RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Alexander Sinsel

Linuxportierung auf einen eingebetteten PowerPC 405 zur Steuerung eines neuronalen Netzwerks

Diplomarbeit

HD-KIP-03-14

KIRCHHOFF-INSTITUT FÜR PHYSIK

Fakultät für Physik und Astronomie
Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von
Alexander Sinsel
aus Frankfurt am Main
2003

Linuxportierung auf einen eingebetteten PowerPC 405 zur Steuerung eines neuronalen Netzwerks

Die Diplomarbeit wurde von Alexander Sinsel ausgeführt am
Kirchhoff-Institut für Physik
unter der Betreuung von
Herrn Prof. Dr. Karlheinz Meier

Inhaltsverzeichnis

Einleitung

Unter einem eingebetteten System versteht man im allgemeinen ein aus Hardware und Software bestehendes Computersystem, das zum einen selbst Bestandteil eines anderen Geräts ist und zum anderen einem bestimmten Zweck dient. Der vielfältige Einsatzbereich eingebetteter Systeme erstreckt sich von Steuerungssystemen in Industrieanlagen bis hin zu gewöhnlichen Haushaltsgeräten, wie Videorecordern, Kaffeemaschinen, Mikrowellen oder Waschmaschinen. Schon längst werden weitaus mehr Mikroprozessoren für *eingebettete Systeme* (*embedded systems*) als für Desktop-Computer produziert [Dev01]. Zwangsläufig hat dieser Sachverhalt schnell auch die Aufmerksamkeit der Hersteller von Betriebssystemen auf sich gezogen. Wurden eingebettete Systeme anfänglich noch mit einer eigens dafür entworfenen Betriebssoftware (*home-grown operating systems*) ausgeliefert, so sind derartige Eigenentwürfe heute vielfach durch marktgängige eingebettete Betriebssysteme verdrängt worden. Die bis dahin auf Personalcomputer und Workstations spezialisierte Firma Microsoft kam beispielsweise 1996 mit dem eingebetteten Betriebssystem Windows CE auf den Markt. Inzwischen gibt es eine Vielzahl solcher eingebetteten Betriebssysteme wie VxWorks, LynxOS, BSD oder QNX, um nur einige Beispiele zu nennen.
Linux wurde ausgesprochen spät erstmals im Jahre 1999 als eingebettetes Betriebssystem öffentlich in Erwägung gezogen, als verschiedene freie und erste kommerzielle *embedded Linux* Projekte im Internet erschienen. Von da an zeichnete sich jedoch eine überraschend schnelle Entwicklung ab [Dev03]. Mit einem geschätzen Anteil zu über 25 Prozent ist Linux auf 32-Bit-Prozessoren mittlerweile das meistverwendete eingebettete Betriebssystem.

Als offenes und kostenloses Unixderivat wurde Linux in seinen Anfängen 1991 von Linus Torvalds für einen PC mit dem Intel 80386-Prozessor entworfen. Verbreitung fand es in den folgenden Jahren jedoch vorwiegend durch seinen Einsatz auf Servern und Workstations in Netzwerken. Der offene Quellcode ermöglicht es Programmierern aus aller Welt, an der Beseitigung von Sicherheitslücken, Fehlerbehebung, Verbesserung und Erweiterung von Linux mitzuwirken. Dies ist gewiß ein maßgeblicher Grund für die Sicherheit, Stabilität und rasante Weiterentwicklung von Linux. Die unerwartet schnelle Festigung auf dem Wachstumsmarkt eingebetteter Systeme verdeutlicht die Flexibilität und Portabilität dieses Betriebssystems. Linux wurde zum größten Teil in der Sprache C geschrieben und mit der *GNU Toolchain* der Free Software Foundation [FSF] entwickelt. Diese Sammlung von Entwicklungswerkzeugen bietet die Unterstützung für eine breite Palette von verschiedenen Prozessortypen, worin die hohe Portabilität von Linux letztendlich begründet ist. Die Entscheidungskriterien, welche zur Wahl von embedded Linux

führten, untersuchte eine im Mai dieses Jahres abgeschlossene Internetumfrage von LinuxDevices.com's [Dev03]. An erster Stelle wurden die Entwicklungswerkzeuge der GNU Toolchain genannt. Sie stellen die entscheidende Voraussetzung bei der Portierung von Linux dar. Was man im Englischen mit „embedding Linux" bezeichnet, bedeutet in erster Linie eine Portierung, nämlich die Portierung von Linux auf ein eingebettetes System mit einem speziellen Mikroprozessor.
Aus dem Quellcode des Betriebssystems und der benötigten Anwenderprogramme muß der Maschinencode für die Zielplattform generiert werden. Darum ist die Unterstützung der auf der Zielplattform vorliegenden Befehlssatzarchitektur durch die Entwicklungswerkzeuge die wesentliche Voraussetzung für eine solche Portierung.
Dennoch umfaßt die Portierung eines Betriebssystems gelegentlich mehr als die reine Codeübersetzung durch geeignete Kompilierwerkzeuge. Teile des Betriebssystems, die unmittelbar auf der Hardware operieren, liegen in einem hardwarespezifischen Quellcode vor. Ein derartiger Code wird auf anderen Plattformen mitunter unbrauchbar und muß gegebenenfalls neu durchdacht werden.
Neben der Portierung auf eine andere Prozessorarchitektur ergibt sich aus der speziellen Peripherie eingebetteter Systeme oftmals die Notwendigkeit weiterer Anpassungen in bezug auf das Dateisystem und die Benutzerinteraktivität. Zwar ist die Peripherie eingebetteter Systeme typischerweise mit sehr wenigen Geräten ausgestattet, gerade deswegen erfordern Dateisystem und Benutzerschnittstellen jedoch eine spezielle Implementierung mit individueller Treibersoftware. Oftmals verlangt schon der Systemstart spezielle Konzepte.

Bei dem übergeordneten System, das dieser Arbeit zugrundeliegt, handelt es sich um ein skalierbares, verteiltes neuronales Netz auf der Basis integrierter Schaltkreise (ASICs[1]). Neuronale Netze folgen dem biologischen Vorbild des Nervensystems. Sie sind im Gegensatz zum klassischen Computer lernfähig und zeichnen sich durch hochgradig parallele und lokale Verarbeitungsstrukturen aus. Die vorliegende Konzeption überträgt die systemimmanente Parallelität des Netzwerkes auf dessen Lernprozeß zugunsten einer größeren Skalierbarkeit. Dabei beruht das Lernen auf dem Prinzip der lokalen Optimierung innerhalb eines verteilten Systems. Dessen Konstituenten sind eingebettete Systeme mit je einem Netzwerkchip und einem IBM PowerPC 405 Prozessor. Der Gegenstand der vorliegenden Arbeit ist es, diese eingebetteten Systeme mit Linux in Betrieb zu nehmen.

Anhand der zugrundeliegenden Hardware ist es möglich, alle oben geschilderten Probleme und deren Lösungen exemplarisch aufzuzeigen. Das nötige Grundlagenwissen über das Betriebssystem Linux und dessen Portierung, sowie der Umgang mit einer speziellen Peripherie und die daraus folgenden Problematik des Systemstarts werden im Rahmen dieser Arbeit erläutert.

[1] **A**pplication **S**pecific **I**ntegrated **C**ircuits

Kapitel 1

Konzeption eines verteilten neuronalen Netzwerkes

Wie die Turingmaschine oder die rekursiven Funktionen, so sind auch neuronale Netze ein alternatives Berechenbarkeitsmodell zum klassischen Computer [Roj96]. Im Unterschied zu anderen Berechenbarkeitsmodellen sind neuronale Netze jedoch am Vorbild biologischer Nervensysteme orientiert und wie diese lernfähig. Neuronale Netze bestehen graphentheoretisch aus einer Menge von Knoten und gerichteten Kanten, wobei im Rückgriff auf die biologische Nomenklatur die Knoten als Neuronen und die Kanten als Synapsen bezeichnet werden. Die Verbindungsstruktur der Neuronen durch die Synapsen, man spricht von der Netzwerktopologie, kann durch einen gerichteten Graphen beschrieben werden. Es gibt zwei ausgezeichnete Teilmengen dieser Neuronen, die üblicherweise als Eingangs- und Ausgangsneuronen bezeichnet werden. Von den Eingangsneuronen bis zu den Ausgangsneuronen erfolgt eine Informationsübertragung. Jede Information wird als Signal von einem Neuron durch die auslaufenden Synapsen an die jeweils folgenden Neuronen weitergeleitet. Dabei gibt es erregende und hemmende Synapsen. Das Ausgangssignal einzelner Neuronen ist üblicherweise eine Schwellenwertfunktion über der Summe der Eingangssignale. Somit implementiert das gesamte Netzwerk eine Transferfunktion über den Signalen der Eingangsneuronen. Das hiermit beschriebene erste, einfache Modell eines neuronalen Netzes stammt von *McCulloch* und *Pitts* [MP43] aus dem Jahre 1943. Es hat jedoch einen entscheidenden Nachteil. Um unterschiedliche Funktionen zu implementieren oder diese zu optimieren, muß die Netzwerktopologie geändert werden. Für einen Optimierungsprozeß wie das Lernen bedeutet das eine ständige Änderung der Netzwerkstruktur. Ein solches Vorgehen ist in jedem Falle umständlich und für Hardwareimplementierungen gänzlich ungeeignet. Überdies entspricht es nicht dem biologischen Paradigma des Nervensystems. Dessen Lernprozesse basieren in erster Linie auf einer Änderung der synaptischen Übertragungseigenschaften. In Anlehnung an die biologischen Erkenntnisse verwenden heutige Modelle künstlicher neuronaler Netze überwiegend gewichtete Synapsen. Bei der synaptischen Übertragung wird das Signal mit dem jeweiligen Synapsengewicht multipliziert. Hemmende Synapsen bekommen einen negativen Gewichtswert. Ein solches neuronales Netz wird als *Perzeptron* bezeichnet und wurde 1958 erstmals von Rosenblatt [Ros58] vorgeschlagen.

Beim Perzeptron besteht das Lernen im Auffinden geeigneter Synapsengewichte. Zu diesem Zwecke gibt es verschiedene Lernalgorithmen. Darunter haben die genetischen Lernalgorithmen ihr Vorbild wiederum in der Natur, die im Laufe der Evolution das Vernetzungsmuster biologischer neuronaler Netze entwickelt und verbessert hat. Ein genetischer Algorithmus erzeugt mehrere sogenannte Individuen als Gewichteverteilung über alle Synapsen und faßt diese zu einer Population zusammen. Anhand von ausgewählten Funktionsargumenten, welche den Eingangsneuronen als Lernmuster angelegt werden, kann den einzelnen Individuen ein Fehler zugeordnet werden, der sich aus der Abweichung der an den Ausgangsneuronen gemessenen Werten vom tatsächlichen Wert der zu erlernenden Funktion ergibt. Durch Mutation, Kreuzung und Selektion der besten Individuen entsteht eine neue Generation. Im Laufe einer Evolution, wie man die Abfolge solcher Generationen bezeichnet, sollte sich der Fehler des besten Individuums minimieren. Man spricht hierbei vom *Training* des Netzwerkes. Im Idealfall, nämlich dann, wenn das Netzwerk die ausgewählte Funktion erlernt hat, stimmen die Funktionswerte auch für Argumente, welche nicht zu den vorgelegten Lernmustern gehören. Das neuronale Netz hat in diesem Falle eine Generalisierung vollzogen, eine Fähigkeit, über die ein klassischer Computer nicht verfügt. Dadurch ist ein neuronales Netz in der Lage, Algorithmen selbständig aufzufinden, wohingegen ein Computer solche nur ausführen kann. Neuronale Netze besitzen demnach Abstraktionsfähigkeit.

Zum einen erhofft man sich durch Untersuchungen an künstlichen neuronalen Netzen, Aufschlüsse über biologische Systeme zu gewinnen, zum anderen versprechen technische Anwendungen vielerlei praktischen Nutzen. Die Einsatzgebiete künstlicher neuronaler Netze sind Probleme, bei denen der Algorithmus nicht bekannt oder in einer herkömmlicher Darstellung zu kompliziert ist. Typischerweise handelt es sich dabei oftmals um Formen der Interaktivität zwischen Mensch und Maschine, wie beispielsweise Sprachanalyse, Handschriften- oder Gesichtserkennung.

Die Menge der von einem Netzwerk mit einer festgelegten Genauigkeit erlernbaren Funktionen ist unter anderem durch die Netzwerktopologie, insbesondere durch die Netzwerkgröße begrenzt. Die Skalierbarkeit des Netzwerkes ist deshalb ein wünschenswertes Leistungsmerkmal. Bei der vorliegenden Hardwareimplementierung ist eine größere Skalierbarkeit nur durch die Vernetzung mehrerer ASICs zu erreichen, denn der Chip-Fläche eines integrierten Schaltkreises sind technische und physikalische Grenzen gesetzt. Derartige Grenzen gelten auch für die Bandbreite bei der Kommunikation zwischen den Chips. Der dabei auftretende Datenfluß kann durch lokale Datenverarbeitung verringert werden. Darum beruht das im folgenden erläuterte Konzept auf der Verteilung eines neuronalen Netzwerks auf *autarke Evolutionsmodule* [Grü03]. Autark bedeutet hierbei, daß das Training innerhalb der einzelnen Module durchgeführt wird, welche über das Teilnetz hinaus auch die Trainingssoftware beinhalten. Zu diesem Zweck sind sie mit einem eingebetteten Mikroprozessor ausgestattet. In dem verteilten neuronalen Netz haben die einzelnen Evolutionsmodule das Charakteristikum eines eingebetteten Systems und erfordern ein Betriebssystem.

1.1 Der neuronale Netzwerkchip HAGEN

Der zentrale Bestandteil des vorliegenden Netzwerkes ist der in $0{,}35\mu m$ CMOS[1]-Technik gefertigte HAGEN[2]-Chip mit vier gleichartigen Netzwerkblöcken [Sche03]. Auf jedem dieser Blöcke befinden sich 128 Eingangsneuronen, die über 8192 analoge Synapsen mit 64 Ausgangsneuronen verbunden sind. Die Neuronen sind digital und nehmen binäre Werte an, welche das Neuron als aktiv oder inaktiv kennzeichnen. Jeder Netzwerkblock implementiert somit ein vollständig vernetztes Perzeptron. Mit 4 solchen Netzwerkblöcken verfügt HAGEN über insgesamt 32768 analoge Synapsen auf einer Gesamtfläche von $12,3\,mm^2$.
Netzwerke, die digitale und analoge Technik vereinen, werden als hybride Netzwerke („mixed signal neural networks") bezeichnet. Sie ermöglichen die Vorteile analoger Signalverarbeitung mit der Skalierbarkeit digitaler Schnittstellen zu vereinigen. Das Neuron muß die synaptischen Analogssignale aufsummieren und die Summe mit einem Schwellenwert vergleichen. Dies kann bei analogen Synapsen in einer einfachen Summationsschaltung realisiert werden [Sche02]. Neben dem Vorteil einer geringeren Baugröße ist die synaptische Übertragung schneller, als bei einem digitalen n-Bit-Addierer. Dadurch kann eine wesentlich höhere Rechenleistung erreicht werden. Auch elektrische Leistungsaufnahme und Wärmeentwicklung sind bei diesem analogen Design geringer. Großflächige analoge Datenübertragung ist jedoch mit Signalabschwächungen und Rauschen verbunden. Durch die Kapselung in kleinere Blöcke analoger Synapsen mit digitalen Schnittstellen, werden diese Probleme umgangen.
Die Gewichte sind als programmierbare Stromquelle umgesetzt, deren Einstellung als Ladung auf einem Kondensator gespeichert wird. Die damit verbundenen Leckströme erfordern eine regelmäßige Aktualisierung im Zeitraum zwischen 10 bis 100 Millisekunden. Dies geschieht durch insgesamt 16 im Chip integrierte 10 Bit Digital Analog Converter (DAC). Dadurch erhalten die analogen Gewichte nach außen hin eine digitale Schnittstelle mit einer Präzision von 10 Bit. Durch das Setzen eines Vorzeichens, können hemmende und erregende Synapsen dargestellt werden. Zum Beschreiben der Gewichte benötigten die DACs 40 μs, sodaß auf einem einzigen HAGEN-Chip maximal $16 \cdot \frac{1}{40\mu s} = 400$ Millionen Gewichte pro Sekunde aktualisiert werden können. Das Netzwerk ist zeitdiskret mit einem Takt von bis zu 50 MHz, womit ein Durchsatz von 1,64 Tera CPS[3], das sind $1,64 \cdot 10^{12}$ synaptische Übertragungen pro Sekunde, erreicht wird.

Die Schnittstelle der digitalen Neuronen ermöglicht ein großes Ausmaß an Skalierbarkeit, welche bei analogen Signalen infolge von Leitungsverlusten so nicht möglich wäre. Die Ausgangsneuronen lassen sich nicht nur mit den Eingängen anderer Netzwerkblöcke verbinden, sondern ebenso auf die Eingangsneuronen rückkoppeln. Dadurch sind mehrschichtige Netzwerktopologien, auf mehrere Takte verteilt, schon innerhalb eines einzigen Netzwerkblocks möglich. Als Konstituenten gewährleisten die einzelnen Netzwerkblöcke einen großen Spielraum bei der Gestaltung verschiedener Netzwerktopologien. Prinizipell ermöglichen HAGEN-Chips die Umsetzung von geschichteten Perzeptronen beliebiger Größe.

[1] **C**omplementary **M**etal **O**xide **S**emiconductor
[2] **H**eidelberg **A**nalo**G** **E**volvable **N**eural **N**etwork
[3] **C**onnections **P**er **S**econd

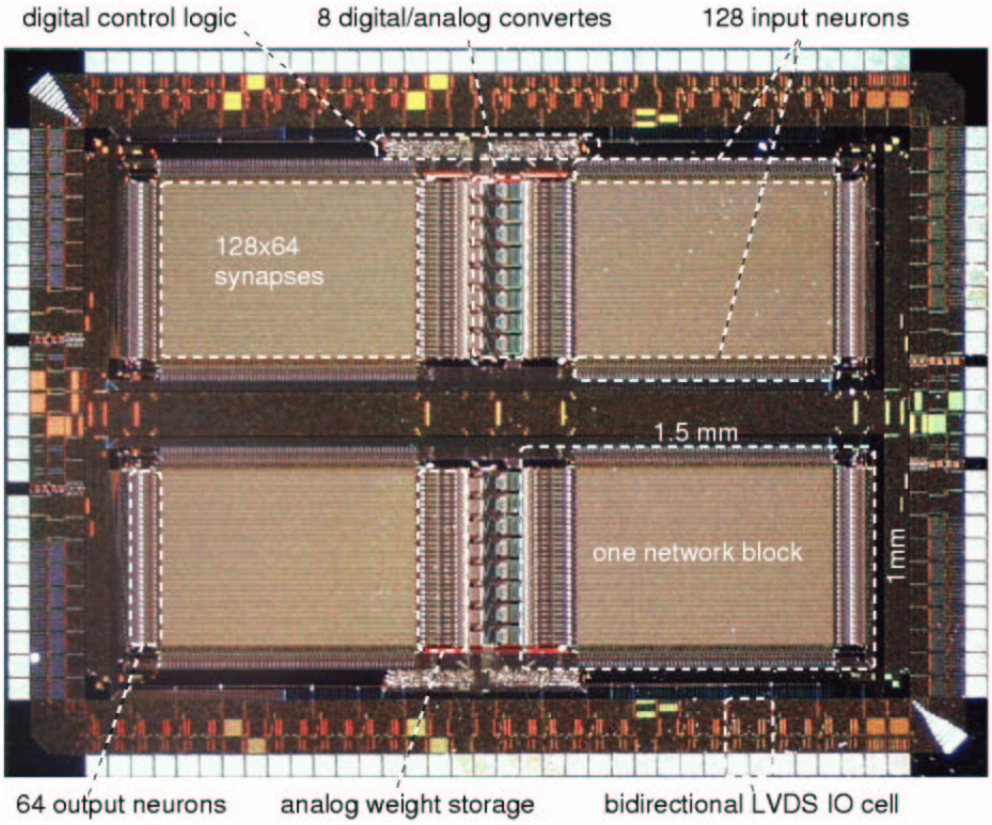


Abbildung 1.1: *Der neuronale Netzwerkchip HAGEN [Sche02]*

1.2 Der Lernalgorithmus

Das vorliegende Analogdesign hat die Eigenschaft, daß die homogen entworfenen Synapsen und Neuronen durch Prozeßvariationen bei der Chipherstellung sich elektrisch nicht genau identisch verhalten. Darüberhinaus kann es durch Effekte wie Rauschen auf Referenzspannungen oder Übersprechen von Signalen zu elektrischen Fluktuationen kommen. Die Transferfunktion ist deshalb nicht genau bekannt. Lernalgorithmen, die auf der Gradientenbildung der Fitnessfunktion basieren (der bekannteste ist *Backpropagation*), sind für das Netzwerk ungeeignet. Vielmehr legt es der hohe Durchsatz von 1,64 TCPS nahe, stark iterative Lernsalgorithmen zu verwenden, die auf der Auswertung der tatsächlichen Transferfunktion beruhen. Diese Erwägungen und das biologische Paradigma begründen die Verwendung eines genetischen Algorithmus [Sche01].
Dabei wird ein einzelner Gewichtswert durch ein *Gen* repräsentiert. Oftmals faßt man die Gewichte aller an einem Neuron einlaufenden Synapsen zu einem *Chromosom* zusammen, um eine größere Einheit zu erhalten, deren lokaler Bezug im Netzwerk dennoch auf ein Neuron beschränkt bleibt. Die Gesamtheit aller Gene oder Chromosomen stellen das *Genom* eines *Individuums* dar. Ein Individuum verkörpert eine spezielle Netzwerkkonfiguration, die durch dessen Genom als Belegung der Gewichtematrix definiert ist. Jedem Individuum kann ein Fehler oder eine *Fitness* zugeordnet werden, indem das Verhalten

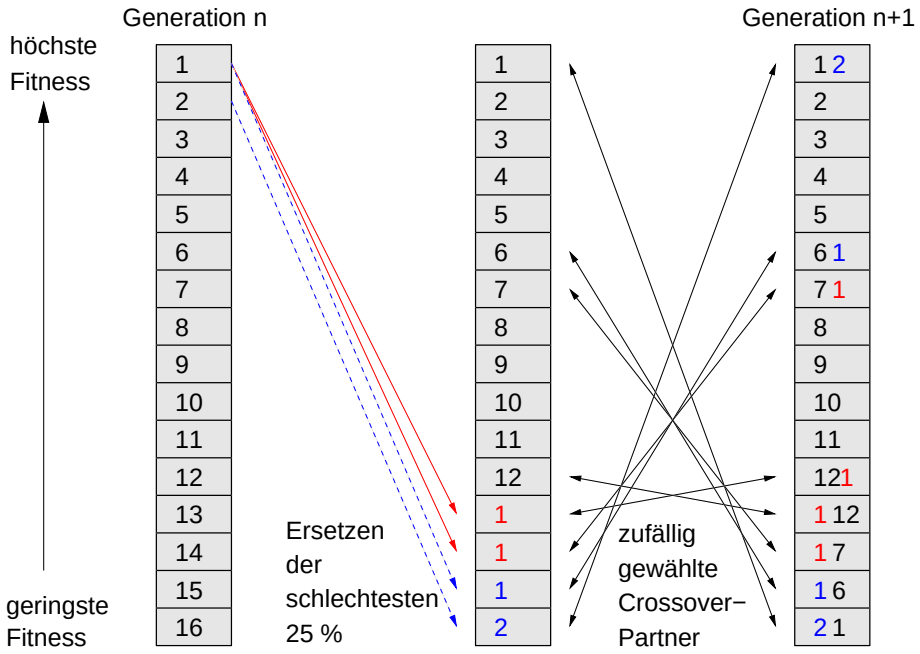


Abbildung 1.2: *Schematische Darstellung des Crossovers in der gegenwärtigen Ausführung [Sche01]*

des Netzwerkes bei der dem Genom entsprechenden Gewichtung untersucht wird. Dazu werden sogenannte Testmuster als Eingangsbelegung verwendet und die Ausgabe des Netzwerks mit den jeweiligen Sollwerten verglichen. Die Fitness nimmt bei hoher Übereinstimmung große Werte und bei starken Abweichungen kleine Werte an. Der Fehler verhält sich umgekehrt. Die genaue Berechnung der Fitness ist problemspezifisch und unter Umständen sehr aufwändig.
Eine künstliche Evolution beginnt mit der Generierung einer zufälligen Menge von Individuen (**Population**) als Ausgangsgeneration. In einem ersten Schritt wird die Fitness der einzelnen Individuen berechnet, um diese danach zu sortieren. Anschließend folgt eine Kreuzung (**Crossover**), das heißt ein Austausch von Cromosomen zwischen den Individuen. Die schlechtesten Individuen werden dabei durch Kopien der besten, sowie durch Kreuzungen verschiedener Individuuen ersetzt. Den Abschluß stellt der Mutationsoperator dar, welcher auf zufällig ausgewählte Individuen angewendet wird. Bei diesen Individuen werden alle Gene mit einer gewissen Wahrscheinlichkeit mutiert. Die Mutation ersetzt ein vorhandenes Gen durch ein zufällig generiertes neues Gen. Die Abfolge von Generationen sollte in einen stabilen Zustand konvergieren, bei dem die besten Individuen eine hohe Fitness aufweisen. Durch den Mutationsoperator kann eingestellt werden, ob und wie schnell eine Evolution konvergiert. Eine hohe Mutationsrate verhindert die Konvergenz, eine zu geringe Mutationsrate bewirkt eine schnelle Konvergenz, im allgemeinen jedoch mit dem Ergebnis einer schlechten Fitness.

Bei einer optimalen Einstellung des Lernalgorithmus ist die Trainingsgeschwindigkeit im wesentlichen durch die Neubelegung der Gewichte bestimmt. Sie wird üblicherweise in

CUPS (**C**onnection **U**pdates **P**er **S**econd) angegeben. Die effektive Umsetzung der auf jedem HAGEN-Chip erreichbaren 400 Mega CUPS stellt zweierlei Bedingungen. Zum einen müssen die Verarbeitungseinheiten für die Berechnung der Fitness (FCU[4]) und des genetischen Algorithmus (EAU[5]) eine entsprechende Rechenleistung erbringen. Zum anderen muß die Bandbreite für die erforderliche Datenrate bei der Kommunikation zwischen diesen Verarbeitungseinheiten und dem Netzwerk garantiert sein.

1.3 Das bestehende Testssystem

HAGEN kann von einem Personalcomputer aus über den PCI[6]-Bus angesteuert werden. Dazu steht die PCI-Steckkarte *Darkwing* zur Verfügung [Bec01]. Sie wurde zum Test von gemischten analog-digitalen ASICs entworfen und kann ein Testboard für den HAGEN-Chip über eine Adapterplatine aufnehmen. Ein in Darkwing integrierter FPGA[7] ist für den Datenaustausch zwischen dem Personalcomputer und dem HAGEN-Chip konfigurierbar. Darkwing ist zudem mit einem lokalen Speicher (RAM[8]) ausgestattet, in dem die Netzwerkgewichte abgelegt werden können. Der Datentransfer zwischen HAGEN und dem FPGA geschieht über einen LVDS[9]-Bus, der eine Übertragungsrate bis zu 11,4 GBit/s ermöglicht.
Derzeit sind FCU und EAU Bestandteil des Softwarepakets HANNEE[10], welches auch die mehrschichtige Ansteuerungssoftware für HAGEN und eine graphische Benutzeroberfläche beinhaltet. HANNEE wurde in C++ geschrieben und läuft auf einem Linux-PC. Damit wird der gesamte Lernalgorithmus auf einem PC ausgeführt. Dieser kommuniziert wie in der Abbildung 1.3 dargestellt mit HAGEN.

Die vom Lernalgorithmus berechneten Gewichte müssen über den 32 Bit breiten PCI-Bus übertragen werden. Bei einer Taktung von 33 Mhz beträgt die maximale Übertragungsrate 132MB/s. Im Falle einer reinen Übertragung von Gewichtsdaten wären damit höchstens $\frac{132*8\,MBit/s}{11\,Bit/Gewicht} \approx 96$ Millionen Gewichte pro Sekunde übertragbar. Der Protokollaufwand[11] bewirkt zudem, daß die reale Übertragungsrate geringer ist. Schon die 400 Mega CUPS eines einzelnen HAGEN-Chips können bei einem Gewichtetransfer über den PCI-Bus nicht annähernd umgesetzt werden. Es hat sich darüberhinaus gezeigt, daß ein 2 GHz Pentium 4 Prozessor des öffteren mit dem Training eines einzelnen Netzwerkchips schon ausgelastet ist. Das Training größerer Netzwerke wäre auf einem einzigen PC in dieser Form äußerst ineffizient.

[4] **F**itness **C**alculation **U**nit
[5] **E**volutionary **A**lgorithm **U**nit
[6] **P**eriphal **C**omponent **I**nterconnect
[7] **F**ield **P**rogrammable **G**ate **A**rray
[8] **R**andom **A**ccess **M**emory
[9] **L**ow-**V**oltage **D**ifferential **S**ignal
[10] **H**eidelberg **A**nalog **N**eural **N**etwork **E**volution **E**nvironment
[11] Ein Protokoll definiert eine Menge von Regeln für die Verständigung unter Kommunikationspartnern. Die Regeln ergeben eine exakte Spezifikation, wie die bei der Kommunikation ausgetauschten Daten zu verarbeiten sind. Beim PCI-Bus werden für die eigentlichen Daten und das Protokoll die gleichen Leitungen verwendet.

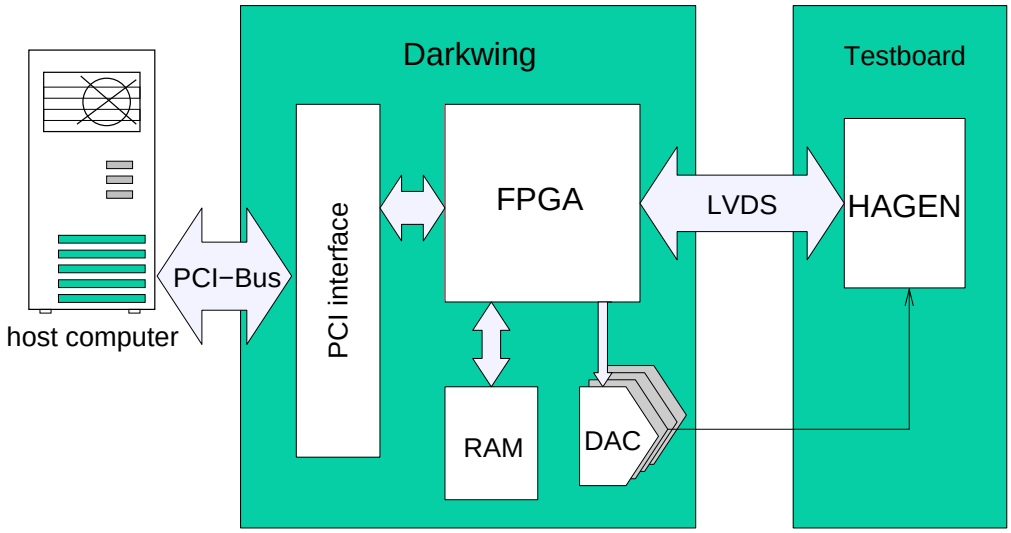


Abbildung 1.3: *Schematische Darstellung der PCI-Steckkarte Darkwing [Sche01]*

1.4 Distributed HAGEN

Um die Skalierbarkeit von HAGEN oder zukünftiger Netzwerkchips zu unterstützen, wurde die Plattform *Distributet HAGEN* entworfen [Grü03]. *Distributed HAGEN* besteht aus den Evolutionsmodulen NATHAN. Dies sind separate Platinen, welche jeweils einen Netzwerkchip aufnehmen können. Die einzelnen Evolutionsmodule werden über eine Backplane miteinander verbunden. Diese kann in ihrer gegenwärtigen Ausführung bis zu 16 Evolutionsmodule aufnehmen und gewährleistet den Anschluß weiterer Backplanes.

Die Ausstattung der NATHAN-Module ermöglicht es, einen Software-Lernalgorithmus autark innerhalb der Module auszuführen. Auf jedem NATHAN befindet sich ein *Virtex II Pro* von Xilinx, sowie ein 265 MB großes DDR-SDRAM[12]. Bis zu einem GB DDR-SDRAM können maximal aufgeneommen werden. Der *Virtex-II Pro* besteht aus einem FPGA [Xil02a] mit einem darin eingebundenen *PowerPC*[13] *405* Prozessor [Xil02b]. Zusätzlich befinden sich auf dem NATHAN und zwei SRAM[14] Blöcke zu je einem MB. Das SRAM kann zum Speichern von kleineren Datenbeständen verwendet werden, für die ein schneller Zugriff erforderlich ist. Dynamischer Speicher ist kostengünstiger, infolge längerer Zugriffszeiten jedoch weitaus langsamer als statischer Speicher. Aus Kostengründen haben solche Speicherhierarchien, bei denen schnellere, kleine Speicherienheiten mit langsameren, großen Speichereinheiten kombiniert werden, in den vergangenen Jahren weite Verbreitung gefunden [PH96].
Zur Entlastung des eingebetteten Prozessors wurde ein Coprozessor entworfen, der es erlaubt, das Crossover und die Mutation innerhalb des FPGA auszuführen [Schm03].

[12] **D**ouble **D**ata **R**ate - **S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory
[13] **P**erformance **O**ptimization **W**ith **E**nhanced **RISC**
[14] **S**tatic **R**andom **A**ccess **M**emory

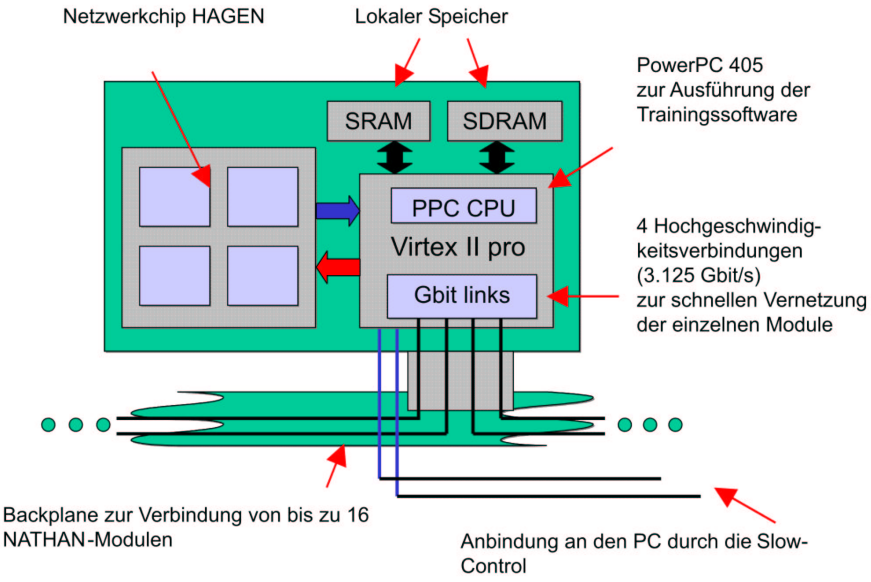


Abbildung 1.4: *Schematische Darstellung eines NATHAN-Moduls.*

Diese FPGA-basierte EAU trägt den Namen EVOCOP und erreicht auf dem *Virtex-II Pro* einen Durchsatz von 450 MByte/s. Dies ermöglicht, pro Sekunde 343 Millionen Gewichte zu berechnen, was einer 86-prozentigen Ausnutzung der auf HAGEN maximal erreichbaren Trainingsgeschwindigkeit entspricht.
Für modulinterne Netzwerktopologien ist vorgesehen, die Trainingssoftware autark auf den einzelnen Evolutionsmodulen auszuführen. Der Lernalgorithmus soll unter Verwendung des EVOCOP auf dem PowerPC ablaufen. Zur Ablaufkontrolle und Steuerung der Evolutionsmodule kann nach wie vor die graphische Benutzeroberfläche von HANNEE auf einem PC eingesetzt werden. Dies gewährleistet eine sogenannte *Slow-Control* auf der Backplane, die von einem PC aus über Darkwing angesteuert werden kann. Die *Slow-Control* ist eine serielle Ringverbindung zwischen allen NATHANs und Darkwing. Sie ist derzeit mit 40 MHz getaktet. Die Kommunikation über die *Slow-Control* verwendet ein Token-Ring-Protokoll [Tan88], das in den FPGAs implementiert wurde (Abbildung 1.5).

