Lehrstuhl Organisation
und Management
von Informationssystemen
Prof. Dr. H.P. Großmann

Universität Ulm

Studienarbeit

Bootloader und embedded Linux Anpassungen für ein ARM9 basierendes Mikrocontrollerboard

Carsten Groß

Februar 2006

Prüfer: Prof. Dr. H.P. Großmann
Betreuer: Dipl.-Ing. Andreas Schmeiser

Erklärung

Die vorliegende Studienarbeit

Bootloader und embedded Linux Anpassungen
für ein ARM9 basierendes Mikrocontrollerboard

wurde von mir selbständig und ohne unzulässige fremde Hilfe angefertigt.

Ulm, den 9. Februar 2006

Carsten Groß

Inhaltsverzeichnis

1 Einleitung

Mikrocontroller im Kraftfahrzeug

Zur Modellierung von Verkehrslenkungs- und Informationssystemen in Kraftfahrzeugen wurden in der Abteilung Organisation und Management von Informationssystemen der Universität Ulm Modellfahrzeuge entworfen um eine definierte Testumgebung für typische Kommunikationssituationen zu schaffen. Auf diesen Versuchsfahrzeugen werden Platinen im Europaformat montiert die mit einem Mikrocontroller bestückt sind. Diese Mikrocontrollerplatinen sind im Rahmen einer Studienarbeit an der Universität Ulm entwickelt worden [HW]. Mit Hilfe der Mikrocontrollerplatinen, können zwischen den Modellfahrzeugen adhoc Netzwerke gebildet, untersucht und analysiert werden.

Durch die Verwendung von Modellfahrzeugen, können reproduzierbare Kommunikationssituationen mit geringem Aufwand simuliert werden. Dies wäre nicht möglich wenn man normale Kraftfahrzeuge im Straßenverkehr verwenden würde.

Überblick

Im Rahmen dieser Studienarbeit wurde ein passendes Betriebssystem für diese Mikrocontrollerplatinen zusammengestellt und ein Bootloader zum Starten des Betriebssystems programmiert. Der Bootloader initialisiert hierzu die Hardwarekomponenten des Mikrocontrollers und lädt das Betriebssystem von einem nichtflüchtigen Speicher in den Hauptspeicher und startet es. Der Bootloader verwendet etwas Code in der nativen Assemblersprache der ARM CPU Familie und ist ansonsten im überwiegenden Teil in der Programmiersprache C implementiert. Der Bootloader ist an den von [KB] verwendeten Bootloader angelehnt. Das Betriebssystem basiert auf einem Linux-Kernel und verwendet die uClibc (sprich: müc-libc) als C-Laufzeitbibliothek. Im Abschnitt 7.4 wird die Konfiguration und Übersetzung des Linux-Kernels und der uClibc näher erläutert.

Als Entwicklungssystem, welches im folgenden Text als Host-PC oder Host-System bezeichnet wird wurde ein IBM kompatibler PC mit installiertem SUSE Linux 9.3 verwendet. Auf diesem System wurden zusätzlich zur Standardinstallation ein C-Compiler und die sog. „Development“-Pakete installiert. Um die durchgeführten Beschreibungen nachzuvollziehen sind Grundkentnisse im Umgang mit Linux basierenden Systemen von Vorteil. Hierzu gibt es umfangreiche Literatur, [Kofler] gibt eine gute Einführung und eine breite, allgemeine Übersicht.

Im folgenden Kapitel 2 wird die Hardware der Mikrocontrollerplatine vorgestellt, gefolgt von einer Übersicht über den Bootvorgang des Systems, Linux und der Erstellung von Programmcode. In den darauf folgenden Kapiteln werden der im Rahmen der Studienarbeit implementierte Bootloader und Monitor vorgestellt und die verwendeten Parameter detailliert beleuchtet.

2 Aufbau der Hardware

Ein hochintegrierter Atmel AT91RM9200 bildet die zentrale Komponente der Mikrocontrollerplatine.

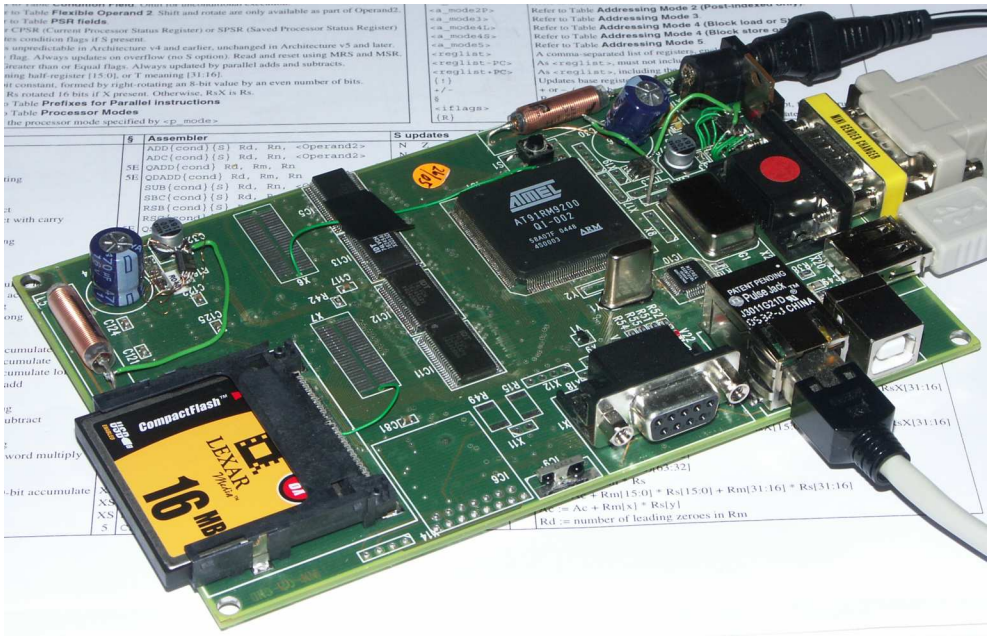

Abbildung 2.1: Ansicht der teilbestückten Mikrocontrollerplatine

Dieser Mikrocontroller enthält nicht nur eine CPU (*Central processing Unit, Zentrale Verarbeitungseinheit*) sondern auch eine MMU (*Memory Management Unit*) und zusätzlich viele periphere Komponenten. Es stehen programmierbare I/O Leitungen, serielle Schnittstellen und ein direkt integrierter Ethernetcontroller zur Verfügung. Zusätzlich befinden sich im Mikrocontrollergehäuse ein programmierbarer SDRAM- und Speichercontroller für statische Speicher wie Flash oder SRAM. Compact Flash und SD/MMC Steckplätze können ohne aufwändige Zusatzschaltungen direkt am Mikrocontroller angeschlossen werden. An Speichern stehen im Gehäuse 16 KiByte statisches RAM und 128 KiByte ROM zur Verfügung.

Bei der im Mikrocontroller integrierten CPU handelt es sich um eine ARM9 CPU. Das heißt die CPU des Atmel AT91RM9200 Mikrocontrollers versteht den Maschinencode der ARM CPU Familie und ist eine typische RISC (*Reduced instruction set computing*) CPU. Diese CPU

ist nicht Binärcodekompatibel zu den in IBM kompatiblen PCs üblichen Intel x86 CPUs. Deshalb muß bei der Programmentwicklung auf einem PC ein sog. Crosscompiler verwendet werden. Die Einrichtung und Installation eines Crosscompilers ist in Abschnitt 7 näher erläutert. Wie schon im ausgeschriebenen Namen der ARM CPU sichtbar handelt es sich um eine RISC Architektur.

Vorteile der RISC Architektur sind unter anderem die hohe Registerzahl der CPU mit der Möglichkeit flexible Befehle mit drei voneinander unabhängigen Parametern (zumeist Registern) zur Verfügung zu haben. Man nennt diese Operationen mit drei unabhängigen Parametern auch *3 Wege Operationen*. Dies ermöglicht — nicht nur bei der Programmierung in Assembler, sondern speziell bei der automatischen Codeerzeugung durch einen Compiler — recht eleganten und kompakten Programmcode.

In Abbildung 2.2 ist ein stark schematisiertes Blockschaltbild skizziert, um die beschriebenen Komponenten zu verdeutlichen.

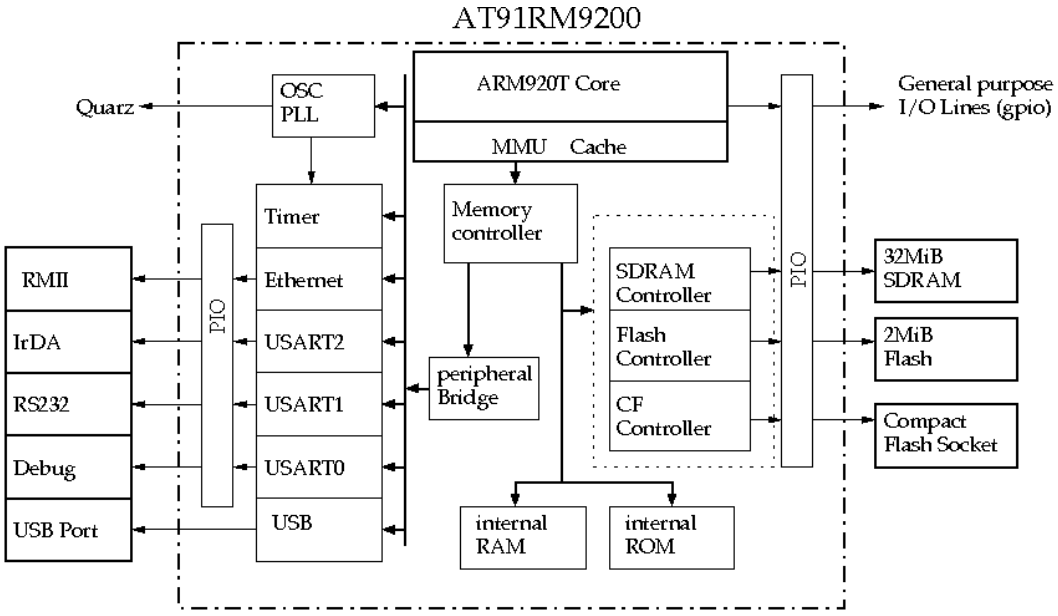

Abbildung 2.2: Blockschaltbild der Platine [HW, Atmel]

Eine Besonderheit des AT91RM9200 sind die zahlreichen Optionen, die dem Verwender des Mikrocontrollers je nach Programmierung und äußerer Beschaltung zur Verfügung stehen. Im Maximalfall stehen bis zu $4 \cdot 32$ frei programmierbare I/O Leitungen zur Verfügung. Aufgrund der sehr flexiblen Verwendbarkeit, werden diese auch als GPIO „general purpose I/O“ bezeichnet. Die programmierbaren I/O („PIO“) Controller werden PIOA bis PIOD genannt. Allerdings ist die Zahl der Anschlüsse am CPU Gehäuse begrenzt und beträgt beim hier verwendeten PQFP (*Plastic quad flat package*) Gehäuse 208 Pins. Deshalb wird bei der Verwendung der meisten Optionen die Menge der frei programmierbaren I/O Leitungen reduziert, da die für die Option nötigen Leitungen den Anschlußpin mit GPIO Leitungen

teilen. Der Anwender muß die gewünschte Verwendung eines Mikrocontrolleranschlußpins über entsprechende Kontrollregister festlegen.

Auf der Mikrocontrollerplatine sind die folgenden Komponenten implementiert worden: Als zusätzliche Speicher stehen 32 MiByte SDRAM als Hauptspeicher und 2 MiByte Flash ROM als persistenter Speicher für das Betriebssystem und Applikationen zur Verfügung. Über den Compact Flash Port oder über die USB Schnittstelle kann dem System weiterer Massenspeicher zur Verfügung gestellt werden. Die USB Schnittstelle ist für diesen Zweck nicht sehr geeignet, da es sich um eine USB 1.1 Schnittstelle mit einer maximalen Datenrate von 12 MBit/s handelt.

Als Peripherieschnittstellen sind eine drahtlose Infrarotschnittstelle nach dem IrDA Standard, eine serielle RS-232 Schnittstelle mit voller Belegung, eine weitere, die sog. Debug Schnittstelle, mit reduzierter Belegung und ein 100 MBit/s Ethernet Anschluß implementiert. Der Ethernetcontroller im Mikrocontrollergehäuse wird mit einem RMII (Reduced Media Independent Interface) Controller verbunden und darüber wird dann die Verbindung zur Außenwelt hergestellt. In Kapitel 4 auf Seite 13 wird die Verwendung der Peripheriekomponenten unter Linux beschrieben. Eine detaillierte Beschreibung der Mikrocontrollerplatine befindet sich in [HW].

2.1 Über Compact Flash Karten

Die Compact Flash Association wurde 1995 gegründet, um eine kleine universelle Steckkarte mit geringer Stromaufnahme für mobile Computeranwendungen zu definieren.

Hierbei handelt es sich um einen herstellerunabhängigen Standard der die logischen, elektrischen und mechanischen Eigenschaften dieser Steckkarten definiert. Durch die Herstellerunabhängigkeit ist eine große Interoperabilität gesichert.

Die bekannteste und wichtigste Compact Flash Kartenanwendung ist die Compact Flash Speicherkarte. Durch einen integrierten Controller kann diese Karte elektrisch und logisch wie eine IDE Festplatte angesprochen werden. Das heißt der in der Compact Flash Karte in der Regel vorhandene Flash Speicher kann gelesen, beschrieben und formatiert werden wie wenn es sich um eine Festplatte handeln würde.

Aus Sicht eines Betriebssystem kann eine Compact Flash Speicherkarte demzufolge auch wie eine Festplatte verwendet werden. Zur Zeit sind Kapazitäten von 16 bis zu 4096 MiByte gebräuchlich. Auf einer Compact Flash Speicherkarte können also problemlos umfangreiche Datenmengen gesichert werden.

Da die gesamte Programmierschnittstelle der Compact Flash Speicherkarten exakt dem IDE Standard entspricht stehen zum Teil auch unnötige oder auch veraltete Funktionalitäten zur Verfügung (so z.B. das „formatieren einer Spur“ oder ähnliche Funktionen die für einen Flash Speicher so natürlich keinen Sinn ergeben). In dieser Studienarbeit wurden alle Operationen auf der Compact Flash Karte im sog. „LBA“ Modus durchgeführt. LBA bedeutet *logical block addressing* und heißt, dass die einzelnen Sektoren einfach, von Null beginnend,

aufsteigend durchnumeriert sind. Ein Sektor ist hier ein 512 Byte großer Datenblock. Compact Flash Karten unterstützen auch noch die ältere sogenannte CHS (*Cylinder, Head, Sector*) Adressierung, die sich mit der Festplattenherkunft leicht erklären lässt, aber als Programmierschnittstelle unhandlich zu handhaben ist und hier daher nicht verwendet wird.

Mittlerweile ist dieser Standard erweitert worden. Es sind nicht mehr nur Speicherkarten möglich sondern auch Karten die andere Funktionalität bereitstellen (z.B. Soundkarten, Wireless LAN Adapter usw.).

In [CF] sind alle Eigenschaften von Compact Flash Karten ausführlich beschrieben.

2.2 Flash ROM

Auf dem Mikrocontrollerboard sind 2 MiByte Flashspeicher als nichtflüchtiger Speicher integriert. Da während der Entwicklungsphase festgestellt wurde dass dies zuwenig Speicher für den Linuxkernel, das Rootfilesystem und zusätzliche Benutzerdaten ist, wurde beschlossen in einer nächsten Version 8 MiByte Flashspeicher zu integrieren.

Bei Flashspeicher handelt es sich um Speicher, die Ihren Speicherinhalt auch ohne Betriebsspannung aufrechterhalten können. Es sind also persistente bzw. nichtflüchtige Speicher. Im Gegensatz zu ROM (*read only memory*) oder EPROM (*erasable programmable readonly memory*) sind Flash Speicher aber durch die CPU mehrfach beschreibbar.

Es gibt grundsätzlich verschiedene Arten von Flash Speichern, die sich durch die verwendete Technologie unterscheiden. Auf dem Mikrocontroller Board ist ein Flash Speicher in NOR-Technologie untergebracht. Flash Speicher in NOR-Technologie haben den großen Vorteil direkt an den Mikrocontroller angeschlossen werden zu können, da sie einen getrennten Adress– und Datenbus besitzen. Während das Lesen genauso wie aus einem normalen ROM oder RAM erfolgt muß beim Schreiben berücksichtigt werden, dass der Flashspeicher in Blöcken organisiert ist, die hier Sektoren genannt werden. Zwar können einzelne Bytes unabhängig voneinander geschrieben werden, allerdings funktioniert das nur wenn der Flashspeicher vorher „leer“ war, das heißt alle Bits auf ‘1’ gesetzt waren. Ansonsten muß vor dem Schreiben vorher der ganze zugehörige Block gelöscht werden. Alle Bits in diesem Block werden beim Löschen wieder auf ‘1’ gesetzt.

Dieser Vorgang, sowohl des Schreibens und insbesondere des Löschens ist relativ langsam. Zudem ist die Zahl der Schreibvorgänge begrenzt, laut Datenblatt des verwendeten Flashspeichers können mindestens eine Million Schreibvorgänge pro Sektor (64 KiByte Abschnitt) durchgeführt werden.

Da das Ansteuern der einzelnen Speichertypen unterschiedlich ist gibt es einen Konfigurationsblock, der über ein standardisiertes Verfahren abgefragt werden kann. Dieser CFI (*Common flash memory interface*) genannte Bereich enthält die für die Ansteuerung durch die CPU nötigen Informationen über den Flash Speicher.

3 Bootvorgang

3.1 Systemstart und Bootprogramm

Der Mikrocontroller AT91RM9200 verfügt über mehrere Möglichkeiten ein Betriebssystem zu starten [Atmel]. Hierzu befindet sich im Mikrocontrollergehäuse ein direkt integriertes ROM in dem ein Bootprogramm gespeichert ist, welches nach einem Reset der CPU ausgeführt wird. Dieses direkt vom Hersteller programmierte, integrierte ROM vereinfacht die Entwicklung eines eigenen Betriebssystemladers ganz erheblich. Das Testen und Entwickeln erfordert keine zeitaufwändigen Programmierzyklen sondern kann direkt auf einem bestückten Board durchgeführt werden.

Nach Einschalten der Stromversorgung und Freigabe der CPU von einem definierten Reset wird das Bootprogramm aus dem Mikrocontroller-ROM ausgeführt. Das Bootprogramm überprüft verschiedene, mögliche Stellen ob sich dort gültiger Programmcode befindet. Die Gültigkeit des Programmcodes wird anhand von acht Sprungvektoren überprüft, die sich am physikalischen Beginn des jeweiligen Speichers befinden müssen. Hierbei wird lediglich überprüft ob es sich um entsprechende Kommandos für unbedingte Sprünge handelt, **nicht** ob der Programmcode plausibel ist o.ä.

Zuerst wird überprüft ob sich in einem über die SPI (*Serial peripheral interface*) Schnittstelle erreichbaren Flash Speicher gültiger Code befindet. Ist dies nicht der Fall, so wird ein serielles EEPROM an der Zweidrahtschnittstelle auf gültigen Code geprüft. Da sich keine der beiden Speichertypen auf der Mikrocontrollerplatine befinden, wird dieser Test immer fehlschlagen. In der Skizzierung des prinzipiellen Vorgangs in Abbildung 3.1 ist dies daher nicht aufgeführt.

Als letzter Test dieser Abfolge wird ein normaler 8 Bit paralleler Speicher angesprochen und der Inhalt dort geprüft. Dieser Speichertyp ist in Form des Flash ROMs auf dem Mikrocontrollerboard integriert. Während der Entwicklung des Betriebssystems wurde das Betriebssystem aber ausschließlich über die Compact Flash Karte und den seriellen Boot Uploader gestartet.

