenahen Problemen erlaubt.

In Kapitel 6 ab Seite 145 finden sich verschiedene Beispiele für den Einsatz von Python als Programmiersprache für Prototypen und Machbarkeitsstudien.

4.7 Zusammenfassung und Diskussion

In diesem Kapitel erfolgte eine detaillierte Auseinandersetzung mit dem Betriebssystem Linux, wobei weniger technische Faktoren im Vordergrund stehen, sondern die für den Systementwurf interessanten Zusammenhänge zwischen den Komponenten des Betriebssystems. Die im Verlauf des Kapitels bereits begonnenen Diskussionen werden an dieser Stelle im Anschluss an eine kurze Zusammenfassung noch einmal aufgegriffen und – soweit möglich – zu einem Ende geführt.

[13] Beispielsweise der `Bash` oder der `Zsh`

4.7.1 Zusammenfassung

Das Kapitel startet mit der Einführung von Begriffen und der kompakten Darstellung von technischen Grundlagen. Neben allgemein bekannten Punkten wie dem Systemaufbau, den Kernel-Schnittstellen, Werkzeugen zur Erstellung von Software und dem Ablauf des Systemstarts wird in Vorbereitung auf die weiteren Themen auch die Abhängigkeit zwischen Kernel, Bibliotheken und Anwendungsprogrammen dargestellt.

Es folgt eine ausführliche Diskussion zum Thema Distributionen, die sich besonders auf die Zusammenhänge und Abläufe beim Software-, Paket- und Versionsmanagement im Linux-Umfeld konzentriert. Damit leistet die vorliegende Arbeit einen Beitrag zur systematisierten Darstellung der zwar etablierten, aber kaum dokumentierten Abläufe bei der Erstellung und Pflege von Linux-Distributionen.

Auch bei der nachfolgenden Übertragung der Erkenntnisse auf eingebettete Systeme liegt der Fokus nicht auf technischen Details, sondern auf den Gegebenheiten und Abläufen bei der Erstellung und Pflege von Linux-Systemen für eingebettete Systeme. Dabei ermöglichen die Erfahrungen des Autors eine detaillierte Diskussion zu den wesentlichen Themen: Werkzeuge, Buildsysteme sowie Patch- und Versionsmanagement. Diese Diskussion dient wiederum der systematisierten Darstellung etablierter, aber kaum dokumentierter Abläufe.

Abschnitt 4.5 und Abschnitt 4.6 widmen sich der Lösung zweier technischer Probleme. Abschnitt 4.5 verbindet die Ausführungen aus Abschnitt 3.5 mit den spezifischen Möglichkeiten zur Implementierung von Kommunikation und Konfiguration im Linux-Betriebssystem. Abschnitt 4.6 greift die Schwierigkeiten beim Cross-Development noch einmal auf, zeigt einen derzeit oft anzutreffenden Lösungsansatz und geht kurz auf dessen Randbedingungen ein. Plattformunabhängige Sprachen erlauben eine schnelle und meist einfachere Implementierung von Anwendungen. Diesem Vorteil stehen der erhöhte Ressourcenbedarf und die erhöhte Latenz gegenüber.

4.7.2 Diskussion der Erkenntnisse

Natürlich haben technische Faktoren einen wesentlichen Anteil an der Entscheidung für Linux als Betriebssystem für eingebettete Systeme. „Wir nutzen Linux“ steht stellvertretend für den Einsatz eines aktiv gepflegten, in vielen Bereichen eingesetzten, ausgereiften und erprobten Systems. Ebenso sind allerdings bestimmte Randbedingungen an den Einsatz von Linux geknüpft. Die zugrundeliegende Hardware muss die minimalen Voraussetzungen für das Betriebssystem erfüllen und sollte im Idealfall bereits von Kernel und Compiler unterstützt werden. Mit der Entscheidung für Linux bindet sich der Entwickler an die damit verbundene Softwarearchitektur aus Bootloader, Linux-Kernel, Bibliotheken und Anwendungen. Die Entwicklung von Software sollte sich dieser Architektur anpassen, um von ihren Vorteilen zu profitieren. Gleichzeitig begibt sich der Entwickler in das Linux-Ökosystem und die damit verbundenen Philosophien. Dies betrifft zum einen die verwendeten Lizenzen (vgl. Abschnitt 1.3) und die damit verbundenen Verpflichtungen, zum anderen die verschiedenen Prozesse, Gepflogenheiten und Standards. Dazu zählen u. a. die Verwendung bestimmter Werkzeuge unabhängig von ihrem Entwicklungsstand, ihrer Eignung und Verbreitung, der Community-getriebene Softwareentwurf oder Distributionen und das mit ihnen verbundene Paket-, Versions- und Qualitätsmanagement.

Beim Einsatz von Linux in eingebetteten Systemen sind bereits im Vorfeld einige Entwurfsentscheidungen zu treffen. Eine dieser Entscheidungen, die meist auch Folge- und Schwestersysteme betrifft, ist die Frage, ob eine Distribution oder ein individuell abgestimmtes Linux basierend auf einem Buildsystem zum Einsatz kommt.

Für den schnellen Einstieg in das Gebiet der eingebetteten Systeme mit Linux als Betriebssystem bietet sich der Einsatz einer der verfügbaren Linux-Distributionen an. Dem geringen initialen Aufwand, der durch den Distributor gewährleisteten Softwarequalität und dem abgestimmten Portfolio aus Software und Werkzeugen stehen die Einschränkungen durch die vorgefertigten Pakete und die Abhängigkeit vom Distributionsanbieter gegenüber.

Mit der Bindung an eine Distribution folgt man den Vorgaben des Distributors bezüglich Software- und Werkzeugauswahl sowie Paketabhängigkeiten und ist bei Updates den Veröffentlichungszyklen der Distribution unterworfen. Die

auf ein breites Anwendungsspektrum ausgelegten Distributionen benötigen zudem in der Regel mehr RAM und Festspeicher. Aufgrund ihrer Ausrichtung implementieren sie zudem Funktionen, die im Desktop- und Serverbereich notwendig, bei eingebetteten Systemen jedoch durchaus hinderlich sind. Tiefe Eingriffe in die Basissoftware einer Distribution ebenso wie das Ersetzen von Paketen oder Werkzeugen sind zwar möglich, führen aber zu einem ungleich höheren Aufwand, der den Beweggründen zur Nutzung einer Distribution entgegensteht.

Dem gegenüber steht die große Flexibilität der Buildsysteme, die als Metadistribution den Nutzer in die Position des Distributors bringen. Den Freiheiten bei der Wahl von Funktionsumfang, Paketen, Abhängigkeiten und Werkzeugen und den damit verbundenen Möglichkeiten (z. B. benötigter Speicherplatz) stehen hier der erhöhte initiale und laufende Aufwand, die Verantwortung für die Kontrolle und Pflege der gewählten Software und die Verwaltung der Infrastruktur für Erstellung und Verbreitung der Software gegenüber.

Diesen offensichtlichen Punkten gliedern sich weitere Fakten an, die ebenfalls direkt oder indirekt mit der Wahl des Basiskonzepts korrelieren. Hinzu kommen allgemeine Probleme der derzeitigen Entwicklung im Bereich der freien Software.

Tabelle 4.2: Systemgröße eines Raspberry Pi bei verschiedenen Ansätzen

Ansatz	Größe (MebiByte)	Anzahl Pakete
Raspbian „Wheezy“	1828	683
Angepasstes Debian „Wheezy“	216	122
PTXdist	36	86

Bei den Paketen der Distributionen sind neben den notwendigen auch die optionalen Abhängigkeiten zu anderen Paketen fest bzw. dynamisch vorgegeben. Eine Änderung ist zwar durch den Ersatz des Pakets möglich, die Auswirkungen auf andere Software der Distribution aber nur durch intensive Tests abschätzbar. Auch bei Buildsystemen kann es je nach Umsetzung vorgegebene Abhängigkeiten aufgrund optionaler Funktionen geben. Eine Änderung ist hier jedoch mit geringerem Aufwand (bei gleichem Risiko) verbunden. Diese

nicht immer benötigten Funktionen erhöhen nicht nur den Bedarf an Festspeicher unnötig, sondern bringen auch zusätzliche Software – und damit zusätzliche Fehlerquellen – in das System. Tabelle 4.2 auf der vorherigen Seite zeigt an einem ausgewählten Beispiel den Einfluss dieses Punktes. Die Firmware eines Raspberry Pi sollte jeweils die gleichen Aufgaben erfüllen, wurde aber einmal mit dem Standard Raspberry Pi Debian Linux (Raspbian) implementiert, einmal mit einer optimierten Raspbian-Version und einmal mit einem per PTXdist erstellten Linux-System.

Nicht alle Projekte der *Free and Open Source (FOS)*-Szene folgen den in Abschnitt 4.2.1 skizzierten Vorgaben und Abläufen. Dies erschwert das Versionsmanagement und die Qualitätskontrolle. Ebenso erscheint das Schritthalten mit den Projekten, die eine sehr hohe Veröffentlichungsfrequenz haben, kaum realisierbar. Einige Ansätze, wie beispielsweise der von Weigelt in [Wei10a; Wei10b] beschriebene, versuchen dies zu mildern, werden jedoch nicht allgemein anerkannt. Die daraus erwachsenden Herausforderungen betreffen sowohl die im Linux-System verwendete Software als auch die für ein Buildsystem notwendige Umgebung. Im Rahmen des Produktmanagements sollten sicherheitskritische Updates der verwendeten Softwareprojekte Dritter auch in eigene Systeme übernommen werden. Daher muss die verwendete Infrastruktur in allen Teilen darauf ausgelegt sein – ein Umstand der weder bei der Verwendung von Distributionen noch bei Buildsystemen automatisch gegeben ist.

Gleiches gilt für das Versionsmanagement der Firmware, das aufgrund der wachsenden Vernetzung eingebetteter Systeme an Bedeutung gewinnt. Zugehörige Strategien werden daher ein Teil der Systemspezifikation und müssen im Entwurfsfluss berücksichtigt werden. Ein Update sollte bei allen potentiellen Empfängern in der erwarteten Weise funktionieren. Dementsprechend muss immer ein Weg vom alten zum neuen Zustand existieren. Dabei kann sich der Vorteil feingranularer Ansätze schnell erschöpfen, da hier die Reihenfolge von Updates oft von größerer Bedeutung ist. Partielle Updates stellen sicher, dass die neu eingespielte Firmware in allen Punkten dem erwarteten Zustand entspricht.

Abschließend lässt sich feststellen, dass die Umsetzung eines eingebetteten Systems mit Linux-Betriebssystem immer ein individuelles Vorgehen bei Spezifikation und Entwurf erfordert. Eine wesentliche Reduktion des Entwurfs-

aufwands durch die Nutzung von Distributionen ist nicht zu erkennen. Wird das Zielsystem ähnlich einem Desktop- oder Serversystem verwendet, bieten Distributionen einen vielversprechenden Ansatz, da sie grundsätzlich auf derartige Szenarien abgestimmt sind. Je spezieller das System auf seine Anwendung zugeschnitten wird, desto mehr überwiegen die Vorteile von Buildsystemen.

5 Spezifikation, Test und Verifikation von Linux-Systemen

Mit der Entscheidung für Linux als Betriebssystem eröffnet sich eine breite Basis an vorhandener Software, mit der die gewünschten Funktionen umsetzbar sind. Die Herkunft und Qualität dieser Software variiert entsprechend den Darstellungen in Kapitel 4 aufgrund ihrer Entstehungsgeschichte und der damit verbundenen Prozesse. Dies spricht jedoch nicht gegen den Einsatz dieser Software. Allerdings müssen die genutzten Pakete sorgfältig ausgewählt und so intensiv wie möglich getestet werden, um Funktionsumfang, Stabilität und Funktionalität zu sichern, was einen beträchtlichen Arbeits- und Zeitaufwand bedeutet.

5.1 Spezifikation von Linux-Systemen

Die in Abschnitt 2.2 vorgestellten Entwurfsmodelle sind auch für eingebettete Systeme mit Linux-Betriebssystem gültig. Allerdings beeinflusst die Entscheidung für die Verwendung von Linux die weiteren Entwurfsschritte nachhaltig. Bereits bei der HW/SW-Partitionierung ist zu berücksichtigen, dass ein Linux-System bestimmte Anforderungen an Prozessorsystem und Speicher stellt. Daraus ergibt sich die Frage, welche Funktionen mit der durch das Betriebssystem geforderten Hardware erfüllbar sind, wo zusätzliche Hardware notwendig ist und wo Aufgaben in die Software verlagert werden können.

Bei der Implementierung eigener Software kommt das in Abschnitt 2.6.3 beschriebene Vorgehen zum Tragen. Die Software sollte sich allerdings an

der späteren Betriebssystemumgebung orientieren, also verfügbare Bibliotheken, typische Schnittstellen und gängig Programmierparadigmen berücksichtigen.

Eine besondere Herausforderung stellen Spezifikation und Entwurf des Linux-Systems dar. Ein streng formales Vorgehen bei der Spezifikation erscheint aufgrund der Komplexität und des Funktionsumfangs kaum möglich. Da die Granularität bei der Umsetzung auf Kernel, Bibliotheken und Anwendungen begrenzt ist, stellt die Spezifikation letztlich nur den minimalen Funktionsumfang dar. Der reale Funktionsumfang dürfte nahezu immer darüber liegen, da die zur Umsetzung der Spezifikation gewählten Bibliotheken und Anwendungen weitere Funktionen bieten. Während dieser Umstand aus Sicht der Spezifikation nicht notwendigerweise von Nachteil ist, erschwert er Test und Verifikation, da hier auch das Fehlverhalten aufgrund zusätzlicher Funktionalitäten ausgeschlossen werden muss.

Ein weiterer, kritischer Punkt ist die enorme Anzahl an Softwarepaketen und Konfigurationsoptionen. Die Gentoo Distribution bietet beispielsweise mehr als 17.000 Pakete zur Auswahl, Debian „wheezy“ sogar über 48.000 und PTXdist als schlankes Buildsystem bereits über 850. Die Anzahl von Konfigurationsoptionen für ein Linux-System ist ebenso üppig. Wie Sincero et. al. und She et. al. in ihren Studien [She+10; Sin+10] feststellen, hat der Linux-Kernel weit über 8.000 Konfigurationsoptionen und PTXdist bietet für seine nur 850 Softwarepakete bereits mehr als 4.500 Optionen. Somit umfasst der Entwurfsprozess mehrere Auswahlschritte, um aus dem Pool verfügbarer Pakete und Optionen ein der Spezifikation entsprechendes Linux-System zu „konfigurieren“. Speziell gilt es, einen Kernel mit dem notwendigen Funktionsumfang zu konfigurieren und aus der vorhandenen Auswahl jene Pakete mit ihren jeweiligen Funktionsoptionen zu bestimmen, die die Spezifikation erfüllen und möglichst wenig zusätzlichen Verifikationsaufwand verursachen.

Die persönliche Erfahrung des Autors zeigt, dass Spezifikation und Umsetzung von Linux-Systemen einigen allgemeinen Abläufen folgen, um die Komplexität von Entwurf und Test beherrschbar zu gestalten. Häufig erfolgt die Spezifikation der Software in Form von (umgangssprachlichen, nicht formalen) Funktionsbeschreibungen des Zielsystems[1], beispielsweise:

[1] z. B. im Rahmen des Lastenhefts

- bezieht seine IP-Adresse per *DHCP*,
- bietet Zugriff via *HTTP* oder
- prüft zyklisch, ob …
- usw.

Aus dieser Beschreibung ergibt sich der minimale Funktionsumfang, der die Basis für die Auswahl geeigneter Softwarepakete darstellt. Dabei greifen Entwickler nach Möglichkeit immer auf die gleichen Pakete zurück, was gleich mehrere Vorteile bietet. Zum einen ist der Umgang mit dem Paket geläufig, z. B. (optionale) Funktion, Konfiguration, Abhängigkeiten. Zum anderen besteht aus der Erfahrung heraus eine zunehmende Sicherheit, dass das Paket wie erwartet funktioniert. Test und Verifikation konzentrieren sich somit zusätzlich zum Gesamtsystem auf die Auswirkungen der weniger bekannten Anwendungen und Bibliotheken.

Ein ähnliches Verhalten ist im Übrigen auch bei der Auswahl der Hardware zu beobachten. Auch hier wird nach Möglichkeit auf bekannte Systeme oder wenigstens bekannte Prozessorsysteme und Hardwarekomponenten zurückgegriffen, da so beispielsweise die Konfiguration der Kernels ebenfalls unter Berücksichtigung der vorhandenen Erfahrungen erfolgen kann.

5.2 High-Level Spezifikation einer Linux-Firmware

Distributionen und Buildsysteme bieten eine breite Auswahl an Software, sodass für einzelne Aufgaben meist mehrere, gleichwertige Lösungen existieren. Dies führt bei beiden zu einer nahezu unbeherrschbaren Anzahl an Paketen, unter denen der Entwickler wählen muss. Die gebotenen Auswahlwerkzeuge bieten dabei nur mäßige Unterstützung, wobei Distributionen hier meist detailliertere Informationen zu Abhängigkeiten (obligatorisch bzw. starke und schwache optionale) und Konflikten (Paket A schließt Paket B aus) bieten. Buildsysteme erfordern ein deutlich höheres Maß an Verständnis für die Interaktion der Pakete innerhalb des Systems, um im Fehlerfall auf Ursachen auch bei der Softwareauswahl zu schließen.

Das in Abschnitt 5.1 beschriebene Vorgehen erinnert an eine Art Expertensystem, das basierend auf Erfahrungen und Anforderungen zur Lösung eines Problems führt. Die bereits erwähnten Betrachtungen verschiedener Distributionen und Buildsysteme zeigten weiterhin, dass speziell bei den Buildsystemen viele Parallelen in der Umsetzung sowie der Speicherung der Konfigurationen bestehen. Daher entstand die Idee, die Spezifikation eines Linux-Systems durch ein Werkzeug zu vereinfachen und zugleich die Nutzung verschiedener Buildsysteme zu ermöglichen. Das erlaubt in Abhängigkeit von der jeweiligen Anwendung, die Nutzung der speziellen Stärken einzelner Systeme. Diese Basisidee ist in Abbildung 5.1 veranschaulicht.

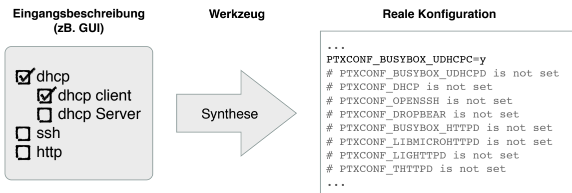


Abbildung 5.1: Basisidee für die vereinfachte Spezifikation einer Linux-Firmware

Die *Eingangsbeschreibung* ist hierarchisch aufgebaut, bleibt aber auf die Beschreibung von Systemfunktionalität beschränkt. Über eine flache Hierarchie lassen sich Details direkt einpflegen. Eine weitere Verfeinerung ist jedoch nicht vorgesehen. Diese Darstellung ermöglicht die Erfassung der eingangs erwähnten minimalen bzw. gewünschten Systemanforderungen auf hohem Abstraktionsniveau, z. B. im Gespräch mit möglichen Kunden ohne tiefgreifenden technischen Hintergrund. Die Umsetzung dieses als *Frontend* bezeichneten Teils ist auf vielfältige Weise vorstellbar. Für eine spätere Wiederverwendung und als semiformales „Spezifikationsdokument" wird die Eingangsbeschreibung gespeichert.

Zentrale Komponente ist das Werkzeug, welches die Synthese der Eingangsbeschreibung übernimmt. Nach der Synthese liegt eine Konfiguration vor, die direkt für die Nutzung mit einem Buildsystem oder einer Distribution geeignet ist. Abbildung 5.2 auf der nächsten Seite zeigt die gewählte Architektur für das Synthesewerkzeug. Alternative Konzepte erläutert Anhang E.1 auf Seite 235.

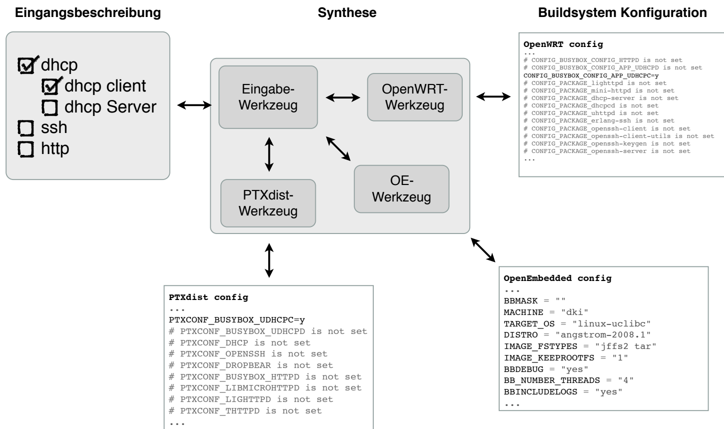


Abbildung 5.2: Architektur des Synthesewerkzeugs

Das *Eingabewerkzeug* liest die Eingangsbeschreibung und stellt sie über ein definiertes API zur weiteren Verarbeitung zur Verfügung. Somit sind alle weiteren Verarbeitungsschritte von der Art der Eingangsbeschreibung unabhängig. Für die einzelnen Zielsysteme gibt es dedizierte Werkzeuge, die jeweils systemspezifisch umgesetzt sind. Vorteil dieser Architektur ist die Möglichkeit, die Besonderheiten der Zielsysteme besser abbilden zu können. Die Notwendigkeit ergibt sich daraus, dass die Systeme nicht nur verschiedene Darstellungsformen für ihre Konfigurationen nutzen, sondern auch verschiedene Aufteilungen und Strukturen.

Die Umsetzung der zielsystemspezifischen Werkzeuge erfordert ein hohes Maß an Verständnis für das adressierte Buildsystem bzw. die adressierte Distribution. Daher wird die Umsetzung in den meisten Fällen durch einen entsprechend erfahrenen Entwickler erfolgen, dem im Gegenzug der Umgang mit der API leicht fällt. Wie die Synthese im Einzelnen erfolgt, ist durch die gewählte Architektur nicht vorgegeben und kann somit unter Beachtung der Eigenheiten des Zielsystems auf effiziente Weise erfolgen.

Abbildung 5.3 auf der nächsten Seite zeigt das letztlich implementierte System. Einen alternativen Ansatz der Implementierung erläutert Anhang E.2 auf Seite 236. Das Frontend nutzt das wohlbekannte, textbasierte *kconfig*-System, wel-

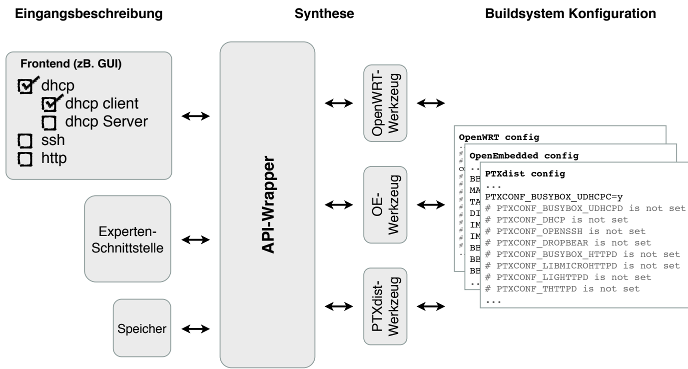


Abbildung 5.3: Finales System

ches im Linux-Kernel seinen Ursprung hat und auch von anderen bekannten Projekten genutzt wird. Es basiert auf einer einfachen, textuellen Beschreibung der Nutzerschnittstelle (vgl. [SB10] und Anhang E.3 auf Seite 238), die dann durch einen Satz von Werkzeugen (z. B. `mconf`, `xconf`, `qconf`) zur Anzeige gebracht wird. Nach Beendigung des kconfig-Dialogs speichern die Werkzeuge die vom Nutzer getroffene Auswahl in einer einfachen Textdatei, im folgenden als *.config* bezeichnet. Mit Hilfe des Frontends können auch bestehende Konfigurationen angepasst werden.

Die .config Datei dient in dieser Implementierung zum Speichern der Spezifikation und als API zwischen dem Frontend und den Synthesewerkzeugen. Für die Verwaltung von Versionen der Konfiguration wird ein DVVS genutzt. Die Synthesewerkzeuge analysieren die .config Datei und setzen die darin enthaltenen Symbole in eine Entsprechung in den Konfigurationen des jeweiligen Zielsystems um. Die dazu verwendeten Abbildungsvorschriften stellen den Kern des Synthesewerkzeugs dar.

Eine exemplarische Umsetzung für PTXdist und Buildroot nutzt dazu viele kleine Dateien, die jeweils die zur Synthese einer Eingangsoption notwendigen Ausgangswerte beinhaltet. Durch die Kombination dieser „Schnipsel" entsteht so die Zielkonfiguration. Abbildung 5.4 auf der nächsten Seite stellt den entstehenden Ablauf für PTXdist dar.

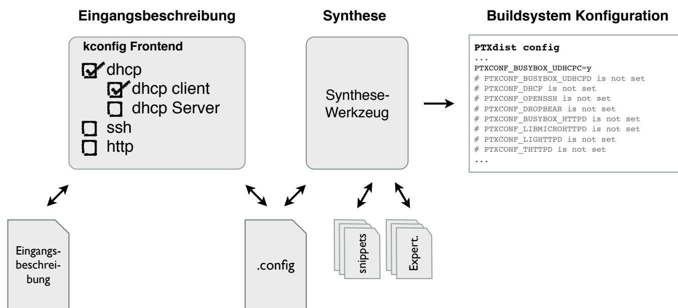


Abbildung 5.4: Ablauf und Umsetzung des Syntheseflows bei PTXdist

In Abbildung 5.3 auf der vorherigen Seite ist außerdem eine Expertenschnittstelle dargestellt, die verschiedene Aufgaben erfüllt. Einerseits bietet sie einen bequemen Zugriff auf die Eingangsbeschreibung, um diese nach Bedarf anzupassen und zu erweitern. Ebenso dient sie der Erweiterung der Synthesewerkzeuge, um diese zu den Optionen der Eingangsbeschreibung synchron zu halten. Darüber hinaus erlaubt sie weitere Interaktionen mit den Synthesewerkzeugen. Wie bereits erläutert, wird für bestimmte Funktionen nach Möglichkeit immer der gleiche Satz von Paketen bzw. Anwendungen genutzt. Unterschiede in der Softwareauswahl ergeben sich meist aus den Anforderungen an das Zielsystem oder durch dessen Hardware. Dieses Verhalten wird berücksichtigt, indem neben der $1:1$ Abbildung von Eingangsfunktion zu Softwarepaket eine $1:n$ Abbildung unter Beachtung weiterer Parameter möglich ist. Im Lauf der Entwicklung und Nutzung des Systems, hat sich die Schnittstelle auch als geeignetes Mittel zur Verknüpfung von Softwarefunktionen mit Test- und Verifikationsfunktionen herausgestellt, was in Abschnitt 5.3 weiter ausgeführt wird.

5.3 Test und Verifikation

Für die Fehlerfreiheit der eigenen Software stehen dem Entwickler alle Möglichkeiten bereit (siehe Abschnitt 2.6.3). Die Beachtung von Entwurfsstandards und Regeln für die Implementierung testfreundlicher Software, sowie

die Nutzung entwurfsunterstützender Werkzeuge verringern das Fehlerrisiko und erleichtern Test und Verifikation. Trotzdem bleiben unentdeckte Fehler in der Software ein ernstes Problem, wie z. B. [Spi+11] zeigt. Die Korrektheit von Fremdsoftware sollte also nicht vorausgesetzt, sondern im Bedarfsfall geprüft werden.

Besonders kritisch sind Fehler im Linux-Kernel, da diese das gesamte System instabil und ggf. sogar unbenutzbar machen. In der Literatur existieren Ansätze für die Generierung bzw. Verifikation von Linux-Gerätetreibern [CKK06; PK07; PCK08; PSK09], die sich aber in der Praxis bisher nicht etabliert haben. In den meisten Fällen muss ein Entwickler bereits einen sehr hohen Kenntnisstand besitzen, bevor er mit der Entwicklung eigener Treiber beginnen kann. Das verringert den Nutzen für ein Werkzeug zur Generierung von Treiber-Skeletten. Einen Gewinn würde der Entwickler aus der Generierung von Schnittstellen zu den von ihm verwendeten Subsystemen ziehen. Doch entsprechende Generatoren müssen sich in der gleichen Geschwindigkeit wie der Kernel und seine Schnittstellen weiter entwickeln, was vermutlich ein nicht im Verhältnis zum Gewinn stehender Aufwand ist. Ähnliches gilt für die Ansätze, bei denen der Treiber über eine abstrakte Sprache beschrieben wird (vgl. [CKK06]). Hier ergibt sich lediglich ein minimaler Vorteil aus einer eventuell verbesserten Art der Spezifikation. Verifikationsansätze über Werkzeuge zur statischen und dynamischen Codeanalyse eröffnen einen interessanten Weg für die Fehlersuche, leiden aber ebenfalls an vielen Stellen unter der hohen Dynamik des Kernels und bieten daher nur einen geringen Vorteil gegenüber den klassischen Werkzeugen einer integrierten Entwurfsumgebung (engl. *Integrated Design Environment (IDE)*). Mit seinen integrierten Mechanismen zur Fehlersuche (wie dem KDB) unterstützt der Kernel den Entwickler bereits bei Test und Verifikation.

Für die Bewertung des Gesamtsystems lassen sich aufgrund der Komplexität des Problems kaum automatisierte Lösungen finden. Mit Hilfe einer funktionalen Spezifikation können über interne und externe Unittests Nachweise für die Anwesenheit/Abwesenheit geforderter Eigenschaften gebracht werden. Weitergehende Nachweise erfordern immer einen hohen manuellen Aufwand. „Betatests“, wie sie im Consumer-Bereich oft mit einigen hundert oder gar tausend Testern erfolgen, sind in industriellen Anwendungen mit geringer Stückzahl kaum vorstellbar. Allerdings ist der Einsatzbereich im industriellen Umfeld oft besser eingegrenzt bzw. klar spezifiziert.

Hier greift wiederum das in Abschnitt 5.2 vorgestellte Spezifikationssystem. Es bietet einen Ansatzpunkt für Test und Verifikation basierend auf funktionalen Spezifikationen. Die Eingangskonfigurationen können dazu über die Expertenschnittstelle mit passenden Testansätzen verknüpft werden. Die Ausprägung ist dabei nicht fest vorgegeben, im Moment jedoch hauptsächlich auf die Umsetzung im genutzten Buildsystem beschränkt. So können beispielsweise gezielt Werkzeuge für das Hostsystem erstellt werden, die beim Test des Zielsystems helfen und für die erstellte Software angepasst sind. So hat sich das Mitführen von Skripten in einem Unittest-Verzeichnis bewährt, die die jeweils genutzten Funktionen des Zielsystems (z. B. *DHCP*, Webserver) überprüfen.

Die Bewertung des Gesamtsystems erfolgt häufig mit Hilfe von Entwickler- oder Evaluationsboards, die beispielsweise von den Prozessorherstellern angeboten werden. Diese Systeme sind oft recht teuer und auf das vom Hersteller gebotene Maß an Flexibilität beschränkt. Eine kostengünstige Alternative stellt die Nutzung vorhandener Hardware[2] dar, auch wenn sie der Zielarchitektur nicht entspricht. Von Vorteil ist hier der Einsatz eines Buildsystems zur Erstellung des Linux-Systems, da so das identische User Land auch bei verschiedenen Architekturen erstellt und untersucht werden kann. In jedem Fall existieren bei der Emulation Fehlerquellen aufgrund der Unterschiede zwischen Emulator und Zielsystem. Durch steigende Rechenleistung und verbesserte Ausstattung der Hostrechner, die Entwicklern zur Verfügung stehen, erfreut sich auch die Emulation der Zielhardware mit Hilfe virtueller Maschinen wachsender Beliebtheit [SR09; LM10; Wan+11]. *QEMU*, eine unter der GNU GPL stehende Emulationssoftware, erlaubt dabei auch eigene Anpassungen des emulierten Systems an das Zielsystem. Nachteile bei der Emulation sind bekanntermaßen die verringerte Geschwindigkeit, die möglichen Unterschiede zwischen dem emulierten und dem realen System und die grundsätzliche Verfügbarkeit der Zielarchitektur. In jedem Fall erlaubt die Emulation den frühzeitigen Test des entstehenden Linux-Systems.

[2] beispielsweise alte PC-Hardware oder Entwicklerboards aus vorangegangenen Projekten

5.4 Zusammenfassung und Diskussion

Das Kapitel betrachtet grundsätzliche Methoden für Spezifikation, Test und Verifikation von Linux-Systemen. Diese folgen in weiten Teilen den klassischen Methoden des Softwareentwurfs. Wie aus Kapitel 4 hervorgeht, kann jedoch bei Paketen Dritter zusätzlicher Testaufwand entstehen. Besonderes Augenmerk liegt im Allgemeinen auf dem Linux-Kernel, der eine ausgesprochen komplexe und gleichzeitig die zentrale Software eines Linux-Systems darstellt. Dies spiegelt sich im vorliegenden Kapitel wieder, auch wenn Test und Verifikation nur überblicksartig betrachtet werden.

Eine weitere Herausforderung und gleichzeitig Schwerpunkt dieses Kapitels ist die Auswahl geeigneter Softwarepakete aus der Vielzahl vorhandener Alternativen. Zur Vereinfachung dieses Prozesses stellt Abschnitt 5.1 ein Spezifikationssystem für Linux-Systeme vor, dass die Erfahrungen des Autors aus eigenen Arbeiten sowie aus der Zusammenarbeit und Diskussion mit Partnern und Interessenten umsetzt. Es basiert auf einer vereinfachten Eingangsbeschreibung, die das Systemverhalten auf einer höheren Abstraktionsebene ähnlich einem Lastenheft beschreibt und in eine Konfiguration für verschiedene Buildsysteme bzw. Distributionen überführt. Neben der Vereinfachung und minimalen Formalisierung der Spezifikation spielte die Reproduzierbarkeit bei der Umsetzung des Systems eine wesentliche Rolle. Außerdem wurde die Integration von Testansätzen verfolgt, sodass der gesamte Entwurfsprozess durch den Einsatz des Systems profitiert. Die Umsetzung ist bewusst einfach gehalten. Das verringert die Hürden bei der Nutzung und erleichtert die Anpassung und Weiterentwicklung.

Auch wenn sich das System für Buildsysteme und Distributionen gleichermaßen eignet, liegt das Interesse des Autors in der vereinfachten Nutzung von Buildsystemen. Diese erfordern ein sehr hohes Maß an Vorkenntnis und Einarbeitung, erlauben im Gegenzug aber die Nutzung aller Vorteile des so mächtigen und flexiblen Linux-Ökosystems. Durch den vorgestellten Ansatz vereinfacht sich der Einstieg, da das System die mühselige Auswahl von Softwarepaketen erleichtert und ein sukzessives Vordringen in die komplexen Buildsysteme erlaubt. Erfahrenen Entwicklern erleichtert das System die parallele Nutzung verschiedener Buildsysteme sowie das Aufsetzen neuer Linux-Systeme, da es die Vorlieben des Entwicklers berücksichtigt und durch sein Konzept eine individuelle Anpassung ermöglicht.

6 Umsetzung der Systemkonzepte

Die in den Kapiteln 3, 4 und 5 erläuterten Konzepte und Methoden wurden anhand von realen Anwendungsbeispielen entwickelt und verifiziert. Die Arbeiten orientierten sich dabei unter anderem an der Zielstellung, eine modulare Plattform zur Erfassung und Verarbeitung von Sensordaten zu schaffen. Die Systeme, ihre Besonderheiten und die mit ihnen erfolgten Implementierungen werden nachfolgend beschrieben. Die verwendeten Hardwareplattformen umfassen dabei ein breites Spektrum, angefangen bei preiswerten Systemen für private Anwender, über auf den industriellen Bereich abgestimmte Systeme, bis hin zu FPGA-basierten Plattformen.

6.1 Vorstellung der genutzten Entwicklungsumgebung

Einleitend wird die genutzte Umgebung für den Hard- und Softwareentwurf vorgestellt. Sie unterstützt und integriert alle der nachfolgend vorgestellten Systeme und ermöglicht so die übergreifende und effiziente Nutzung aller Ressourcen. Bereits bei der Konzeption der Umgebung wurden einige Punkte berücksichtigt, die den finalen Aufbau prägen. Dies waren unter anderem:

- die dezentrale Nutzung der Entwicklungsumgebung,
- die Nutzung durch ein Team,
- die Möglichkeit, Entwicklungsstände reproduzierbar zu sichern,
- die Unterstützung verschiedener Plattformen und Architekturen,

- die Unterstützung verschiedener Buildsysteme,
- die Möglichkeit, den Zielsystemen einen sicheren Zugriff auf das Internet zu ermöglichen.

