Architektur-übergreifende Firmware-Entwicklung für eingebettete Systeme

Florian Frank

31.03.2010

Zusammenfassung

Inhaltsverzeichnis

Teil I

Einleitung

Software wird heutzutage auf leistungsfähigen Arbeitsplatzsystemen mit guter Ausstattung sowohl hinsichtlich Software als auch Hardware entwickelt.

Dies begünstigt zwar die schnelle Entwicklung neuer Software, führt aber auch dazu, dass die Software nicht mehr ohne Weiteres auf den Ziel-Systemen lauffähig ist, die nicht über eine vergleichbare Ausstattung verfügen.

Der Schritt von der voll ausgestatteten Entwickler-Workstation zum Endprodukt gestaltet sich für den Kunden zunehmend schwieriger. Als Lösung wird oftmals ein System als Endprodukt angeboten, welches einen Großteil der Entwicklungsumgebung nachbildet.

Ziel- und Entwicklungssysteme haben jedoch gänzlich verschiedene Anforderungen.

Ein Entwicklungssystem ist flexibel und gut ausgestattet, um die vielfältigen Anforderungen des Entwicklungsprozesses zu erfüllen. Ein Produktivsystem hingegen hat lediglich die Aufgabe, die Applikation auszuführen.

Eventuell werden die Rahmenbedingungen für ein Produktivsystem durch äußere Faktoren weiter eingeschränkt. So kann es beispielsweise durch Einsatzbedingungen notwendig werden, die Applikation auf ein eingebettetes System mit anderer Hardware-Architektur zu transferieren, oder es kann notwendig werden, durch Maßnahmen die Widerstandsfähigkeit des Systems bezüglich Standfestigkeit und Bedienungsfehler zu erhöhen.

Für diese Aufgabe muss eine entsprechend den Anforderungen angepasste Firmware für das eingebettete System erzeugt werden, die auf der einen Seite die Anforderungen des eingebetteten Systems berücksichtigt, andererseits aber natürlich auch die Funktion der Applikation sicherstellen muss.

Im Rahmen dieser Arbeit soll zunächst eine Bestandsaufnahme über existierende Build-Umgebungen für eingebettete Systeme durchgeführt werden. Hierbei sollen insbesondere deren Funktionalität und Erweiterbarkeit um eigene Software bewertet werden.

Auf Grund dieser Bewertung soll darauf aufbauend ein Konzept entwickelt werden, welches möglichst einfach die Erweiterbarkeit und Integrierbarkeit eigener Software in eingebettete Systeme gewährleistet, ohne jedoch die speziellen Aspekte eingebetteter Systeme, wie Architekturunabhängigkeit und Begrenztheit der Ressourcen außer Acht zu lassen.

Teil II
Bestandsaufnahme

Um ein angepasstes auf die oben genannten Anforderungen zugeschnittenes Build-System zu konzeptionieren, ist es zunächst erforderlich sich einen Überblick über die bereits auf dem Markt existierenden Freien Build-Umgebungen zu verschaffen und eine Bewertung dieser hinsichtlich ihrer Eignung für die Integration eigener Software vorzunehmen.

Hierbei ist zu beachten, dass bereits existierende Freie Build-Systeme unterschiedliche Ansätze und Konzepte verfolgen. Gerade in der Art und Weise, wie die Prämisse der architekturübergreifenden Arbeitsweise umgesetzt wird, unterscheiden sich die verfügbaren Build-Systeme stark voneinander.

1 Cross-Kompilierung allgemein

Cross-Kompilierung[1] ist ein neues Konzept in Relation zur Geschichte von Rechnern und Software. In den Anfängen der Geschichte der Rechner, war Cross-Kompilierung kein beachtenswertes Thema, da Software und Programme für eine ganz spezielle Architektur und Aufgabe geschrieben wurde. Der Ansatz, Software-Komponenten oder gar ganze Programme in anderem Zusammenhang wiederzuverwenden, existierte nicht.

Als sich herauskristallisierte, dass manche Architekturen und Prozessoren[2] wesentlich mehr Rechenleistung als andere hatten, und auch die Programmierer erkannt hatten, dass es von Vorteil sein kann, Software-Komponenten auch in anderen Programmen und Zusammenhängen wiederzuverwenden, war zu erkennen, dass auf manchen Architekturen Programme in wesentlich kürzerer Zeit in Maschinencode übersetzt werden konnten, als auf anderen. Die Idee des Cross-Kompilierens wurde somit geboren, um die Vorteile der schnelleren Architektur im Softwareentwicklungsprozess ausnutzen zu können.

Die grundlegende Idee hinter Cross-Compiling ist es, einen bestimmten Prozessor[3] zu nutzen, um Programme oder Software für einen anderen Pro-

[1] Compiler, der auf einer Plattform ausgeführt wird und Programmcode für eine andere Plattform, zum Beispiel ein anderes Betriebssystem oder eine andere Prozessorarchitektur, erzeugt. Eine typische Anwendung ist die Erstellung von Programmen für ein eingebettetes System, das selbst keine oder keine guten Werkzeuge zur Softwareerstellung enthält.

[2] Ein Prozessor ist eine Maschine oder eine elektronische Schaltung, welche gemäß übergebener Befehle andere Maschinen oder elektrische Schaltungen steuert. Am populärsten sind Prozessoren als zentrale Recheneinheiten von Computern, in denen sie Befehle von Software ausführen.

[3] Auch Host-System oder einfach nur HOST genannt

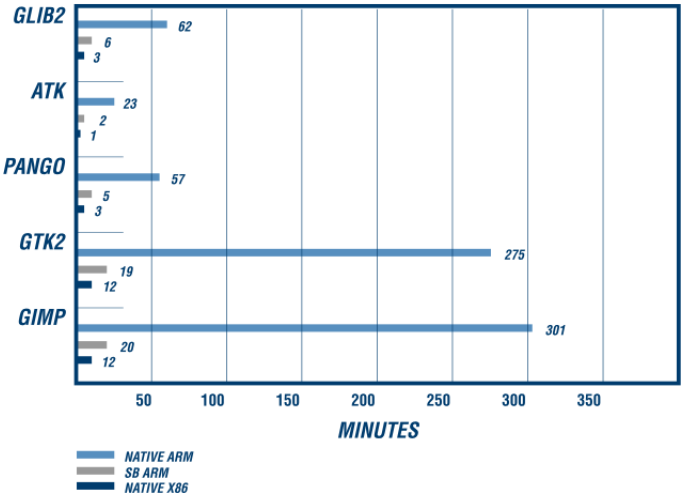


Abbildung 1:

zessor[4] zu kompilieren, der einer anderen Architekurfamilie angehört. Dies bedeutet, dass der Rechner, auf dem die Software kompiliert wird, selbst gar nicht in der Lage ist, die Software, die er gerade übersetzt selbst auszuführen. Denn die Software wurde für eine ganz andere Rechnerarchitektur übersetzt. Dies stellt eine der ersten Herausforderungen des Cross-Kompilierens dar. Eine detailliertere Betrachtung hierzu erfolgt zu einem späteren Zeitpunkt. Viele Build-Umgebungen setzten nämlich darauf, während des Build-Prozesses[5] bestimmte Programme auszuführen, um Anpassungen an das System vorzunehmen, oder Tests durchzuführen. Dies führt natürlich im Falle des Cross-Kompilierens zu Problemen, und lässt im Regelfall den gesamten Build-Prozess scheitern.

Um diesem Problem von vorne herein aus dem Weg zu gehen, setzen manche Programmierer darauf, ihre Programme direkt auf der Zielhardware zu kompilieren. Jedoch ist dies, auf Grund der gänzlich anders ausgelegten Architekturen, ein sehr langsamer und langwieriger Prozess, und kann in Zeiten der schnellen und optimierten Softwareentwicklung nicht mehr als eine gangbare Option angesehen werden.

Hierzu die Grafik auf dieser Seite, die sehr anschaulich die unterschiedlichen Kompilierzeiten zwischen Cross-Kompilierung und nativer Kompilierung direkt auf der Zielhardware darstellt.

Das im Vergleich auf dieser Seite verwendete ARM[6]-System bestand aus:

[4] Ziel-System oder kurz TARGET genannt

[5] Erstellungsprozess oder Build-Prozess (von englisch to build „etwas bauen“) bezeichnet in der Programmierung einen Vorgang, durch den ein fertiges Anwendungsprogramm automatisch erzeugt wird.

[6] Die ARM-Architektur ist ein Kern-Design für eine Familie von 32-Bit-Mikroprozessoren, die dem RISC-Konzept folgen. ARM steht für Advanced RISC Ma-

- Intel SA110 CPU 233Mhz
- 40GB IDE HDD
- 256MB RAM
- Debian Sarge mit Linux-Kernel 2.4.25

und das x86[7]-System bestand aus:

- Intel Xeon CPU 2.80Ghz
- 80GB IDE HDD
- 2GB RAM
- Debian Sarge mit Linux Kernel 2.6.8-1-686
- Scratchbox 1.0.1

1.1 Portabilität von Software

Im Allgemeinen tendieren Programmierer heutzutage dazu, ihre Programme und Software entsprechend portabel zu programmieren, was auch prinzipiell ein gutes Programmierparadigma darstellt. Um dies zu erreichen, stehen dem Programmierer viele Werkzeuge zur Verfügung, die ihn dabei unterstützen. Viele dieser Werkzeuge, werden zur Compile-Zeit benutzt, um Informationen über den Rechner, auf dem die Software kompiliert wird, zu erhalten. Entsprechend dieser Informationen werden dann Variablen und "defines" gesetzt, um in der Software darauf angemessen reagieren zu können. Diese Werkzeuge machen es dem Entwickler einfach, seine Software auf der aktuellen Rechnerplattform zu kompilieren. Dies ist ein guter Ansatz, allerdings stößt dieser an seine Grenzen, sobald Cross-Compiler eingesetzt werden, da eben hier die aktuelle Umgebung nicht mit der Zielumgebung übereinstimmt. Dies bedeutet, dass die Informationen nicht von der laufenden Maschine (HOST) stammen sollten, sondern von der Zielmaschine (TARGET).

2 Konzepte

Im Wesentlichen existieren zwei große konzeptionelle Unterscheidungsmerkmale für Build-Umgebungen für eingebettete Systeme. Diese stellen grundlegend unterschiedliche Herangehensweisen und Arbeitsweisen dar, weshalb eine genauere Betrachtung geboten ist.

chine.

[7] x86 ist die Abkürzung einer Mikroprozessor-Architektur und der damit verbundenen Befehlssätze, welche unter anderem vom Chip-Hersteller Intel entwickelt werden.

2.1 Architekturunabhängigkeit durch Softwareanpassungen

Um Software, die in sich natürlich bereits portabel gestaltet sein muss, auf verschiedenen System zu übersetzen, ist es oft notwendig, entweder nicht erwünschte oder auch für den gewählten Einsatzzweck einfach nur überflüssige Funktionalität abzuschalten, oder auch Anpassungen an die gewählte Plattform vorzunehmen.

Beispielsweise kann dies beinhalten, dass diverse "includes" verändert werden müssen, oder auch diverse plattformspezifische Variablen gesetzt werden müssen. Letztendlich resultiert jede Änderung des Programms, und dessen Einsatzzweck, in einem "Patch"[8], der dann oftmals in der Zukunft über die Softwareversionen hinweg gepflegt werden muss.

Auch wird des öfteren der zu verwendende Compiler direkt fest vorgeschrieben, zum Beispiel, weil sich das Programm nur mit einer bestimmten Version des Compilers einwandfrei übersetzten lässt. Gerade im C++ Bereich ist dies relativ häufig der Fall, dass sich hier binäre Schnittstellen ändern, so dass eine Neukompilierung der Programme und oftmals auch der zugehörigen Bibliotheken nötig wird.

Des weiteren wird auch oft angenommen, dass der Compiler einfach der Host-Compiler ist, und die Möglichkeit der Cross-Kompilierung wurde außer Acht gelassen. Hier ist dann mühevolle Kleinarbeit angezeigt, um die entsprechenden direkten Aufrufe des Host-Compilers in die äquivalenten Aufrufe des Cross-Compilers anzupassen. Die Optionen des Linkers, müssen meist ebenso nochmals manuell angefasst werden, da die Bibliotheken eben nun nicht an den üblichen Stellen im Dateisystem zu finden sind, wie dies bei der nativen Kompilierung gegeben wäre

Diese Überprüfungen und gegebenenfalls auch Anpassungen an die Software, sind bei kleineren Programmen durchaus noch handhabbar, sobald die Programme aber eine gewisse Komplexitivität und Größe überschreiten, ist es nur mit sehr großem Aufwand möglich, dies eben nicht nur ein einziges Mal zu tun, sondern auch über die Zeit und die Weiterentwicklung der Software hinweg weiter zu pflegen.

Und natürlich sind diese Maßnahmen auch für jedes einzelne der Programme nötig, die letztendlich auf dem eingebetteten System laufen sollen. Dies kann schnell den eigentlichen Entwicklungsaufwand für die eigene Software, die auf dem eingebetteten System laufen soll, übersteigen.

[8] Ein Patch (von engl. Flicken) für Quellcode enthält nur die geänderten Zeilen im Programmcode. Am weitesten verbreitet sind die Formate „Context-diff“ und „Unified-diff“. Diese Patches dienen dazu die Änderungen zu dokumentieren und kommunizieren. Patches sind ein essenzieller Bestandteil der Softwareentwicklung.

2.2 Architekturunabhängigkeit durch Emulation

Ein weiterer Ansatz, der gerade in letzter Zeit sehr viel Zuspruch erhalten hat, ist es, nicht die Software, die nativ problemlos kompiliert, auf Cross-Kompilierung anzupassen, sondern der Software, und zum Teil auch der gesamten Build-Umgebung, eben eine fast native Umgebung zu schaffen.

Genau an diesem Punkt setzen viele Virtualisierungs- und Emulationslösungen an. Es geht also darum, zumindest Teilen der Build-Umgebung ein möglichst realitätsnahes System zu emulieren.

Diesen Weg beschreitet unter anderem Scratchbox, welches später noch näher betrachtet werden wird. Hierbei wird eine Art Hybrid aus Crosscompilierung und Prozessor-Emulation angewendet. Die eigentliche Kompilierung geschieht ganz klassisch mittels eines Cross-Compilers, allerdings wird dieser durch eine Prozessor-Emulation unterstützt, die es ermöglicht zur Laufzeit des Build-Prozesses bereits Programme auszuführen, die für die Zielplattform übersetzt wurden. Ein typisches Beispiel, ist die Anwendung der Werkzeuge “autoconf” und “automake” welche zwar dafür sorgen, dass Software plattformunabhängig übersetzt werden kann, aber hierfür oft auf kleine selbst erstellte Test-Programme zurückgreifen. Diese Test-Programme werden dann natürlich durch den Cross-Compiler erzeugt, und sind auf der Host-Plattform nicht funktionsfähig. Bereits hier bricht dann der Build-Prozess ab.

Nun kann man wieder den Weg gehen, und versuchen die Skripte hinter “automake” und “autoconf” zu verändern oder anzupassen. Dadurch wäre man aber auch wieder bei einer Software-Anpassung gelandet, die man ja eigentlich vermeiden wollte.

Spätestens hier setzt nun die Emulation ein, und macht eben diese Programme dennoch ausführbar, indem eine CPU-Emulation diese ausführt.

Dieser Hybrid-Ansatz versucht den außerordentlichen Geschwindigkeitsgewinn aus der Cross-Kompilierung mit einer minimalen Emulation zu verbinden. Was allerdings nur in Grenzen gelingt, da nur bestimmte Programme emuliert werden können.

Ein sehr viel weitergehender Ansatz ist die vollständige Emulation einer Zielhardware inklusive eines emulierten Prozessors. Hierbei wird das Zielsystem in einer virtuellen Umgebung im Host-System nachgebildet. Es lässt sich darauf arbeiten, als hätte man direkt die Zielplattform vor sich.