Im nächsten Schritt wird der Boot Uploader ausgeführt. Dieser ist dafür zuständig Programmcode über eine der beiden seriellen Schnittstellen zu laden und in dem im Mikrocontroller integrierten RAM zu speichern und auszuführen. Hierbei werden Daten sowohl von der RS-232 Debug Schnittstelle (im folgenden auch kurz Debug Schnittstelle genannt) als auch von der USB Schnittstelle angenommen.

Die Übertragung über die serielle RS-232 Schnittstelle erfolgt mit dem X-Modem Protokoll. Das X-Modem Protokoll ist ein sehr einfaches Protokoll zur gesicherten Übertragung von

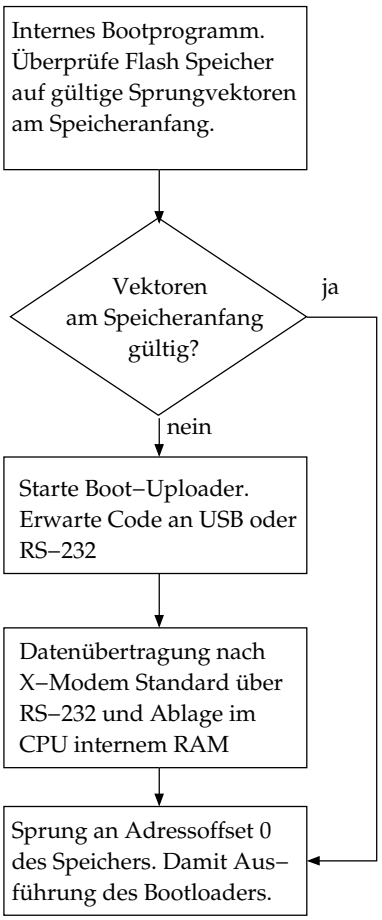

Abbildung 3.1: Vereinfachter Ablauf im CPU ROM bis zum Start des Bootloaders

Daten über diese Schnittstelle. Jeweils 124 Datenbytes werden in Blöcken, beginnend mit dem ASCII Zeichen SOH, und der Blocknummer übertragen. Den Abschluß eines Blocks bilden eine 8 Bit Prüfsumme sowie das Komplement dieser Prüfsumme. Jeder Block wird von der Gegenstelle bestätigt.

Da für die RS-232 Schnittstelle deutlich mehr Programmwerkzeuge existieren lag es nahe die Übertragung eines Bootloaders nicht über die USB Schnittstelle sondern über die Debug Schnittstelle durchzuführen. Dies hat den zusätzlichen Vorteil dass man mit einem Terminalprogramm wie minicom eine Konsolensitzung mit dem Minimalsystem aufbauen kann. Auf der Mikrocontrollerplatine ist die Debug-Schnittstelle seperat herausgeführt und kann direkt mit einer RS-232 Schnittstelle des PCs verbunden werden.

Leider kann über den Boot Uploader lediglich ein kleines Minimalsystem geladen werden welches vollständig in das interne RAM des Mikrocontrollers passt. Da vom Boot Uploader selber etwas Speicher benötigt wird, darf ein über den Boot Uploader geladenes System nur maximal 13 KiByte groß sein. Es kann sich also nur um ein Hilfsprogramm, meist einen Bootloader handeln.

3.2 Bootloader und Monitorprogramm

Um eine einfache Möglichkeit zu haben ein umfangreiches Linux Betriebssystem laden und starten zu können muß ein Bootloader erstellt werden. Der Bootloader muß die Hardware des Mikrocontrollers und der Mikrocontrollerplatine initalisieren. Daraufhin wird der Linuxkernel und ggf. ein Rootfilesystem von einem nichtflüchtigen Speicher in den Hauptspeicher geladen und dort der Linuxkernel gestartet. Der Bootloader muß hierzu die Speicheraufteilung geeignet vornehmen und dem Linuxkernel passende Parameter übergeben.

Schlägt der Start von Linux aus irgendeinem Grund fehl so kann ein sogenantes Monitorprogramm gestartet werden. Ein Monitorprogramm stellt einen interaktiven Kommandoprompt mit einfachen Kommandos zu Verfügung um eine Analyse des Systems durchzuführen. Mit einem Monitorprogramm können Speicherinhalte untersucht und verändert werden und zusätzlich auch wichtige Einstellungen zum Systemstart festgelegt werden. Außerdem ist es möglich das Betriebssystem manuell zu laden und zu starten.

4 Linux

4.1 Der Linuxkernel und das Rootfilesystem

Der Linuxkernel, der von einem nichtflüchtigen Speicher in den Hauptspeicher geladen wird, ist im weiteren Sinne eine komprimierte, selbstextrahierende Programmdatei. Dieser komprimierte Linuxkernel wird auch als `zImage` (*zipped Image*) bezeichnet. Dieses `zImage` entpackt sich beim Start selbstständig in den Speicher, selbstverständlich nur, wenn die nötigen Parameter wie in Abschnitt 8.2 beschrieben richtig übergeben wurden.

Geht man davon aus dass der Kernel richtig gestartet wird, so wird vom Linuxkernel nach der Initialisierung der unterstützten Hardware das Rootfilesystem gemounted. Die Art und der Ort des Rootfilesystems wird über Linuxkernel-Startparameter festgelegt. Der weitere Startvorgang wird dann durch die im Rootfilesystem abgelegte Konfiguration gesteuert.

Hierzu wird, nach dem Mounten des Rootfilesystems, nach der Datei `/sbin/init` bzw. `/etc/init` oder `/bin/init` in dieser Reihenfolge gesucht und bei Vorhandensein ausgeführt. Das in der Regel im Verzeichnis `/sbin/init` abgelegte `init`-Programm liest die Konfigurationsdatei `/etc/inittab` und führt die weitere Systeminitialisierung wie in dieser Datei definiert durch.

Bei einem üblichen Linux System auf einem IBM kompatiblen PC handelt es sich bei `init` um ein eigenständiges Programm. Diese Programm führt über die Datei `/etc/inittab` üblicherweise eine sehr umfangreiche und konfigurierbare Initialisierung des Linuxsystems in einem sogenannten *Runlevel* durch. Entweder in der Datei `/etc/inittab` selber oder über einen dem Kernel beim Systemstart mitgegebenen Parameter wird der Runlevel des Systems definiert. Je nach Runlevel werden unterschiedliche Skripte ausgeführt, die Konfigurationsdateien berücksichtigen und entsprechend der gewünschten Konfiguration Serverdienste starten oder diesen Dienst überspringen. Dadurch können verschiedene Systemzustände durch den *Runlevel* definiert werden. Üblich sind Konfigurationen wie z.B. Single-User ohne und mit Netzwerk oder Multiuserbetrieb.

Bei einem typischen embedded System wie dem beschriebenen Mikrocontrollerboard ist es sinnvoll einen modifizierten Ansatz des Systemstarts und der Programmwerkzeuge zu wählen.

Ein embedded System wie die Mikrocontrollerplatine unterscheidet sich in folgenden Punkten von einem üblichen Desktop PC:

- Es ist deutlich weniger Hauptspeicher vorhanden.
- Es existiert keine Festplatte sondern nur ein, im Vergleich hierzu, sehr kleines Flash ROM als nichtflüchtiger Speicher.

- Die CPU Leistung ist deutlich kleiner um den Stromverbrauch zu senken. Fließkommaoperationen werden in Software durchgeführt.

Durch die genannten Einschränkungen, ist es sinnvoll den Systemstart zu vereinfachen und den Schwerpunkt mehr in Richtung schneller Systemstart und Verwendung möglichst weniger Programmwerkzeuge zu legen.

4.2 Gerätedateien

Nachdem das vollständige Linuxsystem, also Linuxkernel und Root Filesystem, geladen bzw. gemounted sind und das System gestartet ist, ist das System bereit für Interaktion mit dem Benutzer und dem Verarbeiten von Eingabedaten.

Allgemein sind unter allen unixartigen Betriebssystemen und damit auch Linux praktisch alle I/O Zugriffe ähnlich wie Zugriffe auf normale Dateien. Die Hardware wird vom Betriebssystem virtualisiert und abstrahiert.

Damit das Linux Betriebssystem erkennt dass es sich nun um einen Hardwarezugriff handelt gibt es spezielle Dateien, sogenannte Gerätedateien über die der Hardwarezugriff abgehandelt wird.

Unter Linux gibt es zwei grundsätzlich unterschiedliche Typen von Gerätedateien, zeichenorientierte und blockorientierte Geräte. Oft wird auch in der deutschsprachigen Literatur von *character* bzw. von *block devices* gesprochen.

Das Konzept der zeichenorientierten Geräte ist recht simpel: Ein zeichenorientiertes Gerät stellt einen quasi unendlichen Datenstrom bereit bzw. kann diesen empfangen. Ereignisse sind typischerweise nicht wiederholbar. Damit kann in diesem Datenstrom nicht neu positioniert werden. Typische zeichenorientierte Geräte sind Tastatur, Computermaus, serielle Schnittstellen, Audio-Schnittstelle, Bandspeicher usw. Die meisten zeichenorientierten Geräte können nur mit Hilfe einer Applikation sinnvoll genutzt werden.

Blockorientierte Geräte bestehen im Gegensatz hierzu aus einer endlichen Menge an Datenblöcken. In diesen Datenblöcken kann positioniert werden, Lesevorgänge sind daher in der Regel wiederholbar. Positionieren heißt dass der Lesevorgang jederzeit an einer anderen, beliebigen Stelle fortgesetzt werden kann. Typische blockorientierte Geräte sind Festplatte, Diskette, CD-ROM und ähnliche Plattenspeicher. Normalerweise wird auf blockorientierten Geräten ein Dateisystem angelegt und über den `mount()` Aufruf in den Verzeichnisbaum eingehängt.

Jede zeichen– und blockorientierte Gerätedatei besteht aus dem Typ (block– oder zeichenorientiert) und einer sog. Major und Minor Number. Im wesentlichen identifiziert die Major-Number den Gerätetreiber und die Minor-Number die Gerätenummer. Wie bei normalen, regulären Dateien werden vom Linuxkernel die Zugriffsrechte der Gerätedateien beim Zugriff beachtet.

Prinzipiell könnte man einer Gerätedatei beliebige Namen geben, der Linuxkernel erkennt das angesprochene Gerät alleine anhand des Typs und der Major und Minor Number. Allerdings hat sich eine recht strenge Konvention herausgebildet und die Namen der Gerätedateien sind damit festgelegt.

Um eine Gerätedatei anzulegen existiert der Systemaufruf `mknod()` und eine gleichnamige Applikation `mknod` die die Gerätedatei anlegt. Zum Anlegen von Gerätedateien muß man priviligierter Benutzer ('root') sein.

Besonderheiten bilden das Netzwerk und die Speicherverwaltung. Unter Linux wird für die meisten Netzwerkgeräte keine gesonderte Gerätedatei benötigt. Um für ein Netzwerkinterface die IP Adresse einzustellen wird die Applikation ifconfig bzw. ip verwendet. Der Programmspeicher wird ebenfalls nicht über Gerätedateien verwaltet sondern zumeist über die C-Laufzeitbibliothek angefordert. Die C-Laufzeitbibliothek kümmert sich darum bei Bedarf den zur Verfügung stehenden Hauptspeicher über entsprechende Kernelaufrufe zu erweitern.

In Abschnitt 10 wird die Integration der Peripherieschnittstellen des Mikrocontrollerboards in das Linuxsystem vorgestellt. Es werden die Devicenodes erläutert und die Besonderheiten der Benutzung und Bedienung erläutert. Zusätzlich werden die im Rootfilesystem integrierten Applikationen vorgestellt die diese Gerätedateien verwenden.

5 Erstellen von ARM9 Programmcode

5.1 Crosscompiler

Um für die verwendete Zielplattform ARM Programmcode übersetzen zu können, muß eine sogenannte *Crosscompiler-Toolchain*, die Kombination aus den sog. binutils, dem C-Compiler und einer C-Laufzeitbibliothek für das Host-System erstellt werden. Generell wird dies als Crosscompiler bezeichnet.

In den meisten Fällen ist durch die Crosscompiler-Toolchain die Zielplattform, bzw. hier das Zielsystem „ARM Linux mit uClibc“ festgelegt. Eine Ausnahme ist die Übersetzung einer direkt ausführbaren Binärdatei die keinerlei Referenzen auf externe Bibliotheken hat. Dieser wichtige Sonderfall trifft auf den in Kapitel 8 beschriebenen Bootloader und den Linuxkernel selber zu. Der Bootloader programmiert die Hardware direkt und enthält keinerlei Bilbliotheksfunktionen.

In der Abbildung 5.1 ist der prinzipielle Ablauf einer Programmübersetzung schematisch dargestellt.

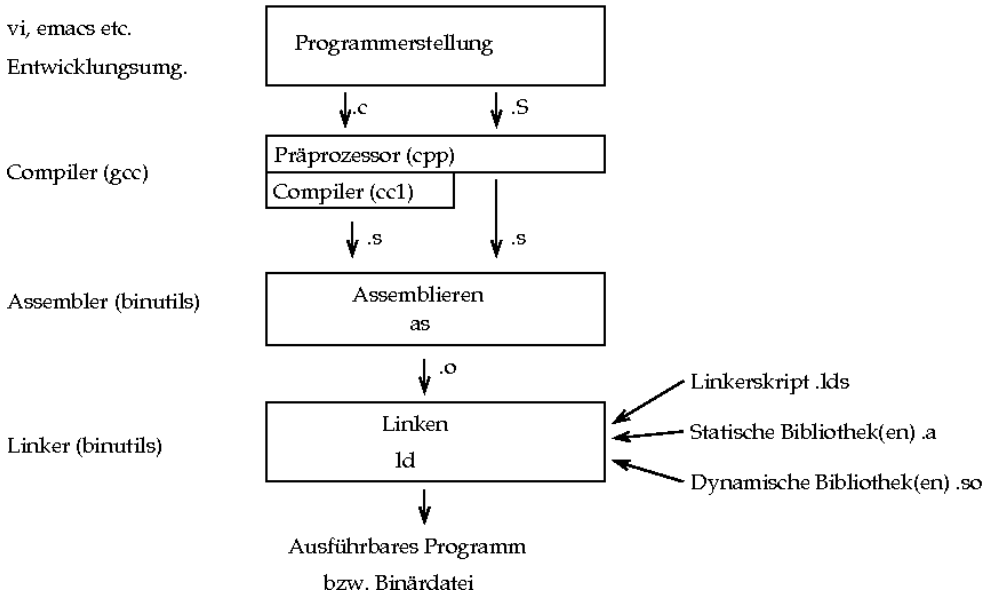

Abbildung 5.1: Ablauf einer Programmübersetzung

Während üblicherweise der Ablauf einer Programmübersetzung durch ein Frontend (entweder gcc oder eine grafische Entwicklungsumgebung) verborgen ist, läuft im Hintergrund trotzdem ein dreigeteilter Vorgang ab:

- Im ersten Schritt, dem Compilieren, wird aus einer vom Benutzer erstellten, maschinenunabhängigen Hochsprache eine Assembler Quelldatei zur Übersetzung mit demselben erstellt. Falls direkt in Assembler programmiert wurde entfällt dieser Schritt. Lediglich Dateien mit der Endung '.S' werden noch vom C Präprozessor cpp vorverarbeitet. Dies entfällt bei Assemblerdateien mit der Endung '.s'.
- Der Assembler erzeugt aus dieser Datei eine Objektdatei. Eine Objektdatei enthält schon den binären Maschinencode für die Zielmaschine. Diese Objektdatei im ELF (*Executable and linkable format*) Format enthält sowohl den Maschinencode, als auch Verwaltungsinformationen über die Position veränderlicher Variablen und externe Referenzen. Eine Objektdatei kann in dieser Form nicht direkt von der CPU ausgeführt werden, da beispielsweise die Position von Variablen im Speicher oder die Ladeadresse noch nicht festgelegt wurden. Dies muß in einem weiteren Schritt vom Linker erledigt werden.
- In der üblichen Konfiguration legt der Linker die Speicheradressen der verwendeten globalen Variablen fest und bindet Systembibliotheken dynamisch oder statisch an die Objektdatei an. Es wird eine ausführbare Datei für das Zielsystem erzeugt, normalerweise eine ELF Datei. Über ein sogenanntes Linkerskript, eine Steuerdatei für den Linker ld kann man das Ausgabeformat des Linkers aber auch verändern und z.B. die Ladeadresse, Einsprungadresse oder das Speicherlayout verändern. Dies wird bei der Übersetzung des Bootloaders verwendet, da der Bootloader ein „standalone" Programm ist und direkt vom Beginn der Datei an ausgeführt wird.

Um einen vollständigen Crosscompiler zu erzeugen muß man in 4 Schritten vorgehen:

1. Im ersten Schritt müssen die binutils erzeugt werden, die für das Erzeugen und Manipulieren von Binärdateien besonders wichtig sind. Es handelt sich hier um Assembler, Linker und verschiedene Werkzeuge, um binäre Objektdateien zu erzeugen und zu verändern. Zusammen mit dem C-Compiler werden diese Programme benötigt, um den eigentlichen Objektcode wie oben beschrieben für die Zielmaschine zu erzeugen.
2. Im nächsten Schritt muß ein vorläufiger C-Compiler erzeugt werden. Als vorläufig bezeichnet man diesen C-Compiler deswegen weil er noch nicht genügend Kenntnisse über das Zielsystem hat. Die verwendete CPU steht zwar fest, aber noch nicht die Laufzeitumgebung, also die C-Laufzeitbibliothek und das Betriebssystem. Dies sind aber wichtige Systemeigenschaften des Zielsystems, die aber erst bekannt sein können, wenn die C Laufzeitbibliothek und das Betriebssystem mit dem Compiler übersetzt wurden. Um dieses Problem aufzulösen wird lediglich ein „vorläufiger" C-Compiler übersetzt, der nur zum Übersetzen der C-Laufzeitbibliothek geeignet ist.

3. Im vorletzten Schritt wird die C-Laufzeitbibliothek übersetzt. Prinzipiell besteht die Möglichkeit verschiedene C-Laufzeitbibliotheken zu verwenden. Die Auswahl wird im Abschnitt 5.2 näher erläutert. Bei der Übersetzung der C-Laufzeitbibliothek wird auch gleichzeitig das verwendete Betriebssystem festgelegt, entweder alleine durch die Auswahl der Bibliothek, wie in unserem Fall oder durch entsprechende Optionen beim Übersetzen.

4. Im letzten Schritt wird nun der finale C-Compiler erzeugt. Zusammen mit der C-Laufzeitbibliothek und den binutils aus dem ersten Schritt, werden die zusammengehörigen Dateien und Programme in ein gemeinsames Verzeichnis gespielt damit sie problemlos zum Übersetzen weiterer Programme und Bibliotheken und des eigentlichen Linux Kernels benutzt werden können.

Um nun Programme für das ARM System zu übersetzen kann nun statt des normalen, systemweiten C-Compilers der Crosscompiler verwendet werden. In Abschnitt 7.2 ist hierzu ein Beispiel aufgeführt.

5.2 C Laufzeitbibliothek

Ebenso wichtig wie die Wahl des eigentlichen Betriebssytems für das Verhalten eines Systems ist die Auswahl der grundlegenden C Laufzeitbibliothek.

Dies ist weniger wichtig für Testprogramme bzw. den Bootloader, die direkt in den 16 KiByte internen Speicher des Mikrocontrollers geladen werden. Diese werden entweder direkt in Assembler oder als echte „stand alone“ Programme geschrieben. Sie bringen alle nötige Funktionalität mit und müssen mit Hilfe eines Linkerskriptes und des Programmes objcopy direkt in das Binärformat konvertiert werden.

Wichtig ist diese Auswahl für das später installierte System. Die C Laufzeitbibliothek legt einerseits das zu verwendende Betriebssytem fest, andererseits bestimmt sie auch die API und die ohne weiteres verfügbaren Funktionalitäten (Lokalisierung, erweiterte Funktionen).