Während des Aufbaus und der Nutzung des umrissenen Konzepts zeigten sich weitere Anforderungen, die das letztlich entstandene System beeinflussten.

Beispielsweise läuft die Entwicklungsumgebung in einer administrierten Umgebung und darf demnach einen vertretbaren administrativen Aufwand nicht überschreiten. Speziell die Schritte, welche die Systemrechte eines Administrators erfordern sind kritisch. Ebenso muss sich das System in die verfügbaren Infrastrukturen eingliedern, angefangen von Rechnerarchitekturen über Betriebssysteme bis hin zu Restriktionen beim Zugriff auf gemeinsame Ressourcen.

6.1.1 Entwicklungsumgebung

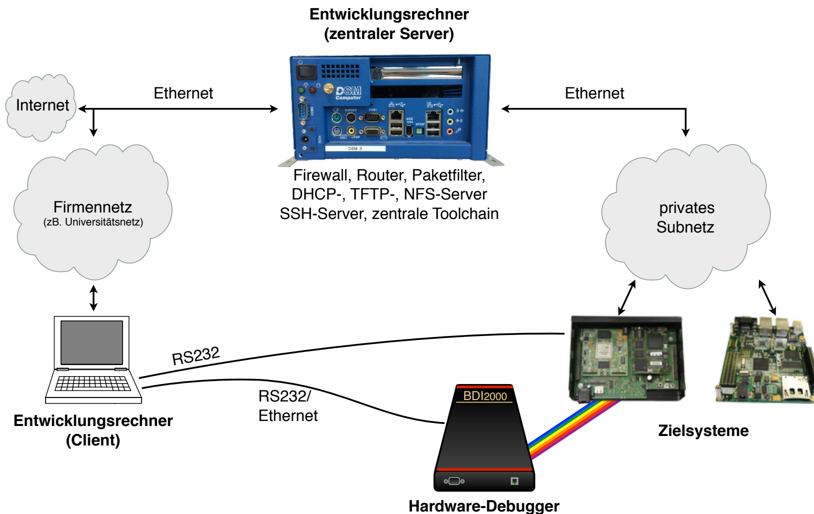


Abbildung 6.1: Genutzte Entwicklungsumgebung

Die aufgebaute Entwicklungsumgebung ist in Abbildung 6.1 auf der vorherigen Seite[1] dargestellt. Zentrales Element ist ein Server, der eine Vielzahl an Aufgaben übernimmt. Der Server ist sowohl mit dem Netzwerk der Universität (und damit auch mit dem Internet) verbunden als auch mit einem privaten Subnetz, in das zudem die netzwerkfähigen Zielsysteme integriert sind. In diesem Subnetz bietet der zentrale Server grundlegende Dienste, wie eine Adresszuordnung per *DHCP* oder als Firewall und Router den geregelten Zugriff auf das Firmennetz der Universität und das Internet. Filter sorgen dafür, dass das Subnetz für die Zielsysteme von den anderen Netzwerken abgeschirmt ist. So können keine unbefugten Zugriffe von Außen erfolgen und Zielsysteme werden erst nach einer grundsätzlichen Freigabe mit den öffentlichen Netzwerken verbunden. Dieses Vorgehen erhöht die allgemeine Sicherheit und verhindert ungewollte Rückwirkungen der Zielsysteme auf die öffentlichen Netzwerke.

Weitere Dienste des zentralen Servers erleichtern die aktive Entwicklungsarbeit. Auf Wunsch werden während des Bootvorgangs per *Trivial File Transfer Protocol (TFTP)* Kernel-Abbilder an die Zielsysteme verteilt. Ebenso stehen Wurzeldateisysteme für die einzelnen Zielsysteme per Netzwerk bereit. Hierfür kommt das *Network File System* (*NFS*) zum Einsatz. Die Wurzeldateisysteme und das Basisverzeichnis für den TFTP-Dienst sind auch auf dem zentralen Server und den Entwicklungsrechnern verfügbar. Das ermöglicht einen transparenten Austausch von Dateien zwischen Entwicklungsrechner, zentralem Server und Zielsystem. Aufgrund der beiden Dienste ist es nicht notwendig, den Festspeicher eines Zielsystems nach jeder Änderung an der Software zu aktualisieren. Ebenso sind Modifikationen, die auf dem Zielsystem erfolgen, direkt auf den Entwicklungssystemen sichtbar. Daraus ergibt sich ein dynamischer Arbeitsablauf ohne aufwändige Synchronisation.

Der zentrale Server dient gleichzeitig als Entwicklungsrechner, indem er Entwicklern den Zugriff per *Secure Shell (SSH)* erlaubt. Auf dem Server sind die notwendigen Werkzeuge, Toolchains und Buildsysteme verfügbar. Über seine Verzeichnisdienste stellt er die Buildsysteme[2] auch im Universitätsnetz bereit, so dass Entwickler direkt auf ihren Rechnern arbeiten können. Da der Zugriff

[1] Die Abbildung des Laptops stammt vom Nutzer „deusinvictus" von openclipart [deu10].

[2] Das Bereitstellen der Werkzeuge und Toolchains bringt aufgrund der Heterogenität der Rechner keine Vorteile.

auf die Zielsysteme per Netzwerk nicht immer möglich ist, wird parallel ein lokaler Zugang benötigt. Im Unix-Umfeld kommen dafür oft Terminalverbindungen per RS232 zum Einsatz. Auch bei eingebetteten Systemen hat sich diese Form des lokalen Zugriffs durchgesetzt. Für die Fehlersuche direkt am Zielsystem gibt es eine Vielzahl an Möglichkeiten. Aufgrund ihrer Flexibilität werden oft Software-basierte Ansätze bevorzugt (vgl. Abschnitt 2.6.3, sowie [Yag+08, Kapitel 11], [LGS13]). Allerdings existieren auch sehr gute Geräte speziell für das Debugging eingebetteter Systeme, wie der in Abbildung 6.1 auf Seite 146 gezeigte *BDI2000*.

Die Entwicklungsumgebung berücksichtigt auch diese Geräte. Sie sind ebenfalls in das lokale Subnetz integriert und beziehen ihre Konfiguration vom zentralen Server. Die Verbindung zum Entwicklungsrechner erfolgt – je nach Gerät – per RS232 oder Ethernet. Am Zielsystem existiert meist ein spezieller Anschluss, wobei oftmals das JTAG Protokoll verwendet wird.

Die dezentrale Struktur der Entwicklungsumgebung ist somit in mehrfacher Hinsicht ein Vorteil. Der zentrale Server weist nur eine begrenzte Anzahl RS232 Schnittstellen auf, während die Rechner der einzelnen Entwickler jeweils hinreichend Schnittstellen zum Anschluss eines Zielsystems und eines Debuggers bereithalten. Ein direkter Zugriff auf die Hardware[3] erleichtert zudem die Arbeit, beispielsweise beim Anschluss von Peripherie oder beim gelegentlichen Drücken des „Reset"-Knopfes. Die Verteilung des lokalen Subnetzes an die Arbeitsplätze der Entwickler ist hingegen in den meisten Infrastrukturen möglich.

6.1.2 Versionsverwaltung

Abschnitt 4.4.4 zeigt bereits ein konkretes Modell für das Versionsmanagement der Werkzeuge, Buildsysteme und der Software am Beispiel von Git. Git existiert jedoch erst seit dem Jahr 2005 und es vergingen einige Jahre, bevor es sich auf breiter Front gegen andere Systeme behaupten konnte.

Vorher galt Subversion, das auch heute noch an vielen Stellen zum Einsatz kommt, als Quasi-Standard für die Versionsverwaltung. Diese Entwicklung spiegelt sich auch in der hier genutzten Umgebung wieder. Anfangs erfolgte

[3] „Direkt" meint hier physischen Zugriff am Arbeitsplatz des Entwicklers.

die Versionsverwaltung mit Subversion, bei dem Zweige (*Branches*) keine so ausgeprägte Rolle spielen wie bei Git. Das Grundkonzept, stabile Versionen in der Versionsverwaltung zu halten, wurde aber schon hier genutzt. Mit der wachsenden Popularität von Git erfolgte sowohl die Umstellung der Versionsverwaltung auf das neue Werkzeug als auch die Umstellung des Versionsmanagements auf die in Abschnitt 4.4.4 vorgestellten Konzepte.

Die Entscheidung brachte speziell in der Anfangsphase verschiedene Probleme mit sich. Viele Projekte der Linux-Community setzten noch auf Subversion, ebenso wie die automatisierten Abläufe in der Entwicklungsumgebung. Außerdem gab es trotz der Vorteile von Git ein Akzeptanzproblem im Entwicklerteam.

Für die Zusammenarbeit von Git und Subversion gibt es das Programm `git-svn`, das die Nutzung von Subversion-Repositories mit Git erlaubt. Die Abläufe innerhalb der Entwicklungsumgebung mussten auf das neue Versionsmanagement portiert werden. Dabei konnten unter anderem der geringere Speicherplatzbedarf für eine Arbeitskopie (*Repository*) und der daraus resultierende Geschwindigkeitsvorteil von Git nachgewiesen werden, was exemplarisch in Listing 6.1 und Listing 6.2 dargestellt ist.

```
krid@dsm1 $ du -hs gps-box.svn gps-box.svn
574M    gps-box.svn
248K    gps-box.svn/build-system
2,2M    gps-box.svn/demo-software
360K    gps-box.svn/dki-box
5,5M    gps-box.svn/hl-synthese
126M    gps-box.svn/system-software
441M    gps-box.svn/xup-board
```

Listing 6.1: Speicherplatzbedarf eines Subversion-Repositories

```
krid@dsm1 $ du -hs gps-box gps-box
489M    gps-box
72K     gps-box/build-system
600K    gps-box/demo-software
52K     gps-box/dki-box
2,5M    gps-box/hl-synthese
50M     gps-box/system-software
202M    gps-box/xup-board
```

Listing 6.2: Speicherplatzbedarf eines Git-Repositories

Die Veränderung etablierter Abläufe stößt meist auf Akzeptanzprobleme. Durch die Verbesserung der Abläufe, speziell im internen Versionsmanagement, konnten diese jedoch schnell überwunden werden. Beispielsweise lässt sich der dezentrale Ansatz der Entwicklungsumgebung mit einem DVVS besser abbilden. Außerdem wechseln immer mehr Projekte zu Git, was die Nutzung von Subversion weiter einschränkt. Ein Wechsel zu Abläufen, die Git integrieren, wird daher unabhängig von durchaus existierenden Nachteilen unausweichlich.

6.1.3 Buildsysteme

Bereits in Kapitel 4 wurde festgestellt, dass sich je nach Aufgabe und Entwicklungsziel verschiedene Distributionen und Buildsysteme für die spezifische Umsetzung eignen. Auch bei den nachfolgend vorgestellten Anwendungsbeispielen kamen daher unterschiedliche Systeme zum Einsatz. Hauptsächlich wurden PTXdist [Pen12], OpenWrt [Fai08], Buildroot [Viz13] und Debian GNU/Linux [Jur13] genutzt.

Buildroot und das mit ihm verwandte OpenWrt sind klassische Buildsysteme mit lokaler Installation und optionalen Out of Tree Builds. Beide Systeme erzeugen nach der notwendigen Konfiguration von Zielsystem und Softwarepaketen erst eine passende Toolchain und im Anschluss die entsprechenden Bibliotheken und Anwendungen. Es erfolgt keine Trennung von hardwareabhängigen und -unabhängigen Teilen der Konfiguration. Die Integration neuer Zielsysteme bedarf damit einer Anpassung des Buildsystems.

PTXdist baut auf eine zentrale Installation von Buildsystem und Toolchain. Die Erstellung des Linux-Systems erfolgt in einem separaten Projektverzeichnis mit vorgegebener Struktur. PTXdist unterscheidet zwischen den Teilen der Konfiguration, die hardwarenah sind und denen, die die Softwareauswahl beinhalten. Neue Hardwareplattformen erfordern nur die Erstellung einer neuen Hardwarekonfiguration. Veränderungen am Buildsystem sind nicht notwendig. Auch alle anderen Anpassungen können im Projektverzeichnis erfolgen. Mit *OSELAS.Toolchain()* existiert ein auf PTXdist abgestimmter Weg zur Erzeugung von Toolchains.

Mit Debian GNU/Linux kam eine etablierte und als besonders stabil geltende Distribution zum Einsatz. Für Debian existieren sehr viele Arbeitsabläufe,

die eine weitgehende Automatisierung bei der Erstellung des Linux-Systems erlauben. Pakete für die Distribution stehen üblicherweise als kompilierte Binärpakete bereit. Für eigene Erweiterungen ist eine Toolchain notwendig, mit der diese Pakete erstellt werden. Der Unterschied in den Philosophien von Buildsystem und Distribution wird bei der Erstellung eines individualisierten Linux-Systems sehr deutlich.

Im Rahmen der Untersuchung und Bewertung geeigneter Buildsysteme (vgl. [KKH11]) kamen zusätzlich das *Embedded Linux Development Kit* (*ELDK*), *Gentoo Linux* [Wro08] und *OpenEmbedded* [STM10] zum Einsatz. Außerdem erfolgt eine kontinuierliche Beobachtung der Entwicklung im Bereich der Buildsysteme und Distributionen. Eine Beurteilung der Systeme erfolgt anhand einiger Faktoren, die sich aus den Anforderungen der Entwicklungsumgebung, denen der Hardwareplattformen und denen der Anwendungsfälle ergeben.

Die Systeme müssen sich in die bestehende Umgebung einfügen und den administrativen Gegebenheiten genügen. Das betrifft eine einfache Versionierung von Buildsystem bzw. Distribution, BSP und Toolchain ebenso wie die Möglichkeiten zur zentralen Installation und Nutzung des jeweiligen Buildsystems. Bei den betrachteten Distributionen wurde zudem darauf geachtet, wie gut bzw. wie genau sich ein Systemzustand reproduzieren lässt und in welchem Maß eine Automatisierung der Systemerstellung möglich ist. Für die Erstellung von Toolchain und Wurzeldateisystem sollten zudem keine administrativen Systemrechte benötigt werden.

Weiterer Beurteilungspunkt ist die Integration neuer Hardwaresysteme in den jeweiligen Ablauf. Bei Buildsystemen bedeutet dies entweder eine Erweiterung des Systems (z. B. bei OpenWrt oder Buildroot), oder nur das Anlegen eines neuen BSP (z. B. bei PTXdist). Soll eine Distribution zum Einsatz kommen, muss diese für die Zielarchitektur existieren und nach Möglichkeit die Zielplattform unterstützen.

Bei der Umsetzung konkreter Entwicklungsaufgaben kommt es schnell dazu, dass der gebotene Softwareumfang des gewählten Entwicklungssystems nicht ausreicht. Entweder unterstützen vorhandene Pakete einige Features nicht, oder benötigte Software wird nicht mitgeliefert. In beiden Fällen ist eine Anpassung oder Erweiterung notwendig. Weiterhin kann die Korrektur von Fehlern in Programmpaketen durch eigene Patches notwendig sein. Der Umgang

und Aufwand für die Anpassung und Integration von Softwarepaketen sowie die Möglichkeiten bei der Nutzung eines eigenen Linux-Kernels bildeten einen weiteren Punkt bei der Beurteilung der genutzten Ansätze.

Zur allgemeinen Beurteilung einzelner Systeme werden die hier vorgestellten Plattformen teilweise oder ganz integriert, was zwangsläufig auch Anpassungen am Linux-Kernel zur Folge hat. Die bestehende Softwareauswahl wird um eine Minor-Version der Skriptsprache Python 2 mit Unterstützung der Grafikbibliothek *Tkinter* ergänzt. Zudem erfolgt die Integration eigener Anwendungen sowie die Anpassung der beim Systemstart ausgeführten Dienste. Damit ergibt sich eine Auswahl an Aufgaben, die dem üblichen Vorgehen bei der Erstellung angepasster Linux-Systeme nahe kommt.

6.2 Xilinx System „embedded FPGA“

Zu den verschiedenen FPGA-Versionen stellt Xilinx im Rahmen seines Universitätsprogramms [Xil09b] sehr umfangreiche Test- und Evaluationsplattformen bereit. Von diesen wurde das *Virtex-II Pro Development System (XUP-V2P)* [Xil05] für die Umsetzung verschiedener Projekte genutzt. Später erfolgte die Übertragung eines Teils der Arbeiten auf das neuere ML507-System [Xil11b].

Beide Systeme erfreuen sich aufgrund der verwendeten Virtex-FPGAs mit ihren integrierten PowerPC-Kernen und der Möglichkeit zur dynamisch-partiellen Rekonfiguration großer Beliebtheit. In den nachfolgenden Beispielen kamen die Systeme als flexibles PSoC zum Einsatz. Die Möglichkeiten zur dpR wurden hingegen nicht umfassend genutzt, da die Anwendungen dies nicht erforderlich machten.

6.2.1 Hardware

Das XUP-V2P basiert auf dem *xc2vp30*, einem Virtex-II Pro FPGA mit zwei integrierten PowerPC-405 Kernen. Beide Kerne sind als Hardmakro ausgeführt und über zwei Bussysteme[4] mit der umgebenden Logik verbunden. Weiterhin

[4] Den *On-chip Peripheral Bus (OPB)* und den *Processor Local Bus (PLB)*.

bietet das FPGA beispielsweise 13.969 *Slices*, 8 digitale Taktmanager und 2448 KiB Block-RAM. Das Entwicklungssystem verfügt über verschiedene Schnittstellen und Möglichkeiten zur Erweiterung, unter anderem:

- zwei DIMM-Steckplätze für bis zu 2 GiB DDR-SDRAM,
- einen VGA-Anschluss für externe Bildschirme,
- RS232-Anschlüsse,
- einen Ethernet-Anschluss,
- einen Platform Flash PROM,
- und einen Steckplatz für eine Compact-Flash Karte.

Das ML507 basiert auf einem Virtex-5 FPGA, der zwei PowerPC-440 Kerne enthält. Im Gegensatz zum XUP-V2P wird das ML507 auch von den aktuellen Xilinx-Werkzeugen unterstützt. Daher bietet es momentan die geeignete Plattform für die Weiterführung von Arbeiten auf Basis der PowerPC-Architektur. Denn weder die Virtex-6 noch Series-7 FPGAs bieten integrierte Prozessorkerne. Mit der Zynq-Architektur [Xil12f] hat Xilinx schließlich einen Wandel hin zu eigenständigen, ARM-basierten SoCs mit angebundenem FPGA vollzogen.

Um auf den Virtex-FPGAs ein Linux-System zu nutzen, ist die Kombination der PowerPC-Kerne mit weiteren Komponenten, die in den Logikblöcken des RMs instanziiert werden, notwendig. Ein minimales System besteht dabei aus einem RAM-Controller zur Nutzung des DDR-SDRAMs und einigen Kilobyte des internen Block-RAMs für den initialen Bootloader. Zur Nutzung dieser Komponenten sind weiterhin einige Subsysteme, wie die Busbrücken für OPB und PLB, notwendig. Zusätzliche IP-Cores verbessern die Nutzbarkeit des Systems. Unter anderem lassen sich Blöcke für *UART*[5], Ethernet, Grafik (VGA) und PS/2 integrieren. Durch weitere IP-Cores kann die Hardware an die Erfordernisse der Anwendung angepasst werden. So bieten sich für den embedded Bereich Schnittstellen wie SPI und I^2C ebenso an, wie frei verfügbare GPIO-Pins.

[5] Der *Universal Asynchronous Receiver Transmitter (UART)*-Core dient als Basis für eine RS232-Schnittstelle.

Durch die Möglichkeiten der Rekonfiguration müssen nur die im jeweiligen Anwendungsfall benötigten Schnittstellen implementiert werden, was den Energieverbrauch senkt und mögliche Fehlerquellen reduziert. Mittels dR kann das System auch im Feld auf neue Anforderungen reagieren. Durch die festgelegte Hardwarearchitektur des XUP-V2P-Boards können allerdings nicht alle Möglichkeiten des verwendeten Virtex-II-FPGAs ausgeschöpft werden.

6.2.2 Software und Entwurfsumgebung

Zur Erstellung des notwendigen Hardwaredesigns wird der *Base System Builder* aus dem Xilinx *Embedded Development Kit (EDK)* verwendet. Er erlaubt die bequeme Konfiguration eines PSoC mit Hilfe eines GUI. Ist das Design vorbereitet, kann der zugehörige Bitstrom jederzeit durch die Kommandozeilenprogramme von Xilinx erzeugt werden.

Zur Erstellung des Linux-Systems und der Toolchain kommt ein modifiziertes OpenWrt zum Einsatz. Es unterstützt die zentrale Erstellung von Toolchains bzw. die Nutzung zentral installierter Toolchains. Außerdem wurde es um eine entsprechende XUP-V2P-Plattform ergänzt.

Eine Besonderheit bei der Verwendung eines PSoCs ist die variable Hardware des Systems. Diese wirkt sich auf den Linux-Kernel, den Bootloader und gegebenenfalls sogar auf die zu verwendende Toolchain aus. Daher müssen die hardwarebezogenen Informationen in das Linux-Buildsystem überführt werden.

Anfangs geschah dies über spezielle Header-Dateien, welche die Xilinx-Werkzeuge während der Erstellung des Bitstromes mit erzeugten. Diese enthielten die notwendigen Angaben (z. B. Adressbereiche und Interrupts) der verwendeten Komponenten. Eine angepasste Konfiguration für den Linux-Kernel lies sich damit aber nicht ohne Weiteres erzeugen. Somit musste für jede neue Konfiguration auch ein neues BSP erstellt werden.

Deutlich mehr Flexibilität bieten hier der *Device Tree* [GH06; LB08; Lik08] bzw. der im Linux verwendete *Flattened Device Tree (FDT)*. Dieser beschreibt in einer Baumstruktur die Hardwarekomponenten eines Systems zusammen mit ihren wesentlichen Eigenschaften (z. B. Adressbereiche, Interrupts). Der

sogenannte *Device Tree Blob*[6], entsteht durch Kompilieren der textuellen Beschreibung des Device Trees. Er wird beim Booten an den Kernel übergeben, der ihn dann auswertet und so über die vorhandene Hardware informiert ist. Entsprechend der Informationen im Device Tree werden die benötigten Treiber geladen und konfiguriert. Auf diese Weise kann ein Linux-System flexibler auf geänderte Hardware reagieren. In den meisten Fällen[7] ist ein BSP für mehrere ähnliche Boards ausreichend.

Anfangs unterstützte nur die PowerPC-Architektur die FDTs. Inzwischen setzt sich die Verwendung auch bei der ARM Architektur durch. Xilinx gilt schon seit langer Zeit als Unterstützer des FDT und bietet mit dem *Device Tree Generator* ein Werkzeug für dessen automatisierte Erstellung.

Das SRAM-basierte FPGA muss bei jedem Einschaltvorgang konfiguriert werden. Dafür bieten das XUP-V2P ebenso wie das ML507 verschiedene Möglichkeiten. Von diesen haben sich drei als besonders interessant herausgestellt:

- eine externe JTAG-Quelle,
- das per SelectMAP angebundene PFP und
- die optional nutzbare Compact-Flash Karte.

Die externe JTAG-Quelle erlaubt jederzeit die Rekonfiguration des FPGA durch einen PC. Dieser Weg eignet sich besonders während der aktiven Hardwareentwicklung, da hier die Ergebnisse direkt in das FPGA geladen werden können. Das PFP fasst bis zu zwei Konfigurationen, zwischen denen per Schalter gewählt werden kann. Somit können beispielsweise eine „sicher funktionierende„ und eine „unsichere“ Konfiguration hinterlegt werden. Besonders interessant erscheint jedoch die Compact-Flash-Karte, die bis zu acht Konfigurationen aufnehmen kann und zusätzlich hinreichend Platz für ein Linux-Wurzeldateisystem bietet.

Die Konfiguration von der Compact-Flash-Karte erfolgt mit Hilfe des SystemACE-Controllers. Er überträgt den Bitstrom in das FPGA, sorgt für ein Systemreset und den Start der in der Konfigurationsdatei hinterlegten Software. Für die XUP-V2P und ML507 wurden dabei versuchsweise sowohl der

[6] Oft auch nur als „Blob“ bezeichnet.

[7] Noch nutzen nicht alle Treiber und Architekturen den FDT. Für hochgradig optimierte Linux-Systeme ist der FDT nicht geeignet, was aber an dieser Stelle nicht weiter ausgeführt werden soll.

Bootloader „Das U-Boot“ als auch der Linux-Kernel in die Konfigurationsdateien integriert.

Die Automatisierung der einzelnen Abläufe erfolgt mit Hilfe eines eigens dafür entwickelten Shell-Skripts. Dieses Skript arbeitet auf Basis von Stufen, die jeweils eine bestimmte Aufgabe erfüllen. Durch die Aneinanderreihung der Stufen entstehen:

- der Bitstrom für das FPGA,
- falls benötigt die zugehörige Toolchain,
- ein korrekt auf die Hardware abgestimmter Bootloader und
- ein korrekt auf die Hardware abgestimmtes Linux-System.

Dazu führt das Skript die verschiedenen Kommandos aus und lädt fehlende Komponenten von verschiedenen Servern nach. Beachtenswert ist, dass das Skript die Verwaltung und Verwendung mehrerer Nutzer, mehrerer Hardwaresysteme und verschiedener Softwarekonfigurationen unterstützt. Dadurch ergänzt es OpenWrt um die fehlende Trennung von hardwareabhängigen und hardwareunabhängigen Konfigurationen und erweitert den zentralen Ansatz von OpenWrt. Dadurch können beispielsweise mehrere Nutzer parallel an einem XUP-V2P-Board arbeiten.

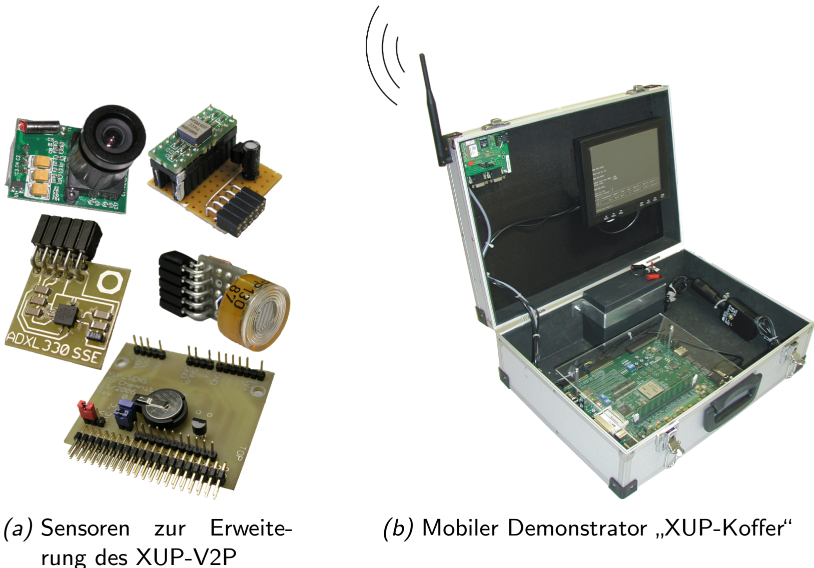
6.2.3 Systemkonzept und -entwurf

Entsprechend seiner Natur eignet sich das PSoC als flexible Plattform für die Entwicklung von angepassten eingebetteten Systemen. Außerdem erlaubt es die Integration eigener Hardwaremodule, beispielsweise eigener Schnittstellen oder Co-Prozessoren. Damit erscheint es als ideale Plattform für das in Anhang F auf Seite 241 beschriebene Projekt.

Konzeption und Entwurf folgen dementsprechend dem in Abschnitt 2.7 beschriebenen Ablauf. Je nach Anwendungsfall muss zuerst eine Partitionierung in Hard- und Softwaremodule erfolgen. Anschließend werden die jeweiligen Komponenten weiter verfeinert, implementiert, getestet und integriert. Die Entwurfsumgebung unterstützt dabei besonders bei der Integration, da sie auf einfache Weise die einzelnen Komponenten zusammenführt.

6.2.4 Beispielanwendung

Mit Hilfe des XUP-V2P wurden prototypische Entwicklungen von IP-Cores durchgeführt. Interessant war dabei die Möglichkeit, die Komponenten direkt in Verbindung mit einem Linux-System zu testen. Während der Umsetzung der verschiedenen IP-Cores konnten außerdem weitreichende Erkenntnisse zu Linux-Systemen und zum Zusammenspiel von Linux mit externen und internen Hardwarekomponenten gewonnen werden. Die begrenzte Geschwindigkeit des PSoC (100 MHZ PLB-Bus, 300 MHz Prozessortakt, keine FPU) förderte auch das Verständnis für grundlegende Optimierungsstrategien innerhalb des Linux-Systems.



(a) Sensoren zur Erweiterung des XUP-V2P

(b) Mobiler Demonstrator „XUP-Koffer“

Abbildung 6.2: Mobiler Demonstrator „embedded FPGA“ und zugehörige Sensoren

Das XUP-V2P wurde mit verschiedenen Sensoren erweitert, von denen einige in Abbildung 6.2a dargestellt sind. Die Sensoren nutzen unterschiedliche Bussysteme zur Kommunikation mit dem Host, u. a. One-Wire, SPI, I^2C sowie per

GPIO realisierte, nicht standardisierte. Je nach Sensor kamen außerdem Filter in der FPGA-Logik zum Einsatz, die die Signale bereits vorverarbeiteten.

Abschließend entstand ein mobiles FPGA basiertes, eingebettetes System mit Bildschirm, Tastatur und Funksystem (vgl. Abbildung 6.2b auf der vorherigen Seite), das als Demonstrator in verschiedenen Szenarien zum Einsatz kam. Unter anderem diente es als „Roadside-Unit“ in Verbindung mit dem PowerPC-System „CarBox“, das in Abschnitt 6.4.4 genauer erläutert wird.

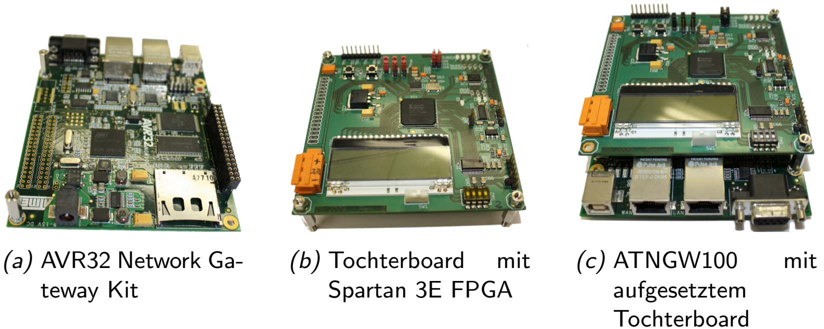
6.3 Prototyping-System „AVR32“

Mit der *AVR32*-Serie [Atm09] bietet die *Atmel Corporation*, einer der weltweit führenden Hersteller (vgl. [LV10]) verschiedener ICs, eine SoC-Plattform basierend auf ihrer eigenständigen 32-Bit-*RISC* Prozessorarchitektur. Ausgehend von einem Kooperationsprojekt [SKH09] erfolgte die Analyse und Untersuchung dieser SoCs. In Anbetracht der Marktdominanz der Intel-, ARM-, PowerPC- und *MIPS*-Architekturen erscheint eine Alternative als interessantes Forschungs- und Ausbildungsobjekt, was zur weiteren Nutzung führte.

6.3.1 Hardware

Als Referenzdesign für die AVR32-Serie bietet Atmel das *AVR32 Network Gateway Kit (ATNGW100)* an, das in Abbildung 6.3a auf der nächsten Seite abgebildet ist. Das Board ist mit einem *AVR32AP7000* SoC bestückt, verfügt über 32 MiB RAM und 8 MiB Flash-Speicher. Es bietet verschiedene Schnittstellen wie zwei Ethernet-Anschlüsse, einen RS232-Anschluss, einen SD-Karten-Einschub, einen USB-B-Anschluss und mehrere als *General Expansion Headers* bezeichnete Steckleisten, die weitere Schnittstellen des SoCs zugänglich machen.

Für die Umsetzung verschiedener heterogener rekonfigurierbarer Systeme kommen die bereits in Abschnitt 6.2 vorgestellten Boards XUP-V2P und ML507 zum Einsatz. Zusätzlich wurde ein zum Formfaktor und zu den Schnittstellen des ATNGW100 kompatibles Tochterboard entworfen. Das in



(a) AVR32 Network Gateway Kit

(b) Tochterboard mit Spartan 3E FPGA

(c) ATNGW100 mit aufgesetztem Tochterboard

Abbildung 6.3: ATNGW100 und Tochterboard

Abbildung 6.3b dargestellte System wird bei der Überwachung von Faserkunststoffverbunden eingesetzt [Wol12]. Es ist mit einem Spartan 3E FPGA bestückt und verfügt über weitere Peripherie, unter anderem einen PFP, eine 7-Segment Anzeige, einen *Analog-Digital-Wandler (A/D-Wandler)*, einen *Digital-Analog-Wandler (D/A-Wandler)*, *Light Emitting Diodes (LEDs)* und DIP-Schalter. Für nähere Erläuterungen zum Tochterboard siehe [Put10]. Das Board bietet aus Sicht des AVR32 verschiedene Möglichkeiten der Konfiguration. So können das FPGA und der PFP per JTAG erreicht werden. Alternativ lässt sich das FPGA im Slave-Serial Modus konfigurieren. Die dazu notwendigen Pins sind ebenfalls vom AVR32 aus zugänglich. Idealerweise werden dazu beide Boards verbunden, wie es in Abbildung 6.3c dargestellt ist.

6.3.2 Software und Entwurfsumgebung

Das SoC und das Board werden grundsätzlich vom Linux-Kernel unterstützt. Atmel liefert als Entwicklungsumgebung eine angepasste Version von Buildroot. Auch vom OpenWrt-Projekt wird das ATNGW100 unterstützt. Somit sind Inbetriebnahme und Softwareentwicklung mit geringem Aufwand verbunden. Allerdings ist das Erstellen einer Toolchain außerhalb der beiden vorgenannten Buildsysteme mit erhöhtem Aufwand verbunden. Die Unterstützung der AVR32-Architektur erfordert sowohl Anpassungen an der GCC als auch an den binutils. Weiterhin sind nur ausgewählte Kombinationen von bi-

nutils und GCC für die AVR32-Architektur geeignet. Daher dient für die hier vorgestellten Anwendungen das von Atmel gelieferte Buildroot als Ausgangspunkt. Je nach Bedarf wird das Buildsystem angepasst und erweitert, beispielsweise für das in [SKH09] vorgestellte Firmware-Update.

Das ATNGW100 nutzt U-Boot als Bootloader. Weiterhin kommt ein Linux-Kernel der 2.6er-Serie zum Einsatz. Buildroot setzt traditionell die uClibc und Busybox ein, was zu einem ausgesprochen schlanken Linux-System[8] führt.

6.3.3 Systemkonzept und -entwurf

Das System dient der Untersuchung von Update-Strategien für das Linux-System und die angebundenen FPGAs. Dabei kommen verschiedene Konfigurationsarchitekturen (vgl. Abschnitt 3.2.3) zum Einsatz, die sich an den Möglichkeiten der Referenzsysteme XUP-V2P, ML507 und Tochterboard orientieren. Damit bleiben die Speicherung eines Bitstroms in einem PFP, die Konfiguration eines FPGAs per JTAG und die Konfiguration per Slave-Serial-Verbindung. Firmware und Bitströme werden dem System sowohl per Netzwerk als auch über die SD-Karte zugeführt.

Die Ablaufsteuerung des Update-Vorgangs wird in allen Fällen vom AVR32 übernommen. Als Referenz für die Konfiguration eines FPGAs können die in [Xil07b], [DF00], [Car99] und [PK06] beschriebenen Varianten dienen. Nachteil dieser Ansätze ist deren Low-Level Umsetzung der Update-Algorithmen auf *Complex Programmable Logic Device (CPLD)*- oder Mikrocontrollerbasis. Wünschenswert bei der Realisierung eines Firmware-Updates ist jedoch in vielen Fällen eine High-Level Umsetzung auf Betriebssystemebene. Demnach erfolgt die Untersuchung derartiger Strategien, wobei Linux als Referenzbetriebssystem genutzt wird.