Eine Prozessor-Emulation ist jedoch sehr komplex, aufwendig und ressourcenintensiv, Weshalb sich zunächst der oben genannte Hybrid-Ansatz entwickelt hat. Durch die stark steigende Rechenleistung auf modernen PCs oder Workstations, ist mittlerweile jedoch selbst eine vollständige Emulation bereits um ein vielfaches schneller und leistungsfähiger, als es die Zielplattform in der nächsten Zukunft sein könnte.

Für den Entwickler bietet dies bisher ungeahnte Möglichkeiten. Er kann nicht nur seine Software an einem normalen PC erstellen und testen, er kann seine Software auch auf eine andere Hardwareplattform anpassen, und auch ohne diese Hardwareplattform physikalisch zu besitzen, seine Software testen, und gegebenenfalls anpassen. Der letzte Test auf der physikalischen Hardware, kann so im Entwicklungsprozess sehr weit nach hinten verschoben werden.

3 OpenWRT/FreeWRT

FreeWRT[9] ist eine Linux-Distribution, die in eingebetteten Systemen wie WLAN-Geräten der Unternehmen Linksys[10] und ASUS[11] eingesetzt werden kann. Es handelt es sich um eine Abspaltung des OpenWRT-Projekts[12].

Das FreeWRT-Projekt verfolgt etwas andere Ziele als das OpenWRT-Projekt. Ziel der FreeWRT-Entwickler ist die Ausrichtung der Distribution auf professionellen Einsatz im Unternehmensumfeld. Die Abspaltung des OpenWRT-Projekts entstand, um den jeweiligen Interessen und Fähigkeiten der FreeWRT-Entwickler Rechnung zu tragen und weil eine entsprechende Lösung innerhalb von OpenWRT aus ihrer Sicht nicht möglich war. Da zwei der Initiatoren BSD-Nutzer beziehungsweise -Entwickler sind, hatte insbesondere die Umstellung auf ein Build-System, welches BSD-Ports[13] ähnelt, hohe Priorität.

Derzeit widmen sich die Entwickler vergleichsweise wenigen Modellen, u. a. spezialisiert sich die Unterstützung auf Modelle der Produktfamilien Linksys WRT54G(L/S/3G), Asus WL500g (Deluxe/Premium) und Netgear WGT634u, dafür werden diese Modelle jedoch umfassender unterstützt. Ein weiteres Ziel des Projektes ist es, regelmäßig stabile Versionen zu veröffentlichen, die aktuelle, gut ausgestattete Hardware unterstützen. Zum professionellen Konzept gehört auch der Kontakt zu den Hardwareherstellern, um dort Einfluss auf die Produktentwicklung zu nehmen.

Da Linksys für ihre Router das unter der GNU General Public License[14] im Quelltext frei verfügbare Linux nutzt und modifiziert, war sie gemäß dieser Lizenz ebenfalls dazu verpflichtet ihre Firmware wiederum frei zu veröffentlichen. Dadurch war es möglich, das Betriebssystem des Routers wiederum

[9] http://www.freewrt.org

[10] http://www.linksysbycisco.com/DE/de/home

[11] http://www.asus.de/

[12] http://www.openwrt.org

[13] Mit Ports werden Software-Paketverwaltungssysteme in der Welt der Unix-Derivate, speziell der BSD-Betriebssysteme bezeichnet. Ein Port bezeichnet meist ein Verzeichnis, in dem sich alle für die Installation benötigten Dateien sowie das Makefile („Kochrezept“) befinden.

[14] Die GNU General Public License (oft abgekürzt GPL, GGPL oder auch GNU GPL) ist eine von der Free Software Foundation herausgegebene Lizenz mit Copyleft für die Lizenzierung freier Software.

zu modifizieren, weiterzuentwickeln und als OpenWRT/FreeWRT zu veröffentlichen.

Die FreeWRT-Distribution erlaubt eine Handhabung, wie sie für Linux-Systeme typisch ist. Damit können auf WLAN-Geräten auch nicht vom Hersteller vorgesehene Funktionen genutzt werden. Die Konfiguration erfolgt über eine auf Textkonsole basierenden SSH[15]-Verbindung. Durch eine Kooperation mit dem Tntnet[16]-Projekt sind die ersten Konzepte für ein modulares und effizientes Webinterface fertiggestellt. In der Version FreeWRT 1.1 wird es die ersten Ansätze für Webinterface Frameworks (C++/PHP) geben.

Auf der Projekthomepage ist es zudem möglich, sich online eine individuelle Softwarekonfiguration zu erzeugen, welche dann automatisch erstellt und als Paket zum Download angeboten wird.

FreeWRT und OpenWRT ähneln sich in Konzept und Umsetzung sehr stark. Die Details liegen hier vor allem auf der Zielsetzung des Projektes. Während OpenWRT eine sehr viel breitere Unterstützung für eingebettete Systeme verschiedener Architekturen bietet und auf Aktualität wert legt, sieht sich FreeWRT eher etwas konservativer im Business-Umfeld, wo es auf Langzeitstabilität ankommt.

Auch wenn auf den ersten Blick für die weitere Betrachtung FreeWRT geeigneter scheint, so stellt jedoch die Beschränkung der möglichen Plattformen auf nur wenige MIPS[17]-basierte Systeme eine zu große Einschränkung dar. Deshalb bietet sich eine genauere Betrachtung von OpenWRT an.

3.1 Aufbau

OpenWRT lässt sich wie folgt beschreiben:

- minimalisiere Busybox/Linux Distribution unter der GNU Public License

- Zusammenstellung von Makefile und Werkzeugen um eine eingebettete Firmware zu erzeugen

- Softwarepakete und dazugehörige Software-Repositories

- Hardwarelieferanten, Paket-Betreuern und einigen sog. Kernel-Hackern

[15] Secure Shell oder SSH bezeichnet sowohl ein Netzwerkprotokoll als auch entsprechende Programme, mit deren Hilfe man auf eine sichere Art und Weise eine verschlüsselte Netzwerkverbindung mit einem entfernten Gerät herstellen kann.

[16] http://www.tntnet.org/

[17] Die MIPS-Architektur (engl. Microprocessor without interlocked pipeline stages; auf Deutsch etwa „Mikroprozessor ohne Pipeline-Sperren“) ist eine RISC-Prozessorarchitektur, die ab 1981 von John Hennessy und seinen Mitarbeitern an der Stanford-Universität entwickelt wurde.

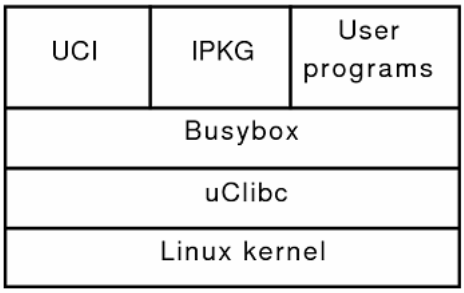


Abbildung 2:

3.1.1 uClibc

Im PC-Bereich stellt die uClibc eine besonders für eingebettete Linux-Systeme konzipierte, kleine C-Bibliothek dar. Die uClibc ist freie Software unter der GNU Lesser General Public License[18] lizenziert.

Die uClibc wurde ursprünglich konzipiert um uClinux[19], eine Version des Linux-Kernels ohne Speicherverwaltungseinheit (MMU) zu unterstützen und eignet sich deshalb hervorragend für Mikrocontroller. Daher auch das "uC" in ihrem Namen.

Im Gegensatz zur normalerweise gebräuchlichen glibc[20], der Standard C Library der meisten Linux-Distributionen, ist die uClibc sehr viel kleiner und kompakter. Sie benötigt sowohl wesentlich weniger Speicherplatz auf dem Datenträger, als auch im RAM-Speicher während ihrer Ausführung. Während die glibc vor allem darauf ausgelegt ist, alle relevanten C-Standards auf einer Vielzahl von Plattformen komplett zu unterstützen, beschränkt sich die uClibc nur auf eingebettete Linux-Systeme. Eine ihrer Stärken ist es, dass Funktionen aktiviert oder deaktiviert werden können, je nachdem wie viel Speicherplatz auf dem eingebetteten System zur Verfügung steht.

Die uClibc wurde jedoch auch ständig erweitert, und ist nicht mehr nur auf Mikrocontroller beschränkt. Sie arbeitet sowohl auf Standard- als auch MMU-losen Linux-Systeme. Sie unterstützt von Haus aus eine Vielzahl an Prozessoren, wie i386, ARM(Big/Little Endian), AVR32, m68k, MIPS (Big/-Little Endian), PowerPC und Sun Sparc, um hier nur die wichtigsten Prozessorarchitekturen zu nennen.

[18] Die GNU Lesser General Public License (LGPL) ist neben der GNU General Public License (GPL) eine weitere von der Free Software Foundation entwickelte Lizenz für Freie Software.

[19] µClinux, auch uClinux geschrieben, wird als „you-see-linux" ausgesprochen und steht für „Microcontroller Linux".

[20] glibc, die GNU C-Bibliothek, ist eine freie Implementierung der Standard C Library, die vom GNU-Projekt zusammen mit der GNU Compiler Collection entwickelt wird.

3.2 Busybox

Busybox ist ein Programm, das viele Standard-Unix-Dienstprogramme in einer einzelnen, kleinen ausführbaren Datei vereint. Es stellt die meisten Dienstprogramme, wenn auch oft in abgespeckter Form, bereit, die in der Single Unix Specification[21] gefordert werden. Darüber hinaus auch viele weitere Programme, die ein Benutzer eines Linux-Systems von diesem erwarten würde. Das Haupteinsatzgebiet von Busybox liegt wegen seiner geringen Größe gewöhnlich im Bereich der eingebetteten Systeme. Dank der Vorteile für diese Systeme wird es bereits in vielen eingebetteten Linux-Systeme von Haus aus mitgeliefert, so zum Beispiel auf dem Sharp Zaurus, dem Nokia 770, den Fritzboxen von AVM, vielen Linux-basierten Navigationssystemen oder TV-Receivern. Ein weiterer Einsatzort sind beinahe alle Installations-CDs der üblichen Linux-Distributionen. Busybox ist ebenso, wie viele andere Software, oder auch der Linux-Kernel selbst, unter der GNU Public License lizenziert, und fällt daher in den Bereich der Freien Software.

Busybox wird auch gerne als das "Schweizer Taschenmesser für eingebettete Linux-Systeme" bezeichnet. Um es für diesen Einsatz noch schlanker zu machen, wir es oft in Verbindung mit der oben genannten "uClibc" oder manchmal auch der "dietlibc" verwendet.

Busybox vereint Programme wie tar, gzip, grep, cat, ls, head, vi, dd, mount, ash, kill, touch, netstat... in einem einzigen kleinen binären Programm, und stellt damit sicher, dass fast alle grundlegenden Unix-Kommandos zur Verfügung stehen.

3.2.1 UCI

UCI bedeutet "Universal Configuration Interface", und stellt eine einheitliche Konfigurationsschnittstelle zwischen verschiedenen eingebetteten Geräten her. Es werden nicht mehr, wie in der Anfangszeit auch von OpenWRT, direkt die Möglichkeiten der spezifischen Hardware zur Speicherung von Informationen genutzt, sondern diese über eine Abstraktionsschicht direkt im Dateisystem abgelegt.

3.2.2 iPKG

iPKG, das eine Abkürzung für "Itsy Package Management System" ist wie viele andere Paketsystem unter Linux auch, ein unter der GNU Public License lizenziertes Paketverwaltungssystem.

[21] Die Single UNIX Specification (SUS) ist der Oberbegriff für eine Familie von Standards für Computer-Betriebssysteme, die durch deren Erfüllung den Markennamen UNIX® tragen dürfen.

Seine Entwicklung wurde nötig, weil die unter den Desktop-Systemen verbreiteten Paketverwaltungssysteme, wie APT/DPKG oder RPM, nicht für die beschränkten Ressourcen von eingebetteten Geräten geeignet waren. Zum einen brachten diese viel zu viele Metainformationen über die einzelnen Pakete mit, so dass alleine der Index über alle verfügbaren Pakete, die verfügbaren Hardwareressourcen der meisten eingebetteten Linux-Systeme gesprengt hätte. Des weiteren waren auch die Paketmanager selbst, mit ihren sehr komplizierten Abhängigkeitsbäumen zwar sehr mächtig, aber eben für eingebettete Systeme auch viel zu langsam und schwer-fällig. Auch was die Paketgröße anging, war Verbesserungsbedarf vorhanden. So wird auf einem eingebetteten Linux-System wohl nur sehr selten alle verfügbare Dokumentation zu einer Software vorhanden sein. Dies benötigt einfach zu viel Speicherplatz, der sich sinnvoller mit der eigentlichen Aufgabe des eingebetteten Systems nutzen lässt.

Also wurde es notwendig ein neues minimalistisches Paketverwaltungssystem zu entwerfen, welches das beste aus beiden Welten vereint. Hierzu wurde bei der Entwicklung auf folgende Punkte besonderen Wert gelegt:

- Die Kontrollprogramme selbst sind sehr klein.
- Die installierten Metadaten beschränken sich auf das Wichtigste
- Die verfügbaren Pakete sind möglichst klein. Der Paketbaum soll sehr fein granuliert sein.

Pakete haben für iPKG die Dateiendung .ipk. Das Format der Pakete als solches wurde an das Debian-Paketsystem angelehnt, so dass es auch relativ leicht ist, bestehende Debian-Pakete ins iPKG-Format zu konvertieren, dadurch müssen Pakete in der Regel nicht von Grund auf neu erstellt werden. Dies vereinfacht natürlich die Paketierung von Software außerordentlich.

3.3 Entwicklungsumgebung

Als Entwicklungsumgebung wird das von OpenWRT erstellte “Buildroot-ng” verwendet.

Das Hauptproblem einer Entwicklungsumgebung für eingebettete Systeme ist, dass diese auf einer anderen Hardware Architektur laufen als die Entwicklungsumgebung. Klassisch wird auf einer x86 oder x64 Maschine gearbeitet, während die eingebetteten Systeme auf einer anderen Hardware-Plattform laufen, wie z.B. ARM oder BCM47xx. Um nun Software für ein eingebettete Systeme zu erstellen ist ein Crosscompiler notwendig, welcher den Code in das entsprechende Embedded Format kompiliert.

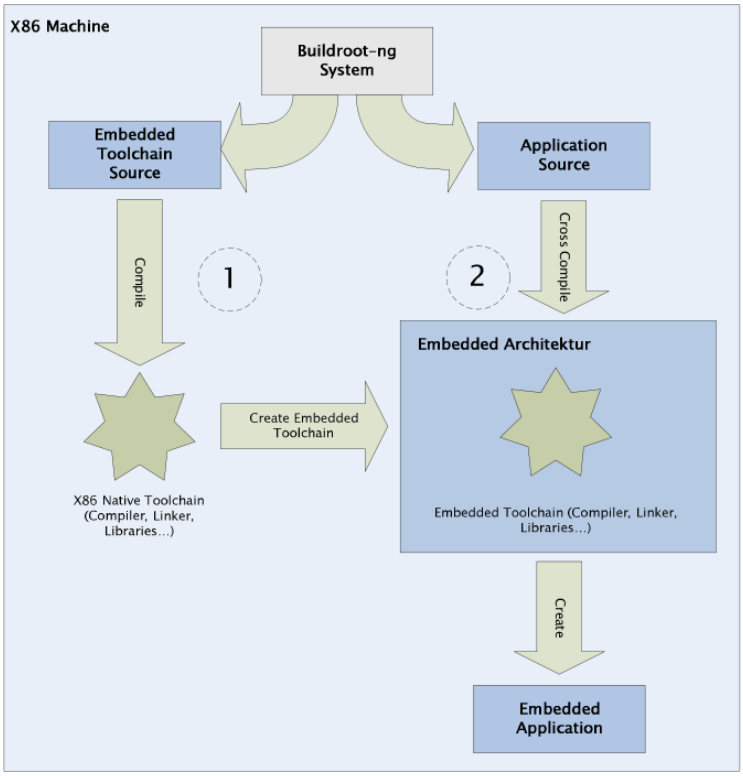

X86 Machine
Buildroot-ng System
Embedded Toolchain Source
Application Source
Compile
1
2
Cross Compile
Embedded Architektur
Create Embedded Toolchain
X86 Native Toolchain (Compiler, Linker, Libraries...)
Embedded Toolchain (Compiler, Linker, Libraries...)
Create
Embedded Application

Abbildung 3:

Ein schematischer Überblick, wie "Buildroot-ng" funktioniert ist in der Darstellung auf der vorherigen Seite gegeben.