Prinzipiell stehen verschiedene C Laufzeitbibliotheken zur Verfügung. Die naheliegenste Lösung wäre die Verwendung der glibc. Hierbei handelt es sich um die GNU C Bibliothek die sich durch Unterstützung vieler Systeme (unter anderem auch ARM) und weitestgehender Unterstützung vieler Standards, z.B. IEEE Std 1003.1 2004, auszeichnet (POSIX). In praktisch allen Linux Distributionen wird die glibc verwendet.

Es existiert noch eine weitere quelloffene C Laufzeitbibliothek. Hier soll kurz die uClibc [uClibc] vorgestellt werden. Bei dieser C Laufzeitbibliothek ist das Entwurfsziel einen möglichst geringen Speicherverbrauch zu haben. Gleichzeitig existieren aber nur geringe Einschränkungen im Funktionsumfang. Die wesentlichen Einschränkungen sind die folgenden:

- Ausschließliche Unterstützung von Linux. Andere Systeme sind nicht unterstützt.

- Verzicht auf speicheraufwenige, komplexe Funktionen - (z.B. ist die Unterstützung von Zeitzonen eher schlicht)
- Vereinfachte Namensdienste und Resolver.

Die uClibc enthält auch eine komplette Crosscompiler-Toolchain. Das primäre Designziel der uClibc ist weitestgehende Kompatibilität zur glibc bei geringem Speicherverbrauch und außerdem weitreichende Konfigurationsmöglichkeiten was benötigt wird und was nicht.

5.3 Auswahl der C Laufzeitbibliothek

Die naheliegenste Möglichkeit die GNU libc (glibc) zu verwenden ist für ein embeddeded System nicht empfehlenswert. Das ist schon aufgrund des Designzieles leicht nachvollziehbar. Der Speicherbedarf einer glibc (Version 2.3.5) auf der Festplatte beträgt inklusive der Namensdienste 1,8 MiByte auf einem i386 System.

Die Unterstützung eines anderen Betriebssystemes als Linux ist nicht erforderlich. Die Einschränkungen der uClibc stören bei einem embeddeded System nicht, daher liegt es nahe diese Laufzeitbibliothek einzusetzen.

Die uClibc (`http://www.uclibc.org/`) Laufzeitbibliothek ist nur für den Einsatz unter Linux vorgesehen und unterstützt keine weiteren Betriebssysteme. Das Einsatzziel sind explizit embedded Linux Systeme. Sie bietet einen guten Kompromiss zwischen Platzverbrauch im knappen Hauptspeicher und angebotenen Funktionalitäten die für ein embedded System benötigt werden.

Leider ist es notwendig den GNU C-Compiler zu verändern damit er in Zusammenhang mit der uClibc als System Laufzeitbibliothek verwendet werden kann. Die notwendigen Veränderungen (kurz auch „Patches“ genannt) werden im sogenannten Buildroot-Projekt `http://buildroot.uclibc.org/` schon zur Verfügung gestellt. Hier gibt es auch Makefiles, die die Erzeugung des vollständigen Crosscompilers vereinfachen. Die im vorherigen Abschnitt beschriebenen vier Schritte zur Übersetzung der binutils, der C-Laufzeitbibliothek und des Crosscompilers werden durch die Verwendung dieses Softwarepaketes zusammengefasst und durch das `Makefile` gesteuert nacheinander ohne Benutzereingriff durchgeführt. Zusätzlich wird nicht nur die Crosscompiler-Toolchain erzeugt, sondern zusätzlich noch ein Rootfilesystem mit wichtigen Programmwerkzeugen, um das Linuxsystem starten zu können.

Im Rahmen der Studienarbeit wurden in der oben genannten Buildroot Umgebung verschiedene Pakete hinzugefügt, um den aktuellen Kernel 2.6.13 zu unterstützen und um einen Webserver hinzuzufügen. Die Konfiguration der Buildrootumgebung ist im Kapitel 7 beschrieben.

6 Verbindungen

Abbildung 6.1 zeigt die zwischen der Mikrocontrollerplatine und dem Host-System während der Entwicklungsphase hergestellen Verbindungen.

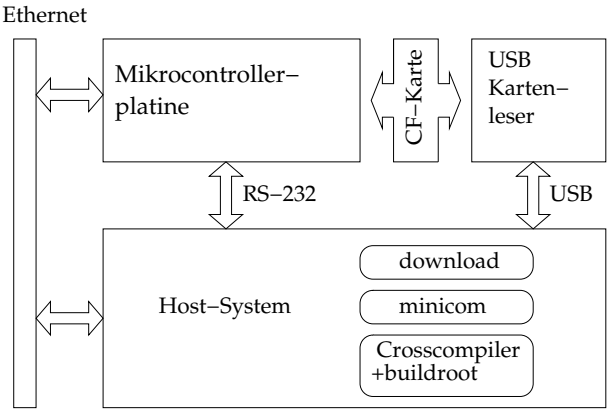

Abbildung 6.1: Wichtige Komponenten bei der Programmierung der Mikrocontrollerplatine

Die serielle RS-232 Verbindung zwischen Host-System und Mikrocontrollerplatine dient mehreren Zwecken:

- Übertragung des Bootloaders
- Kontrolle der Ausgaben des Bootloaders
- Bedienung des Monitor-Kommandoprompts
- Aufbau einer Linux-Shellsitzung wenn Linux gebootet wurde

Folgende Programmwerkzeuge werden hierzu auf dem Host-System verwendet:

- minicom - Ein universelles Terminalprogramm.
- download - Programm zur X-Modem Übertragung von Dateien. Speziell wird es zum Übertragen des Bootloaders in das RAM des Mikrocontrollers verwendet.

Das Programm `download` befindet sich auf der beigefügten CD (Anhang, Kapitel 13)

Über den am Host System angeschlossen USB Compact Flash Kartenleser können Compact Flash Karten partitioniert, beschrieben und gelesen werden. Diese können dann in den entsprechenden Steckplatz der Mikrocontrollerplatine eingesteckt werden.

Sobald das Linux Betriebssystem gestartet wird können Daten natürlich auch über die Ethernetschnittstelle mittels TCP/IP Netzwerkdiensten ausgetauscht werden.

7 Buildroot und Toolchain

7.1 Die Buildrootumgebung

Bevor Programmcode für das Mikrocontrollerboard übersetzt werden kann muß die schon im Abschnitt 5.1 vorgestellte Buildrootumgebung heruntergeladen, konfiguriert und installiert werden.

In der Buildrootumgebung sind zwei wichtige, thematisch eigentlich getrennte Bestandteile kombiniert, nämlich der Crosscompiler mit Crosscompiler-Toolchain und ein Rootfilesystem. Es lässt sich daher nicht vermeiden, dass im Text manchmal Schritte beschrieben werden die ausführlich erst in späteren Abschnitten erläutert werden. Während der Crosscompiler und die Crosscompiler-Toolchain auch benötigt werden um den in Kapitel 8 vorgestellten Bootloader mit integriertem Monitor und den Linuxkernel zu kompilieren wird das Rootfilesystem erst gebraucht wenn der Bootloader und der Linuxkernel selber läuft. Es wird also erst in einem recht späten Stadium der Entwicklungsphase benötigt. Trotzdem wird in diesem Kapitel auch auf das Rootfilesystem eingegangen, weil man schon bei der Compilation des Crosscompilers mit der uClibc Buildrootumgebung damit in Berührung kommt.

Unter der URL `http://buildroot.uclibc.org/` kann die Buildumgebung heruntergeladen werden. Damit kann man die Crosscompiler-Toolchain übersetzen. Zusätzlich kann damit auch die im Abschnitt 5.2 beschriebene uClibc übersetzt werden und ein komplettes Rootfilesystem für das Linuxsystem erstellt werden. Hierzu werden die dazu nötigen und vom Benutzer wählbaren Programmpakete über das Internet nachgeladen.

Nach dem Verfolgen des *Download* Links auf der o.g. URL zum Herunterladen der Buildrootumgebung, gibt es die Optionen entweder direkt mit Hilfe der Applikation Subversion darauf zuzugreifen, oder ein Archiv (*Daily snapshot*) herunterzuladen.

Nach dem Herunterladen eines aktuellen Archives kann dieses wie auf Linux Systemen allgemein üblich mittels des Kommandos `tar` entpackt werden. Im Top-level Verzeichnis des Buildroot Paketes findet man ein Makefile welches als Steuerdatei für das Kommando `make` verwendet wird. Damit das Programm `make` vorhanden ist, müssen der C-Compiler und die Entwicklungswerkzeuge installiert sein.

Bevor man allerdings die `buildroot` Umgebung verwenden kann muß sie zuerst noch konfiguriert werden.

7.2 Konfiguration der Buildrootumgebung

Wenn das Buildroot-Archiv an eine Stelle im Linux-Dateibaum entpackt wird an der Schreib- und Leserechte für den Benutzer vorhanden sind und ausreichend Platz existiert, kann mit der Konfiguration begonnen werden. „Ausreichend Platz“ ist variabel und auch von der späteren Auswahl der zusätzlich im Rootfilesystem zu integrierenden Pakete abhängig. Da Quelldateien und Zwischendateien abgelegt werden sollten mindestens 500 MiByte zur Verfügung stehen. Besser wäre noch mehr als ein GiByte damit man auch mit verschiedenen Paketen experimentieren kann.

Um mit der Konfiguration zu beginnen genügt es im nach dem Auspacken des Archives entstehenden `buildroot` Verzeichnis das Kommando `make menuconfig` aufzurufen.

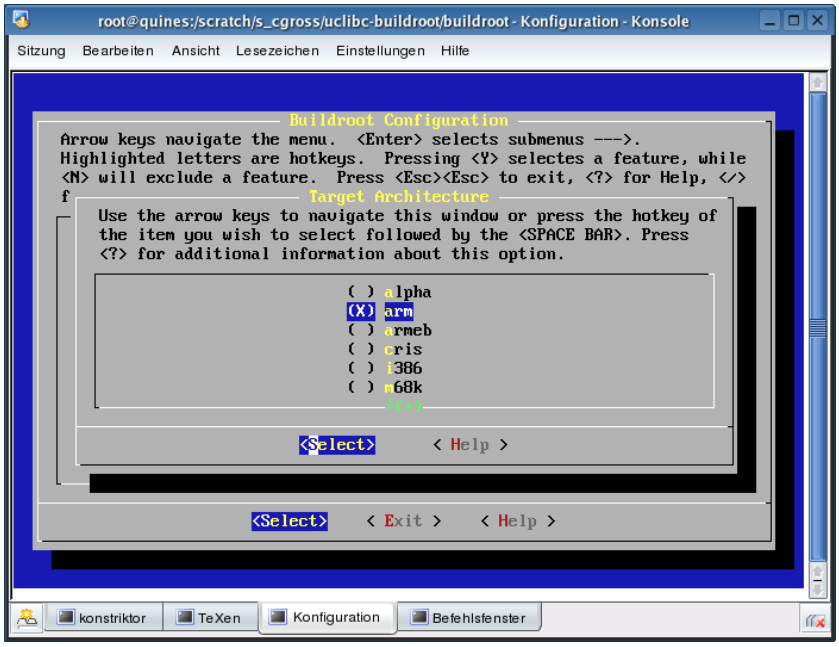

Abbildung 7.1: Menügeführte Konfiguration der Buildrootumgebung. Im Bild gezeigt ist die Auswahl der Zielarchitektur.

Die Konfiguration erfolgt, wie in Abbildung 7.1 gezeigt, menügesteuert. Nach Abschluß der Konfiguration kann die Konfiguration gespeichert werden. Sie wird in der Datei `.config` abgelegt. Es ist ratsam diese Datei separat zu sichern. Damit kann die gewählte Konfiguration jederzeit wiederhergestellt werden, auch nach einem Update der Buildrootumgebung oder versehentlichem Löschen der Konfiguration. Die Menüpunkte und verwendeten Einstellungen werden im folgenden aufgeführt.

Target Architecture

Mit diesem Menüpunkt kann die gewünschte Zielarchitektur gewählt werden. Die zu wählende Zielarchitektur ist natürlich *arm*, da Code für die ARM CPU erzeugt werden soll.

Build Options

Im Menüpunkt *Build Options* kann die Konfiguration der Toolchain und insbesondere das Installationsverzeichnis des Crosscompilers und weiterer Hilfsapplikationen festgelegt werden. Im Punkt *Toolchain and header file location?* wurde `/usr/local/arm-linux-elf` eingestellt. Hierdurch werden der Crosscompiler und die Programmwerkzeuge der Toolchain im `/usr/local` Verzeichnisbaum installiert und sind dort leicht austauschbar und stehen allen Benutzern des Systems zur Verfügung. `arm-linux-elf` symbolisiert, dass dieser Compiler Code für die ARM CPU für ein Linux System erzeugen soll. Das Dateiformat der ausführbaren Dateien ist ELF. Da auf dem Mikrocontrollerboard kein mathematischer Coprozessor zur Verfügung steht muss auch die Option *Use software floating point by default* angewählt werden. Mit dieser Option werden Fließkommaoperationen mittels der in der CPU vorhandenen Integerarithmetik berechnet.

Toolchain Options

Unter *Toolchain Options* kann man Einstellungen zur Toolchain vornehmen und die zu verwendenden Versionen der Toolchain-Programme festlegen. Unter anderem werden der voraussichtlich zu verwendende Linuxkernel festgelegt und welche Optionen der Compiler unterstützen soll. Als Kernelversion wurde *2.6.12* gewählt. Damit kann auch Kernelversion 2.6.13 oder 2.6.14 verwendet werden. Diese Auswahl beinflußt insbesondere das Systeminterface der nachher erzeugten uClibc, hiermit wird noch nicht der Kernel final ausgewählt. Für einen aktuellen Kernel der Version 2.6.13 gibt es umfangreiche Erweiterungen um die im AT91RM9200 Mikrocontroller enthaltene Peripherie zu unterstützen [SAN]. Als GNU C Compiler wird die Version *3.4.2*, die binutils werden in der Version *2.16.1* verwendet. Diese Auswahl erlaubt es den Kernel und alle Programmwerkzeuge problemlos zu übersetzen.

Package Selection for the target

Der Punkt *Package Selection for the target* bestimmt welche Programmpakete im Zielsystem zur Verfügung stehen. Selbstverständlich steht das wichtige Basiswerkzeug `busybox`, welches über 30 Standardbasis-Programme in einer Applikation vereinigt, zur Verfügung. Durch die Installation von `busybox` sind Basiswerkzeuge wie eine Shell (ash), init, Netzwerkkonfigurationsprogramme wie ifconfig, route, ifup und ifdown und weitere, zum Systemstart erforderliche Programmwerkzeuge wie mount, umount, cat, echo usw. vorhanden. Neben `busybox` sind daher nur noch wenige, weitere Programmpakete nötig.

Für das Rootfilesystem wurden noch folgende Pakete ausgewählt:

dropbear_ssh Ein SSH (*secure shell*) Server und Client. Dadurch können gesicherte Terminalsitzungen und Datenübertragungen empfangen und initiiert werden.

hotplug Ermöglicht es eingesteckte USB Geräte automatisch zu erkennen und die nötigen Linuxkernel Module zu laden.

module-init-tools Applikationen um Linuxkernel Module zu laden oder wieder zu entfernen. Dabei handelt es sich um die Version für den Linux Kernel 2.6.x.

mtd/jffs2 Utilities Programmwerkzeuge um den Flash Speicher der Mikrocontrollerplatine beschreiben und löschen zu können. Zusätzlich können auch jffs oder jffs2 (*Journaling flash filesystem*) Dateisysteme erzeugt werden. Mit einem jffs Dateisystem kann das Flash ROM der Mikrocontrollerplatine als nichtflüchtiger Datenspeicher genutzt werden.

pppd Mit dem PPPD können PPP (*point to point protocol*) Verbindungen über serielle oder Ethernetschnittstellen aufgebaut werden. Für Entwicklungszwecke besonders praktisch falls kein Ethernet zur Verfügung steht.

pcmciautils Applikation zur Unterstützung von PCMCIA bzw. Compact Flash Hardware.

Weitere Pakete können auf Wunsch ausgewählt werden.

Target Options

Unter *Target Options* kann man festlegen in welchen Formaten das Rootfilesystem abgelegt wird. Auf jeden Fall gibt es das Rootfilesystem im Dateibaum als normale Verzeichnisstruktur, zusätzlich kann aber noch ausgewählt werden, dass gleich eine Image-Datei erzeugt wird, die das Dateisystem enthält. Im Rahmen der Studienarbeit wurde direkt die `cramfs` (*Compressed RAM filesystem*) Imagedatei erzeugt, da Unterstützung für dieses Format im Bootloader implementiert ist. Eine Imagedatei mit `cramfs` ist im Vergleich zu möglichen Alternativen recht klein, da das Dateisystem komprimiert wird und nur die nötigsten Daten der Dateieigenschaften in der Inode gespeichert werden.

Board Support Options

Mit Hilfe des Punktes *Board Support Options* kann eine Vorauswahl an Paketen getroffen werden die für einen bestimmten Anwendungszweck gedacht sind. Dies wurde hier aber nicht verwendet.

Abgeschlossene Konfiguration

Die verwendete Version der Buildrootumgebung, die von der Buildrootumgebung über das Internet heruntergeladenen Programmpakete und die im Rahmen der Studienarbeit durchgeführte Konfiguration, befindet sich auch auf der beigefügten CD (siehe Anhang, Kapitel 13).

Kompilieren

Nach Abschluß dieser Konfigurationsarbeiten kann nun der Crosscompiler und das Rootfilesystem übersetzt werden indem man das Kommando `make` eingibt.
Die Übersetzung des Crosscompilers und der anderen Programmwerkzeuge beginnt, gesteuert durch das `Makefile` und die soeben vorgenommene Konfiguration, abgelegt in der Datei `.config`.
Nach einiger Zeit, wird der Compilierungsvorgang abgeschlossen. Im Verzeichnis `/usr/local/arm-linux-elf` befindet sich nun die Crosscompilierungs-Toolchain und im Top-Level Verzeichnis der buildroot Umgebung wurde eine Datei `rootfs.arm_nofpu.cramfs` angelegt. Dabei handelt es sich um eine Image Datei des Rootfilesystems die später betrachtet werden soll.

Verwendung des Crosscompilers

Der in `/usr/local/arm-linux-elf` installierte Crosscompiler kann nun, ähnlich wie der schon im System vorhandene „normale“ C-Compiler, zum Übersetzen von Software für die ARM-Linux Plattform verwendet werden.
Als Beispiel soll hier das folgende kleine „Hallo Welt“ Programm dienen:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    printf("Hallo Welt.\n");
    exit(0);
}
```

Es kann unter dem Namen „test.c“ gespeichert werden. Statt nun das Kommando

```
gcc -o test test.c
```

für die Übersetzung des Programms für den Host-PC zu verwenden, kann es mit dem Kommando

```
/usr/local/arm-linux-elf/bin/arm-linux-gcc -o test test.c
```

für das ARM Linux Board übersetzt werden.

Jetzt wird man natürlich einwenden was denn mit der C-Laufzeitbibliothek ist, die sicherlich zur Applikation test hinzugelinkt wird. Im zweiten Fall wird die uClibc als dynamische Bibliothek verwendet. Der Crosscompiler ist also für diese „Standardsituation“ schon vorbereitet.

Die Voreinstellung des Crosscompilers legt die Suchpfade der Laufzeitbibliotheken schon für das Zielsystem fest. Dort wird der zum Ablauf nötige dynamische Linker `/lib/ld-linux.so.2` und die C-Laufzeitbibliothek `/lib/libc.so.6` verwendet. Dies ist ein symbolischer Link auf die uClibc. Im Rootfilesystem finden sich diese Dateien natürlich an dieser Stelle.

Ähnliches gilt für die Verwendung eigener dynamischer Bibliotheken: Diese müssen natürlich für die ARM Architektur compiliert werden und am besten auch unterhalb von `/usr/local/arm-linux-elf` zur Verfügung stehen (Die Bibliothek selber in `lib`, die dazugehörigen Includedateien in `include`).

Handelt es sich um ein Programm ohne externe Referenzen in die C-Laufzeitbibliothek oder andere Bibliotheken wie z.B. den Bootloader so wird das Programm statisch gelinkt, enthält also keine Referenzen auf externe Bibliotheken. Allerdings ist es immer noch im ELF Format, also dem nativen Format von Linux Programmdateien. Es kann nicht direkt ausgeführt werden, da sich am Anfang der Programmdatei verschiedene Verwaltungsdaten befinden.

Um ein Programm zu compilieren das über den seriellen Bootuploader in den CPU internen RAM Speicher geladen und ausgeführt werden kann sind prinzipiell die im folgenden aufgeführten Schritte nötig. An dieser Stelle werden die entscheidenden Punkte hervorgehoben die diesen Vorgang von einer normalen Programmcompilierung unterscheiden. Im Sourcecode des Bootloaders kann die genaue Vorgehensweise an einem Beispiel nachvollzogen werden.