Ein über mehrere Evolutionsmodule verteiltes Netzwerk erfordert eine Hochgeschwindigkeitverbindung (Rocket-I/Os) für den Datenstrom zwischen den Teilnetzen. Beim Einsatz aller Neuronen beträgt der eingehende Datenstrom auf einem Netzwerkblock $128\,Bit \cdot 50\,MHz = 6,4\,Gbit/s$. Einen entsprechend schneller Datenaustausch zwischen den einzelnen NATHAN-Modulen gewährleisten 4 bidirektionale, serielle Hochgeschwindigkeitsverbindungen (Rocket I/Os) auf dem *Virtex-II Pro,* die jeweils durch eine maximale Übertragungsrate von 3,125 Gbits/s gekennzeichnet sind (Abbildung 1.6). Es ist geplant, die unteren Schichten des erforderlichen Netzwerkprotokolls auf dem FPGA auszuführen. Bei parallelem Betrieb aller 4 Netzwerkblöcke und einer entsprechenden Netzwerktoplologie sind die Rocket-I/Os durch die Netzwerksignale voll ausgelastet

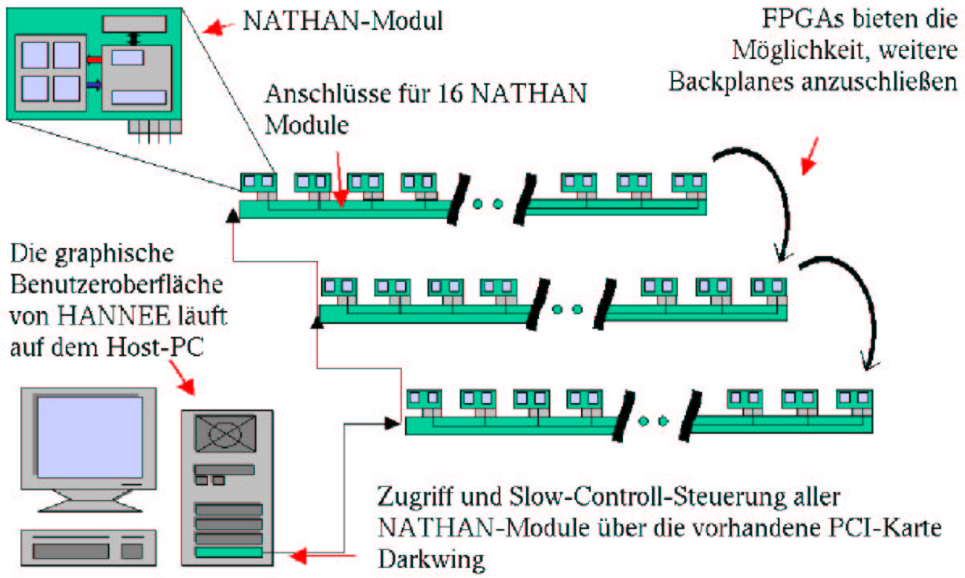
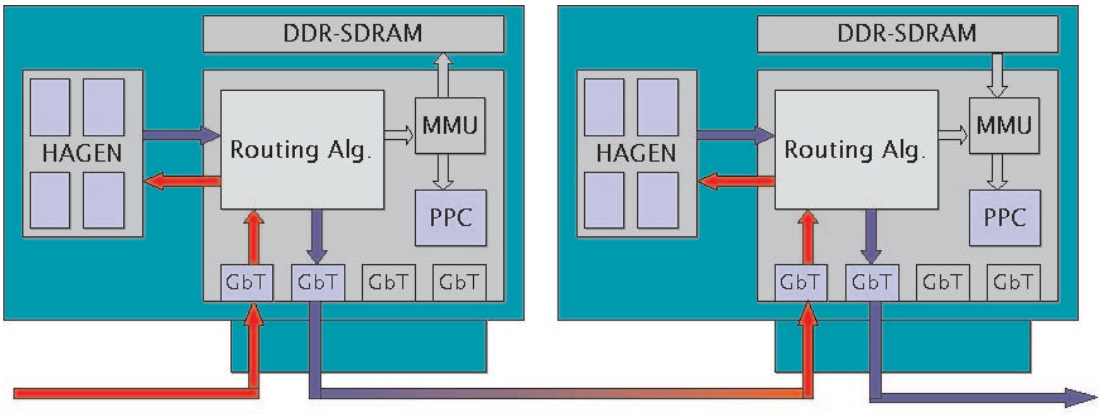


Abbildung 1.5: *Die Verbindung zwischen dem PC und den einzelnen NATHAN-Modulen auf der Backplane*

Während des Trainings bleibt der Datenstrom des verteilten Netzwerkes jedoch geringer. Er ist durch die Trainingsgeschwindigkeit gebremst und wird die Hochgeschwindigkeitsverbindungen keineswegs kontinuierlich auslasten. Diese können in der Trainingsphase nicht nur für den Datenstrom des Netzwerkes, sondern ebenso zum Austausch von Trainingsdaten der einzelnen Evolutionsmodule verwendet werden. Zu diesem Zwecke ist eine Shared-Memory-Architektur vorgesehen, die es jedem PowerPC ermöglicht, über die Rocket-I/O-Verbindungen auf das DRAM eines jeden anderen NATHAN-Moduls zuzugreifen (Abbildung 1.7).

Ein verteiltes System verlangt die Integration aller Kommunikationskanäle von den Anwendungen. Dabei muß infolge der begrenzten Bandbreite angestrebt werden, den Datenrate bei der Kommunikation zwischen den einzelnen Modulen gering zu halten. Der genetische Algorithmus ist in seiner gegenwärtigen Form geeignet, derart auf die einzelnen Module verteilt zu werden, daß keine Informationen über die Gewichte ausgetauscht werden müssen. Da die beim Crossover getauschten Chromosomen nämlich als die Gewichtung der Eingangssynapsen eines Neurons definiert sind, bleiben sie auf den einzelnen NATHANs lokalisiert. Anders ist es mit den Individuen, von denen jedes einzelne über alle NATHANs verteilt vorliegt. Die Entscheidung, welche Individuen gekreuzt werden, sowie die Fitness der einzelnen Individuen ist eine zentrale Information, die allen Modulen eines verteilten Netzwerkes zur Verfügung gestellt werden muß.
In der Trainingsphase werden demzufolge zweierlei verschiedene Daten über das Hochgeschwindigkeitsnetz laufen, die Netzwerksignale und die Trainingsdaten. Die Routingalgorithmen für diese verschiedenartigen Daten werden, wie die unteren Netzwerkprotokolle, in den FPGAs implementiert.



Netzwerksignale auf der Hochgeschwindigkeitverbindung

Abbildung 1.6:

1.5 Auswahl des Betriebssystems

Mit dem eingebetteten PowerPC 405 Prozessor des Virtex-II Pro wurden die NATHAN-Module als eingebettetes System konzipiert. Die Wahl des Virtex-II Pro als zentraler Bestandteil der Evolutionsmodule und die einhergehende grundsätzliche Entscheidung für ein Hardware-Software-Co-Design wurde unter dem Gesichtspunkt der Wiederverwertung bestehender Software getroffen. Das in C++ geschriebene Softwarepaket HANNEE erfordert dazu die Bereitstellung der verwendeten Bibliotheken und ein Betriebssystem auf dem PowerPC 405. Dem Aspekt der Wiederverwertbarkeit ist auch bei der Auswahl des Betriebssystems Rechnung zu tragen. Im Gegensatz zu anderen eingebetteten Betriebssystemen ermöglicht Linux die betriebssysteminterne Migration des linuxbasierten Testsystems nach Distributed HAGEN. Zudem erwies sich Linux gegenüber kommerziellen eingebetteten Betriebssystemen nicht nur als konkurrenzfähig, es bietet im Vergleich zu diesen sowohl Kostenvorteile als auch eine höhere Anpassungsfähigkeit, welche hauptsächlich in der Offenheit des Quellcodes (Open-Source-Modell) begründet ist.
Die Größenordnung des PowerPC 405 als 32-Bit-Prozessor entspricht derjenigen eines Personalcomputers und die NATHAN-Module erfüllen die im Abschnitt 2.1.3 näher erläuterten Hardwarevoraussetzungen für den Betrieb von Linux. Ebenso gehört der PowerPC 405 zu den von der GNU-Toolchain unterstützen Architekturen. In den offiziellen Quellen des Linuxkernels befinden sich seit der Version 2.4 Codeabschnitte für die PowerPC 400er-Reihe. Zudem gibt es eigenständige Kernelentwicklungen mit weiter entwickeltem Quellcode für den PowerPC 405. Diese werden unter der Bezeichnung Linuxppc veröffentlicht.Die Firma Monta Vista spielt dabei eine führende Rolle. Viele Programmteile für die eingebetteten PowerPC Prozessorfamilien 400 und 800 wurden von Monta Vista entwickelt und in den offiziellen Kernelquellcode aufgenommen.
Als Entscheidungskriterien spielen bei der Wahl eines Betriebssystems, neben den Hardwarevoraussetzungen, auch solche Anforderungen eine Rolle, welche der Verwendungszweck eines eingebetteten Systems an das Betriebssystem richtet. Viele eingebettete

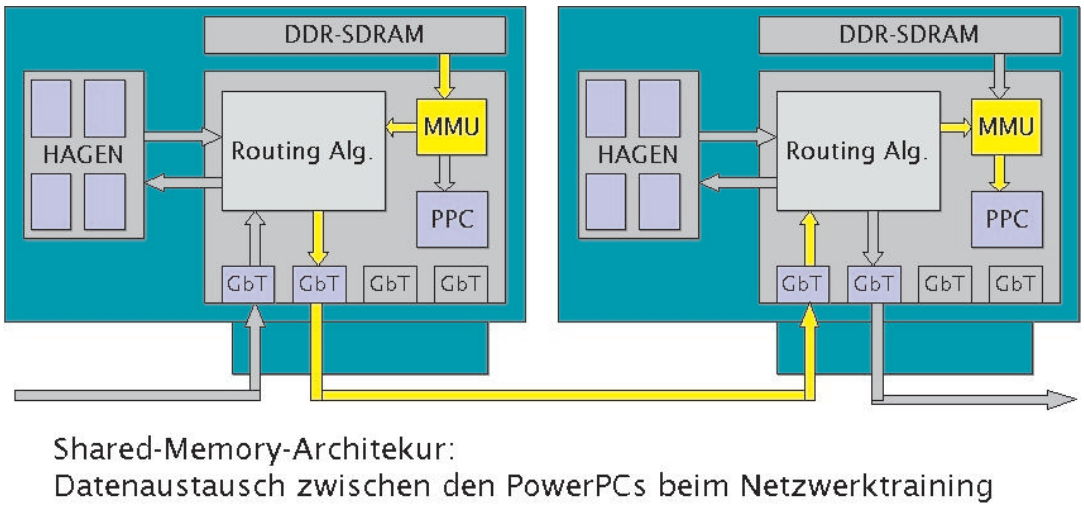


Abbildung 1.7:

Systeme in Steuerungsanlagen verlangen ein Echtzeitverhalten, d.h. die Gewährleistung fester Reaktionszeiten durch das Betriebssystem. Auf dieses Leistungsmerkmal kann für die Evolutionsmodule verzichtet werden. Von diesem Aspekt betrachtet, ist der herkömmliche Linuxkernel für die Anwendungen auf den NATHAN-Modulen geeignet, sodaß keine echtzeitfähige Linuxvariante benötigt wird[15].

1.6 Notwendige Schritte

Das verteilte neuronale Netzwerk mit der Trainingssoftware in Betrieb zu nehmen stellt hardware- und softwareseitige Aufgaben. Bezüglich der Hardware ist die Programmierung der FPGAs auf den NATHAN-Modulen vorzunehmen. Dabei sind die Anbindung des PowerPC-Prozessors und der Slow-Control an das DDR SDRAM notwendige Voraussetzung zum Laden und Ausführen jeglicher Software.
Softwareseitig ist ein funktionsfähiges Betriebssystem für den PowerPC 405 grundlegend. Das Betriebssystems muß für die spezielle Peripherie auf den NATHAN-Modulen konfiguriert und in die Maschinensprache des PowerPC 405 übersetzt werden. Die Vorgehensweise gliedert sich in drei Schritte:

1. Kernelportierung auf den PowerPC 405 (Kapitel 3)

Die Portierung des Kernels auf den PowerPC 405 ist mitunter die aufwändigste, in jedem Falle aber die kritischste Aufgabe, welche mit der Inbetriebsetzung von Linux auf den NATHAN-Modulen verbunden ist. Der Erfolg ist auf die Funktionsfähigkeit der Entwicklungswerkzeuge angewiesen. Die Entwicklungswerkzeuge für Linux sind, wie auch das Betriebssystem selbst, freie Software. Als solche befinden sie sich in einer permanenten

[15] Es gibt verschiedene echtzeitfähige Kernelentwicklungen, wozu unter anderem der Kernel von Monta Vista zählt. Der herkömmliche Linuxkernel ist jedoch nicht echtzeitfähig.

Weiterentwicklung und müssen von den Endbenutzern selbst getestet werden[16]. Die Entwicklungswerkzeuge werden dazu verwendet, den Quellcode des Linuxkernels und seiner Systemprogramme in die Maschinensprache des PowerPC 405 zu übersetzen. Wenn diese den Kernel übersetzen können, sind bei der Portierung der System- und Anwendungsprogramme nur selten weitere Probleme zu erwarten. Um Fehler bei der Übersetzung auf ihre Ursache zu analysieren, sind Kenntnisse über die Entwicklungswerkzeuge nötig, die sich vielfach als die eigentliche Fehlerquelle erweisen. Ursachen können Versionskonflikte, eine fehlerhafte Konfiguration oder die mangelhafte Unterstützung des Befehlssatzes der Zielarchitektur sein. Eine gescheiterte Kernelportierung kann aber auch auf Fehler in den Kernelquellen oder deren Makefiles (Anhang C) zurückzuführen sein.

2. Einrichten des Wurzelverzeichnisses (Kapitel 4)

Das Einrichten des Wurzelverzeichnisses kann nicht nur auf einem PC vorgenommen, sondern dort auch getestet werden. Der Kernel sorgt für die Transparenz der vorhandenen Hardware. Den System- und Anwendungsprogrammen erscheinen alle Peripheriekomponenten in der Darstellung des Kernels. Durch eine entsprechende Konfiguration des Kernels können die Besonderheiten der auf den NATHAN-Modulen vorliegenden Peripherie auf einem Personalcomputer simuliert werden. Dadurch lassen sich die Systemprogramme und die Konfiguration des eingebetteten Systems auf einem PC testen.

3. Installation des Betriebssystems auf den NATHAN-Modulen (Kapitel 5)

Linux auf den NATHAN-Modulen zu installieren bedeutet zum einen, den PowerPC 405 zum Starten des generierten Kernels zu veranlassen. Dazu ist ein Bootloader in den Speicher zu schreiben. Zum anderen muß der Kernel zuvor für den Zugriff auf die Systemprogramme und die spezielle Umsetzung der Benutzerinteraktivität präpariert werden. Dazu müssen dem Kernel individuelle Gerätetreiber zur Verfügung gestellt werden. Erst durch die abgeschlossene Installation des Betriebssystems kann der Erfolg aller vorangegangenen Ausführungen mit Gewißheit beurteilt werden.

[16] „Building a gcc / glibc cross-toolchain for use in embedded systems development used to be a scary prospect, requiring iron will, days if not weeks of effort, lots of Unix and Gnu lore, and sometimes willingness to take dodgy shortcuts. This is a problem not only for individual users, but also for the gcc project as a whole, since the gcc team relies on users to test upcoming releases of gcc, and the difficulty of building the toolchain for embedded targets restricted the number of people able to help with the testing" Dan Kegel [Keg03].

Kapitel 2

Grundlagen des Betriebssystems

Das Betriebssystem Linux besteht aus dem *Linuxkernel* und einem *Wurzelvereichnis* (*root file system*) mit einer Reihe von Systemprogrammen. Unter dem Begriff Kernel wird im allgmeinen der speicherresistenten Teil eines Betriebssystems[1] verstanden. Linux ermöglicht bei der Generierung des Kernels einen großen Gestaltungsspielraum. Der Kernelquellcode beinhaltet alle Gerätetreiber und Systemroutinen. Seit der offiziellen Version 2.4 umfaßt er über 3 Millionen Zeilen. Architekturspezifische Bestandteile und Gerätetreiber ergeben zusammen fast drei Viertel des gesamten Quellcodes und zeichnen das Betriebssystem durch eine breite Unterstützung von Plattformen und Geräten aus. Den Überblick über solch einen gewaltigen Quellcode kann nur eine gute Strukturierung garantieren. Linux hat mit dem *Schichtenmodell* ein erfolgreiches Strukturierungskonzept von Unix geerbt. Dieses kennzeichnet die in Anhang B erläuterte Organisation des Quellcodes und dessen Verzeichnisstruktur.
Ein Schichtenmodell dient der Komplexitätsreduktion. Es unterteilt ein komplexes System in mehrere Teilsysteme, welche als Schichten bezeichnet werden. Diese sind hinsichtlich ihres Abstraktionsgrades hierarchisch geordnet, und die jeweils untere Schicht bietet der darüberliegenden Schicht Dienste an, welche diese in Anspruch nehmen kann. Die Schichten sind bis auf wenige *Schnittstellen* in sich abgeschlossen. Solche Schnittstellen dienen der vertikalen Kommunikation zwischen den einzelnen Schichten. Sie definieren, welche Operationen und Dienste die untere Schicht der darüberliegenden anbietet, und wie Daten zwischen zwei Schichten ausgetauscht werden. Bei einem Betriebssystem endet die oberste Schicht mit der Schnittstelle zu den problemorientierten Anwendungen und die unterste Schicht mit der Schnittstelle zur Hardware. Modifikationen, Fehlerbehebung und Erweiterungen des unübersichtlich großen Quellcodes werden dadurch zu einer enger umrissenen Angelegenheit, die meistens auf eine bestimmte Schicht beschränkt bleibt und auch nur Kenntnisse über diese jeweilige Schicht und ihre Schnittstellen erfordert.

[1] Es herrschen unterschiedliche Auffassungen darüber, welche Komponenten eines Betriebssystems als Bestandteil des Kernels und welche als Systemprogramm umzusetzen sind. Ein monolithischer Kernel integriert weitaus mehr Systemfunktionalitäten als ein Mikrokernel. Die allgemeine Definition des Kernels als der speicherresistente Teil eines Betriebssystems vereinigt diese verschiedenen Auffassungen. Bei dieser Definition sind jedoch die Kernelmodule ausgeschlossen, die zur Laufzeit dynamisch eingebunden werden können. Die Definition ist konsistent, wenn dem Kernel nicht die Systemprogramme, sondern, wie im folgenden, das *root file system* als Vervollständigung des Betriebssystems gegenüber gestellt wird. Im *root file system* befinden sich die Systemprogramme und die Kernelmodule.

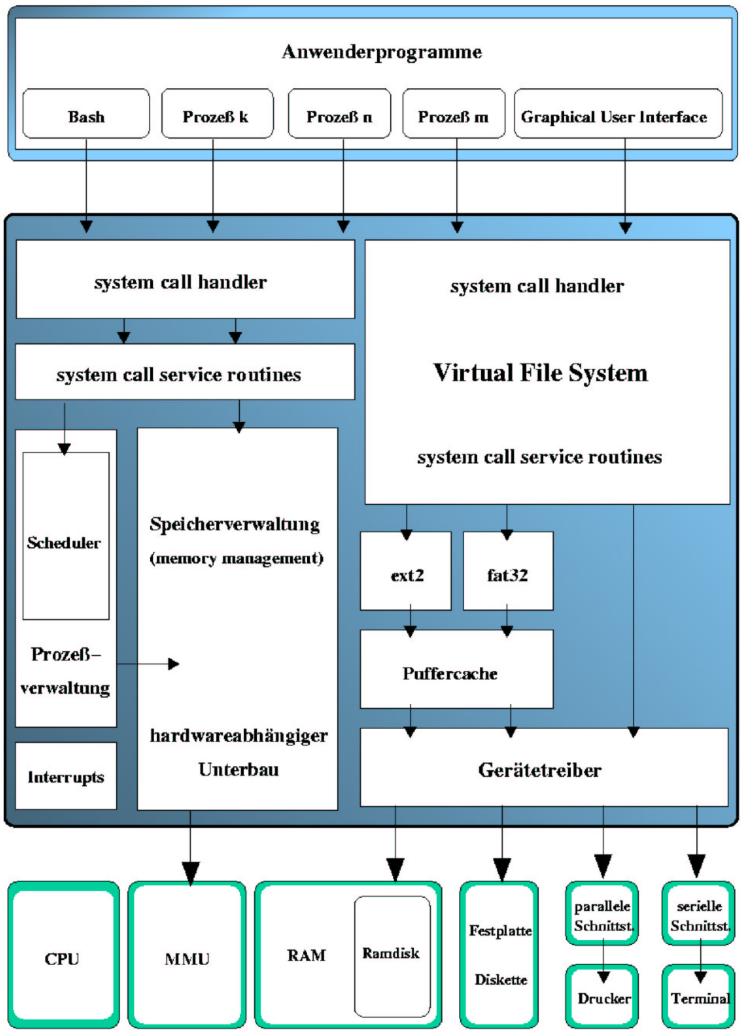


Abbildung 2.1: Übersicht über die wichtigsten Schichten und Schnittstellen des Linuxkernels.

Die einzelnen Zwischenschichten sind durch spezielle Funktionalitäten gekennzeichnet und ihrerseits in *Instanzen* gegliedert. Instanzen sind die konkreten Ausführungen dieser Schichtenfunktionalitäten. Sie liegen beim Betriebssystem in Form von Software- und darunter als Hardwareeinheiten vor. Beispiele für Softwareinstanzen sind die verschiedenen Dateisysteme, Gerätetreiber oder Prozesse. Betrachtet man den Computer in seiner Gesamtheit, so kann das Betriebssystem selbst wiederum als eine Schicht zwischen Hardware und Anwenderprogrammen aufgefaßt werden. Die Abbildung 2.1 möge als grobe Skizze der wichtigsten Schichten und Schnittstellen des Linuxkernels einen ersten Überblick vermitteln. Die Pfeile repräsentieren die Schnittstellen als Dienstanfrage und enden beim jeweiligen Dienstanbieter.

Die Schnittstelle zu den Anwenderprogrammen ist durch die Systemaufrufe gegeben, welche in Abschnitt 2.5 am wichtigen Beispiel der Dateiverwaltung betrachtet werden. Die Hauptaufmerksamkeit gilt jedoch den hardwareabhängigen Komponenten des Linuxkernels. Dies sind in erster Linie die Gerätetreiber, sowie die architekturspezifische Grundlage der Speicherverwaltung. Die Zuständigkeit der Gerätetreiber innerhalb des Kernels wird in Abschnitt 2.6 im Zusammenhang mit den I/O[2]-Schnittstellen des Kernels erläutert. Die Kooperation der Speicherverwaltung mit der Hardware beschreibt der Abschnitt 2.3. Der Arbeitsspeicher und die angeschlossenen Geräte werden beim Start des Kernels initialisiert. Darüberhinaus ist das Booten des Kernels selbst hardwareabhängig und erfordert auf eingebetteten Systemen teilweise andere Methoden als auf einem PC. Der Kernelstart auf einem PC wird im Abschnitt 2.7 beschrieben. Die Prozeßverwaltung und die Ablaufkoordination der Prozesse (*Scheduling*) sind zwar sehr wesentliche Bestandteile eines modernen Betriebssystems, jedoch weitgehend hardwareunabhängig und spielen auf eingebetteten Systemen ohne Echtzeitbedingungen allenfalls eine marginale Rolle. Insbesondere bewirkt der monofunktionelle Charakter eingebetteter Systeme, daß Aspekten der Prozeßverwaltung und des Schedulings dort eine weitaus geringere Bedeutung zukommt, als auf einer für mehrere Benutzer und Multitasking ausgerichteten Workstation.

Ein Betriebssystem besteht über den Kernel hinaus auch aus einer Sammlung von Programmen, die hinsichtlich der Arbeit auf dem System als grundlegend betrachtet werden. Linux verwaltet diese Systemprogramme innerhalb eines Dateisystems, welches als Wurzelverzeichnis oder *Root File System* bezeichnet wird. Diese Organisationsform ist für größere Datenträger mit Blockstrukturen ausgelegt, welche in eingebetteten System üblicherweise nicht zur Verfügung stehen. Für solche Systeme sind Anpassungen des Wurzelverzeichnisses vorzunehmen, um dieses im Arbeitsspeicher unterbringen zu können. Die erforderlichen Maßnahmen werden im Zusammenhang mit dem Dateisystem in Abschnitt 2.2 geschildert.

Es gibt bestimmte Leistungsmerkmale neuerer Mikroprozessoren, welche die konzeptionelle Basis eines modernen Betriebssystems darstellen. Darum muß zur Erläuterung der hardwareabhängigen Komponenten des Linuxkernels gelegentlich auf die Hardware selbst eingegangen werden. Die grundlegenden Eigenschaften neuerer Mikroprozessoren und die für Linux erforderlichen Leistungsmerkmale werden im Abschnitt 2.1 grob dargestellt.

Trotz zahlreicher Portierungen ist der 80386-kompatible PC noch immer die von Linux am besten unterstützte Plattform, und darum Ausgangspunkt der in diesem Kapitel beschriebenen Grundlagen des Betriebssystems Linux. Dieses Kapitel ist keine Einführung in den Linuxkernel oder das Betriebssystem Linux, diesbezüglich sei auf [BC01] oder [BBD01], bzw. [HHMK93] verwiesen. Es werden vielmehr nur diejenigen Grundlagen des Betriebssystem erläutert, welche hinsichtlich der Portierung auf eingebettete Systeme von Bedeutung sind.

[2] I/O steht für **I**nput/**O**utput und bezeichnet den Datenaustausch zwischen einem System und seiner Umgebung

2.1 Die Hardware

2.1.1 Prozessorarchitekturen

Die klassische Computerarchitektur ist durch ihre strikt sequentielle Befehlsverarbeitung gekennzeichnet. Ausgehend von einem System aus Mikroprozessor, Speichereinheit und I/O-Schnittstelle werden zwei grundsätzliche Varianten unterschieden:

- Die Von-Neumann-Architektur
- Die Harvard-Architektur

Dabei ist die gemeinsame Grundstruktur der sequentiellen Befehlsverarbeitung im Prozessor durch dessen Aufteilung in ein *Operationswerk* und eine *Steuereinheit* gegeben [PH98]. Beide verarbeiten binäre elektrische Signale und sind synchron getaktet. Das Operationswerk führt arithmetische und logische Operationen aus. Es kann als elektronische Implementierung einer Menge von Funktionen über die ihm eingegebenen Datensignale betrachtet werden. Die Steuereinheit übernimmt die Ablaufsteuerung im Prozessor und stellt aus mathematischer Sicht einen endlichen Automaten dar. Sie interpretiert Instruktionen und steuert ihre Ausführung, indem sie eine der im Operationswerk implementierten Funktionen über Steuersignale selektiert. Dazu generiert sie die entsprechenden Signale an den Steuerleitungen, die zum Operationswerk führen. Über die Statusleitungen empfängt die Steuereinheit Signale vom Operationswerk. Unter Berücksichtigung dieser Statussignale und des internen Zustands werden aus Instruktionen die Steuersignale für das Operationswerk generiert. Die Steuereinheit bezieht diese Instruktionen aus dem Instruktionsspeicher und das Operationswerk die zu verarbeitenden Daten aus dem Datenspeicher. Die *Harvard-Architektur* richtet dazu zwei verschiedene physikalische Speicher ein, wohingegen bei der *Von-Neumann-Architektur* beide durch einen gemeinsamen Arbeitsspeicher realisiert werden.
Wegen der hohen Kosten schneller Speicher werden bei kommerziellen Architekturen Daten und Instruktionen zuvor aus einem langsameren Arbeitsspeicher auf der Basis von DRAM-Modulen in kleine, prozessorinterne, schnelle Speichereinheiten geladen, für die sich die Bezeichnung *Register* durchsetzte. Die Register sind schnelle SRAMs mit Zugriffzeiten, die der Taktfrequenz von Operationswerk und Steuereinheit entsprechen. Es wird zwischen Instruktionsregistern und Datenregistern unterschieden. Die Größe der Instruktionsregister nimmt Einfluß auf das Instruktionsformat, wohingegen die Menge der verfügbaren elementaren Instruktionen selbst überwiegend durch das Operationswerk festgelegt ist. Die Breite der Datenregister entspricht in vielen Fällen der Größe des adressierbaren Speichers. Die Prozesoren neuerer Personalcomputer verfügen größten Teils über 32 Bit breite Datenregister und unterstützen einen Adreßraum von $2^{32}\,Bit = 4\,GB$. Sie werden darum als 32-Bit-Architekturen bezeichnet. Inzwischen werden auch 64-Bit-Architekturen produziert, die auf Servern und Workstations zum Einsatz kommen.
Bei sehr frühen Rechnern gab es im Prozessor nur ein Datenregister, den Akkumulator, in dem Rechenergebnisse gespeichert wurden. Alle anderen Daten und Instruktionen wurden direkt aus dem Arbeitsspeicher geholt. Derartige Rechner bezeichnet man als *Akkumulatormaschinen*. Bei der *Stackmaschine* beziehen sich die Befehle auf einen Stapel (*Stack*) zur Speicherung von Rechen- und Zwischenergebnissen. Dieser Stapel befindet sich weitgehend im Arbeitsspeicher, jedoch gibt es ein Register im Prozessor, in

dem die Adresse des obersten Stapelelements gespeichert ist, sowie einen oder mehrere weitere Register mit dem Inhalt der obersten Stapelelemente. Die Ausführung moderner Mikroprozessoren als *Registersatzmaschine* ist durch eine Vielzahl von Registern mit unterschiedlichen Funktionen gekennzeichnet. Neben allgemein verwendbaren Registern gibt es Spezialregister, wie beispielsweise Register zum Speichern besonderer Ereignisse, zur Adressierung von Speicher oder zur Unterstützung von Leistungsmerkmalen moderner Betriebssysteme.

Seit der Einführung des Mikroprozessors auf einem integrierten Schaltkreis wurden zusehends höhere Taktfrequenz erzielt. Einhergehend zeichnete sich eine kontinuierlich ansteigende Integrationsdichte ab. So konnten neben dem eigentlichen Mikroprozessor zunehmend mehr Einheiten auf einem Chip untergebracht werden. Zur Unterscheidung haben sich die Bezeichnungen *Prozessorblock* für den gesamten Chip und *Prozessorkern* (CPU[3]) für die Steuereinheit, das Operationswerk und die zugehörigen Register durchgesetzt. Zur typischen Ausstattung eines Prozessorblocks zählen heute eine Speicherverwaltungseinheit (Abschnitt 2.3.3), ein Interrupt-Controler (Abschnitt 2.6) und Spezialeinheiten zur Unterstützung von Fließkommaberechnung (*floating point unit, FPU*) oder Multimediafunktionen (*multimedia extention, MMX*). Ein im Chip integrierter Cache hält häufig verwendete Instruktionen und Daten zum schnelleren Zugriff für die CPU in einem SRAM bereit. Diese kleine Speichereinheit besteht aus einzelnen Zeilen (*Cachelines*) fester Größe und der Arbeitsspeicher wird in eine Anzahl von Blöcken gleicher Größe aufgeteilt. Von diesen befindet sich ein kleiner Anteil in den Cachelines des SRAMs. Die Zugriffszeiten auf den Cache entsprechen der Taktfrequenz der CPU. Wenn diese auf eine im Cache vorliegende Speicheradresse zugreift, wird deren Inhalt direkt aus dem Cache bezogen, anderenfalls wird der Block, der die entsprechende Speicheradresse beinhaltet, in den Cache geladen und ein anderer Block dafür in den Arbeitsspeicherspeicher zurückgeschrieben. Ein Cachecontroller entscheidet nach einem bestimmten Algorithmus, welcher Block in den Arbeitsspeicher zurückgeschrieben wird. Jeder im Cache befindliche Speicherblock wird durch ein sogenanntes *Tag* identifiziert. Die Tags bestehen meistens aus den ersten höherwertigen Bits der Speicheradresse. Durch das Einladen zusammenhängender Speicherbereiche in ganzen Blöcken steigt die Wahrscheinlichkeit, die Adressse des nächsten Zugriffs im Cache vorzufinden [PH96].

Neuere Rechnerarchitekturen zeichnen sich außerdem durch fortschreitende *Parallelisierung* aus. Solche findet zum einen in Multiprozessorsystemen Gestalt oder aber auch in der Parallelisierung der Instruktionsverarbeitung einzelner Prozessoren. Erste Schritte in diese Richtung begannen mit dem *Pipelining* [PH96]. Die klassische Instruktionsverarbeitung zeichnet sich in 4 Phasen ab:

1. Holphase (fetch): Befehl und Operand(en) laden
2. Dekodierphase (decode): Befehl dekodieren
3. Rechenphase (execute): Befehl ausführen
4. Bringphase (write back): Ergebnis zurückschreiben

[3] **C**entral **P**rocessing **U**nit

Beim Pipelining befinden sich gleichzeitig mehrere Instruktionen in verschiedenen Verarbeitungsphasen. Bei einem 4-stufigem Pipelining können unter günstigen Umständen 4 Befehle gleichzeitig bearbeitet werden. Wenn jede Verarbeitungsphase einen Takt benötigt, würde dann im Durchschnitt pro Takt ein Befehl verarbeitet werden. Dies ist nicht immer möglich, da mitunter eine Instruktion vom Ergebnis der vorangegangenen Instruktion abhängt. Codeoptimierung seitens des Programmierers oder durch den Compiler kann das Auftreten solcher Situationen verringern. Zur Unterstützung von Pipelining verwenden manche Computer eine Befehlssatzarchitektur (*reduced instruction set computer, RISC*), die sich durch einfache Instruktionen auszeichnet, welche im Idealfall pro Phase genau einen Prozessortakt benötigen. Man spricht hierbei auch von einer *Load/Store-Architektur*, weil mit Außnahme der exklusiven Instruktionen Load und Store, ausschließlich Register als Operanden verwendet werden. Komplizierte Adressierungsarten werden dabei ausgeschlossen.
Viele Computer, wie der Intel 80386 verfügen jedoch über sehr komplexe Befehle, um die Konstrukte höherer Programmiersprachen besser zu unterstützen (*complex instruction set computer, CISC*). Bei allen Intel-Prozessoren der 80x86-Reihe sollte aus Gründen der Kompatibilität der Befehlssatz nach außen hin den des 80368 beinhalten. Da viele dieser Befehle jedoch weitaus mehr als einen Taktzyklus benötigen, wurden die Prozessoren seit dem Pentium Pro mit einem RISC-Kern konstruiert, dem die 80386-Befehle durch ein zwischengelagertes Schaltwerk in mehrere einfachere Operationen übersetzt werden, um auf diese Art ein Pipelining mit größerem Durchsatz zu erzielen. Auf Seite der Software wird Pipelining von modernen Compilern (Abschnitt 3.3) durch eine Optimierung des Instruktionsflusses unterstützt.

Während beim Pipelining versucht wird, die einzelnen Einheiten des Prozessors voll auszulasten, die Instruktionen jedoch noch nacheinander ausgeführt werden, geschieht die Verarbeitung der Instruktionen beim Konzept der *Superskalarität* tatsächlich parallel. Hierbei werden die Befehle in verschiedene Klassen eingeteilt, die von unterschiedlichen Funktionseinheiten des Operationswerkes bearbeitet werden. So können zwei Befehle verschiedener Klassen dann tatsächlich parallel in der jeweiligen Einheit verarbeitet werden. Bei der vielfachen Superskalarität liegen einzelne Funktionseinheiten mehrmalig vor. Beispielsweise wurde schon der Pentium II mit zwei Integer- und zwei Fließkommaeinheiten ausgestattet, welche zudem ein mehrstufiges Pipelining unterstützen. Superskalare Architekturen sind inzwischen sehr gewöhnlich. Dabei wird das Scheduling, wie die Optimierung der Befehlsfolge genannt wird, von der Hardware übernommen, sodaß Compiler und Betriebssystem keiner Anpassungen bedürfen [PH96]. Im folgenden werden die bei einer Portierung relevanten Hardwareaspekte in einem Schichtenmodel eingegrenzt.

2.1.2 Der Computer in der Darstellung des Schichtenmodells

Eine Verfeinerung des Schichtenmodells für Computer unterteilt auch dessen Hardware in mehrere Ebenen. Im Modell von Tanenbaum sind die einzelnen Ebenen durch eine jeweilige Sprache gekennzeichnet [TG99]. Dabei ist die unterste Schicht die *Ebene der digitalen Logik*, deren Sprache elektrische Signale sind, welche direkt von der elektronischen Schaltung ausgeführt werden können. Diese Ebene besteht aus elementaren logischen Schaltungen, die als Gatter bezeichneten werden. Aus mehreren Gattern kann

beispielsweise ein 1-Bit-Speicher gebildet werden, der einen zweier möglicher elektrischer Zustände speichern kann. Die Register sind auf dieser Ebene Gruppierungen solcher 1-Bit-Speicher.
Die darüberliegende Schicht ist die *Mikro-Architektur-Ebene*. In dieser wird der Aufbau der Steuereinheit und des Operationswerks aus kleineren Funktionseinheiten, sowie die Verbindungsstruktur zwischen Operationswerk und den Registern (der Datenpfad) betrachtet. Die konkrete Implementierung der Instruktionsverarbeitung, sowie die Optimierung und Parallelisierung derselben vollzieht sich ebenso auf dieser Ebene.
Die nächst höhere Schicht ist die *Befehlssatzarchitektur* (instruction set architecture, ISA). Sie legt den Befehlssatz und alle Aspekte der Maschinensprache fest, welche die Programmierung betreffen. Insbesondere definiert sie die Größe der Register, die Instruktionen und deren Format, die Adressierungsarten, den Adreßraum und das Speichermodell. Die Befehlssatzarchitektur ist demzufolge für die Kompatibilität von Maschinenprogrammen verantwortlich. Prozessoren mit der gleichen oder teilweise gleichen Befehlssatzarchitektur bilden eine *Prozessorfamilie*. Beispielsweise ist es in Linux üblich, alle mit dem Intel 80386 kompatiblen Prozessoren als Prozessorfamilie i386 zu bezeichnen. Prozessoren mit teilweise sehr verschiedenartiger Mikro-Architektur können die gleiche Befehlssatzarchitektur haben, weshalb zwei kompatible Rechner ein und dasselbe Programm nicht selten mit verschiedenem Zeitbedarf ausführen. Die Befehlssatzarchitektur ist es, welche der Schicht des Betriebssystems zugrundeliegt.

Schicht	Sprache
Anwendungsprogrammierung	Hochsprachen
Betriebssystem	Systemaufrufe
Befehlssatzarchitektur	Maschinensprache
Mikro-Architektur	Mikroprogrammsprache
Digitale Logik	elektrische Signale

2.1.3 Hardwarevoraussetzungen für Linux

Der Linuxkernel wurde ursprünglich für die 32-Bit-Architektur des Intel 80386 entwickelt. Heute unterstützt Linux eine Vielzahl verschiedener 32-Bit-Prozessoren und darüberhinaus auch einige neuere 64-Bit-Prozessoren. Kleinere Prozessoren werden von Linux derzeit nicht offiziell unterstützt[4]. Linux erfordert mindestens 3 bis 4 MB Arbeitsspeicher. Geringere Speicherressourcen sind mit Linux nur unter einem erheblichen Verzicht auf Funktionalität zu vereinbaren. Eingebettete Systeme, wie sie in MP3-Playern,

[4]In Bezug auf 16-Bit-Architektur kann auf die Arbeiten von Embeddable-Linux-Kernel-Subset verwiesen werden, die unter http://elks.sourceforge.net veröffentlicht sind. Im Rahmen dieses Projektes wurde unter anderem ein Linuxkernel für die 16-Bit-Prozessoren der Intel-Modelle 8086 und 80286 entwickelt. Noch kleinerer Mikroprozessoren, wie sie in zahlreichen eingebetten Systemen, beispielsweise für Kühlschränke, TV-Geräte oder Musikanlagen, zum Einsatz kommen, können mit Linux nicht betrieben werden. Diese eingebetteten Systeme stellen meistens sehr geringe Anforderungen an die Betriebssoftware, sodaß ein ürsprünglich für Mehrbenutzersysteme konzipiertes Betriebssystem wie Linux funktionell überladen und unzweckmäßig ist.

Organizern, PDAs (Personal Digital Assistant), Spielkonsolen oder kleineren Industrie-Robotern vorliegen, werden oftmals mit einem 32-Bit-Prozessor und Speichergrößen zwischen 32 und 512 MB betrieben. Die NATHAN-Module mit dem *PowerPC 405* liegen in der gleichen Größenordnung, welche derjenigen eines Personalcomputers entspricht. Viele Systeme dieser Größenordnung werden von Linux unterstützt. In Telecom-Switches, Steuerungssystemen von Industrieanlagen, Navigationssystemen und für andere rechenintensiven Steuer-, Kontroll- oder Auswertungsaufgaben werden zumeist eingebettete Systeme mit sehr großem Arbeitsspeicher und mehreren CPUs verwendet. Linux setzt der Speichergröße prinzipiell keine obere Grenze, eine solche ist einzig durch den von der CPU oder der Speicherverwaltungseinheit festgelegten Adreßraum gegeben. Es gibt sowohl Linux-Cluster als auch die Unterstützung von Symmetric Multiprocessing durch Linux.