Als erstes muss die Embedded Toolchain[22] erstellt werden. Diese Toolchain dient zur Kompilierung von Code und besteht aus folgenden Komponenten:

- Compiler
- Linker
- Assembler
- Debugger
- Make Tools

Ist die Toolchain erstellt worden, können nun Programme für die Architektur des eingebetteten Systems erstellt werden. Das Erstellen einer Cross Compiler Umgebung ist eine komplexe und schwierige Aufgabe, die mit dem "Buildroot-ng" System gemeistert wird. Für den Anwender von OpenWRT ist der Cross-Compile Vorgang transparent gestaltet. Die OpenWRT Entwicklungsumgebung besteht aus keiner einzigen binären Datei, sondern aus Skripts und Metadaten. Will man eine neue Firmware erstellen, werden die benötigten Quellcode-Dateien via Internet heruntergeladen, bei Bedarf gepatched und kompiliert.

3.4 Vorbedingungen

Damit eine funktionierende Firmware erstellt werden kann, müssen folgende Applikationen auf der Build-Maschine installiert sein:

- Autoconf
- Autotools
- Automake
- gcc
- g++
- zlib

[22] Als Toolchain (englisch Werkzeugkette) wird in der Softwareentwicklung eine systematische Sammlung von Werkzeug-Programmen bezeichnet, welche zur Erzeugung eines Produktes (meist eines anderen Programms oder eines System von Programmen) Verwendung findet.

- gawk
- bison
- ncurses
- perl
- python
- wget
- svn
- bison
- flex
- unzip
- bzip2
- tar

Außerdem sollte der ausführende User nach Möglichkeit keine Root[23]-Rechte verwenden. Das dient zur eigenen Sicherheit, damit keine Daten lokal überschrieben werden. Um zu überprüfen, ob alle Abhängigkeiten erfüllt sind, kann der Befehl "make prereq" verwendet werden.

3.5 Buildprozess

Um einen aktuellen Build von OpenWRT zu erstellen, kann man die aktuelle Version aus dem SVN[24] Repository auschecken. Im folgenden Beispiel wird im Heimat-Verzeichnis des Users ein "trunk" Unterverzeichnis erstellt:

```
svn checkout http://svn.openwrt.org/openwrt/trunk/ ~/kamikaze/
cd ~/kamikaze/
```

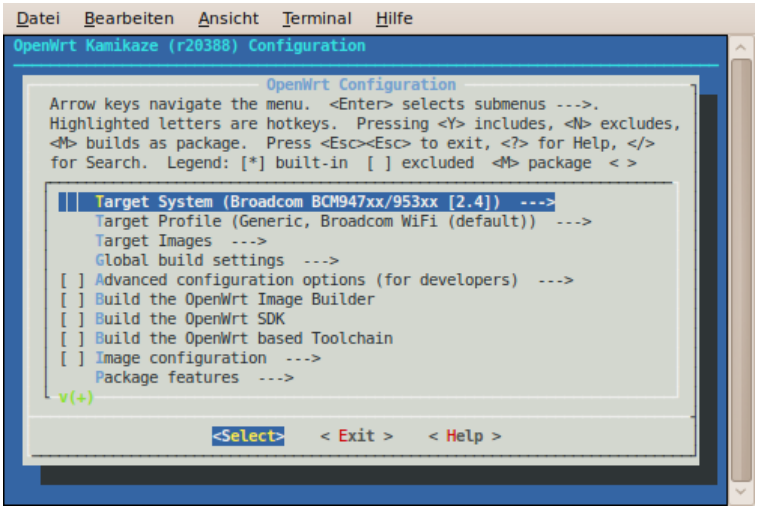
Jetzt müssen noch die Softwarepakete installiert werden:

```
./scripts/feeds update
./scripts/feeds install -a
```

[23] Das Root-Konto oder Superuser-Konto ist das Benutzerkonto, das bei der Installation eines Betriebssystems angelegt werden muss und mit größtmöglichen Zugriffsrechten ausgestattet ist.

[24] Apache Subversion (SVN) ist eine Freie Software zur Versionsverwaltung von Dateien und Verzeichnissen.

Abbildung 4:



Nun ist das gesamte Buildsystem von OpenWRT heruntergeladen und die Firmware kann konfiguriert und erstellt werden.

Die Konfiguration wird mit dem Befehl “make menuconfig” wie in der Abbildung auf dieser Seite erstellt. Bevor ein Dialog erscheint, werden noch diverse Vorbedingungen überprüft. Im folgenden Menü kann die ganze Firmware konfiguriert werden. Vom Hardware Typ, über Kernel Module bis zu den Applikationen kann alles ausgewählt werden.

Weitere häufig benötigte Funktionen der OpenWRT Entwicklungsumgebung:

- Erstellen eines einzelnen Pakete

```
make package/<package-name>-compile
```

- Erstellen eines einzelnen Pakete im Verbose-Modus

```
make package/<package-name>-compile V=99
```

- Löschen eines einzelnen Pakete

```
make package/<package-name>-clean V=99
```

- Schlägt das Kompilieren nach diversen Änderungen fehl, empfiehlt es sich, den Kernel neu zu erstellen. Die erfolgt mit folgendem Befehl:

```
make target/linux/clean
```

- Konfiguration des Kernels

```
make kernel_menuconfig V=99
```

3.6 Portierung

Die Herausforderung bei der Portierung von Applikationen für OpenWRT (oder generell eingebettete Systeme) ist, dass eingebettete Systeme meistens nur einen Bruchteil der Rechenkraft und sonstiger Ressourcen eines "normalen" Desktop Systems haben.

Um eine Applikation für OpenWRT zu portieren, sind folgende Schritte notwendig:

- Erstellen eines Makefile. Hier werden Metadaten über die Applikation gesammelt, d.h. Angaben wie Name, Version, URL[25], Checksumme, Konfigurationsoptionen und Installationsoptionen.
- Evtl. wird für die Applikation eine Code-Änderung benötigt, z. B. muss das (Original-) Makefile angepasst werden. In der Praxis zeigt sich, dass relativ viele Applikationen einen Patch benötigen.

3.6.1 Einfache Portierung

Ein einfaches Beispiel einer portierten Applikation:

```
#
# Copyright (C) 2008 OpenWrt.org
#
# This is free software, licensed under the GNU General Public License v2.
# See /LICENSE for more information.
#

include $(TOPDIR)/rules.mk
PKG_NAME:=bluez-hcidump
PKG_VERSION:=1.40
PKG_RELEASE:=1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.gz
```

[25] Als Uniform Resource Locator (URL, dt. „einheitlicher Quellenanzeiger") bezeichnet man eine Unterart von Uniform Resource Identifiern (URIs). URLs identifizieren und lokalisieren eine Ressource über das verwendete Netzwerkprotokoll (beispielsweise HTTP oder FTP) und den Ort (engl. location) der Ressource in Computernetzwerken.

```
PKG_SOURCE_URL:=http://bluez.sourceforge.net/download
PKG_MD5SUM:=c5793b79c3e7fea3a367c08c26c8e23c

PKG_BUILD_DIR:=$(BUILD_DIR)/$(PKG_NAME)-$(PKG_VERSION)
PKG_INSTALL_DIR:=$(PKG_BUILD_DIR)/ipkg-install

include $(INCLUDE_DIR)/package.mk

define Package/bluez-hcidump
  SECTION:=utils
  CATEGORY:=Utilities
  DEPENDS:=+bluez-libs
  TITLE:=Bluetooth packet analyzer
  URL:=http://www.bluez.org/
endef

define Build/Configure
        $(call Build/Configure/Default, \
                --with-bluez="$(STAGING_DIR)/usr/include" \
        )
endef

define Build/Compile
        $(MAKE) -C $(PKG_BUILD_DIR) \
                DESTDIR="$(PKG_INSTALL_DIR)" \
                all install
endef

define Package/bluez-hcidump/install
        $(INSTALL_DIR) $(1)/usr/sbin
        $(INSTALL_BIN) $(PKG_INSTALL_DIR)/usr/sbin/hcidump $(1)/usr/sbin/
endef

$(eval $(call BuildPackage,bluez-hcidump))
```

Zuerst werden Name, Version und URL der Applikation festgelegt. Anschließend wird der Menüeintrag definiert inklusive Abhängigkeiten von anderen Applikationen. Zum Schluss werden noch die Konfigurations- und Compiler Optionen angegeben.

Für jede Applikation wird im Paket-Verzeichnis ein Unterverzeichnis erstellt. In diesem Paket Unterverzeichnis ist das Makefile vorhanden. Es gibt zusätzlich ein spezielles Unterverzeichnis namens „patch“.

3.6.2 Komplexere Portierung

Als komplexere Applikation dient im Beispiel die Applikation LCD4Linux[26], hier der Aufbau des Verzeichnisses:

```
frank@luna:~/trunk/package/utils/lcd4linux$ ls -a -l
drwxr-xr-x 3 frank frank 4096 2010-02-10 11:04  files      <dir>
drwxr-xr-x 3 frank frank 4096 2010-02-28 13:23  patches    <dir>
-rw-r--r-- 1 frank frank 3960 2010-02-28 14:29  Config.in
-rw-r--r-- 1 frank frank 3025 2010-02-28 14:15  Makefile

frank@luna:~/trunk/package/utils/lcd4linux$ ls files/ -a -l
-rw-r--r-- 1 frank frank 22673 2010-02-10 11:04 lcd4linux.conf
-rw-r--r-- 1 frank frank   296 2010-02-10 11:04 lcd4linux.init

frank@luna:~/trunk/package/utils/lcd4linux$ ls patches/ -a -l
-rw-r--r-- 1 frank frank   420 2010-02-10 11:04 100-drv_RouterBoard.patch
-rw-r--r-- 1 frank frank   881 2010-02-10 11:04 120-remove_parport_outb.patch
-rw-r--r-- 1 frank frank   587 2010-02-10 11:04 140-no_repnop_T6963.patch
-rw-r--r-- 1 frank frank 78429 2010-02-12 03:52 150-addlibmpdclient.patch
```

[26] http://ssl.bulix.org/projects/lcd4linux/

Hier die Beschreibung der einzelnen Dateien:

- lcd4linux / Makefile – das generelle Makefile für diese Applikation
- lcd4linux / Config.in – eine Konfigurations-Datei um einzelne Applikationen auszuwählen
- lcd4linux / files / lcd4linux.conf – die Applikationskonfiguration
- lcd4linux / files / lcd4linux.init – init File um die Applikation als Daemon automatisch zu starten
- lcd4linux / patches / * – Applikationsspezifische Codeänderungen

Das Makefile im Detail:

```
include $(TOPDIR)/rules.mk

PKG_NAME:=lcd4linux
PKG_REV:=877
PKG_VERSION:=r$(PKG_REV)
PKG_RELEASE:=1

PKG_SOURCE:=$(PKG_NAME)-$(PKG_VERSION).tar.bz2
PKG_SOURCE_URL:=https://ssl.bulix.org/svn/lcd4linux/trunk/
PKG_SOURCE_SUBDIR:=lcd4linux-$(PKG_VERSION)
PKG_SOURCE_VERSION:=$(PKG_REV)
PKG_SOURCE_PROTO:=svn
PKG_FIXUP = libtool

include $(INCLUDE_DIR)/package.mk

define Package/lcd4linux
  SECTION:=utils
  CATEGORY:=Utilities
  DEPENDS:= +libusb
  TITLE:=LCD display utility
  URL:=http://lcd4linux.bulix.org/
  MENU:=1
endef

define Package/lcd4linux/config
        menu "Configuration"
             depends on PACKAGE_lcd4linux
             source "$(SOURCE)/Config.in"
        endmenu
endef

define Package/lcd4linux/description
        LCD4Linux is a small program that grabs information from the kernel and
        some subsystems and displays it on an external liquid crystal display.
endef

define Package/lcd4linux/conffiles
       /etc/lcd4linux.conf
endef

LCD4LINUX_DRIVERS:= \
        BeckmannEgle \
        BWCT \
        CrystalFontz \
<...>
        USBLCD \
        USBHUB \
        WincorNixdorf \
        X11 \

LCD4LINUX_PLUGINS:= \
        apm \
        asterisk \
        button_exec \
<...>
        uname \
```

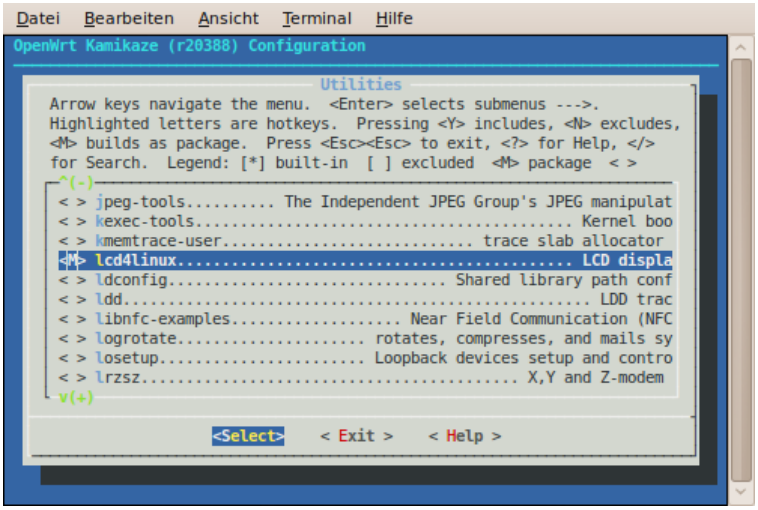
```
        uptime \
        wireless \
        xmms \

LCD4LINUX_CONFIGURE_DRIVERS:= \
        $(foreach c, $(LCD4LINUX_DRIVERS), \
                $(if $(CONFIG_LCD4LINUX_DRV_$(c)),$(c),) \
        )

LCD4LINUX_CONFIGURE_PLUGINS:= \
        $(foreach c, $(LCD4LINUX_PLUGINS), \
                $(if $(CONFIG_LCD4LINUX_PLUGIN_$(c)),$(c),) \
        )

EXTRA_CFLAGS+=-I$(STAGING_DIR)/usr/include -I$(STAGING_DIR)/include \
                -I$(STAGING_DIR)/usr/lib/libiconv/include
EXTRA_LDFLAGS+=-L$(STAGING_DIR)/usr/lib -Wl,-rpath-link,$(STAGING_DIR)/usr/lib \
                -L$(STAGING_DIR)/usr/lib/libiconv/lib

TARGET_CONFIGURE_OPTS+=\
        CC="$(TARGET_CC) $(EXTRA_CFLAGS) $(EXTRA_LDFLAGS)"

CONFIGURE_ARGS += \
        --without-x \
        --without-python \
        --with-drivers="$(LCD4LINUX_CONFIGURE_DRIVERS)" \
        --with-plugins="$(LCD4LINUX_CONFIGURE_PLUGINS)" \
        --disable-rpath \

define Build/Compile
        $(MAKE) -C $(PKG_BUILD_DIR) DESTDIR="$(PKG_INSTALL_DIR)" all install
endef

define Package/lcd4linux/install
        $(INSTALL_DIR) $(1)/usr/bin
        $(INSTALL_BIN) $(PKG_BUILD_DIR)/$(PKG_NAME) $(1)/usr/bin/
        $(INSTALL_DIR) $(1)/etc
        $(INSTALL_CONF) ./files/$(PKG_NAME).conf $(1)/etc/$(PKG_NAME).conf
        $(INSTALL_DIR) $(1)/etc/init.d
        $(INSTALL_BIN) ./files/$(PKG_NAME).init $(1)/etc/init.d/$(PKG_NAME)
endef

$(eval $(call BuildPackage,lcd4linux))
```

Im ersten Teil wird definiert, dass die Quellen direkt aus dem SVN Repository extrahiert werden sollen (PKG_SOURCE_PROTO:=svn).