```
/usr/local/arm-linux-elf/bin/arm-linux-as -o arm_init.o arm_init.s
/usr/local/arm-linux-elf/bin/arm-linux-ld -e 0 -o bootloader.out arm_init.o [...]
/usr/local/arm-linux-elf/bin/arm-linux-objcopy -O binary bootloader.out bootloader.bin
```

Die erste Zeile assembliert ARM Assemblercode in der Datei `arm_init.s` und erzeugt daraus eine Objektdatei `arm_init.o`.

Die mittlere Zeile weist den Linker über die Option `-e` an, dass die Einsprungaddresse (*execution address*) auf Adresse 0 gelegt wird, da nach der Übertragung des Bootloaders in den Speicher des Mikrocontrollers die Ausführung an der Adresse 0 beginnt (siehe Abbildung 3.1 auf Seite 10). Wichtig ist in diesem Fall, dass das erste angegebene Objektfile den Startupcode enthält (Im Falle des Bootloaders also die in Assembler geschriebenenen Teile). Die Punkte symbolisieren hier, dass durchaus noch mehrere Objektdateien dazugelinkt werden können.

Mit der dritten Zeile wird aus der Datei im ELF Format (`bootloader.out`), die vom Linker erzeugt wurde, mit Hilfe der Applikation objcopy der durch die CPU ausführbare Binärcode herauskopiert und in der Datei `bootloader.bin` gespeichert. Damit befindet sich der ausführbare Programmcode direkt ab dem ersten Byte in der Ausgabedatei.

Im nächsten Kapitel 8 wird der Bootloader beschrieben der mit Hilfe des in diesem Kapitel installierten Crosscompilers übersetzt werden kann.

7.3 Kompilation des Rootfilesystems

Neben dem Crosscompiler wird von der Buildrootumgebung auch noch das Rootfilesystem für die spätere Verwendung übersetzt. Es ist sinnvoll erst nach der Übersetzung des Crosscompilers eine „Feinkonfiguration“ des Rootfilesystems durchzuführen.

Konfigurierbar sind alle Programmpakete. Eine Konfiguration wurde bei den folgenden Paketen durchgeführt:

Das Programm busybox

Der schon oben angesprochene Paketumfang

Die Pakete `busybox` als auch die C-Laufzeitbibliothek `uClibc` können nach dem gleichen Schema konfiguriert werden wie die Buildrootumgebung selber. Es genügt also in das jeweilige Verzeichnis zu wechseln (`build_arm_nofpu/...`) und dort `make menuconfig` aufzurufen um die Konfiguration der Pakete zu starten. Während der menügeführten Konfiguration der Pakete ist es möglich unnötige Funktionalitäten wegzulassen bzw. benötigte Funktionen zu aktivieren. Im Falle der Applikation `busybox` kann man z.B. Unterstützung der Konfiguration der VGA Console entfernen, da das Mikrocontrollerboard keine VGA kompatible Konsole besitzt.

Danach kann dann das Rootfilesystem oder nur das jeweilige Paket neu compiliert werden.

Die Dateien werden dann in das Verzeichnis `build_arm_nofpu/root` gespeichert. Dies ist ein Abbild des Rootfilesystems welches, je nach Konfiguration der Buildrootumgebung, in eine oder mehrere Imagedateien gesichert wird. Wenn — wie hier — konfiguriert wurde, dass ein cramfs Image erzeugt werden soll, so wird eine Datei `rootfs.arm_nofpu.cramfs` im `buildroot` Verzeichnis erzeugt die mit dem Kommando

```
mount -t cramfs -o loop rootfs.arm_nofpu.cramfs /mnt
```

als Benutzer ‘root’ auch auf dem PC in den Dateibaum eingehängt werden kann. Unter `/mnt` kann man das root Filesystem untersuchen, aber *nicht* verändern, da das Dateisystem cramfs ein read-only Dateisystem ist.

Diese Datei kann direkt in die Partition auf der Compact Flash Speicherkarte geschrieben werden die für das Linux Dateisystem angelegt wurde.

Verändern des cramfs

Um das Rootfilesystem zu verändern, können in `build_arm_nofpu/root` entsprechende Änderungen durchgeführt werden. Mit Hilfe des Programms `mkcramfs` welches ebenfalls in der Buildrootumgebung für den Host-PC kompiliert wurde kann eine neue Image Datei wie folgt erzeugt werden. Zuvor muß in das Verzeichnis `build_arm_nofpu` in der Buildrootumgebung gewechselt werden:

```
./cramfs-1.1/mkcramfs root/ ../rootfs.arm_nofpu.cramfs
```

7.4 Konfiguration und Compilation des Linux Kernels

Ähnlich wie das Übersetzen der Buildrootumgebung erfolgt die Konfiguration des Linuxkernels menügesteuert. Im Gegensatz zur Konfiguration der Buildrootumgebung muss der Linuxkernel mit Hilfe eines externen Patches verändert werden. Dieser Patch bringt die Unterstützung der im Mikrocontroller integrierten Peripheriehardware.

Der Linuxkernel selber kann von zahlreichen Web- und FTP-Servern heruntergeladen werden, z.B. von `http://www.kernel.org/`. Im Rahmen der Studienarbeit wurde der Kernel Version 2.6.13 verwendet.

Die AT91RM9200 Patches stehen unter der URL `http://maxim.org.za/AT91RM9200/2.6/` zur Verfügung. Um den Patch für den Kernel 2.6.13 zu verwenden muss die Datei `http://maxim.org.za/AT91RM9200/2.6/2.6.13-at91.patch.gz` heruntergeladen werden. Diese Datei kann, wie üblich, mit Hilfe des Programmes `gzip` entpackt werden.

Nach dem Entpacken des Kernels an eine Stelle mit ausreichend Speicherplatz, das heißt mindestens 150 MiByte, kann der Patch angewendet werden. Hierzu muß in das Top-Level Verzeichnis des soeben entpackten Kernels gewechselt werden und das Kommando

```
patch -p1 < 2.6.13-at91.patch
```

ausgeführt werden. Ggf. muß natürlich ein Pfad zu dem Patch ergänzt werden, damit das Programm die Datei findet. Durch das Programm `patch` wird nun der originale Linuxkernel verändert.

Damit nun eine Cross-Configuration und -Compilation des Kernels durchgeführt werden müssen zwei Umgebungsvariablen gesetzt werden. Wie man sich leicht vorstellen kann, muss dem Konfigurationswerkzeug mitgeteilt werden für welche Architektur übersetzt werden soll und welcher Cross-Compiler hierzu verwendet werden soll.

Zur Compilation des ARM Linuxkernels auf dem verwendeten Host-System müssen folgende Umgebungsvariablen gesetzt werden:

```
ARCH=arm
CROSS_COMPILE=/usr/local/arm-linux-elf/bin/arm-linux-
```

Über die Umgebungsvariable `ARCH` wird die Zielarchitektur festgelegt, über die Umgebungsvariable `CROSS_COMPILE` der Präfix der Crosscompiler-Toolchain. Nachdem man diese beiden Umgebungsvariablen mit Hilfe des Kommandos `export` in die Umgebung der aktuellen Shell exportiert hat, kann die menügeführte Konfiguration der Linuxkernelcompilation mittels `make menuconfig` begonnen werden.

Auch diese Konfiguration wurde im Rahmen der Studienarbeit durchgeführt. Hierbei wurde bei den ersten Tests der sich aus dem Standardkernel und dem Patch ergebende Linuxkernel verwendet. Später wurde der Code wie in Abschnitt 10.6 beschrieben angepasst um die Hardwarekonfiguration des Mikrocontrollerboards und die Ethernetschnittstelle zu unterstützen.

Nachdem die Konfiguration in der Datei `.config` gesichert wurde, kann die Kompilation mittels `make` gestartet werden. Der Linuxkernel selber befindet sich nach der Compilation in `arch/arm/boot/zImage` und kann direkt in die in Abschnitt 8.3 beschriebene Partition geschrieben werden. Neben dem Linuxkernel werden, zumindest in der in der Studienarbeit verwendeten Konfiguration, auch Linuxkernel-Module compiliert.

Linuxkernel-Module stellen während der Laufzeit des Linuxkernels ladbare Funktionalitäten bereit die zumeist Unterstützung für zusätzliche, hotplugfähige Hardware oder z.B. Kommunikationsprotokolle bereitstellen. Im Falle der USB Unterstützung können nach dem Einstecken eines USB Gerätes die nötigen Gerätetreiber bei Bedarf nachgeladen werden.

Mit den folgenden Kommandos können die Linuxkernel-Module direkt in das Buildroot-`root` Verzeichnis kopiert werden.

Zuerst müssen die Module an die richtige Stelle in das Rootfilesystem kopiert werden. Hierzu ist ein entsprechendes Ziel im `Makefile` definiert:

```
make MODLIB=build_arm_nofpu/root/lib/modules/2.6.13-at91rm9200/ modules_install
```

Damit die Module automatisch geladen werden können müssen die Symbole der Kernelmodule untersucht und evtl. Abhängigkeiten festgestellt werden. Dies erfolgt mit Hilfe des Programmes `depmod`

```
/sbin/depmod \
-v 2.6.13-at91rm9200 \
-b build_arm_nofpu/root/ \
-F System.map
```

Die genannten Optionen bewirken, dass nicht die im Host-System installierten Linuxkernel-Module untersucht werden, sondern die frisch in der Buildroot-root Umgebung installierten Module. Dies wird über die Option `-b` (für engl. *base*) festgelegt. Zusätzlich wird noch die Linux-Kernelversion und der Dateiname der Symboltabelle des zu den Modulen gehörigen Linuxkernels angegeben.

Da die Linuxkernel-Module mit Debug-Symbolen übersetzt werden sind die unter `build_arm_nofpu/root/lib/modules/2.6.13-at91rm9200` installierten Module sehr groß (eine übliche Konfiguration würde z.B. insgesamt 30 MiByte einnehmen). Deshalb ist es nötig

diese Debugsymbole zu entfernen. Dies kann man mit Hilfe des Programmes `strip` erledigen.