Den Ausführungen in Abschnitt 4.1.1 folgend, ergeben sich zwei Möglichkeiten für die Umsetzung von Konfigurationssoftware:

- der Kernel Space und
- der User Space.

[8] Das hier verwendete Linux-System belegt nur etwa 3 MiB des Flash-Speichers.

Die Vor- und Nachteile beider Ansätze wurden bereits in Abschnitt 4.5 diskutiert. Von Interesse sind an dieser Stelle daher die Auswirkungen der jeweiligen Entwurfsentscheidung auf das entstehende System und den Entwurfsablauf.

Beide Ansätze erfordern für die Interaktion mit den verschiedenen Systemteilen einen gewissen Anteil an Software im User Space. Dies umfasst beispielsweise das Empfangen neuer Software per Netzwerk oder SD-Karte. Die Umsetzung im Kernel Space erfordert weiterhin den Entwurf, die Implementierung und die Wartung eines entsprechenden Treibers. Im User Space kann hingegen auf vorhandene Bibliotheksfunktionen aufgebaut werden. Die sich daraus ergebenden Unterschiede werden in den nachfolgend dargestellten Beispielen verdeutlicht.

6.3.4 Beispielanwendung

Motivation für die folgenden Beispiele war unter anderem die Umsetzung einer Low-Cost Lösung zur Realisierung eines schnellen und sicheren Updates von FPGAs und PFPs mit geringem HW-Aufwand. Zum Vergleich und zum besseren Verständnis der Auswirkungen einer Implementierung im User und im Kernel Space entstanden daher zwei Kernel-Treiber. Einer erlaubt die Konfiguration eines PFP per JTAG, der andere die direkte Konfiguration eines FPGAs per Slave-Serial-Modus. Ausgangspunkt für die Betrachtungen war ein Programm für den User Space, das Bitströme per JTAG in dafür vorgesehene PFPs überträgt. Tabelle 6.1 auf der nächsten Seite gibt die dafür benötigten Zeiten wieder. Um die fehlerfreie Übertragung der Konfigurationsdaten sicherzustellen, erfolgt nach der Konfiguration jeweils eine Verifikation der Daten. Die Zeit gibt die Summe aus Konfiguration und Verifikation an.

Konfiguration per JTAG

Für die Verwendung des JTAG-Ports muss der AVR32-SoC aus den Konfigurationsdaten entsprechende Steuersignale generieren, die dann auf die Boundary-Scan-Kette der zu programmierbaren Bausteine geschrieben werden. Die Boundary-Scan-Kette besteht aus den in Reihe geschalteten FPGAs und PFPs des zu konfigurierenden Systems.

Tabelle 6.1: JTAG Modus – User Space Software

PROM-Typ	Dateigröße (MiB)	verify	Zeit (Stunden)
XCF16P	9,77	ja	01:02
XCF32P	26,40	ja	01:48
XCF32P	5,35	ja	00:49

Zur Verwendung der Schnittstelle muss der JTAG-spezifische *Test Access Port (TAP)*-Controller auf dem SoC nachgebildet werden. In der vorliegenden Implementierung werden die beim JTAG verwendeten Signale[9] über GPIO-Pins des AVR32AP7000 ausgegeben. Mit jeder steigenden Taktflanke auf TCK werden TDI und TMS abgetastet und daraufhin entsprechende Zustandsübergänge im TAP realisiert. In [Xil07a] wird der in den PFPs eingesetzte TAP näher erläutert.

Als Datenaustauschformat kommt das XSVF zum Einsatz, was die Größe der Konfigurationsdateien gegenüber dem SVF um ca. $\frac{1}{4}$ verringert. Das XSVF besteht aus einer Folge von Befehlen (vgl. Anhang G.1 auf Seite 243) und Daten, die zur Stimulation des TAP-Controllers in korrespondierende JTAG-Signale umgesetzt werden.

Als Basis für die Implementierung der JTAG Software diente der von Xilinx in [Xil07b] referenzierte Quellcode. Der Code implementiert eine *Finite State Machine (FSM)* mit 16 Zuständen. Er ist für die direkte Ausführung auf einem Mikrocontroller ohne Betriebssystem vorgesehen. Er interpretiert XSVF-Datenströme und setzt sie in die zur Stimulation der in einer Boundary-Scan-Kette verbundenen TAP-Controller notwendigen Signale um (vgl. Anhang G.1 auf Seite 243).

Die eingangs erwähnte Umsetzung der Konfiguration im User Space basiert auf einer modifizierten Variante des Xilinx-Codes. Sie nutzt für den Zugriff auf die GPIOs des AVR32 das configfs (vgl. Abschnitt 4.1.2).

Beim vorliegenden Kernel-Treiber erfolgt die gesamte Verarbeitung der XSVF-Kommandos im Kernel Space. Dieser Ansatz erfordert einen komplexeren

[9] *Test Data In (TDI)*, *Test Data Out (TDO)*, *Test Clock (TCK)* und *Test Mode Select (TMS)*

Treiber, vereinfacht jedoch die Schnittstelle zum User Space und erlaubt die Nutzung des Treibers ohne zusätzlichen Implementierungsaufwand im User Space. Da der von Xilinx vorgestellte Code bereits mehrfach verwendet wurde, konnte seine grundsätzliche Funktion und Fehlerfreiheit vorausgesetzt werden. Das begünstigt die Umsetzung im Kernel Space. Auf umfangreiche Test- und Verifikationsansätze kann für diesen Teil des Codes verzichtet werden.

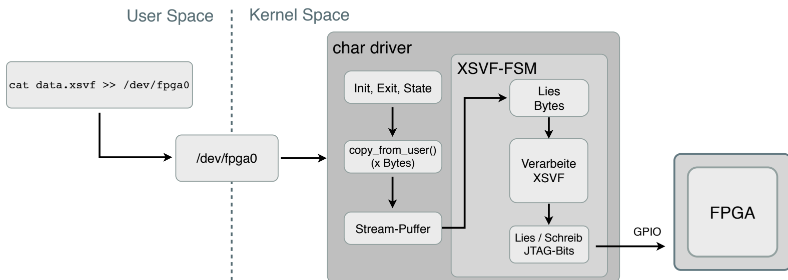


Abbildung 6.4: Schematische Darstellung des `siconf`-Treibers

Die Umsetzung erfolgt als zeichenorientierter Gerätetreiber. Die sich daraus ergebenden standardisierten Schnittstellen und die mit ihr verbundenen Funktionen sind exemplarisch in Anhang G.2 auf Seite 244 dargestellt. Der Xilinx-Code wurde an die Erfordernisse des Kernels angepasst und erweitert. Wie bei der User Space-Implementierung wird die FSM zur Interpretation des XSVF-Datenstroms genutzt. Der Zugriff auf die GPIOs ist aus dem Kernel Space heraus sehr einfach. Als schwierig erweist sich hingegen die Anpassung der Code-Teile zum Einlesen des XSVF-Datenstroms. Während die XSVF-Datei im User Space als kontinuierlicher Datenstrom vorliegt, kann ein zeichenorientierter Gerätetreiber jeweils nur eine bestimmte Datenmenge vom User in den Kernel Space kopieren. Daher ist eine Umsetzung der Methoden des Kernels beim Verarbeiten des Datenstroms notwendig. Die per `copy_from_user( )` blockweise aus dem User Space übernommenen Daten werden innerhalb des Treibers serialisiert, sodass aus Sicht der FSM ein Datenstrom bereitsteht. Wesentliche Randbedingung dafür ist ein angepasstes Timing zwischen Kopierfunktion und FSM. Abbildung 6.4 stellt die gefundene Lösung schematisch dar.

Tabelle 6.2: JTAG Modus – Kernel Space Software

PROM-Typ	Dateigröße (MiB)	verify	Zeit (Minuten)
XCF16P	9,77	ja	ca. 3
XCF32P	26,40	ja	ca. 5
XCF32P	5,35	ja	ca. 2:20

Tabelle 6.2 zeigt die für die Konfiguration mit dem `siconf`-Treiber benötigten Zeiten. Dabei kamen die schon zur Messung in Tabelle 6.1 verwendeten Bitströme und PFPs zum Einsatz. Auch hier erfolgt nach der Konfiguration eine Verifikation der geschriebenen Daten.

Die Werte zeigen eine deutliche Verringerung der Konfigurationszeit. Der erzielte Geschwindigkeitsgewinn beträgt mehr als Faktor 20. Auf weitergehende Optimierungen des Treibers wurde aufgrund dieser Ergebnisse verzichtet. Die Geschwindigkeit lässt sich durch eine Vergrößerung des Puffers für die `copy_from_user()`-Funktion weiter erhöhen. Dies ist ohne Eingriff in den Quellcode möglich, da die Puffergröße als Moduloption implementiert ist. Allerdings belegt der Kernel dementsprechend mehr Speicher, da der Puffer immer alloziert ist.

Der Treiber kann während eines Konfigurationsvorgangs nicht unterbrochen werden. Daher gibt der Kernel – je nach Einstellung – nach etwa einer Minute eine Fehlermeldung aus. Das stellt den wesentlichen Nachteil der momentanen Umsetzung dar. Bei der Verwendung eines Watchdogs muss diese Besonderheit beachtet werden, da das System sonst neu startet. Eine Umsetzung mit möglichen Unterbrechungen wirkt sich negativ auf die Serialisierung der Blockweise empfangenen Daten aus. Die Umstellung des Treibers hätte demnach nicht nur eine deutliche Anpassung aller Codeteile notwendig gemacht, sondern auch zu einer Verlängerung des Konfigurationsvorgangs geführt. Versuche zeigten zudem negative Einflüsse auf die Stabilität des Prozesses. Die Ursache hierfür sind Timingprobleme in Verbindung mit dem genutzten FPGA.

Konfiguration per Slave-Serial Mode

Im Gegensatz zur aufwändigen Interpretation des XSVF-Datenstroms kann bei der Konfiguration im Slave-Serial Modus direkt der generierte Bitstrom (Xilinx .bit-Datei) genutzt werden. Der Treiber setzt eine vollständige Rekonfiguration um. Signalisierung und Datenaustausch folgen dem in Abbildung 6.5 dargestellten Ablauf. Eine Low-Flanke auf dem `PROGRAM_B` Signal führt zum Reset und der anschließenden Rekonfiguration. Dabei signalisiert `INIT_B` die Bereitschaft des FPGAs zum Datenempfang. Anschließend wird mit jeder steigenden Taktflanke auf `CCLK` das Datum auf `DATA` übernommen. Ist die Konfiguration erfolgreich, signalisiert das FPGA dies auf der `DONE`-Leitung.

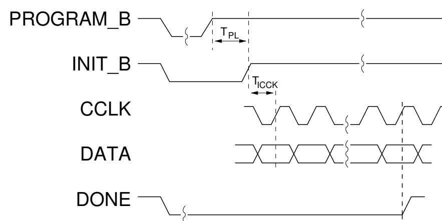


Abbildung 6.5: Taktschema im Slave-Serial Modus

Für die leistungsfähige Implementierung der seriellen Konfiguration ist besonders eine Funktion des AVR32AP7000 interessant. Der im SoC integrierte *Synchronous Serial Controller (SSC)* [Atm07] dient ausschließlich der Realisierung serieller Kommunikation. Der SSC kann über seine internen Register an eine sehr große Anzahl serieller Protokolle angepasst werden. Aus Sicht des Programmierers bietet der SSC eine sehr einfache Schnittstelle für die synchrone Datenübertragung. Der SSC kann Daten mit dem vollen Prozessortakt von 35 MHz übertragen und beherrscht DMA, was den Prozessor während der Datenübertragung entlastet. Für die Signalisierung werden neben dem SSC noch einige GPIO-Pins benötigt. Im Linux-Kernel existiert bereits ein Treiber, der die Basisfunktionen des SSC bereitstellt. Basierend auf diesem Treiber ergibt sich der in Abbildung 6.6 auf der nächsten Seite dargestellte Lösungsansatz.

Trotz des vorhandenen Treibers für den SSC ergaben sich Probleme bei der Inbetriebnahme. Entgegen der Dokumentation ist die Übertragungsrate auf

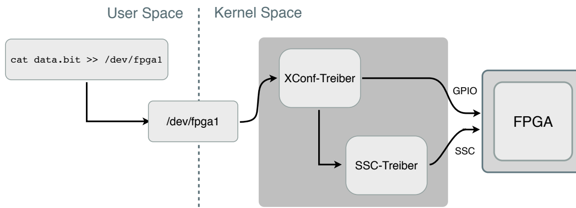


Abbildung 6.6: Schematische Darstellung des XcConf-Treibers

maximal den halben Prozessortakt beschränkt. Außerdem kommt es zu reproduzierbaren Systemabstürzen, sobald der SSC in den Interrupt-Betrieb versetzt wird. Dieser ist jedoch Voraussetzung für die Nutzung der DMA-Funktionalität. Das beschränkt die momentane Lösung auf eine Übertragungsrate vom 17,5 MHz. Die Statussignale des SSCs werden durch zeitaufwändiges Polling abgefragt.

Aufgrund der notwendigen Verbindungen zwischen FPGA und Controller konnte die bitserielle Konfiguration nur mit dem in Abbildung 6.3c auf Seite 159 beschriebenen Aufbau getestet werden. Mit der bestehenden Implementierung konnten die in Tabelle 6.3 wiedergegebenen Übertragungsraten bei der Konfiguration eines Spartan 3E FPGAs erreicht werden.

Tabelle 6.3: Slave-Serial Modus, Kernel-Treiber

Frequenz (kHz)	max. Datenrate $\left(\frac{KiB}{s}\right)$	reale Datenrate $\left(\frac{KiB}{s}\right)$	Erreichte Datenrate (in %)
35,0000	4,3750	4,2884	98,0214
136,7000	17,0875	16,7468	98,0062
273,4000	34,1750	33,3205	97,4998
546,8000	68,3500	66,4462	97,2146
1024,0000	128,0000	77,9078	60,8655
4480,0000	560,0000	79,8740	14,2632
8960,0000	1120,0000	85,6446	7,6468
17920,0000	2240,0000	85,7687	3,8290

Es ist deutlich zu erkennen, dass sich die praktisch erreichte Datenrate mit steigender Geschwindigkeit von der theoretisch möglichen entfernt. Im Bereich oberhalb 1 MHz stagniert die Beschleunigung. Die Ursache für dieses Verhalten liegt in der derzeitigen Implementierung. Die Daten müssen erst vom User Space in den Kernel Space gelangen, was mit der Funktion `copy_from_user( )` geschieht. Bei geringen Geschwindigkeiten bis ca. 300 kHz können die Daten mit ausreichender Geschwindigkeit kopiert werden. Bei höheren Datenraten verhindern systembedingte Latenzen eine weitere Beschleunigung. Dieser Effekt kann wiederum durch die Variation des genutzten Puffers im Kernel beeinflusst werden.

6.4 PowerPC-System „CarBox“

Die PowerPC-Architektur ist ebenfalls *RISC*-basiert und wurde 1991 durch ein Konsortium aus Apple, Motorola und IBM spezifiziert. PowerPC-Systeme finden sich unter anderem in eingebetteten Systemen für die Automobil- und Raumfahrtindustrie. Ausgehend von den bereits erläuterten Plattformen XUP-V2P und ATNGW100 mit Tochterboard stellt das hier verwendete System die konsequente Weiterführung beider Ansätze dar. Als Basis dient ein digitales Kombiinstrument der Firma Unicontrol Systemtechnik GmbH [Uni98], das freundlicherweise für eigene Weiterentwicklungen zur Verfügung gestellt wurde. Aufgrund seiner Herkunft eignet es sich besonders für den Einsatz im Automobil. Trotz der leichten Unterschiede in der Hardware wurde der Begriff DKI-*Box* (*Digitales Kombiinstrument* (*DKI*)) beibehalten. Abbildung 6.7 auf der nächsten Seite zeigt das verwendete System.

6.4.1 Hardware

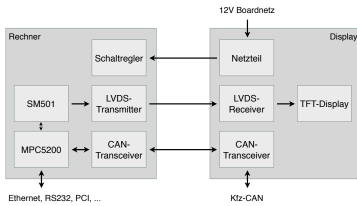
Dieser Abschnitt beschreibt die Hardware der *Digitales Kombiinstrument (DKI)*-Box mit der zugehörigen Displayeinheit und die eigens für das DKI entwickelte Erweiterungskarte.

Abbildung 6.7: Das digitale Kombiinstrument

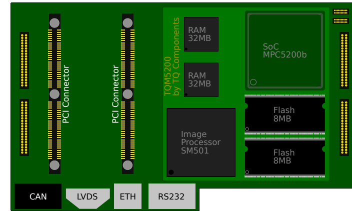
DKI-Grundsystem

Die DKI-Box kombiniert das SoM *TQM5200* der Firma *TQ-Components*[10] mit einem Basisboard, welches diverse Schnittstellen des SoMs verfügbar macht. Das System passt in einen DIN-Schacht (175 x 130 x 50 mm), was den Einsatz des Systems im *Kraftfahrzeug (Kfz)* erlaubt. Zum Gesamtsystem gehört weiterhin ein digitales Display mit einer Auflösung von 800 x 480 Punkten, das in seinem Gehäuse auch die Spannungsversorgung und den Anschluss an den CAN-Bus eines Fahrzeugs beherbergt. Das SoM vereint ein Motorola *MPC5200b* SoC, einen Silicon Motion SM501 Mutlimedia Chip, der unter anderem eine *Graphics Processing Unit (GPU)* enthält, sowie 16 MiB Flash-Speicher und 64 MiB RAM. Das Basisboard führt im Wesentlichen die Schnittstellen des SoM nach außen und implementiert die dafür notwendigen Anschlüsse und Hardwarekomponenten. Unter anderem bietet das DKI: zweimal RS232, Ethernet, I^2C, GPIOs, CAN und PCI. Die Übertragung von Daten zwischen der DKI-Box und dem Display erfolgt mit einer *Low Voltage Differential Signaling (LVDS)*- und einer CAN-Verbindung. Abbildung 6.8a auf der nächsten Seite zeigt ein Blockdiagramm des Systems, während Abbildung 6.8b auf der nächsten Seite das SoM und das Basisboard darstellt.

[10] Heute TQ-Embedded



(a) Blockdiagramm DKI mit Display

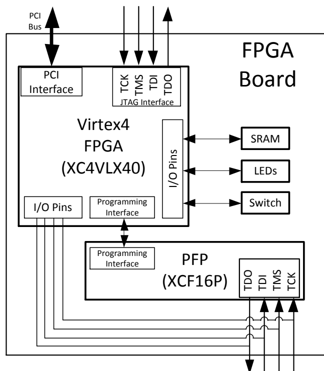


(b) DKI SoM und Basisboard

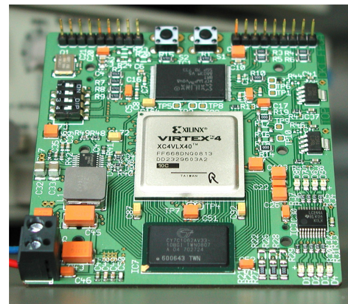
Abbildung 6.8: Schematische Darstellungen zur DKI-Box

FPGA-Extensionboard

In Abbildung 6.8b ist auch der ungenutzte PCI-Anschluss des Systems dargestellt. Dieser bietet die direkte Verbindung zum PCI-Bus des MPC5200B. Um die Flexibilität des DKI weiter zu erhöhen, entstand eine als *FPGA-Extensionboard* bezeichnete Erweiterungskarte. Die Karte ist in Abbildung 6.9 dargestellt.



(a) FPGA-Extensionboard (Schema)



(b) FPGA-Extensionboard (Real)

Abbildung 6.9: FPGA-Erweiterungsboard

Hauptkomponente ist ein Xilinx FPGA der Virtex-4 Serie (XC4VLX40). Dieses FPGA bietet hinreichend Ressourcen für eine Vielzahl von Experimenten. Als Konfigurationsspeicher befindet sich ein Xilinx PFP *XCF16P* auf der Karte. Außerdem hat das FPGA Zugriff auf eine kleinere Menge lokalen SRAMs. Weiterhin finden sich vier LEDs, DIP-Schalter und je ein JTAG-Interface für das FPGA und den PFP. Neben der PCI-Verbindung sind eine I^2C-Schnittstelle und zwei GPIO-Pins verfügbar. Der Systemtakt für das FPGA wird von einem externen 100 MHz Quartz bereitgestellt.

Um das System so kompakt wie möglich zu halten, ist das FPGA direkt mit dem PCI-Bus verbunden. Auf spezielle Bauteile für die Busanbindung wurde verzichtet. Diese Entwurfsentscheidung hat direkte Auswirkungen auf das FPGA-Design (vgl. Abschnitt 6.4.3). Einige I/O-Pins des FPGAs sind mit dem JTAG-Interface des PFPs verbunden, um so die dynamische Rekonfiguration des PROMs zu ermöglichen. Die extern erreichbaren JTAG-Interfaces zum FPGA und zum PFP dienen hier lediglich Debugging-Zwecken.

6.4.2 Software und Entwurfsumgebung

Das DKI-System fand bei einer großen Anzahl von Anwendungen Verwendung. Dementsprechend vielfältig sind sowohl die Auswahl an verwendeter Software als auch die zu deren Umsetzung verwendeten Arbeitsabläufe. In allen Fällen kamen als Bootloader „Das U-Boot“ und als Betriebssystem Linux zum Einsatz. Wie die anderen Systeme integrierte sich das DKI in die allgemeine Entwicklungsumgebung (vgl. Abschnitt 6.1.1).

Software

Obwohl die Kernkomponenten auf einem SoM zusammengefasst sind, ist das Gesamtsystem als komplett kundenspezifisch anzusehen. Dies spiegelt sich auch bei der Unterstützung der Komponenten durch den Bootloader und der Kernel wieder.

Das TQM5200 ist grundsätzlich in den Bootloader integriert. Allerdings wird eine systemspezifische Konfiguration benötigt, die die detaillierte Kenntnis

der Hardware voraussetzt. Für die bequeme Nutzung des Displays wurde der Bootloader um zusätzliche Kommandos erweitert.

Das verwendete SoM wird ebenso wie das darauf eingesetzte SoC weitgehend vom Linux-Kernel unterstützt. Wie beim Bootloader sind jedoch für das konkrete System (DKI) Anpassungen in der Konfiguration und im Quellcode des Kernel notwendig.

Im Rahmen des projektspezifischen Softwareentwurfs kamen neben den Programmiersprachen C und C++ auch Python [KE12], Tcl/Tk [RT99] und Java [Ull12] zum Einsatz.

Entwurfsumgebung und Arbeitsabläufe

Für die Anwendung „Connected CarBox mit 802.15.4“ (Abschnitt 6.4.4), die zusammen mit der Unicontrol entwickelt wurde, kam die Firmware der Unicontrol zum Einsatz. Ein Zugriff auf die Firmware oder die mit ihr verbundenen Werkzeuge bestand nicht. Die notwendigen Software-Erweiterungen wurden im Quellcode übergeben und durch Mitarbeiter der Firma in das bestehende System integriert. Zur IPC diente Shared Memory, wobei nur die für den zu erweiternden Teil notwendigen Strukturen bekannt waren. Dies erschwerte den Test, sodass eine besondere Teststrategie notwendig wurde. Der Quellcode wurde durch die Verwendung von Standardfunktionen und -bibliotheken sowie den Einsatz spezieller Makros so gestaltet, dass er sich auch auf einem PC nutzen ließ. Damit konnte ein prinzipieller Funktionsnachweis erbracht werden. Fehler, die auf die Unterschiede zwischen Entwurfssystem (x86, Little-Endian) und der Laufzeitumgebung (PowerPC, Big-Endian) zurückgingen, konnten nur durch Fehlerbeschreibungen der Bearbeiter bei der Unicontrol und detaillierte Analyse des Quellcodes gefunden werden.

Für eigene Projekte erfolgte zuerst die Integration der DKI-Box in die Entwurfsumgebung des XUP-V2P. Dieser Ansatz ist besonders in Hinblick auf die Verwendung des FPGA-Extensionboards von Vorteil. Allerdings bestehen zwischen dem Extensionboard und dem Linux-System beim DKI keine direkten Abhängigkeiten, da das Board als „normaler“ Teilnehmer am PCI-Bus erscheint. Die von OpenWrt gebotene Softwareauswahl war nicht für alle Anwendungen geeignet. Daher wurde parallel PTXdist als alternatives Buildsystem eingeführt. Die parallele Nutzung zweier Buildsysteme wird durch das in

Kapitel 5 vorgestellte Spezifikationssystem deutlich vereinfacht, was den einfachen Wechsel zwischen den Systemen erlaubt.

Für das schnelle Entwickeln von Prototypen und Demonstratoren erfolgt an vielen Stellen der Einsatz von plattformunabhängiger Programmiersprachen. Je nach Anforderung werden dabei Python, Java und Tcl/Tk eingesetzt. Die Vorteile dieser Methode wurden bereits in Abschnitt 4.6 dargelegt. Im Zusammenhang mit dem DKI hat die Methode jedoch einen weiteren Vorteil. Als Entwicklungssystem kommen in den meisten Fällen PCs mit x86-Architektur zum Einsatz, bei der als Byte-Reihenfolge Little-Endian einsetzt wird. Beim PowerPC erfolgt die Ablage von Werten jedoch nach dem Big-Endian-Prinzip. Das führt gerade bei der Verwendung der Programmiersprache C schnell zu Fehlern. Der Einsatz von Skriptsprachen bot hier einen pragmatischen Ausweg.

Der Hardwareentwurf für das FPGA erfolgt mit den von Xilinx bereitgestellten Werkzeugen. Speziell kommen die *Integrated Software Environment Design Suite (ISE)* und das EDK in Version 10 und 12 zum Einsatz, was in [Pan10] weiter ausgeführt ist.

6.4.3 Systemkonzept und -entwurf

Das DKI-System aus Box und Display stellt ein leistungsfähiges eingebettetes System für den Einsatz im Automobil dar. Sein ursprünglicher Verwendungszweck sah einen Einsatz als vollgrafisches, frei programmierbares Kombiinstrument im Kfz vor. Sein Design und die Flexibilität des genutzten Betriebssystems beschränken es jedoch nicht auf diesen Bereich. Das derzeitige Basisboard ist jedoch für den Einsatz im Fahrzeug optimiert, was sich beispielsweise in den ausgeführten Schnittstellen (CAN und I^2C, aber kein USB) widerspiegelt.

Die Besonderheit bei der DKI-Box mit Extensionboard sind die Möglichkeiten zur dynamischen und dynamisch-partiellen Rekonfiguration. Die Entwicklung des FPGA-Extensionboards hatte diese Einsatzfälle von Beginn an zum Ziel. Dementsprechend wurde bereits bei der Spezifikation und Entwicklung auf geeignete Konzepte geachtet. Maßgebliche Beschränkungen waren

die vorgegebene Bauform und das begrenzte Budget. Die nachfolgenden Erläuterungen können anhand von Abbildung 6.9a auf Seite 169 nachvollzogen werden.

Das FPGA ist direkt an den PCI-Bus angebunden. Daher muss immer ein funktionsfähiger PCI-IP-Core (sog. PCI-Bridge) in das FPGA-Design integriert werden. Aus diesem Grund benötigt das Extensionboard einen PFP, in dem sich eine entsprechende initiale Konfiguration befindet. Da das FPGA Zugriff auf den JTAG-Port des PROMs besitzt, kann es den Bitstrom im PFP neu schreiben. Leider unterstützt die Virtex-4-Serie noch keine Fallback-Konfiguration, so dass die Konfiguration im PROM funktionsfähig sein muss. Aus Platzgründen ist auch der Zugriff auf die `MODE`-Pins (vgl. [Xil08e, S. 13]) zur Nutzung mehrerer Bitströme in einem PFP nicht möglich. Da das FPGA aber über den ICAP dynamisch-partiell rekonfiguriert werden kann, ist diese Einschränkung nicht gravierend. Abbildung 6.10 zeigt das finale Zusammenspiel von Software auf dem Host-PC, der PCI-Bridge und den Modulen für die dynamische und die dynamisch-partielle Rekonfiguration. Die PCI-Bridge bietet drei logische Schnittstellen, so dass jedes Modul einzeln erreichbar ist. Die Design-Teile werden in den folgenden Abschnitten näher erläutert.

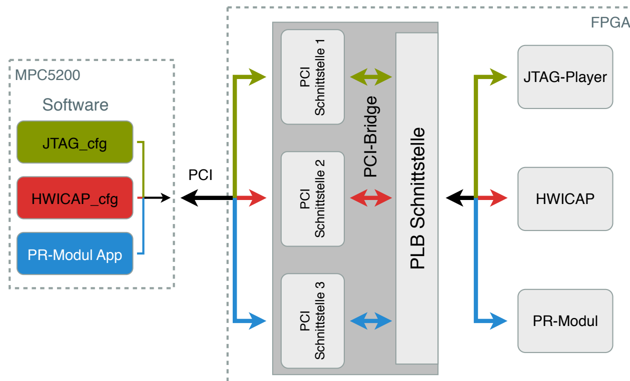


Abbildung 6.10: Zusammenspiel der Module für die RTR beim DKI

Konfiguration des Platform Flash PROM Wie bereits erläutert, hat das FPGA Zugriff auf den JTAG-Port des PFP. Voraussetzung für die Programmierung des PFPs mit Hilfe des FPGAs ist ein IP-Core, der die notwendigen JTAG-Kommandos erzeugt. Xilinx erläutert in verschiedenen *Xilinx Application Notes (XAPPs)* Lösungsansätze für die Programmierung eines PROMs [Xil07b; WK08; HP08]. Der Ansatz aus XAPP975 [HP08] kommt der hier benötigten Lösung am nächsten und dient als Grundlage für die Implementierung.

Kern der Umsetzung ist der von Xilinx beschriebene JTAG-IP-Core, im Folgenden als JTAG-Player bezeichnet. Er übernimmt die drei wesentlichen Aufgaben:

1. PROM löschen,

2. neuen Bitstrom in den PROM schreiben und

3. Verifikation der übertragenen Daten.

Zur Integration in das Extensionboard wurden einige Anpassungen und Erweiterungen am Xilinx-Design durchgeführt. Abbildung 6.11 zeigt das finale System.

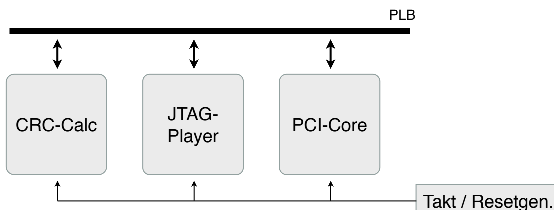


Abbildung 6.11: Rahmendesign des JTAG-Players

Per Definition verwenden alle IP-Cores im Extensionboard den PLB zur Kommunikation. Somit wurde der JTAG-Player um die notwendige PLB-Logik erweitert. Der JTAG-Player arbeitet mit einer maximalen Frequenz von 10 MHz. Da der Systemtakt 100 MHz beträgt, ist eine entsprechende Taktanpassung notwendig. Die Kommunikation mit dem Host-System erfolgt über Status-

und Kontrollregister, die ebenfalls in das Design integriert wurden. Die Verifikation erfolgt mit Hilfe einer 8-Bit Prüfsumme. Diese Prüfsumme ist im originalen Bitstrom enthalten und wird im FPGA gespeichert. Nach dem Schreiben werden die Daten aus dem PFP zurückgelesen. Während dieses Vorgangs wird eine Prüfsumme gebildet und mit der ursprünglichen verglichen. Nur wenn beide Prüfsummen übereinstimmen, ist die Konfiguration erfolgreich. Die Berechnung der Prüfsumme erfolgt direkt im FPGA.

Die Implementierung unterliegt einigen Einschränkungen, die für das Extensionboard jedoch keine Nachteile mitbringen. Im Einzelnen sind dies:

- FPGA und PROM dürfen sich nicht in der selben JTAG-Kette befinden.
- In der vom PFP-Modul betriebenen JTAG-Kette darf sich außer dem PFP kein weiteres Gerät befinden.
- Es werden nur die Datenfelder des PROM ab Startadresse 0×00 beschrieben.
- Die internen Konfigurationsregister des PFP werden nicht verändert.

Aufgrund der vereinfachten Umsetzung benötigt das gesamte Design lediglich etwa 4 % der FPGA-Ressourcen. [KPH10] und [Pan10] liefern weitere Details zur Implementierung des JTAG-Players.

Die Rekonfiguration wird vom Host-PC aus mit einer einfachen User Space Anwendung gesteuert, die auf die `mmap( )`-Funktion zurückgreift. Sie nutzt die mit dem JTAG-Player assoziierte PCI-Schnittstelle. Der Konfigurationsvorgang inklusive Verifikation benötigt etwa 30 Sekunden. Eine Umsetzung als Kernel-Treiber hat sich aufgrund der geringen Ansprüche an die Konfigurationsgeschwindigkeit nicht angeboten.

Wie bei der vollständigen Rekonfiguration üblich (vgl. Abschnitt 2.5.2), wird ein neues Design erst nach einem Reset des FPGAs aus dem PFP übernommen. Danach befindet sich der IC in seinem initialen Zustand. Somit sind auch die PCI-Schnittstellen nicht mehr konfiguriert. Erst nach einer erneuten Initialisierung aller Busteilnehmer ist die Kommunikation wieder möglich.

Dynamisch-Partielle Rekonfiguration Wie die dynamische war auch die dynamisch-partielle Rekonfiguration Ziel beim Entwurf des FPGA-Extensionboards. Mit der Einführung der ISE Version 12 hat Xilinx die damit verbundenen Arbeitsabläufe deutlich vereinfacht (vgl. Abschnitt 2.5.2). Dem Ansatz von Xilinx folgend, wird auch für das Extensionboard der Design-Flow nach [Xil12b] genutzt. Dazu erfolgt die Integration des ICAP, der die Schnittstelle zwischen dem PLB und den Konfigurationsregistern des FPGAs darstellt, in das Design. Außerdem muss ein IP-Core existieren, der die PR-Module aufnimmt und sie ebenfalls über den PLB zugänglich macht. Dieser IP-Core richtet sich natürlich nach den Notwendigkeiten der PR-Module. Das entstehende Design ist in Abbildung 6.12 dargestellt. Aufgrund des geringen Ressourcenbedarfs wurde der JTAG-Player beibehalten. Somit kann auch der statische Teil des FPGA-Designs jederzeit geändert werden.

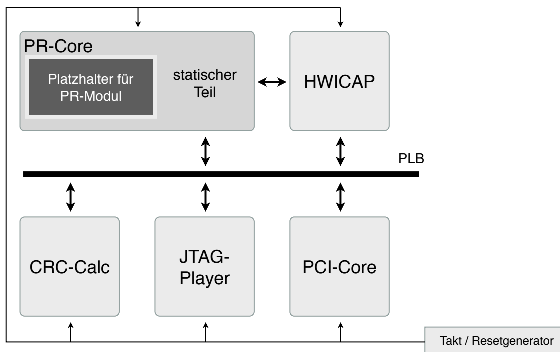


Abbildung 6.12: Blockschaltbild des dpR-Designs

Als einfaches Beispiel dient ein Rechner, der erst die Rechenoperation (Addition, Subtraktion, Multiplikation, Vergleich) in das FPGA lädt, dann die Operanden und das Ergebnis zurückliest. Für die Konfiguration und die Kommunikation wurden verschiedene PCI-Schnittstellen vorgesehen (vgl. Abbildung 6.10 auf Seite 173), was den Aufbau des statischen Teils vereinfacht. Konfiguration und Kommunikation werden von einem Programm im User Space gesteuert. Es liest zuerst die Operanden und die Operation, konfiguriert das passende PR-Modul, überträgt beide Operanden und zeigt nach erfolgter Operation das Ergebnis an.

Bei der dpR bleibt der statische Teil des Designs, der auch die PCI-Bridge umfasst, intakt. Daher gibt es keine Nebenwirkungen mit dem PCI-Bus. Die Konfigurationszeit ist durch die Größe des PR-Moduls und die Geschwindigkeit des PCI-Busses begrenzt. Im vorliegenden Fall arbeitet der Bus mit einem Takt von 33 MHz. Die Konfiguration im oben genannten Beispiel dauerte somit 2,4729 ms[11]. Weitere Details zur Implementierung und zum Beispiel liefert [Pan10].

6.4.4 Beispielanwendungen

Das DKI-System wurde in mehreren Szenarien eingesetzt. In der Mehrzahl der Fälle kam es dabei als Gateway zur Kommunikation mit dem CAN-Bus von Fahrzeugen zum Einsatz.