Im Menü-Konfigurations-Abschnitt wird definiert, dass eine externe Datei namens “Config.in“ verwendet werden soll. Diese “Config.in“ Datei ist folgendermaßen aufgebaut:

```
comment "LCD4Linux Drivers ---"

config LCD4LINUX_DRV_BeckmannEgle
        bool
        prompt "BeckmannEgle"

config LCD4LINUX_DRV_Curses
        bool
        default y
        select PACKAGE_libncurses
        prompt "Curses (depends on libncurses, you need to install it manually!)"
<...>

comment "LCD4Linux Plugins ---"

config LCD4LINUX_PLUGIN_apm
        bool
        prompt "apm"

config LCD4LINUX_PLUGIN_gps
        bool
        select PACKAGE_libnmeap
        prompt "gps (depends on libnmeap, you need to install it manually!)"

<...>
```

Abbildung 5:



Für jeden möglichen Treiber und jedes mögliche Plugin wird ein Menüeintrag erstellt, dies ist nötig, um im Konfigurationsmenü von OpenWRT komfortabel die benötigten Optionen ein- oder auszuschalten. Hier können ebenfalls zusätzliche Abhängigkeiten direkt erfasst werden (select PACKAGE_name). Folgender Ausschnitt aus dem Makefile:

```
LCD4LINUX_CONFIGURE_DRIVERS:= \
        $(foreach c, $(LCD4LINUX_DRIVERS), \
                        $(if $(CONFIG_LCD4LINUX_DRV_$(c)),$(c),) \
        )
```

überprüft nun, welcher Eintrag ausgewählt wurde, und übergibt diesen String anschließend dem Konfigurations-Skript.

Bei der Installation der Applikation (define Package/lcd4linux/install) wird das Init-File und das Konfigurationsfile mitkopiert. Für den Endanwender ist das sehr praktisch, da

1. das Default Konfigurationsfile abgeändert werden kann und
2. der Befehl „/etc/init.d/lcd4linux enable“ die Applikation nach jedem Reboot automatisch startet.

Wie oben bereits erwähnt, benötigt LCD4Linux diverse Patch-Dateien, damit die Applikation kompiliert werden kann. Es folgt ein Beispiel eines einfachen Patches namens 100-drv_RouterBoard.patch:

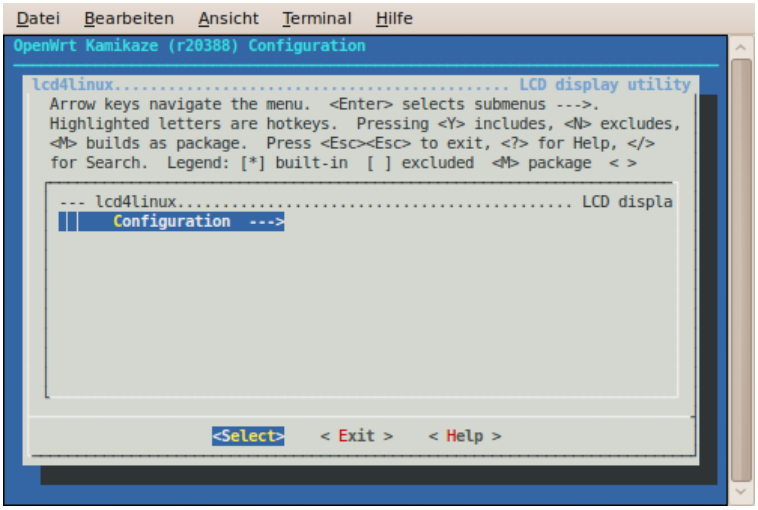


Abbildung 6:

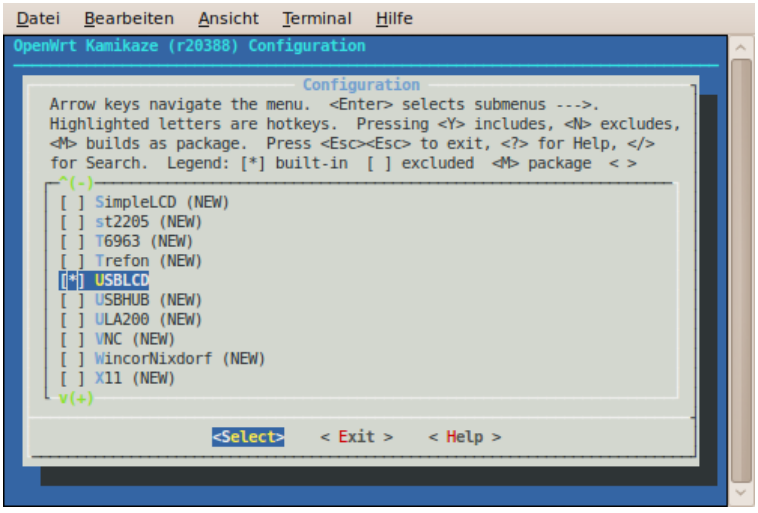


Abbildung 7:

```
Index: lcd4linux-847/drv_RouterBoard.c
===================================================================
--- lcd4linux-847.orig/drv_RouterBoard.c        2007-12-01 17:15:10.000000000 +0100
+++ lcd4linux-847/drv_RouterBoard.c     2007-12-01 17:15:10.000000000 +0100
@@ -106,7 +106,7 @@
 #include <string.h>
 #include <errno.h>
 #include <unistd.h>
-#include <sys/io.h>
+#include <asm/io.h>
 #include "debug.h"
 #include "cfg.h"
```

Dieser Patch korrigiert ein Include-Verzeichnis, welches von der verwendeten Kernel-Version abhängig ist.

Ein Patch kann jedoch auch funktionaler Natur sein, so entfernt der Patch "120-remove_parport_outb.patch" die Funktion, auf den Parallel Port zugreifen zu können, da kein Router einen Parallel-Port haben wird:

```
Index: lcd4linux-847/drv_generic_parport.c
===================================================================
--- lcd4linux-847.orig/drv_generic_parport.c    2007-12-01 17:15:10.000000000 +0100
+++ lcd4linux-847/drv_generic_parport.c 2007-12-01 17:15:11.000000000 +0100
@@ -39,16 +39,6 @@
 #include <sys/stat.h>
 #include <sys/ioctl.h>
-#ifdef HAVE_SYS_IO_H
-#include <sys/io.h>
-#define WITH_OUTB
-#else
-#ifdef HAVE_ASM_IO_H
-#include <asm/io.h>
-#define WITH_OUTB
-#endif
-#endif
-
 #if defined (HAVE_LINUX_PARPORT_H) && defined (HAVE_LINUX_PPDEV_H)
 #define WITH_PPDEV
 #include <linux/parport.h>
@@ -65,6 +55,11 @@
 #define PARPORT_STATUS_BUSY        0x80
 #endif

+#ifndef WITH_OUTB
+#define inb(foo) 0
+#define outb(foo,bar) 0
+#endif
+
 #if !defined(WITH_OUTB) && !defined(WITH_PPDEV)
 #error neither outb() nor ppdev() possible
 #error cannot compile parallel port driver
```

3.6.3 Patch erstellen

Ein Patch wird mit dem Command Line Tool „diff" erstellt. Ein einfaches Beispiel, es sollen zwei Files verglichen werden:

Datei test_a:

```
ich bin ein file
ich bestehe nur aus
text und solXXXXl die
funktion der applikation
diff aufzeigen
```

Datei test_b:

```
ich bin ein file
ich bestehe nur aus
text und soll die
funktion der applikation
diff aufzeigen
```

Patch erzeugen:

```
frank@luna:/tmp$ diff -urN test_a test_b
--- test_a      2008-04-28 15:48:44.000000000 +0200
+++ test_b      2008-04-28 15:49:16.000000000 +0200
@@ -1,5 +1,5 @@
 ich bin ein file
 ich bestehe nur aus
-text und solXXXXl die
+text und soll die
 funktion der applikation
 diff aufzeigen
```

Um einen Patch zu erstellen, wird das "trunk/build_dir/<ARCH>/<PACKAGE>" Verzeichnis verwendet. Die Originaldatei wird kopiert und umbenannt (z. B. in myfile.orig), danach werden die Änderungen der Datei hinzugefügt. Nun kann mit Werkzeug „diff" der Patch erstellt werden („diff –urN <PACKAGE>/myfile.orig <PACKAGE>/myfile.c > mein.patch").

Im Folgenden ein Beispiel, um die Applikation bemused-mpd-hack zu patchen:

```
~/trunk/build_dir/mipsel$ cp bemused-mpd-r060/main.cpp bemused-mpd-r060/main.orig

[edit bemused-mpd-r060/main.cpp]

~/trunk/build_dir/mipsel$ diff -urN bemused-mpd-r060/main.orig bemused-mpd-r060/main.cpp
--- bemused-mpd-r060/main.orig 2008-04-29 08:24:22.000000000 +0200
+++ bemused-mpd-r060/main.cpp   2008-04-29 08:25:47.000000000 +0200
@@ -27,6 +27,7 @@
 void show_usage(void)
 {
     cout << "Bemused Linux Server v" << VERSION_STRING << endl;
+    cout << "Test Patch" << endl;
     cout << "usage: bemusedlinuxserver [-d]" << endl;
     cout << "   -d daemonize" << endl;
 }
```

Diese Datei kann nun dem Patch-Verzeichnis des Paketes hinzugefügt werden, danach kann die Applikation neu erstellt werden.

3.7 Disklayout / Bootvorgang

In den meisten SOHO-Routern sind folgende Komponenten zu finden:

- Flash Chip (2MB, 4MB, 8MB)
- RAM, meistens 4x Flash Speicher Größe

Beim Aufstarten des Routers wird kein ROM[27] benötigt, die CPU führt den Code am Flashbeginn direkt aus. Dieser Part wird als Bootloader bezeichnet und kann mit dem BIOS eines Computers verglichen werden. Die Hauptaufgabe des Bootloaders besteht aus der Initialisierung der Hardware und des RAM's[28], danach wird die Firmware gebootet. Meistens hat der Bootloader noch einen Recovery-Mechanismus eingebaut, um die Firmware neu zu flashen. Eine beschädigte oder fehlerhafte Firmware kann also relativ einfach "repariert" werden und verhindert, dass der Router als teurer Briefbeschwerer endet.

Der Flash Chip besteht aus einem durchgängigen Speicherbereich, es gibt keine Partitionen, sondern nur "hard coded locations".

Wie oben bereits bemerkt wurde, führt der Bootloader den Beginn der Firmware-Location aus - es wird also lediglich ein ausführbarer Code benötigt. Im Normalfall würde hier der Kernel liegen. Da jedoch nur begrenzt Flash-Speicher zur Verfügung steht und der Kernel relativ groß ist, bietet es sich an, den Kernel zu komprimieren. OpenWRT komprimiert den Kernel mit dem effizienten LZMA[29] Verfahren.

Bis jetzt sind noch keine User oder Systemdaten vorhanden. Diese müssen natürlich auf einem Dateisystem abgespeichert werden. OpenWRT verwendet folgende Dateisysteme:

- SquashFS: ein komprimiertes READ ONLY Dateisystem.
- JFFS2: bedeutet "Journaling Flash File System Version 2".

Dieses Dateisystem ist beschreibbar und unterstützt "Wear Leveling". Wear Leveling versucht, die Lebenszeit der Flash Medien zu verlängern. Ein Flash Medium besteht aus mehreren Segmenten, jedes Segment kann zwischen 10'000 und 1'000'000 mal überschrieben werden. Wear Leveling versucht nun, Schreibzugriffe gleichmäßig auf alle Segmente zu verteilen und erreicht somit eine höhere Lebensdauer des Flash Mediums.

[27] Ein Nur-Lese-Speicher oder Festwertspeicher (engl. auch Read-only Memory, ROM) ist ein Datenspeicher, der nur lesbar ist, im normalen Betrieb aber nicht beschrieben werden kann und nicht flüchtig ist.

[28] Random-Access-Memory (das; engl.: random[-]access memory, zu Dt.: „Speicher mit wahlfreiem/direktem Zugriff"), abgekürzt RAM, ist ein Speicher, der besonders bei Computern als Arbeitsspeicher Verwendung findet.

[29] Der Lempel-Ziv-Markow-Algorithmus (LZMA) ist ein freier Datenkompressionsalgorithmus, der von Igor Pavlov seit 1998 entwickelt wird und vergleichsweise gute Kompressionsraten und eine hohe Geschwindigkeit beim Entpacken erreicht.

3.7.1 Warum 2 Dateisysteme?

Die beiden Dateisysteme SquashFS[30] und JFFS2 verwenden LZMA um die Daten zu komprimieren. SquashFS komprimiert jedoch 20-30% besser als das R/W JFFS Filesystem. Die Vorteile jedes Dateisystems:

- Pro SquashFS: Effizienteste Speichernutzung
- Pro JFFS: Beschreibbar, Journaling, Wear Leveling

Diese Vorteile werden nun folgendermaßen in OpenWRT genutzt:

- Installation des Basis OS Image erfolgt im SquashFS
- Installation von weiteren Applikationen (Packages) und User-Änderungen erfolgen im JFFS2 Dateisystem
- OpenWRT verwendet das virtuelle Dateisystem mini_fo, welches die SquashFS und JFFS2 Dateisysteme zusammenfasst und als ein ganzes Dateisystem Linux präsentiert. Wird eine Datei, welche in SquashFS enthalten ist, geändert und abgespeichert, leitet mini_fo die Datei um und speichert sie in der JFFS2 Partition ab. Die Verknüpfung zum originalen SquashFS wird dann gelöscht, die Datei selbst bleibt dann als "Leiche" zurück (da SquashFS Read Only ist). Das ist der Grund, warum möglichst wenige Applikationen im OS Image sein sollten.

Das Ganze hat einen interessanten Nebeneffekt: Es ermöglicht das Booten des Routers im Fail-Safe Mode. Hat man, warum auch immer, sein JFFS2 Dateisystem zerstört oder kann nicht mehr "normal" Booten, kann man durch Drücken einer Taste am Router während des Bootens in den Fail-Safe Mode schalten. In diesem Modus verhält er sich wie nach einem reflash - es werden nur Daten auf dem SquashFS berücksichtigt. So hat er per Default die IP Adresse 192.168.1.1.

Das Flash Layout kann nun vervollständigt werden:

- Bootloader
- Firmware
  - LZMA decompress
  - LZMA komprimierter Kernel
  - SquashFS R/O
- JFFS2 R/W

[30] SquashFS (.sfs) ist ein von Phillip Lougher entwickeltes freies (GPL) komprimiertes Dateisystem für GNU/Linux-Betriebssysteme, welches nur lesbar ist. SquashFS komprimiert Dateien, Inodes und Verzeichnisse, und unterstützt zur besseren Komprimierung Blockgrößen bis zu 1 MiB.

3.7.2 Bootvorgang im Detail

- Der Kernel startet vom SquashFS und startet /etc/preinit
- /etc/preinit startet /sbin/mount_root
- mount_root mountet die jffs2 Partition (/jffs) und verbindet diese mit der SquashFS Partition (/rom) um daraus ein neues virtuelles Root Dateisystem (/) zu erzeugen
- Der Systemstart geht nun wie gewohnt über /sbin/init weiter.

4 QEMU

Da im Folgenden öfter von der Emulation sowohl eines kompletten Systems, als auch nur von einer bestimmten CPU Architektur die Rede sein wird, bietet es sich an, zunächst eine genauere Betrachtung von QEMU vorzunehmen.

QEMU ist eine freie virtuelle Maschine, die die komplette Hardware eines Computers emuliert und durch die dynamische Übersetzung der Prozessor-Instruktionen für den Gast-Prozessor in Instruktionen für den Host-Prozessor eine sehr gute Ausführungsgeschwindigkeit erreicht.