```
cd build_arm_nofpu/root/lib/modules/2.6.13-at91rm9200/kernel
for i in $(find . -type f);
do /usr/local/arm-linux-elf/bin/arm-linux-strip --strip-unneeded $i;
done
```

Nach diesem Schritt kann man das das cramfs wie in Abschnitt 7.3 beschrieben neu erzeugen. Nun können auch die mitcompilierten, ladbaren Kernelmodule verwendet werden.

Abgeschlossene Konfiguration

Der verwendete Kernel, die nötigen, im Rahmen der Studienarbeit angefertigten, Patches und die Kernelkonfiguration befinden sich auch auf der Begleit-CD (Anhang, Kapitel 13)

8 Systeminitialisierung und Bootloader

8.1 Initialisierung der Hardwarekomponenten

In einem Bootloader muß zuerst der Mikrocontroller und die dort integrierten Peripheriekomponenten initalisiert werden. Das heißt die Peripheriekontrollregister des Mikrocontrollers müssen vom Bootloader so programmiert werden, dass die auf dem Board installierte Hardware verwendet werden kann. In der Abbildung 8.1 ist der Ablauf der Hardwareinitialisierung abbgebildet. Im Text dieses Abschnitts werden die aufgeführten Punkte erläutert.

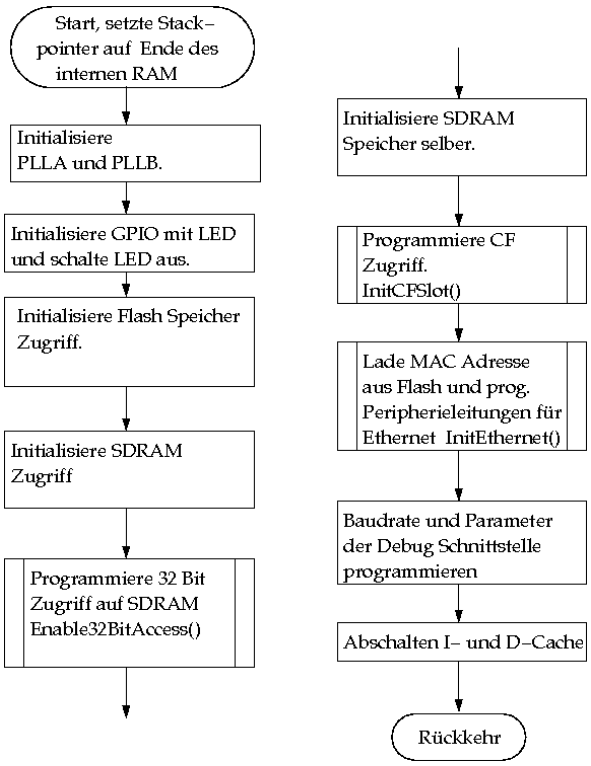

Abbildung 8.1: Flußdiagramm der Hardwareinitialisierung des Boards. Die einzelnen Punkte sind im Text näher erläutert.

Mit Hilfe der Datenblätter der Hardwarekomponenten und den Designvorgaben des Boards wurden die Parameter bestimmt, die in den einzelnen Schritten in die Peripheriekontrollregister der CPU programmiert werden müssen.

Nach der Initalisierung des Stackpointer-Registers der CPU auf das Ende des CPU internen RAMs kann die erste Subroutine aufrufen werden. Die initialisiert im ersten Schritt zwei im Mikrocontroller integrierte PLLs. Als Basisfrequenz der PLLs wird eine von einem Quarzoszillator mit Hilfe eines externen Quarzes erzeugte Frequenz von 18,432 MHz verwendet. Die PLLs werden mit PLLA bzw. PLLB bezeichnet. Es steht jeweils ein Divisor d und Multiplikator m zur Verfügung. Über Sie können die durch die PLL erzeugte Frequenz aus der Basisfrequenz durch ganzzahlige Multiplikation und Divison festgelegt werden. Hierbei gilt folgender Zusammenhang für die PLL Frequenz bei beiden PLLs.

$$f_{PLL} = \begin{cases} 0 & \text{für } m = 0 \text{ (Stromsparmodus)} \vee d = 0 \\ \frac{(m+1)f_{OSZ}}{d} & \forall\, m \in \mathbb{N}\, m \leq 255; d \in \mathbb{N}\, d \leq 255 \end{cases}$$

$$f_{OSZ} = 18,432 \text{ MHz};\ 80 \text{ MHz} < f_{PLL} < 240 \text{ MHz}$$

In der Bootloaderapplikation und später im Linuxkernel wird PLLA als Basis für die Zeitgeber („Timer“), CPU Takt und Peripherietakt verwendet. Die PLLA wird auf die Erzeugung von $\approx$ 180 MHz programmiert. Diese Frequenz wird aus der Oszillatorfrequenz von 18,432 MHz abgeleitet. Da die externen Komponenten nicht mit einer so hohen Taktfrequenz angesprochen werden können wird für die Erzeugung des Peripherietaktes, der sogenannten Masterclock ein Teilerfaktor von 3 programmiert, womit sich eine externe Masterclock von $\approx$ 60 MHz ergibt. Die Masterclock ist der Takt der als Basis für den Zugriff auf externe Komponenten dient. Die PLLB wird auf die Erzeugung von $\approx$ 96 MHz programmiert, da dieser Takt für die integrierte USB Schnittstelle benötigt wird. Genaugenommen werden für die USB Schnittstelle $48 \pm 0,5\%$ MHz Basistakt benötigt. Dazu wird aber noch ein weiterer im Mikrocontroller zu diesem Zweck vorhandener Takthalbierer aktiviert. In Tabelle 8.1 sind die im Bootloader verwendeten Faktoren aufgelistet.

	f_{OSZ}	m	d	f_{PLL}
PLLA	18,432 MHz	174	18	179,2 MHz
PLLB	18,432 MHz	72	14	96,11 MHz

Tabelle 8.1: Divisoren und Multiplikatoren der PLLs

Auf dem Board sind fünf Kontroll-LEDs angebracht die mit GPIO Schnittstellen des Mikrocontrollers verbunden sind. Im nächsten Schritt werden diese fünf GPIO Schnittstellen als Ausgang programmiert und die LEDs abgeschaltet. Im Anhang in Tabelle 12.1 sind diese Ausgänge aufgeführt.

Die nächste Peripheriekomponente die im Bootloader initialisiert wird, ist der SMC (*Static Memory Controller*). Dadurch kann dann später auf den Flash Speicher des Mikrocontrollerboards zugegriffen werden.

Auf dem Board wird ein AM29LV160DT-90EC 2MiByte Flash Speicher verwendet [HW]. Die Busbreite des Datenbusses beträgt 16 Bit. Damit es beim Zugriff auf den Flash Speicher zu keinen Fehlern kommt muss das Timing des Speichers eingehalten werden. Der verwendete

t_{DF}	*Data float time* – Benötigte Zeitdauer nachdem die low aktive CE (Chip Enable) Leitung auf high gelegt wurde und bis vom Flash Speicher der Datenbus tatsächlich wieder für andere Zugriffe freigegeben wurde. Ist in der Dokumentation des Flash Speichers genauso bezeichnet.
t_{WS}	*Wait states* – Zeitdauer zwischen dem Beginn eines Zugriffs (Zugriffsadresse wird vom Mikrocontroller an A[0..24] angelegt und CE auf low gesetzt) und dem endgültigen Bereitstellen der Daten durch den Flash Speicher. Im Datenblatt des Flash Speichers wird dies mit t_{RC} bezeichnet, die eigentlich wichtigen Zeiten sind allerdings t_{CE} (*Chip enable to output delay*) und t_{OE} (*Output enable to output delay*) die die Verzögerung zwischen Aktivierung des Bauteils durch den Mikrocontroller bis zur Bereitschaft der Daten anzeigen. Entscheidend ist also die Aktivierung des Flash Speichers durch das Setzen der CE Leitung auf low Pegel. Daher muß $t_{RWSetup}$ auf 0 gesetzt werden und die Adressen mindestens für die größere der beiden Zeiten t_{CE} und t_{OE} gültig anliegen, siehe Tabelle 8.3 und Abbildung 8.2.
$t_{RWSetup}$	*Read/Write Setup* – Verzögerung zwischen Anlegen der gültigen Addressen (Adressen liegen an und CE ist auf low) des Mikrocontrollers bis zum Aktivieren der Schreib- (WE) oder Leseleitung (RD/OE). Diese extra Zeit wird für den verwendeten Flash Speicher nicht benötigt und wird im Datenblatt des Speichers nicht gesondert bezeichnet.
t_{RWHold}	*Read/Write Hold* – Zeitdauer in der die Adresse nach dem Zurücknehmen von Schreib- (WE) oder Leseleitung (RD/OE) noch gültig anliegt. Im Datenblatt des Flash Speichers für Lesezugriffe nicht gesondert aufgeführt, für Schreibzugriffe mit t_{DH} (*Data hold time*) bezeichnet.

Tabelle 8.2: Bedeutung der verwendeten Kurzbezeichnungen des SMC (Static memory controller) des Mikrocontrollers und deren Entsprechung beim Flash Speicher.

Flash Speicher benötigt ca. 90ns um angeforderte Daten bereitzustellen. Erfolgt das Einlesen des Datenbusses durch die CPU zu früh, so werden u.U. ungültige Daten vom Datenbus übernommen. Deshalb müssen die Timingparameter in den Registern des im Mikrocontroller integrierten SMC programmiert werden. Die Schnittstelle hierzu bildet ein Register welches die Timingparameter festlegt: `SMC_CSR0`. In Tabelle 8.2 ist die Bedeutung der Parameter aufgeführt.

In der Abbildung 8.2 wird gezeigt wie ein typischer Lesezugriff auf den Flash Speicher abläuft. Nachdem die Adressdaten gültig am Adressbus anliegen, wird die CE (Chip Enable) Leitung auf logischen low Pegel gelegt. Da es sich um einen Lesezugriff handelt wird auch die OE (Output Enable) Leitung auf low gelegt. Wie man der Tabelle 8.3 entnehmen kann benötigt der verwendete Flash Speicher nun maximal 90 ns bis die Daten am Datenbus bereitstehen und vom Mikrocontroller eingelesen werden können [AMDflash]. Deshalb muß der Parameter t_{WS} so festgelegt werden, daß die Daten erst 90 ns nachdem OE und CE auf low gelegt wurden eingelesen werden. Das Einlesen der Daten durch die CPU erfolgt grundsätzlich mit der steigenden Taktflanke des Signals Master clock (MCK).

Quelle	f_{Master}	$\frac{1}{f_{\text{Master}}}$	t_{DF}	t_{CE}	t_{OE}	t_{WS}	t_{RWSetup}	t_{RWHold}
Flash	-	-	30 ns	90 ns	35 ns	-	0 ns	0 ns
μC	60 MHz	16,7 ns	2 Clk	-	-	6 Clk	0 Clk	0 Clk

Tabelle 8.3: Verwendetes Timing des SMC, Flash steht für die jeweiligen Höchst- bzw. Mindestwerte aus dem Datenblatt [AMDflash] und μC für die im SMC des Mikrocontrollers programmierten Werte. Als statischer Speicher benötigt der Flash Speicher keinen externen Takt. t_{CE}, t_{OE} und t_{WS} sind in Tabelle 8.2 näher beschrieben.

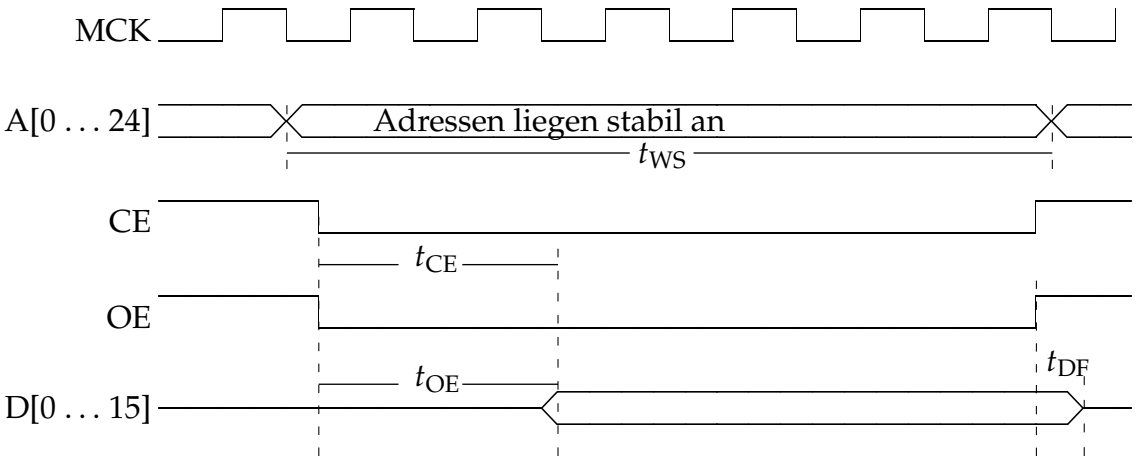

Abbildung 8.2: Prinzipieller Ablauf eines Lesezugriffes auf den Flash Speicher. Die Leitung Master Clock (MCK) ist nicht verdrahtet, ist aber die Taktbasis der Mikrocontrollerzugriffe auf den Bus, deshalb hier gezeigt. Im gezeigten Beispiel ist t_{WS} für 5 Wartezyklen programmiert.

Wie in Abbildung 8.1 auf Seite 33 gezeigt wird danach der integrierte SDRAM Controller programmiert um auf den SDRAM Speicher des Boards zugreifen zu können. Bei der Programmierung werden Organisation, Größe und Timings der extern an den Controller angeschlossenen SDRAM ICs festgelegt. Wie schon in der Einführung erläutert teilen sich einige Funktionen die Anschlüsse am Mikrocontrollergehäuse. Damit man mit 32 Bit Busbreite auf den Hauptspeicher zugreifen kann muß auf die GPIO Bits 16 bis 31 des PIOC Controllers verzichtet werden. Der Zugriff mit 32 Bit statt 16 Bit Busbreite hat den Vorteil die doppelte Datenmenge pro Zugriff übertragen zu können und damit doppelt so schnell zu sein.

Im Gegensatz zu einem statischen Speicher wie dem oben beschrieben Flash Speicher ist das SDRAM wie andere dynamische Speicher in Spalten und Zeilen (englisch *rows* und *columns*) organisiert. Der im Mikrocontroller enthaltene SDRAM Controller kann die für den Zugriff auf das SDRAM nötigen RAS (*row address select*) und CAS (*column address select*) Signale erzeugen und legt bei einem Speicherzugriff an den Adressleitungen A0 bis A12 die entspre-

chende Zeilen- und Spaltenaddresse an wie es in Abbildung 8.3 gezeigt ist. Die Aufteilung der Spalten und Zeilenadressen und das Speichertiming sind durch Programmierung des in der Mikrocontrollerdokumentation als `SDRAMC_CR` bezeichneten Kontrollregisters möglich. Die programmierbaren Timingwerte haben die in Tabelle 8.4 aufgeführte Bedeutung.

- t_{WR} *Write recovery* – Zeitdauer zwischen dem letzten Schreibzugriff und einem erneuten „Precharge“ also dem Laden einer neuen Zeilen- und Spaltenaddresse (die einen Lese- oder Schreibzugriff vorbereiten kann).
- t_{RC} *Row cycle delay* – Zeitdauer zwischen einem Refresh Kommando an das SDRAM und einem erneuten Lesezugriff.
- t_{RP} *Precharge delay* – Die Zeitdauer *Precharge delay* wird immer eingefügt wenn das Lesen einer neue Zeile aus dem SDRAM vorbereitet wird.
- t_{RCD} *Row to column delay* – Zeitdauer zwischen Einlesen der Zeilenaddresse und Einlesen der Spaltenaddresse durch das SDRAM.
- t_{RAS} *Active to precharge delay* – Minimale Zeitdauer zwischen einem Zugriff und dem erneuten Laden einer Zeilen- und Spaltenaddresse.
- t_{TXSR} *Exit self refresh to active* – Zeitdauer zwischen Beendigung des Selbst-Refresh Modus des SDRAMs und erstem Lesezugriff.

Tabelle 8.4: Bedeutung der Kurzbezeichnungen bei den Timingangaben des SDRAM

Beide angeschlossenen SDRAM Speicher (Typ:MT48LC8M16A2-75) sind in 4096 ($= 2^{12}$) Zeilen à 512 ($= 2^9$) Spalten mit je 16 Bit pro Zelle organisiert und besitzen insgesamt 4 Bänke. Jeder Speicherbaustein besitzt damit $4096 \cdot 512 \cdot 4 \cdot 16\,\text{bit} = 16\,\text{MiByte}$ Speicher. Da jeder Speicherbaustein nur je 16 Datenleitungen hat, ist der erste der Speicher mit den Leitungen D[0...15], der andere an die Datenleitungen D[16...31] des Mikrocontrollers angeschlossen. Zusammen ergeben sich damit 32 MiByte.

Bei der Programmierung sowohl des SDRAM als auch des Static Memory Controllers des Mikrocontrollers muß darauf geachtet werden, dass im Datenblatt des Speichers minimal einzuhaltende Zeiten fast ausschließlich in Nanosekunden (ns) angegeben werden. Im Mikrocontroller können aber nur ganzzahlige Vielfache des externen Taktes (der Masterclock) programmiert werden. In der Tabelle 8.5 ist dies durch das Kürzel Clk verdeutlicht. Es ist dabei immer darauf zu achten, dass sich dadurch immer eine etwas größere Verzögerung als minimal zulässig ergibt, da die Werte zwar über- aber nicht unterschritten werden dürfen.

Quelle	$f_{\text{Master}} = \text{SDCK}$	$\frac{1}{f_{\text{Master}}}$	t_{WR}	t_{RC}	t_{RP}	t_{RCD}	t_{RAS}	t_{TXSR}
SDRAM	≤ 133 MHz	7,5 ns	20 ns	66 ns	20 ns	20 ns	44 ns	75 ns
μC	60 MHz	16,7 ns	2 Clk	4 Clk	2 Clk	2 Clk	3 Clk	5 Clk

Tabelle 8.5: Verwendetes Timing des SDRAM Controllers, SDRAM steht für die jeweiligen Höchst- bzw. Mindestwerte aus dem Datenblatt [micron] und μC für die im SDRAM Controller des Mikrocontrollers programmierten Werte

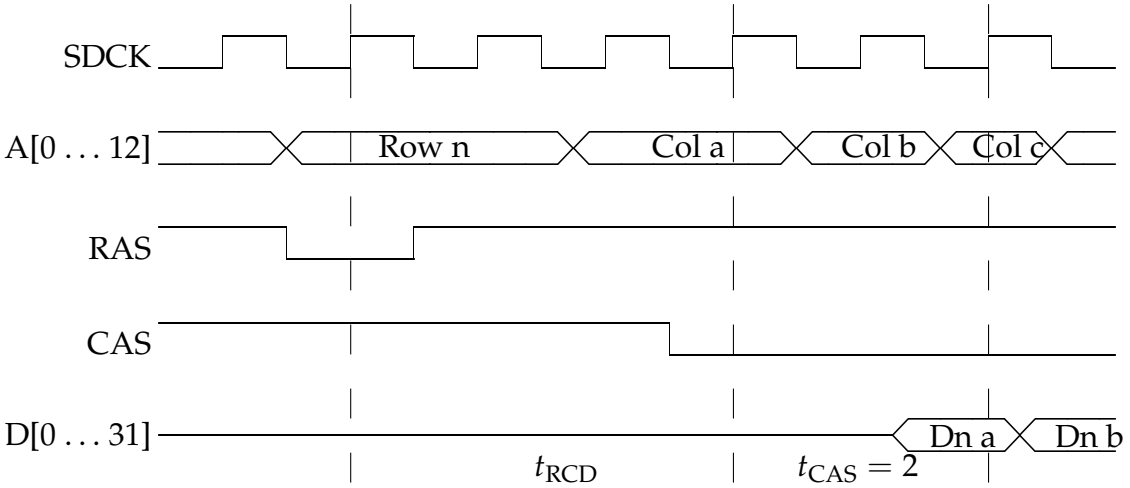

Abbildung 8.3: Ablauf eines einfachen SDRAM Lesezugriffes mit aufeinanderfolgendem Anlegen von Zeilenaddresse n und Spaltenaddresse a. Bei Zugriffen auf aufeinanderfolgende Speicherstellen wie hier dargestellt muss lediglich die neue Spaltenaddresse angelegt werden. Erst ein Zeilenwechsel macht ein erneutes Precharge nötig.

Anders als der Flash Speicher muss nun auch der SDRAM Speicher selber initialisiert werden. Der Speicher wird mit Hilfe dieses Vorgangs zurückgesetzt und die sog. „CAS-Latency“ (*column address select*, siehe Abbildung 8.3) wird festgelegt. Da der Mikrocontroller nur eine „CAS-Latency“ von zwei unterstützt muss das SDRAM entsprechend programmiert werden. Dieser Vorgang ist in [Atmel] genau beschrieben. Wichtig ist es diese Programmroutine **ohne** C-Compiler Optimierung zu übersetzen bzw. das Schlüsselword `volatile` für diese Zugriffe zu verwenden, da keine Umordnung der Zugriffe und kein „Überspringen unnötiger Operationen“ stattfinden darf. Während der Initialisierung des SDRAM wird z.B. 8 mal direkt hintereinander auf die Basisaddresse des Speichers zugegriffen. Typischerweise würde ein optimierender C-Compiler dies durch einen einzigen Zugriff ersetzen, was in diesem Fall aber nicht passieren darf.

Um den Compact Flash Steckplatz und die Ethernetschnittstelle nutzen zu können, müssen ähnlich wie bei der Initalisierung des SDRAM Controllers auch hier einige GPIO Leitungen zugunsten der Nutzung als periphere Steuerleitung umgewandelt werden.

Im Bootloader wird die Ethernet-Schnittstelle zwar nicht verwendet, aber sie wird für das später laufende Linux soweit vorbereitet, dass die 48 Bit MAC-Adresse (*Media access control*) in den Ethernetcontroller des Mikrocontrollers gespeichert wird. Hierzu wird im onboard Flash (Der ja an dieser Stelle schon zum Zugriff initialisiert ist) in den letzten 64 KiByte des Speichers nach der Zeichenkette `lan_hwaddr=` gesucht. Der Hintergrund dieser Vorgehensweise ist im Abschnitt 10.4 erläutert. Die für das Mikrocontrollerboard zu verwendende MAC-Adresse ist direkt danach als ASCII-Zeichenkette in der Form 00:10:20:30:40:50:60 gespeichert. Dies ist die Darstellung die auch von vielen Programmen verwendet wird und leicht lesbar und verständlich ist. Die Zeichenkette wird durch ein Zeilenschaltungszeichen

(ASCII-Code 0x0a) beendet.

Die Initialisierung der Debug-Schnittstelle folgt dem gleichen Prinzip wie schon die Initalisierung der beiden genannten Komponenten: Die Peripheriefunktionalität der entsprechenden GPIO Leitungen wird entsprechend konfiguriert, danach werden die Baudrate 115200 Baud und die Übertragungsparameter (8N1 - 8 Datenbits, keine Paritätsprüfung, 1 Stoppbit) durch Programmierung festgelegt.

In Tabelle 12.1 im Anhang sind die für Peripheriefunktionalität verwendeten I/O Leitungen nochmals aufgeführt.

Nach der Rückkehr aus dieser Subroutine wird der Stackpointer der CPU auf das Ende des SDRAM Speichers gelegt um Platz auch für komplexere C-Programmaufrufe zu haben. Für den Bootloader wurden 1 MiByte gemeinsamer Heap und Stack am Ende des physikalischen Speichers vorgesehen. Dies reicht für diese Anwendung auf jeden Fall sicher aus. Es ist keine Fehlerbehandlung für den Fall eines Speicherüberlaufs vorgesehen.

Konform zur Spezifikation in [Linux] `Documentation/arm/Booting` wird daraufhin der CPU Daten- und Instruktionscache abgeschaltet.

8.2 Vorbereiten des Linuxstarts

Um das Betriebssystem Linux auf dem Mikrocontrollerboard starten zu können muss der Linux Kernel in Speicher geladen werden der im Adressraum der ARM CPU liegt. Eine zusätzliche Bedingung ist, dass der geladene Linux Kernel in einem Adressraum liegt der „ausführbar“ ist.

Der SDRAM Speicher auf dem Board erfüllt beide Bedingungen. Wie man der Abbildung 8.4 entnehmen kann, ist der SDRAM Speicher an die Adresse 0x20000000 in den Adressraum der CPU eingeblendet.

Damit im Speicher genügend Platz für die zum Start nötigen Parameter und der Page-table für die MMU Speicherverwaltung bleibt wird empfohlen das Linux Kernel `zImage` an eine um 32 KiByte vom unteren Ende des physikalischen Speichers entfernte Adresse zu laden [Linux]. In unserem Fall handelt es sich um die Adresse 0x20008000.

Um nicht nur den Linux Kernel selber erfolgreich zu starten sondern auch das Betriebssystem im Root Filesystem zu booten muß dem Kernel eine ganze Liste von Parametern übergeben werden. Die meisten werden in einer verketteten Liste übergeben, die kurz als `ATAG` Liste (ARM tagged list) bezeichnet wird.

Die einzige Ausnahme bildet die Maschinennummer. Um dem Linuxkernel mitzuteilen um welche Hardware es sich handelt, sind alle von Linux unterstützten ARM basierenden Maschinen durchnummeriert. Beim Start des Kernels wird diese Maschinennummer über das CPU Register r1 übergeben. Anders als z.B. PC Hardware deren Aufbau durch die Firma IBM standardisiert wurde sind die auf der ARM CPU basierenden Systeme sehr unterschiedlich aufgebaut. Es gibt keinen Standard der bestimmte Implementierungsdetails vorschreibt. Dazu sind die Computer sehr unterschiedlich. Auf der einen Seite gibt es kleine

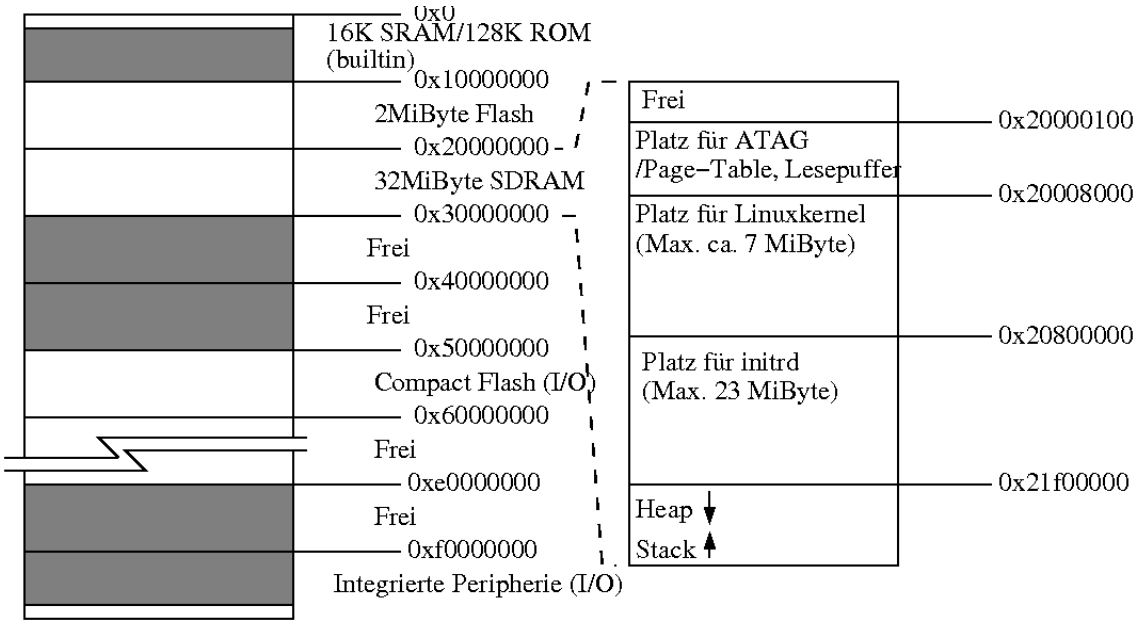

Abbildung 8.4: Speicheraufteilung des Boards, Speicherbelegung des Bootloaders. Nicht benutzer Addressraum ist grau gefärbt.

embedded Systeme, kleine „Handhelds“ und persönliche Organizer, auf der anderen Seite existieren aber auch Desktop-Computer mit dieser CPU. Je nach Typ und Aufbau der Hardware ist der Systemstart jeder dieser Maschinen sehr verschieden. Da jede Peripheriehardware unterschiedlich aufgebaut ist und unterschiedlich initialisiert werden muß, wird anhand dieser Maschinennummer festgelegt welcher Initalisierungscode verwendet wird und welche Hardware vom Linuxkernel angenommen wird. Programmcode zur Unterstützung nicht vorhandener Hardware muss bei der Übersetzung eines Linuxkernels nicht eingebunden werden.

Die Maschinennummer wird unter der URL
`http://www.arm.linux.org.uk/developer/machines/` zentral verwaltet. Durch die zentrale Verwaltung werden Kollisionen der Maschinennummern vermieden. Im Rahmen der Studienarbeit wurde für das Board eine Maschinennummer angefordert, und es wurde die Maschinennummer 882 zugeteilt.

Die folgenden Parameter werden in der `ATAG` Liste übergeben:

- Die Start– und Endadresse des physikalischen Hauptspeichers: In unserem Falle ist die Startaddresse 0x20000000 und die Größe 32 MiByte. Die in der CPU integierten 16 KiByte RAM werden nicht verwendet.

- Die Kernelparameterzeile: Dies ist eine Textzeile die es ermöglicht dem Linuxkernel Optionen zu übergeben oder das Startverhalten zu beinflussen. Die möglichen Zuwei-

sungen bestehen in durch Leerzeichen getrennte 'parameter=wert' Zuweisungen. Die Verwendung dieser Zuweisungen ist zum Teil etwas unübersichtlich, aber in [Linux] `Documentation/kernel-parameters.txt` gut dokumentiert. Für das OMI Board werden die Parameter `console=ttyS0,115200` zur Konfiguration der seriellen Konsole und `mtdparts=phys_mapped_flash:64K(boot),1024K(linux),2048K(root),64K(nvram)` zur Konfiguration der Partitionen des Flash ROMs verwendet (Näher erläutert im Abschnitt 10.4).

- Die Position und Größe der initrd im Hauptspeicher: Bei der initrd („inital ramdisk") handelt es sich um ein root Filesystem, dass direkt im Hauptspeicher untergebracht ist. Wie in Abbildung 8.4 gezeigt wird die initrd immer an die gleiche Adresse geladen.
- Gerätename oder Major– und Minornummer (siehe Kapitel 4) des Rootfilesystems.

Nachdem diese Liste im Speicher an Adresse 0x20000100 abgelegt wurde, kann der Linux Kernel folgendermaßen gestartet werden:

```
mov r0, #0          @; Im Register r0 muß der Wert 0 geladen werden.
mov r1, #882        @; r1 enthält die Maschinennummer des OMI Boards
mov r2, #0x20000100 @; r2 enthält die Speicherposition der tagged Liste
mov pc, #0x20008000 @; Hier wird schließlich der Linuxkernel gestartet
```

Der Linuxkernel kann also durch einfaches Ausführen des Codes beginnend mit dem Adressoffset 0 gestartet werden.
Der Bootloader führt diese Codesequenz (nur) nach erfolgreichem Laden des Linux Kernels und der initrd aus.

8.3 Partitionierung der Compact Flash Karte

Die Partitionstabelle, die vom Bootloader auf der Compact Flash Karte erwartet wird entspricht genau der Partitionstabelle die auch von IBM kompatiblen PCs verwendet wird.
Um den Programmcode des Bootloaders nicht zu komplex werden zu lassen sind nur primäre Paritionen unterstützt. Das Konzept erweiterter Partitionen wird hier daher nicht betrachtet und es sei auf die Literatur verwiesen [PCIntern].
Im sogenannten MBR, dem Master Boot Record ist die Partitionstabelle für die vier maximal möglichen primären Partitionen gespeichert. Der Aufbau ist wie in Tabelle 8.6 und 8.7 beschrieben.
Der Name MBR rührt daher, dass dieses Format auf PC Systemen nicht nur die Partitionstabelle des Datenträgers beinhaltet sondern auch einen (optionalen) Bootloader der das Betriebssystem vom Datenträger laden und starten kann.
In der Bootloader-Applikation werden die Daten die im MBR abgelegt sind in den folgenden C-Strukturen modelliert.
Der gesamte MBR wird durch die Struktur `mbr` beschrieben:

Offset	Größe	Inhalt	Aufgabe
0x00	0x1bd	Bootcode	Enthält ausführbaren Programmcode der bei PC das Betriebssystem lädt
0x1be	0x16	Daten für Partition 1	Partitionstabelle
0x1ce	0x16	Daten für Partition 2	
0x1de	0x16	Daten für Partition 3	
0x1ee	0x16	Daten für Partition 4	
0x1fe	0x02	Magischer Wert 0x55aa	Markiert Gültigkeit des Bootcodes

Tabelle 8.6: Aufbau des Master Boot Records

Offset	Größe	Bezeichnung
0x00	0x1	Bootbar?
0x01	0x3	CHS Daten des ersten Sektors
0x04	0x1	Partitionstyp
0x05	0x3	CHS Daten des letzten Sektors
0x08	0x4	LBA Startsektor
0x0c	0x4	Anzahl LBA Sektoren dieser Partition (relativer Offset)

Tabelle 8.7: Aufbau eines der vier Einträge in der Partitionstabelle

```
struct mbr {
    unsigned char bootcode[0x1be];
    struct partition_table p_table;
    unsigned char magic[2];
} __attribute__ ((__packed__));
```

Besonderes Augenmerk muß bei dieser Struktur darauf gelegt werden, dass der C-Compiler angewiesen wird diese Struktur „gepackt“ zu interpretieren. Im Falle des GNU C-Compilers passiert dies über die Modifizierungsanweisung `__attribute__ ((__packed__))`. Ohne diese Anweisung würde der C-Compiler 2 sogenannte „Pad“ Bytes zwischen die Strukturmitglieder `bootcode` und `p_table` einfügen um `p_table` auf eine durch 4 teilbare Adresse zu legen. Dies beschleunigt den Zugriff der CPU auf die Daten die in der Struktur gespeichert sind. Da es sich aber um eine Datenstruktur aus einer Spezifikation handelt die so auf dem Datenträger gespeichert ist, darf diese Optimierung hier nicht erfolgen.

Die Struktur `partition_table` ist ein Teil des MBR und beinhaltet die 4 gleich aufgebauten primären Partitionen. Eine extra Compileranweisung zum „packen“ der Struktur ist hier nicht erforderlich, da die einzelnen Einträge auf 16-Byte Grenzen angeordnet sind.

```
struct partition_table {
    struct partition_entry partition[4];
};
```

Schlußendlich bildet ein Partitionseintrag wie oben beschrieben die Basis der Gesamtstruktur.

```
struct partition_entry {
    unsigned char bootable;
    unsigned char head;
    unsigned char sector;
    unsigned char cylinder;
    unsigned char partition_type;
    unsigned char last_head;
    unsigned char last_sector;
    unsigned char last_cylinder;
    unsigned long int lba_start;
    unsigned long int lba_count_sectors;
};
```

Verwendung

Durch die Verwendung einer standardisierten PC Partitionstabelle kann die Compact Flash Karte mit dem Standard-Tool fdisk auf einem Linux System partitioniert werden. Da alle PC Betriebssysteme und viele Fremdsysteme dieses Format verstehen ist auch auf anderen Systemen leicht zu erkennen, dass die Compact Flash Karte gültige Daten enthält.

Die Karte wird für die Benutzung mit der Mikrocontrollerplatine in mindestens zwei Partitionen unterteilt. Nach Möglichkeit sollte die erste Partition zwischen 2 und 4 MiByte groß sein. Mindestens 2 MiByte damit der Linux-Kernel sicher hineinpasst, nicht wesentlich größer als 4 MiByte damit nicht unnötig Platz verschwendet wird (ein unkomprimierter Linux-Kernel (`zImage`) kann maximal 4 MiByte groß sein). Diese Partition wird als „Bootbar“ markiert und bekommt den Partitionstyp **0xda („Non-FS-Data“)**. Dieser Partitionstyp wurde gewählt da in dieser Partition kein Dateisystem abgelegt wird, sondern nur ein komprimierter Linuxkernel.

In der anderen Partition wird das Root-Filesystem abgelegt. Der Partitonstyp muß Typ 0x83 sein(„Linux-Dateisystem“). Prinzipiell könnte man minixfs, ext2 oder andere Linuxdateisystemformate verwenden. Im Bootloader wurde aber nur Unterstützung für das komprimierende cramfs implementiert.

Die in den Abschnitten 7.2 und 7.4 erzeugten Imagedateien (`rootfs.arm_nofpu.cramfs` bzw. `zImage`) können mit Hilfe des Kommandos `dd` direkt in die jeweiligen Partitionen geschrieben werden.

8.4 Laden des Linuxkernels und des Rootfilesystems

Um den Linuxkernel und das Root-Filesystem in den SDRAM Hauptspeicher zu laden wurden Funktionen zum Lesen der Daten von der Compact Flash Karte im Bootloader imple-

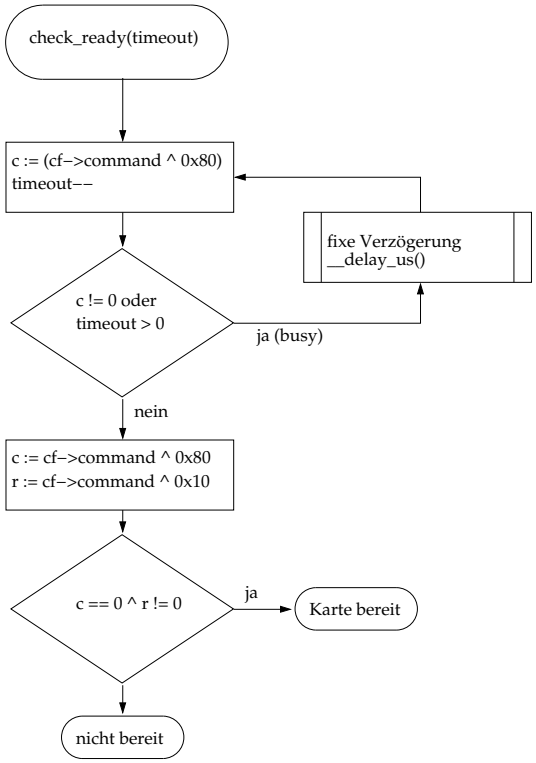

Abbildung 8.5: Flußdiagramm: Überprüfung der Bereitschaft der Compact Flash Karte. `cf->command` ist die Adresse des Compact Flash Kommando- und Statusregisters.

mentiert. Sie sind im folgenden Beschrieben

Nach dem Initialisieren der Hardware, wie es in Abschnitt 8.1 beschrieben ist, wird versucht einen Linuxkernel und das Rootfilesystem von einer evtl. eingesteckten Compact Flash Karte zu laden und zu starten.

Bevor allerdings Daten von der Compact Flash Karte gelesen werden können muß zuvor geprüft werden ob die Karte eingesteckt ist und ob sie bereit ist Kommandos zu verarbeiten.

Die Funktion `check_ready` deren Ablauf in Abbildung 8.5 gezeigt wird prüft ob die Karte bereit ist Kommands anzunehmen und zu verarbeiten. Hierzu wird das Busy Flag des Statusregisters der Karte ausgewertet. Solange das Busy Flag gesetzt ist, bedeutet dies, dass die Karte mit internen Operationen beschäftigt ist. In diesem Fall ist der Zustand der anderen Flags dieses Registers nicht definiert. Ist die Karte beim erstmaligen Ansprechen nicht bereit so wird im Programmcode davon ausgegangen, dass die Karte nicht eingesteckt ist und ein entsprechender Fehlercode an den Bootloader zurückgemeldet. Dieser zeigt dann eine Fehlermeldung an und zeigt danach dann den Monitorprompt um den Benutzer zu einer Eingabe aufzufordern.

Da die Karte bei späteren Aufrufen evtl. noch mit der Abarbeitung eines Kommandos be-

schäftigt ist wird im Falle der nicht-Bereitschaft eine beim Funktionsaufruf definierbare Zeit gewartet. Ist die Karte auch nach Erreichen des Timeoutwertes nicht bereit, so wird ein Fehlercode zurückgegeben. Andernfalls ist die Karte bereit, sobald das Busy Flag gelöscht wird. In diesem Fall wird das „Bereit“ Flag gelesen, zeigt dieses ebenfalls Bereitschaft der Karte für Kommandos, so wird die Ok Meldung als Ergebnis der Funktion zurückgegeben.

Die andere Basisfunktion zum Lesen von Daten von der Compact Flash Karte ist das Lesen eines Sektors. Das Flußdiagramm des Sektorlesens wird in Abbildung 8.6 gezeigt. Diese

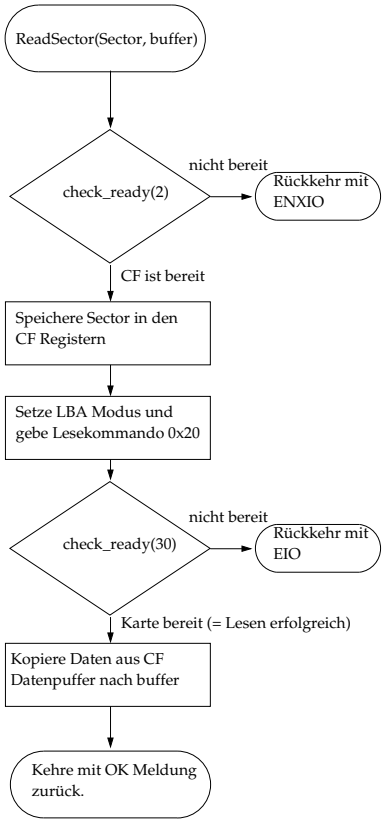

Abbildung 8.6: Flußdiagramm: Lese einen in einer LBA Adresse angegebenen Sektor der Compact Flash Karte. Die Funktion `check_ready` wird verwendet.

Funktion baut auf die Funktionalität von `check_ready()` auf um die Bereitschaft der Karte zu überprüfen und auf den Abschluß des Lesevorgangs zu warten. Beim Funktionsaufruf müssen zwei Parameter übergeben werden. Erstens der zu lesende Sektor und zweitens eine Adresse im Speicher. An diese angegebene Speicheradresse werden die gelesenen Daten des 512 Byte großen Sektors abgelegt. Die Sektoradresse wird dabei als LBA (logical block address) angegeben.

Falls die Compact Flash Karte nicht bereit ist, so wird ein entsprechender Fehlercode zurückgegeben. Diese Fehlercodes sind an die entsprechenden Linuxkernel Fehlercodes angelehnt. Ist die Compact Flash Karte schon vor dem Absetzen eines Kommandos nicht bereit, so wird

davon ausgegangen daß keine Karte eingesteckt ist. Es wird dann der Fehlercode für „No such device“ als Rückgabewert zurückgegeben. Ist die Karte erst nach dem Lesekommando nicht mehr in Bereitschaft, so wird ein „I/O Error“ als Resultat zurückgegeben. Dieser Fall trat bisher aber nicht ein.

Diese beiden in diesem Abschnitt vorgestellten Funktionen genügen um das Betriebssystem von der Compact Flash Karte einlesen zu können. Das Laden des Linuxkernels und der initrd geschieht dabei wie es im Flußdiagramm in Abbildung 8.7 gezeigt wird. Da sich beide Operationen sehr ähneln sind sie hier in einem Bild zusammengefasst. Der Linuxkernel der auf der Compact Flash Karte gespeichert ist, ist eine `zImage` Datei. Bei der initrd handelt es sich um ein cramfs.

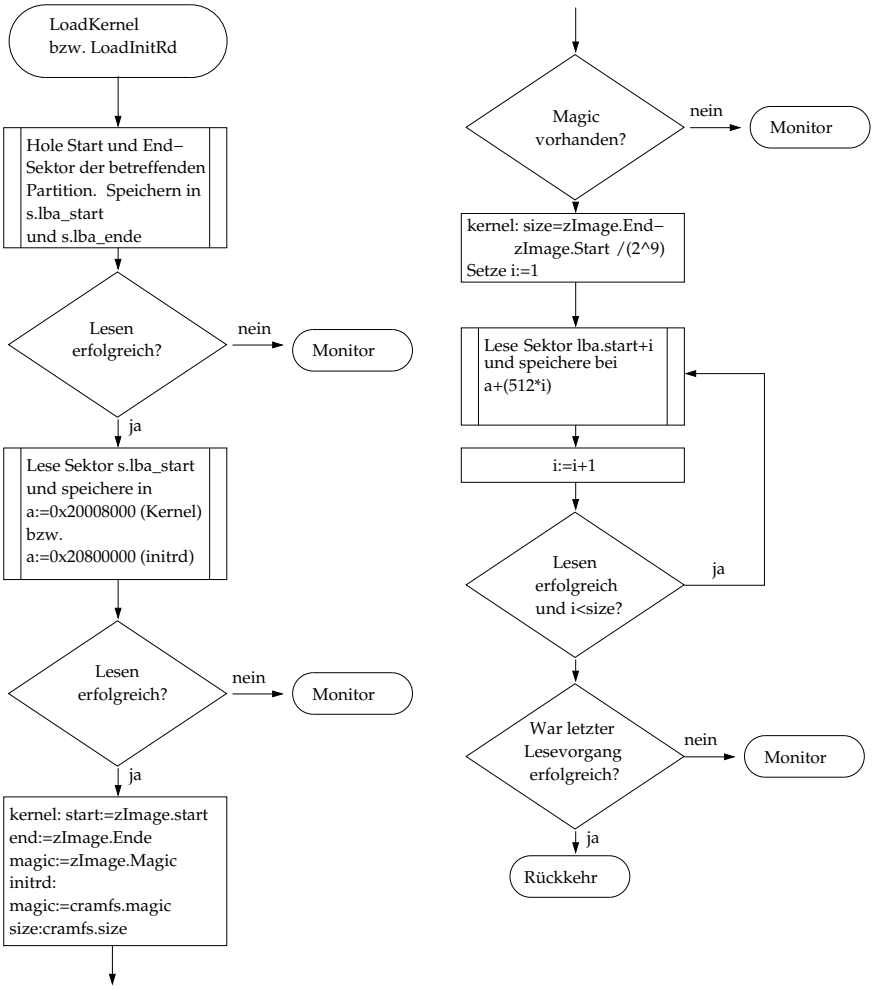

Abbildung 8.7: Flußdiagramm: Lesen des Linuxkernels und der initrd in den SDRAM Hauptspeicher. Die Routinen Lesen eines Sektors und Lesen der Partitionstabelle sind im Text erläutert.

Nachdem der Linuxkernel erfolgreich geladen wurde wird nach dem gleichen Schema auch

das Rootfilesystem geladen. Das Rootfilesystem wird in der ersten Linuxpartition gesucht wie es in Abschnitt 8.3 beschrieben ist. Mit dem implementierten Bootloader muß das Rootfilesystem auf einem cramfs basieren, da nur für dieses Dateisystem die Überprüfung des „magischen Wertes“ (engl. *magic value*, ein willkürlich festgelegter 32 Bit Wert der auf Vorhandensein geprüft werden kann) und eine Auswertung der realen Größe implementiert ist. Es wird also nicht die ganze Partition in den SDRAM Speicher geladen, sondern es werden nur die tatsächlich Daten enthaltenden Teile geladen. Die genaue Größe ist im Superblock des cramfs festgehalten, ähnlich wie im Linuxkernel `zImage`.

Falls in einem der Schritte ein Fehler auftritt, so wird vom Bootloader das integrierte Monitorprogramm gestartet. Die Bedienung des Monitorprogramms ist in Kapitel 9 beschrieben.

Falls sowohl der Linuxkernel auch die initrd erfolgreich geladen wurde, so wird vom Bootloader daraufhin die ATAG Liste wie in Abschnitt 8.2 beschrieben, erzeugt und daraufhin der Linuxkernel gestartet.

Abgeschlossene Konfiguration

Der implementierte Bootloader befindet sich auf der beigefügten CD im Verzeichnis `Bootloader` (Siehe Anhang, Kapitel 13).

9 Monitorprogramm

9.1 Die Monitorkommandos

Wenn der Bootloader kein Linux starten kann, so wird das eingebaute Monitorprogramm gestartet. Das eingebaute Monitorprogramm ermöglicht es den Linuxstart später durchzuführen (nachdem man z.B. eine passend beschriebene Compact Flash Karte eingesteckt hat), oder einfache Operationen durchzuführen.

Über die serielle Debug Schnittstelle können Kommandos an das Monitorprogramm gesendet werden und Ausgabedaten empfangen werden. Am PC kann zum Beispiel die Applikation minicom verwendet werden um das Monitorprogramm zu bedienen.

Nach dem Start und der Konfiguration der Applikation minicom sieht man – bei angeschlossenem Mikrocontrollerboard – den Prompt des Monitorprogramms '>'.

Dieses Zeichen zeigt die Bereitschaft an Kommandos entgegenzunehmen. Das vielleicht wichtigste Kommando ist ?, abgeschlossen durch das Zeilenschaltungszeichen, welches den in Listing 9.1 gezeigten Hilfebildschirm anzeigt.