Basisanwendung Dashboard

Um den vollen Funktionsumfang des DKI-Systems verfügbar zu machen, kommen als Alternative zum proprietären Betriebssystem der Unicontrol GmbH ein selbst erstelltes Linux-System und eine angepasste Version des Bootloaders „Das U-Boot“ zum Einsatz. Trotzdem sollte die Grundfunktion eines Digitales Kombiinstrument mit verschiedenen Zustandsinformationen erhalten bleiben. Abbildung 6.13 auf der nächsten Seite vermittelt einen Eindruck der grafischen Anzeige im Display des DKI. Zu den angezeigten Informationen gehören unter anderem:

- die aktuelle Geschwindigkeit,
- die aktuelle Motordrehzahl,
- die aktuelle Uhrzeit,
- der aktuelle Kilometerstand (Tages- und Gesamtkilometer),
- Tankfüllstand,
- Kühlwassertemperatur sowie

[11] Es müssen 32973 Byte für das PR-Modul und 16 Kommando-Bytes übertragen werden.

- Zustand der Beleuchtungsanlage und der Blinker.

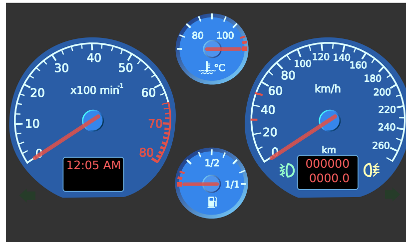


Abbildung 6.13: Grafische Darstellung im Display des DKIs

Die Implementierung eines ersten Prototyps erfolgte mit Python, Tk und PyTk[12]. Die Implementierung nutzt die Vorteile des Softwareentwurfs per Skriptsprache. Zeitkritische Teile sind in Bibliotheken ausgelagert, die mit der Programmiersprache C implementiert wurden. Unter anderem betrifft dies die Programmteile, die auf den CAN-Bus zugreifen.

Die Lösung mit PyTk setzt das X-Window System voraus. Dadurch besitzt dieser Ansatz verschiedene Nachteile. Vom Einschalten bis zum Start der Dashboardanwendung vergehen mehr als 90 Sekunden. Außerdem belegt das X-Window System zusätzliche Ressourcen, besonders RAM und Flash, aber auch CPU-Zyklen.

Eine deutliche Verbesserung konnte mit der Umstellung der Dashboardanwendung auf PyQt[13] erzielt werden. Hier konnte durch Nutzung des Kernel-Framebuffers auf das X-Windows System verzichtet werden. Damit einher ging nicht nur eine leichte Verbesserung der Startzeit (auf unter 75 Sekunden), sondern auch eine Verringerung des benötigten RAM.

Connected CarBox mit 802.15.4

Die Kommunikation zwischen Fahrzeugen bzw. zwischen Fahrzeug und Umwelt ist bereits seit mehreren Jahren Thema bei Automobilherstellern und in Forschungsprojekten. Obwohl der IEEE-Standard 802.11p [GC09] für die

[12] PyTk ist eine Bibliothek, die die grafischen Elemente von Tk per Python nutzbar macht.
[13] PyQt ist eine Bibliothek, die die Funktionen der Qt-Bibliotheken per Python nutzbar macht.

Fahrzeugkommunikation ausgelegt ist, sind auch Untersuchungen alternativer Technologien notwendig. Das Interesse in Zusammenhang mit den Arbeiten am DKI galt dabei den Protokollen aus dem Nahbereichsfunk, speziell denen nach dem IEEE-Standard 802.15.4. Eigene Untersuchungen (vgl. [RFH07; Fre11]) zeigen, dass verfügbare Module sehr gute Eigenschaften für den Einsatz in mobilen Szenarien aufweisen. Die hardwarebasierte Distanzmessung zwischen Funkknoten nach dem IEEE-Standard 802.15.4a ist eine attraktive Zusatzfunktionalität.

Die DKI-Box wurde daher durch entsprechende Funkmodule erweitert. In der Beispielanwendung aus dem Umfeld der *Car-to-Roadside* Kommunikation überträgt ein als *Roadside-Unit* bezeichnetes, stationäres System am Straßenrand Informationen an vorbeifahrende Fahrzeuge. Die Fahrzeuge ihrerseits liefern Statusinformationen an die Roadside-Unit. Tabelle 6.4 zeigt, welche Daten im Beispiel übertragen wurden.

Tabelle 6.4: Übertragene Daten in der Beispielanwendung

Vom Fahrzeug gesendet	Von der Roadside-Unit gesendet
Fahrzeugtyp und -identifikation	Roadside-Unit-ID
Geschwindigkeit	Zulässige Höchstgeschwindigkeit
Motordrehzahl	
Außentemperatur	Eventuelle Warnungen
GPS-Position und Fahrtrichtung des Fahrzeugs	

Die Firma Unicontrol GmbH, die das Basissystem der DKI-Box entwickelt und bereitgestellt hat, verfügt über ein Fahrzeug mit integriertem DKI. Dieses Fahrzeug diente als erster Versuchsträger. Nach erfolgreicher Umsetzung wurde die Lösung für die Forschungsfahrzeuge der Professur Schaltkreis- und Systementwurf (BMW 754d und BMW X5) angepasst und dort ebenfalls integriert. Mit den Fahrzeugen wurden verschiedene Tests durchgeführt. Die erfolgreiche Datenübertragung konnte bis zu einer Geschwindigkeit von 70 $\frac{km}{h}$ nachgewiesen werden. Sowohl an den Fahrzeugen als auch an der Roadside-Unit kamen einfache Stabantennen ohne spezielle Richtcharakteristik zum Einsatz.

Connected CarBox mit Ethernet

Die Car-to-Roadside Anwendung hat zwei generelle Nachteile. Zum einen bietet das verwendete Funksystem nur eine sehr geringe Übertragungsrate von 2 Mbit (Netto), was perspektivisch nur wenig Spielraum lässt. Zum anderen ist die Installation einer hinreichenden Menge von Roadside-Units erforderlich. Eine offensichtliche Alternative sind zellulare Mobilfunknetze wie *GSM*, *UMTS* oder *LTE*. Sie bieten eine höhere Bandbreite und sind inzwischen nahezu flächendeckend verfügbar. Die benötigte Kommunikationshardware ist preiswert und in verschiedensten Varianten verfügbar.

Mit der weiteren Integration der DKI-Box in die Versuchsfahrzeuge steigt die verfügbare Datenmenge. Neben Fahrzeugdaten können über externe Sensoren beispielsweise Informationen über den Fahrer gewonnen werden, z. B. Vitaldaten. Zur Verteilung dieser Daten ist eine leistungsfähigere Kommunikationsverbindung notwendig. Allerdings bietet die DKI-Box außer Ethernet und PCI-Bus keine Schnittstelle mit hoher Datenrate. Für eine Erweiterung ist somit ein zusätzliches Gateway notwendig. Ein ebensolcher Aufbau wurde realisiert und ist schematisch in Abbildung 6.14 dargestellt.

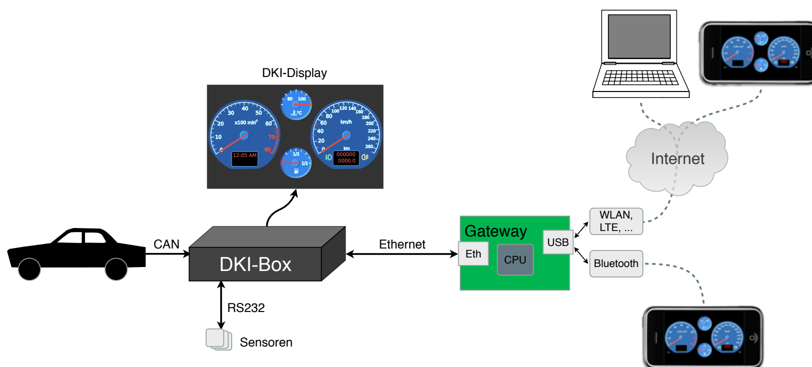


Abbildung 6.14: Schematische Darstellung des Fahrzeugdemonstrators mit Breitbandanbindung

Hier fungiert das DKI als Gateway zum CAN-Bus des Fahrzeugs und bindet weitere externe Sensoren an. Die erhobenen Daten werden sowohl im Display

des DKI-Systems dargestellt, als auch per Ethernet an ein weiteres Gateway übertragen. Das zweite Gateway ist per USB flexibel erweiterbar und bietet Zugang zu weiteren Netzwerken wie *Wireless Local Area Network (WLAN)* oder *LTE*, ebenso wie zum Nahbereichsfunk, z. B. per Bluetooth.

Auf diese Weise können verschiedene Wege zur Kommunikation mit dem Fahrzeug genutzt werden. Zum einen existiert eine *comming home* Funktion. Erkennt das System ein WLAN der TU Chemnitz, überträgt es selbstständig alle neu erfassten Sensorwerte auf einen Server im Netzwerk der Universität. Außerdem werden Statusinformationen in Form von *Activity Streams* via Twitter bereitgestellt. Je nach Situation ist auch eine aktive Verbindung zum Fahrzeug möglich, um beispielsweise zur Diagnose Daten aktiv abzufragen.

6.5 Raspberry Pi

Der etwa Scheckkarten große „Einplatinencomputer“ *Raspberry Pi* [HU12] wird seit Anfang 2012 als Serienprodukt von der Raspberry Pi Foundation zu Preisen von 25 US-$ (Modell A) bzw. 35 US-$ (Modell B) vertrieben. Ursprünglich sollte der Rechner als preiswerte Experimentierplattform für interessierte Jugendliche dienen. Das System stieß jedoch auf große Akzeptanz[14] und erhielt ein entsprechendes Medienecho. Mithin gilt der Raspberry Pi als Ursprung für eine Serie von preiswerten Kleinstrechnern verschiedener Hersteller.

Auch Linux hat durch den Erfolg des Raspberry Pi bzw. der preiswerten Kleinstrechner mit geringen Ressourcen profitiert. Die schnelle Verbreitung und der rege Zuspruch auch von Nutzern mit wenig oder keiner Linux-Erfahrung führt zur Portierung der bisher nur wenig beachteten ARMv6-Architektur. So stehen inzwischen bekannte Distributionen mit entsprechend großen Nutzergemeinden wie:

- Fedora (Pidora),
- Debian (Raspbian, die offiziell empfohlene Distribution der Raspberry Pi Foundation),

[14] Bis zum Herbst 2013 wurden 2 Mio. Stück ausgeliefert.

- OpenSUSE und
- ArchLinux

bereit. Auch die verschiedenen Buildsysteme, unter anderem OpenWrt, PTX-dist, OpenEmbedded und Buildroot unterstützen den Raspberry Pi.

Der geringe Anschaffungspreis[15], die vorhandene Software und die große Nutzergemeinde lassen den Raspberry Pi als geeignetes System für Prototypen und Kleinserien eingebetteter Systeme erscheinen. Es sollte jedoch vor der Benutzung klar sein, dass derart günstige Geräte mit entsprechend preiswerten Bauelementen bestückt sind. Nutzerberichte im Internet wie auch eigene Erfahrungen zeigen, dass dies mit Qualitätseinbußen einhergeht, was sich beispielsweise in einer erhöhten Serienstreuung zeigt.

6.5.1 Hardware

Für verschiedene prototypische Umsetzungen wurden 13 Raspberry Pi[16] untersucht und genutzt. Aufgrund dieser Anzahl können auch Aussagen zu Unterschieden zwischen den Systemen getroffen werden. Die wesentlichen Hardware-Komponenten des Raspberry Pi sind:

- das SoC BCM 2835 von Broadcom, unter anderem mit:
  - einem ARM1176JZF-S Prozessor und
  - einem Broadcom VideoCore IV,
- HDMI- und FBAS-Ausgänge für die Videoausgabe,
- einen CSI-Videoeingang,
- 512 MiB RAM,
- einen Steckplatz für eine SD-Karte,
- einen 10/100 Mbit Ethernet-Anschluss,

[15] Raspberry Pi (Modell B) inklusive Gehäuse, passendes Netzteil und 4 GB-SD-Karte erhält man schon ab etwa 45 €.

[16] Alles Modell B-Systeme, drei ältere mit 256 MiB RAM und zehn der hier erläuterten zweiten Generation.

- zwei USB-Anschlüsse, die mit maximal je 100 mA belastbar sind und
- eine Stiftleiste mit GPIO-Pins, I^2C, SPI, UART.

Beim Modell A wird zugunsten einer geringeren Stromaufnahme auf den Ethernet-Anschluss und einen der zwei USB-Anschlüsse verzichtet. Beiden Modellen fehlt eine gepufferte Echtzeituhr, so dass nach jedem Neustart eine Synchronisation mit einem Zeitserver notwendig ist. Das setzt allerdings eine funktionsfähige Netzwerkverbindung voraus.

Die Stiftleiste erlaubt durch die Bereitstellung gängiger Schnittstellen eine bequeme Erweiterung des Raspberry Pi. Daher entstanden im Zusammenhang mit den Beispielanwendungen (vgl. Abschnitt 6.5.3) zwei Erweiterungsplatinen.

Die Erste stellt eine gepufferte Echtzeituhr und einen Schalter für USB-Geräte bereit. Die Echtzeituhr vom Typ DS3232 kann vom Raspberry Pi per I^2C angesprochen werden. Der Linux-Kernel bietet einen entsprechenden Treiber, so dass kein zusätzlicher Aufwand entsteht. In Verbindung mit einem kleinen Energiespeicher bleibt die Zeit beim Reboot oder bei kurzen Stromausfällen auch ohne Netzwerkverbindung erhalten. Auf der Platine ist außerdem ein digitaler Schalter (MAX1607), der es erlaubt eine USB-Verbindung mit Hilfe eines 3,3 V TTL-Ausgangs zu schalten. Abbildung 6.15 zeigt ein Blockschaltbild der Platine.

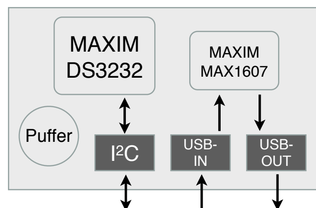


Abbildung 6.15: Blockschaltbild der Erweiterungsplatine mit Echtzeituhr und USB-Schalter

Die Möglichkeiten zur Aufnahme von Messwerten mit dem Raspberry Pi sind sehr begrenzt. Daher entstand eine Erweiterungsplatine, die verschiedene Kanäle für die Erfassung von analogen und digitalen Messwerten bietet.

Auch das Erzeugen von Werten mit mehr als nur 3,3 V-Pegel ist möglich. Außerdem bietet der verwendete Mikrocontroller (ATXmega128) mehrere serielle Schnittstellen, die ebenfalls zur Verfügung stehen. Abbildung 6.16 zeigt ein vereinfachtes Blockschaltbild der Platine.

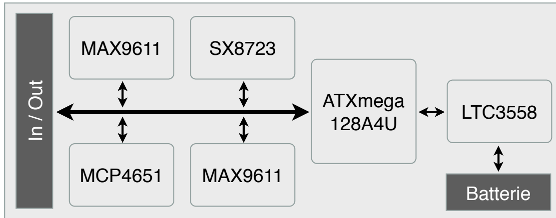


Abbildung 6.16: Blockschaltbild der Messwert-Platine

Die Kommunikation mit dem Raspberry Pi erfolgt über eine UART Verbindung, was die Datenrate auf 115 $\frac{Kib}{s}$[17] begrenzt. Der Mikrocontroller hat jedoch hinreichend Reserven, um auch Vorverarbeitungsaufgaben zu übernehmen. Dazu muss der Raspberry Pi Kommandos an den Mikrocontroller senden. Zu diesem Zweck entstand ein einfaches Kommunikationsprotokoll, das neben der Übertragung von Messdaten auch den Austausch von Kommandoinformationen erlaubt. Weitere Details zur Umsetzung der Leiterplatte und des Kommunikationsprotokolls finden sich in [Len13].

6.5.2 Systemkonzept und -entwurf

Wie in den nachfolgenden Beispielen (vgl. Abschnitt 6.5.3) weiter ausgeführt wird, liegen den Untersuchungen zwei sehr unterschiedliche Anwendungsfälle zugrunde, die zu weitgehend getrennten Vorgehensweisen führen. Beide Fälle erfordern eine Erweiterungsplatine, die konzeptioniert und umgesetzt wurde. Für die Überwachung des *Long Term Evolution (LTE)*-Netzwerkes musste zusätzlich das Überwachungssystem entwickelt werden, während bei der Instrumentierung des E-Bikes Aufwand in die Verringerung des Energiebedarfs fließt.

[17] Versuche, die Datenrate auf 1 $\frac{Mib}{s}$ zu erhöhen, sind mit dem verwendeten Raspberry Pi nicht geglückt.

Ebenso unterscheidet sich die Vorgehensweise bei der Softwarearchitektur. Im Fall des E-Bikes kommt ein angepasstes, per Buildsystem erstelltes Linux zum Einsatz. Das ermöglicht die größtmögliche Einflussnahme auf die ausgeführten Systemdienste, was zur Verringerung der Bootzeit ebenso beiträgt, wie zur Verringerung der Energieaufnahme. Das Linux-System hat eine sehr scharf abgegrenzte Aufgabe und es werden nahezu keine „Kompfortdienste“ benötigt. Da weite Teile der Software des Raspberry Pi mit Python umgesetzt werden, kann die Softwareentwicklung bequem am PC erfolgen.

Für das LTE-Überwachungssystem kam bewusst Raspbian zum Einsatz. Das System ist auf den Desktopbetrieb ausgelegt und bietet entsprechende „Kompfortdienste“, wie das automatische Verbinden mit Netzwerken. Außerdem lässt ein fertiges Linux-System eine schnelle Fertigstellung des finalen Betriebssystems erwarten. Weitere zu erwartende Vorteile sind die Pflege von Paketen durch die Distribution, die Verfügbarkeit von Software und eine breite Nutzerbasis. Nachteile wie die längere Bootzeit, der erhöhte Platzbedarf auf dem Festspeicher und die voreingestellten Systemdienste erscheinen im Kontext der Anwendung als unkritisch.

6.5.3 Beispielanwendung

Wie bereits erläutert, stellt sich die Frage, inwieweit sich der Raspberry Pi für Prototypen und Kleinserien eignet. Zwei Beispiele erlauben erste Aussagen zu dieser Fragestellung.

E-Bike Überwachung und Steuerung

In [Kri+12] wird am Beispiel eines E-Bikes ein Algorithmus für die Vorhersage und Optimierung der Reichweite von Elektromobilen vorgestellt. Für die Entwicklung und Verifikation des Algorithmus kommt ein instrumentiertes Fahrrad zum Einsatz. Das ursprüngliche System besteht aus einem Mikrocontroller-basierten Messsystem, das die erhobenen Daten aufzeichnet und Offline verfügbar macht. Die Kommunikation mit einem angebundenen Smartphone umfasst nur die zur Korrektur der Schätzungen notwendigen Daten.

Weitergehende Untersuchungen sowie die Anwendung des Systems in anderen Elektromobilen (andere Fahrräder, Motorräder, Autos) sind mit dieser Hardware nur schwer möglich. Eine auf dem Raspberry Pi aufbauende Plattform hingegen bietet hinreichend Rechenleistung, um die Algorithmen direkt im Fahrzeug anzuwenden. Dabei ermöglicht das auf dem Raspberry Pi verwendete Linux-System die Nutzung der zugrundeliegenden Modellierungs- und Entwicklungsumgebung. Eine Portierung von Entwicklungsversionen bzw. Zwischenständen der Algorithmen auf ein Smartphone ist so nicht mehr notwendig.

Für die Mehrzahl der Fahrzeuge ist weiterhin eine Instrumentierung mit zusätzlichen Sensoren notwendig (vgl. auch [Win11]). Für die Kommunikation mit diesen Sensoren dient eine eigens zu diesem Zweck entwickelte Erweiterungsplatine für den Raspberry Pi (vgl. Abbildung 6.16).

Tabelle 6.5: Werkseitig eingestellte Werte für Frequenzen und Spannungen

Bezeichnung	Wert	Min.	Max.
ARM-Kern	700 MHz	700 MHz	700 MHz
GPU	250 MHz	250 MHz	250 MHz
GPU- Subfrequenzen	250 MHz	250 MHz	250 MHz
RAM	400 MHz	400 MHz	400 MHz
Kernspannung	1,2 V	1,2 V	1,2 V
RAM-Spannung	1,2 V	1,2 V	1,2 V

Um die Messwerte am Elektromobil nicht zu verfälschen, werden Raspberry Pi und Erweiterungsplatine aus einer eigenen Batterie versorgt. In diesem Zusammenhang sind die Leistungsaufnahme des Systems sowie die Möglichkeiten zu deren Reduktion von Interesse. Beim Modell B sind die Hauptverbraucher das SoC, der auf der Platine integrierte Schnittstellen-IC für Ethernet und den zweiten USB-Anschluss sowie der Spannungswandler zur Erzeugung der 3,3 V Betriebsspannung aus der 5 V Eingangsspannung. Die Leistungsaufnahme von Schnittstellencontroller und Spannungswandler sind nur mit erheblichem Aufwand zu reduzieren. Die Leistungsaufnahme des SoC kann hingegen per Software beeinflusst werden. Verschiedene Frequenzen (z. B. die des ARM-Kerns, des RAMs und der GPU) können mit Hilfe einer Konfigurationsdatei

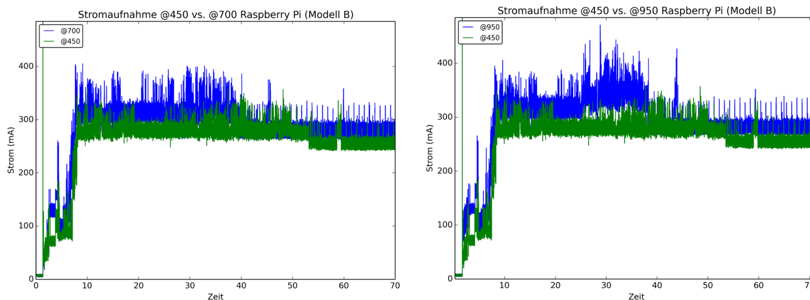
eingestellt werden. Ebenso ist die Variation der SoC-internen Betriebsspannung möglich.

Tabelle 6.6: Erprobte Werte für Frequenzen und Spannungen

Bezeichnung	Wertebereich	Max bei 450 MHz	Max bei 700 MHz	Max bei 950 MHz
ARM-Kern	200...1000 Mhz	450 MHz	700 MHz	950 MHz
GPU	160...500 MHz	180 MHz	250 MHz	250 MHz
GPU-Subfreq.	40...500 MHz	40 MHz	250 MHz	250 MHz
RAM	300...500 MHz	300 MHz	400 MHz	450 MHz
Kernspannung	1,0...1,35 V	1,05 V	1,2 V	1,35 V
RAM-Spannung	1,2...1,2 V	1,2 V	1,2 V	1,2 V

Ab Werk sind verschiedene Arbeitsbereiche eingestellt, die einen sicheren Betrieb des Systems ermöglichen. Unter anderem gelten die Grenzwerte in Tabelle 6.5 auf der vorherigen Seite. Aus der Tabelle geht ebenfalls hervor, dass keine dynamischen Anpassungen der Werte durch das Betriebssystem vorgesehen sind (da jeweils $Min = Max$), obwohl diese Funktion sowohl vom SoC als auch vom Linux-Kernel unterstützt wird. Während die meisten Quellen im Internet ein Heraufsetzen der Frequenzen und eine entsprechend notwendige Anpassung der Betriebsspannung diskutieren, erfordert das Einsparen von Energie das Herabsetzen von Frequenz und Spannung. Gesicherte Informationen zu den dabei zu nutzenden Werten existieren nicht. Eigene Versuche und Vergleiche mit den Erfahrungen anderer Nutzer führen zu den Werten, die in Tabelle 6.6 als „Wertebereich“ wiedergegeben sind.

Abbildung 6.17 auf der nächsten Seite zeigt die Stromaufnahme eines Raspberry Pi Modell B bei den in Tabelle 6.6 angegebenen Werten. Die Wahl der Werte beruht dabei auf den Erfahrungen, die mit zehn der Raspberry Pi gemacht und als hinreichend stabil eingeschätzt wurden. Eine Veränderung der werkseitig eingestellten Minimalwerte erfolgte nicht. Daher sind bei der Einstellung „@950 MHz“ die Auswirkungen der Stromsparfunktionen des Linux-Kernels zu erkennen. Trotz der verrauschten Signale lässt sich der Unterschied in der mittleren Stromaufnahme gut nachvollziehen. Weiter ist zu erkennen, dass am Ende der Bootsequenz die Stromaufnahme in beiden Fällen sinkt. Die Ver-



(a) Stromaufnahme des Raspberry Pi (Modell B) bei @450 MHz und @700 MHz

(b) Stromaufnahme des Raspberry Pi (Modell B) bei @450 MHz und @950 MHz

Abbildung 6.17: Stromaufnahme des Raspberry Pi (Modell B)

mutung liegt nahe, dass dies mit der SD-Karte in Verbindung steht, die während der Bootsequenz intensiv genutzt wird. Die Kurvenverläufe folgen dem gleichen Muster. Aufgrund der geringeren Geschwindigkeit hängt die grüne der blauen Kurve zeitlich etwas hinterher. Das Muster zeigt auch, dass sich das Ausführen bestimmter Programme in der Bootsequenz deutlicher auf den Stromverbrauch auswirkt. Wie zu erwarten erfolgt das Ausführen von Programmen bei höheren Frequenzen schneller. Damit bleibt die Frage bestehen, mit welcher Strategie der Betrieb energetisch günstiger ist. Diese Frage lässt sich nur mit Hilfe einer Analyse wie der hier gezeigten beantworten und ist von der jeweiligen Anwendung abhängig.

Die Erfahrungen des Autors mit der Serienstreuung zeigt, dass auch die in Tabelle 6.6 angegebenen Werte nicht immer störungsfreies Arbeiten ermöglichen. Bei häufigen Abstürzen oder ungewöhnlichem Verhalten von SD-Karte und USB führt eine Variation der Werte gegebenfalls zu einer Verbesserung. Speziell das Verändern der GPU-Core-Frequenz kann hier zu Problemen führen.

Überwachung eines LTE-Netzwerkes

Die TU Chemnitz verfügt über ein abgeschlossenes LTE-Netzwerk aus drei Basisstationen. Das Netz wird durch die E+-Gruppe zur Verfügung gestellt und kann für Forschungsaufgaben im Bereich der Mobilfunkkommunikation eingesetzt werden. Jede der drei Basisstationen hat je drei Antennen mit einem Öffnungswinkel von 120°. Somit gibt es neun Netzwerksektoren, die zusammen nahezu den gesamten Campus sowie Teile des Stadtgebietes mit LTE versorgen. Das Netzwerk kann nur von Angehörigen der TU Chemnitz genutzt werden, was die Nutzerzahl begrenzt. Daher ist der Zustand des Netzwerkes nicht jederzeit bekannt.

Ein preiswertes Überwachungssystem soll daher Informationen zum Zustand des Netzwerkes sammeln. Aufgrund der neun Sektoren sind wenigstens neun Überwachungseinheiten notwendig. Diese erfassen zyklisch sowohl passiv als auch aktiv Informationen zur Situation im Netzwerk. Passiv bedeutet dabei, dass lediglich die Daten erfasst werden, die die Verbindungshardware (LTE-Modem) über das Netzwerk hat. Das betrifft beispielsweise die Anzahl und Kennung der sichtbaren Nachbarzellen oder die Signalstärke der Zellen. Weitere Daten entstehen durch aktive Kommunikation innerhalb des Netzwerkes. Dazu werden Verbindungen zu verschiedenen Gegenstellen aufgebaut und Messwerte wie beispielsweise die Latenz oder die Datenrate der Kommunikationsverbindung aufgezeichnet. Der sich daraus ergebende, prinzipielle Aufbau ist in Abbildung 6.18 auf der nächsten Seite dargestellt.

Das Überwachungssystem muss mit Hilfe eines LTE-Modems und einiger Testprogramme zyklisch Messwerte erfassen, speichern und an einen zentralen Punkt weiterleiten. An diesem zentralen Punkt kann dann die Auswertung der Messwerte erfolgen, was eine Beurteilung des aktuellen Zustands des Netzwerks erlaubt. Das Netzwerk nutzt das 2,6 GHz-Band, das inzwischen auch von preisgünstigen USB-Sticks unterstützt wird. Um die Messwerte im LTE-Netzwerk nicht zu verfälschen, erfolgt die Übertragung der Daten zum zentralen Sammelpunkt möglichst über eine weitere Netzwerkverbindung, wobei sich je nach Aufstellungsort WLAN oder Ethernet anbieten. An die Messsysteme werden einige weitere Ansprüche gestellt. Es soll ein unterbrechungsfreier, autonomer Betrieb gewährleistet werden. Dazu ist es notwendig, die Hardware, die Software und die Netzwerkverbindungen zu überwachen und bei Bedarf zu aktivieren bzw. zu deaktivieren. Wartung und Updates müssen ohne

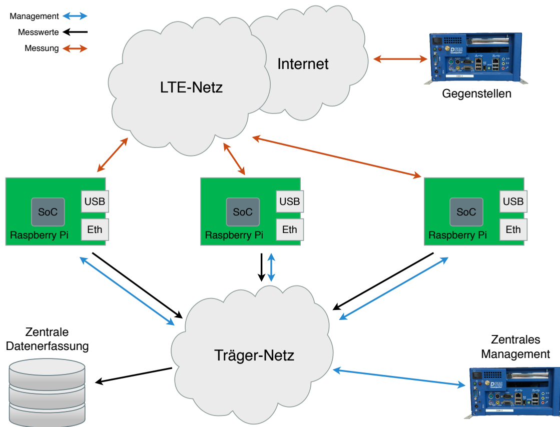
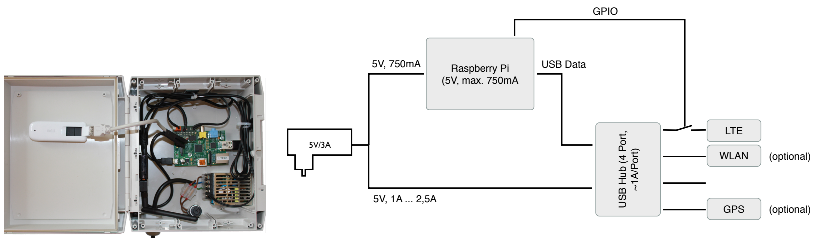


Abbildung 6.18: Prinzipieller Aufbau des LTE-Überwachungssystems

physischen Zugriff möglich sein und auch bei nur unregelmäßig vorhandener Netzwerkverbindung funktionieren. Für die Koordination und Durchführung von Messaufgaben ist ein zentrales Management der Stationen notwendig. Bereits diese grobe Aufstellung von Anforderungen zeigt, wie wichtig – aber auch komplex – die vollständige Spezifikation für das Überwachungssystem wird. Denn neben Hard- und Software umfasst es auch den zentralen Server sowie die Anforderungen an Wartung und Management.

Durch seine Eigenschaften erscheint das Modell B des Raspberry Pi als ideale Plattform für derartige Aufgaben: preisgünstig, klein, mit geringer Leistungsaufnahme und ausreichend Schnittstellen. Auch die weiteren Komponenten sind als preiswerte Consumer-Geräte verfügbar. Abbildung 6.19a auf der nächsten Seite zeigt ein fertig aufgebautes Messsystem mit Raspberry Pi, Erweiterungsplatine, USB-Hub und LTE- sowie WLAN-Stick. Der Preis für die Hardware beläuft sich auf etwa 150 €. Insgesamt existieren derzeit zehn dieser Messsysteme.

Als Betriebssystem für die Messsysteme kommt ein nahezu unverändertes Raspbian zum Einsatz. Das System wird von vielen Nutzern verwendet. Fehler sollten dementsprechend schnell erkannt werden. Berichte im Internet bescheinigen dem System hinreichende Aktualität und Stabilität auch bei Lang-



(a) Fertig montiertes LTE-Messsystem *(b)* LTE-Messsystem, Schematische Darstellung

Abbildung 6.19: LTE-Messsystem

zeitbetrieb. Die in Kapitel 4 diskutierten Einschränkungen bei der Verwendung von Distributionen trifft auch hier zu. So umfasst bereits das Basissystem ca. 1,8 GiB und bietet Dienste, die beim Überwachungssystem keine Anwendung finden. Im vorliegenden Fall sind jedoch die Vorteile von Interesse. Das Basissystem kann aus dem Internet bezogen werden und ist direkt einsatzbereit. Für erste Versuche sind kaum Vorarbeiten notwendig. Die Paketverwaltung kann auf das Debian-eigene *Advanced Package Tool (apt)* und die zugehörigen Paketquellen zurückgreifen. Auf diese Weise steht ein großer Softwareumfang bereit.

Das Paketverwaltungssystem von Debian erlaubt eine sehr weitgehende Beeinflussung der zu installierenden Pakete und Paketversionen. Durch den Einsatz eigener Paketserver kann auch Software über das System bereitgestellt werden, die nicht in den offiziellen Verzeichnissen verfügbar ist. Die Mechanismen zur Übertragung und Installation der Pakete erlauben auch den Einsatz von apt in Umgebungen mit schwankender Netzwerkqualität. Allerdings wächst der Aufwand, je stärker das System auf spezielle Bedürfnisse zugeschnitten wird. Um zehn Systeme auf dem exakt gleichen Stand zu halten, ist der Aufwand bereits sehr hoch. Hinzu kommt, dass es für die Verwaltung von Konfigurationsdateien keinen Ansatz in apt gibt. Spezielle Software für die Verwaltung großer Rechnerpools, die auch die Verwaltung von Konfigurationsdateien unterstützt, ist für die Verwaltung von ca. zehn Systemen mit zu viel Aufwand verbunden. Daher werden die Messsysteme mit einem gemischten Ansatz aus Paketverwaltung und Versionsverwaltungssystem gepflegt.

Für die Aktualisierung der Raspbian-Distribution kommen apt und die offiziellen Server zum Einsatz. Das Risiko von Versionsüberschneidungen wird dabei in Kauf genommen. Eine Versionsüberschneidung entsteht, wenn das Paket auf den offiziellen Servern aktualisiert wird, während die Messsysteme ihre Paketlisten aktualisieren. Dann haben einige Systeme die ältere und einige die neuere Version. Eigene, abgeschlossene Pakete werden ebenfalls per apt gepflegt. Dazu wird ein eigener Server betrieben, der nur für die Messsysteme erreichbar ist. Für die Verwaltung von Konfigurationsdateien sowie speziellen Werkzeugen, die häufigen Änderungen unterliegen, wird das DVVS Git genutzt. Der verwendete Ablauf folgt dabei dem in Abschnitt 4.4.4 entwickelten Modell. Die Entwicklung findet auf einem eigens dafür bereitgestellten Messsystem statt und fließt zu Testzwecken in einen Staging-Bereich. Dieser wird dann auf Messsysteme verteilt, die zu dem Zeitpunkt an keinem aktiven Messlauf beteiligt sind. Erst nachdem auch diese Systeme sich als weiterhin stabil erwiesen haben, erfolgt eine Übernahme in den Master-Zweig sowie eine Verteilung an alle Messsysteme.

Die Pflege von neun (bzw. zehn) Messsystemen erfordert bereits einen signifikanten Aufwand, speziell wenn eine Gleichartigkeit der Systeme gegeben sein soll. Daher erfolgt die Verwaltung über ein zentral angelegtes System (siehe Abbildung 6.18 auf Seite 190). Alle Messsysteme melden wesentliche Systemeigenschaften regelmäßig an eine zentrale Stelle[18]. Somit ist der Zustand aller Stationen mit nur geringer Verzögerung bekannt. Ein ähnliches System wird für die Verwaltung von Wartungsaufgaben genutzt. Die Messsysteme prüfen auf verschiedenen Servern, ob Wartungsaufträge vorliegen und führen diese selbstständig aus. Aufgaben behalten bis zur Erfüllung ihre Gültigkeit, so dass Unterbrechungen der Kommunikation zwischen Messsystem und Server keine Auswirkungen haben. Eine Rückmeldung zum Wartungsauftrag erfolgt über das Pandora Flexible Monitoring System. Die Sicherheit des Systems wird über Einmal-Token, Passwörter und die Beschränkung auf existierende Wartungsskripte gewährleistet.