QEMU emuliert derzeit Systeme mit den folgenden Prozessorarchitekturen:

- x86
- AMD64 und x86-64 (x64)
- PowerPC
- ARM
- Alpha[31]
- m68k[32] (Coldfire)
- MIPS
- Sparc32/64[33]

[31] Der Alpha-Prozessor wurde von der Computerfirma DEC entwickelt und 1992 unter der Bezeichnung „Alpha AXP“ auf den Markt gebracht. Es handelt sich um einen 64-Bit-RISC-Prozessor.

[32] Die Motorola 68000er-Familie, auch als 680x0 oder m68k bzw. 68k bezeichnet, ist eine Serie von CISC-Mikroprozessoren der Firma Motorola.

[33] Die SPARC-Architektur (Scalable Processor ARChitecture) ist eine Mikroprozessorarchitektur, die hauptsächlich in Produkten von Sun Microsystems Verwendung findet.

QEMU ist auf den Betriebssystemen GNU/Linux, Windows, FreeBSD, NetBSD, OpenBSD, OpenSolaris[34], OS/2/eComStation, DOS und Mac OS X lauffähig, kann den gesamten Status einer virtuellen Maschine speichern und auch ohne die Maschine anzuhalten auf ein anderes Host-System übertragen und dort weiterlaufen lassen (Live-Migration).

Unter Linux, BSD und Mac OS X unterstützt QEMU auch die Userspace-Emulation. Diese API-Emulation ermöglicht es, dass ausführbare Programme, die für andere dynamische Bibliotheken kompiliert wurden, im Userspace betrieben werden können. Dabei werden die Prozessoren x86, PowerPC, ARM, 32-bit MIPS, Sparc32/64 und ColdFire(m68k) unterstützt.

4.1 x86

Für virtuelle x86-Maschinen auf x86-Rechnern steht mit kqemu ein Zusatzmodul bereit, das einen erheblichen Geschwindigkeitszuwachs bewirkt. Es wird allerdings von der aktuellen Weiterentwicklung nicht mehr unterstützt, da diese auf KVM[35] fokussiert. Das Beschleuniger-Modul kqemu ist daher nur in QEMU bis Version 0.11 verwendbar.

Weiterer Geschwindigkeitszuwachs kann auf Linux-Hosts durch Verwendung der auf QEMU basierenden Kernel-based Virtual Machine (KVM) erzielt werden. Dafür ist jedoch ein Prozessor mit den Hardware-Virtualisierungstechniken von Intel (Intel VT) oder AMD (AMD-V) erforderlich.

Mittels HX DOS Extender ist QEMU auch in FreeDOS und DR-DOS lauffähig.

Emuliert wird neben dem Hauptprozessor auch:

- CD-ROM/DVD-Laufwerk über ISO-Abbild oder reales Laufwerk
- Diskettenlaufwerk
- Grafikkarte (Cirrus CLGD 5446 PCI VGA-Karte oder Standard-VGA-Grafikkarte mit Bochs-VESA-BIOS-Extensions – Hardware Level, inklusive aller Nichtstandardmodi, über einen experimentellen Patch auch mit einer vereinfachten 3D-Beschleunigung per OpenGL)
- Netzwerkkarte (NE2000-PCI-Netzwerkadapter) und ein DHCP-Server
- Parallel-Schnittstelle
- Systemlautsprecher

[34] OpenSolaris ist ein quelloffenes Unix-Betriebssystem von Sun Microsystems.

[35] Die Kernel-based Virtual Machine (KVM) ist eine Linux-Kernel-Infrastruktur für Virtualisierung und läuft auf x86-Hardware mit den Hardware-Virtualisierungstechniken von Intel (VT) oder AMD (AMD-V) und auf der System-z-Architektur.

- zwei PCI-ATA-Schnittstellen mit Unterstützung für maximal vier Festplatten-Abbilder im eigenen Format oder im Format von VMware, VirtualPC, Bochs, Knoppix (cloop) und dd (Rohformat)
- PCI und ISA-System (i440FX host PCI bridge und PIIX3 PCI to ISA bridge)
- PS/2-Maus und -Tastatur
- Serielle Schnittstelle
- Soundkarte (Soundblaster 16, ES1370 PCI, GUS)
- USB-Controller (Intel SB82371, UHCI)

Das PC-BIOS stammt von Bochs und das VGA-BIOS von Plex86/Bochs.

4.2 PowerPC

Als PowerPC-BIOS wird Open Hack'Ware, ein Open-Firmware-kompatibles BIOS, verwendet.

4.2.1 PowerMac

- QEMU emuliert die folgenden PowerMac-Peripheriegeräte:
- UniNorth PCI Bridge
- PCI-VGA-kompatible Grafikkarte mit VESA Bochs Extensions
- zwei PMAC-IDE-Interfaces mit Festplatten- und CD-ROM-Unterstützung
- NE2000-PCI-Adapter
- Non Volatile RAM
- VIA-CUDA mit ADB-Tastatur und -Maus

4.2.2 PReP

Die PowerPC Reference Platform (PReP) bezeichnet einen Standard für PowerPC-basierte Computer und soll eine Referenz-Implementation darstellen. PReP wurde bereits von der Common Hardware Reference Platform (CHRP) abgelöst.

QEMU emuliert die folgenden PReP-Peripheriegeräte:

- PCI Bridge
- PCI-VGA-kompatible Grafikkarte mit VESA Bochs Extensions
- zwei IDE-Interfaces mit Festplatten- und CD-ROM-Unterstützung
- Diskettenlaufwerk
- NE2000-Netzwerkadapter
- Serielle Schnittstelle
- PReP Non Volatile RAM
- PC-kompatible Tastatur und Maus

4.3 Sparc

Als BIOS der JavaStation (sun4m-Architektur) wurde bis Version 0.8.1 Proll, ein PROM-Ersatz, verwendet, in Version 0.8.2 wurde es durch OpenBIOS ersetzt.

QEMU emuliert die folgenden sun4m-Peripheriegeräte:

- IOMMU
- TCX Frame buffer
- Lance (Am7990) Ethernet
- Non Volatile RAM M48T08
- Slave I/O: timers, interrupt controllers, Zilog serial ports

5 Scratchbox (Maemo)

Ende Mai 2005 stellte der Mobilfunk-Riese Nokia ein Handheld-Gerät vor, das kein Telefon oder Smartphone ist und noch dazu ein Embedded-Linux als Betriebssystem einsetzt. Zeitgleich veröffentlichte Nokia auch die freie Entwicklungsumgebung Maemo Development Plattform, welche auf dem freien Buildsystem für eingebettete Systeme Scratchbox basiert. Maemo ist eine universelle Entwicklungsumgebung für eingebettete Systeme aller Art.

Die Anwendungs-Entwicklung unterscheidet sich auf der Maemo-Plattform kaum von der Programmierung herkömmlicher Desktop-Applikationen, da die grafische Oberfläche auf X11 und GTK+ basiert. So genannte Hildon-Funktionen aus den Bibliotheken “hildon-libs” und die “libosso” vereinfachen

die Bedienung von Handhelds. Die Interaktion zwischen Applikation und Desktop ist sehr an Gnome angelehnt, einige Funktionen verwenden zum Beispiel intensiv "gconv" und "dbus", Windowmanager ist die aus GPE bekannte Matchbox.

Grundstein der Maemo-Entwicklungsumgebung ist Scratchbox, mit der man mehrere Entwicklungssysteme für unterschiedliche Prozessor-Plattformen verwalten kann. Die passenden Scratchbox-Toolchains, die die richtigen Binärdateien für die jeweilige Prozessor-Plattform erzeugen, bekommt man ebenfalls auf der Scratchbox-Homepage.

5.1 Prozessor-Transparenz

Die GNU Compiler Toolchain unterstützt es schon seit vielen Jahren, Programme für eine andere Prozessorarchitektur als die des Hostsystems zu übersetzen. Diese Methode stößt aber schnell an Grenzen, wenn Werkzeuge wie die GNU Autotools zum Einsatz kommen - sie erzeugen meist Testprogramme für die Zielplattform und führen sie aus, etwa um bestimmte Prozessorfunktionen zu prüfen. Die GNU Compiler Toolchain erzeugt zwar die Testprogramme für die Zielplattform problemlos, sie funktionieren jedoch nicht auf der inkompatiblen CPU des Hostsystems - der Build-Prozess bricht hier ab.

Die Scratchbox bietet für diese Fälle die Funktion der CPU-Transparenz: Wann immer innerhalb der Scratchbox Code der Zielplattform ausgeführt werden soll, leitet Scratchbox den Code an eine Instanz weiter, die ihn auch ausführen kann - zum Beispiel an den Software-Emulator QEMU oder an "sbrsh", das den Code über eine SSH-Variante direkt auf dem Zielgerät laufen lässt, in diesem Fall dem Nokia 770. Alle x86-Binaries hingegen benutzen weiterhin die CPU des Entwicklungsrechners.

Die Maemo-SDKs enthalten alle für die Entwicklung auf der jeweiligen Plattform notwendigen Programme und Bibliotheken in genau der Version, die Sie für die Programmierung benötigen. Es handelt sich um virtuelle Linux-Systeme, Rootstraps genannt, die von den Bibliotheken des Wirtsystems vollkommen unabhängig sind. Mit dem virtuellen X-Server Xephyr können Sie sogar eine grafische Oberfläche benutzen, die vom Wirtsystem unabhängig ist.

5.2 Scratchbox vorbereiten

Die komplette Maemo-Entwicklungsumgebung besteht aus etlichen sehr großen Paketen. Da das Maemo-System selbst auf Debian aufbaut, ist die Installation auf einem herkömmlichen Debian-System oder einer Debian-basierten Distribution wie Ubuntu am leichtesten und zudem am besten getestet.

Zuerst installiert man Scratchbox, Maemo unterstützt zurzeit jedoch nur die Version 0.9.8.5. Insgesamt sind das ein halbes Dutzend Pakete mit ungefähr 200 MByte Umfang: "scratchbox-core", "scratchbox-devkit-debian" für die Erstellung von Debian-Paketen, "scratchbox-doctools", "scratchbox-libs", die Toolchain für den Arm-Prozessor "scratchbox-toolchain-arm-glibc" und die Toolchain für den x86-Prozessor des Wirtsystems, also die "scratchbox-toolchain-i686-glibc".

Während der Einrichtung fragt Scratchbox die Benutzer ab, die später mit dem System arbeiten sollen, und legt für sie neue Homeverzeichnisse unterhalb des Scratchbox-Verzeichnisses "/scratchbox" an. Diese Benutzer werden automatisch Mitglieder der neuen Benutzergruppe »sbox«, die Scratchbox ebenfalls während der Installation hinzufügt. Wichtig ist, dass man vor dem ersten Start von Scratchbox auch effektiv Mitglied dieser Gruppe ist, also eventuell noch einmal aus- und wieder einloggen.

Die Maemo-SDK-Pakete von umfassen noch einmal 125 MByte für i386 und 117 MByte für die Arm-Plattform und müssen nach der Scratchbox-Einrichtung unter "/scratchbox/packages" gespeichert sein. Die aktuelle stabile Version ist Release 1.0, für Experimentierfreudige gibt es noch den Release Kandidaten 5 der Version 1.1. Der Speicherort für die Maemo-Pakete ist wichtig, weil jeder Scratchbox-Entwickler seine eigene Maemo Development Plattform in seinem Scratchbox-Homeverzeichnis unterhalb von "/scratchbox/users" installiert, sodass mehrere Entwickler unabhängig voneinander arbeiten können.

Für die Maemo-Einrichtung ruft der jeweilige Benutzer den Befehl "scratchbox" auf. Innerhalb der Scratchbox sollte man zwei Umgebungsvariablen definieren:

```
export LANGUAGE=en_GB
export PAGER=less
```

Am besten schreibt man die beiden Zeilen in die Datei ".bash_profile", damit die Variablen bei jedem Neustart der Scratchbox wieder gesetzt werden.

5.3 Zwei Maemo-Targets

Scratchbox kann Programme für verschiedene Zielplattformen übersetzen, für jede gibt es ein eigenes so genanntes Target, das sind virtuelle Linux-Systeme. Für die Maemo-Entwicklung benötigen Sie ein Target für den Prozessor Ihres PC und eins für den Arm-Prozessor des Nokia 770 - die Toolchains habt man bereits zusammen mit der Scratchbox installiert. Zunächst legt man in der Scratchbox ein Target für den PC an:

```
sbox-config -ct SDK_PC
```

Der Name des Target muss nicht wie hier im Beispiel "SDK_PC" lauten, er sollte lediglich keine Leer- und Sonderzeichen enthalten. Das Konfigurationsprogramm fragt die zu benutzende Scratchbox-Toolchain ab, für die Zielplattform PC ist dies "i686". Bei der Frage nach der CPU-Transparenz wählt man "none", da der von dieser Toolchain erzeugte Code ohne Emulation auf dem Prozessor des Hostsystems läuft und keine Emulation benötigt. Als Development Kit, das der Paketerzeugung dient, wählt man Debian aus.

Ist die Konfiguration abgeschlossen, wählt man das PC-Target mit dem Befehl "sbox-config -st SDK_PC" als Standard-Target aus und aktiviert es. Zu diesem Zeitpunkt ist die Scratchbox vergleichbar mit einem virtuellen PC, auf dem man noch das Betriebssystem installieren muss - Rootstrap genannt.

Bei dem Maemo-Paket für die i386-Plattform handelt es sich genau um ein solches Rootstrap-Paket, das jeder Scratchbox-Benutzer einzeln mit folgendem Befehl einrichtet:

```
sbox-config -er /scratchbox/packages/Maemo_Dev_Platform_RS_v1.0_i386.tgz
```

Abschließend initialisiert man noch den Compiler und die Fakeroot-Umgebung mit den Befehlen "sbox-config -cc" und "sbox-config -cf". Nun ist man in der Lage, in der Scratchbox Programme für die Zielplattform i686 zu entwickeln, unter Verwendung aller Bibliotheken und Einschränkungen, die das Maemo-SDK mitbringt.

Die Einrichtung des Arm-Target unterscheidet sich kaum von jener der x86-Umgebung:

```
sbox-config -ct SDK_ARM
sbox-config -st SDK_ARM
sbox-config -er /scratchbox/packages/Maemo_Dev_Platform_RS_v1.1rc5_arm.tgz
sbox-config -cc sbox-config -cf
```

Für das Arm-Target wählt man aus der Liste den Arm-Compiler und als CPU-Transparenz-Modus "qemu-arm" - damit führt QEMU Arm-Binärprogramme aus, während x86-Binaries nativ auf der CPU des Entwicklungsrechners laufen. Mit den Befehlen "sbox-config -st SDK_PC" und "sbox-config -st SDK_ARM" schaltet man bei laufender Scratchbox zwischen den beiden Targets um.

Für den Test der selbst übersetzten Programme benötigt man noch einen separaten X-Server, der eine vereinfachte grafische Hildon-Oberfläche des Nokia 770 enthält. Am besten eignet sich hierfür "Xephyr", der im Gegensatz zu Xnest einen virtuellen X-Server mit anderen Farbtiefen und Erweiterungen als der X-Server des Hostsystems emulieren kann. Im Release Kandidaten 5 der Maemo Development Plattform 1.1 ist Xephyr bereits enthalten.

Für den Aufruf von Xephyr verwenden Sie am besten ein Skript, es startet Xephyr mit der Auflösung und Farbtiefe des Nokia 770. Läuft Xephyr, müssen Sie die grafische Oberfläche noch von Hand starten:

```
export DISPLAY=:2 af-sb-init.sh start
```

Zum Erproben eigener Programme eignet sich der virtuelle X-Server Xephyr besonders gut, da er eine andere Farbtiefe darstellen kann als der X-Server des Hostsystems.

```
#!/bin/sh -e
prefix=/scratchbox/users/${LOGNAME}/targets/SDK_PC/usr
export LD_LIBRARY_PATH=${prefix}/lib; export LD_LIBRARY_PATH
exec ${prefix}/bin/Xephyr :2 -host-cursor -screen 800x480x16 -dpi 96 -ac
```

Als Beispiel für die Programmentwicklung dient der Notizblock "maemopad". Dazu lädt man die Quellen innerhalb der Scratchbox per "wget" herunter, nachdem man auf das x86-Target umgeschaltet hat. Alternativ kann man auf dem Wirtsystem das Archiv ins Verzeichnis "/scratchbox/users/Benutzer/home/Benutzer" kopieren - das Homeverzeichnis der Scratchbox.