```
>?
Commands available, [] means options:
all addresses are in hex
        c to frm size Copy in memory from <frm> to <to>, <size> bytes.
        d [adr] Show 1K memory contents at <adr>
        e adr Execute code at <adr>
        f frm size val Fill memory from <frm> to <frm+size> with <val>
        l ssec esec adr Load from CF, beginning with sector <ssec> ending at <esec> to <adr>
        start [line] Try to run Linux from compact flash, set <line> as kernel command line
        type [type] Set Linux machine type to <type>/Show type
        t option Set Linux kernel command line to <option> (only manual boot)
        x adr X−Modem transfer stored @<adr>.
>
```

Listing 9.1: Hilfebildschirm des Monitorprogramms

Im wesentlichen kann man die möglichen Monitorkommands in zwei Kategorien einteilen:

- Debugfunktionen, Funktionen um Daten zu übertragen, im Speicher zu betrachten, Speicherbereiche zu verschieben usw.

- Funktionen die im wesentlichen zum Ziel haben Linux zu laden oder zu starten oder diesen Start vorbereiten.

Eine wichtige Eigenschaft ist es, dass alle Angaben als Hexadezimalzahlen, also im 16er System eingegeben werden müssen. Eine evtl. Angabe in C-Notation (mit führendem 0x vor der Zahl) wird weggeschnitten, stört also nicht. Die Werte 10 . . . 15 werden, wie sonst auch üblich durch die Buchstaben a . . . f repräsentiert.

In der folgenden Auflistung werden die Kommandos und Ihre Verwendung alphabetisch sortiert vorgestellt.

c - Kopieren von Speicherinhalten

Das Kommando `c` (für engl. *copy*) dient dazu Blöcke im Speicher kopieren zu können. Folgende Argumente können verwendet werden:

```
c wohin woher wieviel
```

Hierbei werden die Angaben wohin, woher und wieviel natürlich in Hexadezimalzahlen angegeben, um z.B. 1 KiByte aus dem Flash Speicher in das SDRAM zu kopieren würde das Kommando wie folgt geschrieben werden müssen

```
c 0x20000000 0x10000000 0x400
```

Dieses Kommando ist immer erfolgreich. Die Operation wird auf jeden Fall ausgeführt, auch wenn dies zum Crash führt (weil z.B. das Monitorprogramm überschrieben wird).

d - Betrachten von Speicherinhalten

Mit dem Kommando `d` (für engl. *dump*) können Speicherinhalte angezeigt werden. Die Daten werden byteweise dargestellt, jeweils 16 Speicherzellen in einer Zeile, mit der ASCII Repräsentation der Inhalte am Zeilenende. Es werden immer 1024 Bytes, also 64 Zeilen, dargestellt, dann wird wieder der Monitorprompt gezeigt.

Die Verwendung des Kommandos erfolgt wie folgt:

```
d Speicheradresse
```

Die Verwendung einer Speicheraddresse ist optional. Bei Weglassen der Speicheradresse wird die Ausgabe an der Stelle fortgesetzt, die auf die zuletzt angezeigte Adresse folgt.

Die folgende Ausgabe zeigt einen typischen Speicherauszug wie er vom *dump* Kommando erzeugt wird.

```
>d 0x101f0000
 101F0000  6C 61 6E 5F 69 70 61 64  64 72 3D 31 33 34 2E 36 |lan_ipaddr=134.6|
 101F0010  30 2E 33 30 2E 38 35 0A  6C 61 6E 5F 6E 65 74 6D |0.30.85.lan_netm|
 101F0020  61 73 6B 3D 32 35 35 2E  32 35 35 2E 32 35 35 2E |ask=255.255.255.|
 101F0030  30 0A 6C 61 6E 5F 67 61  74 65 77 61 79 3D 31 33 |0.lan_gateway=13|
 101F0040  34 2E 36 30 2E 33 30 2E  39 39 0A 6C 61 6E 5F 64 |4.60.30.99.lan_d|
 101F0050  6E 73 3D 31 33 34 2E 36  30 2E 31 2E 31 31 31 0A |ns=134.60.1.111.|
 101F0060  6C 61 6E 5F 69 66 6E 61  6D 65 3D 65 74 68 30 0A |lan_ifname=eth0.|
 101F0070  6C 61 6E 5F 68 77 61 64  64 72 3D 30 30 3A 30 30 |lan_hwaddr=00:00|
 101F0080  3A 31 30 3A 39 39 3A 38  38 3A 37 37 0A 6C 61 6E |:10:99:88:77.lan|
 101F0090  5F 70 72 6F 74 6F 3D 73  74 61 74 69 63 0A 68 6F |_proto=static.ho|
 101F00A0  73 74 6E 61 6D 65 3D 62  61 62 62 61 67 65 0A 00 |stname=babbage..|
```

e - Ausführen von Programmcode bzw. dem Linuxkernel

Mit dem Kommando `e` (für engl. *execute*) kann Programmcode an einer vom Benutzer angegebenen Speicherstelle ausgeführt werden. Dieses Kommando ist, anders als in üblichen Monitorprogrammen, schon für das Starten von Linux vorbereitet und übergibt die nötigen Parameter wie es in Abschnitt 8.2 auf Seite 39 beschrieben ist.

Die Verwendung des Kommandos erfolgt wie folgt:

```
e Speicheradresse
```

Falls sich an dieser Speicherstelle kein gültiger Programmcode befindet wird das Mikrocontrollerboard mit hoher Wahrscheinlichkeit „abstürzen“.

f - Füllen von Speicherbereichen

Mit dem Kommando `f` (für engl. *fill*) kann ein Speicherbereich mit einem angegebenen Bytewert gefüllt werden. Diese Funktion kann dazu verwendet werden definierte Startbedinungen herzustellen indem der ganze SDRAM Speicher mit einem fixen Wert gefüllt wird. Die Verwendung erfolgt wie folgt:

```
f wo wieviel wert
```

Hierbei werden die Angaben wo, wieviel und wert in Hexadezimalzahlen angegeben. Um das erste KiByte des SDRAM Speichers mit dem Wert 0xAA zu füllen muß das Kommando wie folgt geschrieben werden müssen

```
f 0x20000000 0x400 0xaa
```

l - Lade vom Compact Flash

Mit dem Kommando `l` (für engl. *load*) können Daten aus dem Compact Flash Speicher in das SDRAM geladen werden. Diese Funktion kann dazu verwendet werden einzelne Sektoren in den Hauptspeicher zu laden. Um den Kernel oder die initrd zu laden steht aber das komfortablere `start` Kommando zur Verfügung.

Die Verwendung erfolgt wie folgt:

```
l Startsektor Endsektor Adresse
```

Hierbei ist Endsektor der letzte zu lesende Sektor (absolut, nicht relativ zum Startsektor).

Wieder soll hier ein Beispiel vorgestellt werden. Um die Sektoren 1 bis 10 von der Compact Flash Karte an die Adresse 0x20008000 zu lesen, beginnend mit dem Sektor 1 (Also 5120 Byte) muß das folgende Kommando gegeben werden:

```
l 1 10 0x20008000
```

s - zeige Partitionstabelle

Das Kommando `s` (für engl. *show*) zeigt die Partitionstabelle der eingelegten Compact Flash Karte in einem vereinfachten Format an. Das Kommando kennt keine Optionen. Eine typische Ausgabe sieht wie folgt aus:

```
>s
 Partition 01, LBA-Start: 0x0000001f, LBA-End: 0x00000f61, Type: 0xda, Boot?: 0x80
 Partition 02, LBA-Start: 0x00000f80, LBA-End: 0x00001ea3, Type: 0x83, Boot?: 0x00
 Partition 03, LBA-Start: 0x00002e23, LBA-End: 0x00000f61, Type: 0x81, Boot?: 0x00
 Partition 04, LBA-Start: 0x00000000, LBA-End: 0x00000000, Type: 0x00, Boot?: 0x00
```

Das Kommando liest den Sektor 0 der Compact Flash Karte ein und interpretiert die dort gefundenen Daten wie es im Abschnitt 8.3 auf Seite 41 zur Partitionstabelle beschrieben ist.

start - Lade und starte Linux

Das Kommando `start` führt einige Schritte nacheinander aus, die auch beim Start des Bootloaders ausgeführt werden und in Abschnitt 8.4 auf Seite 43 näher beschrieben sind. Es gibt einen optionalen Paramter, nämlich die Kernel Kommandozeile über die dem Linuxkernel Startparameter übergeben werden können.

Fehler werden abgefangen und führen zum Abbruch des Lade- oder Startvorgangs.

type - Setze oder überprüfe Maschinennummer

Das Kommando `type` ermöglicht es die aktuell eingestellte Maschinennummer zu verändern oder abzufragen. Die Maschinennummer wird vom Linuxkernel verwendet um die verwendete Hardware zu bestimmen. Diese Funktion ist im wesentlichen aus Testgründen implementiert da im Laufe der Studienarbeit die Maschinennummer festgelegt wurde, gibt es keinen Grund diese noch zu verändern. Die Voreinstellung beträgt 882.

```
>type

The current machine type is (hex) 0x882
```

t - Setze Kernelkommandozeile

Das Kommando `t` (für engl. *tag*) ermöglicht es die Kernelkommandozeile zu setzen, wenn der Linuxkernel „manuell“ über das `e` Kommando gestartet wird. Durch das mächtige start Kommando wird dieser Befehl normalerweise nicht benötigt.

x - Lade Daten mit xmodem

Das Kommando `x` ermöglicht es Daten vom Host-PC zum Mikrocontrollerboard mit dem X-Modem Protokoll zu übertragen. Hierbei erfolgt die Übertragung ebenfalls über die Debugschnittstelle. Der einzige Pflichtparameter ist die Zieladresse.
Mit folgendem Kommando kann man den Linuxkernel an die richtige Stelle mit dem X-Modem Protokoll in den SDRAM Speicher übertragen.