Dieser dezentrale und eher asynchrone Ansatz zeigt die Vor- und Nachteile der verschiedenen Firmware-Management-Strategien sehr deutlich auf. Der gesamte Aktualisierungsprozess ist von einer Netzwerkverbindung abhängig. Bei der Nutzung der Paketverwaltung muss zuerst eine Liste der neuen und

[18] Hier wird auf das Pandora Flexible Monitoring System [Ler04] zurückgegriffen.

geänderten Pakete empfangen werden. Danach werden die entsprechenden Pakete ausgewählt. Jedes Paket wird dann einzeln geladen und aktualisiert. Im Falle eines Kommunikationsfehlers bricht die Aktualisierung ab. Alle bis zu diesem Moment erfolgten Änderungen sind unumkehrbar. Das kann zu inkonsistenten Zuständen führen, wenn durch die unterbrochene Aktualisierung Paketabhängigkeiten verletzt wurden. Bei der Nutzung eines VVSs werden erst alle Änderungen geladen. Danach erfolgt die Anwendung der Änderungen. Eine Unterbrechung des Ladevorgangs wird von aktuellen VVS toleriert. Es kommt zu keiner Zustandsveränderung. Bei beiden Verfahren muss ein erneuter Versuch explizit gestartet werden. Als Alternative zu diesen beiden block-basierten Methoden ist ein partielles Update möglich. Auch hier werden zuerst alle zur Aktualisierung benötigten Daten geladen und verifiziert. Im Gegensatz zum VVS kann bei einer individuellen Implementierung des Verfahrens auch auf die Nachteile einer schlechten Netzwerkverbindung Rücksicht genommen werden, beispielsweise durch blockweise Übertragung der Daten. Nachteil ist der deutlich höhere Aufwand bei einer Eigenentwicklung.

Die Entwicklung der Messsysteme zeigt auch Probleme auf, die in Verbindung mit der preiswerten Hardware und den daraus resultierenden Unterschieden zwischen den scheinbar gleichen Geräten stehen.

Zur Dokumentation des Raspberry Pi gehört auch eine Liste kompatibler SD-Karten. Und selbst mit Karten dieser Liste gibt es gelegentlich Probleme. Dabei wird das Dateisystem der Karte irreparabel beschädigt, so dass das Linux-System abstürzt und eine manuelle Neuinstallation erforderlich ist. Neuere Softwareversionen mildern das Problem weiter ab, doch es ist noch immer anzutreffen.

Die verwendeten LTE-USB-Sticks erfordern gelegentlich ein Reset, dass nur durch Trennung der Stromversorgung erreicht werden kann. Dazu verfügen die aufgebauten Messsysteme über einen vom Linux-System steuerbaren Schalter. Allerdings kommt es beim Verbinden des Sticks zu Systemabstürzen. Problemberichte zu USB-Geräten in Verbindung mit dem Raspberry Pi finden sich auch immer wieder im Internet. Ein Teil dieser Probleme lässt sich auf die harte Begrenzung auf 100 mA je Port zurückführen. Um das Problem zu umgehen, kommt im Messsystem ein USB-Hub mit einer maximalen Leistungsabgabe von 12,5 W zum Einsatz. Trotzdem kommt es immer wieder zu

Systemabstürzen, die sich eindeutig auf das Verbinden von USB-Geräten zurückführen lassen. Wie bei den SD-Karten verbessern neuere Softwareversionen das Problem, doch unter den zehn Messsystemen gibt es drei, die von diesen Instabilitäten besonders stark betroffen sind.

6.6 Zusammenfassung

Das Kapitel gibt einen Einblick in die praktische Umsetzung der in den vorherigen Kapiteln entwickelten Methoden und Ansätze. Es erläutert zuerst die vom Autor genutzte Entwicklungsumgebung sowie die verwendeten Werkzeuge. Anschließend werden vier konkrete Hardwareplattformen vorgestellt. Jede dieser Plattformen hat einige Besonderheiten, die im Text herausgearbeitet und näher betrachtet werden.

Umgebungen mit TFTP und *Network File System (NFS)* sind die empfohlene Vorgehensweise bei der Arbeit mit eingebetteten Systemen ([Hal06, Kapitel 9 und 12], [Yag+08]). In Erweiterung zu den in der Literatur beschriebenen Umgebungen berücksichtigt der hier vorgestellte Ansatz das verteilte Arbeiten, das Arbeiten im Team und die parallele Nutzung mehrerer Systeme. Bei der erforderlichen Hardware und den Diensten des Servers bedarf es dabei kaum Anpassungen. Lediglich die Filter zur korrekten Separierung der Netzwerke sind zu beachten. Bei Bedarf müssen weitere Anpassungen für den Zugriff der Entwickler auf den Server und die eingebetteten Systeme erfolgen. Mehr Aufwand verursachen die Werkzeuge, Buildsysteme und Software-Entwicklungsumgebungen. Mitglieder eines Teams sollten hier jeweils die gleiche Umgebung vorfinden. Dazu müssen, wie im Text beschrieben, Anpassungen an den jeweiligen Werkzeugen und Arbeitsabläufen vorgenommen werden.

Die vorgestellten Hardwaresysteme decken eine große Bandbreite der Architekturen und Ansätze aus den Kapiteln 3 und 4 ab. Ein Wechsel zwischen den Systemen gelingt mit Hilfe des Spezifikationssystems aus Kapitel 4 ohne Probleme. Ebenso erleichtert das System das Erstellen und Anpassen neuer Linux-Systeme.

Das Xilinx System „embedded FPGA“ entspricht einer der zwei wesentlichen Architekturen der heterogenen rekonfigurierbaren Systeme. Das Erstellen von

Linux-Systemen für die Xilinx FPGAs hat sich durch das intensive Engagement von Xilinx beim Linux-Kernel und bei den Device Trees deutlich vereinfacht. Unterstützung für die Konfiguration findet sich jedoch keine. Hier sind Entwickler weiterhin auf sich gestellt. Effiziente Lösungen erfordern daher ein vertieftes Verständnis für die Vor- und Nachteile der existierenden Konfigurationsarchitekturen. Die PSoCs sind aufgrund der mit ihnen realisierbaren, adaptiven Systeme von speziellem Interesse. Denn mit jeder Änderung wird ein neuer Entwurfszyklus notwendig. Dies betrifft nicht nur die konfigurierbare Logik, sondern auch das Betriebssystem. Es muss Treiber und Software bereitstellen sowie die Hardware korrekt erkennen und initialisieren.

Das Prototyping-System „AVR32“ entspricht der zweiten wesentlichen Architektur. Es verbindet das FPGA über eine geeignete Schnittstelle mit einem SoC. Die Konfiguration erfolgt parallel dazu ebenfalls über verfügbare Anschlüsse. Die Wahl der Schnittstellen erfolgt dabei in Abhängigkeit der Anforderungen an Geschwindigkeit und Flexibilität, aber auch in Abhängigkeit der Erfahrungen des Entwerfers. Unabhängig von der tatsächlichen Umsetzung kann das SoC während der Konfiguration verfügbar bleiben. Qualität und Geschwindigkeit der (Re-) Konfiguration hängen direkt von der sauberen Umsetzung aller Entwurfsschritte ab. Dazu gehören die Partitionierung in Hard- bzw. Software, die Wahl der Schnittstellen und die Umsetzung der Treiber und Programme. Schlüssel ist wiederum das detaillierte Verständnis der Architektur.

Am Rande der Arbeiten trat das beschriebene Problem mit dem SSC auf (vgl. Abschnitt 6.3.4), der sich nicht wie erwartet programmieren lässt. Für prototypische Umsetzungen stellt dies kein Problem dar. Bei zeitkritischen kommerziellen Anwendungen hingegen kann ein solcher Fehler zu beträchtlichen Problemen führen.

Das PowerPC-System „CarBox“ repräsentiert die gleiche Architektur wie das Prototyping-System „AVR32“, bietet aber mit dem gewählten Ansatz eine deutlich höhere Dynamik. Die breitbandige Busverbindung erlaubt eine schnelle Kommunikation und damit auch eine schnelle Konfiguration. Aufgrund der Architektur ist aber immer ein PFP notwendig. Die Probleme bei der Reinitialisierung des PCI-Busses zeigen wiederum, wie detailliert das Systemverständnis bereits vor dem Entwurf sein muss. Die Nutzung der dynamisch-partiellen Rekonfiguration stellt hier eine elegante Lösung dar. Im

PFP befindet sich ein sicher funktionierendes Design, das bei Bedarf verändert werden kann. Die eigentliche Funktionalität erhält das FPGA dann per dpR.

Von ursprünglich sechs DKIs sind während der Arbeiten vier durch ein Defekt des SoM ausgefallen. Die Fehlerursache konnte nicht genau geklärt werden. Das Fehlerbild lässt auf defekte RAM- oder Flash-ICs schließen. Da die Fehler nur nach längerem Betrieb auftraten, würde der Ausfall auch erst beim Kunden bemerkt werden. Eine Situation, die bei Serienprodukten zu ernsten Komplikationen führt.

Der Raspberry Pi, ebenso wie die vielen anderen, preiswerten Einplatinencomputer, weckt den Traum der preiswerten Kleinserie, die sich direkt aus dem ebenso preiswerten Prototyp ableiten lässt. Doch die Erfahrungen des Autors mit zehn identischen und drei ähnlichen Systemen zeigen, dass die Serienstreuung beim Raspberry Pi eine sehr genaue Bewertung jedes einzelnen Systems erfordert. Im konkreten Fall wirken sich die spürbaren Abweichungen zum einen auf die Haltbarkeit der SD-Karte und zum anderen auf die Systemstabilität bei der Nutzung von USB-Geräten aus.

Doch die Entwicklung, die rund um das große Interesse an dieser Plattform stattgefunden hat, ist durchaus beachtenswert. Mehrere Linux-Distributionen stehen nun auch für die ARMv6-Architektur zur Verfügung. Prototypen eingebetteter Systeme können damit auf einfache Weise und ohne besondere Vorkenntnisse entstehen. Erst bei der Umsetzung spezieller Projekte mit besonderen Anforderungen ist der Einsatz eines Buildsystems eine beachtenswerte Option. Ein ebenso lohnenswerter Zwischenschritt ist der Aufbau einer angepassten Linux-Distribution. Diese wird Bottom-Up beginnend beim minimalen Basissystem erstellt. Das Debian-Projekt bietet dazu mit *debootstrap* [Ker13] das passende Werkzeug.

Die vorgestellten Beispiele spiegeln nur einen Bruchteil der Bandbreite wieder, in der Linux-Systeme zum Einsatz kommen. Verbindendes Element aller Beispiele ist das methodische Vorgehen beim Systementwurf, dass sich an den in Abschnitt 2.7 vorgestellten Entwurfsabläufen orientiert, bzw. zu deren Entstehung und Validierung beigetragen hat. Heterogene rekonfigurierbare Systeme profitieren von der Flexibilität, die Linux bietet. Die vom Linux-Kernel gebotenen Konzepte für den Umgang mit wechselnder Hardware wiederum kommen der Dynamik von hrS entgegen. Für die effiziente Umsetzung gegebener

Problemstellungen ist letztlich neben einem vertieften Systemverständnis auch eine integrative und flexible Entwurfsumgebung von Bedeutung.

7 Konzepte zur plattformunabhängigen Erfassung von Sensordaten

Eine häufige Anwendung für eingebettete Systeme ist die Erfassung von Umweltinformationen. Und bereits in Abschnitt 1.1 wird klar, dass die Vernetzung von Systemen allgegenwärtig ist und noch immer voranschreitet. Beides betrifft auch die Systeme aus Kapitel 6. Dieses Kapitel beschreibt Lösungen, die im Rahmen der Umsetzung einer „generalisierten Plattform zur Sensordatenerfassung“ (vgl. Anhang F auf Seite 241) und in Verbindung mit den vorgestellten Beispielanwendungen vom Autor untersucht und realisiert wurden.

7.1 Sensordatenerfassung auf einer zentralen Plattform

Eine der wesentlichen Aufgabe für viele eingebettete Systeme ist die Erhebung und Speicherung von Daten. Daher wurden verschiedene Konzepte für die nachhaltige Datenspeicherung evaluiert, wobei neben den in Kapitel 6 vorgestellten Systemen und Anwendungsfällen auch weitere berücksichtigt wurden. Das letztlich umgesetzte und in Abbildung 7.1 auf der nächsten Seite dargestellte Konzept basiert auf einer Datenbank sowie einer modularen Infrastruktur, welche die Anbindung nahezu beliebiger Systeme zulässt.

Zentrale Komponente ist ein *Datenbankmanagementsystem (DBMS)*, das auf einem dafür geeigneten, zentralen Server zur Verfügung steht. Um die Unabhängigkeit vom DBMS zu erhalten, werden alle Zugriffe über ein eigenes API

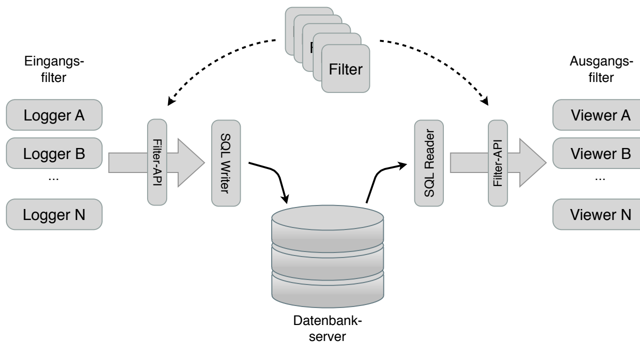


Abbildung 7.1: Konzept für die zentrale Datenhaltung

gekapselt. Das realisierte System nutzt hierbei die SQL, die von der Mehrzahl gängiger Datenbanken unterstützt wird.

Die Eingangsfilter realisieren die Umsetzung vom Datenformat eines spezifischen Loggers auf das Datenbankschema. Dieses Vorgehen eröffnet eine Vielzahl von Möglichkeiten. Logger können sich direkt mit ihrem Eingangsfilter verbinden und ihre Daten z. B. über eine Netzwerkverbindung permanent an die Datenbank übertragen, wobei der Mehraufwand durch den Eingangsfilter und das Datenbank-API unbedeutend ist. Daten von Systemen, die mangels Netzwerkverbindung ihre Messwerte lokal zwischenspeichern, werden nach Abschluss der Messungen in die Datenbank übernommen. Hier wandelt der Eingangsfilter die aufgezeichneten Messwerte in das Datenbankformat. Somit kann zur Aufzeichnung im Logger ein geeignetes, effizientes Verfahren ohne Berücksichtigung der späteren Sicherung in der Datenbank gewählt werden. Durch die Filter besteht weiterhin die Möglichkeit, alle Daten bereits vor dem Speichern in der Datenbank zu filtern, zu aggregieren oder in anderer Form zu verarbeiten. Hierbei ermöglicht ein API den Zugriff auf vorgefertigte Filter.

Für das Auslesen der Daten aus der Datenbank kommt ein vergleichbares Konzept zur Anwendung. Die eigentlichen Datenbankzugriffe werden abstrahiert und die Daten an die Ausgabefilter übergeben. Über das Filter-API besteht wiederum die Chance, die Daten zu filtern, zu aggregieren oder in anderer Form zu verarbeiten. Das System ist in dieser Form auch lokal auf einem Logger umsetzbar, sodass hinreichend leistungsfähige eingebettete Systeme ihre

Daten lokal in einer Datenbank speichern und somit auch auf die Vorteile der datenbankgestützten Ein- und Ausgabe zurückzugreifen können.

7.1.1 Datenmodelle zur Sensordatenerfassung

Die Nutzung eines DBMS für die Archivierung von Sensordaten wird durch zwei Kernpunkte geprägt. Zum einen muss die Speicherung der Daten hinreichend *schnell und zuverlässig* erfolgen, sodass ein Datenverlust ausgeschlossenen werden kann. Zum anderen müssen die Messreihen auch nach längerer Zeit *reproduzierbar* sein. Vor der Definition der für ein DBMS benötigten Datenmodelle erfolgt eine Beschreibung und Benennung der zur Erfassung von Messwerten notwendigen Systeme bzw. Komponenten.

Messwert Der Messwert ist ein zu einem definierten Zeitpunkt erhobenes Datum.

Messwertkanal Ein Messwertkanal erhebt Messwerte. Er beschreibt die Bedeutung der durch ihn erhobenen Messwerte, beispielsweise durch eine Einheit.

Messgerät Ein Messgerät vereint mehrere Messkanäle. Typische Messgeräte sind beispielsweise eine „GPS-Maus“ oder das Software-Kommando `traceroute`.

Messsystem Ein Messsystem stellt ein physisch vorhandenes System dar, dass ein oder mehr Messgeräte vereint. Oft findet sich dafür auch der Begriff „Logger“.

Messlauf Ein Messlauf umfasst alle zu einem Vorgang gehörenden Messaufbauten und beschreibt die Zeitspanne seiner Gültigkeit.

Messaufbau Der Messaufbau beschreibt die Verbindung von zu untersuchendem Objekt (engl. *Device Under Test (DUT)*), Messsystem und Messlauf.

Messvorgang Ein Messvorgang stellt einen Bezug zwischen verschiedenen Messwerten her. Der Bezug kann zeitliche, räumliche oder thematische Aspekte umfassen.

Zusätzlich gilt die Festlegung, dass ein Messaufbau immer auf ein DUT und ein Messsystem beschränkt ist. Wird ein DUT durch mehrere Messsysteme beobachtet, spiegelt sich dies in mehreren Messaufbauten wieder. Gleiches gilt, wenn ein Messsystem zeitgleich mehrere DUTs beobachtet.

Zur Vervollständigung und Beurteilung des Datenmodells erfolgte auch eine Betrachtung von geeigneten Strategien für die Datenerfassung. Drei gängige Verfahren konnten hierbei identifiziert und bei der Umsetzung der Datenmodelle berücksichtigt werden:

Fauler Ansatz (Lazy Approach) Bei diesem Ansatz entstehen alle notwendigen Einträge in der Datenbank nach Bedarf. Demnach muss der Messaufbau im Vorfeld nicht bekannt sein. Der zuständige Eingangsfilter analysiert ankommende Daten und ordnet diese existierenden Messsystemen, Messgeräten und Messwertkanälen zu oder legt neue an.

Vorheriges Setup (Initial Setup) Dieser Ansatz ist das Gegenteil zum vorgenannten. Hier werden Messwerte nur in die Datenbank übernommen, wenn alle beteiligten Komponenten im Vorfeld bekannt und entsprechend in der Datenbank hinterlegt sind.

Gemischter Ansatz (Mixed Approach) Bei diesem Ansatz werden einige Elemente bereits im Vorfeld festgelegt, während andere ihre Dynamik behalten.

Für die Umsetzung des in der Datenbank verwendeten Datenmodells wurden zwei Ansätze implementiert und auf ihre Eignung hin untersucht. Wesentlicher Unterschied beider Modelle ist das Konzept zur Abbildung der Systemumgebung auf die Datenbank. Die Umsetzung erfolgte auf Basis relationaler Datenbanken. Diese eignen sich im Besonderen für die Speicherung strukturierter Daten, die miteinander in Beziehung stehen. Die Ablage der Daten erfolgt in Form von Tabellen. Für die Darstellung der Beziehungen zwischen den Daten nutzt man im Allgemeinen ein Entity-Relationship-Modell (ERM) [Baloo].

Hierarchisches Datenmodell

Abbildung 7.2 auf der nächsten Seite zeigt das erste im Rahmen dieser Arbeit entstandene ERM zur Speicherung beliebiger, zeitbezogener Messdaten in der

ersten Normalform. Es verfolgt den Ansatz einer objektorientierten und daher hierarchischen Darstellung des Messaufbaus. Das Modell ist dabei trotzdem flexibel gestaltet.

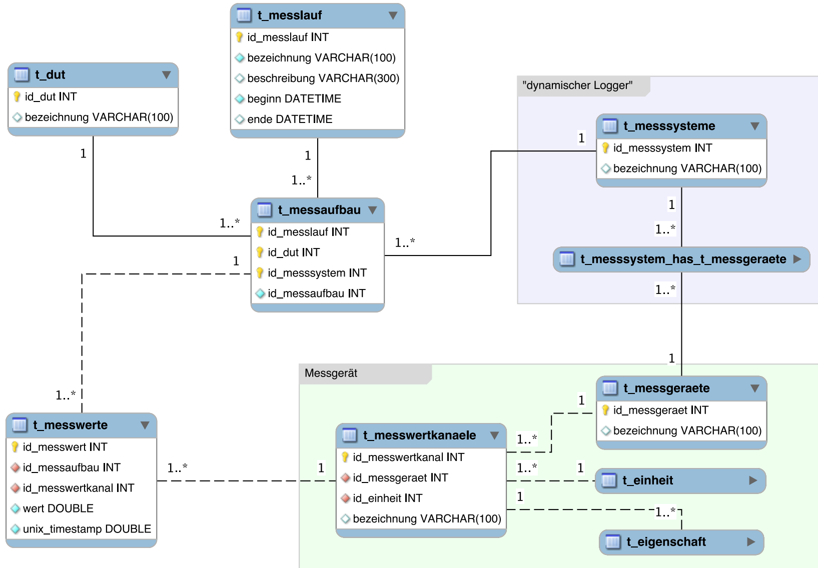


Abbildung 7.2: Entity-Relationship-Modell der Datenbank

Das zentrale Element dieses Modells ist der Messaufbau. Hier wird das zu untersuchende Objekt (DUT) mit Hilfe eines Messsystems (z. B. mit einem Logger) über einen bestimmten Zeitraum beobachtet. Diese Beziehung gibt die Tabelle `t_messaufbau` wieder, die die Tabellen `t_dut`, `t_messlauf` und `t_messsysteme` in Beziehung setzt. Per Definition (vgl. Abschnitt 7.1.1) ist ein Messaufbau immer auf ein DUT und ein Messsystem beschränkt. Im Rahmen eines Messlaufes können aber beliebig viele Messaufbauten genutzt werden.

Bei den Beschreibungsmöglichkeiten für die Messsysteme wird berücksichtigt, dass ein physisch vorhandenes System je nach Ausprägung verschiedene Messwerte liefern kann. Diese Eigenschaft wird im Rahmen des Modells durch die Verbindung von Messsystem und Messwertkanal über die Messgeräte wiedergegeben.

Messgeräte[1] (`t_messgeraete`) erzeugen häufig mehrere Messwerte, was wiederum im Modell umgesetzt wurde. Für jeden durch ein Messgerät erfassten Wert bildet sich ein Kanal (`t_messwertkanaele`) aus, der immer die gleiche Art Messwert liefert. Dieser Messwertkanal ist durch den ihn erzeugenden Kanal gekennzeichnet und lässt sich mit Hilfe seiner Einheit (`t_einheit`) weiter beschreiben. Gelegentlich sind weitere Informationen wie (Software-) Kommandos, Korrekturfaktoren oder Wertebereiche von Interesse, die in einer zusätzlichen Tabelle (`t_eigenschaft`) festgehalten werden. Diese Art der Betrachtung erlaubt auch die Einführung *virtueller Messgeräte*, die beispielsweise zur Speicherung vorverarbeiteter Messwerte genutzt werden können. Messwertkanäle verweisen auf die von ihnen erhobenen Werte, die zusammen mit der Messzeit gespeichert werden. Eine zusätzliche Relation bindet die Messwerte an den Messaufbau, in dessen Zusammenhang sie erhoben werden. Diese Verbindung ermöglicht eine deutliche Vereinfachung beim Zugriff auf Messwerte. Weiterhin ist sie für die Konsistenz und Reproduzierbarkeit des Modells notwendig.

Für das Darstellen der Messungen existieren verschiedene Startpunkte, die im Sinne der Reproduzierbarkeit immer zum gleichen Ergebnis führen. Einstiegspunkte könnten

- ein bestimmter Messlauf,
- ein bestimmtes Messgerät oder Messsystem,
- ein bestimmtes DUT oder
- ein bestimmter Messaufbau

sein. Ziel ist es, unabhängig vom Startpunkt, den gesamten Messlauf mit allen zugehörigen Elementen zu finden.

Die dafür notwendigen Schritte können direkt im ERM nachvollzogen werden. Der wesentliche Knotenpunkt ist `t_messaufbau`. Hier erfolgt die eindeutige Verbindung von DUT und Messlauf mit einem Messsystem und damit auch über die entsprechenden Messgeräte mit den Messwertkanälen. Über die Kenntnis der Kanäle können alle zugehörigen Randbedingungen bestimmt werden. Messaufbau und Messwertkanäle erlauben die Identifikation der zugehörigen Messwerte. Der Messaufbau als zentraler Punkt in diesem Modell

[1] Der Begriff „Sensor" wird aufgrund seiner Mehrdeutigkeit bewusst vermieden.

dient auch zur wesentlichen Vereinfachung beim Zugriff auf die mit ihm erzeugten Messwerte. Da per Festlegung ein Messaufbau immer auf ein DUT, ein Messsystem und einen Messlauf festgelegt ist, verringert sich bei typischen Anfragen[2] die benötigte Menge an SQL-Abfragen deutlich.

Insgesamt stellen jedoch die Menge an notwendigen Abfragen und der hierarchische Aufbau die wesentlichen Nachteile des Modells dar. Für das Eintragen von Messwerten sind die Primärschlüssel von `t_messaufbau` und `t_messwertkanaele` notwendig, die, je nach Eintrittspunkt in das Modell, über typischerweise sechs Abfragen zu bestimmen sind. Dieser Aufwand stört besonders bei Ansätzen, die ein häufiges Schreiben weniger Messwerte erfordern, das Speichern der notwendigen Primärschlüssel aber nicht erlauben. Beim Anlegen neuer Messläufe ist außerdem eine klare Abbildung der erwarteten Messwerte auf das Modell notwendig, was die Nutzung des faulen Ansatzes erschwert. Somit empfiehlt sich diese Variante eher bei Messsystemen, die ihre Daten speichern und bei denen erst im Anschluss an den Messlauf eine Übertragung ins DBMS erfolgt. Für das zeitnahe Erfassen von Messwerten, besonders lokal auf Systemen mit geringer Rechenleistung, erscheint die Lösung hingegen nicht optimal.

Das System bietet aber durch die vorgegebene Hierarchie auch einige Vorteile. Das Modell unterstützt die korrekte Restauration von Messläufen, da alle notwendigen Informationen bereits beim Speichern gefordert werden. Die Nähe zum realen Aufbau und das objektorientierte Design erleichtern auch das Auslesen der Daten, da alle Beziehungen klar dargestellt sind. Und nicht zuletzt bietet das Modell mehrere Einstiegspunkte, die immer zu den Messwerten und dem mit ihnen verbundenen Messaufbau führen, was die Implementierung einer Nutzerschnittstelle zur Datenbank erleichtert.

Flaches Datenmodell

Als Alternative zum hierarchischen wurde ein sehr flaches Datenmodell implementiert, um dessen Vor- und Nachteile zu evaluieren. Abbildung 7.3 auf der nächsten Seite stellt das Modell dar.

[2] z. B. beim Bestimmen aller zu einem Messaufbau gehörenden Werte

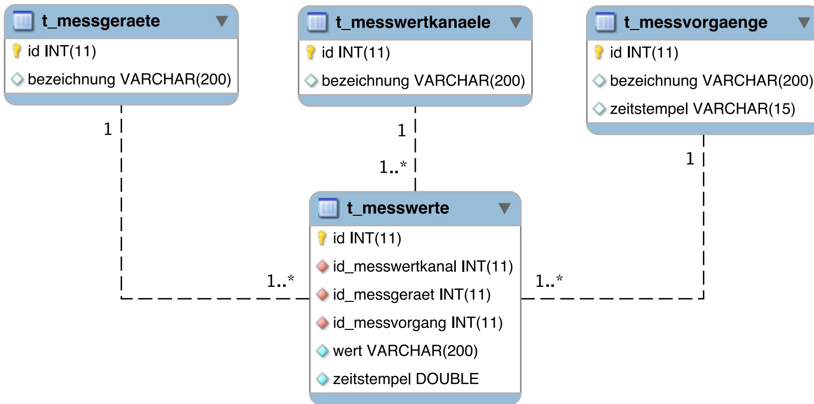


Abbildung 7.3: Flaches Datenmodell

Das Speichern von Messwerten und ergänzenden Informationen erfolgt ausschließlich in der Messwerttabelle (`t_messwerte`). Diese ähnelt der des hierarchischen Modells, die Messwerte sind jedoch direkt mit den erzeugenden Instanzen Messgerät und Messwertkanal verbunden. Das Sammeln von Geräten und Kanälen erfolgt in einfachen Tabellen, die höchstens das Hinterlegen einer Bezeichnung unterstützen. Um die Reproduzierbarkeit zu gewährleisten, ist eine weitere Tabelle notwendig (`t_messvorgänge`). Messvorgänge gruppieren zusammengehörige Messwerte, z. B. alle Werte einer gelesenen Datei oder alle Werte einer erfolgreichen Übertragung per Netzwerk. Alle weiteren Informationen werden aus Sicht des Modells bereits als Messwert betrachtet und in der entsprechenden Tabelle hinterlegt. Das betrifft im Speziellen auch alle Eigenschaften der Messsysteme, Messgeräte und Messwertkanäle sowie die Beschreibung des gesamten Messlaufs.

Der entscheidende Vorteil ist die Flexibilität des so entstehenden Systems. Lediglich die Elemente Messsystem und Messgerät sind geblieben, in ihrer Bedeutung aber freier als im hierarchischen Modell. Das Speichern weiterer Informationen geschieht quasi unabhängig vom Modell. Da alle Werte in einer Tabelle mit nur drei Schlüsseln gehalten werden, ist auch das Eintragen neuer Messwerte einfach und schnell. Damit ist das Modell sehr gut für das zeitnahe Erfassen von Messwerten geeignet. Durch seine freie Gestaltung und das Mit-

führen aller Eigenschaften in gesonderten Kanälen kommt das Modell auch dem faulen Ansatz entgegen.

Die starke Abstraktion des Modells (nahezu alles ist ein Kanal) erschwert jedoch den Bezug zum realen Messaufbau. Es wird keine feste Struktur vorgeschrieben. Damit fehlt auch die Unterstützung bei der Erfassung der Daten, die für eine vollständige Reproduktion notwendig sind. Diese Aufgabe liegt vollständig beim Entwickler des Eingangsfilters (vgl. Abschnitt 7.1). Auch das Auslesen der erfassten Informationen ist aufgrund der Abstraktion weniger intuitiv. Die gesamte Rekonstruktion basiert auf den vier Größen Messgerät, Messwertkanal, Messvorgang und Zeit. Die Restauration eines Messlaufes ist damit Aufgabe des Ausgabefilters[3]. Alternativ können Funktionen des DBMS genutzt werden[4], was technisch dem Konzept des *Fat Servers*[5] entspricht. Damit entsteht nach außen eine Schnittstelle, die sich der des hierarchischen Modells annähert. Dieses Vorgehen vereinfacht die Umsetzung eines Ausgangsfilters, belastet jedoch den Datenbankserver stärker und erfordert größere Sorgfalt beim Datenbankdesign.

7.1.2 Datenübertragung an einen Datenbankserver

In Abschnitt 7.1 wurde bereits darauf hingewiesen, dass Systeme mit Netzwerkanbindung ihre Daten direkt an den Datenbankserver ausliefern können. Allerdings sind im Besonderen drahtlose Netzwerkverbindungen nicht zwingend permanent verfügbar und auch der Datenbankserver ist mit einem Ausfallrisiko behaftet – speziell im semiprofessionellen Bereich. Daher sollte das Übertragungssystem auf diese Probleme vorbereitet sein und eventuelle Ausfälle ohne Datenverlust überbrücken. Demnach ist eine direkte Anbindung des Messsystems (bzw. der Software auf dem Messsystem) an das DBMS keine Option. Als robuste Lösung hat sich das in Abbildung 7.4 auf der nächsten Seite skizzierte System etabliert.

Die auf den Messsystemen arbeitende Messsoftware legt die erhobenen Daten im lokalen Dateisystem ab. Durch die Architektur können dabei mehre-

[3] Im Sinne des *Fat Client* Modells, bei dem die datenverarbeitenden Funktionen beim Client liegen.

[4] Bei MySQL beispielsweise Views, Prozeduren und Funktionen.

[5] Datenverarbeitende Funktionalität wird in den Datenbankserver ausgelagert.

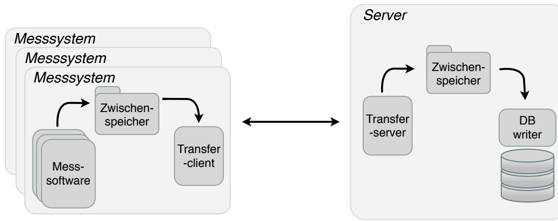


Abbildung 7.4: Konzept der Messdatenübertragung

re Messprozesse quasi parallel ihre Daten hinterlegen. Ein unabhängiger Prozess (im Bild als *Transferclient* bezeichnet) verbindet sich bei Bedarf mit dem Server und überträgt die Messdaten aus dem lokalen Zwischenspeicher zu einer dort laufenden Software (*Transferserver*). Nach erfolgreicher Übertragung können die Daten auf dem Messsystem gelöscht werden, um nicht unnötig Speicherplatz zu belegen. Der Transferserver speichert die empfangenen Messwerte ebenfalls im lokalen Dateisystem, wo sie dann durch das in Abschnitt 7.1 beschriebene System in die Datenbank geschrieben werden. Der lokale Zwischenspeicher verhindert Datenverlust beim Ausfall der Datenbank und erlaubt dem Transferserver ein zügiges Bedienen eingehender Verbindungen, sodass das System auch für den Umgang mit einer größeren Anzahl von Messsystemen geeignet ist. Die Nutzung unabhängiger Prozesse für das Messen, Übertragen und Ablegen der Daten vereinfacht zudem die Softwarearchitektur. Eine aufwändige Synchronisation der Prozesse ist nicht nötig. Ein Maß für die Leistungsfähigkeit des Systems ist die Verweildauer eines Datensatzes in den jeweiligen Zwischenspeichern. Für den Echtzeitbetrieb bzw. für die Einhaltung vorgegebener Zeitschranken ist das System nicht konzipiert und auch nicht geeignet. Ein Betrieb mit mehreren Transferservern ist hingegen möglich und wurde im Rahmen der Arbeit auch genutzt.

7.1.3 Implementierung des Systems

Der Anspruch des vorgestellten Konzepts ist nicht nur die schnelle, zuverlässige Speicherung von Messwerten in einer Datenbank, sondern auch die Sicherung des zur Erhebung der Daten verwendeten Aufbaus. Das ermöglicht die

Reproduktion der Messwerte zusammen mit dem Kontext, in dem sie entstanden sind – auch ohne vorherige Kenntnis des verwendeten Messaufbaus.

Dazu wird ein Datenbankmodell benötigt, dass unabhängig vom konkreten Anwendungsfall bleibt, um so die größtmögliche Flexibilität zu bieten. Dadurch können die Daten verschiedener bereits existierender Systeme erfasst werden, bei deren Design eine Speicherung in einer Datenbank nicht berücksichtigt wurde. Ebenso ist das Sichern von Messwerten möglich, die unter derzeit noch nicht bekannten Umständen entstehen. So ist beispielsweise die Dynamik von Messsystemen, die ihre Messwerte per Software erzeugen und dadurch häufigen Änderungen unterliegen, eine bisher kaum beachtete Problematik.

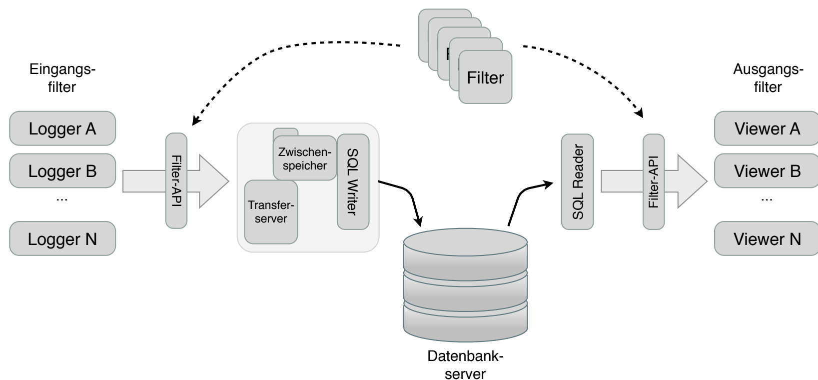


Abbildung 7.5: Schema des implementierten Systems

Die entwickelten Konzepte für die Datenerfassung mit Ein- und Ausgangsfilter (vgl. Abbildung 7.1) und die Datenübertragung (vgl. Abbildung 7.4) haben sich als robust und tragfähig erwiesen. Die Entkopplung der Datenerfassung und der Speicherung im DBMS über einen dateisystembasierten Zwischenspeicher bietet so viele Vorteile, dass inzwischen auch das Verarbeiten von Messwertdateien kleinerer Messsysteme über diesen Mechanismus erfolgt. Damit ergibt sich im derzeitigen Entwicklungsstand das in Abbildung 7.5 dargestellte System.