Die Speicherverwaltung ist ein zentraler Bestandteil neuerer Betriebssysteme. Eine Speicherverwaltungseinheit (Memory Management Unit, *MMU*) ist im offiziellen Linuxkernel, wie bei nahezu allen modernen Betriebsystemen, die Grundlage der Speicherverwaltung (Abschnitt 2.3.3). Wenn ein eingebettetes System über eine MMU verfügt, muß diese im Quellcode des Linuxkernels unterstützt werden. Für eingebettete Systeme ohne MMU wurde das Linuxderivat uClinux entwickelt (http://www.uclinux.org/).

Eine weitere Hardwaregrundlage moderner Betriebssysteme sind Unterbrechungen (Abschnitt 2.5). Auf dieser Erweiterung der klassischen Prozessorarchitektur basieren wichtige Mechanismen des Linuxkernels. Unterbrechungen werden auf verschiedenen Prozessoren in unterschiedlicher Weise implementiert. Der Linuxkernel muß die jeweilige Umsetzung der Unterbrechungen im Quellcode unterstützen.

Über all dies hinaus sind geeignete Entwicklungswerkzeuge notwendig, welche die Befehlsatzarchitektur des Zielsystems unterstützen. Linux wurde mit der GNU-Toolchain entwickelt und verwendet viele besondere Eigenschaften des GNU C-Compilers, welcher das wichtigste Werkzeug bei der Linuxportierung darstellt. Die Unterstützung der Zielplattform und ihrer Befehlsatzarchitektur durch die GNU-Toolchain gilt darum als Voraussetzung für eine Portierung.

2.2 Das Dateisystem

Über die Systemaufrufe stellt der Linuxkernel den System- und Anwendungsprogrammen alle Hardwareressourcen zur Verfügung, jedoch beinhaltet er selbst keine Instanz der Benutzerinteraktivität. In einem Linuxsystem ist solche als Systemprogramm im Wurzelverzeichnis implementiert. Das Wurzelverzeichnis muß in eingebetteten Systemen meistens im Speicher angelegt werden. Dazu ist ein Dateisystem innerhalb des Speicher anzulegen.

2.2.1 Der Begriff der Datei in Unixsystemen

Eine der grundlegenden Anforderungen an ein Betriebssystem ist es, die aus Anwendersicht relevanten Möglichkeiten auf einem Rechner mit all seinen Hardwarebestandteilen

in einer den Anwenderbedürfnissen entsprechenden Weise zu repräsentieren. Dies beinhaltet sowohl das Verstecken der Details technischer Umsetzung, als auch die Simplifizierung und Vereinheitlichung von Zugriffsmethoden auf verschiedenartige Ressourcen. Ein Abstraktionskonzept hierzu ist die *Datei*. In Unixsystemen wird diese Konzept weit ausgedehnt, sodaß über die Repräsentation von ausführbaren Programmen und Datenmengen hinaus auch alle Geräte und laufende Prozesse als Datei abbildet werden.
Unter Unix sind dem Anwender im allgemeinen 3 Operationen auf Dateien gestattet, das *Lesen*, *Schreiben* (oder Ändern), sowie das *Ausführen* einer Datei. Für die ersten beiden Operationen führt Unix den Begriff des *Datenstroms* ein, demgemäß das Schreiben als Datenstrom in eine Datei und das Lesen als Datenstrom aus einer Datei verstanden wird. Entsprechend wird der Zugriff auf die periphere Hardware in Form von Datenströmen dargestellt, welche sich auf die jeweiligen Gerätedateien[5]beziehen. Die Datei ist dadurch neben dem Konzept des Prozesses ein essentieller Bestandteil von Unixsystemen.
Der *Prozeß*, bezieht sich auf das Ausführen einer Datei. Hierbei wird deren Inhalt als Sequenz von Instruktionen für den Prozessor verstanden (sogenanntes Maschinenprogramm oder binary file), welche dieser in einem oder mehreren Prozessen ausführt. Ein Prozeß ist in der Theorie der Betriebssysteme eine Einheit aus solchen Instruktionen und den zugehörigen Daten und Speicherbereichen, die dem Prozessor vom Betriebssystem zugewiesen werden kann. Zur Verwaltung der Prozesse und ihrer jeweiligen Zustände initialisiert das Betriebssystem für jeden neugestarteten Prozeß eine komplexe Datenstruktur mit der Bezeichnung Prozeßdeskriptor. Alle Prozeßdeskriptoren sind hierarchisch verkettet und werden zudem in einer internen Prozeßtabelle verwaltet. Die Prozeßverwaltung ist bei einem modernen Betriebssystem zentraler Bestandteil, zugleich aber auch die Komponente, welche die kleinste Schnittstelle mit der zugrundeliegenden Hardware aufweist. Abgesehen von der Auslagerung eines Prozesses auf externe Speichermedien (sogenanntes Swapping) ist der Prozeß grundsätzlich im Arbeitsspeicher und Prozessor lokalisiert und damit in einem eingebetteten System prinzipiell genauso wie auf einem PC oder Großrechner implementiert.
Dies gilt nicht für die Datei. Bei einem PC liegen die Dateien normalerweise auf externen Datenträgern, wie Disketten oder Festplatten, welche die Möglichkeit des wahlfreien Zugriffs bieten, im Gegensatz zu Bändern also nicht sequentiell von einem Anfangspunkt aus eingelesen werden müssen. Um den wahlfreien Zugriff auf Disketten oder Festplatten einheitlich und von den hardwarenahen Eigenschaften (Köpfe, Spuren, Sektoren etc.) abstrahiert darzustellen, werden derartige Geräte vereinfacht in eine bestimmte Anzahl gleich großer Blöcke unterteilt, die aufeinanderfolgend durchnumeriert und anhand dieser Blocknummer direkt zu erreichen sind. Solche Geräte werden deswegen als blockorientiert bezeichnet. Ein Dateisystem ermöglich es, über verschiedene Blöcke verteilte Dateien zu lokalisieren und auf die einzelnen Dateien zuzugreifen. Es ist in Unix generell üblich, die zu diesen Dateiverwaltungsmaßnamen nötigen Informationen strikt von den eigentlichen Daten zu trennen. Die Verwaltungsinformationen stehen in sogenannten Information Nodes oder *Inodes* (Informationsknoten), welche für kleinere Dateien unter anderem deren Blocknummer beinhalten, bei größeren Dateien die Nummern indirekter Blöcke, die ihrerseits die entsprechenden Blocknummern beinhalten. Alle für die Verwaltung des gesamten Dateisystems wichtigen Informationen sind in einem als Superblock

[5]Diese befinden sich standardmäßig als sogenannte device files im Verzeichnis /dev/

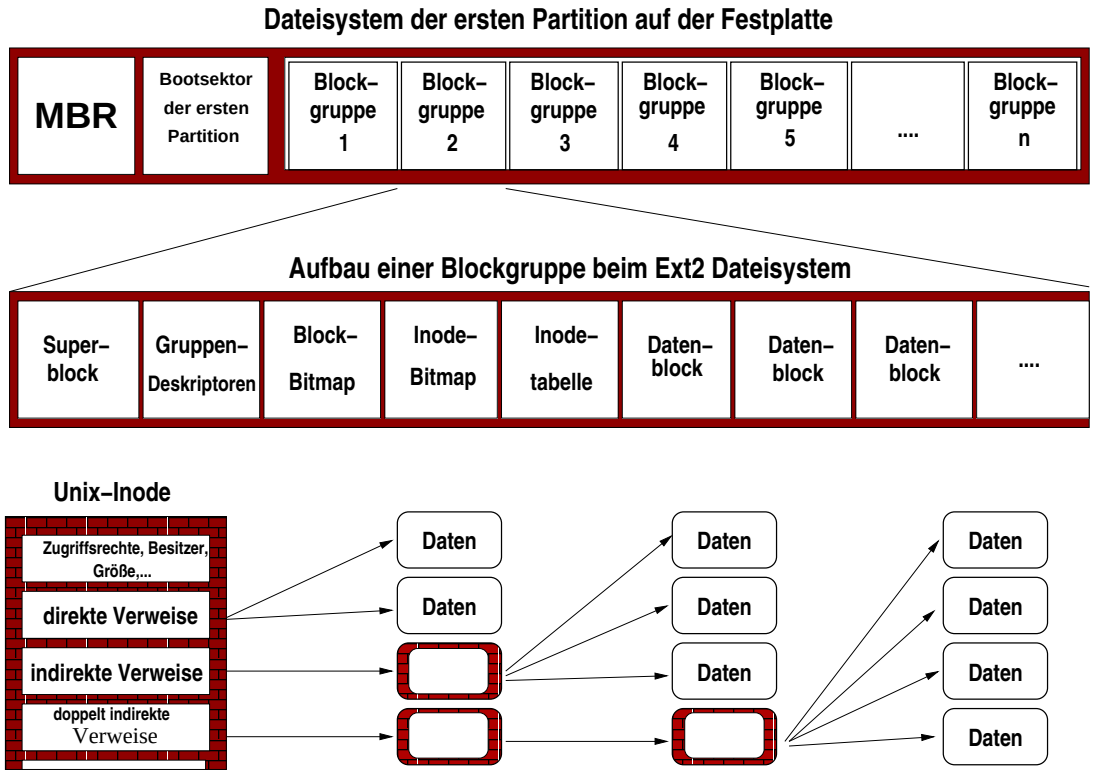


Abbildung 2.2: Veranschaulichung eines Dateisystems am Beispiel des EXT2 (extended file system 2).

ausgezeichneten Block gespeichert. Das Anlegen eines solchen Dateisystems, also das Schreiben des Superblocks und das Reservieren von Blöcken für Inodes, bezeichnet man als *Formatieren* des Datenträgers.

2.2.2 Das Linux Wurzelverzeichnis

Als Unixderivat benötigt auch Linux mindestens ein Dateisystem, welches als Wurzelverzeichnis oder *root file system* bezeichnet wird. In dieses baumartige Dateisystem lassen sich weiere Speichermedien mit diversen Dateisystemformaten als Äste einhängen (*mounten*) und auch wieder aushängen. Das Wurzelverzeichnis selbst ist jedoch als essentieller Bestandteil des Betriebssystems unentbehrlich und muß in einem der Linux eigenen Formate (z.B. Extended Linux File System, Second Extended File System oder Reiser File System) vorliegen. Üblicherweise befindet sich dieses Wurzelverzeichnis bei PCs auf einer Festplatte oder Festplattenpartition. Für ein embedded Linux kommt hingegen oftmals nur der Arbeitsspeicher als Träger des Wurzelverzeichnisses in Frage. Einen zu diesem Zweck im Arbeitsspeicher reservierten Bereich bezeichnet man als *Ramdisk*. Solch eine Ramdisk läßt sich ebenso mit einem Dateisystem formatieren, wie jedes blockorientierte Speichermedium oder ein Teilbereich auf einem solchen. Bei der Formatierung dieses eigentlich linearen Speichermediums werden Blockstrukturen mit

entsprechenden Inodes und Datenblöcken wie auf einer Festplatte angelegt.
Der im Arbeitsspeicher resistente *Linuxkernel* ist ein Maschinenprogramm, welches lediglich fundamentale Aufgaben wie Prozeß-, Speicher- und IO-Verwaltung übernimmt, selbst jedoch noch keinen Kommandointerpreter oder ähnliche Systemprogramme zur Verfügung stellt. Solche liegen als ausführbare Dateien (also als Maschinenprogramme des Betriebssystems[6]) im Wurzelverzeichnis, welches der Kernel demzufolge notwendigerweise einbinden muß. Befindet sich dieses in einer Ramdisk, so ist der Kernel entsprechend zu konfigurieren. Diese Konfigurationen stehen im Zusammenhang mit dem Startvorgang des Linuxkernels, welcher in Abschnitt 2.7 beschrieben wird.

2.3 Die Speicherverwaltung

Ein modernes Betriebssystem bietet dem Anwender die Möglichkeit, mehrere Programme im gleichen Zeitraum auszuführen. Diese Nebenläufigkeit wird auch als *Multitasking* bezeichnet. Auf der Seite des Betriebssystems werden Programme bei ihrer Ausführung als ein oder mehrere Prozesse gehandhabt. Wie in 2.2.1 bereits erwähnt, sind die dabei verwendeten Speicherbereiche Bestandteil der jeweiligen Prozesse. Die Speicherverwaltung eines multitaskingfähigen Mehrbenutzerbetriebssystems muß den dynamisch erzeugten Prozessen den erforderlichen Arbeitsspeicher einräumen. Da der vorhandene Arbeitsspeicher eine oftmals sehr knappe Ressource ist, sollte er so effizient wie möglich ausgenutzt werden. Eine Voraussetzung dafür sind *relozierbare* Prozesse, die im Arbeitsspeicher ortsunabhängig sind. Die Relozierbarkeit ermöglicht das Einlagern eines Prozesses an eine beliebige freie Stelle im Arbeitsspeicher. Den Programmen wird ein virtueller Adreßraum zur Verfügung gestellt. Die Aufteilung des Arbeitsspeichers ist dann nicht die Aufgabe des Programmierers, sondern die des Betriebssystems, welches eine absolute Adressierung erst bei der Programmausführung vornimmt. Überdies müssen verschiedene Prozesse vor gegenseitigem Überschreiben oder Ausspionieren geschützt werden. Für all diese Aufgaben erhält das Betriebssystem grundlegende Unterstützung von Seiten der Hardware durch eine Speicherverwaltungseinheit. Demzufolge gehört die Speicherverwaltung zu den besonders hardwareabhängigen Bestandteilen eines Betriebssystems. Die Speicherverwaltung ist auf Intel-Prozessoren nicht unkompliziert. Sie trägt die geschichtlichen Spuren der Kompatibilitätszwänge, die sich aus der marktführenden Position Intels mit dem Betriebssystem MS-DOS ergeben haben. Linux versucht diese Eigenarten weitgehend zu umgehen, mit dem Ziel einer umfassenderen Portabilität.

Eingebettete Mikroprozessoren haben oftmals eine spezielle Speicherverwaltungseinheit, sodaß der architekturabhängige Teil der Speicherverwaltung für diese modifiziert werden muß. Veränderungen sind dann auch an den entsprechenden Initialisierungsroutinen beim Kernelstart vorzunehmen.

[6]Der Linuxkernel ist kein solches Programm. Der Versuch die Datei mit dem Abbild des Kernels (*Kernelimage*) als Superuser auszuführen liefert die Fehlermeldung *cannot execute binary file*. Dies gilt sowohl für einen komprimierten als auch für einen unkomprimierten Kernel. Der wesentliche Grund dafür ist, daß sich die Speicherverwaltung und der Prozessor bei laufendem Betriebssystem in einem anderen Zustand als nach einem Reset befinden (Abschnitt 2.3.2).

2.3.1 Speicher und Adreßraum

Der klassischen von Neumann-Architektur entsprechend, arbeiten auf Intel-Prozessoren basierende Personalcomputer mit einem gemeinsamen Arbeitsspeicher für Daten und Instruktionen, welche der Prozessor über den *Datenbus* aus dem Arbeitsspeicher beziehen oder dort ablegen kann. Man spricht immer dann von einem *Bus*, wenn sich mehrere Einheiten oder Geräte ein Leitungsbündel so teilen, daß zu jedem Zeitpunkt höchstens eines davon diese Leitungen in Anspruch nimmt. Die Anzahl der Leitungen wird als die Breite des Busses bezeichnet und legt fest, wie viele Bit gleichzeitig übertragen werden können. Das Bussystem einer von Neumann-Architekur setzt sich im allgemeinen aus drei spezialisierten Bussen zusammen:

- Der *Datenbus* besteht aus Leitungen zum parallelen Datentransport (beim Intel 80286 mit 16 Bit, beim 80386 mit 32 Bit und beim Intel Pentium mit der Breite von 64 Bit, dh. 64 Leitungen)
- Der *Adreßbus* dient dem Übermitteln von Adressen, und die Anzahl seiner Leitungen bestimmt die Größe des physikalischen Adreßraums (ab dem 80386 mit einer Breite von 32 Bit im Protected Mode)
- Der *Steuerbus* besteht aus einer Gruppe von Leitungen zum Übertragen von Steuersignalen an die angebundenen Schaltkreise. Beispielsweise wird über den Steuerbus festgelegt, ob lesend oder schreibend auf den Arbeitsspeicher zugegriffen werden soll.

Auf welchen Bereich des Speichers der Zugriff über den Datenbus erfolgt, wird durch den *Adreßbus* von der CPU festgelegt. Der in seiner Größe von der Adreßbusbreite bestimmte physikalische Adreßraum beinhaltet die sogenannten *physikalischen Adressen*. Sie stellen den adressierbaren Speicher dar. Arbeitsspeicher und *adressierbarer Speicher* sind grundsätzlich zu unterscheiden, denn im Normalfall umfaßt der Adreßraum über die Adressen des Arbeitsspeichers hinaus sowohl I/O-Geräteadressen als auch schlicht ins Leere verweisende Adressen. Daß die Geräteregister der I/O-Geräte und der Arbeitsspeicher einen gemeinsamen Adreßraum erhalten, bezeichnet man als *Memory-Mapped-I/O*[7]. Die Größe des physikalischen Adreßraumes ist bei n Bit Breite des Adreßbusses zu 2^n Bit gegeben und der adressierbare Speicher errechnet sich bei byteweiser Adressierung gemäß

$$adressierbarer\ Speicher[Byte] = 2^{Adressbusbreite} * 1Byte$$

So konnte ein Intel 8086-Prozessor beispielsweise maximal nur 1 MB Speicher adressieren, da er lediglich über 20 Adreßleitungen verfügte. Aus Gründen der Kompatibilität hat die Firma Intel auf nachfolgenden Prozessoren den sogenannten Real Mode[8] eingeführt.

[7] Der Gegensatz hierzu ist I/O-Mapping, wobei neben dem Adreßraum für den Arbeitsspeicher ein weiterer für den Peripheriespeicher existiert. Bei letzterem spricht man nicht von Adressen, sondern von I/O-Ports. Durch ein zusätzliches Signal des Prozessors wird zwischen Arbeitsspeicherzugriff und Peripheriezugriff unterschieden. Dieses Signal wird bei jedem Zugriff auf den Arbeitsspeicher oder auf die Peripherie durch einen vorgeschalteten Adreßdecoder ausgewertet. Intelprozessoren sind für I/O-Mapping konzipiert, was die Möglichkeit zusätzlicher Adressierung durch Memory-Mapped-I/O nicht ausschließt. Zur Adressierung der I/O-Ports existieren die speziellen Instruktionen **in** und **out**.

[8] Manche Prozessoren anderer Hersteller kennen ebenso die Bezeichnung Real Mode. Dieser unterscheidet sich in der Ausführung jedoch von dem im folgenden beschriebenen Real Mode der Intel-80386-

2.3.2 Der Adreßraum im Realmode

Unter DOS arbeiten auch neuere Intel-Prozessoren in einem 20 Bit breiten Adreßraum, welcher von diesen Prozessoren im *Real Mode* emuliert wird [Ti92]. Zwar findet dieser Modus bei der Ausführung moderner Betriebssysteme keine Verwendung, jedoch bei deren Initialisierung, da sich alle Intel-80386-kompatiblen Architekturen nach dem Einschalten aus Kompatibilitätsgründen zunächst im Real Mode befinden.
Bei den 16 Bit breiten Prozessorregistern des 8086 waren 2 Register zur Adressierung nötig, da ein Register nur 2^{16} Speicheradressen und damit einen Bereich von 64KB (0x0000 bis 0xFFFF) erfassen konnte. Aus diesem Grunde unterteilte Intel den Speicher in als *Segmente* bezeichnete Abschnitte zu je 64 KB, welche fortlaufend numeriert wurden. Im Real Mode neuerer Intel-Prozessoren gibt es diese Segmente gleichermaßen numeriert. Die Segmentnummer wird in einem der *Segmentregister*[9] der CPU abgelegt. Die genaue Adressierung innerhalb eines Segments erfolgt durch Addition des Offsets (des Abstands vom Segmentbeginn), der in einem zweiten Register gespeichert wird. Dadurch entstehen die sogenannten *logischen Adressen* im Real Mode, welche üblicherweise in der Form *Segment:Offset* notiert werden[10]. Die einzelnen Segmente überschneiden sich und zwei Segmente sind um genau 16 Byte verschoben, sodaß im Real Mode insgesamt 1 MB Speicher adressierbar ist, und folgende Umrechnungsformel gilt[11]:

$$physikalische\ Adresse = Segmentnummer * 16 + Offset$$

Auch die Initialisierungsroutinen von Linux verwenden diese Adressierung, solange sich der Prozessor noch im Real Mode befindet. In diesem Modus kann auf die Routinen des BIOS[12]zugegriffen werden. Nach der frühen Startphase wechselt Linux jedoch in den Protected Mode. Detailliertere Erläuterungen zum Real Mode findet man [Ti92] und [Pot94].

2.3.3 Segmentierung im Protected Mode

Moderne Betriebssysteme arbeiten nicht im Real Mode, sondern im sogenannten *Protected Mode*, der den gesamten physikalischen Adreßraum[13] und Speicherschutzmechanismen zur Verfügung stellt. Intel-Prozessoren unterstützen in Kombination mit der

kompatiblen Architekturen.

[9]Dies sind unter anderem die Register code segment (cs) und data segment (ds), jenachdem ob das entsprechende Segment Instruktionen oder Daten beinhaltet.

[10]So zum Beispiel **cs:ip** für die nächte zu bearbeitende Instruktion im Speicher. Dabei ist ip der instruction pointer, der oft auch Programmzähler (program counter, pc) genannt wird.

[11]Man beachte die sich daraus ergebende Mehrdeutigkeit: Mehrere logische Adressen verweisen auf die gleiche physikalische Speicheradresse.

[12]**B**asic **I**nput **O**utput **S**ystem

[13]Bei den 32-Bit-Architekturen, d.h. 32 Bit Register und 32 Bit Adreßbusbreite, ergibt sich ein Adreßraum mit 2^{32} Adressen. Die byteweise adressierbare Speicherkapazität beträgt dann $2^{32} * 1Byte = 4$ Giga Byte. In der Praxis wird bei 32 Bit Datenbusbreite dieser entsprechend eine doppelwortweise Adressierung (4 Byte) verwendet und die letzten beiden Bit des Adreßbusses markieren das Byte innerhalb des Doppelworts. Die übrigen 30 Bit können dann ebenso eine Speicherkapazität von $2^{30} * 4Byte = 4$ Giga Byte adressieren.
Wie im Abschnitt 1.5 erläutert, setzt der herkömmliche Linuxkernel mindestens eine 32-Bit-Architektur voraus.

Einführung von 4 Privilegstufen[14] zweierlei Konzepte der Speicherverwaltung: Segmentierung und Seiteneinteilung [Sta01]. Bei der *Segmentierung* bekommen der Kernel und jeder Prozeß einen eigenen virtuellen Adreßraum zugeteilt, der auch hier als Segment bezeichnet wird. Diese Segmente enthalten jedoch nur einen Prozeß[15], und grundsätzlich kann auch nur dieser auf das entsprechende Segment zugreifen. Demzufolge dürfen sich die Segmente nicht überschneiden, und ihre Größe ist auch nicht mehr einheitlich, sondern variiert mit den Speichererfordernissen der einzelnen Prozesse. Den bei 0 beginnenden *virtuelle Adressen* innerhalb jedes Segments werden physikalische Speicheradressen zugewiesen. Diese Virtualisierung des prozeßinternen Adreßraums wird von einer Speicherverwaltungseinheit (MMU) unterstützt, die zwischen CPU und Adreßbus die sogenannten *logischen Adressen* der CPU in die *physikalischen Adressen* des Adreßbusses übersetzt. Jedem Segment wird eine der 4 Privilegstufen und ein als *Deskriptor* bezeichneter MMU-Registersatz zugeordnet, welcher unter anderem die physikalische Basisadresse und die Länge des Segments, sowie die Zugriffsrechte (Lesen, Schreiben, Ausführen) enthält. Durch das Addieren der Basisadresse zur virtuellen Adresse errechnet die MMU die physikalische Adresse

$$physikalischeAdresse = Basisadresse + virtuelleAdresse$$

Eine oder mehrere lokale Deskriptortabellen (**LDT**) enthalten die Segmentdeskriptoren der Prozesse, jeweils für deren Code und deren interne Daten. Daneben gibt es eine globale Deskriptortabelle (**GDT**) mit verschiedenen Deskriptoren des Kernels. Die Deskriptortabellen müssen bei der Systeminitialisierung im Arbeitsspeicher eingerichtet werden. Im Prozessor sind zwei spezielle Register **gdtr** und **ldtr** für die Adressen der GDT und der aktuellen LDT angelegt. So können diese an beliebiger Stelle im Arbeitsspeicher plaziert werden. Die logischen Adressen der CPU beinhalten jedoch nicht die Basisadressen der Segmente. Stattdessen bestehen sie aus einem *Segmentselektor* im entsprechenden Segmentregister und der virtuellen Adresse in einem anderen Register. Der Segmentselektor verweist auf den entsprechenden Deskriptor in der Deskriptortabelle. Bei der Ausführung eines Prozesses werden dessen Deskriptoren in der MMU aktiviert. Dies geschieht automatisch durch einen Mechanismus in der MMU, welcher bei jeder Neubelegung eines Segmentregisters mit einem Segmentselektor auch den durch diesen referenzierten Deskriptor lädt. Der Speicherschutz erfolgt dadurch, daß die MMU jeden Prozeß innerhalb seines Speichersegments eingrenzt und eine Instruktion, die über den virtuellen Adreßraums des Prozesses hinaus zugreifen möchte, sofort abbricht. So verhindert die MMU, daß ein Prozeß versehentlich durch einen anderen überschrieben oder absichtlich ausspioniert werden kann. Desweiteren werden durch die MMU die Prozesse im Arbeitsspeicher ortsunabhängig. Sie können dort an beliebigen physikalischen Adressen stehen. Lediglich dürfen sich die korrespondierenden Segmente dabei nicht überschneiden.

[14] Linux verwendet von den 4 Privilegstufen grundsätzlich nur 2, die als **Kernel Mode** und **User Mode** bezeichnet werden. Abschnitt 2.5 erläutert deren Verwendung unter Linux.

[15] Eine Ausnahme bilden Threads, das sind mehrere Kontrollflüsse innerhalb eines Prozesses, die sich einen virtuellen Adreßraum und damit auch das gleiche Segment teilen. Ihr Vorteil besteht darin, daß ein Threadwechsel ohne einen Wechsel des virtuellen Adreßraumes geschehen kann, was erheblich Zeit spart. Dafür ist der gegenseitige Speicherschutz für Threads nicht mehr gewährleistet. Dies muß vom Programmierer berücksichtigt werden.

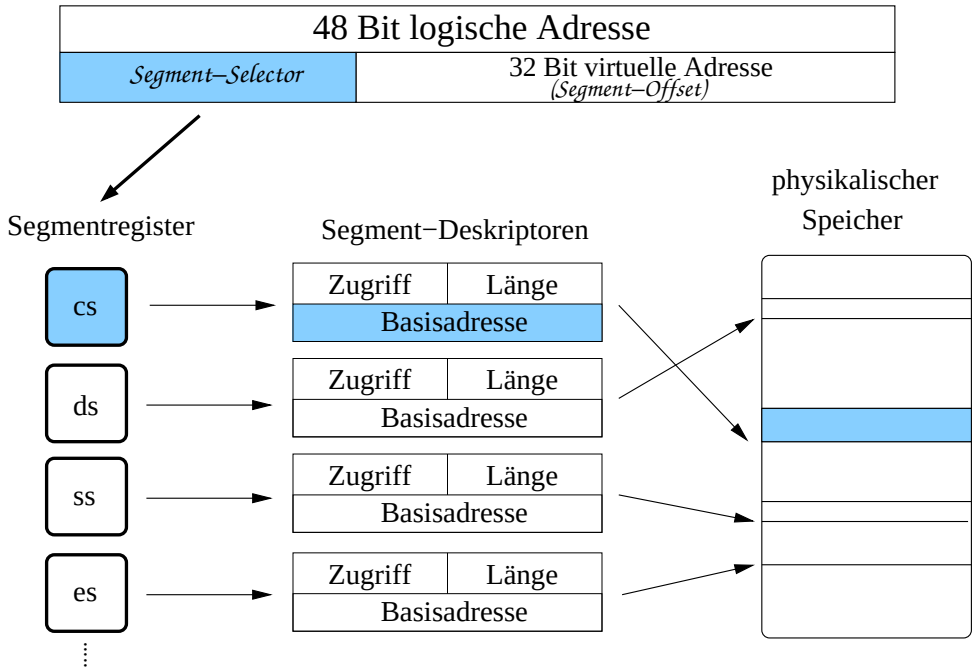


Abbildung 2.3: *Die Segmentierung beim Intel 80386. Die 48 Bit breite logische Adresse in der CPU setzt sich aus dem 16 Bit breiten Segmentselektor in einem jeweiligen Segmentregister und dem 32 Bit breiten Segment-Offset zusammen, welcher die im Prozeß verwendete virtuelle Adresse darstellt. Die Segmentselektoren verweisen auf den jeweiligen Segmentdeskriptor, der die physikalische Basisadresse und die Länge des Segments im Arbeitsspeicher beinhaltet.*

Die Abbildung 2.3 erläutert die Adreßübersetzung durch die MMU am Beispiel einer Instruktion. In der CPU befindet sich die 48 Bit breite logische Adresse, welche sich aus dem Segmentselektor des aktiven Prozesses und dessen virtueller Adresse als Segment-Offset zusammensetzt. Der Segmentselektor für die Instruktionen des aktiven Prozesses befindet sich Code-Segment-Register **cs**, die virtuellen Adressen der einzelnen Instruktionen stellen den jeweiligen Segment-Offset dar. Die Übersetzung in die physikalische Adresse verrichtet die MMU. Diese hat beim Laden des Segmentselektors den zugehörigen Segmentdeskriptor aus der entsprechenden Deskriptortabelle aktiviert. Daraus bezieht die MMU die Basisadresse des Segments und berechnet durch Addition des Segment-Offsets die physikalische Adresse der jeweiligen Instruktion. Dabei überprüft die MMU anhand des Eintrags für die Segmentlänge, ob die adressierte Instruktion tatsächlich innerhalb des Segments liegt. Ist dies nicht der Fall, so meldet die MMU einen Fehler[16].

Mit der Segmentierung gewährleistet die MMU Speicherschutz und ermöglicht die Relozierbarkeit von Prozessen. Die Anzahl und die Größe der startbaren Prozesse ist dabei jedoch durch die Kapazität des Arbeitsspeichers limitiert. Die Einschränkung bezüglich der Anzahl der Prozesse kann dadurch abgemildert werden, daß die Segmente

[16] Segmentation fault

inaktiver Prozesse auf externe Datenträger ausgelagert werden. Die Auslagerung eines Segments wird in den Segmentdeskriptoren vermerkt. Eine Auslagerung ganzer Prozeße bezeichnet man als *Swapping*[17].
Die Beschränkung der Segmentgröße einzelner Prozesse ist ein grundsätzliches Merkmal der Segmentierung und kann auch durch Prozeßauslagerung nicht aufgehoben werden, solange diese ganze Prozesse betrifft. Da Arbeitsspeicher eine begrenzte Ressource ist, sollte er möglichst effizient verwaltet werden. Im allgemeinen bleiben bei der Segmetierung jedoch größere Bereiche des Arbeitsspeichers ungenutzt. Durch das Starten und Beenden, wie durch die Ein- und Auslagerung von Prozessen entstehen zwischen den einzelnen Segmenten Freiräume. Im Laufe des Betriebs bewirkt dies eine zunehmende Fragmentierung des Arbeitsspeichers. Obwohl die Speicherkapazität all dieser ungenutzten Speicherfragmente unter Umständen für die Aufnahme mehrerer Segmente ausreichen würde, können sich die einzelnen Speicherfragmente als zu klein erweisen. Um der Fragmentierung entgegenzuwirken, müßten die Prozesse im Arbeitsspeicher regelmäßig verschoben und damit auch die Deskriptoren aktualisiert werden, was einen großen Verwaltungsaufwand erfordern würde. Stattdessen bieten Intel-MMUs Unterstützung für eine weitere Aufteilung der Segmente in einzelne Seiten, die in beliebigen Speicherbereichen untergebracht werden können.

2.3.4 Segmentierung in Kombination mit Seiteneinteilung

Um die Anforderung der Prozesse nach zusammenhängendem Arbeitsspeicher aufzuheben, wird dieser in gleich große Seitenrahmen (Page Frames) und der virtuelle Adreßraum der Prozesse in ebenso große Seiten (Pages) unterteilt. Von den Seiten müssen sich nur diejenigen in einem Seitenrahmen des Arbeitsspeichers befinden, auf die in einem engeren Zeitraum zugegriffen wird. Die Auslagerung der übrigen Seiten wird dann nicht mehr als Swapping, sondern als *Paging*[18] bezeichnet. Damit können auch Teile eines Prozesses auf Sekundärspeicher ausgelagert werden, wodurch sogar Prozesse ermöglicht werden, deren Speicherbedarf über die Größe des Arbeitsspeichers hinaus geht. Die Adreßübersetzung in der MMU besteht dabei aus zwei Schritten. Zuerst wird die *logische Adresse* mittels des jeweiligen Deskriptors in eine *lineare Adresse* übersetzt. Intel verwendet für den 32 Bit breiten Adreßraum eine Seitengröße von 4 KB. Die lineare Adresse besteht aus einer 20 Bit großen *Page-ID*, gefolgt von der Adresse innerhalb der Seite. Anhand dieser Page-ID wird die lineare Adresse mittels verschiedener *Page Tables* in die physikalische Adresse übersetzt. Die Page-Tables weisen den Seiten die entsprechenden Seitenrahmen zu und befinden sich im Speicher.
Um die Übersetzung zu beschleunigen und einen doppelten Speicherzugriff zu vermeiden, verfügen Intel-Prozessoren seit dem 80386 über einen speziellen Cache für die Einträge der Page Tables, den Translation Lookaside Buffer (TLB). Der TLB ist dem allgemeinen Cache vorgelagert, sodaß letzterer die physikalischen Adressen beinhaltet. Die untere Abbildung skizziert die beiden Stufen der Adreßübersetzung bei einer Intel-MMU durch die Segmentation-Unit und die Paging-Unit. Die Seiteneinträge der Page Tables entspre-

[17] Da eingebettete Systeme in der Regel über keine beschreibbaren externen Datenträger verfügen, fehlt die Möglichkeit für ein Swapping und die Limitierung der Prozesse bleibt durch die Größe des Arbeitsspeichers vorgegeben.

[18] Paging bezeichnet meistens die Seiteneinteilung in Kombination mit deren Auslagerung.

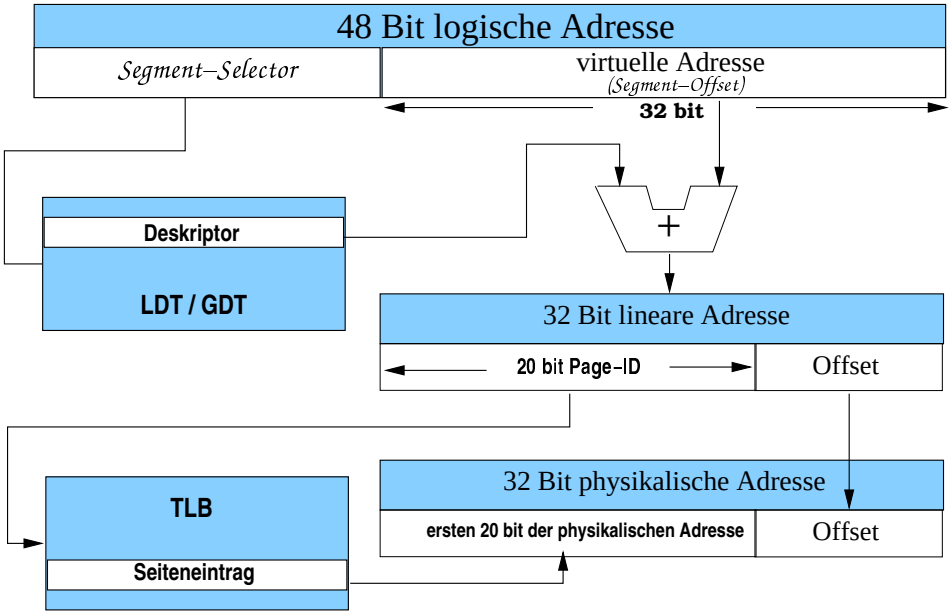


Abbildung 2.4: *Segmentierung in Kombination mit Seiteneinteilung. Hierbei besteht die MMU aus einer Segmentation-Unit und einer Paging-Unit. Die Segmentation-Unit wandelt die 48 Bit breite logische Adresse der CPU, wie zuvor erläutert, in eine 32 Bit breite Adresse um. Diese ist jedoch nicht die physikalische Adresse im Arbeitsspeicher, sondern die sogenannte lineare Adresse innerhalb eines Segments. Erst die Paging-Unit wandelt diese lineare Adresse in die physikalische Adresse des Arbeitsspeichers um. Dabei verwendet sie einen Cache, den TLB, in welchem Einträge der Seitentabellen zum schnelleren Zugriff gespeichert werden können.*

chen den Segmentdeskriptoren und beinhalten neben der physikalischen Adresse ebenso Zugriffsrechte und die Information, ob sich die Seite im Arbeitsspeicher befindet oder ausgelagert wurde.

Auch das Paging ist auf einem eingebetteten System ohne externe Datenträger nicht möglich, dennoch kann aber eine Seiteneinteilung vorgenommen werden, wenn dies durch eine im System vorhandene MMU unterstützt wird. Segmetierung und Seiteneinteilung dienen schließlich nicht nur der Prozeßauslagerung, sondern in erster Linie der dynamischen und flexiblen Verwaltung verschiedener Prozesse im Speicher, sowie deren gegenseitigem Speicherschutz. Wenn keine Möglichkeit der Prozeßauslagerung besteht, ist die effiziente Nutzung des Arbeitsspeichers umso wichtiger.