Wenn man die Quellen ausgepackt hat, ruft man das Skript "autogen.sh" im Quellenverzeichnis auf, das mit Hilfe der Autotools das "configure"-Skript und das Makefile anlegt. Da wir im i686-Target arbeiten, meldet "configure" beim Aufruf:

```
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
```

Wenn man hingegen auf das Arm-Target umschaltet, meldet der gleiche "configure"-Aufruf:

```
checking build system type... arm-unknown-linux-gnu
checking host system type... arm-unknown-linux-gnu
```

Die Autotools haben die virtuelle Arm-Umgebung erkannt und gehen davon aus, dass auf einem nativen Arm-System ein Arm-Binärprogramm erzeugt wird, ohne Crosscompiling. Die CPU-Transparenz von Scratchbox führt allerdings nur "configure" hinters Licht, alle x86-Binaries des Arm-Build-Systems laufen nach wie vor ohne Emulation direkt auf der Wirts-CPU. Ein abschließender Aufruf von "make" übersetzt das Programm auf dem jeweiligen Target.

Mit "file src/maemopad" überprüft man im Arm-Target, dass tatsächlich ein Arm-Programm entstanden ist und kein x86-Binary. Allerdings kann man den Notizblock nicht einfach aufrufen und ausprobieren, denn Nokia hat in die Oberfläche einige Sicherungsmechanismen eingebaut. So muss sich jede Applikation nach erfolgreichem Start bei der Oberfläche registrieren, andernfalls werden sie nach gewisser Zeit von einer Überwachungsinstanz zwangsweise beendet.

Zur vollständigen Anmeldung benötigt das Programm noch einige Dateien aus dem "data"-Verzeichnis, die man erst installieren müsste. Einen Ausweg bietet der Befehl "run-standalone.sh <Programm>", das die Zwangsregistrierung überflüssig macht und "maemopad" im virtuellen X-Server startet.

Die Anwendung direkt auf dem Entwicklungssystem starten zu können erleichtert die Entwicklung. Bei herkömmlichen Crosscompiler-Umgebungen muss das Programm erst auf der Zielplattform installiert werden.

5.4 Paketdienst

Auch bei der Paketherstellung sorgt Scratchbox dafür, dass "dpkg-buildpackage" die dem Target entsprechende Prozessor-Plattform erkennt. So liefert "dpkg-buildpackage -rfakeroot -b" auf dem x86-Target die Datei "maemopad_1.1_i386.deb", während der gleiche Aufruf auf dem Arm-System die Datei "maemopad_1.1_arm.deb" erzeugt.

Im Vergleich zu einem herkömmlichen Crosscompiler-System, bei dem man meist das Makefile und auch das "configure"-Skript von Hand anpassen muss, ist die Maemo Development Plattform in der Scratchbox eine große Erleichterung. Auch das Testen der Programme ist komfortabel: Statt die Programme ständig vom Entwicklungsrechner auf das Zielsystem zu übertragen probiert man mit Xephyr die Applikationen gleich auf dem virtuellen Desktop aus. Selbst der Platzbedarf von über einem halben GByte schreckt kaum - bei heutigen Festplatten findet Maemo bestimmt noch Platz.

Die Maemo Development Plattform eignet sich nicht nur für das Nokia 770 Internet Tablet, sondern für Embedded-Linux-Systeme allgemein.

6 Embedded Debian

Das Ziel von Embedded Debian[36] ist es, die freie Distribution Debian[37] so anzupassen, bzw. zu beeinflussen, dass es sich zu einer vorzüglichen Wahl für eingebettete Systeme entwickelt. Debian hat hierfür einige sehr gute Voraussetzungen. Es bietet schon von Beginn an Unterstützung für viele verschiedene Rechner-Architekturen, ist komplett herstellerunabhängig, und hat die größte Auswahl an fertiger freier Software, die direkt installiert werden kann, ohne auf Dritthersteller angewiesen zu sein. Allerdings ist die Distribution Debian sehr stark auf Systeme optimiert, welche als Server oder Desktop im PC-Bereich laufen. Eine der Richtlinien für Paketbetreuer verlangt, dass bei

[36] http://www.emdebian.org
[37] http://www.debian.org

der Pflege eines Paketes, alle "sinnvollen" Möglichkeiten und Optionen aktiviert werden sollen. Eine andere Richtlinie verlangt, immer die vollständige Dokumentation und Beispiele in einem Softwarepaket mit auszuliefern.

Im konkreten Fall bedeutet dies, dass Programme, wie z. B. der Mailserver "postfix"[38] so erzeugt werden, dass alle Möglichkeiten zur Anbindung an externe Datenbanken und Verzeichnisse, aktiviert und vorhanden sind, so dass der Benutzer nicht das von Debian bereitgestellte Paket nochmals verändern und/oder neu erzeugen muss, sollte er eine bestimmte Funktionalität benötigen.

Diese Herangehensweise mag für Desktop- oder Serversysteme hervorragend geeignet sein, da hier zum einen genügend Ressourcen vorhanden sind, um alle die benötigen Bibliotheken aufzunehmen, und zum anderen auch eben eine große Anzahl an möglichen Nutzungsprofilen abgedeckt werden soll, ohne den Benutzer wieder mit den Tiefen des Systems zu konfrontieren.

Für eingebettete Systeme stellt dies allerdings im Allgemeinen ein Ressourcen-Problem dar. Um hier bei der Betrachtung von "postfix" zu bleiben: Die Aktivierung der Authentifizierungsmöglichkeit im Mailserver, verlangt nicht im weiteren, viele Bibliotheken, wie z. B. "libsasl2", "libldap2", "libmysql5" oder "libssl". Und natürlich setzt sich diese Kaskade entsprechend fort.

So kann es schnell passieren, dass für eben diese eine Funktion in einer Software, sogleich mehrere Megabyte an zusätzlichen Bibliotheken und Zusatzprogrammen benötigt werden, auch wenn diese Funktion aller Wahrscheinlichkeit nach in einem eingebetteten System gar nicht verwendet wird.

Hier setzt Embedded Debian an, welches, neben anderen Zielen versucht, Debian soweit zu verkleinern und zu minimalisieren, dass dieser überschüssige Ballast für eingebettete Systeme weg fällt, aber die Flexibilität und die Vorteile von Debian auf eingebetteten System dennoch erhalten bleibt.

Um diesem Ziel gerecht zu werden, bietet Embedded Debian 2 Komponenten und Erweiterungen, die Debian als Ganzes für den Einsatz in eingebetteten Systemen vorbereiten, ohne jedoch auf viele Annehmlichkeiten einer vollwertigen Distribution zu verzichten.

Als eine große Komponente, bietet Embedded Debian sehr gute Möglichkeiten, um Crosscompiler und die evtl. benötigten Komponenten und Bibliotheken in ein bestehendes Debian-System zu integrieren, so dass die eigentliche Crosscompiling und Paketierung von Software maßgeblich vereinfacht wird.

Um nun wiederum zu einem funktionstüchtigen System zu gelangen, das auf die Bedürfnisse von eingebetteten System zugeschnitten ist, setzt Embedded

[38] Postfix ist ein Mail Transfer Agent für Unix und Unix-Derivate. Die Software sollte zum Entwicklungszeitpunkt eine kompatible Alternative zu Sendmail sein. Dabei achteten die Programmierer insbesondere auf Sicherheitsaspekte.

Debian darauf, soviel wie möglich aus der "normalen" Debian-Distribution zu verwenden, und nur Anpassungen vorzunehmen. Es ist sozusagen kein "bottom-up" Ansatz, wie beispielsweise OpenWRT, sondern ein "top-down" Ansatz, um ein Debian-System zu minimalisieren.

6.1 Basisinstallation

Hier soll nun gezeigt werden, wie eine Crosscompiler-Umgebung mit Hilfe von Embedded Debian aufgebaut und genutzt werden kann, und wie stark die von Embedded Debian bereitgestellten Werkzeuge die eigentliche Crosscompilierung vereinfachen.

Zunächst ist es notwendig ein normales Debian-System zu installieren. Im Rahmen dieser Betrachtung wurde auf Grund der stark fortschreitenden Entwicklung im Bereich Embedded Debian nicht auf das aktuelle stabile Debian-System "lenny" zurückgegriffen, sondern auf das bereits etwas weiter entwickelte Debian-System "squeeze", welches aber noch nicht offiziell veröffentlicht wurde.

Zu Beginn ist es erst einmal notwendig, die Werkzeuge und Programme von Embedded Debian zu installieren. Erfreulicherweise sind diese bereits fast komplett Bestandteil der offiziellen kommenden Debian-Distribution "squeeze", so dass sich diese sehr einfach installieren lassen:

```
apt-get install emdebian-buildsupport emdebian-grip emdebian-tools
```

Um die volle Funktionalität auszunutzen sind auch noch die mittlerweile in Debian "squeeze" bereits integrierten Programme "apt-cross" und vor allem "dpkg-cross" notwendig.

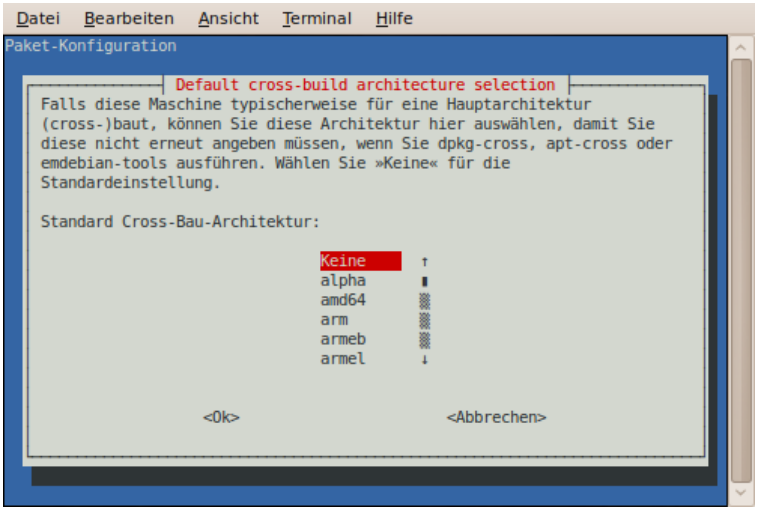
```
apt-get install apt-cross dpkg-cross
```

Nach der Installation stellen die Post-Install Skripte der Embedded-Debian Pakete noch einige grundlegende Fragen an den Benutzer:

Zunächst ist es möglich, sich eine "Standard"-Architektur für die Cross-Kompilierung auszuwählen. Wird vornehmlich, oder gar ausschließlich für eine einzige Zielarchitektur gearbeitet, dann bietet es sich an, hier bereits die Vorauswahl zu treffen. Diese Auswahl stellt keine Einschränkung dar, jedoch ist es bei den Befehlen "dpkg-cross", "apt-cross" oder aus den "emdebian-tools" dann nicht mehr nötig, wie die Standard-Architektur, diese bei jedem Aufruf explizit anzugeben.

Da wir jedoch im weiteren öfter zwischen den Architekturen wechseln werden, bleiben wir, wie in der Abbildung auf der nächsten Seite bei der Vorauswahl "Keine", so dass bei jedem Aufruf der zuvor genannten Programme, stets die Zielarchitektur explizit angegeben werden muss.

Abbildung 8:



Wie für die "normale" Debian-Distribution auch, gibt es die Möglichkeit, Paketabhängigkeiten über das altbewährte "apt-get" zu installieren, oder über das etwas modernere "aptitude". Da "apt-get" hier die bewährte und besser getestete Wahl ist, sollte man auch die Standard-Einstellung, wie in der Abbildung auf der nächsten Seite dargestellt, einfach übernehmen. Zumindest hier bietet "aptitude" keine nennenswerten Verbesserungen, die es rechtfertigen würden, sich hierfür zu entscheiden:

Davon, auch gleich Patches oder Aktualisierungen bei Embedded Debian einzureichen ist man noch etwas entfernt. Sollte man jedoch bereits einen Benutzernamen für das Subversion-Repository von Embedded Debian besitzen, kann man diesen natürlich bereits hier angeben. Ansonsten sollte das Feld einfach, wie auf der nächsten Seite gezeigt, leer lassen werden.

Sollte es nötig werden, seine eigene sog. Toolchain selbst erstellen zu müssen, beispielsweise, weil es für die geplante Zielarchitektur, keine fertige Toolchain zum Installieren via "apt-get" oder Download direkt von Embedded Debian gibt, so sollte man hier nun ein entsprechendes Verzeichnis angeben, in das diese erzeugt werden kann. Die Abbildung auf Seite 42 veranschaulicht die entsprechende Eingabemaske.

Allerdings bietet Embedded Debian für sehr viele, und vor allem alle gebräuchlichen Architekturen bereits vorgefertigte Toolchains an, so dass diese Möglichkeit wohl nur sehr selten zum Einsatz kommen wird.

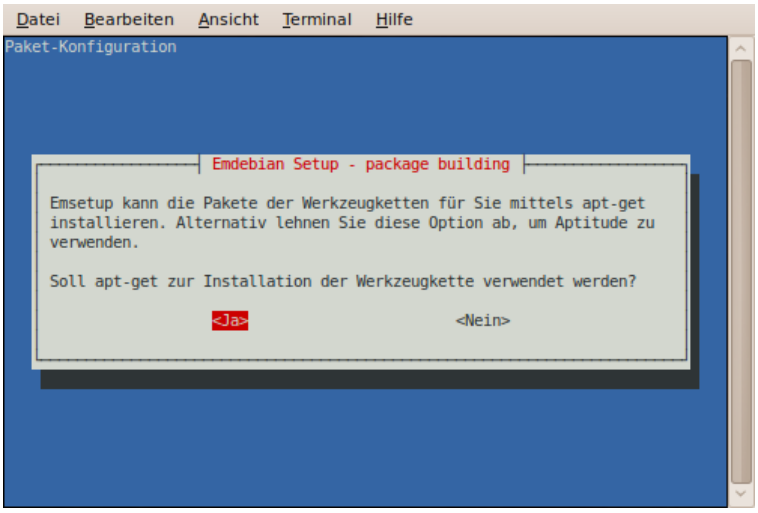


Abbildung 9:

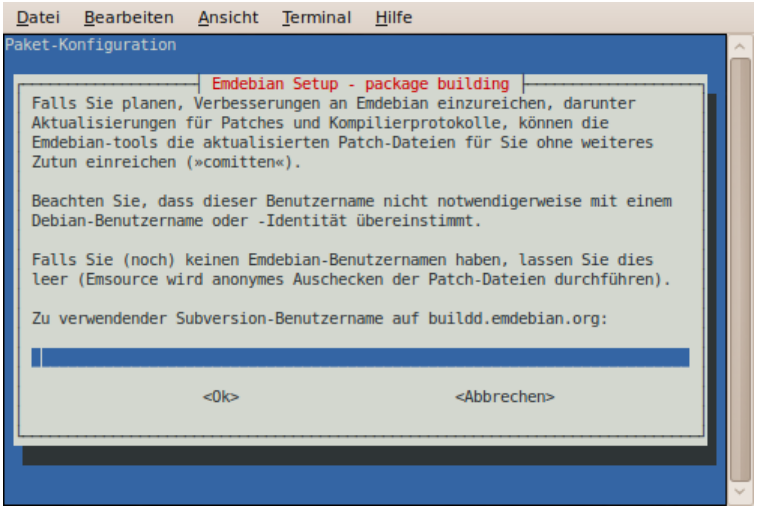
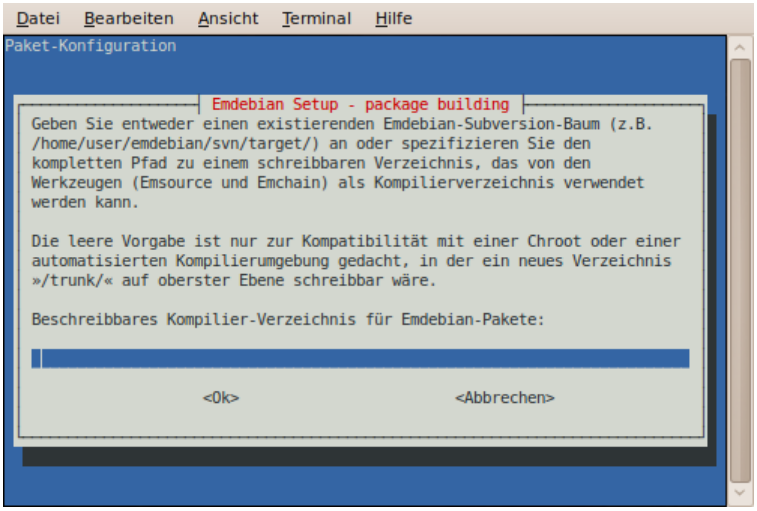


Abbildung 10:

Abbildung 11:



Da wir auf dem Desktop-Debian System die Distribution “squeeze” also “testing” einsetzen, bietet es sich an, dies auch für das Erzeugen der Cross-Pakete zu verwenden. Nicht nur, da es dem besseren Überblick dient, sondern auch, um möglichen Unstimmigkeiten zwischen Host- und Ziel-Umgebung schon möglichst frühzeitig aus dem Weg zu gehen. Dennoch ist es durchaus möglich auch cross-compilierte Pakete und Software für andere Distributionen wie beispielsweise “lenny”, also “stable” zu erzeugen wie auf der nächsten Seite dargestellt.