Als problematisch hat sich ein geeignetes Datenbankschema erwiesen, dass den Ansprüchen an Geschwindigkeit und Flexibilität beim Erfassen und Le-

sen der Messwerte ebenso gerecht wird, wie bei der Speicherung der zur Reproduktion notwendigen Kontextinformationen. Grundsätzlich erleichtert ein hierarchisches Modell die Abbildung des Kontextes. Dies gilt beim Erfassen der Daten ebenso wie beim Auslesen. Allerdings ist das Speichern der Daten in ein sehr flach gehaltenes Schema nach einer korrekten Modellierung deutlich einfacher. Ebenso kommt ein sehr flaches Modell der Flexibilität und Dynamik des Gesamtsystems und der abzubildenden Szenarien entgegen.

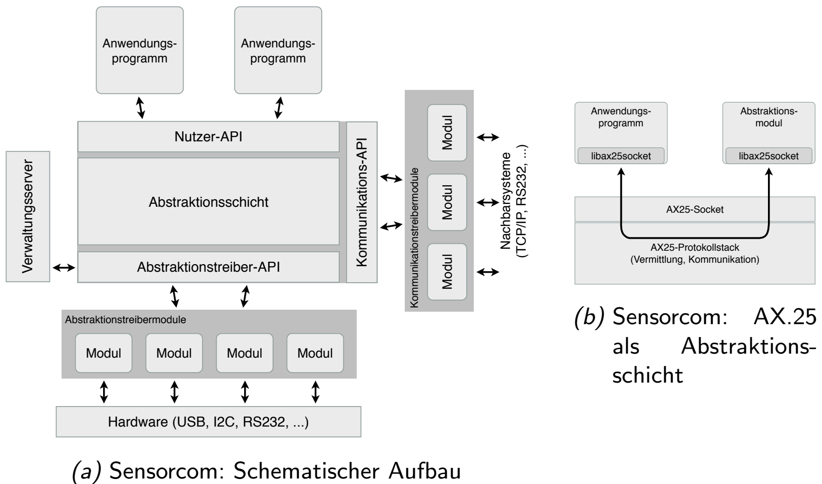
Mit Hilfe der SQL Reader und Writer (vgl. Abbildungen 7.1 und 7.5) sowie durch Nutzung der Funktionen eines modernen DBMS können die Schwierigkeiten bei der Abbildung des realen Messaufbaus in das flache Schema verringert werden. Dazu Transformieren die Systeme die flache Struktur der Datenbank auf hierarchische Modelle, die einer objektorientierten Betrachtung näher kommen. Die Ein- und Ausgangsfilter übernehmen dann die jeweilige Abbildung auf das Zieldatenformat. Somit kann die Komplexität des gesamten Vorgehens auf mehrere Ebenen verteilt werden.

7.2 Abstraktion von Kommunikationskanälen

Die Arbeit mit heterogenen und verteilten Sensor-Aktor-Systemen hat gezeigt, dass die im Linux gebotenen Abstraktionsmechanismen für den Zugriff auf Geräte (vgl. Abschnitt 4.1.2) in diesem Fall an ihre Grenzen stoßen. Der Quellcode für Kommunikation und Verwaltung ist in jeder Anwendung erneut zu implementieren, was schnell zu unnötigem Aufwand führt. Daher wurde nach einer Möglichkeit gesucht, die Kommunikation innerhalb eines verteilten Sensor-Aktor-Sensorsystems weitgehend unabhängig vom Übertragungskanal zu halten.

Zu diesem Zweck entstand ein Kommunikationssystem namens „Sensorcom“ [Fri07], das basierend auf dem AX.25-Protokoll eine Kommunikationsinfrastruktur aufbaut. Diese ist unabhängig vom darunter liegenden Übertragungssystem. Sensorcom bietet sowohl die Abstraktion lokaler Verbindungen als auch solcher über Netzwerke. Die modular aufgebaute Software ist dabei so ausgelegt, dass sie sowohl auf einfachen Sensorsystemen mit geringen Ressourcen als auch auf besser ausgestatteten Netzwerkknoten eingesetzt werden kann.

Abbildung 7.6a zeigt den modularen Aufbau. Rund um den Kern, bestehend aus der Abstraktionsschicht und den Schnittstellen zu den einzelnen Modulen, gliedern sich einfache Kommunikations- und Abstraktionstreiber. Neue Hardware- und Kommunikationsschnittstellen können mit wenig Aufwand zugefügt werden. Die API, über die Anwendungsprogramme sowie Kommunikations- und Abstraktionstreiber mit Sensorcom interagieren, orientiert sich an den bekannten Berkeley Sockets [04]. Außerdem ist das System so angelegt, dass redundante Verbindungen zu Sensoren, Aktoren oder weiteren Sensorcom-Instanzen erkannt und verwaltet werden, ohne dass dafür zusätzlicher Implementierungsaufwand notwendig ist. Der dafür benötigte Verwaltungsserver sorgt dafür, dass verteilte Sensorsysteme möglich sind. Eine Anwendung erfragt dazu lediglich einen Sensortyp oder ein von ihr gewünschtes Datum und der Server vermittelt transparent zum korrekten Sensor bzw. Wert.



(a) Sensorcom: Schematischer Aufbau

(b) Sensorcom: AX.25 als Abstraktionsschicht

Abbildung 7.6: Sensorcom Konzept

Den Kern des Abstraktionssystems bilden der AX.25-Protokollstack und ein I/O-Verwaltungssystem. Die Integration der AX.25-Protokolls stellt Abbildung 7.6b dar. Das I/O-Verwaltungssystem ist für das Management der Verbindungen zu den Anwendungsprogrammen, zum Verwaltungsserver sowie zu

den Abstraktionstreibern verantwortlich. Der Datenaustausch zwischen dem Abstraktionssystem und den externen Komponenten[6] erfolgt über Berkeley Sockets, wobei es einen Daten- und einen Kommandokanal gibt. Zur Kapselung der spezifischen Funktionalität dient eine Bibliothek (libax25socket), die in allen Komponenten zum Einsatz kommt. Sie vereinfacht und vereinheitlicht die Programmierung der Abstraktionsmodule und Anwendungsprogramme. Weiterhin ermöglicht sie die Anpassung des Abstraktionssystems, ohne dass eine Veränderung der Abstraktionstreiber und Anwendungsprogramme notwendig ist. Dadurch ist es beispielsweise möglich, als Alternative zum AX.25-Protokollstack des Linux-Kernels eine eigene Umsetzung im User Space zu verwenden, was grundsätzlich den Einsatz des Abstraktionssystems auch ohne AX.25-Unterstützung durch den Kernel erlaubt.

7.3 Zusammenfassung

Das Kapitel befasst sich mit zwei Lösungen, die bei der effizienten Umsetzung von heterogenen Sensor-Aktor-Systemen zum Tragen kommen. Die zentrale Speicherung erhobener Daten, unabhängig vom zugrundeliegenden Messaufbau, und der Zugriff auf Knoten eines Netzwerks, unabhängig vom zugrundeliegenden Datenkanal. Beide Modelle bauen auf die Abstraktion des realen Systems zur Vereinfachung der Informationsverarbeitung.

Die Speicherung von Messwerten in Datenbanken wird bei vielen Anwendungen genutzt. Die dabei verwendeten Datenbankmodelle orientieren sich meist am konkreten Anwendungsfall. Das betrifft sowohl das Speichern der Daten in der Datenbank als auch das Auslesen für die spätere Verarbeitung. Da der konkrete Kontext der Erhebung der Daten bekannt ist, muss das Modell keine besonderen Vorkehrungen für eine spätere Reproduktion des Aufbaus aus den Werten der Datenbank treffen. Dies trifft in vielen Fällen auch für eingebettete Systeme zu, bei denen sich die Hard- und Software sowie der Kontext der Messwerterfassung oft über lange Zeiträume nicht verändern. Viele eingebettete Systeme speichern ihre Daten zudem in Form von Dateien. Eine dauerhafte Sicherung in einer Datenbank ist nicht vorgesehen.

[6] Anwendungsprogramme, Abstraktionstreiber, Verwaltungsserver

Das in Abschnitt 7.1 vorgestellte Modell erfüllt die gestellten Anforderungen. Es erlaubt die flexible Speicherung von Messwerten und die Reproduktion des Kontextes ihrer Erfassung. Das System ist trotz seiner Flexibilität überschaubar und erlaubt eine bedarfsgerechte Erweiterung. Durch die mehrstufige Abstraktion des DBMS konnten sowohl der hierarchische als auch der flache Ansatz exemplarisch implementiert und mit verschiedenen Szenarien evaluiert werden. Die bereits in den jeweiligen Abschnitten diskutierten Vor- und Nachteile der Systeme konnten so bestätigt werden. Eine Empfehlung für eines der Datenbankmodelle kann somit nicht gegeben werden.

Sensorcom setzt ein modulares Konzept für die Abstraktion von Kommunikationsverbindungen in verteilten Sensornetzen prototypisch um. Für die Verbindung zwischen den einzelnen Kommunikationskonten innerhalb des Abstraktionssystems kommt das AX.25-Protokoll zum Einsatz, ein Protokoll aus dem Amateurfunkbereich. Der gewählte Ansatz erlaubt die transparente Verbindung von Anwendungsprogrammen mit Sensoren und Aktoren in einem verteilten Sensornetz, auch über Systemgrenzen hinweg. Das System ist durch die Verwendung gängiger Schnittstellen und Programmiertechniken leicht erweiterbar und grundsätzlich auch für die Verwendung außerhalb von Linux-Systemen geeignet.

Die vorliegende Implementierung erlaubt die Evaluierung von Konzepten für den Aufbau verteilter Sensor-Aktor-Systeme. Gleichzeitig diente sie der Erforschung verschiedener Techniken für die IPC, die Protokollabstraktion und die Realisierung modularer Softwaresysteme. Ein wesentliches Ziel war die Orientierung an eingebetteten Systemen mit geringen Ressourcen und begrenzter Leistung. Des Weiteren konnten während der Implementierung des Systems Erkenntnisse zu Funktionen und Abhängigkeiten innerhalb des Linux-Kernels gewonnen werden. Die erarbeiteten Konzepte sind weitgehend unabhängig von der spezifischen Implementierung und erlauben die Beurteilung ähnlicher Systeme und Konzepte.

8 Zusammenfassung und Ausblick

Die rasante Geschwindigkeit, mit der sich der Markt für eingebettete Systeme entwickelt, verringert die Zeit, die für die Entwicklung eines Produktes zur Verfügung steht. Dies erfordert neue Methoden und Werkzeuge, die die Arbeitsabläufe des Systementwurfs beschleunigen. Gleichzeitig erhöht sich die Komplexität der zu entwickelnden Systeme in ebenso rasantem Tempo. Diese Komplexität ist derzeit nicht allein mit Hilfe von Werkzeugen zu beherrschen. Hier ist der erfahrene Entwickler gefordert, der durch sein Know-how und durch den effektiven Einsatz der ihm zur Verfügung stehenden Werkzeuge zum Erfolg der Produktentwicklung beiträgt.

Software stellt einen der Innovationsträger bei eingebetteten Systemen dar. Sie erlaubt die herstellerspezifische Anpassung von Geräten mit identischer Hardware. Das ist besonders in Märkten mit hohem Preisdruck wie der Heimelektronik oder der Unterhaltungsindustrie anzutreffen. Sie erlaubt aber auch die Bereinigung von Fehlern, nachdem das Produkt bereits an den Kunden ausgeliefert wurde. Das kann ein Qualitätsmerkmal oder gar eine notwendige Funktionalität sein. Nach Meinung des Autors verleitet diese Funktion aber auch zu vorschneller Auslieferung von Produkten – bewusst oder unbewusst. Dieser Eindruck ergibt sich aus der fortwährenden Beobachtung des Smartphonemarktes durch den Autor. Hier folgten kurz nach den letzten größeren Aktualisierungen der Betriebssysteme der beiden Marktführer jeweils mehrere kleine Bugfix-Releases. Dabei wurden teilweise Fehler behoben, die die Frage nach der Qualität und Ausführlichkeit der vorher erfolgten Tests als durchaus gerechtfertigt erscheinen lassen.

Die Zielsetzung der Arbeit ist die Vermittlung eines vertieften Verständnisses für die Systemkonzepte von heterogenen rekonfigurierbaren Systemen und eingebetteten Systemen mit Linux-Betriebssystem. Dabei liegt der Fokus nicht

ausschließlich auf dem wissenschaftlichen Beitrag, sondern auch auf der praktischen Relevanz und Umsetzbarkeit der Ergebnisse. Beides trägt zur Verbesserung des Entwurfsflusses und damit zur beschleunigten Produktentwicklung bei. Ein wesentlicher Teil der Arbeit befasst sich dabei mit dem Betriebssystem Linux.

8.1 Zusammenfassung

Die vorliegende Dissertationsschrift betrachtet verschiedene Facetten des Systementwurfs eingebetteter heterogener rekonfigurierbarer Systeme mit Linux-Betriebssystem am Beispiel einer modularen Plattform zur Erfassung und Verarbeitung von Sensordaten. Der wesentliche Beitrag der Arbeit liegt in der Darstellung und Diskussion von Konzepten und Architekturen für heterogene rekonfigurierbare Systeme und Linux-Systeme. Es werden Sachverhalte diskutiert, die nach Meinung des Autors einen wesentlichen Beitrag für die effiziente und erfolgreiche Umsetzung von Projekten im Bereich der eingebetteten, rekonfigurierbaren Systeme leisten. Eine Darstellung der dafür notwendigen Entwurfsmethodik findet sich bisher nicht in der Literatur. Außerdem erfolgt die Abbildung der Erkenntnisse auf konkrete Beispiele bzw. Implementierungen.

Das Grundlagenkapitel (Kapitel 2) führt eine für die vorliegende Arbeit einheitliche Begriffswelt ein. Weiterhin erläutert es verschiedene Modelle des allgemeinen Systementwurfs, die die Basis für jedes Entwicklungsprojekt darstellen sollten. Es folgen grundsätzliche Betrachtungen zu den für diese Arbeit relevanten Forschungsfeldern — eingebettete Systeme, Betriebssysteme, Programmable Logic Devices (mit Schwerpunkt FPGAs) und run-time Reconfiguration. Jedem dieser Arbeitsgebiete liegt ein domänenspezifischer Entwurfsablauf zugrunde, der im Verlauf des jeweiligen Abschnitts näher erläutert wird. Diese einzelnen Abläufe werden am Ende des Kapitels in einen gemeinsamen Entwurfsablauf zusammengeführt.

Sowohl Kapitel 3 als auch Kapitel 4 vermitteln dem Leser zuerst spezifische Grundlagen, um dann gezielt weiterführende Diskussionen zum jeweiligen Themengebiet zu führen. Ziel beider Abschnitte ist die Vertiefung des behandelten Themengebietes, um entwurfsbezogene Entscheidungen zu erleichtern.

Die inhaltlichen Schwerpunkte liegen dabei auf Prozessen, die bereits in der Spezifikationsphase zu berücksichtigen sind, deren Auswirkungen aber erst während der Implementierung spürbar werden.

Kapitel 3 konzentriert sich auf die Rekonfiguration von heterogenen rekonfigurierbaren Systemen. Im Speziellen werden konfigurationsbezogene Systemarchitekturen, eingebettete Systeme mit rekonfigurierbarem Modul und betriebssystemspezifische Fragestellungen betrachtet. Die Ausführungen zu Kommunikations- und Konfigurationsarchitekturen sind eine Verallgemeinerung von Ansätzen aus Literatur, Implementierungen des Autors und gängigen Vorschlägen der Hersteller von FPGAs. Nicht alle der diskutierten Architekturen eignen sich für eingebettete Systeme. Das erlaubt eine Vereinfachung der Systemarchitekturen. Es wird deutlich, dass ein direkter Zusammenhang zwischen Konfigurationsstrategie und Systemarchitektur besteht. Dieser Zusammenhang spiegelt sich auch bei den Betrachtungen zu Konfigurationsstrategien und Konfigurationsmanagement aus Betriebssystemsicht wieder. Hier werden vier Konfigurationszeitpunkte eingeführt und diskutiert. Ebenso erfolgt eine Diskussion grundsätzlicher Managementabläufe aus Betriebssystemsicht.

Kapitel 4 befasst sich ausführlich mit dem Betriebssystem Linux. Im Vordergrund stehen Fragen zum Software- und Versionsmanagement. Dennoch startet das Kapitel mit wesentlichen technischen Grundlagen. Es folgt eine Diskussion zu Distributionen sowie zu Zusammenhängen und Abläufen beim Software-, Paket- und Versionsmanagement im Linux-Umfeld. Die Erkenntnisse der Diskussion werden auf eingebettete Systeme übertragen. Anschließend erfolgt die Umsetzung der Konfigurationsstrategien und des Konfigurationsmanagements aus Kapitel 3 mit den technischen Möglichkeiten des Linux-Betriebssystems. Es schließt sich eine ausführliche Diskussion der im Kapitel erarbeiteten Sachverhalte an. Hier werden hauptsächlich die vorgestellten Realisierungsmöglichkeiten im Kontext des Systementwurfes betrachtet.

Die Erkenntnisse und Erfahrungen des Autors mit Linux als Betriebssystem für eingebettete Systeme führten zur Entwicklung einer Software, die den Entwurf eines Linux-Systems unterstützt. Die Konzepte und die Umsetzung dieser Software werden in Kapitel 5 vorgestellt. Neben seiner Funktion und dem daraus entstehenden Gewinn sind das einfache Konzept und die ebenso einfache

Umsetzung die großen Vorteile des Ansatzes. Außerdem gibt der Abschnitt einen allgemeinen Überblick zu Entwurf, Test und Verifikation von Linux-Systemen.

Die Heterogenität der in Kapitel 6 vorgestellten Beispiele und Anwendungen reflektiert die Bandbreite, die die Konzepte der vorhergehenden Kapitel abdecken. Allen Abläufen ist der zugrundeliegende Systementwurf gemein, der vom Autor als Ausgangspunkt für die erfolgreiche Umsetzung gesehen wird. Im Text werden vier konkrete Systeme sowie einige mit ihnen umgesetzte Anwendungen näher erläutert. Jede der Plattformen hat Besonderheiten, die herausgearbeitet und näher betrachtet werden. Außerdem erfolgt eine Darstellung aufgetretener, ungelöster Probleme und eine Interpretation zu ihren Ursachen.

In Kapitel 7 beschreibt der Autor abschließend zwei Konzepte für die effiziente Umsetzung von heterogenen Sensor-Aktor-Systemen. Im ersten Teil des Kapitels werden zwei konkrete Modelle für die Erfassung und zentrale Speicherung von Sensordaten in einer relationalen Datenbank vorgestellt, diskutiert und verglichen. Der zweite Teil des Kapitels erläutert ein modulares Konzept für die Abstraktion von Kommunikationsverbindungen in verteilten Sensornetzwerken. Beide Lösungen stehen in direkter Verbindung mit den Anwendungen aus Kapitel 6. Sie erläutern die grundlegenden Probleme und stellen jeweils prototypische Implementierungen der Lösung vor.

8.2 Ausblick

Derzeit verkürzen sich die Entwicklungszyklen für elektronische Produkte weiter. Gleichzeitig nimmt die Leistungsfähigkeit der Geräte zu. Sie bieten mehr Rechenleistung, mehr Speicher und immer mehr zusätzliche Funktionen. Software wird daher auch in den kommenden Produktgenerationen eine entscheidende Rolle spielen. Durch ihre Flexibilität werden auch FPGAs weiterhin zum Einsatz kommen. Ebenso wird Linux aufgrund seiner Vielseitigkeit in weiteren Produkten das Betriebssystem der Wahl bleiben und werden. All diese Faktoren führen dazu, dass Werkzeuge für den Entwurf eingebetteter Systeme – ob mit oder ohne rekonfigurierbarem Modul – mit Linux als Betriebssystem ein Thema in Forschung und Industrie bleiben werden.

Der im Rahmen dieser Arbeit entwickelte Ansatz für den verbesserten und vereinheitlichten Entwurf von Linux-Systemen bietet verschiedene Möglichkeiten zur Weiterführung und Verbesserung. Wie bei vielen Softwareprojekten umfasst dies Detailverbesserungen der derzeitigen Implementierung und die Integration weiterer Distributionen und Buildsysteme. Die bisherigen Erfahrungen mit der Software zeigen, dass ein aktives Verfolgen von Paketabhängigkeiten die Qualität der entstehenden Linux-Systeme deutlich verbessern würde. Bisher ist dieser Vorgang nicht automatisiert und bietet daher nur wenig Flexibilität. Ebenso kann der bisherige Ansatz zur automatisierten Unterstützung der Verifikation weiter ausgebaut werden.

Auch erscheint die Anwendung des Konzepts auf andere Domänen vielversprechend. Die Erfahrungen des Autors mit den Entwurfsmethoden für Leiterplatten zeigt, dass sich hier deutliche Parallelen zeigen. Auch der Leiterplattenentwurf profitiert von modularem Aufbau und Re-Use. Die Erfahrung der Entwickler führt dazu, dass bestimmte Bauelemente immer wieder zum Einsatz bekommen. Ähnlich dem Ansatz bei komplexer Software sind die Besonderheiten der Bauelemente bekannt und ihre Integration in neue Projekte somit einfacher. Auch Stromlaufpläne von Teilschaltungen werden häufig aus bereits existierenden Projekten kopiert und weiterverwendet. Hingegen sind Bauteilplatzierung und Leiterbildentwurf immer projektspezifisch. Dieses Vorgehen kann mit existierenden EDA-Werkzeugen umgesetzt werden. Häufig sind diese jedoch gerade für kleine Projekte zu komplex oder zu teuer. Das Grundprinzip der Speicherung von Teilschaltungen und Bauteilen mit ihren Besonderheiten sollte sich mit geringem Aufwand realisieren lassen. Auch hier könnten zusätzliche Informationen über Nebenbedingungen („Abhängigkeiten“) hinterlegt und geprüft werden.

Als Konsequenz aus diesen Betrachtungen steht auch einer Kombination beider Ansätze nichts entgegen. Offensichtlicher Nachteil ist die damit einhergehende Komplexität des entstehenden Werkzeugs – sowohl bei der Umsetzung als auch bei der Anwendung. Wesentlicher Vorteil ist die Möglichkeit, auch Abhängigkeiten zwischen Hardware und Software einzuführen und zu überprüfen. So kann bei der Wahl bestimmter Softwarefunktionen ein Hinweis auf fehlende Hardwareressourcen erfolgen. Gleichwohl ist ein Hinweis auf ungenutzte Hardwareressourcen realisierbar, wenn auch weniger naheliegend.

Anhang

A Programmierbare Logikschaltkreise

Als programmierbare Logikschaltkreise (engl. Programmable Logic Device (PLD)) werden gemeinhin alle Schaltkreise bezeichnet, die es dem Entwickler erlauben, logische Funktionen im Schaltkreis zu implementieren. Die Idee wurde bereits in den 1960er Jahren geboren. Zu dieser Zeit brachte Harris Semiconductor einen IC auf den Markt, der im Wesentlichen aus einer programmierbaren Diodenmatrix (Fuse Configurable Diode Matrix) bestand. Seither wurden verschiedene Architekturen und Konzepte programmierbarer Logikschaltkreise vorgestellt und implementiert, die die Bedürfnisse verschiedener Applikationen, Märkte und Kunden abdecken. Diverse Hersteller [Alt09a; Atm; Mic; Sil09; Xil09a] bieten PLDs als kommerzielle Produkte an.

A.1 Klassifikation Programmierbarer Logikschaltkreise

Es gibt verschiedene Ansätze, rekonfigurierbare Schaltkreise zu klassifizieren. So bietet sich beispielsweise die Klassifikation nach Architekturmerkmalen und Komplexität an. Die einfachste Klasse stellen die einfachen programmierbaren Logikschaltkreise (engl. *Simple Programmable Logic Device (SPLD)*) dar, die aus zwei verbundenen Matrizen bestehen und dadurch programmierbare `UND`-`ODER`-Verbindungen zulassen. Vertreter dieser Klasse sind in Tabelle A.1 auf der nächsten Seite aufgeführt.

Die nächste Klasse stellen die komplexen programmierbaren Logikbausteine (engl. Complex Programmable Logic Device (CPLD)) dar. Gegenüber den SPLDs haben diese ICs an Flexibilität gewonnen. Sie bestehen aus mehreren

Tabelle A.1: SPLD-Typen

SPLD-Typ	UND	ODER
Programmable Array Logic (PAL)	variabel	fix
Programmable Read Only Memory (PROM)	fix	variabel
Fieldprogrammable Logic Array (FPLA)	variabel	variabel

Makrozellen, die in ihrem Aufbau SPLDs gleichen. Die Makrozellen sind über Busse miteinander verbunden. Zusätzlich bieten CPLDs eine geringe Anzahl von Flipflops als Zwischenspeicher. Je nach Hersteller und Ausführung sind die Busse konfigurierbar. Außerdem gibt es CPLDs, deren Makrozellen in Funktionsblöcken (engl. *Function Blocks (FBs)*) gruppiert sind. CPLDs werden ebenfalls von verschiedenen Herstellern wie Xilinx oder Altera angeboten, bieten aber bei der Rekonfiguration nicht die notwendige Flexibilität. Daher werden sie in den weiteren Arbeiten nicht explizit betrachtet.

Die Field Programmable Gate Arrays (FPGAs) bilden die letzte und wohl komplexeste Klasse der PLDs. Auch sie haben einen sehr homogenen Aufbau aus konfigurierbaren Logikblöcken, die über ebenfalls konfigurierbare Busse miteinander verbunden sind. Die weitere Ausführung der Logikblöcke, der konfigurierbaren Busse sowie zusätzliche Merkmale der FPGAs sind sehr herstellerspezifisch[1]. Daher haben sich für die Klassifikation von FPGAs weitere Schemata etabliert. Einige dieser Schemata, die dem Leser den Einstieg und das Verständnis der Thematik erleichtern, sollen im Folgenden kurz umrissen werden. Für eine Vertiefung sei auf weiterführende Literatur verwiesen [CH02; VBT04].

Ein häufig verwendetes Schema ist die Klassifikation nach der Granularität, also der minimalen Größe der konfigurierbaren Logikelemente. *Grobkörnige* Architekturen setzen sich aus Funktionsblöcken wie beispielsweise Rechenwerken (ALUs) zusammen. *Feinkörnige* Architekturen erlauben die Konfiguration auf Bitniveau. Bei den feinkörnigen Architekturen bestehen die Logikblöcke meist aus Lookup-Tabellen und Flipflops. In aktuellen Produkten sind mehrere dieser als *Slice* bezeichneten Elemente zu einem *Configurable Logic*

[1] Ein Vergleich der verschiedenen Hersteller findet sich u. a. in [Bec08]. Eine Beschreibung einzelner Architekturen in [Sin09].

Block (CLB) gruppiert. Die CLBs können wiederum über konfigurierbare Busse kombiniert werden. Bei den Produkten der momentan führenden Hersteller (Xilinx [Xil09a], Altera [Alt09a]) ist schon länger eine Vermischung bei der Granularität zu beobachten. Zusätzlich zu den konfigurierbaren Logikblöcken enthalten die ICs Hardmakros[2] für dedizierte Aufgaben, wie DSPs für Berechnungen, dedizierten RAM, Hardware für Bus- und Netzwerkverbindungen oder Prozessor-Cores.

Weitere Klassifikationsansätze beziehen sich auf Arten und Möglichkeiten der Konfiguration von FPGAs. Nach [CH02] ist ein Merkmal, ob die Konfiguration dauerhaft im Schaltkreis gespeichert wird (nicht flüchtig) oder ob sie nach Abschalten der Betriebsspannung verloren geht (flüchtig). ICs mit flüchtigem Konfigurationsspeicher nutzen SRAM. Das macht sie bei der Herstellung günstiger und ermöglicht beliebig viele Konfigurationsvorgänge. Allerdings muss immer ein externer Speicher für die Konfiguration vorhanden sein. Bei den Schaltkreisen mit nicht flüchtigem Speicher bleibt die Konfiguration durch Nutzung von EEPROMs- oder Flash-Zellen auch ohne Betriebsspannung erhalten. Derzeit basiert die überwiegende Zahl der FPGAs auf der SRAM-Technologie.

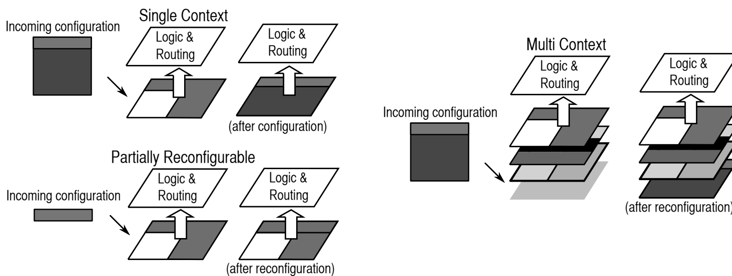


Abbildung A.1: Konfigurationsschichten (Quelle: [CH99])

Ein anderes konfigurationsbezogenes Klassifikationsmerkmal ist nach [CH99] die Anzahl der *Konfigurationsschichten*. Compton unterscheidet drei grundsätzliche Konfigurationsarchitekturen: *Single Context*, *Partially Reconfigurable*

[2] Hardmakro: „Makros, bei denen sowohl die Topologie ihrer interzellulären Verdrahtung als auch ihre äußere Geometrie festliegen, nennt man auch Hardmakros.“ (Quelle: [HM04, S. 140])

Tabelle A.2: Klassifikationsmerkmale für FPGAs

Klassifikationsmerkmal	Eigenschaften
Granularität	Grobkörnig (engl. coarse-grained) Feinkörnig (engl. fine-grained)
Konfigurationsart	nicht flüchtig flüchtig
Konfigurationsschichten	Single Context Partially Reconfigurable Multi-Context

und *Multi-Context*. Als Single Context werden die FPGAs bezeichnet, bei denen bei jedem Konfigurationsvorgang der ganze Schaltkreis konfiguriert wird. Dies trifft auch heute noch auf die meisten FPGAs zu. Bei partiell rekonfigurierbaren ICs ist es möglich, die Konfigurationsbits zu adressieren, sodass nur ein Teil der Konfiguration eines FPGAs geändert werden kann. Auch wenn diese Architektur in der Forschung sehr beliebt ist, hat sie sich noch nicht weitreichend durchgesetzt. Bei Multi-Context FPGAs existieren für jeden Konfigurationspunkt mehrere Speicherbits, die man sich wie mehrere Schichten vorstellen kann. Immer eine der Schichten ist aktiv, während die anderen mit neuen Konfigurationsinformationen beschrieben werden können. Das Umschalten zwischen den Konfigurationsschichten kann sehr schnell erfolgen. Abbildung A.1 auf der vorherigen Seite stellt diese drei Architekturen dar und Tabelle A.2 fasst die soeben vorgestellten Klassifikationsmerkmale noch einmal zusammen.

B Der Entwurfsprozess bei FPGA

Dieser Abschnitt umreißt den Entwurfsablauf für ein FPGA-Design und führt die damit verbundenen Begriffe ein. Der Entwurfsablauf umfasst die notwendigen Schritte, um ein funktionsfähiges FPGA-Design zu erhalten. Nach [HM04] ist der Entwurfsprozess einer der wichtigsten Schritte bei der Entwicklung von spezifikationsgerechten Schaltungen und Schaltkreisen. Alle Eigenschaften des Designs werden in der *Systemspezifikation* erfasst. Ausgehend davon erfolgt der *Schaltungsentwurf*, bei dem die Schaltung in ein für die weitere Verarbeitung geeignetes Format gebracht wird. Wie bereits in Abschnitt 2.2 beschrieben, gliedert sich der Prozess in mehrere Entwurfsebenen. In jeder dieser sechs Ebenen sind verschiedene Aufgaben durchzuführen, die im Folgenden vorgestellt werden und in Abbildung B.1 auf der nächsten Seite dargestellt sind.

1. *Systemebene:* Die Systemebene dient der Festlegung globaler Eigenschaften des Systems. Darunter zählen zum Beispiel technische Parameter wie Arbeitsfrequenz, Verlustleistung oder Temperaturbereich, aber auch nichttechnische Parameter wie Kompatibilität oder die Aufteilung von Hard- und Software. In der Systemebene erfolgt ebenfalls eine erste Aufteilung des Gesamtsystems in kleinere Subsysteme. Eine Codierung der Datenformate sowie ein Zeitmodell liegen in dieser Ebene noch nicht vor. Vielmehr existiert lediglich ein Kausalitätsschema, welches gewisse Abhängigkeiten einzelner Komponenten voneinander darstellt.

2. *Algorithmische Ebene:* In dieser Entwurfsebene erfolgt die Festlegung von Befehlssätzen, Datenbreiten, Speichergrößen, Ein- und Ausgabe usw.. Hier werden außerdem Entscheidungen über die Notwendigkeit eines Betriebssystems getroffen. Die Formate sowie die Codierung von Informationen wird in dieser Ebene bereits teilweise festgelegt. Das

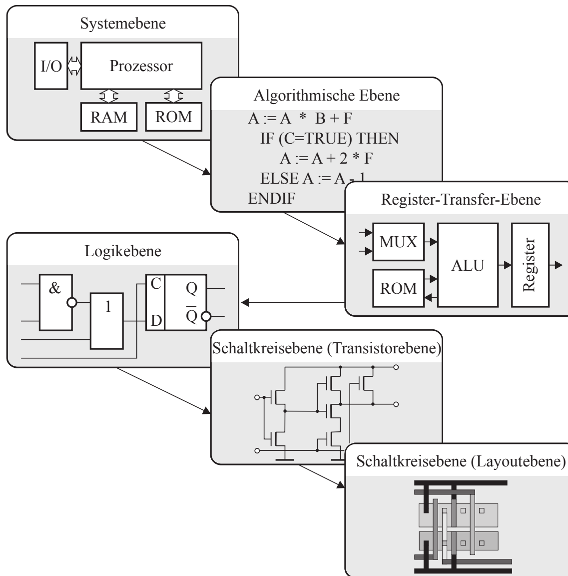


Abbildung B.1: Schematische Darstellung der Entwurfsebenen (Quelle: [HM04])

Zeitmodell der algorithmischen Ebene ist eine Verfeinerung des Kausalitätsschemas der Systemebene.

3. *Register-Transfer-Ebene:* Die *Register-Transfer-Ebene (RT-Ebene)* dient zur Definition von Größe und Anzahl der benötigten Register sowie Art und Anzahl der Verarbeitungswerke. Auch werden Entscheidungen zum Steuer- und Operationswerk des Systems getroffen. Es muss beispielsweise festgelegt werden, ob eine Mikroprogrammsteuerung oder eine Folgesteuerung verwendet werden soll. In der RT-Ebene ist die Codierung aller Informationen vollständig beschrieben. Zum ersten Mal im Entwurfsprozess wird eine grobe Abschätzung der Zeit vorgenommen. Das verwendete Zeitmodell sieht hier eine Zählung von Takten (Zeitabschnitten) vor.

4. *Logikebene:* Auf der Logikebene wird der Schaltungsentwurf mit Makrobibliotheken durchgeführt. Es werden sowohl logische als auch zeit-

liche Parameter der Funktionselemente festgelegt. Darunter zählen Verzögerungszeiten oder der Eingangslastfaktor (Fan-In) der Makros. Die gesamte Schaltung liegt nun als uninterpretiertes System stimulierbarer Boole'scher Gleichungen vor.

5. *Schaltkreisebene (Transistorebene):* Auf dieser Ebene werden die Transistordimensionen der Schaltung festgelegt. Dies geschieht mittels Netzwerkanalyse. Das Zeitmodell der Transistorebene sieht nun einen kontinuierlichen Zeit- und Werteverlauf vor.

6. *Schaltkreisebene (Layoutebene):* Die letzte Entwurfsebene ist die Layoutebene. Das gesamte System liegt nun als konkrete Schaltung mit genauen Maßen vor.