2.3.5 Die Speicherverwaltung unter Linux

Die eben beschriebene Kombination aus Segmentierung und Paging birgt eine gewisse Redundanz und Umständlichkeit in sich. Dies ist der Grund dafür, daß die MMUs mancher Prozessoren vorwiegend oder ausschließlich Paging unterstützen, bei dem die Fragmentierung des Arbeitsspeichers prozeßintern, überschaubarer und im allgemeinen

geringer bleibt[19] [Sta01]. Paging ist deshalb nicht nur das effizientere Konzept der Speicherverwaltung, sondern gewährleistet infolge der breiteren Hardwareunterstützung auch eine höhere Portabilität. Aus diesen Gründen verzichtet Linux weitgehend auf Segmentierung zugunsten eines architekturunabhängigen Speichermodells [BC01]. Ermöglicht die Hardware Segmentierung, wie bei Intel-Prozessoren, so werden nur wenige Segmente angelegt und deren Deskriptoren bei der Kernelinitialisierung in der GDT abgelegt. Unter anderem sind dies jeweils ein Daten- und ein Codesegment, zum einen für den Kernel zum anderen für Benutzerprozesse. Sie überschneiden sich größtenteils, sodaß man im eigentlichen Sinne nicht von Segmentierung sprechen kann. Die einzelnen Prozesse liegen dadurch im gleichen linearen Adreßraum. Die tatsächliche Speicherverwaltung wird innerhalb dieser Segmente auf Paging basierend durchgeführt[20].
Beim Einrichten der Seiten muß folgender Sachverhalt ins Auge gefaßt werden: Mit wachsender Seitengröße steigt die Speicherfragmentierung an, mit fallender Seitengröße wächst die Anzahl der Speicherseiten und damit der Speicherbedarf und Verwaltungsaufwand für die Page-Tables. Zudem kann der Datenverkehr mit Blockgeräten vereinfacht werden, wenn die Seitengröße ein Vielfaches der üblichen Blockgrößen[21] darstellt. Für eine feste Anzahl von Blöcken wird dann jeweils genau eine Seite allokiert. Bei 32 Bit Architekturen sind 4 KB (20 Bit Page-ID und 12 Bit Offset), bei 64 Bit Architekturen 8 KB Seitengröße (13 Bit Offset) üblich. Im letzten Falle werden nicht alle restlichen 51 Bit zur Adressierung verwendet, da ansonsten 2^{51}Page-IDs zu verwalten wären. Stattdessen wird der potentielle Adreßraum in kleinerem Umfang ausgenutzt. Linux verwaltet die Page-IDs in drei Tabellen, einer Page Global Directory mit Verweisen auf eine Page Middle Directory, die ihrerseits Verweise auf die entsprechende Page Table beinhaltet. Dies bezeichnet man als dreistufiges oder *Three-Level Paging*.
Die MMUs der 32 Bit Architekturen von Intel unterstützen nur ein zweistufiges Paging (*Two-Level Paging*). In diesem Falle werden die Page Middle Directorys jeweils nur mit einem Eintrag gefüllt, jedoch in der Übersetzungssequenz mitgeführt, um den Code für Two-Level Paging und Three-Level Paging kompatibel zu halten. Beim Intel Pentium werden die höchsten 10 Bit der linearen Adresse in der Page Global Directory selektiert, deren Einträge auf eine entsprechende Page Table verweisen. Aus dieser werden anhand der unteren 10 Bit der Page-ID die entsprechenden physikalischen Adressen der einzelnen Seiten im Speicher ermittelt. Die beiden Tabellen haben jeweils 1024 Einträge. Wird eine lineare Adresse zum ersten mal angesprochen, so erfolgt die Übersetzung in die physikalische Adresse über den langsamen Zugriff auf die entsprechenden Page-Tables

[19]Bei der Segmentierung tritt das Phänomen der externen Fragmentierung auf, d.h. die unverwertbaren Speicherfragmente liegen außerhalb der Adreßräume der Prozesse und ihre Größe variiert. Bei reiner Seiteneinteilung ist die Fragmentierung ausschließlich intern und dadurch gegeben, daß ein allokierter Seitenrahmen nicht vollständig aufgefüllt ist. Die ungenutzen internen Speicherfragmente können nie größer als eine Seitengröße sein und bleiben auf die einzelnen Prozesse beschränkt. Zur Defragmentierung ist keine Relozierung der Prozesse nötig.

[20]Unter Linux wird der Kernelprogrammierer demnach niemals (außer bei der Systeminitialisierung) mit logischen Adressen konfrontiert. Dennoch wird in manchen Büchern die Bezeichnung logische Kerneladressen verwendet, welche von den viruellen Kerneladressen unterschieden werden. In beiden Fällen sind damit lineare Adressen im Kernelsegment gemeint. Die logischen Kerneladressen zeichnen sich lediglich dadurch aus, daß sie beim Paging bis auf einen konstanten Offset auf die physikalischen Adressen abgebildet werden. Solche Adressen lassen sich mit *kmalloc*() allokieren, nicht jedoch mit vmalloc() oder kmap(). Letztere Allokierung gewährleistet keine direkte Abbildung auf den physikalischen Speicher.

[21]Die übliche Blockgröße für Festplatten beträgt deren Sektorgröße entsprechend 512 Byte.

im Arbeitsspeicher. Anschließend wird die so ermittelte physikalische Adresse jedoch im TLB gespeichert, sodaß der nächste Zugriff schneller erfolgt. Änderungen an den Page-Tables sind deswegen mit einem Zeitverlust verbunden, weil die betroffenen Einträge im TLB daraufhin wieder erneut aus dem Arbeitsspeicher eingelesen und synchronisiert werden müssen.

Bei Linux bekommt jeder Prozeß seine eigene Page Global Directory und demnach auch seine eigenen Page Tables. Die physikalische Adresse der aktuellen Page Global Directory wird bei den 32-Bit-Architekturen von Intel im Register cr3 abgelegt. Bei jedem Prozeßwechsel wird dieser Registereintrag geändert. Dadurch gewährleistet das Paging, daß Prozesse im unprivilegierten Modus nur auf den ihnen zugewiesenen Speicher zugreifen können. Dies gilt für alle Benutzerprozesse, bei denen der Prozessor in der untersten Privilegstufe arbeitet, die unter Linux als User Mode bezeichnet wird. Jeder Prozeß besitzt einen Speicherdeskriptor (memory descriptor), der Teil seines Prozeßdeskriptors ist. Darin befindet sich unter anderm ein Zeigereintrag **pgd** auf die Page Global Directory des Prozesses.
Theoretisch könnte auf diese Weise jedem Benutzerprozeß ein virtueller Adreßraum von 4 GB mit den linearen Adressen von 0x00000000 bis 0xFFFFFFFF zugeteilt werden. Linux beschränkt den im User Mode verfügbaren virtuellen Adreßraum jedoch. So werden den Benutzerprozessen auf dem Intel Pentium beispielsweise nur 3 GB zur Verfügung gestellt, welche in Anlehnung an den Segmentbegriff der Intel-MMUs als Nutzersegment bezeichnet werden. Das übrige Giga Byte des Kernelsegments ist nur im privilegierten Modus (Kernel Mode) verfügbar und liegt bei allen Prozessen in denselben Page Tables. Von dieser Beschränkung abgesehen nimmt Linux keine weitere Unterscheidung zwischen virtuellen und linearen Adressen vor. In der Linuxliteratur werden die beiden Begriffe deshalb gelegentlich auch synonym gebraucht.

2.4 Ausführbare Programme

Ein ausführbares Programm ist im einfachsten Falle lediglich eine Hintereinanderreihung von Instruktionen für den Prozessor. Beginnt diese Instruktionsfolge im ersten Sektor der als Bootmedium bestimmten Festplatte, so wird sie beim Einschalten eines Personalcomputers automatisch gestartet[22]. Doch der Start eines derartigen Programms erfordert bereits rudimentäre Betriebsfunktionen, welche in der Lage sind, das externe Speichergerät anzusprechen, die Instruktionen aus dessen Bootsektor in den Arbeitsspeicher zu laden und den Programmzähler[23] auf die Startadresse der Instruktionsfolge im Arbeitsspeicher zu setzen. Beim PC ist diese elementare Startroutine, wie in Anhang D erläutert, im BIOS implementiert.

Im Gegensatz dazu stellen Betriebssysteme eine Umgebung bereit, die es ermöglicht,

[22] Es ist jedoch zu beachten, daß selbst das BIOS eine einfache Formatierung in Form einer sogenannten *magic number* im Bootsektor erwartet und nach einer erfolglosen Überprüfung mit einer Fehlermeldung abbricht.

[23] Der Programmzähler ist das Register im Prozessor, welches die Adresse der jeweils als nächstes zu bearbeitenden Instruktion beinhaltet.

beliebige Programme zu starten. Das Betriebssystem selbst beinhaltet einen *Lader*, welcher diese Programme in den Speicher lädt und zur Ausführung für die CPU präpariert. Dabei muß das Betriebssystem einen neuen Prozeß erzeugen. Wie im vorangegangenen Abschnitt erläutert, wird jedem Prozeß ein eigener virtueller Speicherbereich eingeräumt. Darüberhinaus wird der Prozeß durch einen Prozeßdeskriptor in der Prozeßtabelle registriert, worin unter anderem Daten über die Laufzeit des Prozesses, den Benutzer und daraus folgende Rechte und Prioritäten vermerkt sind.

Unterschiedliche Programme beinhalten teilweise identische Daten und Programmabschnitte. Eine effektive Speicherausnutzung muß die daraus folgende Redundanz bei der Belegung von Arbeitsspeicher minimieren. Zu diesem Zweck unterstützt Linux modularisierte Programme auf der Grundlage einer dynamischen Adreßbindung zur Laufzeit. Diese Programme unterscheiden sich jedoch von den oben angeführten Instruktionssequenzen durch eine bestimmte Struktur, die als *Binärformat* bezeichnet wird. Sie sind ausführbare Programme eines Betriebssystems und im allgemeinen ohne ein solches gar nicht lauffähig. Ihre Ausführung erfordert die dynamische Adreßbindung durch einen Programminterpreter, dessen genaue Funktionalität im Zusammenhang mit der modularen Programmgenerierung ersichtlich wird.

2.4.1 Erzeugung von Programmen

Ausführbare Programme werden für gewöhnlich nicht direkt in Maschinensprache, sondern in einer höheren Programmiersprache entwickelt. Auf Unixderivaten, wie Linux, ist dies in den meisten Fällen die Sprache C, da Unixsysteme selbst in C geschrieben sind und diese Sprache mit einer Vielzahl verschiedener Dateien umfangreich unterstützen. Der in einer höheren Programmiersprache geschriebene Text wird üblicherweise als Quellcode bezeichnet. Ein *Compiler* übersetzt (kompiliert) Dateien mit diesem Quellcode entweder direkt in Maschinencode (Maschinensprache) oder zunächst in Assemblercode. *Assembler* bezeichnet sowohl eine mnemonische Darstellung von Maschinencode, als auch das Programm, welches diese Darstellung in reinen Maschinencode umwandelt. Der Compiler generiert Maschinencode entweder selbständig oder über den Aufruf eines Assemblers. Die Datei mit dem vom Compiler generierten Maschinencode wird als *Objektmodul* oder Objektdatei (*Object File*) bezeichnet. Einzelne Objektmodule sind ansich noch keine ausführbaren Programme. Sie benötigen weitere Objektmodule mit den Initialisierungs- und Terminierungroutinen eines ausführbaren Programms. Ein *Binder* (*Linker*) fügt alle erforderlichen Objektmodule zu einem ausführbaren Programm zusammen, welches von dem Lader des Betriebssystems zur Ausführung in den Arbeitsspeicher geladen werden kann. Spätestens bei der Ausführung des Programms müssen die Adressen symbolischer Namen von Variablen oder Sprungzielen aus dem Quelltext des Programms durch die jeweiligen absoluten Adressen ersetzt werden, an denen die entsprechenden Variablen oder die adressierten Instruktionen im Speicher zu finden sind. Diese *Adreßbindung* kann an unterschiedlichen Stellen vorgenommen werden:

1. Zur Übersetzungszeit durch den Compiler

2. Zur Bindezeit durch den Linker

3. Zur Ladezeit durch den Lader des Betriebssystems

4. Dynamisch zur Laufzeit durch das Betriebssystem

Eine Adreßbindung zur Übersetzungszeit durch den Compiler ist für modulare Programmierung ungeeignet, da der Linker die einzelnen Module nicht mehr sinnvoll zusammenfügen kann. Darum geschieht die Adreßbindung unter Linux frühestens zur Bindezeit durch den Linker. Die stets bei Null beginnenden relativen Adressen der ungebundenen Objektmodule entsprechen nicht den endgültigen linearen Adressen des ausführbaren Programms. Erst die gebundenen, ausführbaren Dateien besitzen lineare Adressen, welche bei ihrer Ausführung durch die Speicherverwaltung des Kernels mit Hilfe der MMU in die physikalischen Adressen übersetzt werden. Sind nach dem Binden alle linearen Adressen vollständig aufgelöst und alle Programmteile in der ausführbaren Datei vorhanden, so spricht man von statischer Programmbindung oder einfach von *statischem Binden*. Beim *dynamischen Binden* befinden sich hingegen nicht alle Module in der ausführbaren Datei. Stattdessen werden externe Sprungziele und Variablen lediglich durch Marken (sogenannte Stubs) mit Verweisen auf diese Module referenziert. Die vollständige Adreßbindung kann entweder beim Programmstart durch einem Lader oder dynamisch zur Laufzeit durch das Betriebssystem erfolgen. Der Vorteil der dynamischen Adreßbindung gegenüber einer Adreßbindung zur Ladezeit besteht darin, daß Objektmodule nur bei effektivem Bedarf eingebunden werden, der beim Programmstart nicht immer entscheidbar ist. Dynamische Adreßbindung setzt einen Programminterpreter oder *Runtime-Linker* voraus, der benötigte Module in den Speicher lädt und die vollständige Adreßauflösung zur Laufzeit der Programme organisiert. Dazu sind Konventionen über die Anordnung der einzelnen Module eines ausführbaren Programmes innerhalb der Dateien notwendig, welche im *Binärformat* festgelegt werden. Die vom Binder erzeugten Objektdateien und der Runtime-Linker müssen auf einem kompatiblen Binärformat basieren. Linux verwendet das *Executable and Linking Format* (*ELF*).
In diesem Format bestehen ausführbare Dateien und Objektdateien aus einem Kopf gefolgt von verschiedenen Sektionen. Der Kopf beinhaltet unter anderem Informationen über die Eigenschaften des Codes, dessen Relozierbarkeit und die virtuelle Adresse des Einsprungspunkts. Eine Symboltabelle beinhaltet alle symbolischen Namen von Variablen oder Sprungzielen. Die Unterteilung in Sektionen erfolgt nach inhaltlichen Aspekten. Es gibt unter anderem Sektionen für den Programmcode, für initialisierte und uninitialisierte Programmdaten, sowie Sektionen für den Initialisierungscode und den Terminierungscode von Programmen. Hinsichtlich einer Dokumentation des *Executable and Linking Format* wird auf [TIS] verwiesen.
In Linuxsystemen befindet sich der ELF-Runtime-Linker standardmäßig im Verzeichnis /lib mit dem Namen ld-X.so, wobei X die Versionsnummer angibt. Da der Runtime-Linker somit kein Bestandteil des Kernels ist, können unter Linux prinzipiell verschiedene Binärformate verwendet werden. Für die im nächsten Kapitel erläuterte Kompilierung des Kernels ist zu beachten, daß der Kernel selbst im Executable and Linking *Format* vorliegt. Dadurch kann er zur Laufzeit Module dynamisch einbinden.

2.4.2 Dynamisches Binden und Shared Libraries

Es gibt Objektmodule die immer wieder in bestimmten Zusammenhängen benötigt und darum in *Bibliotheken* zusammenfaßt werden. Dies sind einerseits Bibliotheken für mathematische, graphische oder sonstige spezifische Anwendungen, aber auch Bibliotheken mit dem Objektcode für die Programminitialisierung und Programmterminierung, welche in jedes ausführbare Programm gebunden werden müssen. Beim statischen Binden befindet sich demzufolge ein und derselbe Objektcode in verschiedenen ausführbaren Programmen vielfach im Dateisystem und bei parallel ablaufenden Programmen sogar mehrfach im Arbeitsspeicher. Um derartige Redundanz zu vermeiden, führen moderne Betriebssysteme dynamisch gebundene Bibliotheken als sogenannte *Shared Libraries* ein. Diese sind Objektmodule, welche erst zur Laufzeit des sie benötigenden Programms dynamisch gebunden werden. Der enthaltende Maschinencode befindet sich nur einmal[24]im Dateisystem und kann von beliebig vielen Programmen verwendet werden. Dadurch beanspruchen Betriebssystem und Benutzerprogramme weniger Speicherplatz auf der Festplatte. Weitaus entscheidender ist jedoch der Bedarf an Arbeitsspeicher. Wenn mehrere laufende Prozesse dieselbe Bibliothek verwenden, ist es ineffizient, den gleichen Maschinencode innerhalb mehrere Segmente zu laden. Darum werden Shared Libraries nur einmal in den Arbeitsspeicher geladen und stehen von diesem Zeitpunkt an allen Prozessen lesbar zur Verfügung. Die dynamische Adreßauflösung durch die Speicherverwaltung ermöglicht das Einblenden physikalischer Adressen in den virtuellen Adreßraum mehrerer Benutzersegmente. Neben dem Vorteil des geringeren Speicherbedarfs ergibt sich bei mehrfacher Verwendung eine kürzere Ladezeit, denn der allgemein verfügbare Maschinencode muß nur einmal von einem Prozeß in den Arbeitsspeicher geladen werden und steht daraufhin allen anderen Prozessen zur Verfügung.

Haben dynamisch gebundene Bibliotheken einen großen Umfang, so besteht die Wahrscheinlichkeit, daß diese das Dateisystem mit überflüssigen Bibliotheksfunktionen füllen, die von keinem der je laufenden Prozesse genutzt werden. Da viele eingebettete Systemen über sehr knapp bemessene Speicherressourcen verfügen, sollten Größe und Inhalt der verwendeten Shared Libraries für jene ressourcenarme Systeme besonders zweckmäßig gewählt werden. In kritischen Fällen kann der geringe Speicherplatz ein statisches Binden der Dateien erzwingen.

2.5 Unterbrechungen und Systemaufrufe

Im Abschnitt 2.3 wurden die beiden von Linux verwendeten Privilegstufen User Mode und Kernel Mode in Zusammenhang mit der Speicherverwaltung erwähnt. Während die Kernelroutinen im Kernel Mode laufen und damit auf den gesamten Speicher, sowie auf alle Peripheriegeräte uneingeschränkten Zugriff haben, arbeiten die Benutzerprozesse grundsätzlich im User Mode. In diesem ist ein Speicherzugriff ausschließlich auf zugewiesene Adreßbereiche erlaubt. Jeder Zugriff auf prozeßexterne Speicheradressen oder Peripheriegeräte wird durch die MMU unterbunden. Dadurch entscheidet der Kernel über die Zuteilung aller Ressourcen an die einzelnen Prozesse. Auch die Aktivierung

[24]als shared object file mit der Endung **.so** bei Linux im Verzeichnis /lib

einzelner Prozesse durch die Zuweisung des Prozessors wird vom Kernel organisiert. Nebenläufigkeit auf einem Betriebssystem oder Multitasking bedeutet im engeren Sinne nämlich nicht nur, daß mehrere Prozesse virtuell gleichzeitig laufen, sondern verlangt darüberhinaus *preemptiv* organisierte Benutzerprozesse. Preemptiv heißt, daß der jeweils aktive Prozess in regelmäßigen Zeitabständen durch einen sogenannten *Scheduler* unterbrochen wird[25]. Der Scheduler ist Bestandteil des Linuxkernels und entscheidet, welchem Prozess für die nächste Zeiteinheit der Prozessor zugeteilt wird.
Da Benutzerprozesse im User Mode sehr eingeschränkte Möglichkeiten haben, müssen sie immer wieder auf die Dienste des Kernels zurückgreifen. Dies geschieht durch *Systemaufrufe*. Der Prozessor schaltet dabei in den Kernel Mode und nach einer Überprüfung der Rechte bedient die jeweils aufgerufene Kernelroutine den sie aufrufenden Prozeß. Es ist üblich, vom Übergang des Prozesses aus dem User Mode in den Kernel Mode zu sprechen. Da durch die Speicherverwaltung erzwungen wird, daß jeder Benutzerprozeß nur in einem vom Kernel zugeteilten Adreßbereich operieren kann, ist es keinem Benutzerprozeß möglich, in eine auf Speichermedien liegende Datei zu schreiben oder daraus zu lesen, was ebenso für die Gerätedateien gilt. Der Zugriff auf Dateien und Geräte ist nur dem Kernel möglich und muß demzufolge bei allen Benutzerprozessen über einen Systemaufruf erfolgen, sodaß der Kernel unumgänglich über die Einhaltung aller Zugriffsbestimmungen wacht.

Wie jedes Multitasking-Betriebssystem arbeitet auch Linux mit Unterbrechungen. Bei Systemen mit interaktiver Benutzerschnittstelle ist es sehr häufig, daß ein Prozeß auf ein bestimmtes Ereignis wartet (z.B. Tastendruck). Eine *zyklische Abfrage* (busy waiting oder *polling*) würde den Prozessor bis zum Eintreffen des Ereignisses belasten. Bei unterbrechungsgetriebenen Betriebssystemen wird der wartende Prozeß deaktiviert. Der Scheduler übergibt den Prozessor an andere Prozesse. Wenn das erwartete Ereignis eingetreten ist, wird der Prozessor durch einen Unterbrechungsaufruf (*interrupt request, IRQ*) darüber informiert und kann bei einem der folgenden Schedulings den wartenden Prozeß wieder aktivieren.
Unterbrechungsaufrufe werden vom Betriebssystem durch spezielle Behandlungsroutinen (*interrupt service routines*, *ISR*) bearbeitet, welche wie die Behandlungsroutinen der Systemaufrufe im Kernel Mode ausgeführt. Verschiedene Unterbrechungsaufrufe werden mit einer individuellen Unterbrechungsnummer (IRQ number) versehen, für den die von Intel geprägte Bezeichnung *Interrupt-Vektor* geläufig ist, und anhand derer die entsprechende Behandlungsroutine zugewiesen wird. Im Gegensatz zu den Prozessen, laufen diese in keinem eigenen Prozeßumfeld, ihnen wird auch kein Eintrag in der Prozeßtabelle zugewiesen. Sie sind daher leichtgewichtig, d.h. schneller und ohne größeren Verwaltungsaufwand eingerichtet. Für die ISR gibt es einen eigenen Deskriptor und eine

[25] Das eigentliche Unterbrechungssignal ist ein Hardware-Interrupt (weiter unten erläutert). Dieser kommt vom Zeitgeber (*Timerinterrupt*) und bewirkt den Aufruf des Schedulers. Ohne ein *Timerinterrupt* ist kein preemtives Multitasking möglich. Benutzerprozesse sind unter Linux grundsätzlich preemptiv, man spricht in diesem Falle von echtem oder preemptiven Multitasking. Im Gegensatz dazu steht das kooperative Multitasking, bei dem die Prozesse (wie bei Windows 3.x) selbst entscheiden, wann sie den Prozessor abgeben. Weil im herkömmlichen Linux manche Routinen im Kernel Mode nicht preemptiv laufen, ist Linux nicht echtzeitfähig. Es gibt sogenannte *critical regions* im Kernel, bei denen alle Unterbrechungen abgeschaltet werden.

eigene Deskriptortabelle (*interrupt descriptor table, IDT*), in der für jeden Interrupt-Vektor die Speicheradresse der jeweiligen Behandlungsroutine (ISR) vermerkt ist. Von modernen Prozessoren wird dies durch ein Register **idtr** mit der Speicheradresse der IDT unterstützt, sodaß dieser an beliebiger Stelle im Speicher plaziert werden kann. Wenn ein Unterbrechungsaufruf ankommt, unterbricht der Prozesor den aktiven Prozeß, speichert zur späteren Restauration notwendige Daten, wie den Programmzähler und das Code-Segment-Register, in einem *Kernelstack*[26] und lädt in den Programmzähler die Adresse, auf die der entsprechende Interrupt-Vektor in der IDT verweist. Nach Beendigung der ISR werden die gespeicherten Register des unterbrochenen Prozesses vom Kernelstack genommen und der vorangegangene Zustand restauriert. Der unterbrochene Prozeß kann seine Arbeit fortsetzen.

Prozessorintern wird zwischen *Interrupts* und *Exceptions* differenziert. In diesem Sinne werden nur die durch die Peripheriegeräte ausgelösten asynchronen Unterbrechungen (*Hardware-Interrupts*) als Interrupts verstanden. Synchron mit der Instruktionsverarbeitung als deren Resultat hervorgerufene Unterbrechungen werden als Exceptions bezeichnet. Solche sind entweder infolge eines außergewöhnlichen Zustands der CPU (Beispiel: division by zero flag) indirekt oder durch den Aufruf **int** direkt verursacht. Weil der Assemblerbefehl **int** Teil eines Programms ist, spricht man im letzten Fall auch von programmierten Exceptions oder *Software-Interrupts*. Diese verwendet das Betriebssystem für die Systemaufrufe, welche damit im Prozessor als Exceptions umgesetzt sind. Die Systemaufrufe in dieser Weise zu implementieren hat den Vorteil, die Mechanismen die der Prozessor für Unterbrechnungen bereitstellt, auch für die Systemaufrufe ausnutzen zu können.

Wie im Abschnitt 2.2.1 bereits erwähnt, werden in Unixsystemen Geräte als Dateien abgebildet. Darum nehmen die Sytemaufrufe der Dateiverwaltung beim Zugriff auf diese Geräte eine zentrale Stellung ein. Sie sind unter Linux die oberste Schnittstelle zwischen den Benutzerprozessen und den Treiberfunktionen der Peripheriegeräte. Zwischen beiden liegen mehrere Softwareschichten. Jeder Gerätezugriff kann nur über einen Systemaufruf erfolgen. Der Gerätetreiber muß diese Systemaufrufe kennen, um sie mit einer bestimmten Methode zu bedienen. Wegen der oftmals speziellen Peripherie eingebetteter Systeme, erfordert dort die Betriebnahme von Geräten und Schnittstellen gelegentlich individuelle Gerätetreiber. Dies gilt auch für die in Abschnitt 5.2 genauer beschriebene Schnittstelle zur Slow-Controll. Im folgenden wird der Zusammenhang der Gerätetreiber mit den oberen Softwareschichten erläutert. Die Schnittstellen zur Hardware sind das Thema des Abschnitts 2.6. Beide zusammen ergeben den Rahmen in dem sich ein Gerätetreiber befindet.

2.5.1 Systemaufrufe der Dateiverwaltung

Seitens der Systemaufrufe wird die Initialisierung des Dateizugriffs als das Öffnen einer Datei dargestellt. Jede weitere Dateioperation setzt eine geöffnete Datei voraus. Ein Prozeß öffnet eine Datei mit dem impliziten Systemaufruf *open()*:

[26] Beim Sprung in eine Kernelroutine müssen verschiedene Register gesichert werden, um diese beim Rücksprung wieder herstellen zu können. Der Kernel verwaltet Daten in einem Stapel (*Stack*)

file_deskriptor = *open*(Pfadname, Zugriffsart, Zugriffsrechte)

Der Systemaufruf erzeugt dabei ein offenes *Dateiobjekt* als kernelinterne Datenstruktur für die Verwaltungsdaten des jeweiligen Gerätetreibers. Dem aufrufenden Prozeß wird ein *Dateideskriptor (file descriptor)* zurückgegeben, welcher trotz dieser Bezeichnung in keinem Zusammenhang zu den in den vergangenen Abschnitten erläuterten Deskriptoren des Kernels steht. Um eine neue Datei anzulegen, kann auch der Aufruf *create()* anstelle von *open()* verwendet werden. Beendet wird der Dateizugriff mit *close()*.

Die Dateiverwaltung umfaßt mehrere Schichten. Die oberste Schicht ist der Benutzerprozeß, der durch einen Funktionsaufruf eine sogenannte *Wrapper-Routine* zum eigentlichen Systemaufruf veranlaßt. Die Wrapper-Routinen sind die Implementierungen der aufgerufenen Funktionen und befinden sich in der Standard C-Bibliothek. Durch das Verstecken der Wrapper-Routinen in der Bibliothek wird der C-Quellcode systemunabhängig. Auf einem anderen Betriebssystem muß lediglich die entsprechende Bibliothek ausgetauscht werden. In der Wrapper-Routine befindet sich der eigentliche Systemaufruf durch den Sofware-Interrupt int 0x80. Dadurch wechselt der Benutzerprozeß in den Kernel Mode. Im Kernel bedient die Behandlungsroutine der Systemaufrufe (system call handler) die Anfrage, indem sie eine entsprechende Serviceroutine (system call service routine) aufruft. Die Serviceroutinen der oben vorgestellten Systemaufrufe sind in der Datei */usr/src/linux/fs/open.c* zu finden und heißen dort *sys_open()*, *sys_close()* und *sys_creat()*. Alle Systemaufrufe, die eine Pfadangabe beinhalten verwenden zudem die Funktion *namei*() in */usr/src/linux/fs/namei.c*. Diese Funktion sucht die mit dem Pfadnamen korrespondierende Inode im Dateisystem.

Bei einer Gerätedatei rufen die Serviceroutinen die entsprechenden Treiberfunktionen auf. Die Kommunikation mit einem Gerät besteht im wesentlichen darin, daß Daten in das Gerät geschrieben werden und Daten aus dem Gerät gelesen werden. Zum Lesen und Schreiben sind die Systemaufrufe *read*() und *write*() vorgesehen. Sie werden von den Serviceroutinen *sys_read*() und *sys_write*() in */usr/src/linux/fs/read_write.c* abgefangen, welche ihrerseits die jeweiligen Treiberfunktionen aufrufen. Beide Systemaufrufe übergeben einem Dateideskriptors als Parameter und bekommen die Anzahl erfolgreich übertragener Bytes als Rückgabewert oder -1, wenn ein Fehler signalisiert wird. Weitere Parameter sind eine Puffervariable, in die eingelesen, bzw. aus der heraus geschrieben wird und die Anzahl der Bytes, die geschrieben oder eingelesen werden sollen.
Die Einheitliche Darstellung unterschiedlicher Hardware kann anhand des Kopierens einer herkömmlichen Datei in eine Gerätedatei veranschaulicht werden:

cp *testdatei* /dev/lp0

Die Struktur des zugrundeliegenden C-Quelltextes kann unter Vernachlässigung der Fehlerprüfung wie folgt skizziert werden:

```
quellfd = open("testdatei", O_RDONLY, QuellZugriff);
zielfd  = open("/dev/lp0", O_WRONLY, ZielZugriff);
```

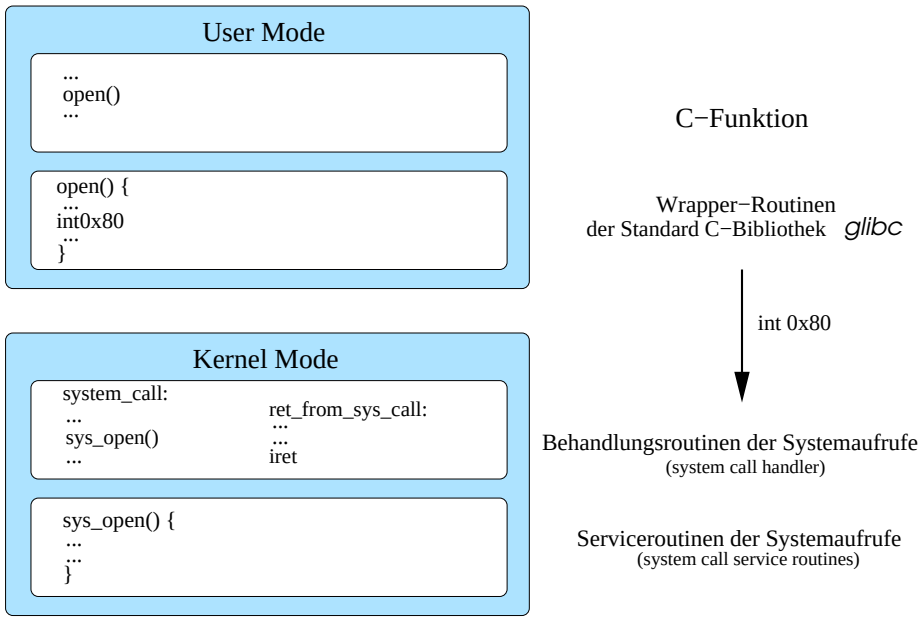


Abbildung 2.5: *Schematische Darstellung der Verarbeitung eines Systemaufrufs am Beispiel von open(). Eine C-Funktion im User Mode ruft die Funktion open() auf, welche in der C-Bibliothek libc implementiert ist. Diese führt den eigentlichen Systemaufruf durch den Software-Interrupt 0x80 aus. Die Behandlungsroutine des Software-Interrupts für die Systemaufrufe löst die entsprechende Serviceroutine aus.*

```
do {
    bytes = read(quellfd, puffervariable, pagesize);
    write(zielfd, puffervariable, bytes);
    } while (bytes);
    close(quellfd);
    close(zielfd);
```

Die Aufrufe read() und write() verzweigen zunächst beide in die Serviceroutinen sys_read() bzw. sys_write(). Diese rufen hingegen jeweils ganz verschiedenartige Kernelfunktionen auf, im einen Fall die des Dateisystems und des Treibers für das zugrundeliegende Blockgerät, im anderen Fall die Funktionen der Gerätetreiber des angeschlossenen Druckers und des Parallelports in /usr/src/linux/drivers/parport[27]. Der Kernel gewährleistet mit den Systemaufrufen der Dateiverwaltung den einheitlichen Zugriff auf verschiedene Geräte oder auch gleichartige Speichergeräte mit unterschiedlichen Dateisystemformaten. In der Handhabung erscheinen all diese unterschiedlichen Geräte wie ein einziges. Darum bezeichnet man die Systemaufrufe der Dateiverwaltung mit ihren Behandlungs- und Serviceroutinen als *virtuelles Dateisystem* (*virtual filesystem* , *VFS*).

[27] Bei Personalcomputern war die parallele Schnittstelle (Parallelport) lange Zeit der Standardanschluß für Drucker. Inzwischen sind auch USB-Drucker üblich geworden.

2.5.2 Hardware-Interrupts

Zur Unterstützung von Hardware-Interrupts gibt es eine Hardwarekomponente, den programmierbaren Interrupt-Controller (PIC). Dieser unterscheidet sich auf verschiedenen Computerplattformen. Die Aufgabe des PIC ist es, die Hardware-Interrupts mehrerer Geräte zu verwalten. Der Prozessor kann immer nur einen IRQ gleichzeitig bearbeiten. Bei den asynchronen Hardware-Interrupts ist es jedoch nicht vorhersehbar, wann diese ausgelöst werden. Der Interrupt-Controller ist dafür zuständig, bei gleichzeitigem Eintreffen mehrerer IRQs, diese nach bestimmten Prioritäten geordnet dem Prozessor nacheinander weiterzuleiten. Alle existierenden IRQ-Leitungen der Peripheriegeräte sind mit den Eingangsanschlüssen dieser Steuereinheit verbunden. Der PIC kontrolliert, ob auf einer Leitung ein IRQ eintrifft und führt im zutreffenden Falle folgende Maßnahmen durch:

- Das eingehende Signal wird in den entsprechenden Interrupt-Vektor übersetzt
- Der Vektor wird in einem Interrupt-I/O-Port abgelegt, sodaß ihn der Prozessor über den Datenbus lesen kann
- Ein Signal wird an den mit dem Interrupt-Register (intr) verbundenen Eingang des Prozessors gesendet, was diesen veranlaßt, die Unterbrechung einzuleiten.
- Sobald eine Empfangsbestätigung vom Prozessor eintrifft, wird das Signal auf der intr-Leitung abgebrochen

Auch wenn der Interrupt-Controler heute oftmal im Prozessorblock integriert ist, um den übrigen Datenbusverkehr nicht durch die IRQs zu bremsen, so ist er dennoch ein eigenständiges Gerät. Als solches benötigt er in Linux einen Gerätetreiber. Zudem muß er beim Systemstart initialisiert werden. An den Interrupt-Controler werden Interrupt-Leitungen verschiedener Geräte angeschlossen. Diese sind mit absteigender Priorität durchnumeriert. Die Zuweisung der Prioritäten geschieht bei der Initialisierung der Interrupts, die zusammen mit den Exceptions in der IDT angelegt werden.

Neuere PCs können 15 Hardware-Interrupts verwalten. Beim Start des PCs legt das BIOS eine IDT an und belegt die unteren Vektoren mit Hardware-Interrupts. DOS übernimmt die IDT mit diesen Einträgen. Linux verwendet den unteren Bereich seiner IDT jedoch für Exceptions, weshalb diese Vektoren in der frühen Startphase in einen oberen Bereich der IDT verlegt werden. Die vom BIOS eingerichtete IDT wird von Linux provisorisch verwendet, im weiteren jedoch nicht übernommen. Vielmehr initialisiert der Kernel Geräte und Unterbrechungen selbst. Beim PC setzt er dabei den idtr auf den Speicherbereich der eigens eingerichteten IDT und füllt diesen mit vorläufigen Einträgen (*null interrupt handlers*), die zunächst nur die Funktion ignore_int() aufrufen. Im Laufe der weiteren Startphase des Kernels werden diese vorübergehenden Einträge der IDT durch Verweise auf die eigentlichen ISR ersetzt.