Zu guter Letzt ist es noch nötig, den “emdebian-tools” die Adresse eines sogenannten primären Spiegelservers mitzuteilen. Ein primärer Spiegelserver ist ein Server, der alle von Debian unterstützten Architekturen vorhält und zur Verfügung stellt, und nicht nur, zum Beispiel aus Speicherplatzgründen, nur eine Auswahl bereitstellt. Glücklicherweise bietet die Installationsroutine der “emdebian-tools” hier gleich eine Auswahl der öffentlich verfügbaren primären Spiegelserver an. Eine Auswahl an angebotenen Spiegelservern findet sich auf der nächsten Seite.

Diese Einstellungen müssen bei der Installation der “emdebian-tools” direkt vorgenommen werden. Sollte man hieran später etwas ändern wollen, weil man sich beispielsweise doch hauptsächlich mit der Portierung auf eine einzige Architektur beschäftigt, kann man dies jederzeit über

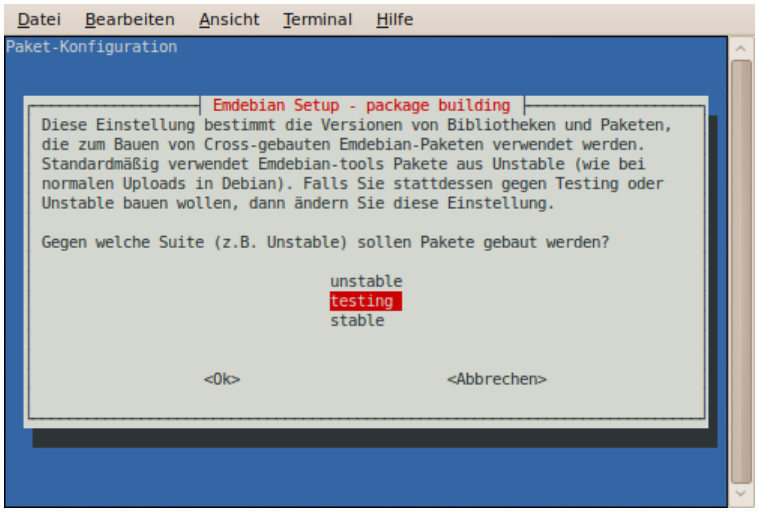


Abbildung 12:

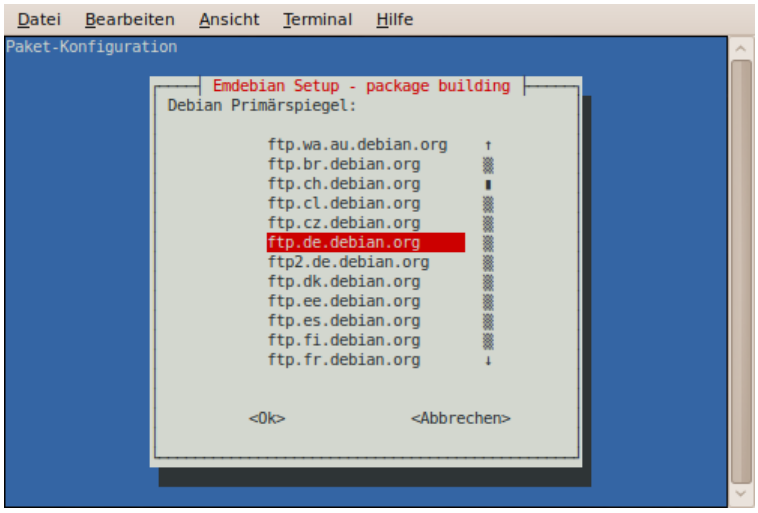


Abbildung 13:

```
dpkg-reconfigure emdebian-tools
```

oder auch über das mitgelieferte Programm "emsetup" tun.

Im Zuge der Installation von "emdebian-tools" wurde auch automatisch der Pfad zum Embedded Debian Paketverzeichnis mit in das System aufgenommen:

```
root@excelsior:~# cat /etc/apt/sources.list.d/emdebian.sources.list
 # The Emdebian toolchain repository
deb http://www.emdebian.org/debian/ testing main
deb-src http://www.emdebian.org/debian/ testing main
```

Somit stehen dem System nun auch die direkt von Embedded Debian bereitgestellten Crosscompiler und Toolchains zur Verfügung.

6.2 Crosscompiler und Toolchain

Ab diesem Zeitpunkt, sind alle notwendigen Vorarbeiten abgeschlossen, die Umgebung ist eingerichtet, und die notwendigen Crosscompiler und Toolchains aus dem Embedded Debian Projekt stehen den normalen Paketverwaltungswerkzeugen zur Verfügung. Dies lässt sich leicht mittels "apt-cache" überprüfen:

```
root@excelsior:~# apt-cache search powerpc cross
binutils-powerpc-linux-gnu - The GNU binary utilities, for powerpc-linux-gnu target
g++-4.1-powerpc-linux-gnu - The GNU C++ compiler
g++-4.3-powerpc-linux-gnu - The GNU C++ compiler
gcc-4.1-powerpc-linux-gnu - The GNU C compiler
gcc-4.3-base-powerpc-cross - The GNU Compiler Collection (base package)
(for cross-compiling)
gcc-4.3-powerpc-linux-gnu - The GNU C compiler
lib64gcc1-powerpc-cross - GCC support library (64bit)
lib64stdc++6-powerpc-cross - The GNU Standard C++ Library v3 (64bit)
libatomic-ops-dev-powerpc-cross - A library for atomic operations (development files)
(for cross-compiling)
libc6-dev-powerpc-cross - GNU C Library: Development Libraries and Header Files
(for cross-compiling)
libc6-dev-ppc64-powerpc-cross - GNU C Library: 64bit Development Libraries for
PowerPC64 (for cross-compiling)
libc6-powerpc-cross - GNU C Library: Shared libraries (for cross-compiling)
libc6-ppc64-powerpc-cross - GNU C Library: 64bit Shared libraries for PowerPC64
(for cross-compiling)
libdb1-compat-powerpc-cross - The Berkeley database routines [glibc 2.0/2.1
compatibility] (for cross-compiling)
libgcc1-powerpc-cross - GCC support library (for cross-compiling)
libstdc++5-3.3-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++5-3.3-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
libstdc++5-3.3-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++5-powerpc-cross - The GNU Standard C++ Library v3
libstdc++6-4.0-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++6-4.0-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
libstdc++6-4.0-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++6-4.1-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++6-4.1-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
libstdc++6-4.1-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++6-4.3-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++6-4.3-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
libstdc++6-4.3-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++6-4.4-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++6-4.4-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
```

```
libstdc++6-4.4-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++6-dbg-powerpc-cross - The GNU Standard C++ Library v3 (debugging files)
libstdc++6-dev-powerpc-cross - The GNU Standard C++ Library v3 (development files)
libstdc++6-pic-powerpc-cross - The GNU Standard C++ Library v3 (shared library
subset kit)
libstdc++6-powerpc-cross - The GNU Standard C++ Library v3
linux-kernel-headers-powerpc-cross - Linux Kernel Headers for development
(for cross-compiling)
linux-libc-dev-powerpc-cross - Linux support headers for userspace development
(for cross-compiling)
```

Äquivalent zum oben benutzten Beispiel anhand der PowerPC-Architektur, lässt sich dies natürlich auch für die anderen unterstützten Architekturen durchführen.

Allerdings ist es etwas mühselig, sich hier alle nötigen und zueinander passenden Pakete her auszusuchen, und zu schnell vergisst man etwas, oder übersieht ein Paket.

Auch hierfür bietet "emsetup" seine Hilfe an. Es kann vollautomatisch anhand der angegebenen Architektur überprüfen ob alles notwendige installiert ist, oder, sofern gewünscht, auch die Installation der benötigten Pakete direkt vornehmen:

```
root@excelsior:~# emsetup --arch powerpc -s
Checking for suitable apt sources.
Updating main system apt cache (enter your sudo password if prompted).
OK    http://www.emdebian.org testing Release.gpg
Ign http://www.emdebian.org testing/main Translation-de
OK    http://www.emdebian.org testing Release
OK    http://ftp.de.debian.org squeeze Release.gpg
OK    http://ftp.de.debian.org squeeze/main Translation-de
Ign http://ftp.de.debian.org squeeze/contrib Translation-de
Ign http://www.emdebian.org testing/main Packages/DiffIndex
Ign http://ftp.de.debian.org squeeze/non-free Translation-de
Ign http://www.emdebian.org testing/main Sources/DiffIndex
OK    http://ftp.de.debian.org squeeze Release
Ign http://www.emdebian.org testing/main Packages
Ign http://www.emdebian.org testing/main Sources
OK    http://ftp.de.debian.org squeeze/main Packages/DiffIndex
Ign http://www.emdebian.org testing/main Packages
Ign http://www.emdebian.org testing/main Sources
OK    http://ftp.de.debian.org squeeze/contrib Packages/DiffIndex
OK    http://ftp.de.debian.org squeeze/non-free Packages/DiffIndex
OK    http://ftp.de.debian.org squeeze/main Sources/DiffIndex
OK    http://ftp.de.debian.org squeeze/contrib Sources/DiffIndex
OK    http://ftp.de.debian.org squeeze/non-free Sources/DiffIndex
OK    http://www.emdebian.org testing/main Packages
OK    http://www.emdebian.org testing/main Sources
  Paketlisten werden gelesen... Fertig
Checking apt cache data is up to date ...
Dry run only.
An emdebian toolchain is available to build 'powerpc' on 'i386'.
Packages required for 'powerpc' on 'i386':
binutils-powerpc-linux-gnu gcc-0-powerpc-linux-gnu-base
gcc-0-powerpc-linux-gnu cpp-0-powerpc-linux-gnu g++-0-powerpc-linux-gnu
libc6-powerpc-cross libc6-dev-powerpc-cross libstdc++6-powerpc-cross
libstdc++6-0-dev-powerpc-cross libstdc++6-0-pic-powerpc-cross
libgcc1-powerpc-cross linux-libc-dev-powerpc-cross
Once a suitable toolchain can be installed, setup will be complete.
```

In der "testing"-Version hatte "emsetup" allerdings noch leichte Probleme, mit nicht aufgelösten Paketabhängigkeiten. Dies kann durchaus vorkommen, wenn in der Debian-Distribution "testing", die ja immer noch ständigen Änderungen unterworfen ist, eine Änderung vorgenommen wird, aber die externen Entwickler, wie zum Beispiel Embedded Debian noch nicht dazu gekommen sind, diese Anpassung auch in ihrem Repository vorzunehmen.

Aber der manuelle Weg über "apt-get" funktioniert auch hervorragend, wenn man sich zuvor etwas Gedanken darüber macht, was man alles an Paketen benötigt:

```
root@excelsior:~# apt-get install binutils-powerpc-linux-gnu g++-4.3-powerpc-linux-gnu \
gcc-4.3-base-powerpc-cross libc6-dev-powerpc-cross libstdc++6-4.3-dev-powerpc-cross \
linux-kernel-headers-powerpc-cross
Paketlisten werden gelesen... Fertig
Abhängigkeitsbaum wird aufgebaut
Status-Informationen einlesen... Fertig
Die folgenden zusätzlichen Pakete werden installiert:
  cpp-4.3-powerpc-linux-gnu gcc-4.3-powerpc-linux-gnu
  gcc-4.3-powerpc-linux-gnu-base libc6-powerpc-cross libgcc1-powerpc-cross
  libstdc++6-powerpc-cross
Vorgeschlagene Pakete:
  binutils-doc gcc-4.3-locales g++-4.3-multilib-powerpc-linux-gnu gcc-4.3-doc
  libstdc++6-4.3-dbg-powerpc-cross gcc-4.3-multilib-powerpc-linux-gnu
  libmudflap0-4.3-dev-powerpc-cross libgcc1-dbg-powerpc-cross
  libgomp1-dbg-powerpc-cross libmudflap0-dbg-powerpc-cross
Die folgenden NEUEN Pakete werden installiert:
  binutils-powerpc-linux-gnu cpp-4.3-powerpc-linux-gnu
  g++-4.3-powerpc-linux-gnu gcc-4.3-base-powerpc-cross
  gcc-4.3-powerpc-linux-gnu gcc-4.3-powerpc-linux-gnu-base
  libc6-dev-powerpc-cross libc6-powerpc-cross libgcc1-powerpc-cross
  libstdc++6-4.3-dev-powerpc-cross libstdc++6-powerpc-cross
  linux-kernel-headers-powerpc-cross
0 aktualisiert, 12 neu installiert, 0 zu entfernen und 0 nicht aktualisiert.
Es müssen 20,0MB an Archiven heruntergeladen werden.
Nach dieser Operation werden 39,1MB Plattenplatz zusätzlich benutzt.
Möchten Sie fortfahren [J/n]?
```

Nachdem die Installation der ausgewählten Pakete abgeschlossen ist, steht die ausgewählte Toolchain, hier "powerpc" zur Verfügung, und erste Programme können crosscompiliert werden.

6.3 Test der Crosscompiler-Umgebung

Um die prinzipielle Funktion einer Crosscompiler-Umgebung zu testen, bietet es sich an, ein kleines C[39]-Programm zu schreiben, das nur eine rudimentäre Funktion erfüllt.

Das folgende Beispiel ist stark an die allgemein bekannten "Hello world!"-Programme angelehnt:

```
#include <stdio.h>

int main() {
  printf("Hello cross-compiling world!\n");
  return 0;
}
```

Bevor nun das Programm mit Hilfe des Crosscompilers übersetzt wird, ist es sinnvoll es zunächst noch einmal mit dem nativen Host-Compiler zu übersetzen, um den Unterschied deutlich zu erkennen:

[39] C ist eine imperative Programmiersprache, die der Informatiker Dennis Ritchie in den frühen 1970er Jahren an den Bell Laboratories für das Betriebssystem Unix entwickelte. Seitdem ist sie auf vielen Computersystemen verbreitet.

```
root@excelsior:~# gcc -static hello.c -o hello
```

Um nähere Informationen über eine Datei zu erhalten, hat sich das kleine Programm "file" bewährt. Es hat die Möglichkeit in eine Datei hineinzuschauen, und sehr viele Informationen über Format und Zusammensetzung dieser Datei darzustellen:

```
root@excelsior:~# file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked,
for GNU/Linux 2.6.18, not stripped
```

Wie man aus den ausführlichen Informationen erkennen kann, handelt es sich bei der Datei um ein 32Bit ELF-Binary für die Intel-Plattform (i386). Zusätzlich erkennt man, dass das erzeugte Programm statisch gelinkt wurde, und nicht nicht "gestrippt", also von unnötigen Symbolen befreit wurde.