Zur Entwurfsunterstützung können vorgefertigte Blöcke (IP-Cores) genutzt werden. Am Ende des Entwurfs existiert eine synthesefähige, durch Simulation verifizierte Beschreibung des Designs. Es folgt die *Schaltungssynthese*. In diesem mehrstufigen Entwurfsschritt wird die Schaltungsbeschreibung mit Hilfe von Synthesewerkzeugen in eine Netzliste überführt. Bei der abschließenden *Systemimplementierung* wird die Netzliste auf das tatsächlich verwendete FPGA abgebildet. Am Ende dieses Schrittes existiert eine als *Bitstrom* bezeichnete Datei, mit der sich das FPGA konfigurieren lässt. Der Abgleich zur Systemspezifikation erfolgt in den einzelnen Designphasen durch wiederholte Simulation und nach der Erzeugung des Bitstroms zusätzlich durch In-System Verifikation mit einer realen Schaltung.

Es gibt verschiedene Entwurfsabläufe (engl. *Design-Flows*) speziell für FPGAs, sowohl herstellerübergreifende als auch herstellerspezifische, wie die von Xilinx angebotenen ISE, EDK und *Xilinx Vivado™ Design Suite (Vivado)* [Xil12a; Xil11a; Xil12e]. Eine gute Zusammenfassung zum Xilinx Design-Flow, wie er auch für die Umsetzung in Kapitel 6 ab Seite 145 genutzt wurde, findet sich in [Pan10, Abschnitt 3.5 ab Seite 14].

Verifikation im digitalen Schaltungsentwurf Die Entwurfs- und Verifikationsmethoden sowie die Möglichkeiten für den automatisierten Entwurf und die automatisierte Verifikation sind im digitalen Hardwareentwurf sehr ausgereift. Dies ist unter anderem durch die hohen Kosten begründet, die ein Fehler verursacht, der erst in einem sehr späten Entwurfsstadium oder gar

im fertigen Produkt gefunden wird. Daher wird im gesamten Entwurfsablauf die Verifizierbarkeit des Systems angestrebt. Da die Verifikation durch Simulation bei der heutigen Komplexität digitaler Schaltungen (wie z. B. System-on-Chips) nicht mehr möglich ist, versucht man mathematische Methoden anzuwenden, was als *formale Verifikation* [SR13; HT10] bezeichnet wird.

C Versionsmanagement mit Git

OpenWrt sieht von Haus aus nicht vor, dass mehrere Entwickler mit einem Grundsystem arbeiten. Allerdings können zusätzliche Pakete (sog. feeds) bequem hinzugefügt werden. In Zusammenhang mit den Arbeiten zum Versionsmanagement bei Linux-Systemen (vgl. Abschnitt 4.4.4) wurde durch den Autor auch eine Methodik entwickelt, die die Arbeit mit einer Version des Grundsystems vereinfacht. Dazu wird ein eigenes git-Repository gepflegt, das in unregelmäßigen Abständen mit dem Subversion-Repository von OpenWrt abgeglichen wird. In Abbildung C.1 auf der nächsten Seite ist das der externe Zweig. Dieser entwickelt sich unabhängig von den eigenen Arbeiten. Nach jedem Abgleich werden die Änderungen beider Versionen zusammengeführt. Anschließend folgt die Integration der eigenen Erweiterungen und der Test des Buildsystems, bevor intern auf das so entstandene Basissystem umgestellt wird.

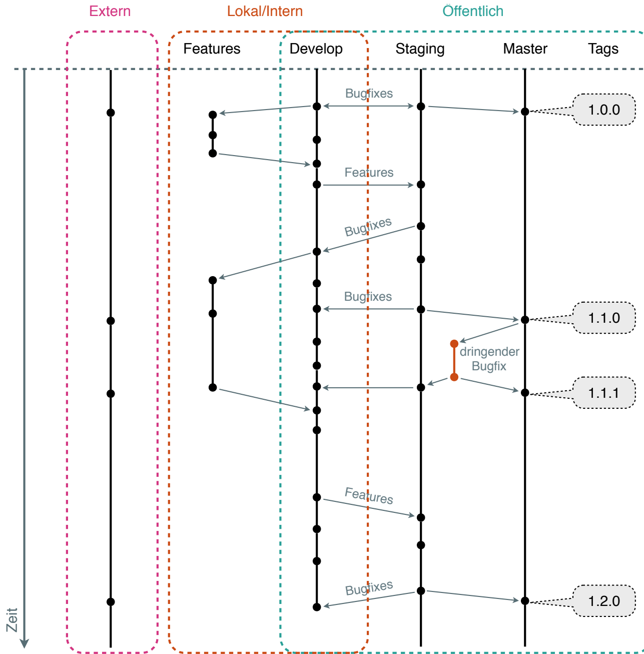


Abbildung C.1: Versionsmanagement der Entwicklungsumgebung mit Git

D Buildsysteme

Aus den Betrachtungen in Kapitel 4 ab Seite 95 und aus den Untersuchungen verschiedener Distributionen und Buildsysteme [KKH11] für eingebettete Systeme werden an dieser Stelle zwei konkrete Ansätze hervorgehoben. Diese setzen die formulierten Ansprüche an Flexibilität, Unabhängigkeit, Aktualität und Reproduzierbarkeit sehr gut um.

PTXdist PTXdist [Pen12] der Pengutronix e. K. ist ein Buildsystem, dass eine sehr saubere Trennung der einzelnen Komponenten (Buildsystem, Toolchain, Werkzeuge des Host-Betriebssystems, Board Support Packages) und der Paketauswahl realisiert (Abschnitt 4.4.4 auf Seite 119), was die Reproduzierbarkeit deutlich erhöht. Besonders interessant ist die Trennung von hardwareabhängigen und hardwareunabhängigen Teilen der Firmware, die direkt im Ansatz des Buildsystems verankert ist und somit einen einfachen Wechsel zwischen verschiedenen Hardwareplattformen ermöglicht. Mit regelmäßigen Releases, einer hinreichend aktuellen Dokumentation, einer aktiven Entwicklergemeinde und aktiv betreuten Mailinglisten für den Support ist der Einstieg in das System mit vertretbarem Aufwand realisierbar. Das per DVVS verwaltete Buildsystem wird zentral installiert und kann in mehreren Versionen parallel existieren. Es erlaubt lokale Erweiterungen für Pakete und Konfigurationsdateien und nutzt eine exklusive Patchstrategie. Die Konfiguration der Pakete und des BSP erfolgen per *kconfig*.

Gentoo Linux Gentoo Linux [Wro08] ist eine Desktop-Distribution mit Rolling Releases, bei der die Pakete erst während der Installation auf dem lokalen System kompiliert werden. Das *portage* genannte Paketmanagement der Distribution liefert lediglich die dafür notwendigen „Bauanleitungen". Dieser

Ansatz widerspricht im ersten Moment dem Problem der mangelnden Ressourcen einer möglichen Zielplattform, doch portage erlaubt auch die Nutzung von Binärpaketen und unterstützt im Ansatz deren Cross-Kompilation. Auch wenn es eine Entwicklergruppe gibt, die die Nutzung von Gentoo Linux für eingebettete Systeme unterstützt, bereitet das Cross-Kompilieren an vielen Stellen noch größere Probleme, sodass ein Einsatz als produktives Buildsystem nicht von Vorteil erscheint. Allerdings nutzt die Distribution ein sehr interessantes und dynamisches System für die Verwaltung von optionalen Abhängigkeiten. Diese werden durch sogenannte *Use Flags* systemweit oder pro Paket definiert und können jederzeit angepasst werden. Die Verwaltung der Use Flags erfolgt an zentraler Stelle. Damit existiert ein sehr mächtiger Mechanismus, um im System vorhandene Pakete effizient zu nutzen. Der sehr einfachen Installation von Paketen (`emerge <Paketname>`) steht eine feingranulares Abhängigkeitsmanagement zur Seite. Das Buildsystem OpenEmbedded [STM10] orientiert sich an den Mechanismen von Gentoo.

Ein wesentlicher Fortschritt wäre die Kombination beider Mechanismen, sodass „globale“ Funktionen (etwa die Verfügbarkeit einer Bibliothek) durch alle an einem System beteiligten Pakete wahrgenommen werden. Dies stellt allerdings einen sehr weitreichenden Eingriff in die Funktionen des Buildsystems oder in die Softwareerstellung der Gentoo-Distribution dar. Daher ist es nicht möglich, eines der beiden Systeme um die Funktionen des anderen zu ergänzen. Die Entwicklung eines eigenen Buildsystems, dass beide Ansätze kombiniert, bietet nicht hinreichend Vorteile gegenüber den existierenden Systemen.

E Linux High-Level Spezifikation

E.1 Alternative Konzepte des Synthesesystems

Neben der in Abschnitt 5.2 vorgestellten Architektur wurden zwei weitere Konzepte untersucht, die im Folgenden kurz gegenüber gestellt werden.

Abbildung E.1: Implementierung mit einem Werkzeug

Die in Abbildung E.1 dargestellte Variante stellt die komplexeste Umsetzung dar. Ein Werkzeug sorgt für die Umsetzung von der Eingangsbeschreibung in das jeweilige Zielformat. Die Umsetzungsvorschrift wird dabei in einer separaten Datei hinterlegt. Hier bietet sich ein strukturiertes Format wie *Extensible Markup Language (XML)* an. Nach der Fertigstellung des Werkzeugs ist

somit keine Arbeit am Quelltext mehr notwendig. Neue Ausgabeformate werden ausschließlich über die Übergangsbeschreibung definiert. Allerdings erweist sich die Implementierung als schwierig, da die Zielformate zu viele Unterschiede haben. So erwartet OpenWrt eine Datei in Kconfig-Format, PTXdist zwei Dateien und das Ausgangsformat für OpenEmbedded ist gänzlich anders. Die Abbildungsvorschriften werden schnell sehr komplex und damit schlecht wartbar.

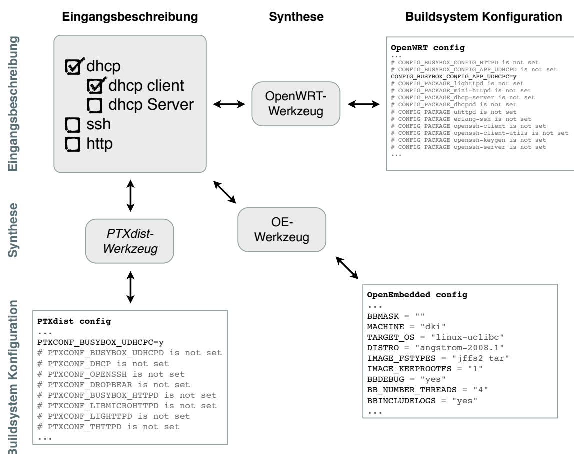


Abbildung E.2: Implementierung mit einem Werkzeug je Ausgangsformat

Für erste Versuche entstanden einzelne Werkzeuge, die ausgehend von einer beispielhaften Eingangsbeschreibung eine zielsystemspezifische Synthese vornahmen. Gemeinsame Schnittstelle bildet somit einzig die Beschreibung der Eingangskonfiguration. Allerdings doppeln sich viele Teile der Implementierung, speziell beim Einlesen der Eingangsbeschreibung. Eine Änderung des Formates erfordert zudem eine Anpassung aller Werkzeuge.

E.2 Alternative Implementierungsvariante

Als Schnittstelle nahezu ohne Abhängigkeit auf Nutzerseite bieten sich Internetseiten an. Daher wurde als Alternative zum kconfig-basierten Ansatz

die Implementierung des in Abschnitt 5.2 vorgestellten Systems mit Hilfe eines Web-Frameworks umgesetzt. Dabei wurde auf das in Python geschriebene Web-Framework *django* zurückgegriffen. Django bietet unter anderem ein Nutzersystem sowie einen Object-Relational-Mapper für die bequeme Verwaltung von Daten in einer Datenbank. Für die Datenhaltung wurde dementsprechend auf eine SQL-Datenbank zurückgegriffen. Im Frontend kommen *Hypertext Markup Language (HTML)*-Formulare und -Auswahllisten zum Einsatz. Die einzelnen Synthesewerkzeuge sind als Django-Plugins umgesetzt. Die Schnittstellen und Austauschformate sind somit durch das Datenbankmodell und das Django-Framework vorgegeben. Für die jeweilige Umsetzung der Spezifikation in eine Buildsystem-spezifische Konfiguration zeichnen die Plugins verantwortlich. Neben einer Umsetzung basierend auf Python-Dicts wurde auch die Nutzung einer weiteren Datenbank für das Mapping untersucht. Speziell letzteres bietet sehr interessante Möglichkeiten, da über die in Django vorhandenen Mechanismen auch die Umsetzung von Nutzerschnittstellen zur Erweiterung der Synthesewerkzeuge erfolgen kann. Das Konzept dazu ist in Abbildung E.3 veranschaulicht. Neben der Datenbank mit den Informationen zum Frontend existiert eine Datenbank mit den Synthesevorschriften. Über ein zusätzliches Web-GUI können hier neue Vorschriften eingebracht werden. Damit nähert sich diese Implementierung etwas an die aus Abbildung E.1 auf Seite 235 an.

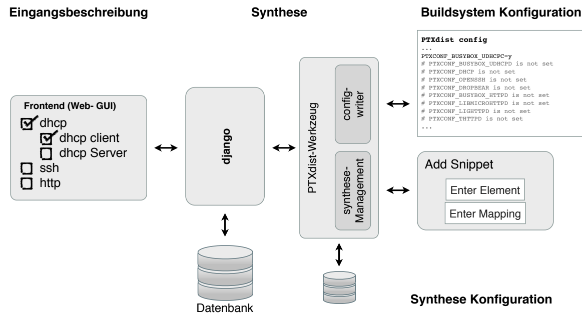


Abbildung E.3: Django basiertes Konzept der High-Level Konfiguration

E.3 Das kconfig-System

Im Quellcode des Linux-Kernels wird zur Konfiguration der verschiedenen Kernel-Optionen ein Text-basiertes Konfigurationssystem verwendet. Es ist unter dem Namen *kconfig*-System bekannt. Inzwischen wird das System auch von anderen bekannten Projekten wie Buildroot oder PTXdist verwendet. Es basiert auf einer einfachen, textuellen Beschreibung der Nutzerschnittstelle, wie in Listing E.1 dargestellt. Diese wird durch einen Satz von Werkzeugen (z. B. `mconf`, `xconf`, `qconf`) zur Anzeige gebracht.

Nach Beendigung des kconfig-Dialogs speichern die Werkzeuge die vom Nutzer getroffene Auswahl in einer einfachen Textdatei, die meist mit *.config* benannt ist. Listing E.2 auf der nächsten Seite zeigt einen zu Listing E.1 passenden Ausschnitt aus einer .config-Datei.

```
mainmenu "Linux␣High-Level␣Spezifikation"

choice TargetFramework
  prompt "TargetFramework"
  help
    The buildsystem or distribution to use

  config PTXDIST
    bool "Ptxdist"
  config BUILDROOT
    bool "Buildroot"
  config OPENWRT
    bool "OpenWrt"
  config DEBIAN
    bool "Debian"
endchoice

config KERNEL
  string "Kernel␣Version"
  default "3.5.0"
  help
    The Linux kernel version to use

choice C-Library
  prompt "The␣c␣library␣to␣use"
  default LIBC_GLIBC
  help
    The C library to choose

```

```
  config LIBC_GLIBC
    bool "glibc"
  config LIBC_EGLIBC
    bool "eglibc"
  config LIBC_UCLIB
    bool "uclib"

endchoice
```

Listing E.1: Ausschnitt aus einer kconfig-Datei

```
# Automatically generated file; DO NOT EDIT.

#
# TargetFramework
#
HLCONF_PTXDIST=y
# HLCONF_BUILDROOT is not set
# HLCONF_OPENWRT is not set
# HLCONF_DEBIAN is not set

#
# Kernel
#
HLCONF_KERNEL="3.5.0"

#
# C-Library
#
# HLCONF_LIBC_GLIBC is not set
# HLCONF_LIBC_EGLIBC is not set
HLCONF_LIBC_UCLIB=y
```

Listing E.2: Ausschnitt aus einer .config-Datei

F Das Projekt „GPSV“

Ziel des Projekts *Generalisierte Plattform zur Sensordatenverarbeitung* (*GPSV*), war die Entwicklung und Umsetzung von Konzepten für eine zentrale, kompakte, flexible Verarbeitungseinheit zur Erfassung, Aggregation, Speicherung und (Vor-) Verarbeitung von Sensordaten, in Zusammenarbeit mit regionalen *kleinen und mittelständischen Unternehmen (KMU)* [Kri+06; Kri11]. Die Plattform sollte die Anbindung von Sensoren über standardisierte Schnittstellen ebenso ermöglichen wie die Nutzung vorhandener Sensoren mit nicht standardisierten Schnittstellen. Neben der Orientierung am Automobilmarkt wurde die Eignung des Konzepts auch für andere Anwendungsbereiche angestrebt. Trotz der zentralen Komponente sollte das Konzept die Einbeziehung intelligenter Sensoren mit der Möglichkeit zur Vorverarbeitung der erfassten Daten berücksichtigen und auch die Vernetzung mehrerer zentraler Systeme zulassen. Die Berücksichtigung der Flexibilität des Systems schon in der Entwurfsphase diente der Vereinfachung der Anpassung der entstehenden Plattform auf weitere Zieldomänen. Robustheit, Ausfallsicherheit und Echtzeitverhalten des Systems waren dabei wesentliche, das Konzept und den Entwurf beeinflussende Größen. Abbildung F.1 auf der nächsten Seite illustriert das zugrundeliegende Konzept.

Zum Zeitpunkt der Beantragung und des Projektstarts zeichnete sich im Bereich der eingebetteten Systeme, an dem sich das *Generalisierte Plattform zur Sensordatenverarbeitung (GPSV)*-Projekt orientierte, der verbreitete Einsatz von Linux als Betriebssystem ab. Die Entwicklung von Hardware für eingebettete Systeme war geprägt von der Erhöhung der Funktionsdichte auf Chip- und Board-Ebene sowie vom Streben nach Energieeffizienz bei gleichzeitig steigender Verarbeitungsleistung. Sowohl die Hardware als auch die Software sollten dabei an Sicherheit, Robustheit und Flexibilität gewinnen. Aus ökonomischer Sicht war die Senkung der Entwicklungs- und Wartungskosten von Interesse.

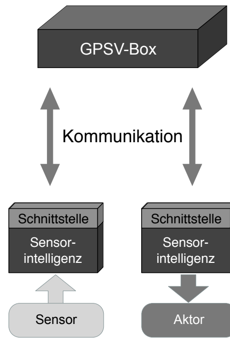


Abbildung F.1: Konzept der *GPSV*-Plattform

Wissenschaftliche Grundlage für die geplanten Arbeiten bildeten die Themengebiete Hardware/Software Co-Design und dynamisch rekonfigurierbare Architekturen. Ein für das Projekt wesentlicher Punkt innerhalb des HW/SW Co-Designs, der zur Beantragungszeit an Relevanz gewann, war die High-Level Synthese. Hierbei wird aus einer Systembeschreibung auf möglichst hohem Abstraktionsniveau sowohl Hardware als auch Software generiert, wobei verschiedene Kriterien der Spezifikation und der Optimierung berücksichtigt werden. Im Bereich der rekonfigurierbaren Architekturen waren besonders die dynamische und die dynamisch-partielle Rekonfiguration von FPGAs von Interesse.

Das Projekt GPSV wurde im Zeitraum April 2006 bis März 2011 durch das Bundesministerium für Bildung und Forschung im Rahmen der Innovationsinitiative „Unternehmen Region“ gefördert. (Fkz.: 03IP505)

G Details zur Umsetzung der Software auf dem AVR32

G.1 Verarbeitung von XSVF-Dateien

Die Daten zur Konfiguration von PFPs bzw. FPGAs liegen im XSVF Format vor. Das XSVF besteht aus einer Folge von Befehlen (vgl. Tabelle G.1) und Daten. Die Befehle müssen zur Stimulation des TAP-Controllers in korrespondierende JTAG-Signale umgesetzt werden.

Tabelle G.1: Übersicht der XSVF Befehle

Befehl	Code	Kurzbeschreibung
XCOMPLETE	0 × 00	Dateiende
XTDOMASK	0 × 01	Maskierung für TDO
XSIR	0 × 02	Schieben von TDI ins Befehlsregister
XRUNTEST	0 × 04	Wartezeit im TAP-Zustand Runtest–Idle
XREPEAT	0 × 07	Anzahl Wiederholversuche bei Fehlschlag
XSDRSIZE	0 × 08	Argumentlänge für XSDRTDO
XSDRTDO	0 × 09	Schieben von TDI ins Boundary Scan Reg.
XSTATE	0 × 12	Go to Testlogic–Reset/Runtest–Idle

Abbildung G.1 auf der nächsten Seite zeigt einen Ausschnitt des dafür notwendigen Programmablaufs. Für alle in Tabelle G.1 aufgeführten Befehle existiert im dargestellten Flow-Chart Diagramm ein Zweig zur Abarbeitung. Die Verarbeitung der Befehle XSDRTDO und XSIR ist komplexer und verlangt jeweils eine ausführlichere Behandlung, für die auf die zugrundeliegende Literatur [Xil07b] verwiesen wird. Alle Kommandos beginnen mit einem

Start-Signal, gefolgt von einem Kommando-Byte, das den weiteren Ablauf bestimmt.

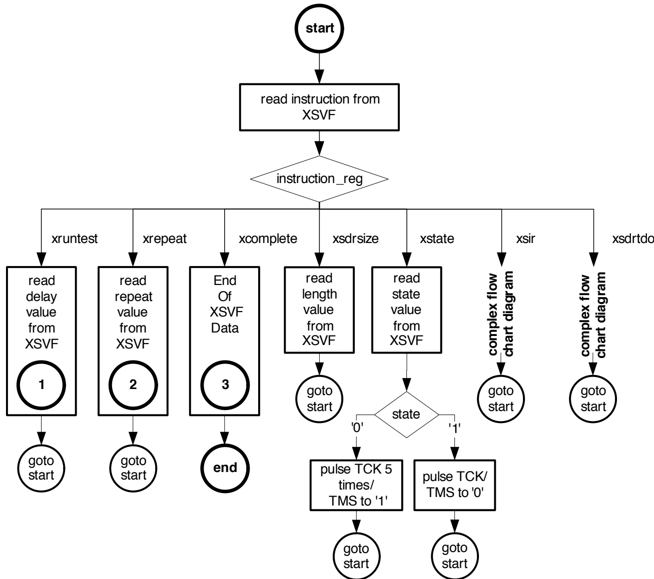


Abbildung G.1: XSVF Programmablauf (vgl. [Xil07b])

G.2 Exemplarische Darstellung eines Kernel-Treibers

Abbildung G.2 auf der nächsten Seite zeigt das Prinzip der Einbindung eines Gerätetreibers in das Linux-Betriebssystem. Über eine Gerätedatei im Dateisystem unter `/dev/` erfolgt der Zugriff der User Space Software auf den Gerätetreiber.

Durch die Verwendung der standardisierten Kernel-Schnittstelle und der zugehörigen Funktion wie `mod_init( )`, `mod_open( )`, `mod_read( )`,

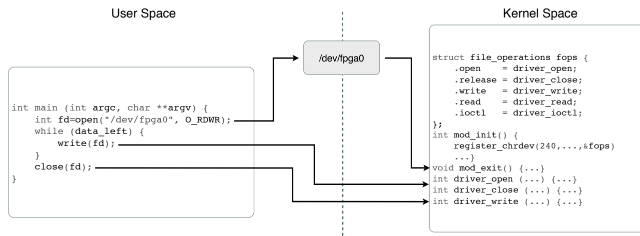


Abbildung G.2: Exemplarische Darstellung eines Kernel-Treibers

`mod_write()`, `mod_close()` und `mod_exit()` erfolgt die Einbindung des Treibers in den Kernel sowie die Steuerung und der Datenaustausch mit dem User Space. Weitere Informationen zur Umsetzung von Gerätetreibern bieten u. a. [BP05; CRK05; Kro06; KQ06].

Literatur

[02] *IEEE Standard Test Access Port and Boundary-Scan Architecture*. IEEE Std. 1149.1-2001. Aug. 2002.

[04] *IEEE Standard for Information Technology – Portable Operating System Interface (POSIX)*. IEEE Std. 1003.1-2001. Apr. 2004.

[94] *IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices*. IEEE Std. 1275-1994. Aug. 1994.

[Ach05] Albrecht Achilles. *Betriebssysteme: Eine kompakte Einführung mit Linux*. eXamen.press. Springer Verlag, 2005. ISBN: 9783540238058.

[AF13] Konstantin Agouros und Markus Feilner. „Unattended Upgrades: Paketmanager im Vergleich“. In: *Linux Magazin* 9 (Aug. 2013), S. 22–28. ISSN: 1432-640 X.

[Alt06] Altera Corporation. *FPGA Integration Increases Flexibility, Reduces Cost in Consumer Applications*. White Paper. Altera Corporation, 2006.

[Alt08a] Altera Corporation. *Error Detection and Recovery Using CRC in Altera FPGA Devices*. Application Note 357. Altera Corporation, Juni 2008.

[Alt08b] Altium Limited. *Altium Designer Training Module – FPGA Design Basics*. Techn. Ber. Altium Limited, 2008.

[Alt09a] Altera Corporation. *Altera Homepage*. Nov. 2009. URL: `http://www.altera.com` (besucht am 02. 12. 2013).

[Alt09b] Altera Corporation. *Configuration Handbook*. Bd. 3.0. Altera Corporation, 2009.

[Alt10] Altera Corporation. *The Nios II Embedded processor*. 2010. URL: http://www.altera.com/products/ip/processors/nios2/ni2-index.html (besucht am 02. 12. 2013).

[Alt12a] Altera Corporation. „9 . Configuration, Design Security, and Remote System Upgrades in Stratix V Devices“. In: *Stratix V Device Handbook*. Bd. 2. 02. 2012, S. 1–60.

[Alt12b] Altera Corporation. „Volume 1: Device Interfaces and Integration“. In: *Stratix V Device Handbook*. Altera Corporation, Dez. 2012. Kap. 8: Configuration, Design Security and Remote System Upgrades in Stratix V Devices.

[Amm87] Peter Ammon. *Entwurf von Leiterplatten*. Hüthig, 1987. ISBN: 9783778510643.

[And09] Erik Andersen. *buildroot Projekt-Homepage*. Nov. 2009. URL: http://buildroot.uclibc.org/.

[And99] Erik Andersen. *μClibc*. 1999. URL: http://www.uclibc.org/.

[Atm] Atmel Corporation. *Field Programmable Gate Array, Project Website*. URL: http://www.atmel.com/products/other/field_programmable_gate_array/default.aspx (besucht am 13. 02. 2013).

[Atm07] Atmel Corporation. *AT32AP7000 Preliminary*. Manual. Atmel Corporation, Sep. 2007.

[Atm09] Atmel Corporation. *ATMEL AVR32 Produktseite*. Nov. 2009. URL: http://www.atmel.com/products/microcontrollers/avr/32-bitavruc3.aspx (besucht am 21. 11. 2013).

[BAK96] Duncan A. Buell, Jeffrey M. Arnold und Walter J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine (Systems)*. Wiley-Blackwell, 1996. ISBN: 081867413X.

[Bal00] Helmut Balzert. *Lehrbuch der Software-Technik: Teil 1: Software-Entwicklung*. 2. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2000. ISBN: 3-8274-0480-0.

[BB04] Miloš Bečv'ař und Tom'aš Brabec. „Implementation of Configurable System on a Chip with MicroBlaze Processor Core“. In: *Proceedings of IFAC Workshop on Programmable Devices and Systems PDS2004*. 2004, S. 222–227. ISBN: 83-908409-8-7.

[BC09] Brendan Bridgford und Justin Cammon. *SVF and XSVF File Formats for Xilinx Devices*. Application Note. Xilinx Inc., Aug. 2009.

[BCV06] Koen. Bertels, João M. P. Cardoso und Stamatis Vassiliadis. *Reconfigurable computing : architectures and applications*. Hrsg. von Koen. Bertels, João M. P. Cardoso und Stamatis Vassiliadis. Springer Verlag, 2006. ISBN: 9783540367086.

[Bec08] René Beckert. *Untersuchung zur Kostenoptimierung für Hardware-Emulatoren durch Anwendung von Methoden der partiellen Laufzeitrekonfigurierung*. Bd. 7. Wissenschaftliche Schriftenreihe Eingebettete, Selbstorganisierende Systeme. Dresden: TUDpress, Okt. 2008.

[Bee98] Gerard Beekmans. *Linux From Scratch!, Homepage*. 1998. URL: `http://www.linuxfromscratch.org/index.html` (besucht am 15. 06. 2013).

[Ben+08] Günther Bengel u. a. *Masterkurs Parallele und Verteilte Systeme*. Vieweg+Teubner, 2008. ISBN: 9783834803948.

[Ber95] Jean-Michel Bergé. *High-Level System Modeling: Specification Languages*. Hrsg. von Oz Levia und Jacques Roulliard. Norwell, MA, USA: Kluwer Academic Publishers, 1995. ISBN: 0792396324.

[BO91] R. G. Bennetts und A. Osseyran. „IEEE standard 1149.1-1990 on boundary scan: History, literature survey, and current status“. English. In: *Journal of Electronic Testing* 2 (1991), S. 11–25. ISSN: 0923-8174.

[Bob07] Christophe Bobda. *Introduction to Reconfigurable Computing: Architectures, algorithms and applications*. Springer Verlag, Nov. 2007. ISBN: 978-1-4020-6088-5.

[BP05] Daniel Bovet und Marco Cesati Ph. *Understanding the Linux Kernel*. Third Edition. O’Reilly Media, 2005. ISBN: 0596005652.

[Bur+01] Jochen Burkhardt u. a. *Pervasive Computing: Technology and Architecture of Mobile Internet Applications*. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 2001. ISBN: 0201722151.

[BW03] N.W. Bergmann und J. Williams. „The Egret platform for reconfigurable system on chip“. In: *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*. Dez. 2003, S. 340–343.

[Car99] Carl Carmichael. *Configuring Virtex FPGAs from Parallel EPROMs with a CPLD*. Application Note 137. Xilinx Inc., März 1999.

[Cat05] John Catsoulis. *Designing Embedded Hardware*. 2. Aufl. O'Reilly Media, Inc., 2005. ISBN: 0596007558.

[CH02] Katherine Compton und Scott Hauck. „Reconfigurable computing: a survey of systems and software“. In: *ACM Computing Surveys (CSUR)* 34.2 (2002), S. 171–210. ISSN: 0360-0300.

[CH99] Katherine Compton und Scott Hauck. „Programming Architectures For Run-Time Reconfigurable Systems“. Master Thesis. Northwestern University Evanston, 1999.

[Cha09] Scott Chacon. *Pro Git*. Books for professionals by professionals. Apress, 2009. ISBN: 9781430218333.

[CKK06] Yong Hoon Choi, Woo Il Kwon und Heung Nam Kim. „Code generation for Linux device driver“. In: *8th International Conference Advanced Communication Technology*. Bd. 1. IEEE, 2006, S. 734–737. ISBN: 89-5519-129-4.

[Cre] Creative Commons. *Creative Commons, Project Website*. URL: http://creativecommons.org/ (besucht am 10. 02. 2013).

[CRK05] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman. *Linux Device Drivers*. 3. Aufl. O'Reilly Media, 2005.

[deu10] deusinvictus. *openclipart – Laptop by deusinvictus*. März 2010. URL: http://openclipart.org/detail/32353/laptop-by-deusinvictus.

[Deu85] Deutsches Institut für Normung. *Informationsverarbeitung DIN 44300*. Techn. Ber. Deutsches Institut für Normung, 1985.

[DF00] Chris Dunlap und Tom Fischaber. *Configuring Xilinx FPGAs Using an XC9500 CPLD and Parallel PROM*. Application Note 079. Xilinx Inc., Juli 2000.

[DH03] Sumanth Donthi und R. L. Haggard. „A survey of dynamically reconfigurable FPGA devices“. In: *35th Southeastern Symposium on System Theory*. 2003, S. 422–426. ISBN: 0780376978.

[Dri10] Vincent Driessen. *A successful Git branching model*. Jan. 2010. URL: `http://nvie.com/posts/a-successful-git-branching-model/` (besucht am 21. 06. 2013).

[Dye12] David Dye. *Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite*. White Paper 1.2. Xilinx Inc., Mai 2012.

[elk07] elkbuntu. *openclipart – Random Access Memory by elkbuntu*. Aug. 2007. URL: `https://openclipart.org/detail/4958/random-access-memory-by-elkbuntu`.

[Eto07] Emi Eto. *Difference-Based Partial Reconfiguration*. Application Note 290. Xilinx Inc., Dez. 2007.

[Fai08] Florian Fainelli. „The OpenWrt embedded development framework“. In: *Proceedings of the Free and Open Source Software Developers European Meeting*. 2008.

[Fär09] Georg Färber. *Eingebettete Systeme*. Manuskript zur Vorlesung. Technische Universität München, 2009.

[FDT] FDTWiki, Project Website. *FDTWiki, Project Website*. URL: `http://www.devicetree.org/Main_Page` (besucht am 25. 09. 2013).

[FHK13] M. Feilner, M. Huber und J. Kleinert. „Paket-Service im Test: Ob und wie Distributionen auf Lücken, Bugs und neue Funktionen reagieren“. In: *Linux Magazin* 3 (März 2013), S. 22–26.

[Fre] Free Software Foundation. *The GNU C Library, Project Website*. URL: `http://www.gnu.org/software/libc/` (besucht am 30. 05. 2013).

[Fre01] Free Software Foundation Europe. *Was ist Freie Software? – FSFE, Project Website*. 2001. URL: `http://fsfe.org/about/basics/freesoftware.de.html` (besucht am 09. 02. 2013).

[Fre11] Martin Frenzel. „Voruntersuchungen zur Funkanbindung eines mobilen verteilten eingebetteten Systems“. Studienarbeit. Technische Universität Chemnitz, März 2011.

[Fri07] Paul Friedrich. „Datenkommunikation über abstrahierte Datenkanäle“. Diplomarbeit. Technische Universität Chemnitz, 2007.

[GC09] Balkrishna Sharma Gukhool und Soumaya Cherkaoui. „Handoff in IEEE 802.11p-based vehicular networks“. In: *Wireless and Optical Communications Networks, 2009. WOCN ’09. IFIP International Conference on*. 2009, S. 1–5.

[Gen01] Gentoo Foundation, Inc. *Gentoo Linux – Project Website*. 2001. URL: `http://www.gentoo.org/` (besucht am 17. 06. 2013).

[GH06] David Gibson und Benjamin Herrenschmidt. *Device trees everywhere*. White Paper. OzLabs, 2006.

[GR09] Thomas Gschossmann und Thomas Rothoerl. „Studienarbeit: Buildumgebungen“. Studienarbeit. Hochschule Augsburg, Sep. 2009.

[Hal06] Christopher Hallinan. *Embedded Linux Primer: A Practical Real-World Approach*. Prentice Hall, 2006. ISBN: 0131679848.

[Han+03] U. Hansmann u. a. *Pervasive Computing: The Mobile World*. Springer Professional Computing. Springer Verlag, 2003. ISBN: 9783540002185.

[Hei09] Ulrich Heinkel. *Das V-Modell im Systementwurf*. Vorlesungsskript EDA-Tools. 2009.

[Hip11] Hippie2000. *Microvoip isdn top.bit - Fritz!Box, Project Website*. Aug. 2011. URL: `http://www.wehavemorefun.de/fritzbox/Microvoip_isdn_top.bit` (besucht am 11. 02. 2013).

[HM04] Göran Herrmann und Dietmar Müller. *ASIC - Entwurf und Test [Taschenbuch]*. Hanser Fachbuchverlag; Auflage: 1, 2004. ISBN: 3446217096.

[HP08] Jameel Hussein und Rish Patel. *Low-Profile In-System Programming Using XCF32P Platform Flash PROMs*. Application Note 975. Xilinx Inc., Mai 2008.

[HT10] Christian Haubelt und Jürgen Teich. *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. eXamen.press. Berlin, Heidelberg: Springer Verlag, 2010. ISBN: 978-3-642-05355-9.

[HU12] Gareth Halfacree und Eben Upton. *Raspberry Pi User Guide*. John Wiley & Sons, Sep. 2012. ISBN: 978-1-118-46446-5.

[IAB06] IABG Industrieanlagen-Betriebsgesellschaft mbH. *V-Modell XT*. Techn. Ber. IABG Industrieanlagen-Betriebsgesellschaft mbH, 2006.

[Jon06] M. Tim Jones. *Inside the Linux boot process*. White Paper. Emulex Corp., Mai 2006.

[Jur13] Heike Jurzik. *Debian GNU/Linux – Das umfassende Handbuch*. 5. Aufl. Galileo Computing, Sep. 2013. ISBN: 978-3-8362-2661-5.