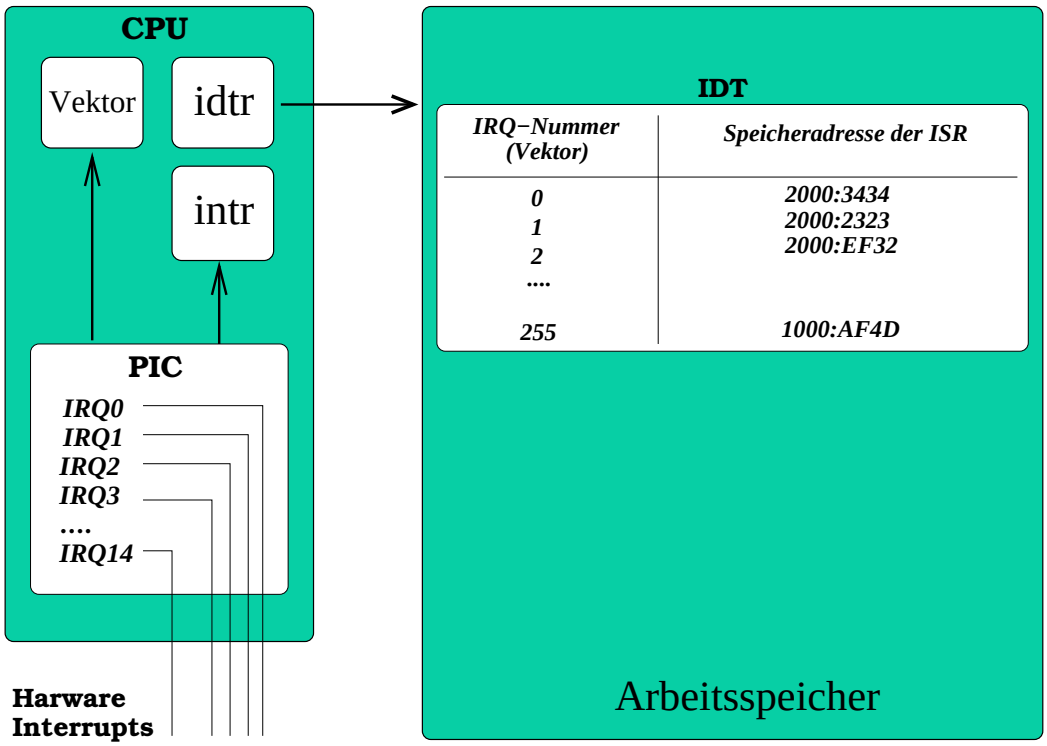


Abbildung 2.6: *Exceptions und Hardware-Interrupts werden von Linux in der IDT verwaltet. Dort werden alle Unterbrechungen mit ihrer IRQ-Nummer aufgeführt und ihnen die Startadresse der jeweiligen Behandlungsroutine im Speicher zugeordnet. Einem Systemaufruf durch den Interrupt 0x80, wie er im vorangegangenen Abschnitt beschrieben wurde, wird dort die Behandlungsroutine der Systemaufrufe zugewiesen. Ein Register idtr verweist auf den Beginn der IDT im Arbeitsspeicher. So kann das Betriebssystem diese dort an eine beliebige Stelle legen. Die CPU kann immer nur ein IRQ gleichzeitig behandeln. Damit nicht mehrere asynchrone Hardware-Interrupts gleichzeitig in der CPU eintreffen, leitet ein PIC solche Interrupts sequentiell und nach Prioritäten geordnet weiter.*

2.6 I/O-Schnittstellen des Kernels

In 2.3.1 wurde das Bussystem als innerer Datenkanal zwischen Arbeitsspeicher und Prozessor erläutert. Intel-Prozessoren unterstützen *I/O-Mapping*, was im Gegensatz zu Memory-Mapped-I/O einen eigenen Adreßraum für Peripheriegeräte implementiert. Im Adreßraum der Peripherie werden die Adressen als *Ports* bezeichnet. Über den Steuerbus wird festgelegt, ob auf den physikalischen Adreßraum des Arbeitsspeichers oder aber auf den Adreßraum des Peripheriespeichers zugegriffen werden soll. Die Assemblerbefehle mov und out implizieren zum Beispiel ein Steuerbussignal, das im ersten Fall den Datentransfer zum Arbeitsspeicher, im zweiten Fall zu einem I/O-Port lenkt. Das softwarebasierte Ansteuern der I/O-Geräte durch die CPU bezeichnet man als programmierte I/O (*PIO*). Dabei nimmt der Datentranfer zwischen der Peripherie und dem Arbeitspeicher den Prozessor in Anspruch. Um den Prozessor zu entlasten, wird exter-

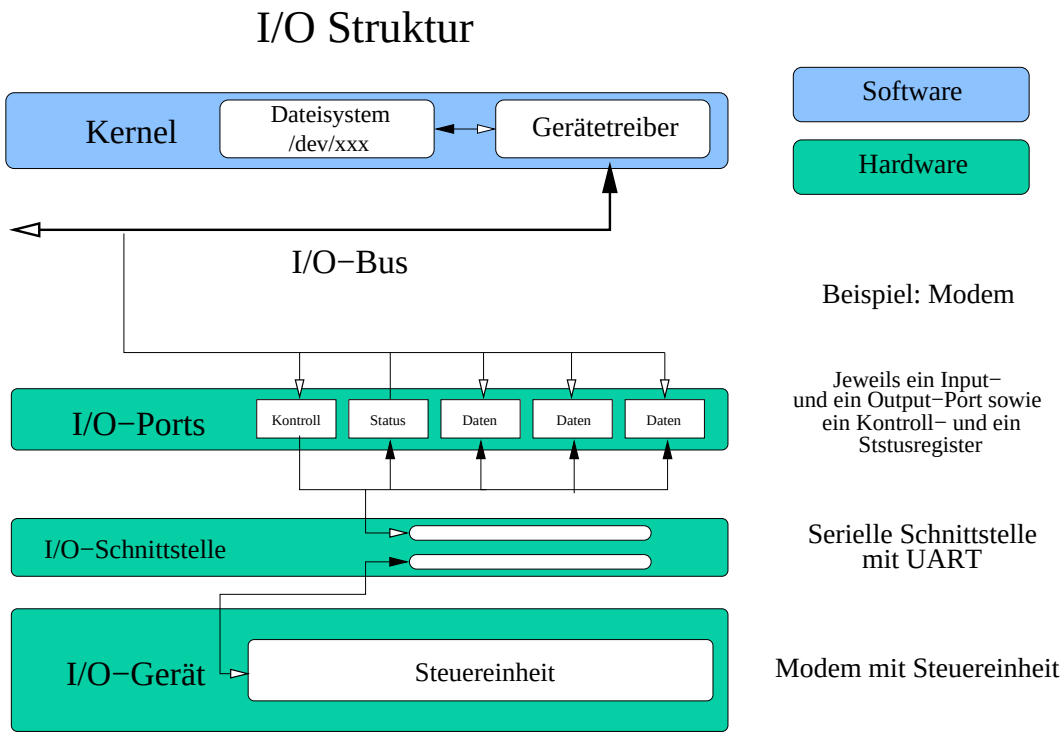


Abbildung 2.7: *Die I/O-Struktur besteht auf der Seite der Hardware im allgemeinen aus den I/O-Ports, welche die Adressen der Geräteregister bezeichnen, einer I/O-Schnittstelle, wie beispielsweise einer seriellen Schnittstelle und dem eigentlichen Gerät mit seiner Steuereinheit. Der Gerätetreiber im Kernel hat die Aufgabe, die auf das jeweilige Gerät bezogenen lesenden und schreibenden Systemaufrufe in Signale an die I/O-Ports umzusetzen. Dabei muß er zwischen der Darstellung des Geräts als Gerätedatei im Dateisystem und den tatsächlichen, physikalisch gegebenen Zugriffsmethoden vermitteln.*

nen Geräten, die große Datenmengen mit dem Arbeitsspeicher austauschen, der direkten Speicherzugriff *DMA* (direct memory access) über einen DMA-Controller ermöglicht.

Externe Geräte sind zumeist erheblich langsamer als der Prozessor. Um sicher zu stellen, daß die Kommunikation zwischen dem Prozessor und diesen langsam reagierenden Geräten erfolgreich ist, werden die I/O-Ports neben den Datenregistern um ein Kontrollregister und ein Statusregister erweitert. Diese im Prinzip vielseitig verwendbaren Register werden beim sogenannten *Handshaking* zur Kontrole des Datenflusses verwendet. Beim Senden der CPU geschieht das wie folgt:

- Die CPU setzt, wenn sie Daten in das Datenregister sendet, ein Signal „Daten gültig" im Kontrollregister.
- Anschließend bestätigt das I/O-Gerät den Empfang der Daten, indem es im Statusregister das Signal „Daten akzeptiert" setzt.

- Die CPU deaktiviert daraufhin das Signal im Kontrollregister und wartet, bis das I/O-Gerät wieder empfangsbereit ist.
- Seine erneute Empfangsbereitschaft signalisiert das Gerät, indem es das Signal im Statusregister wieder deaktiviert und damit der CPU mitteilt, daß weitere Daten gesendet werden können.

Zwischen den I/O-Ports und den Geräten, bzw. deren *Steuereinheiten*, befinden sich die *I/O-Schnittstellen*. Solche Schnittstellen sind beispielsweise das Keyboard-Interface, Disk-Interface, Graphic Interface, SCSI-Interface, die parallele und die serielle Schnittstelle. Letztere 3 können mit verschiedenen Geräten verbunden werden. Die serielle und parallele Schnittstelle werden als die Gerätedateien */dev/ttySn* und */dev/paraport* in das Dateisystem abgebildet. Alle I/O-Schnittstellen sind Harwareschaltkreise, die als Übersetzer des Inhalts der I/O-Ports in Kommandos für die Geräte oder deren Steuereinheiten dienen. Umgekehrt übersetzen sie die Daten und Signale der Geräte in Portbelegungen und lösen bei eintreffenden Ereignissen einen entsprechenden Hardware-Interrupt (IRQ) aus. Am unteren Ende der Schichtenhierarchie befindet sich das Gerät mit seiner Steuereinheit.
Die Gerätetreiber des Linuxkernels bedienen Systemaufrufe, indem sie Daten und Signale an entsprechende I/O-Ports senden oder Daten von diesen auslesen und deren Statussignale auszuwerten. Fast alle Geräte oder Steuereinheiten haben individuelle Kommandos und Statusinformationen, sodaß diese ebenso individuelle Gerätetreiber benötigen.

2.7 Der Kernelstart auf einem PC

Auf dem PC geschieht das Laden des Kernels für gewöhnlich von Diskette oder Festplatte. Die Routine, welche das Kernelabbild (*Kernelimage*) in den Speicher lädt, ist im BIOS implementiert und wird von einem Bootloader aufgerufen. Dieser Vorgang wird in Anhang D beschrieben. Dabei übergibt der Bootloader dem Kernelimage die Hardwareinformationen des BIOS. Bei einem eingebetteten System gibt es kein standardisiertes BIOS. Zwar ist Linux bei der Hardwareinitialisierung nur auf wenige Vorgaben des BIOS angewiesen, zum Systemstart muß jedoch veranlaßt werden, daß das Kernelimage an eine geeignete Stelle im Arbeitsspeicher gelangt und im CPU-Reset-Vektor ein Verweis auf diese Stelle den Systemstart ermöglicht (Abschnitt 5.1). Der CPU-Reset-Vektor bezeichnet die physikalische Adresse, aus welcher die CPU beim Einschalten ihre erste Instruktion bezieht.
Das Kernelimage liegt in vielen Fällen komprimiert vor. Das komprimierte Kernelimage wurde eingeführt, als der Linuxkernel die Größe von einem MB überschritt, welche beim Intel 80368 im Real Mode an freiem Speicher zur Verfügung stehen [AW00]. Ein komprimiertes Kernelimage beinhaltet selbst den für seine Dekomprimierung notwendigen Code. Die Initialisierung des Kernels beginnt im komprimierten, wie im unkomprimierten Kernelimage mit dem sogenannten Kernel-Setup.

2.7.1 Das Kernel-Setup

Obwohl der größte Teil des Linuxkernels in C geschrieben ist, liegen die ersten Funktionen beim Kernelstart im Quelltext als Assemblercode vor. Der gesamte Quellcode befindet

sich in einem Linuxsystem per Standard im Verzeichnis */usr/src/linux/*. Darin existiert ein Unterverzeichnis *arch/*, in welchem sich die von der Rechnerarchitektur abhängigen Funktionen befinden, unter anderem in *arch/i386/* die Funktionen für den üblichen PC mit einem x86 Intelprozessor. Das Untervezeichnis *arch/i386/boot* beinhaltet den kerneleigenen Bootloader *bootsect.S*, sowie die Datei *setup.S*. In dieser Datei befindet sich der in Assembler geschriebene Eintrittspunkt des Kernels *start:*, der vom Binder direkt hinter den Bootloader plaziert wird und mit welchem ab der Speicheradresse 0x00090200 aufwärts die weiteren Ausführungen im Anschluß an den Bootloader beginnen.
Obwohl das BIOS bereits die wichtigsten Hardwarekomponenten initialisiert hat, werden diese im Setup aus Gründen der Stabilität und Portabilität ein weiteres Mal reinitialisiert, insbesondere erleichtert dies die Portierung auf ein eingebettetes System, die über kein BIOS verfügen. Als erstes wird jedoch noch einmal eine BIOS-Routine zurückgegriffen, um die Größe des verfügbaren Arbeitsspeichers zu ermitteln. Wenn der Kernel an die Stelle 0x0010000 geladen wurde (Anhang D), so wird er jetzt an die Stelle 0x0001000 verschoben, um beim Entkomprimieren noch einen freien Puffer hinter dem Kernel zu haben. Darüber hinaus werden eine provisorische IDT (interrupt descriptor table) und eine provisorische GDT (global descriptor table) initialisiert. Der programmable interrupt controller (PIC) wird neu programmiert, und die 16 Hardware-Interrupts (IRQs) werden in der IDT aus den unteren Bereichen in die Vektoren 32 bis 47 verschoben. Abschließend erfolgt das Umschalten in den Protected Mode und der Sprung zur ersten *startup_32()* Assemblerfunktion in *arch/i386/boot/compressed/head.S*.

2.7.2 Die startup_32-Funktionen

Je nachdem, ob der Kernel in den hohen oder tiefen Arbeitsspeicher geladen wurde, befindet sich die erste startup_32-Funktion entweder an der Stelle 0x00100000 oder 0x00001000 im Arbeitsspeicher. Neben weiteren Initialisierungen wird bei einem komprimierten Kernel die Funktion *decompress_kernel()* aufgerufen. Es wird die Meldung „Uncompressing Linux..." auf der Konsole ausgegeben und der Kernel wird zum einen Teil in eine tiefe und zu einem anderen Teil in eine hohe Speicherregion entkomprimiert. Daraufhin werden beide Teile zusammengeführt und an die Stelle 0x00100000 verschoben.

Es folgt ein Sprung zu dieser Adresse und der Kernel mit der bearbeitung der zweiten startup_32-Funktion in *arch/i386/kernel/head.S* fort. Diese Funktion richtet zunächst eine Umgebung für den ersten Prozeß mit der Prozeßnummer 0 ein. Die Segmentregister werden endgültig initialisiert und ein Stack für den ersten Prozeß wird aufgesetzt. Die Funktion *setup_idt()* wird aufgerufen, um die eigentlichen IDT[28] mit sogenannten null interrupt handlers zu initialisieren (Abschnitt 2.5.2). Der Prozessor wird identifiziert, woraufhin die Register für den globalen Deskriptor (gdtr) und den Interrupt-Deskriptors (idtr) mit den Adressen von GDT und IDT geladen werden.

[28] Linux verwendet die vom BIOS initialisierte IDT nur provisorisch und verwirft diesen in der späteren Startphase

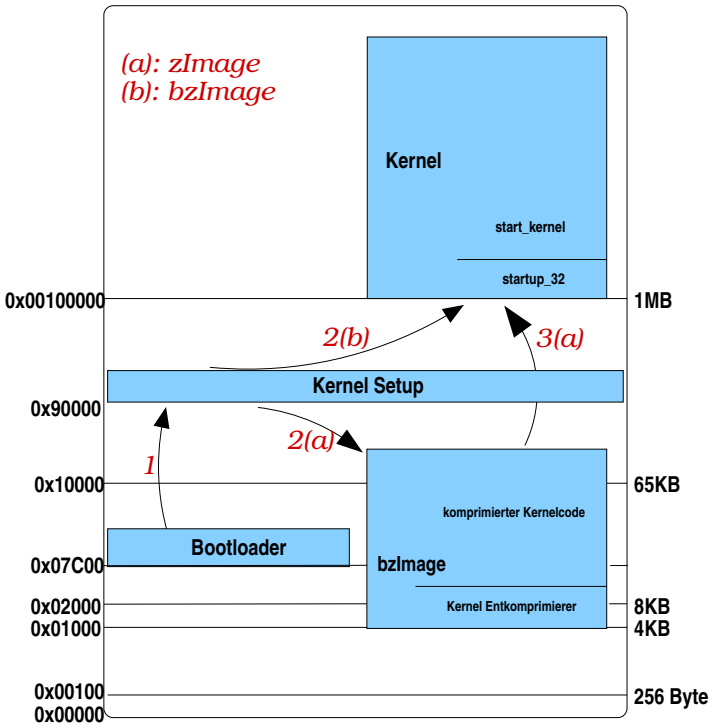


Abbildung 2.8: *Schematische Darstellung der Kernelinitialisierung im Arbeitsspeicher durch die startup_32-Funktionen bei einem stark komprimpierten (b) und schwach komprimierten (a) Kernelimage.*

2.7.3 Die start_kernel-Funktion

Im Anschluß an die vorangegangenen Assemblerfunktionen wird die erste C-Funktion *start_kernel()* aufgerufen, welche sich in der Datei *init/main.c* befindet. Diese sichert zunächst die Daten über die Hardware und führt weitere architekturabhängige Initialisierungen durch [BC01]. Dann werden Paging, Systemaufrufe, IDT, Scheduler und Timer entgültig initialisiert und der Prozeß 0 gestartet, der in einem Kernelthread die Funktion *init()* ausführt. In init() selbst befindet sich wiederum die Funktion *do_basic_setup()*, welche mit *mount_root()* das Wurzelverzeichnis (root file system) einbindet, das ab diesem Zeitpunkt mit all seinen Dateien zur Verfügung steht. Es wird versucht, eine Konsole zu öffnen, um auf dieser entweder direkt einen Kommandointerpreter (eine *shell*) /bin/sh oder das Programm */sbin/init* aus dem Wurzelverzeichnis zu starten. Letzteres startet die unter Linux laufenden Hintergrundprozesse und das Programm *getty*, was eine Benutzeranmeldung ermöglicht. Die Konfigurationsdatei für init befindet sich im Verzeichnis */etc/inittab*. Hier kann festgelegt werden, an welcher Konsole welches Programm wie gestartet werden soll. Dabei besteht die Möglichkeit, einen Kommandozeileninterpreter über die serielle Schnittstelle zu betreiben. Oftmals wird die Benutzerinteraktivität auf eingebetteten Linuxsystemen unter Verwendung der serielle Schnittstelle umgesetzt (Abschnitt 4.3). Ursprünglich wurden auf diese Weise serielle Terminals[29] oder *Telety-*

[29]Terminals sind Bildschirme mit einer Tastatur

pes (tty) an einen Großrechner angeschlossen, weshalb die Gerätedateien für die seriellen Schnittstellen unter Linux auch heute noch mit ttyS0 bis ttySn benannt werden.

Überblick auf die einzelnen Quellen in der Reihenfolge der Ausführung beim Kernelstart auf dem PC

- arch/i386/boot/bootsec.S
- arch/i386/boot/setup.S
- arch/i386/boot/compressed/head.S
- arch/i386/kernel/head.S
- init/main.c: start_kernel()

Kapitel 3

Linuxportierung auf den PowerPC 405

3.1 Der PPC 405 im Rahmen der PowerPC Architekturen

Der IBM PowerPC 405 (PPC 405) entspricht dem Architekturstandard für PowerPCs, der durch eine einheitliche User Instruction-Set Architecture (UISA) definiert ist. Die UISA ist der im User Mode verfügbare Teil der Befehlssatzarchitektur. Alle PowerPCs sind RISC-Architekturen, im Falle des PPC 405 mit einer Unterstüzung für 5 stufiges Pipelining. Die jeweilige Umsetzung des Befehlssatzes auf der Ebene der Mikro-Architektur variiert innerhalb des PowerPC-Standards.
Auf dem Virtex-II Pro ist der PowerPC Prozessorblock in den FPGA integriert und besteht neben dem *embedded IBM PowerPC 405-D5* RISC Kern aus einer Reihe von Schnittstellen zum FPGA. Der PowerPC-Architektur entsprechend, ist die interne Struktur des Prozessorkerns eine Harvard-Architektur mit getrennten Bussen für Daten und Instruktionen und separater Instruction Cache Unit (ICU) und Data Cache Unit (DCU) mit jeweils eigenen Steuereinheiten. Die Nutzung dieser Trennung ist jedoch optional. Der Prozessor besteht aus den folgenden Komponenten:

- der CPU mit einer Fetch/Decode Unit und einer Execution Unit[1]
- einer Speicherverwaltungseinheit (MMU)
- den beiden Cache Units ICU und DCU
- drei verschiedenen Zeitgebereinheiten (Timer) und
- einer Debug Logic Unit

Desweiteren gibt es eine Schnittstelle zu dem im Prozessorblock intergrierten Interrupt Controller und zur Steuereinheit für einen On-Chip Memory (OCM) auf dem FPGA-Block. Dieser ist ebenfalls über getrennte Busse in einen Instruktions- und einen Datenspeicher aufgeteilt.

[1] Das Operationswerk mit einer Intergereinheit ist hierbei Bestandteil der Execution Unit.

Der PPC 405 gehört zur Prozessorfamilie der IBM PowerPC 400er Reihe, deren Modelle als 32-Bit-Architekuren speziell für eingebettete Systeme konzipiert sind. Die allen PowerPCs gemeinsame UISA besteht in einem gemeinsamen Befehls- und Registersatz auf der Basis von 32 oder 64 Bit und betrifft den unprivilegierten Modus. Es gibt jedoch verschiedene Erweiterungen der Befehlssatzarchitektur. Zusätzliche Funktionalitäten im Benutzermodus, die für gewöhnliche Anwendungssoftware nicht unbedingt erforderlich sind, werden unter dem Begriff *Virtual Environment Architecture* (VEA) zusammengefaßt und unterscheiden sich innerhalb der einzelnen PPC-Architekturen. Dazu zählen beispielsweise Fließkommaeinheiten, Multimediaerweiterungen, sowie die Cache-Verwaltung und deren Kontrollinstruktionen. Diese Unterschiede haben eine geringe Bedeutung bei der Portierung des Betriebssystems. Anders verhält es sich mit den Unterschieden in der sogenannten *Operating Environment Architecture (OEA)* verschiedener PowerPCs. Diese Unterschiede beziehen sich auf die Instruktionen und den Registersatz des privilegierten Modus, in dem das Betriebssystem arbeitet. Dies betrifft insbesondere die Speicherverwaltungseinheiten. Der PPC 405 unterscheidet sich von der herkömmlichen PowerPC-OEA durch eine stark vereinfachte Speicherverwaltung mit einem softwaregesteuerten TLB und variabler Seitengröße, sowie durch einige Erweiterungen in Bezug auf die Hardwareunterstützung von Unterbrechungen, den Zeitgeber und die Debugwerkzeuge. Bezüglich einer detaillierten Beschreibung aller Merkmale des PPC 405 wird auf das Benutzerhandbuch von Xilinx [Xil02b] verwiesen, worin unter anderem Registermodell und Instruktionen erläutert werden.

Die Speicherverwaltungseinheit

Viele PowerPC-Architekturen verfügen über MMUs, welche die in Abschnitt 2.3.4 erläuterte Kombination aus Segmentierung und Seiteneinteilung implementieren. Die Speicherverwaltungseinheit des PPC 405 ist hingegen ausschließlich mit einer Paging-Unit zur Unterstützung von Seiteneinteilung ausgestattet. Im *Virtual Mode*, der dem Protected Mode bei Intel-Prozessoren entspricht, wird aus den 32 Bit breiten prozeßinternen virtuellen Adressen zunächst eine 40 Bit breite virtuelle Adresse durch das Voranstellen einer 8 Bit breiten Prozeß-ID erzeugt. Die Prozeß-ID wird dem gleichnamigen Register *pid* entnommen. Anschließend übersetzt die Paging-Unit die 40 Bit breite virtuelle Adresse in die 32 Bit breite physikalische Adresse. Die Seitengröße kann dabei variabel gewählt werden. Es werden acht verschiedene Seitengrößen im Bereich vom 1 KB bis 16 MB unterstützt. Zur schnelleren Adreßauflösung verfügt die MMU des PPC 405 über einen Translation Lock-Aside Buffer (TLB) für 64 Einträge. Die Erzeugung der TLB-Einträge und der Seitentabellen ist Aufgabe des Betriebssystems. Zur schnellen Suche nach TLB-Einträgen unterstützt der PPC405 die Instruktion TLB-Search **tlbsx**. Jeder TLB-Eintrag ist einer von 16 Zonen zugeteilt, für die im *Zone-Protection Register* ZPR individuelle Zugriffsrechte definiert werden können (Abbildung 3.1).

Im *Real Mode* der MMU werden virtuelle Adressen unübersetzt als physikalische Adressen interpretiert. Dabei steht im Gegensatz zum *Real Mode* des Intel 80386 der volle 32 Bit breite Adreßraum zur Verfügung. Zum Umschalten in den *Virtual Mode* muß die Adreßübersetzung aktiviert werden. Dies kann für Instruktionen oder Daten jeweils unabhängig erfolgen, indem das instruction-relocate Bit oder das data-relocate Bit im

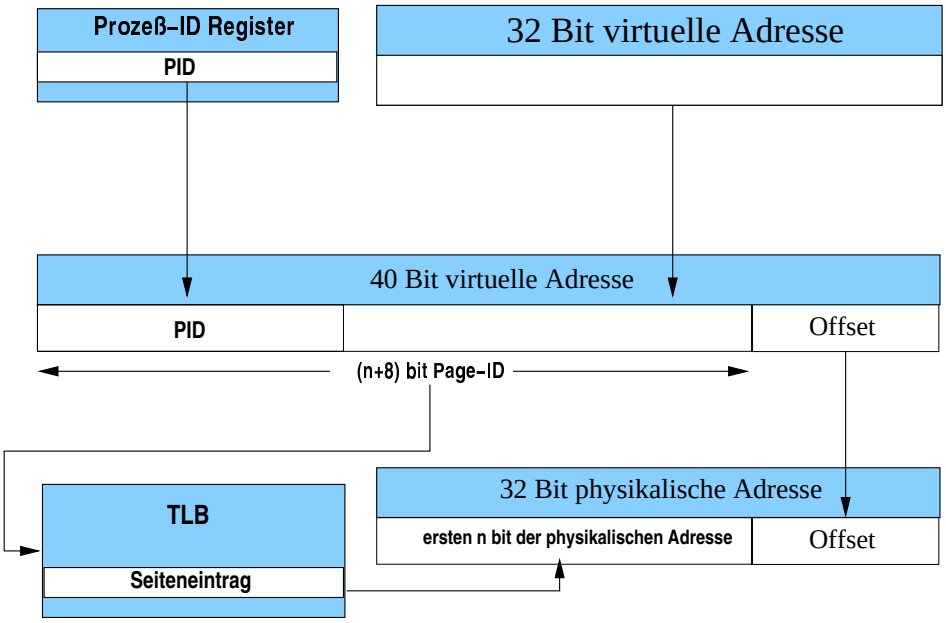


Abbildung 3.1: *Schematische Darstellung der Adressübersetzung beim PPC 405.*

Machine-State Register (MSR) auf 1 gesetzt wird.

Privilegstufen

Der PPC 405 unterstützt zwei Privilegstufen *User Mode* und *Privileged Mode*, was dem Kernel Mode unter Linux entspricht. Die Privilegstufe wird durch das Privilege-Level Bit im *Maschine-State Register* MSR festgelegt. Ist dieses Bit auf 0 gesetzt, so befindet sich der Prozessor im privilegierten Modus. Der Übergang aus dem User Mode erfolgt durch den System-Call Interrupt 0x0C00.

Unterstützung für Interrupts und Exceptions

Eine besondere Eigenschaft der PowerPC 400er Reihe ist die Aufteilung der Unterbrechungen in kritische und unkritische Exceptions, die mit Hinblick auf Echtzeitanforderungen eingeführt wurde. Die vom PPC 405 unterstützen Exceptions können durch Fehlerzustände, den Zeitgeber, Debug-Ereignisse, Softwareinterrupts oder den externen Interrupt-Controller ausgelöst werden. Beim Eintreffen einer Exception werden die Rücksprungadresse und das *Machine-State Register* MSR automatisch gespeichert. Kritische und unkritische Exceptions verwenden dafür jeweils getrennte Speicherregister, SRR0 und SRR1 für unkritische sowie SRR2 und SRR3 für kritische Exceptions. Der Software wird damit eine unabhängige Behandlung beider Kategorien von Exceptions ermöglicht. Beim Aufruf *return-from-interrupt* (**rfi**) oder *return-from-critical-interrupt* (**rfci**) werden die Register mit den Rücksprungadressen automatisch restauriert, sodaß dies keine Aufgabe der Interrupt Behandlungsroutine ist. Die Exceptions und die Werte der Exception-Vektoren unterscheiden sich innerhalb der einzelnen PowerPC-Architekturen.

Der PPC 405 stellt ein exception-vector-prefix Register (EVPR) zur Verfügung, in dem die Basisadresse der Tabelle für die Behandlungsroutinen gespeichert werden kann. Ein 16 Bit breiter Exception-Vektor markiert die Position innerhalb dieser Tabelle. Die Größenordnung läßt es im allgemeinen nicht zu, die Behandlungsroutinen selbst in der Tabelle einzutragen. Stattdessen können Deskriptoren eingetragen und die Tabelle als IDT verwendet werden.

Schnittstellen

Der PPC 405 stellt die folgenden Schnittstellen zur Verfügung:

- Processor Local Bus (PLB) Schnittstelle
- Device Control Register (DCR) Schnittstelle
- Taktgeber und Power-Management Schnittstelle
- JTAG-Port Schnittstelle
- Schnittstelle zum On-Chip-Memory Controller (OCM)
- Schnittstelle zum On-Chip Interrupt-Controller

Der PLB ist der Systembus des PPC405. Er besteht aus einem 32 Bit breiten Adreßbus und drei 64 Bit breiten Datenbussen. Zwei dieser Datenbusse sind mit dem Daten-Cache verbunden, der eine für den lesenden Zugriff, der andere für den schreibenden Zugriff. Der Dritte Datenbus ist mit dem Instruktions-Cache verbunden und wird im lesenden Zugriff zum Laden der Instruktion verwendet.
Neben dem 4 GB Adreßraum, der durch den 32 Bit breiten Adreßbus des PLB adressiert wird, verfügt der PPC 405 über einen weiteren 64 KB großen Peripherieadreßraum zur Adressierung von externen Geräteregistern. Für den Zugriff auf diese existieren die Instruktionen *move-from-DCR* **mfdcr** und *move-to-DCR* **mtdcr**, ähnlich den Intel-Befehlen **in** und **out**.
Die JTAG-Port Schnittstelle dient dem Anschluß externer Debug-Werkzeuge, wie ROM-Monitore oder JTAG-Debugger. Die OCM-Schnittstellen ermöglichen den schnellen, unmittelbaren Zugriff auf zwei 16 KB große Block-RAM-Module, die für den Instruktionscode zeitkritischer Interrupt-Serviceroutinen gedacht sind.

Cacheverwaltung

Alle PowerPCs verfügen über zwei separate Caches für Daten und Instruktionen. Die Organisation der Caches unterscheidet sich innerhalb der einzelnen Architekturen. Der PPC 405 verfügt über zwei 16 KB große Caches mit jeweils 32 Byte großen Cachezeilen. Die Steuereinheiten der Caches verfahren nach dem LRU[2]-Prinzip, bei dem längere Zeit unverwendete Blöcke ersetzt werden. Mit den Befehlen **icread** und **dcread** ermöglicht der PPC 405 das Auslesen einzelner Blöcke zur Unterstützung bei einer Fehlersuche.

[2] least-recently use

Mit den Instruktion **iccci** und **dccci** werden sämtliche Einträge der Caches für ungültig erklärt. Die Einstellungen des Caches befinden sich im *Core-Configuration register* CCR0.

Emulation der Fließkommaeinheit

Das Rechenwerk des PPC 405 beinhaltet keine Fließkommaeinheit (FPU). Dies wird durch die Nullbelegung des Bit 18 im *Machine-State Register* (MSP) gekennzeichnet. Fließkommainstruktionen können vom Compiler in entsprechende Integerinstruktionen übersetzt werden oder zur Laufzeit vom Betriebssystem abgefangen werden. Eine Fließkommainstruktion löst im PPC 405 den *FPU-Unavailable Interrupt* 0x0800 aus. Das Betriebssystem kann in einer Behandlungsroutine die Übersetzung in die entsprechenden Intergerinstruktionen vornehmen.

3.2 Unterstützung des PowerPC 405 im Linuxkernel

Die noch immer aktuelle Version 2.4 des Linuxkernel beinhaltet seit ihrer ersten Veröffentlichung im Januar 2001 bereits Quellcodedateien zur Unterstützung von Prozessoren der PowerPC 400er Familie. Arbeiten zur Erweiterung des Linuxkernels für die PPC 400er Reihe begannen bereits über ein Jahr zuvor unter maßgeblicher Beteiligung von MontaVista, deren Mitarbeiter im Mai 2000 [MV01] einen ersten Linuxkernel auf der IBM PPC 405GB Referenzplattform *Walnut* [IBM00] zum Laufen bringen konnten. Bei den dazu notwendigen Erweiterungen des Linuxkernels handelt es sich um architekturspezifischen Quellcode, der sich den Konventionen entsprechend im Verzeichnis */linux/arch/ppc* befindet. Im folgenden wird ausschließlich Bezug auf den Quellcode genommen, der in allen neueren Ausgaben von Linux, wie auch in den Ausgaben der Parallelentwicklungen aus Linuxppc enthalten ist. Die Quelle ist der *linux-2.4.19* Kernel.

Initialisierungscode

Der Quellcode für die Kernelinitialisierung des PowerPC 405 befindet sich in linux/arch/ppc/kernel/head_4xx.S. Er wird in Anhang E erläutert.

Die Speicherverwaltung

Die Speicherverwaltung des PPC 405 unterstützt Paging und die PID-Einträge in der 40 Bit breiten virtuellen Adresse. Diese Einträge implementieren keine Segmentierung des Arbeitsspeichers. Im Gegensatz zu den Segmenten der Intel-Prozessoren sind die Größe und die Position dieser Einträge in der 40 Bit breiten virtuellen Adresse fest vorgegeben. Die bei Intel-Prozessoren unter Linux angelegten Segmente können für den PowerPC 405 nicht angelegt werden und entfallen im Quellcode für die PPC 400er Reihe.
Wie in Abschnitt (2.3.5) beschrieben, verwendet Linux zur eigentlichen Speicherverwaltung Paging. Die Paging-Unit des PPC 405 unterscheidet sich jedoch erheblich von der Paging-Unit der 32 Bit Intel-Prozessoren. Während bei letzteren die Verwaltung des TLB ausschließlich von der Hardware vorgenommen wird und das Betriebssystem nur die Seitentabellen verändern kann, so müssen die Einträge für den TLB des PowerPC

405 vom Betriebssystem beschrieben und entfernt werden.
Bei jedem Speicherzugriff sucht die Hardware nach einem entsprechendem Eintrag im TLB. Kann eine Seite dort nicht gefunden werden, dann löst die CPU die Exception 0x1200 (TLB miss exception) aus. Linux richtet eine Behandlungsroutine für diese Exception ein. Dort wird zunächst die Ursache für die Exception ermittelt. Die Behandlungsroutine bekommt von der MMU im Register DEAR die virtuelle Adresse, welche nicht aufgelöst werden konnte. Es wird zunächst untersucht, ob ein Schreibschutz übertreten wurde und auf eine Seite schreibend zugegriffen wurde. Wenn der Seitenzugriff gestattet ist und sich die entsprechende Seite im Arbeitsspeicher befindet, so wird sie aus der zuständigen Page-Table in den TLB geladen.

Der Prozeßwechsel

Während im Speicher jeder Prozeß über einen eigenen Adreßraum verfügt, müssen die Register der CPU von allen Prozessen geteilt werden. Die jeweilige Registerbelegung eines aktiven Prozesses wird als dessen Kontext oder *Hardwarekontext* oder *Prozeßkontext* bezeichnet. Der Prozeßwechsel durch den Scheduler impliziert einen Kontextwechsel. Um die spätere Restauration des alten Prozesses zu gewährleisten, muß dessen Kontext gesichert werden. Welche Register davon betroffen sind hängt von dem Registersatz des Prozessors ab und ist somit architekturspezifisch.
Auf Intel-Prozessoren richtet Linux ein spezielles Segment ein, um den Hardwarekontext abzulegen, das *Task State Segment* (TSS). Da der PPC 405 keine Segmentierung unterstützt, erfolgt der Kontextwechsel dort ausschließlich unter Verwendung des Kernelstacks.
Linux verwaltet den Prozeßkontext intern in der Struktur *thread_struct*, die in */include/asm-ppc/processor.h* für die einzelnen PowerPCs definiert ist. Der Prozeßwechsel durch den Scheduler und das Kopieren des Prozeßkontexts für den Aufruf fork(), der ein Kindprozeß mit identischem Kontext erzeugt, basieren auf dieser Struktur. Der Kontextwechsel geschieht durch den Aufruf der Funktion *_switch_to*(), das Kopieren eines Prozeßkontexts durch *copy_thread*(), welche beide in *arch/ppc/kernel/process.c* definiert sind. Der eigentliche Kontextwechsel ist in der Funktion *_switch* in *arch/ppc/kernel/entry.S* implementiert. Bei deren Aufruf durch *_switch_to*() werden die Argumente mit der Adresse des aktuellen Prozeßdeskriptors in r3 und des neuen Prozeßdeskriptors in r4 übergeben.
Der Scheduler organisiert die Prozeßwechsel. Dies geschieht architekturunabhängig und wird in der Funktion *schedule*() in *kernel/sched.c* ausgeführt.

Exceptions

Da die Organisation von Exceptions und die Zuordnung der entsprechenden Vektoren sich bei einzelnen Prozessoren unterscheiden, müssen die Behandlungsroutinen und deren Einleitung auf der Seite des Betriebssystems architekturspezifisch umgesetzt werden. Zudem sind bei der Einleitung der Behandlungsroutinen alle Register zur späteren Rekonstruktion des unterprochenen Zustandes zu sichern. Für den PowerPC 405 werden die wichtigsten Makros in der Datei *arch/ppc/kernel/head_4xx.S* definiert. Vor dem Sprung in die Behandlungsroutine werden die beständigen Register (Abschnitt 3.4) ge-

sichert und nach deren Beendigung restauriert. Dies wird in COMMON_PROLOG und COMMON_EPILOG ausgeführt. Der Code muß auf die Unterscheidung zwischen kritischen und unkritischen Exceptions eingehen und sichert die Adresse der vorangegangenen Instruktion und das MSR in STND_EXCEPTION_PROLOG durch das Auslesen der Register SPR0 und SPR1, beziehungsweise der Register SPR2 und SPR2 in CRIT_EXCEPTION_PROLOG separat. Weiter unten im Quelltext werden die Makros für die spezifischen Exceptions und deren Vektoren definiert. In _GLOBAL(transfer_to_handler) werden die zuvor gesicherten Register auf den Kernelstack gelegt und der Sprung zur jeweiligen Behandlungsroutine ausgelöst. Die einzelnen Behandlungsroutinen befinden sich in *arch/ppc/kernel* oder *arch/ppc/mm*. Systemaufrufe werden prozessorintern als Exception mit dem Vektor 0x0c00 behandelt. Deren Einleitung wird in D_GLOBAL(DoSyscall) in der Datei *arch/ppc/kernel/entry.S* veranlaßt.

3.3 Grundstruktur eines Cross-Compilers

Bei der Portierung von Anwendungssoftware können Compiler verwendet werden, die selbst als Anwendungsprogramm auf der Zielarchitektur laufen, wenn die Zielplattform bereits mit einem Betriebssytem ausgestattet ist. Die Portierung eines Betriebssystems benötigt hingegen einen Compiler, welcher auf einer anderen Plattform betrieben wird, als derjenigen für die er den Maschinencode generiert. Solch ein Compiler wird als Cross-Compiler bezeichnet.
Ein Cross-Compiler ist das wesentliche Werkzeug bei der Softwareportierung. In Abschnitt 2.4.1 wurde ein Compiler als Programm beschrieben, das den Quellcode einer höheren Programmiersprache in Maschinencode übersetzt. Im Unterschied zu einem Interpreter, welcher das Programm während des Übersetzens auf dem Computer ausführt, generiert der Compiler aus der Textdatei mit dem Quellcode eine Datei mit Maschinenbefehlen, eventuell unter Zuhilfenahme eines separaten Assemblers. Die Übersetzung vollzieht ein klassischer Multi-Pass Compiler[3] in mehreren Durchgängen [Wir96].