```
x 0x20008000
```

Nachdem dieses Kommando ausgeführt wurde, muss auf dem Host-PC ein entsprechendes X-Modem Programm gestartet werden das dann die Daten überträgt.

10 Das Rootfilesystem

10.1 init

Nach dem Start des Linuxkernels wird durch den Linuxkernel die Hardware für den Betrieb unter Linux vorbereitet. Nach der Eigeninitialisierung des Linuxkernels wird dann das Rootfilesystem gemountet und das Programm `init` gestartet.

Für die Systeminitialisierung wird ein vereinfachter Ansatz gewählt. Es gibt eine `/etc/inittab` Datei in der ein Systemzustand definiert ist. In diesem werden verschiedene Dienste gestartet die über Skripte in `/etc/init.d` gesteuert werden.

Hierzu existiert ein „Steuerskript“ `/etc/init.d/rcS` welches die weiteren Initialisierungsskripte im Verzeichnis `/etc/init.d` in alphabetischer Reihenfolge abarbeitet. Diese starten bzw. konfigurieren nacheinander die Dienste `syslog`, das Netzwerk, `dropbear_ssh` und `lighttpd`, die somit im gestarteten System zur Verfügung stehen.

Im folgenden werden die auf dem Mikrocontrollerboard zur Verfügung stehenden Schnittstellen betrachtet und ihre Verwendung mit Hilfe des embedded Linux.

10.2 Die seriellen RS-232 Schnittstellen

Die seriellen RS-232 Schnittstellen sind der typische Vertreter eines zeichenorientierten Gerätes. Im Falle des Mikrocontrollerboards stehen dem Benutzer zwei serielle Schnittstellen, nämlich das zeichenorientierte Gerät `/dev/ttyS0` (Major 4, Minor 0) und `/dev/ttyS1` (Major 4, Minor 1) zur Verfügung. Zusätzlich existiert noch `/dev/ttyS2`. Diese Schnittstelle wird aber vom Kernel selber verwaltet, da es sich hierbei um die integrierte IrDA Schnittstelle handelt.

Bei der Schnittstelle `/dev/ttyS0` handelt es sich um die schon in der Einleitung beschriebene, im Mikrocontroller integrierte Debugschnittstelle. Über die `inittab` Datei ist der `init` Prozess so konfiguriert, dass auf dieser Schnittstelle ein getty Prozess läuft. Das bedeutet, dass über diese Schnittstelle eine Terminalsitzung mit dem Linuxsystem aufgebaut werden kann.

Wie schon in der Einleitung erläutert handelt es sich bei `/dev/ttyS0` um eine 3-draht Schnittstelle ohne Handshakeleitungen. Bei der Verwendung dieser Schnittstelle für Terminalsitzungen muß daher das XON/XOFF Protokoll verwendet werden.

`/dev/ttyS1` steht zur freien Verfügung. Hierbei handelt es sich um eine vollverdrahtete RS-232 Schnittstelle. Mann kann hier entsprechende Peripheriegeräte anschliessen oder z.B. mit Hilfe der Applikation ppp eine Internetverbindung mit einer passenden Gegenstelle aufbauen.

In der Tabelle 10.1 ist aufgeführt welche Applikationen Daten über die integrierten seriellen Schnittstellen austauschen können.

Programm	Protokoll	Verwendungszweck	Besonderheiten
getty	-	Terminalsitzung ermöglichen	Start über `/etc/inittab`
ppp	ppp	TCP/IP Verbindung via ppp ermöglichen	
irattach	IrDA	IrDA Gerät konfigurieren (nur `/dev/ttyS2`)	

Tabelle 10.1: Anwendungen, die die serielle Schnittstelle verwenden

10.3 Die USB Schnittstelle

Bei der USB (*Universal Serial Bus*) Schnittstelle handelt es sich ebenfalls um eine serielle Schnittstelle. Im Gegensatz zu der im vorigen Abschnitt besprochenen RS-232 Schnittstelle handelt es sich hier um einen Schnittstellenstandard der maßgeblich von den Firmen Intel, Mikrosoft, Compaq und NEC in der Version 1.1 im September 1998 definiert wurde. Es handelt sich um eine schnelle, bidirektionale, isochrone, hot-plug fähige serielle Schnittstelle. Es gibt dafür eine Vielzahl von USB Peripheriegeräten, Angefangen von Computermäusen, Tastaturen über Compact Flash Lesegeräte bis hin zu Netzwerkadaptern und Fernsehkarten.

Der low-level Datenaustausch mit den USB Peripheriegeräten erfolgt über den im Mikrocontroller integrierten OHCI (*Open Host Controller Interface*) Controller und einem Linuxkernel-Modul.

Sobald ein USB Gerät eingesteckt wird, wird ein entsprechender Hotplug Event erzeugt. Er ermöglicht es ein zum Gerät passendes Linuxkernel-Modul nachzuladen, welches dann die high-level Gerätekommunikation ermöglicht. Es gibt sogenannte Geräteklassen die gleichartig, das heißt mit dem gleichen Gerätetreiber, angesprochen werden können. Solche Geräteklassen gibt es z.B. für Audiosysteme oder Massenspeicher.

In der Studienarbeit wurde dem Rootfilesystem ein embedded hotplug Subsystem namens `hotplug` hinzugefügt. Es ist in der Lage die Events zu verarbeiten und passende Linuxkernel Module nachzuladen sobald ein USB Gerät eingesteckt wird.

Hierzu wurde diese Schnittstelle mit verschiedenen USB Geräten getestet.

Zum einen wurde ein handelsüblicher USB Kartenleser („4 in 1“ Lesegerät von der Firma sitecom) mit verschiedenen Speicherkarten getestet. Mit dem Mikrocontrollerboard konnten die eingesteckten Medien partitioniert und genutzt werden. Hierzu werden die Kernelmodule `sd_mod`, `usb_core`, `scsi_mod`, `ohci-hcd` und `usb-storage` benötigt, die in der Kernelconfiguration enthalten sind.

Ein weiterer Test wurde mit einem USB Audio Adapter durchgeführt. Hierzu wurde ein „Terratec Aureon 5.1“ USB Soundsystem angeschlossen und mp3 und wav Audio-Dateien mit Hilfe der Applikation mpg123 abgespielt.

In beiden Fällen werden die erforderlichen Kernelmodule automatisch nachgeladen und die Geräte können wie vorgesehen genutzt werden.

Programm	Verwendungszweck	Besonderheiten
hotplug	Linuxkernelmodule nachladen	Start direkt durch den Linuxkernel.
/etc/hotplug	Konfigurationsdateien	Skripte die das Verhalten nach einem Hotplugevent steuern.

Tabelle 10.2: Programme zur Unterstützung des USB Betriebes.

10.4 Flash ROM Speicher verwenden

Auf dem Mikrocontrollerboard sind 2 MiByte Flashspeicher integriert. Dies ist zu wenig für Linuxkernel, Rootfilesystem, Bootloader und Konfigurationsdaten. Deshalb wurden in einem neuen Entwurf der Mikrocontrollerplatine 8 MiByte Flash vorgesehen.

Linux kann in der für das Mikrocontrollerboard verwendeten Version 2.6.13 den auf dem Board integrierten NOR Flash Speicher steuern und enthält schon entsprechende Gerätetreiber für die Flash ROM Gerätedateien. Leider passt ein Flash Speicher nicht so richtig zum beschriebenen Konzept von Block- und Zeichenorientierten Geräten. Eigentlich wäre ein Flash ROM ein idealer Kandidat für ein normales blockorientiertes Gerät. Allerdings steht dem die Ungleichbehandlung von Lese- und Schreibvorgängen durch die Flash Hardware entgegen. Die Linux Implementierung ist daher ein Kompromiss zwischen einfacher Benutzbarkeit und den durch die Hardware gegebenen Einschränkungen.

Ähnlich wie bei einer Festplatte kann der Flash Speicher in Partitionen unterteilt werden. Für das Mikrocontrollerboard wurden 4 voneinander getrennte Bereiche definiert: „Bootloader“ `(boot)` in den ersten 64 KiByte, „Linuxkernel“ `(linux)`, „Rootfilesystem“ `(root)` und „nvram“ `(nvram)` in den letzten 64 KiByte des Flash Speichers. Die Aufteilung und Benennung der Partitionen kann mittels der Kernelkommandozeilenoption `mtdparts` erfolgen.

Der Bereich `(boot)` enthält den im Kapitel 3 beschriebenen Startupcode um die Hardware zu konfigurieren und den Linuxkernel zu starten. `(linux)` und `(root)` wurde dafür vorgesehen den Linuxkernel und ein Rootfilesystem aufzunehmen. 2 MiByte abzüglich 128 KiByte sind aber zu wenig um einen Linuxkernel und ein Rootfilesystem zu beherbergen. Dies wird erst mit einem 8 MiByte Flash (s.o.) implementiert werden.

Die Konfigurationsdaten für das System werden in der Partition `(nvram)` gespeichert, so kann man leicht das System updaten („flashen“) und dabei die Konfiguration beibehalten.

In der `(nvram)` Partition befinden sich einfache Wertzuordnungen in der Form Schlüssel=Wert, die durch Newline-Zeichen voneinander getrennt sind. Dies wird schon vom Bootloader

verwendet, um die zu verwendende MAC Adresse des Ethernetcontrollers aus dem Flash zu lesen und zu programmieren. In der Tabelle 10.4 sind die im Rootfilesystem und Bootloader bereits implementierten Zuordnungen aufgeführt.

Um den Flashspeicher anzusprechen gibt es zwei Typen von Gerätedateien. Es gibt die zeichenorientierten Gerätedateien `/dev/mtd{0,1,2,3...}` und die blockorientierten Gerätedateien `/dev/mtdblock{0,1,2,3...}`. `/dev/mtd` ist ein Character Device und hat aufsteigend die Nummern Major 90 und Minor 0 . `/dev/mtdblock` ist ein Block Device und hat aufsteigend Major 31 und Minor 0.

Die Gerätedatei `/dev/mtd` ist zur Unterstützung von Programmen vorhanden, die Speicherinhalte direkt manipulieren. Die Hauptsteuerung erfolgt über den Systemaufruf `ioctl()`. `/dev/mtdblock` ist für die Verwendung des Flash Speichers als Dateisystem erforderlich, da das Programm 'mount' nur Blockgeräte zum Mounten verwenden kann.

Durch die Asymmetrie des Lese- und Schreibbetriebes bei dem verwendeten Flashspeicher, muss ein speziell für diesen Speichertyp geeignetes Dateisystem verwendet werden. Unter Linux kann hierzu das JFFS oder JFFS2 Dateisystem verwendet werden. Andere Dateisysteme sind für Flash Speicher nicht brauchbar, da hier zuviele Daten an die immer gleiche Stelle geschrieben werden. Dies wird der beschränkten Zahl an möglichen Schreibzyklen eines Flashbauteils nicht gerecht.

Programm	Verwendungszweck	Besonderheiten
flashcp	Daten in Flash Partition kopieren	
flash_erase	Flashpartition löschen	
mkfs.jffs2	JFFS2 Dateisystem anlegen	Muß im Hauptspeicher erfolgen, kann dann mittels `flashcp` in die Flashpartition kopiert werden.
nvram	Umgebungsvariablen aus dem nvram lesen	Liest Daten ausschließlich aus der Flash-Partition "nvram"

Tabelle 10.3: Anwendung zum Verändern und Lesen von Flashinhalten

10.5 Der Compact Flash Steckplatz

Da der Compact Flash Steckplatz vom Bootloader zum Laden des Linuxkernels und des Rootfilesystems verwendet wird liegt die Vermutung nahe, dass dieser auch unter Linux unterstützt sein sollte.

Allerdings ist der sog. I/O Mode von der Mikrocontrollerplatine nicht unterstützt, sondern nur der sog. MMIO (*memory mapped IO*) Modus. Dieser ist aber von den Linuxtreibern nicht unterstützt.

Auf eine Anpassung der Linuxtreiber wurde verzichtet da ein neuer Entwurf der Mikrocontrollerplatine den I/O Modus unterstützen wird.

Parameter	Beispiel	Verwendungszweck
lan_hwaddr	00:88:99:AA:BB:CC	MAC Adresse des Ethernetcontrollers
lan_ipaddr	134.60.30.85	IP Adresse der Ethernetschnittstelle
lan_netmask	255.255.255.0	IP Netzmaske der Ethernetschnittstelle
lan_gateway	134.60.30.99	Gatewayaddresse
lan_dns	134.60.1.111	IP Adresse eines DNS Servers. Wird im korrekten Format in `/etc/resolv.conf` eingetragen.
lan_ifname	eth0	Netzwerkinterface das konfiguriert werden soll.
lan_proto	static	Konfigurationsart der Schnittstelle „lan“. In diesem Fall eine statische Konfiguration.
hostname	babbage	Name des Systems.

Tabelle 10.4: In der Partition `(nvram)` definierte Wert/Parameter Zuordnungen. Diese können mit Hilfe des Programmes ‘nvram’ gelesen und verwendet werden.

Im Rootfilesystem wurde trotzdem das Programmpaket `pcmciautil` integriert. Damit kann zumindest der Typ der eingesteckten Karte angezeigt werden.

10.6 Die Ethernetschnittstelle

Während der integrierte Ethernetcontroller direkt von der Linuxkernel Version 2.6.13 mit den Patches zur Unterstützung des Mikrocontrollers unterstützt ist, ist dies bei dem verwendeten RMII (*Reduced Media Independent Interface*) Controller nicht der Fall.

Dieser RMII Controller stellt die Verbindung zwischen dem Ethernetcontroller und dem physikalischen Medium her. Im Falle der Mikrocontrollerplatine ist das physikalische Medium ein 100 MBit/s RJ-45 Anschluß. Der RMII Controller stellt also, vereinfacht gesprochen, die Layer 1 und 2 des ISO/OSI Schichtenmodells (wie in ISO 7498-1 definiert) bereit.

Bei dem RMII Controller handelt es sich um ein IC der Firma Micrel des Typs KSZ 8721. Dieser IC implementiert nicht nur den von der IEEE im 802.3 Standard standardisierten minimalen Registersatz sondern auch Erweiterungen darüber hinaus [Micrel].

Damit die verwendete Linuxversion den Controller verwenden kann, musste der Netzwerktreiber [Linux]`drivers/net/arm/at91_ether.c` entsprechend erweitert werden. Es wurde die Abfrage der Hardware ID und eine Interruptbehandlung für einen Wechsel des Linkstatus hinzugefügt. Damit ist dann eine Verwendung des Controllers unter Linux möglich. Die durchgeführten Änderungen wurden auf der arm-linux Mailingliste veröffentlicht und sind mittlerweile in [SAN] integriert.

Abgeschlossene Konfiguration

Diese Veränderungen am Linuxkernel befinden sich auf der beigefügten CD in der Datei OMI-Board-Patches-2.6.13 (siehe Anhang, Kapitel 13).

11 Ergebnis

11.1 Zusammenfassung

Mit der beschriebenen installierten Crosscompiler-Toolchain können sowohl Standalone-Programme, der Bootloader, der Linuxkernel als auch normale Benutzerapplikationen für embedded Linux übersetzt werden. Der beschriebene Bootloader initialisiert das Mikrocontrollerboard und Linux kann von einer Compact Flash Speicherkarte geladen und gestartet werden. In einem Rootfilesystem wurden übliche Systemwerkzeuge und ein nichtflüchtiges Konfigurationskonzept integriert, so dass nach dem Start des Betriebsystems eine typische Linuxarbeitsumgebung vorhanden ist, die auch Internetzugriff und Internetdienste bereitstellt. Im Mikrocontrollerboard vorhandene Peripherie wie Netzwerk oder serielle Schnittstellen können vom Benutzer verwendet werden und sind schon vorkonfiguriert. Verschiedene USB Peripheriegeräte wurden erfolgreich getestet und als komplexe Testapplikation wurde ein Webserver konfiguriert und getestet.

11.2 Ausblick

Mit Hilfe von Opensource Software konnte ein leistungsfähiges, flexibles Betriebssystem installiert werden, welches die auf dem Board vorhandene Hardware unterstützt. Hierdurch kann man sich auf die Erstellung einer Applikation konzentrieren und die vielfältigen Möglichkeiten eines Linux basierenden Betriebssystems nutzen.

In einem Redesign der Mikrocontrollerplatine konnten bereits gemachte Erfahrungen berücksichtigt werden. So werden hier 8 MiByte Flashspeicher (statt 2) verwendet. Zusätzlich unterstützt der Compact Flash Port dort auch den I/O Modus, so daß Compact Flash Karten mit den bereits zur Verfügung stehenden Linuxkernel-Modulen verwendet werden können.

Die Idee einer `(nvram)`, Partition die die Systemkonfiguration beinhaltet wurde im Moment nur für die Netzwerkgrundkonfiguration umgesetzt. Dies kann und soll deutlich erweitert werden und kann für die vielfältigsten Konfigurationsaufgaben verwendet werden.

Mit dem installierten Linux Betriebssystem kann das Board jetzt für die Simulation von Fahrzeugkommunikation verwendet werden.

12 Anhang

12.1 Bootloader

Funktion	I/O Leitung	Typ	Peripheriefunktion
LED	PIOC 0	Out	Verwendet für Kontroll-LED
LED	PIOC 1	Out	Verwendet für Kontroll-LED
LED	PIOC 13	Out	Verwendet für Kontroll-LED
LED	PIOC 14	Out	Verwendet für Kontroll-LED
LED	PIOC 15	Out	Verwendet für Kontroll-LED
Compact Flash	PIOC6	A	NWAIT - Wartesignal für Asynchrone Zugriffe
Compact Flash	PIOC7	A	A23 - Adressleitung A23
Compact Flash	PIOC9	A	A25 - Adressleitung A25
Compact Flash	PIOC10	A	Chip select Leitung CS für CF
Compact Flash	PIOC11	A	Chip select Leitung CE1 für CF
Compact Flash	PIOC12	A	Chip select Leitung CE2 für CF
SDRAM	PIOC16 - PIOC31	A	Datenleitungen D[16..31]
Ethernet	PIOA 8	A	ETXEN - Transmit enable
Ethernet	PIOA 9	A	ETX0 - Transmit data Bit 0
Ethernet	PIOA 10	A	ETX1 - Transmit data Bit 1
Ethernet	PIOA 11	A	ECRSDV - Carrier sense and data valid
Ethernet	PIOA 12	A	ERX0 - Receive data Bit 0
Ethernet	PIOA 13	A	ERX1 - Receive data Bit 1
Ethernet	PIOA 14	A	ERXER - Receive error
Ethernet	PIOA 15	A	EMDC - Management data clock
Ethernet	PIOA 16	A	EMDIO - Management data input/output
Debug Sch.	PIOA 30	A	DRX - Receive Leitung der Debug Schnittstelle
Debug Sch.	PIOA 31	A	DTX - Transmit Leitung der Debug Schnittstelle

Tabelle 12.1: Für den Compact Flash Steckplatz und Ethernet verwendete Anschlüsse des Mikrocontrollers. Hierbei wird die „Peripherie A“ Funktionalität des Mikrocontrollers verwendet.

13 Inhalt CD

Binary/ARM-zImage

Binary/cramfs.image

Bootloader/BootLoader.tar.gz

Bootloader/decompressed

Buildroot/buildroot-snapshot.tar.bz2

Buildroot/CONFIG-BUILDROOT

Buildroot/CONFIG-BUSYBOX

Buildroot/dl/

Buildroot/CONFIG-UCLIBC

Buildroot/README

Datenblätter/Compact-Flash

Datenblätter/Compact-Flash/CompactFlash3.0_spec.pdf

Datenblätter/Ethernet/KS8721.pdf

Datenblätter/Mikrocontroller

Datenblätter/Flash-ROM

Datenblätter/Flash-ROM/AM29lv160D.pdf

Datenblätter/Flash-ROM/AM29LV640MB.pdf

Datenblätter/Mikrocontroller/ARM 3rd party development Tools.pdf

Datenblätter/Mikrocontroller/ARM9 Memnonic Quick Reference Chart.pdf

Datenblätter/Mikrocontroller/ARM9 Product Overview.pdf

Datenblätter/Mikrocontroller/Atmel - AT91RM9200 TRM.pdf

Datenblätter/README

Datenblätter/SDRAM

Datenblätter/SDRAM/128MSDRAM.pdf

Datenblätter/SDRAM/MT48LC8M16A2.pdf

Linuxkernel/2.6.13-at91.patch.gz

Linuxkernel/CONFIG-BOARD-OMI

Linuxkernel/linux-2.6.13.tar.bz2

Linuxkernel/OMI-Board-Patches-2.6.13

Linuxkernel/README

Utilities/

Utilities/download

Utilities/nvram

14 Literaturverzeichnis

[Atmel] Atmel Corporation *AT91RM9200 Technical Reference Manual*
Atmel Corporation, Version B 2003 6, 9, 38

[SAN] Richard Bronson et.al. *Linux Kernel patches for the AT91RM9200 CPU*
`http://maxim.org.za/AT91RM9200/2.6/`, 2004 - 2005 25, 59

[KB] Kwikbyte *The Kwikbyte Development Kit*
`http://www.kwikbyte.com/`, 2004 - 2005 3

[CF] The Compact Flash Association *The CF+ and CompactFlash specification Revision 3.0*
The Compact Flash Association `http://www.compactflash.org/`, Revision 3 2004 8

[HW] Jochen Schwenniger *Aufbau einer ARM basierten Entwicklungsumgebung zum mobilen Test von Inter-Vehicle Datenübertragungen*
Studienarbeit, Abteilung OMI, Ulm 2005 3, 6, 7, 34

[Kofler] Michael Kofler *Linux*
Verlag Addison Wesley, 7. Auflage 2004 3

[Linux] Linus Torvalds, Russell King, Theodore Ts'o et.al. *Linux Kernelquellen, Version 2.6.13*
`http://www.kernel.org/`, 1991 - 2005 39, 41, 59

[Love] Robert Love *Linux-Kernel-Handbuch Leitfaden zu Design und Implementierung von Kernel 2.6*
Verlag Addison Wesley, 1. Auflage 2005

[Kernighan] Brian Kernighan, Dennis Ritchie *Programmieren in C*
Verlag Carl Hanser, 2. Ausgabe 1990

[Latex] Helmut Kopka *LaTeX Einführung Band 1*
Verlag Addison Wesley, 1. Auflage 1994

[Begleiter] Frank Mittelbach, Michael Goossens *LaTeX Begleiter, 2., überarbeitete und erweiterte Auflage*
Verlag Pearson Studium, 2. Auflage 2005

[PCIntern] Michael Tischler, Bruno Jennrich *PC Intern 5*
Verlag Data Becker, 1. Auflage 1995 41

[Wiki] verschiedene Autoren *The Wikimedia Projekt*
`http://de.wikipedia.org/`, 2003 - 2005

[uClibc] Erik Andersen et.al. *uClibc Quellen, Version 0.9.28*
`http://www.uclibc.org/`, 2000 - 2005 19

[micron] Micron Technology *Datenblatt zum SDRAM MT48LC8M16A2*
`http://www.micron.com/`, 2005 37

[AMDflash] Advanced Micro Devices *Datenblatt zum Flash AM29LV160D*
`http://www.spansion.com/datasheets/22358b6.pdf`, Oktober 2004 35, 36

[CFI] Intel *Common Flash Interface and Command sets*
`http://www.intel.com/design/flash/swb/cfi.htm`, April 2000

[Micrel] Micrel Inc. *Datenblatt zum Micrel KS8721BL*
`http://www.micrel.com/page.do?page=product-info/fastether_trans.jsp`, Rev 2.2 August 2003 59

15 Glossar

ad-hoc Ein ad-hoc Netzwerk ist ein Netzwerk das dynamisch gebildet wird und 2 bis n Teilnehmer haben kann.

CFI Common Flash Memory Interface. Standard der von den Firmen Intel, AMD, Sharp und Fujitsu definierter wurde. Er ermöglicht es die Eigenschaften von Flash Speichern standardisiert abzufragen.

CPU Central Processing Unit ist der Teil eines Computers, der die Steuerung der anderen Bestandteile eines Computersystems übernimmt. Die CPU wird dabei von einem in Speichern abgelegten Programm in From eines Maschinencodes gesteuert. Dieser Maschinencode ermöglicht grundsätzliche Aufgaben wie Lesen und Schreiben in den Arbeitsspeicher, arithmetische Operationen, Verwaltung eines Stapelspeichers und das Ausführen von bedingten und unbedingten Sprüngen sowie Aufruf von Unterprogrammen.

ELF Executable and Linkable Format, Dateiformat von Programmen, Objektdateien und dynamischen Bibliotheken auf einem Linux System. ELF ermöglicht das Einbinden dynamischer Bibliotheken und erlaubt die einfache Integration von Debuginformationen. Wird mittlerweile praktisch ausschließlich für Linux Programme und dynamische Bibliotheken verwendet.

Europaformat Eine Platine im Europaformat hat die Außenabmessungen 160 x 100 mm, durch das standardisierte Format ist die Herstellung und Konstruktion Platinen dieser Größe besonders günstig.

IDE Integrated Drive Electronics ist ein Schnittstellenstandard um eine Festplatte mit einem Computersystem zu verbinden. Dieser Standard wurde 1984 von der Firma Western Digital definiert und mittlerweile mehrfach erweitert. Dabei wurde immer Abwärtskompatibilität sichergestellt.

Inode Eine Inode ist ein Informationsknoten. In einem Unix Dateisystem werden in der Inode alle Eigenschaften einer Datei außer dem Namen und dem eigentlichen Inhalt gespeichert. In der Inode sind also Eigenschaften wie Eigentümer, Gruppe, Dateigröße, Zugriffsrechte, Positition der Blöcke auf der Platte und dergleichen mehr, gesichert.

IrDA Infrared Data Association - Als IrDA Schnittstelle werden Infrarot Schnittstellen bezeichnet die den von der IrDA definierten Protokollstandard implementieren

JFFS Journaling Flash File System, ein Dateisystem für Flash Speicher. Schreibzugriffe werden auf das ganze Gerät verteilt. Zusätzlich werden Schreibzugriffe blockweise durchgeführt.

KiByte Es ist üblich, dass große Datenmengen mit den gängigen Vorsilben für Maßeinheiten (kilo, mega usw.) auf der Basis von Zweierpotenzen 2^{10}, 2^{20} usw. dargestellt werden. Zur Unterscheidung zu den Vorsilben zur Basis 10 wurden hierzu bisher Großbuchstaben verwendet (Damit 1 kByte = 1000 Byte aber 1 KByte = 1024 Byte). In der Norm IEC 60027-2 wurde wegen der häufigen Verwechslung definiert, dass statt eines Großbuchstaben nun *Kilo Binary Byte*, abgekürzt *KiByte* oder *KiB* geschrieben werden soll.

MBR Der Name MBR für Master Boot Record ergibt sich daraus, dass seit dem ersten PC im ersten Sektor ein Bootprogramm untergebracht war welches das Betriebssystem von Diskette nachgeladen hat. Dies ist im Prinzip bis heute unverändert. Auf Festplatten sind lediglich 64 Byte abgezweigt um diese für die Partitionstabelle zu verwenden

MiByte siehe das bei KiByte geschriebene

Mikrocontroller Unter einem Mikrocontroller wird ein Mikroprozessor mit wichtigen Peripheriekomponenten in einem Gehäuse und zumeist auf einer Chipfläche zusammen integriert. Mit sehr wenigen externen Komponenten kann hiermit ein vollständiges Computersystem aufgebaut werden.

MMU Die Memory Management Unit ist eine Komponente um es dem Betriebssystem zu ermöglichen die vorhandene Hardware zu virtualisieren. Zusätzlich ist es dadurch möglich mehrere, unabhängige Prozesse quasi gleichzeitig auszuführen die sich gegenseitig nicht stören können.

PLL Mit PLL ist hier eine softwareseitig programmierbare phase locked loop gemeint.

POSIX Portable Operating System Interface for Unix - Eine Beschreibung einer einheitlichen Programmierschnittstelle für Betriebssysteme. Standardisiert durch die IEEE.

RISC Reduced Instruction Set CPU. Eine CPU mit vereinfachten Maschinencode-Befehlsumfang. Dies wird durch eine hohe Registerzahl und hohe Ausführungseffizienz gut kompensiert.

VGA Versatile Graphic Adapter, ein Grafikkartenstandard für IBM kompatible PC. Obwohl mittlerweile sehr überholt wirkend (Formal ist die größtmögliche Auflösung 640x480 Punkte bei 16 Farben) handelt es sich immer noch um den Quasistandard der von allen PC Grafikkarten implementiert wird.

XON/XOFF Eine Methode des Softwarehandshakes bei Terminalsitzungen. Durch das ASCII Zeichen XOFF (Ctrl-S) wird die Bildschirmausgabe angehalten. Durch das ASCII Zeichen XON (Ctrl-Q) wird sie wieder fortgesetzt. Dadurch kann der Zeichenstrom bei drohendem Pufferüberlauf zeitweise gestoppt werden.