[Kam09] Andreas Kamleiter. „Embedded Linux Build Systeme". Studienarbeit. Hochschule Augsburg, 2009.

[KE12] Peter Kaiser und Johannes Ernesti. *Python 3 - Das umfassende Handbuch*. Bd. 3. Galileo Computing, 2012. ISBN: 978-3-8362-1925-9.

[Keg06] Dan Kegel. *Building GCC cross-toolchains*. Dez. 2006. URL: `http://kegel.com/crosstool/` (besucht am 11. 01. 2011).

[Ker13] Galym Kerimbekov. *Debootstrap - Debian Wiki*. Aug. 2013. URL: `https://wiki.debian.org/de/Debootstrap`.

[Ket09] Boris Kettelhoit. „Architektur und Entwurf dynamisch rekonfigurierbarer FPGA-Systeme". Dissertation. Universität Paderborn, 2009.

[KH09] Andreas Klinger und Martina Hafner. „Embedded-Linux-Systeme mit Buildroot erstellen". In: *Elektronik Praxis* (2009), S. 1–9.

[Kha05] R.S. Khandpur. *Printed Circuit Boards*. Tata McGraw-Hill, 2005. ISBN: 9780070588141.

[Khu01] Arthur Khu. *Xilinx FPGA Configuration Data Compression and Decompression*. White Paper 1.0. Xilinx Inc., Sep. 2001.

[KKH11] Daniel Kriesten, Sebastian Kratzert und Ulrich Heinkel. „Embedded Linux Distributionen – Das Ende der schwarzen Magie?" In: *Chemnitzer Linux-Tage 2011 – Tagungsband –*. Team der Chemnitzer Linux-Tage. Universitätsverlag Chemnitz, 2011, S. 21–28. ISBN: 978-3-941003-29-3.

[KL01] Greg Kroah-Hartman und Linux Hotplug Project. *Linux Hotplugging*. Jan. 2001. URL: `http://linux-hotplug.sourceforge.net/` (besucht am 01. 09. 2013).

[Kno13] Klaus Knopper. „Frust-Pakete: Unbeachtete Bugreports“. In: *Linux Magazin* 3 (März 2013), S. 34–35.

[Koc02] Andreas Koch. „Architectures and Tools for Heterogeneous Reconfigurable Systems“. In: *Proceedings of the IEEE Workshop on Heterogeneous Reconfigurable Systems-on-Chip*. 2002.

[Koc04] Andreas Koch. „Advances in Adaptive Computer Technology“. Habilitation. Technische Universität Braunschweig, 2004.

[Koc09] Hans-Jürgen Koch. *The Userspace I/O HOWTO*. Juli 2009. URL: `http://www.kernel.org/doc/htmldocs/uio-howto.html` (besucht am 30. 08. 2013).

[KPH10] Daniel Kriesten, Volker Pankalla und Ulrich Heinkel. „An Application Example of a Run-Time Reconfigurable Embedded System“. In: *2010 International Conference on Reconfigurable Computing and FPGAs*. Hrsg. von Viktor Prasanna, Jürgen Becker und René Cumplido. IEEE, Dez. 2010, S. 97–102. ISBN: 978-1-4244-9523-8.

[KPR04] H Kalte, M. Porrmann und U. Rückert. „Leistungsbewertung unterschiedlicher Einbettungsvarianten dynamisch rekonfigurierbarer Hardware“. In: *ARCS Workshops*. 2004, S. 235–244. ISBN: 3-88579-370-9.

[KQ06] Eva-Katharina Kunst und Jürgen Quade. *Linux-Treiber entwickeln*. 2. Aufl. dpunkt.verlag, 2006. ISBN: 3898643921.

[KR08] J. Ryan Kenny und David Rupe. *FPGA Run-Time Reconfiguration: Two Approaches*. White Paper 01055. Altera Corporation, März 2008.

[Kri+06] Daniel Kriesten u. a. „Generalisierte Plattform zur Sensordatenverarbeitung“. In: *DASS 2006 – Dresdner Arbeitstagung Schaltungs- und Systementwurf: Tagungsband*. Hrsg. von Günter Elst. Dresden, Mai 2006, S. 79–84.

[Kri+12] Daniel Kriesten u. a. „Vorhersage- und Optimierungsmethoden für die Reichweite von Elektromobilen am Beispiel eines E-Bikes“. In: *DASS 2012 – Dresdner Arbeitstagung Schaltungs- und Systementwurf: Tagungsband*. Hrsg. von Peter Schneider und Thomas Klotz. Dresden: FRAUNHOFER VERLAG, Mai 2012, S. 118–123. ISBN: 978-3-8396-0404-5.

[Kri11] Daniel Kriesten. *Entwicklung einer zentralen, universell einsetzbaren Steuerplattform mit standardisierten Schnittstellen zur Verarbeitung von Sensordaten, sowie eine dazu gehörende flexible Lösung zur Anbindung vorhandener Sensoren an die Plattform, Kurztitel: Genera*. Chemnitz: Technische Informationsbibliothek u. Universitätsbibliothek, 2011.

[Kri12] Daniel Kriesten. *Software Environments of Smartphone Applications*. Vorlesungsskript SESA. 2012.

[Kro06] Greg Kroah-Hartman. *Linux Kernel in a Nutshell: Linux 2.6 Kernel in Detail (In a Nutshell (O'Reilly))*. O'Reilly Media, 2006. ISBN: 0596100795.

[LB08] Grant Likely und Josh Boyer. „A Symphony of Flavours: Using the device tree to describe embedded hardware". In: *Ottawa Linux Symposium*. Hrsg. von Andrew J. Hutton und C. Craig Ross. Juli 2008, S. 27–37.

[LD94] Patrick Lysaght und John Dunlop. „Dynamic reconfiguration of FPGAs". In: *Selected papers from the Oxford 1993 international workshop on field programmable logic and applications on More FPGAs*. Oxford, United Kingdom: Abingdon EE & CS Books, 1994, S. 82–94. ISBN: 0-9518453-1-4.

[Len13] Ulrich Lentz. „Nutzung eines Raspberry Pi mit Linux-OS als Datenlogger für Pedelecs". Bachelorarbeit. Technische Universität Chemnitz, Juni 2013.

[Ler04] Sancho Lerena. *Pandora Flexible Monitoring System – Project Website*. Okt. 2004. URL: `http://pandorafms.com/` (besucht am 02. 12. 2013).

[LGS13] Wolfram Luithardt, Daniel Gachet und Jean-Roland Schuler. „Zuverlässigkeit von GNU/Linux-basierten eingebetteten Systemen". In: *Chemnitzer Linux-Tage 2013 – Tagungsband –*. Team der Chemnitzer Linux-Tage. Universitätsverlag Chemnitz, März 2013, S. 103–110.

[Lik08] Grant Likely. *Using the Device Tree to Describe Embedded Hardware*. Hearing. Nov. 2008.

[Lin] Linux Foundation. *EGLIBC Project Website*. URL: `http://www.eglibc.org/home` (besucht am 30. 05. 2013).

[LM10] Rob Landley und Mark Miller. *Developing for non-X86 targets using QEMU*. Hearing. 2010.

[LP07] Enno Lübbers und Marco Platzner. „ReconOS: An RTOS Supporting Hard-and Software Threads“. In: *International Conference on Field Programmable Logic and Applications (FPL), 2007*. 2007, S. 441–446.

[LP10] Enno Lübbers und Marco Platzner. *Dynamically Reconfigurable Systems*. Hrsg. von Marco Platzner, Jürgen Teich und Norbert Wehn. Dordrecht: Springer Verlag, 2010, S. 269–290. ISBN: 978-90-481-3484-7.

[LSC96] Wayne Luk, Nabeel Shirazi und Peter Y. K. Cheung. „Modelling and optimising run-time reconfigurable systems“. In: *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Comput. Soc. Press, 1996, S. 167–176. ISBN: 0-8186-7548-9.

[LV10] Florian Langenscheidt und Bernd Venohr. *Lexikon der deutschen Weltmarktführer: Die Königsklasse deutscher Unternehmen in Wort und Bild*. Köln: Deutsche Standards Editionen, 2010. ISBN: 978-3-86936-221-2.

[Mac07] Bryce Mackin. *Accelerating High-Performance Computing With FPGAs*. White Paper 01029. Altera Corporation, 2007.

[Mar08] Peter Marwedel. *Eingebettete Systeme*. eXamen.press. Berlin, Heidelberg: Springer Verlag, 2008. ISBN: 978-3-540-34048-5.

[Mar09] Erik Markert. „High-Level-Entwurf von Mikrosystemen“. Dissertation. Technische Universität Chemnitz, 2009.

[McD08] EJ McDonald. „Runtime FPGA partial reconfiguration“. In: *Aerospace Conference, 2008 IEEE*. 2008, S. 1–7. ISBN: 1424414881.

[Mei10] A. Meisel. *Design Flow für IP basierte, dynamisch rekonfigurierbare, eingebettete Systeme*. Univ.-Verl., 2010. ISBN: 978-3-941003-15-6.

[Mes+03] D. Mesquita u. a. „Remote and partial reconfiguration of FPGAs: tools and trends“. In: *Proceedings of the Parallel and Distributed Processing Symposium, 2003*. Apr. 2003.

[Mic] Microsemi SoC Products Group. *Microsemi Programmable Logic Devices Homepage*. URL: `http://www.actel.com/products/devices.aspx` (besucht am 13. 02. 2013).

[Mor05] Casey Justin Morford. „BitMaT - Bitstream Manipulation Tool for Xilinx FPGAs“. PhD Thesis. Bradley Department of Electrical und Computer Engineering Blacksburg, Virginia, Dez. 2005.

[Mor11] Yann E. Morin. *crosstool-NG*. Englisch. Jan. 2011. URL: `http://ymorin.is-a-geek.org/projects/crosstool` (besucht am 11. 01. 2011).

[Mül90] Karl H. Müller. *Elektronische Schaltungen und Systeme. Simulieren, analysieren, optimieren mit SPICE*. Vogel Verlag, 1990. ISBN: 3-8023-0292-3.

[Nag05] William Nagel. *Subversion Version Control: Using the Subversion Version Control System in Development Projects*. Prentice Hall, 2005. ISBN: 0131855182.

[NM09] G. Nicolescu und P.J. Mosterman. *Model-Based Design for Embedded Systems*. Computational Analysis, Synthesis and Design of Dynamic Models Series. CRC Press, 2009. ISBN: 9781420067842.

[ohn11] ohne Verfasser. *Crossdev*. Englisch. Jan. 2011. URL: `http://en.gentoo-wiki.com/wiki/Crossdev` (besucht am 11. 01. 2011).

[Ope09] OpenWrt Team. *OpenWrt Projekt-Homepage*. Nov. 2009. URL: `http://openwrt.org/`.

[Org05] Mario Orgis. „Untersuchung und Implementierung dynamisch partieller Rekonfiguration auf Xilinx FPGA unter Verwendung von Standardentwurfswerkzeugen“. Diplomarbeit. Technische Universität Chemnitz, 2005.

[OSu09] Bryan O’Sullivan. *Mercurial: The Definitive Guide*. O’Reilly Media, 2009. ISBN: 0596800673.

[Pan10] Volker Pankalla. „Untersuchung und Implementierung von Rekonfigurationsstrategien in hybriden Hardware-Systemen“. Diplomarbeit. Technische Universität Chemnitz, Aug. 2010.

[PCK08] Jung Choon Park, Yong Hoon Choi und Tae ho Kim. „Domain Specific Code Generation For Linux Device Driver“. In: *10th International Conference on Advanced Communication Technology, 2008*. Bd. 1. IEEE, Feb. 2008, S. 101–104. ISBN: 978-89-5519-135-6.

[Pen11] Pengutronix e.K. *OSELAS.Toolchain()*. Jan. 2011. URL: `http://www.pengutronix.de/oselas/toolchain/index_de.html` (besucht am 15. 01. 2011).

[Pen12] Pengutronix e.K. *How to become a PTXdist Guru*. White Paper. Pengutronix e.K., Apr. 2012.

[PK06] KY Park und Hyuk Kim. *Remote FPGA Reconfiguration Using MicroBlaze or PowerPC Processors*. Application Note 441. Xilinx Inc., Sep. 2006.

[PK07] Hendrik Post und Wolfgang Küchlin. „Integrated Static Analysis for Linux Device Driver Verification“. In: *Integrated Formal Methods*. Hrsg. von Jim Davies und Jeremy Gibbons. Bd. 4591. Lecture Notes in Computer Science. Springer Verlag, 2007, S. 518–537. ISBN: 978-3-540-73209-9.

[Pre] Tom Preston-Werner. *Semantic Versioning*. URL: `http://semver.org/` (besucht am 03. 10. 2013).

[Pro] Processors, Project Website. *Processors, Project Website*. URL: `http://gaisler.com/index.php/products/processors` (besucht am 09. 05. 2013).

[PSK09] Hendrik Post, Carsten Sinz und Wolfgang Küchlin. „Towards Automatic Software Model Checking of Thousands of Linux Modules – a Case Study with Avinux“. In: *Software Testing, Verification & Reliability* 19.2 (Juni 2009), S. 155–172. ISSN: 0960-0833.

[PTW10] Marco Platzner, Jürgen Teich und Norbert Wehn. *Dynamically Reconfigurable Systems - Architectures, Design Methods and Applications*. Hrsg. von Marco Platzner, Jürgen Teich und Norbert Wehn. Dordrecht: Springer Verlag, 2010. ISBN: 978-90-481-3484-7.

[Put09] Wolfram Putzke-Röming. „MORPHEUS Architecture Overview“. In: *Dynamic System Reconfiguration in Heterogeneous Platforms*. Hrsg. von Nikolaos S. Voros, Alberto Rosti und Michael Hübner. Bd. 40. Lecture Notes in Electrical Engineering. Dordrecht: Springer Verlag, 2009, S. 31–37. ISBN: 978-90-481-2426-8.

[Put10] Marcel Putsche. „Prototypischer Entwurf eines Überwachungssystems für Faserkunststoffverbunde“. Diplomarbeit. Technische Universität Chemnitz, Feb. 2010.

[Reb08] René Rebe. „Embedded-Linux-Distributionen vorgestellt“. In: *Linux Magazin 3* (März 2008).

[RFH07] M. Rößler, A. Froß und U. Heinkel. „Car2Roadside Kommunikation auf Basis von 802.15.4a“. In: *9. Wireless Technology Kongress*. Stuttgart, Sep. 2007, S. 247–255. ISBN: 978-3-89838-084-3.

[RM98] B. Radunovi'c und V. Milutinovi'c. *Field-Programmable Logic and Applications From FPGAs to Computing Paradigm*. Hrsg. von Reiner W. Hartenstein und Andres Keevallik. Bd. 1482. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Verlag, 1998, S. 376–385. ISBN: 3-540-64948-4.

[Rop78] G. Ropohl. *Allgemeine Technologie : eine Systemtheorie der Technik*. Univ.-Verlag Karlsruhe, 1978. ISBN: 9783866443747.

[RT99] Paul Raines und Jeff Tranter. *Tcl/Tk in a Nutshell*. O'Reilly Media Inc., März 1999. ISBN: 978-1-56592-433-8.

[San+11] M.D. Santambrogio u. a. „ReBit: A Tool to Manage and Analyse FPGA-Based Reconfigurable Systems“. In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. Mai 2011, S. 220–227.

[SB10] Steven She und Thorsten Berger. *Formal Semantics of the Kconfig Language*. Techn. Ber. Technical Note. Electrical und Computer Engineering, University of Waterloo, Canada, Jan. 2010.

[SGD09] Joachim Schröder, Tilo Gockel und Rüdiger Dillmann. *Embedded Linux: Das Praxisbuch (X.systems.press) (German Edition)*. Springer Verlag, 2009. ISBN: 3540786198.

[SH10] Christoph Stückjürgen und Gernot Hillier. „Linux inside“. In: *c’t Magazin für Computertechnik* 19 (Aug. 2010), S. 164–169.

[She+10] Steven She u. a. „Variability Model of the Linux Kernel“. In: *Fourth International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010)*. Linz, Austria, 2010.

[Sil09] SiliconBlue Technologies Corporation. *SiliconBlue Homepage*. Nov. 2009. URL: `http://www.siliconbluetech.com/` (besucht am 13. 02. 2013).

[Sin+10] Julio Sincero u. a. „Facing the Linux 8000 Feature Nightmare“. In: *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*. Hrsg. von ACM SIGOPS. Paris, France, 2010.

[Sin09] Surbhi Singhal. *A Survey of Dynamically Reconfigurable FPGA Devices*. Hearing. 2009.

[SKH09] Jörg Schneider, Daniel Kriesten und Ulrich Heinkel. „Fehlertolerante Firmware-Updates FPGA-basierter Funkprototypensysteme mit Linux-OS“. In: *9. Chemnitzer Fachtagung Mikrosystemtechnik*. Technische Universität Chemnitz. Chemnitz, 2009, S. 45–50. ISBN: 978-3-00-029135-7.

[SN92] Frank Schönthaler und Tibor Németh. *Software-Entwicklungswerkzeuge – Methodische Grundlagen*. 2. Auflage. Leitfäden der angewandten Informatik. Teubner, 1992. ISBN: 978-3-519-12417-7.

[So07] Hayden Kwok-Hay So. *BORPH: An Operating System for FPGA-based Reconfigurable Computers*. University of California, Berkeley, 2007. ISBN: 9780549529576.

[Spi+11] Andreas Spillner u. a. *Auswertung der Umfrage: Softwaretest in der Praxis*. Nov. 2011. URL: `http://www.heise.de/developer/artikel/Auswertung-der-Umfrage-Softwaretest-in-der-Praxis-1369290.html` (besucht am 24. 08. 2013).

[SR09] Pradyumna Sampath und Rachana Rao. „Efficient embedded software development using QEMU“. In: *11th Real Time Linux Workshop*. 2009.

[SR13] Christian Siemers und Bernd Rosenlechner. *Handbuch Embedded Systems Engineering*. Hrsg. von Christian Siemers. Bd. 0.61. Online, 2013.

[Sta86] Richard M. Stallman. „What is the Free Software Foundation?“ In: *GNU's Bulletin* 1.1 (Feb. 1986). Hrsg. von Jerome E. Puzo, S. 8–9.

[STM10] Lambert M. Surhone, Miriam T. Timpledon und Susan F. Marseken. *OpenEmbedded*. VDM Publishing, 2010. ISBN: 9786132147998.

[Stö07] Florian Stöhr. *Die GNU Autotools: Leitfaden für die Softwaredistribution*. Computer & Literatur. C&L-Verlag, 2007. ISBN: 978-3-936546-48-4.

[Str+09] Jochen Strunk u. a. „Run-Time Reconfiguration for HyperTransport coupled FPGAs using ACCFS“. In: *Proceedings of First International Workshop on HyperTransport Research and Applications*. Hrsg. von Holger Fröning, Mondrian Nüssle und Pedro Javier Garc'ıa Garc'ıa. 2009, S. 54–63. ISBN: 978-3-00-027249-3.

[Str05] Jochen Strunk. „Entwicklung eines Software-Frameworks zur Unterstützung dynamisch rekonfigurierbarer Hardware auf Basis von Virtex-II Pro FPGAs“. Diplomarbeit. Technische Universität Chemnitz, 2005.

[TAK06] J.G. Tong, I. D L Anderson und M. A S Khalid. „Soft-Core Processors for Embedded Systems“. In: *The 18th International Confernece on Microelectronics (ICM), 2006*. 2006, S. 170–173.

[Tan09] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. 3. aktualisierte Auflage. Pearson Studium, 2009. ISBN: 3827373425.

[Tap07] Stephanie Tapp. *Configuring Xilinx FPGAs with SPI Serial Flash*. Application Note 951. Xilinx Inc., Nov. 2007.

[TD01] Linus Torvalds und David Diamond. *Just for Fun – The Story of an Accidental Revolutionary*. 1st ed. New York, NY: HarperBusiness, 2001. ISBN: 0066620724 (hc).

[TD02] Linus Torvalds und David Diamond. *Linus Torvalds: Just For Fun. Wie ein Freak die Computerwelt revolutionierte. Die Biographie des Linux-Erfinders*. ngerman. Titel der amerikanischen Originalauflage: Just for Fun – The Story of an Accidental Revolutionary. HarperCollins, New York, 2001. Aus dem Amerikanischen von Doris Märtin. München: Deutscher Taschenbuch Verlag, 2002. ISBN: 3-423-36299-5.

[Tod+05] T. J. Todman u. a. „Reconfigurable computing: architectures and design methods“. In: *IEEE Proceedings - Computers and Digital Techniques*. Bd. 152. März 2005, S. 193–207.

[Ull12] Christian Ullenboom. *Java 7 - Mehr als eine Insel*. Galileo Computing, 2012. ISBN: 978-3-8362-1507-7.

[Uni98] Unicontrol Systemtechnik GmbH. *UNICONTROL*. 1998. URL: `http://www.unicontrol.de` (besucht am 22. 11. 2013).

[Uns01] Unsigned Integer Ltd. *DistroWatch.com: Put the fun back into computing*. Mai 2001. URL: `http://distrowatch.com/` (besucht am 31. 05. 2013).

[VBT04] Gerd Van den Branden, Geert Braeckman und Abdellah D. Touhafi. *Commercially available Reconfigurable Components*. 2004.

[Viz13] Daniel Manchón Vizuete. *Instant Buildroot*. Packt Publishing, Sep. 2013. ISBN: 978-1-783-289-455.

[VM05] N. Voros und K. Masselos. *System Level Design of Reconfigurable Systems-on-Chip*. Springer Verlag, 2005. ISBN: 9780387261034.

[VRH09] Nikolaos S. Voros, Alberto Rosti und Michael Hübner. *Dynamic System Reconfiguration in Heterogeneous Platforms*. Hrsg. von Nikolaos S. Voros, Alberto Rosti und Michael Hübner. Bd. 40. Lecture Notes in Electrical Engineering. Dordrecht: Springer Verlag, 2009. ISBN: 978-90-481-2426-8.

[Wal05] Herbert Walder. „Operating System Design for Partiallly Reconfigurable Logic Devices“. Dissertation. ETH Zurich, 2005.

[Wan+11] Zhaoguo Wang u. a. „COREMU: A Scalable and Portable Parallel Full-system Emulator“. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP ’11. New York, NY, USA: ACM, 2011, S. 213–222. ISBN: 978-1-4503-0119-0.

[Wan98] M. Wannemacher. *Das FPGA-Kochbuch*. Internat. Thomson Publ., 1998. ISBN: 9783826627125.

[WB04] J.A. Williams und N.W. Bergmann. „Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip“. In: *The International Conference on Engineering of Reconfigurable Systems and Algorithms*. CSREA Press. 2004, S. 163–169.

[Wei10a] Enrico Weigelt. *OSS-QM Project Normalized sourcecode repositories*. Techn. Ber. metux IT service, 2010.

[Wei10b] Enrico Weigelt. *Overview to the OSS-QM project*. Techn. Ber. metux IT service, 2010.

[Wie12] Joachim Wietzke. *Embedded Technologies: Vom Treiber bis zur Grafik-Anbindung*. Springer Verlag, 2012. ISBN: 978-3-642-23996-0.

[Wik] Wikimedia Foundation. *Wikipedia, Project Website*. URL: `http://www.wikipedia.org/` (besucht am 10. 02. 2013).

[Win11] Katrin Windisch. „Untersuchung und Implementierung von Vorhersage- und Optimierungsmethoden für die Reichweite von Elektromobilen am Beispiel des E-Bikes“. Bachelorarbeit. Technische Universität Chemnitz, Sep. 2011.

[WK08] Roy White und Arthur Khu. *Embedded JTAG ACE Player*. Application Note 424. Xilinx Inc., Apr. 2008.

[Wol10] Wayne Wolf. „Picking the right system design methodology for your embedded apps“. In: *Embedded Systems Design* (2010).

[Wol12] Peter Wolf. „Messsystem zur Überwachung von Faserkunststoffverbunden“. Dissertation. Technische Universität Chemnitz, 2012.

[Wro08] Gunnar Wrobel. *Gentoo Linux. Installation - Konfiguration - Administration*. Hrsg. von Markus Wirtz. Open Source Press, 2008. ISBN: 978-3-937514-34-5.

[WWX10] Bei Wang, Bo Wang und Qingqing Xiong. „The comparison of communication methods between user and Kernel space in embedded Linux“. In: *Computational Problem-Solving (ICCP), 2010 International Conference on*. Dez. 2010, S. 234–237. ISBN: 978-1-4244-8654-0.

[Xil05] Xilinx Inc. *Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual*. v1.0. Xilinx Inc. März 2005.

[Xil07a] Xilinx Inc. *Configuration and Readback of Virtex FPGAs Using JTAG Boundary-Scan*. Application Note 139. Xilinx Inc., Feb. 2007.

[Xil07b] Xilinx Inc. *Xilinx In-System Programming Using an Embedded Microcontroller*. Application Note 58. Xilinx Inc., Okt. 2007.

[Xil08a] Xilinx Inc. *Early Access Partial Reconfiguration User Guide*. 1.2. Sep. 2008.

[Xil08b] Xilinx Inc. *MicroBlaze Processor Reference Guide*. Xilinx Inc. 2008.

[Xil08c] Xilinx Inc. *Platform Flash PROM User Guide*. v1.3. Xilinx Inc. 2008.

[Xil08d] Xilinx Inc. *Spartan-3 FPGA Family Advanced Configuration Architecture*. Techn. Ber. Xilinx Inc., Juni 2008.

[Xil08e] Xilinx Inc. *Virtex-4 FPGA Configuration User Guide*. v1.10. Xilinx Inc. Apr. 2008.

[Xil08f] Xilinx Inc. *Virtex-4 FPGA User Guide*. v2.6. Xilinx Inc. 2008.

[Xil08g] Xilinx Inc. *Virtex-5 FPGA Configuration User Guide*. v3.1. Xilinx Inc. 2008.

[Xil08h] Xilinx Inc. *Virtex-5 FPGA User Guide*. v4.2. Xilinx Inc. 2008.

[Xil08i] Xilinx Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*. v5.0. Xilinx Inc. 2008.

[Xil09a] Xilinx Inc. *Xilinx Homepage*. Nov. 2009. URL: `http://www.xilinx.com/`.

[Xil09b] Xilinx Inc. *Xilinx University Program*. Nov. 2009. URL: `http://www.xilinx.com/univ/`.

[Xil11a] Xilinx Inc. *Command Line Tools User Guide*. v13.1. Xilinx Inc. März 2011.

[Xil11b] Xilinx Inc. *ML505/ML506/ML507 Evaluation Platform: User Guide*. v3.1.2. Xilinx Inc. Mai 2011.

[Xil11c] Xilinx Inc. *Spartan-6 FPGA Configuration User Guide*. v2.3. Xilinx Inc. 2011.

[Xil12a] Xilinx Inc. *A Hands-On Guide to Effective Embedded System Design*. Dez. 2012.

[Xil12b] Xilinx Inc. *Partial Reconfiguration of a Processor Peripheral*. Tutorial. Apr. 2012.

[Xil12c] Xilinx Inc. *Partial Reconfiguration Tutorial*. Tutorial. Mai 2012.

[Xil12d] Xilinx Inc. *Partial Reconfiguration User Guide*. v14.4. Xilinx Inc. Dez. 2012.

[Xil12e] Xilinx Inc. *Vivado Design Suite User Guide – Design Flows Overview*. v2012.4. Xilinx Inc. Dez. 2012.

[Xil12f] Xilinx Inc. *Zynq-7000 All Programmable SoC*. v1.4. Xilinx Inc. Nov. 2012.

[Xil13] Xilinx Inc. *7 Series FPGAs Configuration*. v1.7. Xilinx Inc. Okt. 2013.

[Yag+08] Karim Yaghmour u. a. *Building Embedded Linux Systems*. O'Reilly Media, 2008. ISBN: 0596529686.

Abbildungsverzeichnis

1.1 Teilgebiete beim Entwurf eingebetteter hrS mit Linux-BS . . . 30

2.1 Entwicklung eines eingebetteten Systems (Quelle: [HT10]) . . 38
2.2 Angepasstes V-Modell (Quelle: [Hei09]) 38
2.3 Vereinfachter Informationsfluss beim Entwurf eingebetteter Systeme (Quelle: [Mar08]) 39
2.4 Hierarchischer Designflow nach [Wol10] 40
2.5 Generisches Computersystem in Anlehnung an [Cat05] 44
2.6 Prozessorsysteme in eingebetteten Systemen 47
2.7 Vereinfachte Darstellung des Hardware-Entwurfsablaufes . . . 48
2.8 Entwurfsfluss für rekonfigurierbare Systeme 50
2.9 pR-Entwurfsablauf und pR-Design 62
2.10 Schichtenmodelle von Betriebssystemen 65
2.11 Entwurfsfluss bei der Softwareentwicklung 67
2.12 Zusammenfassung der einzelnen Entwurfsabläufe 69

3.1 Systemarchitekturen für hrS 72
3.2 Position und Aufgabe des RM im Datenpfad 77
3.3 Konzepte zum Datenaustausch zwischen Hostprozessor und RM 78
3.4 Beziehung zwischen Hostprozessor, RM und Konfigurationsspeicher 82
3.5 Schematische Darstellung der Aufgaben bei der dR 91

4.1 Das Linux-Schichtenmodell von Kernel und System 97
4.2 Fiktiver Abhängigkeitsbaum einer Anwendung 98
4.3 Vereinfachter Bootprozess 102
4.4 Softwarezyklus im Linux-Umfeld 105
4.5 Grundsätzliche Funktionsweise von Buildsystemen 114

4.6 Patchstrategien 118
4.7 Versionsmanagement der Entwicklungsumgebung mit Git . . 120
4.8 Typische Aufteilung des ROM-Speichers eines eingebetteten Systems 122
4.9 Update-Strategien verschiedener Granularität 123
4.10 Konzepte für die partielle Aktualisierung 124

5.1 Basisidee für die vereinfachte Spezifikation einer Linux-Firmware 138
5.2 Architektur des Synthesewerkzeugs 139
5.3 Finales System 140
5.4 Ablauf und Umsetzung des Syntheseflows bei PTXdist 141

6.1 Genutzte Entwicklungsumgebung 146
6.2 Mobiler Demonstrator „embedded FPGA“ und zugehörige Sensoren 157
6.3 ATNGW100 und Tochterboard 159
6.4 Schematische Darstellung des `siconf`-Treibers 163
6.5 Taktschema im Slave-Serial Modus 165
6.6 Schematische Darstellung des XcConf-Treibers 166
6.7 Das digitale Kombiinstrument 168
6.8 Schematische Darstellungen zur DKI-Box 169
6.9 FPGA-Erweiterungsboard 169
6.10 Zusammenspiel der Module für die RTR beim DKI 173
6.11 Rahmendesign des JTAG-Players 174
6.12 Blockschaltbild des dpR-Designs 176
6.13 Grafische Darstellung im Display des DKIs 178
6.14 Fahrzeugdemonstrator mit Breitbandanbindung 180
6.15 Erweiterungsplatine mit Echtzeituhr (Blockschaltbild) 183
6.16 Blockschaltbild der Messwert-Platine 184
6.17 Stromaufnahme des Raspberry Pi (Modell B) 188
6.18 Prinzipieller Aufbau des LTE-Überwachungssystems 190
6.19 LTE-Messsystem 191

7.1 Konzept für die zentrale Datenhaltung 200
7.2 Entity-Relationship-Modell der Datenbank 203
7.3 Flaches Datenmodell 206

7.4 Konzept der Messdatenübertragung 208
7.5 Schema des implementierten Systems 209
7.6 Sensorcom Konzept 211

A.1 Konfigurationsschichten (Quelle: [CH99]) 225

B.1 Schematische Darstellung der Entwurfsebenen 228

C.1 Versionsmanagement der Entwicklungsumgebung mit Git . . 232

E.1 Implementierung mit einem Werkzeug 235
E.2 Implementierung mit einem Werkzeug je Ausgangsformat . . 236
E.3 Django basiertes Konzept der High-Level Konfiguration 237

F.1 GPSV-Konzept 242

G.1 XSVF Programmablauf (vgl. [Xil07b]) 244
G.2 Exemplarische Darstellung eines Kernel-Treibers 245

Tabellenverzeichnis

2.1 Geschwindigkeit der Konfigurationsschnittstellen 52
2.2 Abschätzung der Bitstromgrößen 53
2.3 Zusammenfassende Darstellung der RTR mit FPGAs 60
2.4 Schnittstellen für die (Re-) Konfiguration 61

4.1 Schnittstellen zur Hardware 101
4.2 Systemgröße eines Raspberry Pi bei verschiedenen Ansätzen . 132

6.1 JTAG Modus – User Space Software 162
6.2 JTAG Modus – Kernel Space Software 164
6.3 Slave-Serial Modus, Kernel-Treiber 166
6.4 Übertragene Daten in der Beispielanwendung 179
6.5 Werkseitig eingestellte Werte für Frequenzen und Spannungen 186
6.6 Erprobte Werte für Frequenzen und Spannungen 187

A.1 SPLD-Typen 224
A.2 Klassifikationsmerkmale für FPGAs 226

G.1 Übersicht der XSVF Befehle 243

Thesen

1. Jedes der betrachteten Forschungsfelder – eingebettete Systeme, Betriebssysteme, FPGAs und run-time Reconfiguration – besitzt einen domänenspezifischen Entwurfsablauf. Eine Verbindung dieser Abläufe zu einem umfassenden Vorgehensmodell für heterogene rekonfigurierbare Systeme ist möglich.

2. Es existieren spezielle Architekturen für die Kommunikation, den Datenaustausch und die (Re-)Konfiguration zwischen Hostprozessor und rekonfigurierbarem Modul in heterogenen rekonfigurierbaren Systemen.

3. Ein umfassendes Verständnis der Architekturen für Kommunikation und Konfiguration in heterogenen rekonfigurierbaren Systemen erlaubt die angepasste Auslegung der Systemarchitektur und dadurch eine deutliche Verringerung der Komplexität bei Leiterplatten-, Hardware- und Softwareentwurf.

4. Nicht alle Kommunikations- und Konfigurationsarchitekturen eignen sich für eingebettete heterogene rekonfigurierbare Systeme. Es besteht ein direkter Zusammenhang zwischen Konfigurationsstrategie und Systemarchitektur.

5. Für heterogene rekonfigurierbare Systeme lassen sich aus Sicht der Betriebssystemarchitektur vier Konfigurationszeitpunkte definieren: *initial configuration*, *early access*, *early OS access* und *late access*.

6. Eine systematische Darstellung der etablierten Zusammenhänge und Abläufe beim Software-, Paket- und Versionsmanagement im Linux-Umfeld ist möglich. Das umfassende Verständnis der Zusammenhänge ermöglicht die Konzeption und Implementierung generalisierter Update- und Managementstrategien.

7. Mit der Entscheidung für Linux bindet sich der Entwickler an die damit verbundene Systemarchitektur, an die systemspezifischen Philosophien, Prozesse und Gepflogenheiten sowie an Standards bei Werkzeugen, Softwareentwurf, Paket-, Versions- und Qualitätsmanagement.
8. Die Kommunikations- und Konfigurationsarchitekturen heterogener rekonfigurierbarer Systeme und das Konzept der vier Konfigurationszeitpunkte kann mit den im Linux-Betriebssystem üblichen Verfahren implementiert werden.
9. Buildsysteme empfehlen sich für die Erstellung von Linux-Systemen für spezielle Anwendungen. Sie erlauben das Erstellen von distributionsgleichen, individualisierten Linux-Systemen.
10. Die Umsetzung eines eingebetteten Systems mit Linux-Betriebssystem erfordert immer ein individuelles Vorgehen bei Spezifikation und Entwurf. Bei der Umsetzung angepasster Linux-Systeme ist keine wesentliche Reduktion des Entwurfsaufwands durch die Nutzung von Distributionen zu erkennen.
11. Die High-Level Spezifikation von Linux-Systemen ist möglich. Die notwendigen Konzepte sowie ein geeignetes Werkzeug werden dargestellt.
12. Die Arbeit wendet die in ihr vorgestellten Methoden und Konzepte auf eine modulare Plattform zur Erfassung und Verarbeitung von Sensordaten an und verifiziert so die praktische Umsetzbarkeit.
13. Relationale Datenbanken eignen sich für die langfristige Speicherung von Messwerten. Die Implementierung eines Modells für die vollständige Reproduktion eines Messlaufes ist möglich und wird gezeigt.
14. Aufwand und Geschwindigkeit für das Aufzeichnen und Auslesen von Messwerten in eine Datenbank profitieren von gegensätzlichen Datenmodellen.