- Präprozessorphase
- Analysephase (Front-End):
  - lexikalische Analyse
  - syntaktische Analyse
  - semantische Analyse
- Synthesephase (Back-End)
  - Zwischencode-Erzeugung
  - Zwischencode-Optimierung
  - Maschinencode-Erzeugung

[3]Im Gegensatz zu den weniger geläufigen Single-Pass Compilern, die den Maschinencode in einem Schritt erstellen.

Der Präprozessor ist ein einfaches Textersetzungssystem. Er kann dazu verwendet werden, Makros[4] zu expandieren, bedingte Kompilierung umzusetzen, Kommentare zu entfernen und Prototypen aus Header-Dateien einzublenden. Die vom Präprozessor bearbeitete Datei wird dem Scanner zur lexikalischen Analyse übergeben. Dieser untersucht alle Wörter und Zeichen auf ihre Gültigkeit und übersetzt sie in eine Reihe verschiedener Symbole (*Tokens*). Aus diesen Tokens bildet der sogenannte *Parser* einen Strukturbaum und überprüft dabei, ob die grammatikalischen Regeln der Hochsprache erfüllt sind (syntaktische Analyse). Die semantische Analyse überprüft im Anschluß die Deklaration, Initialisierung und Typkonformität der verwendeten Variablen und Funktionsparameter. Wenn in der Analysephase Fehler gefunden wurden, so bricht der Compiler mit einer entsprechenden Fehlermeldung ab. Anderenfalls beginnt die Synthesephase mit der Umwandlung des vom Parser generierten Strukturbaums in einen einfachen Zwischencode. Dieser besteht aus elementaren Operationen mit zwei oder drei Operanden und ist nicht mehr von der höheren Programmiersprache abhängig, aus deren Quelltext er erzeugt wurde. Der Zwischencode ist von der Struktur einem Assemblercode ähnlich, jedoch maschinenunabhängig. Ebenso ist die anschließende Zwischencode-Optimierung maschinenunabhängig. Sie dient in erster Linie dazu, die Ausführungsgeschwindigkeit des generierten Programms zu erhöhen und dessen Speicherbedarf zu verringern. Hierbei werden beispielsweise nicht erreichbare Quellcode-Abschnitte eliminiert, invariante Schleifenteile herausgezogen oder logische Ausdrücke optimiert.
Erst im letzten Schritt der Kompilierung spielt die Zielarchitektur eine Rolle. Der Code wird in diesem Durchgang für die Zielarchitektur optimiert, sodaß architekturspezifische Leistungsmerkmale voll ausgeschöpft werden können. Anschließend folgt die Übersetzung in den entsprechenden Maschinen- oder Assemblercode. Der Unterschied ist unerheblich, da jeder Assemblerbefehl genau einen Maschinenbefehl darstellt und beim Assemblieren eindeutig abgebildet wird. Die Übersetzung des Quellcodes beim Kompilieren ist hingegen variabel, sodaß unterschiedliche Compiler aus dem gleichen Quellcode durchaus verschiedenen Maschinencode generieren. Der von verschiedenartigen Compilern erzeugte Maschinencode kann in Bezug auf den Speicherbedarf oder die Ausführungsgeschwindigkeit variieren. Dies ist der Grund dafür, daß zeitkritische Programmteile gelegentlich in Assembler geschrieben werden. Doch auch bei gleichermaßen effizienter Kompilierung sind Abweichungen in der Codeumsetzung möglich. Der bestehende Spielraum bezieht sich auf die folgenden Punkte, die das sogenannte *Application Binary Interface* definieren:

- das Binärformat
- die Umsetzung verschiedener Datentypen und deren Byte-Anordnung
- die Verwendung der Register
- die Organisation von Stacks
- die Übergabe von Funktionsparametern
- und die Organisation des Zugriffs auf kleine Datenbereiche

[4] Symbolische Namen

Aus Gründen der Kompatibilität wurden verschiedene Richtlinien inform von Standards festgelegt. Wie die Anlehnung des *Application Programmer Interfaces* (API) an den POSIX[5]-Standard in den C Standard Libraries eine Vereinheitlichung der Schnittstelle verschiedener Betriebssysteme auf der Ebene des C-Quellcodes darstellt, so geschieht die Vereinheitlichung auf der Ebene der Maschinenprogramme durch die Kompatibilität des *Application Binary Interfaces* (ABI), als Schnittstelle zwischen den kompilierten Programmen und dem Betriebssystem. Man spricht in diesem Zusammenhang von der *binären Kompatibilität*. Die gleichen Anwendungsprogramme können auf verschiedenen Betriebssystemen gestartet werden, wenn diese ein kompatibles ABI besitzen [Dev02]. Das ABI beinflußt die folgenden Komponenten eines Betriebssystems:

- Das unterstützte Binärformat
- Die binäre Umsetzung
  - der Systemaufrufe im Kernel und
  - der dynamisch gebundenen Bibliotheken
- den Runtime-Linker

Bei der Maschinencode-Erzeugung durch den Compiler wird auch das ABI festgelegt, welches, neben der Befehlssatzarchitektur der Zielplattform, den von einem Cross-Compiler generierten Maschinencode und dessen Kompatibilität ebenso kennzeichnet.

3.4 Das Embedded Application Binary Interface (EABI)

Für die für eingebettete Systeme ausgelegten PowerPC Prozessorfamilien 400 und 800 haben IBM und Motorola ein spezielles Application Binary Interface erstellt, das *PowerPC Embedded Application Binary Interface* (EABI), welches Linux im architekturspezifischen Assemblercode für den PPC 405 verwendet. Das EABI verfolgt in der Hauptsache die Ziele, den Speicherbedarf zu minimieren und die Verarbeitungsgeschwindigkeit zu optimieren [MOT95]. Es stellt unter anderem ein Konzept zur Verfügung, mit dem ein schnellerer Speicherzugriff durch die *registerindirekte Adressierung* auf der Grundlage von Basisadresse und Offset ermöglicht wird, welche in den weiteren Ausführungen erläutert wird. Bei der registerindirekten Adressierung werden in den Registern Referenzen gespeichert, welche die eigentliche Speicherstelle adressieren. Das im EABI festgelegte Binärformat ist das *Executable and Linkable Format* [TIS]. Der Entwurf entstand in Anlehnung an das SVR4[6] ABI für PowerPCs, beinhaltet jedoch einige Unterschiede.

Zu Beginn der Portierung von Linux für die PowerPC 400er Reihe mußte hinsichtlich des im Assemblercode verwendeten ABI eine Entscheidung getroffen werden. Die ersten Codeerweiterungen für den PPC 403 begannen 1999 mit den Arbeiten von Grant Erickson und waren unter anderem Assemblerroutinen des Initialisierungscodes. Dafür wurde das

[5] **P**ortable **O**perating **S**ystem **I**nterface (for UNI**X**)
[6] **S**ystem **V** **R**elease **4**

EABI gewählt. Obwohl dieses durch die Entwicklungswerkzeuge für lange Zeit wesentlich schlechter unterstützt wurde als das herkömmliche SVR4 ABI, fiel die Entscheidung zugunsten eines effizienteren Codes. Die wesentlichen Merkmale des EABI werden im folgenden dargestellt. Sie sind eine Voraussetzung für das Verständnis des im Abschnitt 5.1 betrachteten Initialisierungscodes für den PPC 405. Zudem ergeben sich durch das EABI Probleme bei der Cross-Compilierung, die in Abschnitt 3.6 beschrieben werden.

Datentypen und Anordnung

Alle PowerPC-Architekturen unterstützen eine Byteordnung bei der das signifikanteste Byte die niedrigste Speicheradresse zugewiesen bekommt (big-endian). Die PowerPC 400er Prozessoren unterstützen zusätzlich die umgekehrte Byteordnung (little-endian). Die PowerPC-Architektur definert 5 verschiedene Größen für skalare Datentypen

- Byte
- Halbwort (2 Byte)
- Wort (4 Byte)
- Doppelwort (8 Byte)
- Quadwort (16 Byte)

Unabhängig von der Byteordnung werden alle Datentypen sowohl im Speicher, als auch innerhalb eines Stapelbereichs (*Stack-Frame*) auf Adressen gelegt, die ein Vielfaches der jeweiligen Datengrößen sind. Das Quadwort bildet beim EABI-Standard eine Ausnahme und wird auf eine durch 8 teilbare Adresse gelegt. Für Strukturen ist das Element mit dem größten Datentyp maßgeblich. Die Anordnung der Datentypen auf durch n teilbare Speicherpositionen bezeichnet man als *Alignment*. Die ANSI[7] C Datentypen entsprechen den folgenden PowerPC-EABI-Datentypen:

ANSI C	PPC EABI	Alignment
char	Byte	1
short	Halbwort	2
int	Wort	4
long int	Wort	4
enum	Wort	4
pointer	Wort	4
float	Wort	4
double	Doppelwort	8
long double	Quadwort	8

[7] **A**merican **N**ational **S**tandard **I**nstitute

Registerkonventionen

Die PowerPC-Architektur definiert 32 *General-Purpose Register* (GPR) r0 bis r31 und 32 *Floating-Point Register* (FPR). Letztere sind im PPC 405 nicht implementiert. Das EABI klassifiziert diese Register als flüchtig (*volatile*), beständig (*nonvolatile*) oder dediziert. Beständige Register müssen ihren Wert über einen Funktionsaufruf hinweg beibehalten. Als flüchtig ausgezeichnete Register können bei einem Funktionsaufruf ohne anschließende Restaurierung überschrieben werden. Dedizierte Register sind ausschließlich zu ihrem bestimmten Zweck zu verwenden:

GPR	Typ	Zweckbestimmung
r0	flüchtig	sprachenspezifisch
r1	dediziert	Stack-Pointer (SP)
r2	dediziert	SDA-Basis (nur lesend)
r3-r4	flüchtig	Parameterübergabe; Rückgabewerte
r5-r10	flüchtig	Parameterübergabe
r11-r12	flüchtig	
r13	dediziert	SDA-Basis (lesend und schreibend)
r14-r21	beständig	

Zusätzlich sind die Felder cr2 bis cr4 des *Condition Registers* als beständig deklariert. Alle anderen Register sind als flüchtig klassifiziert.

Konventionen für das Stack-Frame

Die PowerPC-Architekturen beinhaltet keine Instruktionen wie *pop* oder *push*. Die Konventionen des EABI sehen vor, daß jede Funktion, die wiederum eine andere Funktion aufruft oder ein als flüchtig klassifiziertes Register verändert, ein *Stack-Frame* anlegt. Dabei verweist der Stackpointer stets auf das Wort mit der niedrigsten Adresse und jeder Stack wächst in Richtung absteigender Adressen. Das Anlegen des *Stack-Frames* geschieht nicht beim Funktionsaufruf, sondern innerhalb der aufgerufenen Funktion als Bestandteil der Initialisierungsroutine (Prolog). Dies erfolgt durch einmaliges Dekrementieren des Stackpointers um die Größe des erforderlichen Stack-Frames, welches auch alle beständigen Register sichern muß, die im Kontext der Funktion verändert werden. Entsprechend muß die Beseitigung des Stack-Frames in der Beendigungsroutine (Epilog) vorgenommen werden.
Im EABI haben *Stack-Frames* das folgende Format, wobei ein Auffüllbereich dafür sorgt, daß ihre Größe stets das Vielfache eines Doppelworts beträgt:

FPR Speicherbereich
GBR Speicherbereich
CR Speicherwort
Bereich für lokale Variablen
Bereich für Funktionsparameter
Auffüllbereich
LR-Speicherwort
Back-Chain-Wort

Alle *Stack-Frames* besitzen eine Kopfzeile, die aus jeweils einem Feld für das Back-Chain Wort und einem weiteren für das Link Register besteht. Der Stack ist eine einfach verkettete Liste von *Stack-Frames*, die im Back-Chain Wort den Verweis auf ihren Vorgänger speichern. Das Link Register (LR) beinhaltet die Rücksprungadresse zur aufrufenden Funktion und wird vor der Modifikation durch die Funktion ebenso in der Kopfzeile gespeichert. Wenn die für die Parameterübergabe designierten Register r3 bis r10 nicht ausreichen, werden weitere Funktionsparameter im Stack-Frame der aufrufenden Funktion gespeichert. Falls der Vorrat an flüchtigen Registern erschöpft ist, können weitere Funktionsvariablen im Stack-Frame gespeichert werden. Bei einer Modifikation von beständigen Registern, werden diese zuvor im obersten Bereich des Stack-Frames gespeichert, wobei im Falle des PPC 405 keine Floating Point Register (FPR) existieren und deren Speicherbereich im Stack-Frame demnach niemals angelegt wird. Wenn das Back-Chain Wort eingetragen ist, muß der Stack auch effektiv erweitert werden. Die Abfolge beim Auffüllen des Stacks darf nicht unterbrochen werden, man spricht hierbei von einer atomaren Operation. Dies gewährleistet die Instruktion *store-word-with-update* (**stwu**). Zum Rücksprung in die aufrufende Funktion wird die Instruktion *branch-to-link-register* (**blr)** verwendet. Das LR muß zuvor mit dem Wert des LR-Speicherworts aus dem Stack-Frame restauriert werden. Zur Verdeutlichung sei ein Beispiel aus [IBMP98] kommentiert:

Funktionscode im Prolog:

mflr r0	Link Register in r0 holen (*move-from-link-register*)
stwu r1, -88(r1)	Back-Chain speichern und Stackpointer (r1) dekrementieren
stw r0,+92(r1)	Link Register im Stack speichern (*store-word*)
stmw r28, +72(r1)	4 beständige Register r28-r31 speichern (*store-multiple-word*)

Es wird die *registerindirekte Adresierung mit explizitem Index* verwendet. Das Link Register beinhaltet zunächst die Rücksprungadresse. Die Back-Chain-Information für das neue Stack-Frame ist im aktuellen Stackpointer r1 enthalten. Sie wird durch **stwu** in das neue Stack-Frame geschrieben, wobei gleichzeitig der Stackpointer r1 aktualisiert (dekrementiert) wird. Anschließend wird die zuvor aus dem Link Registers in r0 geladene Rücksprungadresse im alten Stack-Frame gespeichert. Zuletzt werden die Register r28 bis r31 im oberen Bereich des neuen Stack-Frames gesichert.

Funktionscode im Epilog:

lwz r0,+92(r1)	Link Register aus dem Stack-Frame in r0 holen (*load-word-and-zero)*
mtlr r0	Link Register restaurieren (*move-to-link-register*)
lmw r28, +72(r1)	Beständige Register restaurieren (*load-multiple-word*)
addi r1,r1,88	Stack-Frame vom Stack nehmen (*add-immediate*)
blr	Rücksprung zur aufrufenden Funktion

Hier werden zunächst das im oberen Stack-Frame gespeicherte Link Register und anschließend die im aktuellen Stack-Frame gespeicherten beständigen Register r28 bis r31 restauriert. Daraufhin wird das letzte Stack-Frame entfernt, indem der Stackpointer r1 einfach inkrementiert wird.

Übergabe von Funktionsparametern

Für die Übergabe von Funktionsparametern sind die Regster r3 bis r10 und bei weiteren Parametern der entsprechende Bereich im Stack-Frame vorgesehen, der auch für größere Rückgabedaten verwendet wird, falls die für die Rückgabewerte dedizierten Register r3 und r4 nicht ausreichen. Das folgende Beispiel aus [IBMP98] veranschaulicht die Assemblerimplementierung eines Funktionsaufrufes gemäß den Konventionen des EABI:

```
...
int var;
main(){
    var = 4;
    funktion(var);
}
```

Dem EABI entspricht die Übersetzung:

var = 4:	Variable initialisieren
li r11, 4	(*load-immediate*)
addis r12, r0, var@ha	(*add-immediate-shifted*)
stw r11, var@l(r12)	(*store-word*)
funktion(var):	
lwz r3, var@l(r12)	Parameter in r3 laden (*load-word-and-zero*)
bl funktion	Rücksprungadresse in LR speichern und in die Funktion springen (branch and link)

Die Initialisierung der Variable erfolgt dabei in drei Schritten. Als erstes erhält r11 den Wert 4. Das obere Halbwort der Variablenadresse wird dann in r12 geladen. Zuletzt wird die Adresse in den Speicher geschrieben, wobei das untere Halbwort der Variablenadresse die Versetzung in Bezug auf die in r12 enthaltene Basisadresse angibt.

Small Data Areas (SDA)

Die Adressierung im PowerPC, wie sie in den oben angeführten Load- und Store-Instruktionen erfolgt, setzt sich aus einer Basisadresse und einer 16 Bit Verschiebung zusammen. Somit kann ein 64 KB großer Adreßbereich abgedeckt werden, ohne das Register mit der Basisadresse zu verändern. Um von dieser Möglichkeit zu profitieren, werden im EABI *Small Data Areas* (*SDAs*) definiert, für welche die Registerinhalte von r2 u r13 als Basisadressen festgelegt sind. SDAs werden für globale und statische Variablen verwendet. Die Initialisierung der Variablen *var* des vorangegangenen Beispiels benötigt als SDA-Variable nur zwei Codezeilen:

```
li r12, 4
stw r12, var@sdaxr(r13)
```

Die Variablen der in r13 verankerten SDA sind beschreibbar und werden in den ELF-Segmenten *.sdata* oder *.sbss* [TIS] aufgenommen. Die in r2 verankerten SDA-Variablen sind unveränderlich und in den ELF-Segmenten *.sdata2 oder .sbss2* enthalten.

3.5 Die GNU-Toolchain

Linux wurde mit der GNU-Toolchain der Free Software Foundation [FSF] entwickelt. Bei dieser handelt es sich, wie auch bei Linux selbst, um freie Software. Die GNU-Toolchain zeichnet sich durch eine breite Unterstützung von Hardware- und Betriebssystemplattformen, aber auch durch eine leistungsstarke Optimierung des generierten Maschinencodes aus. Damit eignet sie sich für eine plattformübergreifende Programmentwicklung (Cross-*Development*). Zumdem unterstützt sie spezifische Spracherweiterungen des ANSI-C, welche im Kernelquellcode von Linux verwendet werden. Die Entwicklungsgeschichte von Linux steht in einer so engen Verbindung mit den GNU-Werkzeugen, daß diese allgemein als Voraussetzung für die Portierung von Linux betrachtet werden.
Unter einer *Toolchain* versteht man im Bereich der Softwareentwicklung die Kette von Werkzeugen, die zur Programmgenerierung (Abschnitt 2.4.1) verwendet werden. In der GNU-Toolchain sind dies der Präprozessor (*cpp*), der GNU C-Compiler (*gcc*[8]), der GNU Assembler (*as* oder *gas*) und der GNU Linker (*ld*). Zudem stehen eine Reihe weiterer Werkzeugen zur Verfügung, insbesondere ein Debugger (*gdb*), der es erlaubt, interne Vorgänge und Zustände eines Programms während der Ausführung zu verfolgen. Ein Debugger ist das wichtigste Werkzeug zur Lokalisierung und Behebung von Fehlern.

Die gesamte GNU-Toolchain kann über das gemeinsame Kontrollprogramm gcc einheitlich bedient werden. Durch dessen Aufrufoptionen wird festgelegt, welche Werkzeuge auf den Quellcode angewandt werden sollen. Zudem können über die Aufrufoptionen die Eigenschaften des zu generierenden Codes manipuliert werden. Damit ist beispielsweise das ABI oder die Unterstützung spezieller Merkmale eines Prozessors innerhalb einer Prozessorfamilie zu steuern. Die Zusammenstellung der Toolchain für die Linuxportierung auf den PPC 405, welche in dieser Arbeit verwendetet wurde, wird im Anhang C beschrieben.

[8] *Inzwischen steht GCC für **G**NU **C**ompiler **C**ollection*

3.6 Probleme bei der Portierung

Die Fehlerquellen beim Versuch, den Linuxkernel für eine fremde Plattform zu kompilieren sind sehr vielseitig und die angezeigten Fehlermeldungen nicht eindeutig auf eine bestimmte Ursache zurückzuführen. Bei der Generierung des Kernels können die Fehlermeldungen vom Compiler, vom Assembler oder vom Linker ausgegeben werden.

Fehlermeldungen beim Kompilieren sind in den überwiegenden Fällen auf die Kernelquellen zurückzuführen. Die Unterstützung der PowerPC 400er Reihe ist im offiziellen Linuxkernel etwas veraltet, sodaß unter anderem die Compileroptionen an neuere Compiler angepaßt werden müssen. Die Makefiles beziehen sich auf die alten Versionen des *gcc*, in denen die Option -m405 noch nicht unterstützt wurde. Es wird empfohlen, die neueren Ausgaben des linuxppc-Kernels für die PowerPC 400er Reihe zu verwenden.

Bis zu der neuen Version *gcc-3.1* konnten Fehlermeldungen des Assemblers oder des Linkers bei einem *powerpc-linux-gcc* beobachtet werden, welche beim *powerpc-eabi-gcc* ausblieben. Dies änderte sich auch nicht, wenn neben der Option -m403 zusätzlich die Option -meabi explizit angegeben wurde. Sehr wahrscheinlich wird das im Quellcode des Linuxkernels verwendete EABI durch die Option -meabi nicht ausreichend unterstützt. Der Nachteil eines *powerpc-eabi-gcc* ist, daß dieser die *glibc* nicht übersetzen kann. Es muß hierbei auf die Verwendung der *newlib* ausgewichen werden, die als eine alternative C-Bibliothek für eingebettete Systeme entstand. Das Binden an die *newlib* kann für den Programmcode von HANNEE zu Portierungsschwierigkeiten führen, da dieser auf der *glibc* basiert.

Im August dieses Jahres wurde der *powerpc-405-linux-gnu-gcc* in der Ausgabe gcc-3-3-1 veröffentlicht. Mit diesem lassen sich alle neueren Ausgaben des linuxppc-Kernels für den PPC 405 kompilieren. Bei der in dieser Arbeit getesteten Zusammenstellung des *powerpc-405-linux-gnu-gcc* wurden die *binutils-2.14.90.0.5* und die *glibc-2.3.2* verwendet. Die im März dieses Jahres herausgegebene *glibc-2.3.2* kann in EABI-konformen Maschinencode für den PowerPC 405 übersetzt werden.

Kapitel 4

Einrichten des Wurzelverzeichnisses

Die Systemprogramme unter Linux entstammen dem GNU-Projekt der Free Software Foundation [FSF], weshalb in manchen Dokumentationen gelegentlich GNU-Linux als Bezeichnung für das Betriebssystem auftritt. Es ist für die jeweiligen Systemanforderungen unter Berücksichtigung der vorhandenen Speicherressourcen zu entscheiden, welche Systemprogramme benötigt werden und welche Pakete demnach im Wurzelverzeichnis zu installieren sind. Der Programmumfang des Wurzelverzeichnisses erstreckt sich in embedded Linuxsystemen, je nach Speicherkapazität, von einem einzigen, statisch gebundenen Anwendungsprogramm bis hin zu vollständigen Mehrbenutzersystemen, wie sie auf einem PC vorliegen. Wann immer es Schnittstellen und Speicherkapazität ermöglichen, sollte ein Kommandozeileninterpreter für Steuerungs- und Wartungsaufgaben im System verfügbar sein. Mittels dessen können die Prozeßdateien ausgelesen und Konfigurationsdateien im laufenden System angepaßt werden. Die Prozeßdateien im Verzeichnis */proc* sind eine unentbehrliche Grundlage vieler Systemverwaltungsprogramme und stellen die wichtigste Informationsquelle über den Systemzustand dar. Darum eignen sie sich häufig zur Fehlersuche, sei es im Falle eines instabilen Systemzustands oder bei eingeschränkter Funktionsfähigkeit. Insbesondere in der Entwicklungsphase eines eingebetteten Systems ist mit derartigen Schwierigkeiten zu rechnen.
Auf den NATHAN-Modulen steht in dem 256 MB großen DDR SDRAM für einen Kommandozeileninterpreter genügend Arbeitsspeicher zur Verfügung. Auf Hardwareebene setzt diese Form der Benutzerinteraktivität eine Tastatur und einen Bildschirm als Konsole voraus. Eine Konsole kann als Bestandteil der Systemperipherie oder aber extern über eine äußere Schnittstelle angeschlossen vorliegen. Im zweiten Fall spricht man von einem Terminal. Obwohl sich auf dem Markt eingebetteter Systeme bisher noch keine mit dem PC vergleichbaren Standards etablieren konnten, wird die serielle Schnittstelle RS232 seitens der Betriebssystementwickler vielfach als Regelfall betrachtet. Die Ausstattung eingebetteter Systeme mit Bildschirm und Tastatur ist hingegen sehr selten.
Die serielle Schnittstelle überträgt Daten als einzelne Bits in zeitlicher Abfolge (seriell) über zwei angeschlossene Eindrahtleitung. Die Daten können von dem Sender in beliebigen, unregelmäßigen Abständen gesendet werden. Es liegt den gesendeten Daten kein Synchronisations- oder Taksignal zugrunde. Da der Empfänger demzufolge die gesendeten Zeichen nicht durch ein mit dem Programmablauf synchronisiertes zeitliches Eintreffen identifizieren kann, besteht das Protokoll aus Start- und Stopsignalen. Mit ei-

ner Null als Startbit signalisiert der Sender, daß die folgenden acht Bit als ein Zeichen zu interpretieren sind. Die Bitfolge wird mit einem oder zwei Stopbits beendet. Wurde ein ganzes Zeichen (ein Byte) übertragen, so befindet sich dieses in einem Puffer der empfangenden Schnittstelle. Dieser Puffer ist Bestandteil des UART (*universal asynchronous receiver transmitter*), welcher die Aufgabe hat, den Prozessor beim Empfang eines vollständigen Bytes durch Senden eines IRQ-Signals darüber zu unterrichten. Zudem muß er die erhaltenen seriellen Daten in die interne, parallele Darstellung des Systembusses umwandeln. Umgekehrt konvertiert er vor dem Senden ganze Bytes in einzelne Bitfolgen. Linux unterstützt verschiedene serielle Schnittstellen mit Gerätetreibern, deren Quellcode sich in *linux/drivers/char/serial.c* befindet.

Auf dem Virtex-II Pro kann eine serielle Schnittstelle zur Slow-Control auf dem FPGA programmiert werden. Auf Seite der Software muß das Betriebssystem für den Zugriff über eine serielle Schnittstelle konfiguriert werden. Ebenso müssen die notwendigen Gerätedateien, Initialisierungsskripte und Programme im Wurzelverzeichnis zur Verfügung gestellt werden. Die Gestaltung des Wurzelverzeichnisses bestimmt die gesamte Systemkonfiguration, womit, im Unterschied zur Kernelkonfiguration, die zur Laufzeit des Systems veränderbaren Systemeigenschaften gemeint sind.
Die Dateien des Wurzelverzeichnisses zusammenzustellen erfordert ebenso ein Entwicklungssystem, wie die Cross-Kompilierung des Kernels. Es gibt vielfältige Gründe dafür, zum Testen der Systemkonfiguration die Umgebung des Zielsystems zunächst auf dem Arbeitscomputer zu simulieren.
In erster Linie ist der Umstand zu nennen, daß im Falle einer fehlerhaften Systeminitilisierung auch die Werkzeuge zur Fehlersuche auf dem Zielsystem versagen können. Außerdem ist eine Simulation auf dem Arbeitscomputer komfortabler, weil man den generierten Code nicht transferieren muß und zumeist bessere Werkzeuge zur Fehlersuche und Fehlerkorrektur, neben einem leistungsstärkeren Rechner, zur Verfügung hat. Je nach Beschaffenheit des eingebetteten Systems kann das Transferieren der erzeugten Software mit erheblichem Aufwand verbunden sein. Überdies wird durch eine einstweilige Simulation das Spektrum der Fehlerquellen veringert. Die bei der Simulation auf dem Arbeitscomputer möglicherweise auftretenden Probleme sind nicht auf den Cross-Compiler, den damit erzeugten Kernel oder die Hardware der Zielplattform zurückzuführen. Nach der eins-zu-eins Portierung eines auf dem Arbeitscomputer stabilen und funktionsfähigen Systems können hingegen die meisten Konfigurationsskripte und die Zusammenstellung der Programme als Fehlerquellen grundsätzlich ausgeschlossen werden. Zuletzt kann es, wie im Falle der vorliegenden Arbeit, sein, daß die Hardware infolge einer zeitgleichen Entwicklung gar nicht zur Verfügung steht.

Wenn mit dem Cross-Compiler der Kernel fehlerfrei kompiliert wurde, sind Kompilierfehler bei den Anwedungsprogrammen nahezu ausgeschlossen. Die verbleibenden Schwierigkeiten ergeben sich im wesentlichen aus zwei Eigenschaften eingebetteter Systeme. Diese sind eine spezielle Peripherie und beschränkte Speicherressourcen. Das Einrichten eines vollständigen eingebetteten Betriebssystems muß in der Hauptsache diese beiden Aspekte berücksichtigen, welche im folgenden auf einem Personalcomputer simuliert werden.

4.1 Der Kernel

Auf dem Entwicklungssystem kann ein Kernel derart konfiguriert werden, daß dem darauf basierenden Betriebssystem nur solche Ressourcen zur Verfügung stehen, welche auch auf der Zielplattform vorhanden sind. Indem dem Kernel die entsprechenden Gerätetreiber fehlen, bleiben dem Betriebssystem diese Geräte verborgen. Es ist auf solche Weise möglich, die gesamte Peripherie einschließlich aller Sekundärspeichergeräte dem Betriebssystem zu entziehen. Ein Kernelparameter [GB00] ermöglicht zudem die Begrenzung des Arbeitsspeichers, beispielsweise auf 256 Megabyte, der Größe des DDR-RAM auf den NATHAN-Modulen entsprechend :

mem=256m

Mit einem weiteren Kernelparameter kann die Systemkonsole auf die serielle Schnittstelle gelegt werden:

console=/dev/ttyS0

Um die Peripherie des PowerPCs auf den NATHAN-Modulen zu simulieren, müssen weitere Leistungsmerkmale eines Personalcomputers durch eine entsprechende Kernelkonfiguration verdeckt werden. Insbesondere ist in *general setup* die PCI-Unterstützung zu deaktivieren, womit sämtliche PCI-Geräte aus dem Kernel genommen werden.
Damit die serielle Schnittstelle als Konsole verwendet werden kann, müssen die Konfigurationsoptionen unter *character devices* für den Gerätetreiber und die Konsole ausgewählt werden:

Standard/generic (8250/16550 and compatible UARTs) serial support
Support for console on serial port

Für das Wurzelverzeichnis in einer Ramdisk sind unter der Kategorie *block devices*

RAM disk support
Initial RAM disk (initrd) support

auszuwählen. Die Größenvorgabe für die Ramdisk ist auf einen im Rahmen der gesamten Speicherkapazität angemessenen Wert einzustellen

(4096) Default RAM disk size.

Zudem muß das im Wurzelverzeichnis verwendete Dateisystem in der Kategorie *file systems* ausgewählt werden.
Auf diese Weise simuliert der Kernel ein eingebettetes System, das zur Ein- und Ausgabe nur über eine serielle Schnittstelle verfügt und die Möglichkeit bietet, das Wurzelverzeichnis in eine Ramdisk zu legen. Das Laden der Ramdisk kann wahlweise von Festplatte oder Diskette geschehen, bessere Netzwerkkarten ermöglichen überdies den Systemstart von einem Netzwerkserver. Wenn solch eine Netzwerkarte nicht zur Verfügung steht, muß wahleweise die Unterstützung für eine Festplatte oder aber ein Diskettenlaufwerk im Kernel vorhanden sein. Es stehen Gerätetreiber für verschiedenartige Festplatten zur Verfügung, wobei grundsätzlich zwischen IDE und SCSI-Festplatten unterschieden wird. In Abhängigkeit vom vorliegenden Peripheriebus sind die entsprechenden Gerätetreiber in der Kategorie ATA/IDE/MFM/RLL oder SCSI support zu finden.

4.2 Die Ramdisk

Welche Programme auf einem eingebetteten System installiert werden müssen, hängt grundsätzlich von dessen spezifischer Funktionalität ab. Dieser Abschnitt erläutert lediglich die generelle Vorgehensweise und die wesentlichen Gesichtspunkte beim Einrichten des Wurzelverzeichnisses. Hinsichtlich weiterer Informationen zum Einrichten eines Linuxsystems wird auf [BG03] verwiesen.
Für die NATHAN-Module wurde ein vollständiges Installationsskript entworfen. Dieses ist variabel ausgelegt und somit für verschiedene Konfigurierationen geeignet. Insbesondere ist der Parameter für die Prozessorarchitektur und den Compiler eine globale Variable. Dadurch kann das System zunächst in der Simulationsumgebung getestet und mit dem gleichen Skript identisch auf die Prozessorarchitektur des PowerPC 405 übertragen werden.

4.2.1 Testumgebung mit VmWare

Beim Konfigurieren eines Betriebssystems muß dieses häufig gestestet werden. Wenn es dazu auf einen anderen Computer kopiert wird, kann dies sehr zeitaufwändig sein. Darum ist es sinnvoll, bei der Entwicklung und Konfiguration des Wurzelverzeichnisses für die Ramdisk eine gesonderte Partition auf dem Entwicklungssystem einzurichten. Dadurch erübrigen sich Kopiervorgänge und die angelegte Partition kann einem Bootloader direkt als *root file system* des fingierten eingebetteten Systems angegeben werden. Diese Vorgehensweise scheint mit dem Nachteil behaftet zu sein, daß das laufende Betriebssystem zum Testen des entwickelten Systems heruntergefahren werden muß. Diese Notwendigkeit besteht jedoch nicht, wenn auf dem laufenden Betriebssystem die Hardware eines weiteren PCs auf der Basis von Software simuliert wird. Zu solchen Simulationen eignet sich die von VMware [VM03] angebotene Workstation, welche in einer zeitlich befristeten Probeversion kostenlos zur Verfügung gestellt wird.
VMware simuliert einen virtuellen Computer, der sich die Ressourcen mit dem Arbeitsrechner teilt. Es ist konfigurierbar, welche Ressourcen VMware zur Verfügung gestellt werden. Somit kann VMware die Partition mit dem Testsystem zur Verfügung gestellt werden. Das Konfigurationsskript des Bootloaders ist dementsprechend zu erweitern. Dabei muß beachtet werden, daß die erste VMware verfügbare Festplatte /dev/hda ist, unabhängig davon, um welche Festplatte es sich auf dem Arbeitscomputer handelt. Beim Verwenden von *grub*[1] [MO02] kann die Konfigurationsdatei für ein Linuxsystem auf der 3. Festplatte beispielsweise wie folgt aussehen:

```
title LinuX
kernel (hd2,0)/boot/vmlinux  root=/dev/hdc1

title VM_Embedded_Linux
kernel (hd0,7)[2]/vmlinux_embeded  root=/dev/hda8  rw  mem=265m
```

In diesem Beispiel befindet sich das Entwicklungssystem auf der ersten Partition und das Testsystem auf der achten Partition. Der Kernelparameter *rw* bewirkt, daß in das einge-

[1] ***gr**and **u**nified **b**ootloader*
[2] Die Zählung der Festplatten und Partitionen beginnt unter GRUB bei Null.

bundene Wurzelverzeichnis schreibend zugegriffen werden kann. Ohne diesen Parameter muß das Wurzelverzeichnis über ein Initialisierungsskript verfügen, welches das Wurzelverzeichnis erneut einbindet, da der Kernel dieses in dieser Funktion *mount_root()* ohne einen Bootparameter ausschließlich mit Leserechten einbindet.
Ein Eintrag in etc/fstab ermöglicht das automatische Einbinden der Testpartition in ein beliebiges Arbeitsverzeichnis auf dem Entwicklungssystem:

```
/dev/hdc8   /home/asinsel/vmembedded    user,exec   1 2
```

4.2.2 Zusammenstellung der Ramdisk

Zunächst ist in einem beliebigen neuen Verzeichnis das minimale Grundgerüst der standardisierten Dateisystemhierarchie [RQ01] anzulegen. Da sich dieses auf große, für mehrere Benutzer ausgerichtete Unixsysteme oder vollständige Linuxdistributionen mit graphischer Benutzeroberfläche bezieht, sind viele Verzeichnisse, wie beispielsweise das Benutzerverzeichnis */home* oder */usr/X11* auf einem eingebetteten System für gewöhnlich überflüssig. Für ein kleineres System möge die folgende Grundstruktur als Beispiel gelten:

```
cd /$TARGET_RFS
mkdir /bin /dev /etc /lib /mnt /proc /sbin /tmp /usr /var
mkdir /usr/bin /usr/lib /usr/sbin
mkdir /var/lib /var/lock /var/log /var/run /var/tmp
chmod a=rwx-t tmp/
```

Benutzer können bei dieser Konfiguration eigene Dateien im Verzeichnis */tmp* ablegen. Die letzte Zeile bewirkt, daß in dem gemeinsam genutzten Verzeichnis */tmp* nur die Eigentümer einer Datei berechtigt sind, diese zu löschen. Die somit erzeugte Verzeichnisstruktur stellt für gewöhnliche Anwendungen ein ausreichendes Grundgerüst dar. Es ist jedoch nicht auszuschließen, daß eine Programminstallation zusätzliche Unterverzeichnisse erwartet. Wahlweise können in diesem Falle weitere Verzeichnisse angelegt oder die Installationsskripte der bestehenden Verzeichnisstruktur entsprechend verändert werden. In einem Betriebssystem ohne Shared Libraries sind die Verzeichnisse /lib und /usr/lib überflüssig. Das Verzeichnis */mnt* dient dem Zugriff auf andere Datenträger innerhalb des Testsystems, wenn von diesem aus auf die Programme der Entwicklungsumgebung zugegriffen werden muß.
Der bevorzugte Kommandozeileninterpreter auf Linuxsystemen ist das Programm *bash*[3], welches sich als Anwendungsprogramm üblicherweise im Verzeichnis */bin* befindet. Der Kernel erwartet bei der Initialisierung die Existenz eines Initialisierungsprogramms *init* oder einer Shell mit dem absoluten Namen */bin/sh* im Wurzelverzeichnis. Dies kann in der Funktion init() der Quellcodedatei *linux/init/main.c* geändert werden:

```
if (execute_command)  execve(execute_command,argv_init,envp_init);
execve("/sbin/init",argv_init,envp_init);
execve("/etc/init",argv_init,envp_init);
```

[3] **b**ourne **a**gain **sh**ell

```
execve("/bin/init",argv_init,envp_init);
execve("/bin/sh",argv_init,envp_init);
panic("No init found. Try passing init= option to kernel.");
```

Wenn von einer Änderung des Kernelcodes abgesehen wird ist demzufolge bei anderen Startanwendungen ein Link auf diese mit dem Namen */bin/sh* anzulegen. Der Kommandozeileninterpreter *bash* beinhaltet eine umfangreiche Funktionalität und ist dementsprechend groß. Die kleine Shell *ash* eignet sich wegen ihrere geringen Größe besser für die Ramdisk in einem eingebetteten System. Sie kann in das Verzeichnis */$TARGET_RFS/bin* kopiert und der entsprechende Link mit dem Namen *sh* angelegt werden:

```
cp .../ash  /$TARGET_RFS/bin/ash
cd /$TARGET_RFS/bin
ln -s /ash  /$TARGET_RFS/bin/sh
```
[4]

Wenn die Entscheidung zum Verwenden von Shared Libraries getroffen wurde, sollte *ash* dynamisch gebunden sein. In diesem Falle sind die benötigten Bibliotheken in /$TARGET_RFS/lib/ zu kopieren. Sie können mit dem Kommando *ldd*[5] ermittelt werden, welches die von einer dynamisch gebundenen Anwendung benötigten Bibliotheken und Ladeprogramme auflistet. Beim dynamischen Binden gegen die GNU C-Library **glibc** beinhaltet die ausgegebene Liste stets den Runtime-Linker für die Bibliotheken /lib/ld-linux.so.X[6] und die C-Standardbibliothek /lib/libc.so.X, wobei X für die Major-Versionsnummer der jeweils benötigten C-Library steht. In der Liste sind ausschließlich Links aufgeführt, welche auf die eigentliche Bibliotheken verweisen. Da dynamisch gebundene Anwendungen intern die Namen dieser Links referenzieren, sind diese über die eigentlichen Bibliotheken hinaus ebenso erforderlich.
Die Bibliotheken der neueren glibc-Versionen sind relativ groß und für Systeme mit geringen Speicherressourcen ungeeignet. In eingebetteten Systemen werden darum oftmals kleinere Bibliotheken, wie uClibc, Diet libc oder ältere Versionen der glibc verwendet[7].
Mit dem Kommando chroot [SSFH00] kann, das aktuelle Wurzelverzeichnis gewechselt werden, um die Funktionsfähigkeit der in */$TARGET_RFS* eingerichteten Programme zu testen:

```
chroot $TARGET_RFS/
```

Nach einem erfolgreichen Test[8] können weitere Systemprogramme hinzukopiert und ebenso getestet werden. Die für Systeme mit geringen Speicherressourcen konzipierte busybox[9]ist auf eingebetten Systemen für gewöhnlich ausreichend und eine Alternative

[4] Da es sich um die Programme des späteren Wurzelverzeichnisses handelt, dürfen an dieser Stelle keine absoluten Pfadangaben gemacht werden.

[5] Bei cross-kompilierten Programmen kann das Kommando *readelf -d* verwendet werden, welches als *Binary Utility* Bestandteil der GNU-Toolchain ist.

[6] Bei Linuxsystemen für MIPS oder PowerPC lautet der Dateiname /lib/ld.so.X

[7] Weitere Informationen findet man unter http://www.superant.com/smalllinux/

[8] Erscheint die Fehlermeldung *„chroot: cannot execute /bin/bash: No such file or directory"*, so fehlt in den meisten Fällen ein Link mit dem Namen sh oder aber eine der erforderlichen Shared Libraries im Verzeichnis *$TARGET/lib*.

[9] http://www.busybox.net/

zu den großen GNU-Programmen. Das Kommando *exit* beendet *ash,* womit die ursprüngliche Verzeichnisstruktur restauriert wird.
Wurden Kernelmodule generiert, so sind diese ebenso in das Wurzelverzeichnis zu kopieren:

```
cp -a .../modules/   $TARGET/lib/
```

Für alle benötigten Geräte müssen die entsprechenden Gerätedateien angelegt werden, die sich per Konvention im Verzeichnis /dev befinden. Die Gerätedateien sind ein Inode, der hierbei keine Blöcke des Datenträgers, sondern die jeweiligen Treiber referenziert. Die Geräte werden durch eine Mayor- und eine Minornummer identifiziert, wobei die erste auf den zuständigen Gerätetreiber verweist und die zweite ein bestimmtes Gerät kennzeichnet, welches diesen Treiber verwendet. Zum Anlegen der Gerätedateien steht das Kommando mknod dem Systemadministrator zur Verfügung:

```
cd $TARGET/dev
mknod -m 600 mem c 1 1
.....
```

Eine ausführliche Dokumentation und Auflistung aller Minor- und Majornummern befindet sich bei den Kernelquellen im Verzeichnis */usr/src/linux/Documentation/devices.txt.*

Ist das Wurzelverzeichnis für das eingebettete System vollständig, so muß daraus ein Abbild (*Ramdiskimage*) erzeugt werden, welches der Kernel beim Systemstart direkt als Ramdisk in den Speicher laden kann. Wie in 2.2.1 erläutert, muß dieses Abbild alle Blockstrukturen und sämtliche Inodes im Format eines Linux eigenen Dateisystems beinhalten. Dazu ist zunächst eine leere Datei mit der Größe der Ramdisk zu erzeugen:

```
dd if=/dev/zero of=rootfs.img bs=512 count=800
```

Disk dump [SSFH00] erzeugt in diesem Beispiel eine mit Nullen gefüllte Datei der Größe $800 * 512B = 400KB$ mit dem Namen *rootfs.img.* Dabei werden in *rootfs.img* 512 MB große Blockstrukturen angelegt, sodaß sich die Datei daraufhin mit einem beliebigen Dateisytem formatieren läßt:

```
mkfs.ext2 -c rootfs.img
```

Mit der Option -o loop läßt sich diese Datei als sogenanntes *loop-back device* wie ein gewöhnliches blockorientiertes Speichermedium mounten:

```
mount -o loop -t ext2 rootfs.img  /mnt
```

Dadurch wird es möglich, das Wurzelverzeichnis aus /$TARGET in das Pseudogerät *rootfs.img* zu kopieren:

```
cp -a $TARGET/  /mnt
umount /mnt
```

Die mit dieser Vorgehensweise eingerichtete Datei *root.img* stellt das exakte Abbild einer Ramdisk dar. Die durch den Inode referenzierten Datenblöcke beinhalten selbst ein Dateisystem. Für ein komprimiertes Ramdiskimage ist das vom Linuxkernel unterstützte Kompressionsformat von *GNU zip* zu verwenden, das auf einer lizenzfreien Variante des Lempel-Ziv-Verfahrens basiert:

```
gzip -v9 -c rootfs.img > rootfs.gz
```

4.3 Zugriff über eine serielle Schnittstelle

Die zwei herkömmlichen Methoden des interaktiven Arbeitens auf einem PC basieren auf einem Kommandozeileninterpreter oder einer graphischen Benutzeroberfläche. Ein eingebettetes System ohne Tastatur und Graphikkarte kann, wenn es mit einer seriellen Schnittstelle ausgestattet ist, über diese durch ein Terminal angesteuert werden. Diese Möglichkeit der Benutzerinteraktivität kann auf einem Personalcomputer simuliert werden.

4.3.1 Ein Terminal an der seriellen Schnittstelle

Auf einem Linux-PC werden die Graphikkarte mit dem Monitor als Stanardausgabe und die Tastatur als Standardeingabe eingerichtet. Die Standardausgabe ist in Linuxsystemen per Konvention mit der Gerätedatei */dev/console* verknüpft. Alle auf einem Kommandozeileninterpreter erzeugten Ausgaben sind als Sequenz von Zeichen zu je einem Byte kodiert, die an die Standardausgabe geleitet werden. Beispielsweise gibt das Kommando

```
cat Textdatei
```

die Zeichenfolge einer Textdatei auf der Standardausgabe aus. Mit der *Ausgabeumleitung* „>" wird der aus der Textdatei gelesene Datenstrom in eine andere Datei umgeleitet werden. Wenn an einer seriellen Schnittstelle ein Terminal angeschlossen ist, kann die Textdatei durch die Ausgabeumleitung über die assoziierte Gerätedatei */dev/ttyS0* auf diesem Terminal ausgeben werden:

```
cat Textdatei > /dev/ttyS0
```

Der Ausgabetext erscheint unmittelbar auf dem Terminal. Die Gerätedatei */dev/ttyS0* repräsentiert den Gerätetreiber der seriellen Schnittstelle.

Um die Benutzerinteraktivität auf einem eingebetteten System über ein serielles Terminal zu ermöglichen, muß bei der Initialisierung des Betriebssystems die serielle Schnittstelle als Konsole definiert werden. In einem gewöhnlichen Linuxsystem legt die SystemV-Version des Programms *init* die Definition der Standardkonsole fest. Init erzeugt Prozesse an den einzelnen Konsolen, die nach ihrer Terminierung automatisch wieder gestartet werden. Für gewöhnlich sind dies Ausführungen von *getty*, welche eine Anmeldung ermöglichen und bei erfolgreicher Authentifizierung einen Kommandozeileninterpreter

starten. Das Verhalten von init ist in der Konfigurationsdatei /etc/**inittab** zu konfigurieren. Die Einträge in inittab sind Zeilen mit dem Format

Code : Runlevel : Aktion : Befehl

Dabei bezeichnet *Code* eine beliebige zweistellige Identifikation der Zeile und *Befehl* den auszuführenden Befehl. Weiter wird ein *Runlevel*[10] spezifiziert, in welchem der Befehl beim Start ausgeführt werden soll. *Aktion* ist eine Anweisung für init, in welcher Weise der Befehl ausgeführt werden soll. Die Aktion *respawn* bewirkt, daß der Befehl nach jeder Terminierung wieder neu aufgerufen wird. Der Eintrag

S1 : 12345 : respawn : /sbin/getty 9600 /dev/ttyS0

ermöglicht eine Anmeldung an der der ersten seriellen Schnittstelle, indem dort das Programm getty gestartet wird. Hierbei werden getty zwei Parameter übergeben, die *Übertragungsrate* und die Gerätedatei. Die Übertragungsrate ist von dem angeschlossenen Gerät und der seriellen Schnittstelle selbst abhängig. Der UART der seriellen Schnittstelle verfügt über einen Puffer, in welchem an den Ports eingetroffene Daten solange gespeichert werden, bis sie der Prozessor als Reaktion auf ein Unterbrechungssignal abfragt. Die Größe des Puffers ist begrenzt, sodaß mit steigender Übertragungsrate ein Überlauf immer wahrscheinlicher wird, bei dem dann Daten verloren gehen. Aus diesem Grunde ist es nicht immer sinnvoll, eine maximale Übertragungsrate festzulegen[11].
Anstelle eines Terminals kann auch ein gewöhnlicher Personalcomputer mit einem Terminalemulator verwendet werden. Dazu müssen dieser PC und das eingebettete System mit einem Nullmodemkabel (ein gekreuztes Kabel) über beide serielle Schnittstellen verbunden werden. Ein Terminalemulatoren für Linux sind beispielsweise Minicom oder Kermit.

4.3.2 Eine Netzwerkverbindung über die serielle Schnittstelle

Auch ohne Netzwerkkarte ist es möglich, einen Computer über seine serielle Schnittstelle in ein Netzwerk einzubinden. Dieser Vorgang geschieht in der Praxis bei der Einwahl in das Internet per Modem. Ist die physikalische Verbindung zwischen dem eingebetteten System und einem Netzwerk gegeben, so bedeutet eine Netzwerkanbindung im wesentlichen nur noch das Aufsetzen der höheren Protokollebenen (beispielsweise TCP/IP), welche im Netzwerk verwendet werden. Die serielle Verbindung ist das unterste Übertragungsprotokoll auf der physikalischen Schicht. Auf der Basis dieser kann mittels PPP

[10] Die Runlevel sind bei SystemV-Init diverse Modi, in denen das Betriebssystem läuft. Normalerweise werden verschiedene Runlevel mit einer Nummer definiert. Ein Wechsel der Runlevel kann mit dem Befehl init *Runlevel* eingeleitet werden. Üblicherweise stehen Runlevel 0 für das Herunterfahren und Runlevel 6 für einen Neustart des Systems, welche ebenso durch *halt* bzw. *reboot* ausgelöst werden. Daneben gibt es Runlevel für den Mehrbenutzerbetrieb mit Netzwerkbetrieb und X-Server, sowie andere Runlevel ohne eine dieser beiden Komponenten. Überdies gibt es den Single-User-Mode zur Pflege des Systems. Die unterschiedlichen Runlevel starten unterschiedliche Hintergrundprozesse (Deamons), welche ebenso in inittab definiert werden können.

[11] Es wird nicht die reale, auf die Information bezogene Übertragungsrate in bps (bit per seconds) angegeben, sondern die Rate der elektrischen Signalwechsel in baud (benannt nach Emile Baudot, dem Erfinder des asynchronen Telegraphen), welche sich in Abhängigkeit von der Kodierung und der Kompression auf die Informationsübertragung umgerechnen läßt.

(*Point to Point Protocol*) oder dem etwas einfacherem SLIP (*Serial Line Internet Protocol*) TCP/IP werwendet werden, um darauf anschließend Protokolle wie NFS, ssh, ftp oder telnet laufen zu lassen. Dazu müssen SLIP und TCP/IP in den Kernel kompiliert sein. Für eine Direktverbindung gibt es das sehr einfach zu bedienende Programm *slattach*, welches SLIP auf eine serielle Schnittstelle unter Vorgabe einer Übertragungsrate betreibt:

```
slattach -p slip /dev/ttySn -s 115200
```

Slattach generiert automatisch die Schnittstelle sl0 und mit jedem weiteren Aufruf eine fortnumerierte Schnittstelle sln, die für TCP/IP konfiguriert werden können

```
ifconfig sl0 (eigene IP-Adresse) pointopoint (IP-Adresse des anderen Computers)
```

Diese beiden Zeilen können in das Startskript integriert werden, sodaß nach dem Booten die Direktverbindung automatisch hergestellt wird.
Der Vorteil gegenüber dem Terminalemulator ist, daß hierbei alle üblichen Netzwerkdienste zur Kommunikation mit dem eingebetteten System verwendet werden können. Beispielsweise wird es möglich das Wurzelverzeichnis des eingebetteten Systems mit NFS in ein Linuxverzeichnis des PCs einzuhängen oder per SMB im Windowsnetzwerk sichtbar zu machen. Ebenso können auf dem eingebetteten System Dienste wie ssh und ftp zur Verfügung gestellt werden.

Kapitel 5

Linuxinstallation auf den NATHAN-Modulen

Wenn der Linuxkernel für den PowerPC 405 generiert werden kann und das Wurzelverzeichnis in der Simulationsumgebung erfolgreich getestet und cross-kompiliert wurde, so ist der Kernel selbst noch an die Umgebung auf den NATHAN-Modulen anzupassen. Die Peripherie bestimmt den Systemstart und daraufhin die untere Protokollebene des externen Zugriffs auf das laufende Betriebssystem.
Der Systemstart erfolgt bei eingebetteten Systemen in den überwiegenden Fällen von einem Boot-ROM[1], in welchem das Kernelimage und das Ramdiskimage dauerhaft gespeichert sind. Da die NATHAN-Module mit einem PC verbunden sind, können zu deren Systemstart das Kernelimage und das Ramdiskimage aus dem PC bezogen werden. Die Slow-Control ermöglicht es, vom PC aus die einzelnen Speichermodule auf einem NATHAN auszulesen und zu beschreiben. Das Kernelimage und die Ramdisk können auf diese Weise in das DDR RAM geschrieben werden. Dabei müssen alle Maßnahmen zur Initialisierung des Kernels getroffen werden, die üblicherweise die Aufgabe eines Bootloaders sind.

Für die Organisation des Systemzugriffs kann entweder die Peripherie an den Kernel angepaßt werden oder dieser an die Peripherie. Die erste Möglichkeit besteht darin dem Kernel eine Schnittstelle auf dem FPGA zur Verfügung zu stellen, für die er bereits über einen Gerätetreiber verügt. Die naheliegendste Ausführung ist eine serielle Schnittstelle mit einem UART 16550. Zum einen ist der VHDL[2]-Code für diese Standardschnittstelle verfügbar und muß nicht eigens programmiert werden. Der vorhandene Code kann direkt auf dem Virtex-II Pro als Verbindung des PowerPCs mit der Slow-Control simuliert werden. Zum anderen ist die serielle Schnittstelle mit dem UART 16550 die bei der Simulation des eingebetteten Systems auf dem PC verwendete Schnittstelle, für welche das Wurzelverzeichnis und die Initialisierungsskripte bereits ausgelegt und getestet wurden. Die Implementierung einer seriellen Schnittstelle auf dem Virtex-II Pro kann zu Testzwecken hilfreich sein, stellt jedoch keine elegante Lösung dar. Da die Slow-Control Daten paketweise erhält und versendet, ist die Aufteilung einzelner Bytes in serielle Bits

[1] Read Only Memory
[2] Very high integrated circuit Hardware Description Language

durch den UART prinzipiell unnötig. Zweckmäßiger ist es, einen Gerätetreiber für die Slow-Control im Kernel zu implementieren.

5.1 Vorabinitialisierung des Kernels

Der Linuxkernel benötigt zu seiner Initialisierung einige Daten über die zugrundeliegende Hardware. Ihm müssen die Größe des verfügbaren Arbeitsspeichers, die Adressen der angeschlossenen Geräte, sowie die Konfiguration der Peripheriebusse und deren Geräte bekannt sein. Auf einem PC bezieht der Kernel diese Informationen aus dem BIOS (Anhang D). Darüberhinaus muß der Kernel seine Speicherposition und die Adresse des Wurzelverzeichnisses kennen. Befindet sich das Wurzelverzeichnis in einer Ramdisk, so wird diese zusammen mit dem Kernel von einem Bootloader in den Speicher geschrieben. Die Startadresse der Ramdisk wird dem Kernel vom Bootloader übergeben.

Bei der Kernelinitialisierung in head_4xx.S (Anhang E) werden die vom Bootloader übergebenen Daten in den Registern r3 bis r7 erwartet. Dies sind 5 Parameter mit der Größe von jeweils 4 Byte. Der erste Parameter wird als Verweis auf eine Adresse interpretiert, an welcher sich Informationen über das verwendete Board zu befinden haben. Im vorliegenden Falle ist diese Adresse mit solchen Information zu belegen, welche dem Virtex-II Pro entsprechen. Die weiteren 4 Parameter werden als die Startadresse der Ramdisk, die Größe der Ramdisk und die Position der Kernelparameter interpretiert:

mr r31,r3 #Adresse der Boardinfo
mr r30,r4 # Startadresse der Ramdisk
mr r29,r5 # Größe der Ramdisk
mr r28,r6 # Startadresse des Strings mit den Kernelparameter
mr r27,r7 # Endadresse des Strings mit den Kernelparametern

Die Aufgaben eine Bootloaders sind es, das Kernelimage und das Ramdiskimage in den Speicher zu schreiben und die notwendigen Informationen in den Registern r3 bis r7 zu übergeben. Der erste Registerinhalt bedarf einer näheren Erläuterung. In diesem wird der Verweis auf eine Struktur erwartet, die in dem Verzeichnis /arch/ppc/platforms für die einzelnen Plattformen definiert wird. Die jeweilige Plattform wurde beim kompilieren des Kernels ausgewählt. In /platforms/walnut.h[3] ist dies die folgende Struktur:

typedef struct board_info { unsigned char bi_s_version[4]; /* Version of this structure */
unsigned char bi_r_version[30]; /* Version of the IBM ROM */
unsigned int bi_memsize; /* DRAM installed, in bytes */
unsigned char bi_enetaddr[6]; /* Local Ethernet MAC address */
unsigned char bi_pci_enetaddr[6]; /* PCI Ethernet MAC address */
unsigned int bi_procfreq; /* Processor speed, in Hz */
unsigned int bi_plb_busfreq; /* PLB Bus speed, in Hz */

[3]Die IBM Testplattform für den PPC 405, auf der Linux zum ersten mal erfolgreich gestartet werden konnte.

unsigned int bi_pci_busfreq; /* PCI Bus speed, in Hz */ }
bd_t;

Die Informationen der Bootinfo-Struktur werden vom Kernel in den Initialisierungsroutinen *identify_machine* und *MMU_init* beim Kernelstart ausgewertet (Anhang E).

Zum Booten des eingebetteten Systems auf den NATHAN-Modulen ist ein Speicherbereich zu reservieren, welcher mit dem angepaßten Inhalt dieser Datenstruktur für den Virtex-II Pro belegt wird. Desweiteren muß zusammen mit dem Kernel- und dem Ramdiskimage ein kleiner Programmcode im CPU-Reset-Vektor stehen, welcher die Register r3 bis r7 beschreibt und anschließend zur Startadresse des Kernelimages springt. Dem Register r3 ist dabei die Adresse der Bootinfo-Struktur im Speicher zu übergeben.

5.2 Anbindung des Kernels an die Slow-Control

5.2.1 Die Schnittstelle zur Slow-Control

Das Netzwerk, welches die einzelnen NATHAN-Module und Darkwing verbindet, ist durch eine ringförmige Struktur gekennzeichnet. Die Slow-Control wird mit einem Token-Ring-ähnlichem Protokoll [Tan88] betrieben. Die Schnittstelle zur Slow-Control hat auf den NATHAN-Modulen die folgende Registerstruktur.

Für die Adressierung ist der Bereich von dem 18. bis zum 24. Byte vorgesehen. Das 19. Byte (0x13: destination) kennzeichnet den Teilnehmer innerhalb des Token-Ring Netzwerks, welches eines von 16 NATHAN-Modulen oder Darkwing sein kann. Das 18. Byte (0x12: modul) bestimmt das Ziel innerhalb des Teilnehmers. Dabei kann es sich beispielsweise um das DDR-SDRAM oder das SRAM eines anderen NATHANs handeln. Das 32 Bit breite Register ab dem 20. Byte (0x14) ist für die Speicheradresse innerhalb des jeweiligen, durch das 20. Byte festgelegten, Adreßraums vorgesehen. Durch den 32 Bit breiten Adreßraum können bis zu 4 GB DDR RAM auf jedem Modul adressiert werden.
Die Register 0x18 und 0x1C beinhalten die eigentlichen Daten von 64 Bit. Beim Zugriff auf das Netzwerk wird das 16. Byte (0x10: Status) als Steuer- und Statussignal verwendet. Zum Senden und Empfangen muß das Token im Netzwerk angefragt werden. Eine Anfrage wird durch das erste Bit (request) signalisiert. Das zweite Bit (acknowledge) bestätigt die Ankunft von Daten und das vierte Bit (error) signalisiert einen Fehler. Das 17. Byte kennzeichnet bei einer Anfrage, ob diese zum Empfangen oder Senden von Daten gestellt wird. Der Wert 0x88 (write) legt einen schreibenden Zugriff, der Wert 0x8A einen lesenden Zugriff auf das Netzwerk fest.
Die Byteanordung innerhalb der Register verwendet little-endian, bei der das signifikanteste Byte die höchste Registeradresse zugewiesen bekommt. Um beispielsweise Daten zum zweiten Modul des ersten Teilnehmers zu senden, muß das Datenwort im 4. Register 0x01.02.88.01 betragen.

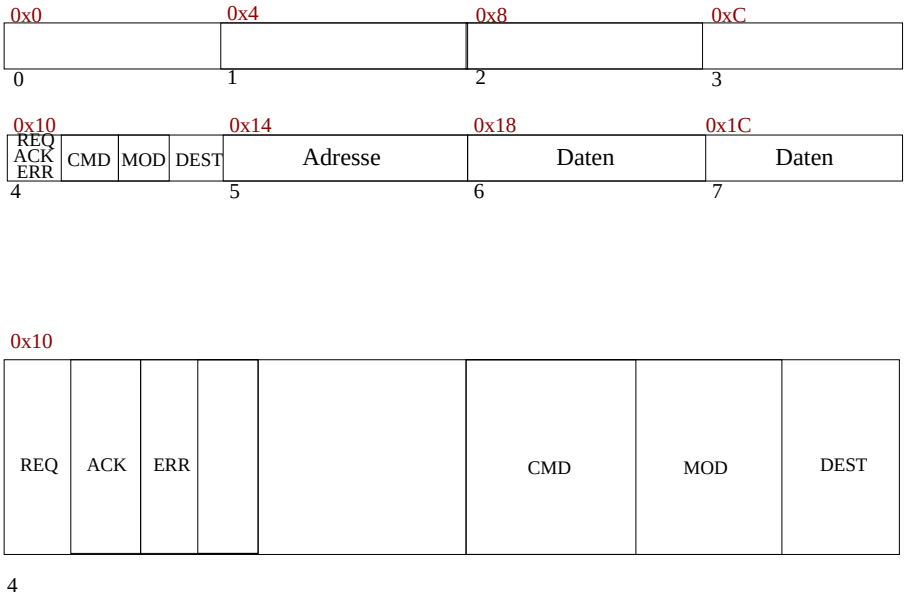


Abbildung 5.1: *Die Schnittstelle zur Slow-Control. Oben sind die 8 Register der Slow-Control auf NATHAN dargestellt, rot die byteweise, schwarz die wortweise Numerierung. Die Register 6 und 7 sind für die Daten vorgesehen, die Register 4 und 5 bedürfen einer genaueren Erläuterung. Im Register 4 (unten dargestellt) signalisiert das erste Bit die Anfrage nach Daten, das zweite Bit die Bestätigung angekommener Daten und das dritte Bit einen Fehler, beispielsweise durch eine Zeitüberschreitung bedingt. Im zweiten Byte (cmd) wird der lesende oder schreibende Zugriff festgelegt, im vierten Byte (dest) der Teilnehmer und im dritten Byte (mod) das einzelne Modul innerhalb eines Teilnehmers. Da solch ein Modul beispielsweise ein Speichermodul auf einem NATHAN sein kann, dient Register 5 zur Adressierung innerhalb des Speichers. Bei dem Gerätetreiber für die Slow-Control als Schnittstelle zwischen dem eingebetteten System auf NATHAN und dem Host-PC ist die Adresse festgelegt und das Register 5 kann zur Kennzeichnung der Größe eines Datenpakets verwendet werden.*

5.2.2 Der Gerätetreiber für die Slow-Control

Bei der Kommunikation mit der Slow-Control werden jeweils höchstens bis zu 8 Byte (Zeichen) gesendet und empfangen. Die Slow-Control wird unter Linux somit als zeichenorientiertes Gerät betrachtet[4]. Sie sendet kein Interrupt-Signal und muß daher zyklisch abgefragt werden. Es bestehen prinizpiell die zwei Möglichkeiten, die Slow-Control in den 4 GB großen Speicheradreßraum oder in den 64 KB großen Peripherieadreßraum des PowerPC 405 abzugebilden (Abschnitt 3.1). In jedem Fall muß der entsprechende Adreßbereich von 16 Byte (sc_start bis sc_end) im Kernel angefragt und allokiert werden. Im Peripherieraum erfolgt die Anfrage durch *check_region* die Allokierung durch *request_region*, im Speicheradreßraum durch *check_mem_region* und *request_mem_region*

[4]Blockorientierte Geräte arbeiten mit Datenblöcken im mehrstelligen Kilobyte-Bereich.

```
#include <linux/ioport.h>
...
unsigned long sc_start=SLOW_CONTROL_START+0x10;
unsigned long sc_end=SLOW_CONTROL_END;
unsigned long sc_range= sc_end - sc_start;
char sc_status=sc_start+ 0x3; //little endian
char sc_com=sc_start+ 0x2;
...
if (err= check_region(sc_start, sc_range) ) return err;
request_region(sc_start, sc_range, „slcont");
return(0);
```

In diesem Beispiel bekommt die Slow-Control den Gerätenamen „slcont". Es ist ein entsprechender Eintrag im Wurzelverzeichnis anzulegen:

```
mknod /dev/slcont  c  254  0
```

Darüberhinaus muß das Gerät unter der hier gewählten Major Nummer 254 im Kernel registriert werden:

```
#include <linux/fs.h>
...
if (err= register_chrdev(254, „slcont", &slcont_fops) ) return err;
```

Dabei ist *slcont_fops* ein Zeiger auf eine Datenstruktur, in welcher alle Dateioperationen (*Methoden*) vermerkt sind. Die Methoden werden bei den entsprechenden Systemaufrufen durch die Serviceroutinen gestartet (Abschnitt 2.5). Alle Methoden der Datei slcont müssen in der Struktur *slcont_fop* festgelegt werden:

```
struct file_operation slcont_fops = {
read:   slcont_read,
write:   slcont_write,
open:   slcont_open,
release   slcont_release,
}
```

Die hierbei verwendete Tag-Initialisierung ist keine im Standard-C unterstützte Konstruktion, sondern eine Erweiterung des GNU C-Compilers.

Die Methoden zum Lesen und zum Schreiben transportieren Daten zwischen dem virtuellen Adreßraum des Benutzers (*User-Space*) und dem des Kernels (*Kernel-Space*). Dazu sind in Linux die Funktionen *copy_to_user* und *copy_from_user* vorgesehen. Wie in Abschnitt 2.5 erläutert, stellt das Anwendungsprogramm bei den Systemaufrufen den Zeiger auf eine Puffervariable bereit. Die Methoden slcont_read und slcont_write verwenden beim Zugriff auf diesen Puffer die beiden oben angeführten Funktionen:

```
    ssize_t slcont_read(struct file *filp, char *buf, ssize_t count, loff_t *f_pos)
{
    ...
    if (copy_to_user(buf, kbuf, count))  return (-EFAULT);
    ...
    }
```

Dabei ist *buf* der Zeiger auf den Puffer im User-Space und *kbuf* der Zeiger auf den Puffer im Kernel-Space, *count* ist die Anzahl der zu übertragenden Bytes, f_pos der Positionszeiger der Datei und filp der *Filepointer*. Letzterer Zeigt auf das beim Öffnen der Datei angelegte Dateiobjekt.
Zum Beschreiben eines Bytes in einem IO-Port stellt der Kernel die Funktion *outb* zur Verfügung. Zum Beschreiben von IO-Speicher *writeb*. Die entsprechenden Funktionen zum Lesen eines Bytes sind *inb* und *read*b. Für den wortweisen Zugriff heißen die Funktionen *in*l, *out*l, *readl* und *write*l. Diese Funktionen verdecken die architekturspezifischen Eigenschaften und ermöglichen es, platformunabhängige Treiber zu schreiben. Sie werden in linux/include/asm-ppc/io.h für den PowerPC definiert.

Die Lesemethode allokiert zu Beginn einen Puffer *kbuf* in der Größe von count. Da die Slow-Control keinen Interrupt sendet, ist ein zyklisches Abfragen erforderlich. Das Prinzip bei der Lesemethode kann wie folgt dargestellt werden:

```
    while(kbuffersize<count){
        // weitere Bytes (hier auf 256 Bytes beschränkt)
        outb((count-kbuffersize), sc_addr)
        // Anfrage nach Daten stellen (set_request=0x01)
        outb(set_request, sc_status);
          //zyklisches Abfragen
          while(inb(sc_status&set_ack) =0);
        // Daten in kbuf schreiben
        add_to_kpuf(inl(sc_data1), inl(sc_data2), (count-kbuffersize));
        //Empfang bestätigen (set_ack=0x02)
        outb((set_ack, sc_status);
        kbuffersize+=8;
}
```

Das Adreßregister sc_addr beinhaltet hierbei die Größe des angefragten Datenblocks. Es hat nur beim Zugriff auf den Speicher einzelner Module den Zweck einer Adressierung. Im Treiber sind überdies noch eine Fehlerabfrage und eine Zeitabschaltung (Timeout) vorhanden.
Ein Treiber für eine unterbrechungsgetriebene Slow-Control würde statt des zyklischen Abfragens *schedule*() aufrufen und den nachfolgenden Teil in der Behandlungsroutine des entsprechenden Interrupts bearbeiten.

Kapitel 6

Zusammenfassung

In der vorliegenden Diplomarbeit wurde ein vollständiges Linuxsystem für einen eingebetteten PowerPC 405 Prozessor in den Evolutionsmodulen eines verteilten neuronalen Netzwerkes entwickelt. Die Vorgehensweise erfolgte in drei Schritten. Zunächst galt es, die Entwicklungswerkzeuge zu generieren, mit welchem der Linuxkernel für den PPC 405 kompiliert werden kann. Nach der erfolgreichen Erstellung des Kernels ließen sich die Systemprogramme ebenso kompilieren. Es wurde ein vollständiges Betriebssystem zusammengestellt, das den Besonderheiten der auf den NATHAN-Modulen vorliegenden Peripherie Rechnung trägt. Zuletzt wurde das System für eine Installation auf den NATHAN-Modulen eingerichtet. Die Installation des entwickelten Betriebssystems konnte im Zeitraum dieser Arbeit nicht vorgenommen werden, da die Hardware-Ansteuerung des PPC noch nicht gelungen war.
Im Laufe der Arbeit ergaben sich unerwartete Schwierigkeiten bei der Generierung des Linuxkernels für den PowerPC 405. Dabei wurde deutlich, daß die Portierung des Linuxkernels nicht nur die Unterstützung der Befehlssatzarchitektur des Zielrechners durch die Entwicklungswerkzeuge erfordert, sondern diese auch ein spezielles Binärformat generieren müssen, welches in den PowerPC-Assemblerroutinen des Linuxkernels verwendet wird. Zur Installation auf den NATHAN-Modulen wurde ein Gerätetreiber für die Schnittstelle der Module geschrieben. Weiter wurde die Initialisierung des Kernels untersucht und die Anforderungen an den Bootloader erläutert. Gegen Ende der Arbeit stand ein Testsystem zur Simulation des Kernels auf dem PowerPC 405 zur Verfügung, mit welchem der Startvorgang nachvollzogen werden kann. Auf dem Simulationssystem konnten erste Bootmeldungen verfolgt werden, jedoch noch kein vollständiger Kernelstart beobachtet werden.

Anhang A

Die Bezugsquellen der Software

Freie Software gründet auf gemeinschaftliche Arbeit mit Hilfe des Internets, weshalb das Internet als die erste Bezugsquelle für freie Softwareprodukte anzuführen ist. Der Linux-Kernel und die GNU-Programme befinden sich auf zahlreichen FTP- oder HTTP-Servern, unter anderem auch auf den Servern der einzelnen Linux-Distributionen. Die Distributionen verschiedener Anbieter werden zudem auf DVDs und CDs veröffentlicht. Im folgenden sind einige Beispielquellen im Internet angeführt:

Der offizielle Linuxkernel in der stabilen Version 2.4

- ftp://ftp.kernel.org/pub/linux/kernel/
- http://www.kernel.org/pub/linux/kernel/
- ftp://ftp.de.kernel.org/pub/linux/kernel/ (Deutscher FTP-Server)
- http://www.de.kernel.org/pub/linux/kernel/ (Deutscher HTTP-Server)

Neben den von Linus Torvalds verwalteten Versionen des offiziellen Linuxkernels gibt es eine Reihe von abweichenden Kernelversionen eigenständiger Entwicklungsprojekte. Solche Projekte verfolgen unterschiedliche Ziele, die sich auf Erweiterungen durch spezielle Leistungsmerkmale oder die Portierung auf verschiedene Hardwarearchitekturen beziehen.

Eigenständige Kernelentwicklungen der Version 2.4 für PowerPC-Architekturen

- http://penguinppc.org/
- ppc.bkbits.net:8080/linuxppc_2_4 (per Bitkeeper)
- source.mvista.com/linuxppc_2_4 (per Bitkeeper)
- MontaVista: www.mvista.com (Preview Kit)

Die per Bitkeeper zu beziehenden Kernelquellen erfordern die Installation der neusten Bitkeeper-Software von *www.bitkeeper.com*. Das Herunterladen der Kernelquellen geschieht mit dem Aufruf **bk** im Kommandozeileninterpreter. Für den Server von BitMover lautet das Kommando

bk clone bk://ppc.bkbits.net/linuxppc_2_4 <Verzeichnis>

und für den Server von Monta Vista entsprechend

bk clone bk://source.mvista.com/linuxppc_2_4 <Verzeichnis>

Beim Zugriff über einen Proxy-Server ist zuvor die folgende Variable zu setzen:

export http_proxy=http://<Name des Proxy-Servers>:8080/
bk clone -d http://ppc.bkbits.net/linux_2_4 <Verzeichnis>

Zudem bietet Monta Vista den eigenen Linuxkernel zusammen mit einer fertigen Entwicklungsumgebung in einem Preview Kit zum Herunterladen an.

GNU-Programme

Eine Liste mit zahlreichen Spieglungen der GNU-Programme befindet sich auf

- http://www.gnu.org/order/ftp.html.

Die offizielle Seite ist

- ftp://ftp.gnu.org/pub/gnu

Busy Box

- http://www.busybox.net/

Newlib

- /http://sources.redhat.com/newlib/

VMware

VMware bietet eine zeitlich befristete Probeversion der VMware Workstation an.

- http://www.vmware.com/download/

Anhang B

Struktur und Organisation des Kernelquellcodes

Der Quellcode von Linux ist durch die von der Sprache C unterstützte modularisierte Programmierung in Dateien strukturiert. Die einzelnen Quelltextmodule kapseln eine bestimmte Funktionalität. Die Prototypen extern definierter Funktionen, sowie die Definition von Strukturen und Makros sollten sich grundsätzlich in Headerdateien befinden. Im allgemeinen sind diesen alle Modulabhängigkeiten zu entnehmen. Es gibt jedoch Ausnahmen. Die Definition einer speziellen Funktion findet kann mit *grep* aufgesucht werden:

grep -r -l '*Rückgabewert Funktionsname*(' *Verzeichnispfad*

Beispielsweise:

grep -r -l 'int printk(' /usr/src/linux/kernel

Die Grundstruktur des Kernelquellcodes ist durch die Aufteilung in Unterverzeichnisse gegeben. Die folgende Abbildung gibt einen Überblick auf die Verzeichnisstruktur der Kernelquellen.

Jedes Verzeichnis hat ein eigenes Makefile. Das zentrale Makefile befindet sich in obersten Verzeichnis. Der gesamte architekturspezifische Quellcode ist im Verzeichnis arch/ und für Header-Dateien in den Verzeichnissen include/asm-x vom übrigen Quellcode separiert. In arch/ befinden sich Unterverzeichnisse für die einzelnen Architekturen, welche von Linux unterstützt werden. Das Verzeichnis arch/ppc/ beinhaltet den Code für alle 32 Bit PowerPC-Architekturen.

|– Documentation
|– arch (*architekturspezifischer Code*)
| |– alpha
| |– arm
| |– cris
| |– i386
| |– ia64
| |– .
| |– ppc
| |– .
| |– .
| |– .
|– drivers (*Gerätetreiber*)
|– fs (*Dateisystem*)
|– include (*Headerdateien des Kernels*)|
|asm
|asm-alpha
|..
|..
|asm-ppc
|..
|– init (*architekturunabhängige Initialisierungsroutinen*)
|– ipc (*SystemV-Interprozeßkommunikation*))
|– kernel (*Der übrige Kernelcode*)
|– lib (*Kernelbibliotheken*)
|– mm (*Speicherverwaltung*)
|– net (*Netzwerk*)
|– scripts (*Skripte*)

Anhang C

Konstruktion einer GNU Toolchain

Die GNU Toolchain besteht aus 3 Programmpaketen:

- Dem GNU C-Compiler (gcc)[1]
- Den GNU Binary Utilities[2]
- Der GNU C-Library

Die Bezugsquellen hierfür sind in Anhang A angeführt. Bei der Wahl des Compilers ist zu beachten, daß erst die neueren Versionen ab gcc-2.95 ohne weiteres als Cross-Compiler zu verwenden sind. Für ältere Versionen müssen entsprechende Patches eingespielt werden. Eine Alternative zu den älteren Versionen ist der egcs, bei dem es sich um eine zeitweilige Zweigentwicklung des gcc handelt, die seit der Version 2.95 wieder mit dem gcc vereint ist. Bei den neueren gcc-Versionen bis 3.2 variiert die Generierung eines Bootstrap-Compiler, da die Option –without-headers fehlerhaft ist. Bei der Wahl eines Compilers zur Kernelkompilierung sollten die mit den Kernelquellen mitgelieferten Empfehlungen berücksichtigt werden, welche in der Datei *Changes* im Verzeichnis /linux/Documentation/ nachzulesen sind. Die Binary Utilities sind in der vorgeschlagenen oder einer neueren Version zu verwenden. Insbesondere sind alte Versionen der GNU-Binutils mit den neueren Versionen des gcc in Hinblick auf das Application Binary Interface inkompatibel. Dies verursacht Fehlermeldungen des Assemblers, die auf unbekannte Relokationssymbole (*unknown relocation type*) verweisen. Bei der Wahl der Bibliotheken ist neben Aspekten der Kompatibilität auch die Größe zu berücksichtigen, welche in der Regel mit steigender Versionsnummer anwächst. Die Bibliotheken sind maßgeblich für den Speicherbedarf eingebetteter Systeme verantwortlich und nicht selten wesentlich größer als der gesamte Kernel.
Die Schwierigkeit bei der Konstruktion einer GNU Toolchain besteht darin, eine geeignete Komposition der jeweiligen Paketversionen zu finden und diese richtig zu konfigurieren. Informationen über die Kompatibilität einzelner Komponenten befinden sich in beiliegenden Erläuterungstexten. In Mailinglisten [YAG03] und speziellen Internetseiten

[1] Inzwischen steht gcc für GNU Compiler Collection, da auch viele andere Programmiersprachen unterstützt werden, wie C++, Objective-C, Ada, Fortran oder Java

[2] Dieses Paket beinhaltet neben dem GNU Assembler und dem GNU Linker weitere Werkzeugen zum Bearbeiten der vom Compiler generierten Binär- oder Objektdateien.

[Keg03] sind weitere und oftmals korrigierende Hinweise zu finden.

Die Installation von GNU-Programmen erfolgt für gewöhnlich nach folgendem Muster:

- Entpacken der Pakete
- Einspielen von Patches, falls solche vorhanden und erforderlich sind
- Konfiguration des zu generierenden Programms
- Programmgenerierung
- Installation des Programms

Zum Entpacken muß je nach Kompressionsformat der Pakete

tar xfv *Dateiname*.tar,
tar xfvz *Dateiname*.tar.gz oder
bzip2 -d *Dateiname*.bz2

verwendet werden. Änderungen des Quellcodes sind vor der Programmgenerierung vorzunehmen. Das Einspielen sogenannter *Patches* bewirkt eine Änderung des Quelltextes. Patches dienen der inkrementellen Änderung von Dateien und werden meistens dazu verwendet, geringfügige Änderungen zur Fehlerbehebung, Aktualisierung oder spezifischen Anpassung am Quellcode größerer Programme vorzunehmen. Die Patches sind dabei wesentlich kleiner, als der gesamte Quellcode der jeweiligen Programme, weil sie nur die Änderungen enthalten. Für das Einspielen von Patches gibt es das Kommando *patch*. Zur Konfiguration von Programmpaketen, die als Quellcode vorliegen, gibt es das GNU-Werkzeug *autoconf* [McKED02]. Dieses ermöglicht es, die zu generierenden Programme den Erfordernissen des Anwenders und den jeweils zugrundeliegenden Hardwareplattformen und Betriebssystemen anzupassen. Die Anpassungen geschehen auf dem Zielcomputer durch die Ausführung des Skripts *configure*. Configure untersucht, ob alle zur Programmgenerierung erforderlichen Werkzeuge vorhanden sind und ist weiter in der Lage, einen Teil der benötigten Systeminformationen, wie den Prozessortyp, selbständig zu erfassen und für das jeweilige Programmpaket ein entsprechendes *Makefile* zu erstellen. Mittels dieses *Makefiles* kann das Kommando *make* [SGS02] alle einzelnen Programmteile durch Aufrufe und Optionen des Compilers und Linkers in der Weise erzeugen, daß diese anschließend auf dem Anwendersystem ausführbar sind. Die von make auszuführenden Schritte werden im Makefile bestimmt. Bei der Programmgenerierung mit der GNU-Toolchain hat der GNU C-Compiler die Funktion eines Kontrollprogramms. Er ruft die zur Programmgenerierung zusätzlich erforderlichen Werkzeuge selbständig auf. Darum müssen besondere Aufrufoptionen für den Linker und den Assembler dem GNU C-Compiler zusammen mit den eigenen Programmparametern übergeben werden. Diese Aufrufoptionen werden innerhalb eines Makefiles in die Variablen CFLAGS geschrieben. Die zu generierenden Programme der Toolchain sollen zwar auf dem eigenen Rechner ausgeführt werden, da der von ihnen erzeugte Code jedoch für einen anderen Rechner bestimmt ist, muß dessen Prozessortyp explizit angegeben werden. Configure ermöglicht

derartige Spezifizierungen durch Aufrufoptionen. Der Prozessortyp des Zielrechners wird mit dessen Betriebssystem durch die Option –target=*Prozessor-Betriebssystem* festgelegt. Für configure gibt es noch viele weitere Optionen, welche üblicherweise in einer beiliegenden README-Datei erläutert werden. Durch die Option –PREFIX=*Verzeichnis* wird das Installationsverzeichnis festgelegt. Nach abgeschlossener Konfiguration stellt *configure* ein vollständig angepaßtes Makefile zur Verfügung.
Die wichtigsten Variablen innerhalb eines Makefiles sind CC für den verwendeten Compiler und CFLAGS für die bereit erwähnten Aufrufoptionen des Compilers.
Es ist sinnvoll, die Target-Option und das Installationsverzeichnis für die Prefix-Option als Variable zu definieren, da sie bei der Generierung der Toolchain häufiger verwendet werden, beispielsweise:

TARGET=*powerpc-linux*
PREFIX=*/home/user/toolchain*

Neben powerpc-linux und powerpc-eabi unterstützt die GNU Toolchain zahlreiche andere Plattformen. Die Variable TARGET muß dazu entsprechend anders definiert werden [Sta02].

GNU Binutils

Die Konfiguration und Generierung der Binutils ist im allgemeinen unproblematisch und steht am Beginn, da diese zur anschließenden Generierung des Compilers benötigt werden:

```
./configure –target=$TARGET prefix=$PREFIX
make
make install
```

Im Verzeichnis $TARGET/bin/ befinden sich nach einer erfolgreichen Generierung Cross-Assembler (*powerpc-linux-as*), Cross-Linker (*powerpc-linux-ld*) und die anderen Binutils. Um diese im weiteren Vorgehen verfügbar zu machen, ist die Variable PATH entsprechend zu erweitern:

```
PATH=$PATH:$TARGET/bin
export PATH
```

Bootstrap-Compiler

Ein wenig umständlicher ist die Erzeugung des Cross-Compilers. Der gcc unterstützt eine Vielzahl von Sprachen, jedoch benötigt er schon zu seiner Generierung Bibliotheken der Zielplattform. Diese sind in diesem Entwicklungsstadium noch nicht vorhanden. Darum wird zunächst ein sogenannten Bootstrap-Compiler erzeugt, welcher lediglich die Sprache C unterstützt, dafür aber ohne Bibliotheken auskommt:

```
configure –target=$TARGET –prefix=$PREFIX \
–with-local-prefix=${PREFIX}/${TARGET} \ –disable-multilib \
```

–with-newlib –with-cpu=405 –enable-cxx-flags=-mcpu=405 –without-headers
\
–disable-nls –enable-threads=no –enable-symvers=gnu –enable-__cxa_atexit
\
–enable-languages=c –disable-shared

make all-gcc
make install-gcc

War die Programmerzeugung erfolgreich, so befindet sich der Cross-Compiler anschließend im Verzeichnis $PREFIX/bin unter dem Namen *$TARGET-gcc*, wobei *$TARGET* nach wie vor die gewählte Zielplattform ist, im oben angeführten Beispiel wäre der Name demnach *powerpc-linux-gcc*.

Kernel-HEADER

Zur Generierung der glibc ist es notwendig, die folgenden Headerdateien zur Verfügung zu stellen:

mkdir -p $PREFIX/$TARGET/include
cp -r $KERNELDIR/include/linux $PREFIX/$TARGET/include
cp -r $KERNELDIR/include/asm-ppc $PREFIX/$TARGET/include/asm
cp -r $KERNELDIR/include/asm-generic $PREFIX/$TARGET/include

Dabei bezeichnet $KERNELDIR das Wurzelverzeichnis mit der Kernelquellen. Es sind die Kernelquellen zu verwenden, welche im Anschluß an die Generierung des Cross-Compilers kompiliert werden sollen.

GNU C-Library

Mit dem Bootstrap-Compiler kann unter Verwendung der oben kopierten Kernelquellen die glibc generiert werden:

CFLAGS="$TARGET_CFLAGS" CC=${TARGET}-gcc AR=${TARGET}-ar RANLIB=${TARGET}-ranlib \
/configure –host=$TARGET –prefix=/usr –with-cpu=405
–enable-cxx-flags=-mcpu=405 –without-tls –without-__thread –enable-kernel=2.4.3 \
–without-cvs –disable-profile –disable-debug –without-gd –enable-clocale=gnu
\
–enable-add-ons=linuxthreads –with-headers=${PREFIX}/${TARGET}/include
make install_prefix=$PREFIX/$TARGET/ install

Vollständiger Compiler

Durch die vorangegangenen Schritte stehen die Bibliotheken zur Verfügung, welche zur Generierung eines vollständigen Compilers benötigt werden. Die Vorgehensweise ist wie

folgt:

```
./configure –target=$TARGET –prefix=$PREFIX
–with-headers=${PREFIX}/${TARGET}/include \
–disable-nls –enable-threads=posix –enable-symvers=gnu –enable-__cxa_atexit
\
–enable-languages=c,c++ –enable-shared –enable-c99 –enable-long-long
make
make install
```

Linuxkernel

Falls der Linuxkernel schon konfiguriert wurde, kann der Cross-Compiler an dieser Stelle in einer Kernelkompilierung getestet werden:

```
make ARCH=ppc CROSS_COMPILE=$PREFIX/bin/$TARGET- old-
config
make ARCH=ppc CROSS_COMPILE=$PREFIX/bin/$TARGET- dep
zImage
```

Anhang D

BIOS und Bootloader beim PC

Startvorbereitung beim PC durch das BIOS

Beim Einschalten des Computers werden die CPU-Register zurückgesetzt. Intel-Prozessoren befinden sich im Real Mode [Ti92]. Dabei werden Code-Segment-Register cs und der Programmzähler ip auf eine herstellerspezifische Adresse gesetzt, die oft als CPU-Reset-Vektor bezeichnet wird. Beim PC ist dies die physikalische Adresse 0x000FFFF0, an der sich ein Sprungbefehl des BIOS zu seinem Startprogramm befindet. Damit hat dieses zunächst die Kontrolle über den PC. Das BIOS erledigt im wesentlichen die folgenden Schritte:

- Identifikation der Hardware und Ausführen einer Reihe von Hardwaretests[1]
- Initialisieren der Hardware; insbesondere Zuweisung der Interrupt-Vektoren (IRQ numbers) und I/O-Ports
- Den als Bootsektor bezeichneten ersten Sektor der Diskette oder Festplatte lesen[2]
- Den im entsprechenden Bootsektor gefundenen Bootloader ab der physikalischen Adresse 0x00007C00 in den Arbeitsspeicher kopieren und durch Setzen des Programmzählers an diese Stelle dessen Ausführung veranlassen .

Zum Starten des Kernels von Diskette oder Festplattenpartition kann diese im Prinzip unformatiert sein. Es ist bei Start- oder Notfalldisketten üblich, den Linuxkernel roh auf die Diskette zu schreiben. Die Diskette enthält dann keine Blöcke, Inodes oder andere Strukturen eines Dateiensystems, sondern ausschließlich Bit für Bit, ab dem ersten Sektor, den mit einem integrierten Bootloader beginnenden Maschinencode des Kernelsimages (Abschnitt 4.2.2). Alternativ dazu kann man den Kernel mittels eines externen Bootloaders auch als Datei in einem bestehenden Dateisystem laden. In vielen Fällen ist das Kernelimage anfänglich komprimiert. In jedem Fall wird es zunächst in den Hauptspeicher eingelesen, wobei ein komprimierter Kernel sich im unmittelbar nächsten Schritt

[1] Sogenannter POST (Power-On Self-Test)

[2] Da eine Festplatte in mehrere Partitionen unterteilt sein kann, besitzt sie einen ausgezeichneten Bootsektor, Master Boot Record (MBR) genannt. Dieser kann eventuell einen Verweis auf den Bootsektor einer bestimmten Partition enthalten, so daß der Bootsektor dieser Partition dann von dem Prozessor ausgelesen und bearbeitet wird.

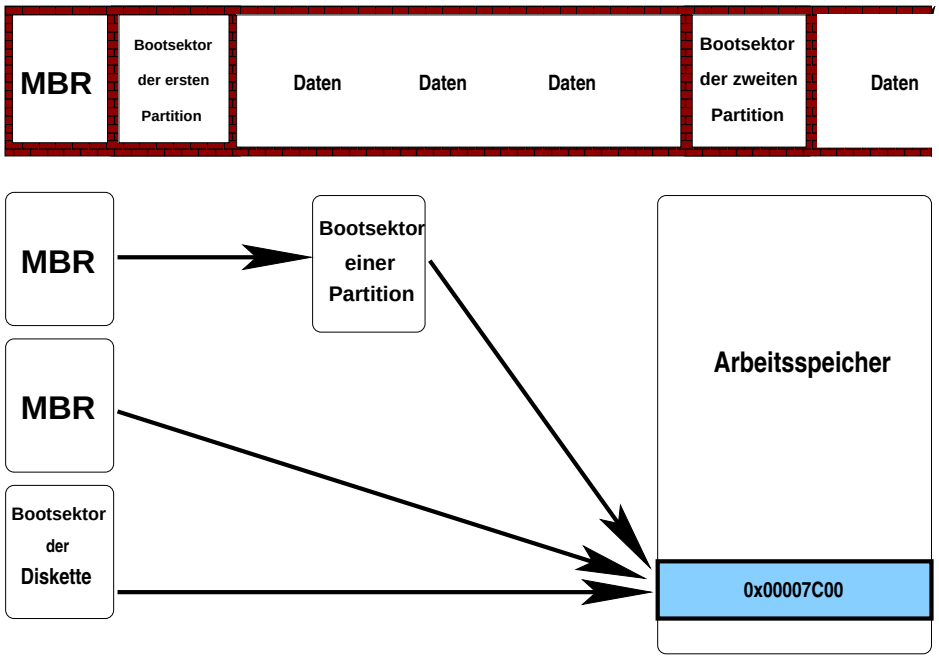


Abbildung D.1: *Die drei Möglichkeiten des Bootens auf dem PC. Jede Festplatte verfügt über ein Master Boot Record (MBR), welches als Bootsektor verwendet werden kann oder als Referenz auf einen anderen Bootsektor. Neben dem MBR verfügt jede einzelne Partition über einen eigenen Bootsektor. Diese können durch den MBR referenziert werden. Zudem kann von einer Diskette gebootet werden. Der jeweilige Bootsektor muß den Bootloader beinhalten und wird beim Starten des PCs vom BIOS an die Adresse 0x7C00 geladen. Bei einem eingebetteten System müssen der Bootloader, das Kernelimage und das Ramdiskimage ebenso in den Speicher geschoben werden. Auf dem NATHAN geschieht dies über die Slow-Control vom Host-PC aus.*

entkomprimiert und anschließend an der gleichen Adresse im Arbeitsspeicher liegt, wie der unkomprimierte Kernel, sodaß der Prozessor beginnen kann, die Maschinenbefehle des Kernels ab dieser Adresse sukzessive abzuarbeiten (Abschnitt 2.7.2).

Der Bootloader

Wurde ein externer Bootloader verwendet, so befinden sich dessen erste Instruktionen an der Stelle 0x00007C00 im Arbeitsspeicher. Beispielsweise ist der gängigste Bootloader *LiLo* [AW00] in zwei Teile aufgeteilt, weil er nicht als Ganzes in den Bootsektor paßt. Der erste Teil kopiert sich selbst von der Adresse 0x00007C00 an die Adresse 0x0009A000, bereitet einen Stack im Bereich 0x0009B000 abwärts bis 0x0009A200 vor und lädt den zweiten Teil in diesen Bereich des Arbeitsspeichers. Nach der Auswahl eines Linuxkernels aus dem LiLo-Auswahlmenü gibt LiLo die Meldung „Loading Linux" aus und kopiert den integrierten Bootloader des Kernels ab der Adresse 0x00090000, den Setup-Code ab der anschließenden Adresse 0x00090200 in den Speicher. Der Rest des Kernels wird entwe-

der bei 0x00010000 beginnend[3] in den tiefen[4] Arbeitsspeicher oder im Falle eines stark komprimierten großen Kernels[5]ab der Adresse 0x00100000 in den hohen Arbeitsspeicher kopiert. Dieser Speicherzugriff außerhalb des Realmode-Adreßbereichs geschieht, wie alle Speicherzugriffe bis dahin, mittels einer BIOS-Routine. Wird eine Ramdisk geladen, dann plaziert sie der Bootloader in einen möglichst hohen Speicherbereich, damit sie der Kernel bei seiner Initialisierung nicht überschreibt. Anschließend beginnt die Ausführung von Setup. Es wird hier nicht detaillierter auf den externen Bootloader eingegangen, da dieser typischerweise zur Auswahl eines von mehreren Betriebssystemen auf formatierten Speichermedien Verwendung findet, welche bei einem eingebetteten System nicht vorliegen.

Der Bootvorgang von Diskette und ohne einen externen Bootloader eignet sich eher als Vorbild für einen eingebetteten Kernel. Auch hier kopiert sich zunächst der kerneleigene Bootloader[6] von der Adresse 0x00007C00 an die Adresse 0x00090000 in den Arbeitsspeicher. Ein Stack wird von der Adresse 0x00003FF4 abwärts eingerichtet, sowie die nötigen Parameter, welche der Diskettenlaufwerkstreiber des BIOS benötigt. Das BIOS wird veranlaßt, auf der Konsole die Meldung „Loading" auszugeben, und über eine weitere BIOS-Routine wird der Initialisierungscode (Setup.S) ab der Adresse 0x00090200 in den Arbeitsspeicher geladen. Der Rest des Kernels wird wieder von einer BIOS-Routine entweder ab 0x00010000 oder bei einem stark komprimierten Kernel ab der Adresse 0x00100000 in den Arbeitsspeicher geladen. Den Abschluß bildet ein Sprung in den Initialisierungscode zur Adresse 0x00090200.

[3] bei den alten Kernels Image oder zImage

[4] Der im Real Mode adressierbare Arbeitsspeicher wird auch als tiefer Arbeitsspeicher bezeichnet, im Gegensatz zu dem hohen Arbeitsspeicher ab der physikalischen Adresse 0x00100000.

[5] bzImage

[6] /usr/src/linux/arch/i386/boot/boot.S

Anhang E

Der Kernelstart auf dem PowerPC 405

Kernelinitialisierung

Der Quellcode für die Kernelinitialisierung des PowerPC 405 befindet sich in *linux/arch/ppc/kernel/head_4xx.S*. Der Einsprungspunkt ist _GLOBAL(_stext) im ELF-Segment für die Programminstruktionen (*.text*).

Als erstes werden die vom Bootloader übergebenen Daten mit der Position der Ramdisk und den Kernelparametern, die sich in den zur Parameterübergabe dedizierten Registern r3 bis r7 befinden, in die Registern r27 bis r31 abgelegt und eine CPU-ID gesetzt.

Speicherresistente Daten speichern

mr r31,r3 # Adresse der Boardinfo
mr r30,r4 # Startadresse der Ramdisk
mr r29,r5 # Größe der Ramdisk
mr r28,r6 # Startadresse des Strings mit den Kernelparameter
mr r27,r7 # Endadresse des Strings mit den Kernelparametern

CPU-ID setzen

li r24,0

Anschließend werden alle Einträge im TLB gelöscht.

tlbia # (*tlb-invalid-all*)
sync #

Im folgenden wird die MMU initialisiert. Der hierbei verwendete Parameter KERNELBASE gibt die Basis des virtuellen Adreßraumes für den Kernel an. Dieser Wert wird in linux/include/asm-ppc/page.h mit dem Wert von PAGE_OFFSET belegt, welcher dort

standardmäßig auf 0xC0000000 gesetzt ist. Die virtuelle Kerneladresse wird in r3 und die physikalische Adresse in r4 gelegt. Dabei wird das in arch/ppc/kernel/ppc_asm.h definierte Makro tophys(rd,rs) verwendet, welches dort als addis rd,rs,-KERNELBASE@h definiert ist und schlicht den in KERNELBASE festgelegten Wert von der virtuellen Adresse subtrahiert.

lis r3,KERNELBASE@h # Lädt die virtuelle Adresse in r3
ori r3,r3,KERNELBASE@l #
tophys(r4,r3) # Lädt die physikalische Adresse in r4

Vor der Konfiguration des TLB muß die Prozeß-ID des Kernels im PID-Register gesetzt werden. Dieses ist, wie in Abschnitt 3.1 erläutert, Bestandteil der Adreßübersetzung. Die alte Prozeß-ID wird in r7 gespeichert und der erste Prozeß mit der Bezeichnung **Init-Task** bekommt die Prozeß-ID 0 zugewiesen.

mfspr r7,SPRN_PID # Save the old PID
li r0,0
mtspr SPRN_PID,r0 # Load the kernel PID

Im folgenden wird der generierte Eintrag in die Cacheline 0 des TLB geschrieben. Die gewählte Seitengröße beträgt 16 Megabyte

clrrwi r4,r4,10 # Seiteneintrag preparieren
ori r4,r4,(TLB_WR | TLB_EX) # Die Bits zum Schreiben und Ausführen setzen

clrrwi r3,r3,10 # Tag-Eintrag preparieren
ori r3,r3,(TLB_VALID | TLB_PAGESZ(PAGESZ_16M))

TLB_DATA ist ein Makro mit dem Wert 1 und TLB_Tag mit dem Wert 0. Das letzte Argument von **tlbwe** gibt an, um welchen Teil des TLB-Eintrags es sich handelt.

t**lbwe r4,r0,TLB_DATA** # Daten
tlbwe r3,r0,TLB_TAG # Tag
isync

mtspr SPRN_PID,r7 # Alte PID restaurieren

Hier wird die Basisadresse der Exception-Vektoren im EVPR eingerichtet (Abschnitt 3.1). Das Register EVPR wird von Linux wie das Register idtr bei Intel-Prozessoren verwendet (Abschnitt 2.5.2). Es muß die physikalische Adresse beinhalten, damit die CPU beim Einleiten einer Exception die Adresse des entsprechenden Exception-Vektor in der Tabelle finden kann.

lis r4,KERNELBASE@h # EVPR beinhaltet nur die signifikanteren 16 Bit

tophys(r0,r4) # Es wird die physikalische Adresse verwendet
mtspr SPRN_EVPR,r0

Die MMU wird aktiviert und zu dem Hauptteil des Initialisierungsprogramms **start_here** im unteren Teil des Quelltextes gesprungen. Das Makro MSR_DR steht für die Adreßübersetzung von Daten MSR_IR für die Adreßübersetzung von Instruktionen. Solange noch keine Adressen referenziert werden, ist die Adreßübersetzung nicht nötig. Die Aktivierung der Adreßübersetzung wird ab dieser Stelle erforderlich, um den Sprungbefehl return-from-interrupt (**rfi**) am Ende auszuführen. Die Sprungadresse für rfi befindet sich im Register SRR0 (Abschnitt 3.1)

mfmsr r0 # MSR laden
ori r0,r0,(MSR_DR | MSR_IR) # Virtual Mode für Daten und Instruktionen im MSR aktivieren
mtspr SPRN_SRR1,r0 # MSR beschreiben
lis r0,start_here@h # Sprungadresse in r0 schreiben
ori r0,r0,start_here@l
mtspr SPRN_SRR0,r0 # Neuen Programmzähler mit Sprungadresse initialisieren
rfi # Sprung nach start_here

In der letzten Zeile erfolgt der Sprung zu **start_here** im unteren Abschnitt der Datei. Dieser Hauptteil beginnt mit der Initialisierung der Sektion **BSS**, die im Executable and Linkable Format für uninitialisierte Daten vorgesehen ist und definitionsgemäß vom Betriebssystem mit Null initialisiert wird. Für die diese Initialisierung ist hier der Kernel selbst zuständig. Zuvor wird die Basisadresse für die SDA (Abschnitt 3.4) des aktuellen Prozesses (Init-Task) gesetzt. INIT_TASK_UNION wird in

start_here:

SDA-Basis-Zeiger des aktuellen Prozesses einrichten

lis r2,init_task_union@h
ori r2,r2,init_task_union@l

BSS-Segment löschen

lis r7,_end@ha # Startadresse des BSS-Segments
addi r7,r7,_end@l #
lis r8,__bss_start@ha # Endadresse des BSS-Segments
addi r8,r8,__bss_start@l #

Die Größe in Worten (4 Byte) berechnen

subf r7,r8,r7 # r7 = &_end - &_bss_start + 1
addi r7,r7,3 # r7 += 3

srwi. r7,r7,2 # *r7* = Größe in Worten zu 4 Byte.
beq 2f # Falls diese Größe Null ist überspringen
addi r8,r8,-4 # r8 -= 4
mtctr r7 # SPRN_CTR = Anzahl der zu löschenden Worte
li r0,0 # r0 = 0
3: stwu r0,4(r8) # Wort löschen
bdnz 1b

Im folgenden wird der Kernelstack des ersten Prozesses (Init-Task) eingerichtet. Dazu wird der Stackponter r1 gesetzt (Abschnitt 3.4) und das oberste Stack-Frame initialisiert. Unter Linux ist die Mindestgröße für ein Stack-Frame in include/asm-ppc/ptrace.h als STACK_FRAME_OVERHEAD mit der Größe von 16 Byte definiert. Der Prozeßdeskriptor task_struct wird mit dem Stack in einer gemeinsamen Struktur definiert. Die Definition befindet sich in arch/ppc/kernel/process.c.

2: addi r1,r2,TASK_UNION_SIZE # Stackpointer auf Beginn des Kernelstacks setzen
li r0,0
stwu r0,-STACK_FRAME_OVERHEAD(r1)

Die speicherresistenten Daten des Bootloaders werden zur Übergabe an die Funktion identify_machine zurückgeschrieben und diese aufgerufen.

mr r3,r31
mr r4,r30
mr r5,r29
mr r6,r28
mr r7,r27
bl identify_machine

Die Funktion **MMU_init** in arch/ppc/mm/init.c wird aufgerufen.

bl MMU_init

In beiden Funktionen werden die vom Bootloader übergebenen Parameter ausgewertet. In MMU_init wird der maximal verfügbare Speicher ermittelt. In identify_maschine werden die übrigen Übergabeparameter ausgewertet. Die Struktur der Übergabeparameter wird in Abschnitt 5.1 erläutert.

Nach dem Rücksprung aus den beiden Funktionen wird der Real Mode wird wieder hergestellt und im weiteren der Kernelkontext geladen.

lis r4,2f@h

```
ori r4,r4,2f@l
tophys(r4,r4)
li r3,MSR_KERNEL & ~(MSR_IR|MSR_DR)
mtspr SPRN_SRR0,r4 # Programmzähler setzen
mtspr SPRN_SRR1,r3 # MSR setzen
rfi
```

Der Kernelkontext wird in SPRG3 gespeichert. Anschließend wird die Adreßübersetzung der MMU entgültig aktiviert und in den Hauptteil des Kernels nach **start_kernel** in init/main.c gesprungen

```
lis r4,MSR_KERNEL@h
ori r4,r4,MSR_KERNEL@l
lis r3,start_kernel@h
ori r3,r3,start_kernel@l
mtspr SPRN_SRR0,r3 # Programmzähler setzen
mtspr SPRN_SRR1,r4 # MSR setzen
rfi # MMU aktivieren und in start_kernel springen
```

Literaturverzeichnis

[AW00] ALMESBERGER, W. : *Booting Linux: The History and the Future.* Proceedings of Ottawa Linux Symposium 2000, July 2000. http://www.almesberger.net/cv/papers.html

[BBD01] BECK, M. ; BÖHME, H. ; DZIADZKA M. ; KUNITZ U. ; MAGNUS R. ; SCHRÖTER C. ; VERWORNER D. : *Linux Kernelprogrammierung.* Addison-Wesley, 2001

[BC01] BOVET, D.P. ; CESATI M. : *Understanding the Linux Kernel.* O'Reilly, 2001

[Bec01] BECKER, J. : *Ein FPGA-basierten Testsystems für gemischt analog-digitale ASICs.* Universität Heidelberg, Diplomarbeit, 2001.

[BG03] BEEKMANNS, G. : *Linux From Scratch.* Version 4.1, 2003. http://archive.linuxfromscratch.org/lfs-museum/4.1/LFS-BOOK-4.1-HTML/

[Dev01] FELDMAN, D. L. : *Embedded Designs Move to Linux and Systems-on-a-Chip.* DeviceForge LLC - Articles & white papers about Linux in embedded applications, 2001 http://www.linuxdevices.com/articles/AT9656887918.html

[Dev02] SUBBARARO, A. : *The Technology behind LynxOS v4.0's Linux ABI compatibility.* DeviceForge LLC - Articles & white papers about Linux in embedded applications, 2002. http://www.linuxdevices.com/articles/AT8943314364.html

[Dev03] LEHRBAUM, R. : *Snapshot of the Embedded Linux market.* DeviceForge LLC - Articles & white papers about Linux in embedded applications, 2003. http://www.linuxdevices.com/articles/AT7301151332.html

[DS97] DAWSON, T. ; SÜTTERLIN, P. : *Linux NET-3 HOWTO.* Version 1.0-2, 1997. http://openfacts.berlios.de/index.phtml?title=Linux_NET-3_HOWTO

[FSF] Free Software Foundation: www.gnu.org/home.html

[Gat02] GATLIFF, B. : *BuildToolchainScript.* 31. Januar 2002. http://billgatliff.com/twiki/bin/view/Crossgcc/BuildToolchainScript

[GB00] GORTMAKER, P. ; BUDDE, M. : *Linux BootPrompt HOWTO.* Version 1.2. 20, 2000. http://www.linuxhaven.de/dlhp/HOWTO/DE-BootPrompt-HOWTO.html

[GnuM] Free Software Foundation: *GNU Manuals Online.* FSF, 2003. www.gnu.org/manual/manual.html

[Grü03] GRÜBEL, A. : *Entwicklung eines FPGA-basierten Evolutionssystems für gemischt analog-digitale ASICs.* Universität Heidelberg, Diplomarbeit, 2003.

[HHMK93] HETZE, S ; HOHNDEL, D. ; MÜLLER, M. ; KIRCH, O. : *LinuX Anwenderhandbuch und Leitfaden für die Systemverwaltung.* 3. LunetIX Softfair, 1993/94. ISBN 3-929764-02-4

[IBM00] IBM : *PowerPC 405GP Reference Board Manual.* Version 1.5. 2000. http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/79BE984407DFC29 C87256A0A006B7210

[IBM98] IBM : *The IBM PowerPC Embedded Environment - Architectural Specifications for IBM PowerPC Embedded Controllers.* Version 1.0, Second Edition, 1998. http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/PowerPC_ Microprocessors_and_Embedded_Processors

[IBMP98] IBM : Developing PowerPC Embedded Application Binary Interface (EABI) Compliant Programs. Version 1.0, 1998. http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/eabi_app.pdf

[Keg03] KEGEL, D. : *Building and Testing gcc/glibc cross toolchains.* http://www.kegel.com/crosstool/

[LINPPC] Linux on PowerPC Mail List Archives: http://lists.linuxppc.org/index.html

[McKED02] MACKENZIE, D. ; ELLISTON, B. ; DEMAILLE, A. : *Autoconf - Creating Automatic Configuration Scripts for version 2.57.* Free Software Foundation, Inc. , 2002. http://www.gnu.org/manual/autoconf-2.57/autoconf.html

[MO02] MATZIGKEIT, G ; OKUJI, Y. K. : *The GRUB Manual.* Free Software Foundation, Inc. , 2002. http://www.gnu.org/manual/grub-0.92/grub.html

[MOT95] MOTOROLA : *PowerPC Embedded Application Binary Interface.* Motorola, Inc. , 1995. www.cloudcaptech.com/MPC555%20Resources/ Programming%20Environment/ppceabi.pdf

[MP43] McCULLOCH, W. ; PITTS W. : *A logical calculus of the ideas immanent in nervous activity.* Bulletin of Mathematical Biophysics, Vol. 5, 1943

[MV01] IBM PowerPC 405GP Linux Port Status Page. http://dev.mvista.com/projects/405gp_status/

[PH96] PATTERSON, D. A. ; HENNESSEY, J. L. : *Computer Architecture: a quantitative approach.* 2. Morgan Kaufmann Publishers, Inc. , 1996

[PH98] PATTERSON, D. A. ; HENNESSEY, J. L. : *Computer Organisation & Design.* 2. Morgan Kaufmann Publishers, Inc. , 1998

[Pot94] PODSCHUN, T. : *Das Assembler-Buch.* 2. Addison-Wesley, 1994

[RC02] RUBINI, A. ; CORBET J. : *LINUX Gerätetreiber.* 2. O'Reilly, 2002

[Roj96] ROJAS, R. : *Theorie der neuronalen Netze.* 4. Springer, 1996

[Ros58] ROSENBLATT, F. *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological Review, Vol. 65, 1958

[RQ01] Russell, R. ; Quinlan, D. : *Filesystem Hierarchy Standard.* http://www.pathname.com/fhs/2.2/

[Sche01] SCHEMMEL, J. ; MEIER, K. ; SCHÜRMANN, F. : *A VLSI Implementation of an Analog Neural Network suited for Genetic Algorithms.* In: 4th International Conference on Evolvable Systems (ICES2001), 3-5 Oct, 2001, Tokyo

[Sche02] SCHEMMEL, J. ; SCHÜRMANN, F. ; HOHMANN, S. ; MEIER, K. : *An Integrated Mixed-Mode Neural Network Architecture for Megasynapse ANNs.* In: *World Congress of Computational Intelligence* (WCCI2002), 2002

[Sche03] SCHEMMEL, J. ; HOHMANN, S. ; MEIER, K. ; SCHÜRMANN, F. : *A Mixed-Mode Analog Neural Network using Current-Steering Synapses.* In: *Kluwer Academic Publishers*, Niederlande, 2003

[Schm03] SCHMITZ, T. , HOHMANN, S. ; MEIER, K. ; SCHEMMEL, J. ; SCHÜRMANN, F. : *Speeding Up Hardware Evolution: A Coprocessor for Evolutionary Algorithms.* In: *International Conference on Evolvable Systems* (ICES) - Proceedings of the ICES 2003, pages 274-285, Springer Verlag, 2003

[Schü02] SCHÜRMANN, F. ; HOHMANN, S. ; SCHEMMEL, J. ; MEIER, K. : *Towards an Artificial Neural Network Framework.* In: STOICA, A. (Hrsg.) ; LOHN, J. (Hrsg.) ; KATZ, R. (Hrsg.) ; KEYMEULEN, D. (Hrsg.) ; ZEBULUM, R. S. (Hrsg.): *The 2002 NASA/DoD Conference on Evolvable Hardware.* Alexandria, Virginia : IEEE Computer Society, 15-18 Juli 2002. - ISBN 0-7695-1718-8

[SGS02] STALLMAN, R. M. ; MCGRATH, R. ; SMITH, P. : *GNU Make - A Program for Directing Recompilation.* Free Software Foundation, Inc. , 2002. http://www.gnu.org/manual/make-3.80/make.html

[SP02] STALLMAN, R. M. ; PESCH, R. ; SHEBS, S. et al. : *Debugging with GDB.* 9. Free Software Foundation, Inc. , 2002. http://www.gnu.org/manual/gdb-5.1.1/gdb.html

[SSFH00] SIEVER, E. ; SPAINHOUR, S. ; FIGGINS, S. ; HEKMAN, J. P. : *LINUX in a nutshell*. 3. O'Reilly, 2000

[Sta01] STALLINGS, W. : *Operating Systems*. 4. Prentice-Hall International, Inc. , 2001

[Sta02] STALLMAN, R. M. : *Using the GNU Compiler Collection*. Updated for GCC 3.3. Free Software Foundation, Inc. , 2002. http://www.gnu.org/software/gcc/onlinedocs/

[Sta03] STALLMAN, R. M. : *GNU Compiler Collection Internals*. Updated for GCC 3.4. Free Software Foundation, Inc. , 2003. http://www.gnu.org/software/gcc/onlinedocs/

[Tan88] TANENBAUM, A. S. : *Computer Networks*. Prentice Hall International, Inc. , 1988

[TG99] TANENBAUM, A. S. ; GOODMAN, J.: *Computerarchitektur*. 4. Prentice-Hall International, Inc. , 1999

[Ti92] TISCHER, M. : *PC intern 3.0*. Data Becker, 1992

[TIS] Tool Interface Standards (TIS): *Executable and Linking Format (ELF)*. Portable Formats Specification Version 1.1. www.skyfree.org/linux/references/ELF_Format.pdf

[VM03] VMware: www.vmware.com

[Wir96] WIRTH, N. : *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996.

[WS98] WARD, B. ; SÜTTERLIN, P. : *Linux Kernel HOWTO*. Version 0.80-3, 1998. http://www.linuxhaven.de/dlhp/HOWTO/DE-Kernel-HOWTO.html

[Xil02a] Xilinx, Inc. : *Virtex-II Pro Platform FPGA User Guide*. www.xilinx.com

[Xil02b] Xilinx, Inc. : *PPC 405 User Manual*. www.xilinx.com

[Yag03] YAGHMOUR, K. : Building Embedded Linux Systems. O'Reilly, 2003. http://www.oreilly.com/catalog/belinuxsys/chapter/ch05.pdf - EmbeddedTUX.org. http://www.embeddedtux.org/index.html

Index