Als letzter kleiner Test, dass das Programm auch korrekt geschrieben und gelinkt wurde, kann man es auch noch ausführen:

```
root@excelsior:~# ./hello
Hello cross-compiling world!
```

Das Programm tut also, was es sollte, nämlich den angegebenen Text einfach auf der Standard-Ausgabe ausgeben. Natürlich ist die textuelle Aussage noch nicht ganz korrekt, denn noch wurde das Programm ja nicht mit einem Cross-Compiler übersetzt. Das werden wir nun aber nachholen.

Übersetzt man nun das Prögrämmchen mit Hilfe des Crosscompilers für "powerpc" so ändert sich der Aufruf zumindest hier in diesem kleinen Beispiel nur minimal:

```
root@excelsior:~# powerpc-linux-gnu-gcc -static hello.c -o hello
```

Schaut man sich nun wiederum die Ausgabe des Programms "file" an, so stellt mehr sehr schnell einen gewaltigen Unterschied fest:

```
root@excelsior:~# file hello
hello: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), stati
cally linked, for GNU/Linux 2.6.8, with unknown capability 0x41000000 = 0xf676e7
5, with unknown capability 0x10000 = 0x70401, not stripped
```

Wie man hier nun erkennen kann, handelt es sich zwar immer noch um ein 32Bit ELF-Binary, allerdings nun nicht mehr für die Intel-Plattform, sondern eben für die PowerPC-Plattform. Die übrigen Informationen sind gleich geblieben.

Würde man nun aber versuchen den kleinen Test von vorhin nochmals durchzuführen, würde dies allerdings nicht zum Erfolg führen. Denn obwohl das Programm statisch gelinkt ist, so dass keine zusätzlichen dynamisch zu ladenden Bibliotheken benötigt werden, lässt sich das Programm nicht ausführen, denn es wurde für eine ganz andere Prozessorarchitektur kompiliert.

```
root@excelsior:~# ./hello
bash: ./hello: Kann die Datei nicht ausführen.
```

Um nun das soeben erzeugte Programm für die PowerPC Architektur auszuführen, kann man auf die Prozessoremulation von QEMU zurückgreifen, sofern man nicht das Zielsystem direkt zur Hand hat:

```
root@excelsior:~# qemu-ppc ./hello
Hello cross-compiling world!
```

6.4 Bibliotheken für Crosscompiler

Eine der größten und vor allem auch zeitraubensten Herausforderungen, die es beim Cross-compilieren zu lösen gilt, ist es, die für die Programme notwendigen Bibliotheken in die Crosscompiler Toolchain entsprechend zu integrieren und zur Verfügung zu stellen.

Im Regelfall wird dies versucht, indem man den Quellcode der Bibliotheken von der Homepage des zugehörigen Projektes herunterlädt, die Pfade der Installation anpasst, und diese ebenfalls mit dem Crosscompiler übersetzt.

Hierbei ist es äußerst wichtig, dass die erzeugten Cross-Bibliotheken, den Bibliotheken der Zielplattform äußerst genau entsprechen. Hierzu ist es notwendig, nicht nur die exakt selbe Version der Bibliotheken zu verwenden, sondern auch dass die notwendigen Compileoptionen übereinstimmen. Jedoch natürlich angepasst an die Crosscompiler-Umgebung.

Dies kann eine sehr zeitaufwendige und auch leicht fehlerträchtige Aufgabe sein, da schon leichte Abweichungen oder Versäumnisse in den eingestellten Optionen zu einem Programm führen können, dass nicht mehr ohne Probleme auf der Zielplattform funktioniert.

6.4.1 dpkg-cross

Um das Problem der genauen Übereinstimmung der Bibliotheken zwischen der Crosscompiler-Umgebung und der Zielplattform zu lösen, existiert bei Embedded Debian eine so einfach, wie an sich geniale Lösung.

Im Prinzip existieren alle Bibliotheken bereits für Debian, in Form von Debian-Paketen für die jeweilige Zielarchitektur. Allerdings sind in den Paketen natürlich gänzlich andere Pfade hinterlegt. So befinden sich die Bibliotheken im "normalen" System unter "/usr/lib", ebenso auch auf der Zielplattform, und damit auch in den Paketen, die von Debian bereit gestellt werden. Für den Crosscompiler jedoch, müssen sich die Bibliotheken in einem Pfad unter "/usr/<arch>/lib" befinden.

Um hier eine Umwandlung vorzunehmen, existiert in Embedded Debian, das Programm "dpkg-cross", welches genau diese Umwandlung selbst vornimmt, und ein neues Cross-Paket erzeugt.

Im Folgenden betrachten wir diese Umwandlung anhand der Bibliothek "libv4l".

Zunächst ist es notwendig, das Paket für die Zielarchitektur von Debian in der gewünschten Version herunterzuladen. Den Pfad zum Herunterladen findet man am besten über die Webseite "packages.debian.org"[40].

Um auf der Kommandozeile etwas herunterzuladen bietet sich hier natürlich direkt das Programm "wget" an, das an sich jeder modernen Distribution beiliegen sollte:

```
wget http://ftp.de.debian.org/debian/pool/main/libv/libv4l/libv4l-0_0.6.4-1_\
powerpc.deb
```

Nun befindet sich das original PowerPC-Paket von Debian auf der Festplatte, und zwar exakt so, wie es auch für das Zielsystem existiert.

Dieses lässt sich nun mit Hilfe des Programms "dpkg-cross" in ein Crosscompiler-Paket umwandeln.

Es ist, zumindest bei unseren Voreinstellungen von oben, dass wir keine bevorzugte Zielarchitektur haben, notwendig die Zielarchitektur explizit anzugeben. Dies geschieht über den Parameter "-a" oder "–arch", wie bei fast allen Werkzeugen aus dem Embedded Debian Projekt.

Über den Schalter "-b" oder "–build" wird "dpkg-cross" angewiesen, nur ein Crosscompiler-Paket zu erzeugen, dieses aber nicht sofort zu installieren. Dies wäre, falls gewünscht, über den Schalter "-i" oder "–install" anstelle von "-b" oder "–build" möglich.

```
root@excelsior:~# dpkg-cross -a powerpc -b libv4l-0_0.6.4-1_powerpc.deb
Building libv4l-0-powerpc-cross_0.6.4-1_all.deb
dpkg-deb: Baue Paket »libv4l-0-powerpc-cross« in »./libv4l-0-powerpc-cross_0.6.4
-1_all.deb«.
```

Nun wurde mittels "dpkg-cross" das Debian-Paket für die Architektur "PowerPC" in ein für alle Plattformen ("all") gültiges Crosscompiler-Paket umgewandelt.

Die Änderungen im Paket werden am deutlichsten sichtbar, wenn man sich einmal kurz den Inhalt der beiden Pakete anschaut, und vergleicht. Dies geht ohne die Pakete zu installieren am einfachsten mit dem Debian-Paket-Werkzeug "dpkg", das mittels des Parameters "–contents" den Inhalt von Debian-Pakete auflisten kann.

[40] http://packages.debian.org

```
root@excelsior:~# dpkg --contents libv4l-0_0.6.4-1_powerpc.deb
drwxr-xr-x root/root         0 2010-01-19 21:57 ./
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/share/
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/share/doc/
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/share/doc/libv4l-0/
-rw-r--r-- root/root      7680 2010-01-19 21:56 ./usr/share/doc/libv4l-0/copyright
-rw-r--r-- root/root       520 2009-11-17 09:42 ./usr/share/doc/libv4l-0/README.
multi-threading
-rw-r--r-- root/root       629 2009-11-17 09:42 ./usr/share/doc/libv4l-0/TODO
-rw-r--r-- root/root      3165 2009-11-17 09:42 ./usr/share/doc/libv4l-0/README.
gz
-rw-r--r-- root/root      1760 2010-01-19 21:56 ./usr/share/doc/libv4l-0/changelog.
Debian.gz
-rw-r--r-- root/root      8392 2010-01-10 07:41 ./usr/share/doc/libv4l-0/changelog.
gz
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/share/lintian/
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/share/lintian/overrides/
-rw-r--r-- root/root        79 2010-01-19 21:56 ./usr/share/lintian/overrides/lib
v4l-0
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/lib/
-rw-r--r-- root/root     19784 2010-01-19 21:57 ./usr/lib/libv4l1.so.0
drwxr-xr-x root/root         0 2010-01-19 21:57 ./usr/lib/libv4l/
-rwxr-xr-x root/root     21040 2010-01-19 21:57 ./usr/lib/libv4l/ov518-decomp
-rwxr-xr-x root/root     16816 2010-01-19 21:57 ./usr/lib/libv4l/ov511-decomp
-rw-r--r-- root/root      5132 2010-01-19 21:57 ./usr/lib/libv4l/v4l2convert.so
-rw-r--r-- root/root      4624 2010-01-19 21:57 ./usr/lib/libv4l/v4l1compat.so
-rw-r--r-- root/root    122296 2010-01-19 21:57 ./usr/lib/libv4lconvert.so.0
-rw-r--r-- root/root     41232 2010-01-19 21:57 ./usr/lib/libv4l2.so.0
```

```
root@excelsior:~# dpkg --contents libv4l-0-powerpc-cross_0.6.4-1_all.deb
drwxr-xr-x root/root         0 2010-03-31 20:30 ./
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/share/
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/share/doc/
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/share/doc/libv4l-0-powerpc-
cross/
-rw-r--r-- root/root       269 2010-03-31 20:30 ./usr/share/doc/libv4l-0-powerpc-
cross/README
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/powerpc-linux-gnu/
drwxr-xr-x root/root         0 2010-03-31 20:30 ./usr/powerpc-linux-gnu/lib/
-rw-r--r-- root/root     19784 2010-01-19 21:57 ./usr/powerpc-linux-gnu/lib/libv4l1
.so.0
-rw-r--r-- root/root     41232 2010-01-19 21:57 ./usr/powerpc-linux-gnu/lib/libv4l2
.so.0
-rw-r--r-- root/root    122296 2010-01-19 21:57 ./usr/powerpc-linux-gnu/lib/libv4l
convert.so.0
```

Wie man sehr gut erkennen kann, wurde die Bibliothek in das Installationsverzeichnis für den passenden Crosscompiler verschoben. Zusätzlich wurde jegliche Dokumentation entfernt, um etwas Speicherplatz zu sparen.

Nun ist es nur noch notwendig, das soeben erzeugte Paket im System zu installieren. Dies geschieht ganz nach Debian-Art mit dem Werkzeug “dpkg”, mit dem auch alle sonstigen Pakete verwaltet werden.

```
root@excelsior:~# dpkg -i libv4l-0-powerpc-cross_0.6.4-1_all.deb
Wähle vormals abgewähltes Paket libv4l-0-powerpc-cross.
(Lese Datenbank ... 22373 Dateien und Verzeichnisse sind derzeit installiert.)
Entpacke libv4l-0-powerpc-cross (aus libv4l-0-powerpc-cross_0.6.4-1_all.deb) ...
Richte libv4l-0-powerpc-cross ein (0.6.4-1) ...
```

6.4.2 apt-cross

Auf dem Werkzeug “dpkg-cross” setzt das Werkzeug “apt-cross”, ebenfalls aus dem Embedded Debian Projekt, auf. Es automatisiert die oben genannten Schritte zur Erzeugung eines Crosscompiler-Pakets aus einem nativen Debian-Paket.

```
root@excelsior:~# apt-cross -a powerpc -i libv4l-0
The following NEW packages will be built and installed:
 libv4l-0
0 to be upgraded, 1 to be newly installed.
Building libv4l-0-powerpc-cross_0.6.4-1_all.deb
dpkg-deb: Baue Paket »libv4l-0-powerpc-cross« in »/tmp/libv4l-0-powerpc-cross_0.
6.4-1_all.deb«.
Wähle vormals abgewähltes Paket libv4l-0-powerpc-cross.
(Lese Datenbank ... 22373 Dateien und Verzeichnisse sind derzeit installiert.)
Entpacke libv4l-0-powerpc-cross (aus .../libv4l-0-powerpc-cross_0.6.4-1_all.deb)
 ...
Richte libv4l-0-powerpc-cross ein (0.6.4-1) ...
Removing temporary archives
```

6.5 Beispiel Editor "nano"

Um die genaue Funktionsweise von Embedded Debian kennen zu lernen, ist es angeraten, dies am Beispiel eines kleinen Programms zu demonstrieren. Dieses sollte nicht nur wie bereits oben ein "Hello World" Programm sein, sondern auch von externen Bibliotheken abhängen, und eine etwas größere Funktionalität aufweisen. Für dieses Beispiel bietet sich der kleine Editor "nano"[41] an. "nano" ist ein Textkonsolen-Editor, der beinahe jeder aktuellen Linux-Distribution beiliegt.

Der Texteditor "nano" benötigt einige Bibliotheken, um die Darstellung auf der Textkonsole vorzunehmen. Hierbei ist besonders die "ncurses"-Bibliothek zu erwähnen.

Um den Editor "nano" crosscompilieren zu können, ist es zunächst notwendig, den aktuellen Quellcode des Debian-Pakets herunterzuladen und auszupacken. Dankenswerterweise stellt Debian hier mit dem Werkzeug "apt-get" eine komfortable Möglichkeit bereit, dies zu bewerkstelligen:

```
root@excelsior:~/nano# apt-get source nano
Paketlisten werden gelesen... Fertig
Abhängigkeitsbaum wird aufgebaut
Status-Informationen einlesen... Fertig
Es müssen 1.558kB an Quellarchiven heruntergeladen werden.
Hole:1 http://ftp.de.debian.org squeeze/main nano 2.2.3-1 (dsc) [1.261B]
Hole:2 http://ftp.de.debian.org squeeze/main nano 2.2.3-1 (tar) [1.528kB]
Hole:3 http://ftp.de.debian.org squeeze/main nano 2.2.3-1 (diff) [29,2kB]
Es wurden 1.558kB in 1s geholt (1.116kB/s)
gpgv: Unterschrift vom Fr 12 Feb 2010 02:18:52 UTC mittels DSA-Schlüssel ID 917A225E
gpgv: Unterschrift kann nicht geprüft werden: Öffentlicher Schlüssel nicht gefunden
dpkg-source: Warnung: Fehler beim Überprüfen der Signatur von ./nano_2.2.3-1.dsc
dpkg-source: Information: extrahiere nano nach nano-2.2.3
dpkg-source: Information: entpacke nano_2.2.3.orig.tar.gz
dpkg-source: Information: entpacke nano_2.2.3-1.debian.tar.gz
```

Nun befindet sich der Debian Quellcode des Editors "nano" im aktuellen Verzeichnis. Sofern "dpkg-source" installiert ist, wird der Quellcode auch gleich im gleichnamigen Verzeichnis ausgepackt, und steht direkt zur Verfügung.

[41] Nano ist ein freier Texteditor für Linux- und UNIX-Systeme. Er ist im Gegensatz zu anderen Editoren für die Shell, wie vi, einfach zu bedienen, wodurch er bei Anfängern beliebt ist.

```
root@excelsior:~/nano# ls -a -l
insgesamt 1544 drwxr-xr-x 3 root root    4096  9. Apr 12:47 .
drwx------ 8 root root     4096  9. Apr 12:47 ..
drwxr-xr-x 7 root root     4096  9. Apr 12:47 nano-2.2.3
-rw-r--r-- 1 root root    29175 12. Feb 03:42 nano_2.2.3-1.debian.tar.gz
-rw-r--r-- 1 root root     1261 12. Feb 03:42 nano_2.2.3-1.dsc
-rw-r--r-- 1 root root  1528017 12. Feb 03:42 nano_2.2.3.orig.tar.gz
```

Da es für "dpkg" oder "apt" in einer Crosscompiler-Umgebung nicht möglich ist, die Paketabhängigkeiten vollautomatisch aufzulösen, ist es zunächst erforderlich, sich manuell die für die Erzeugung notwendigen Abhängigkeiten anzuschauen, und händisch aufzulösen. Die Paketabhängigkeiten stehen in der Datei "debian/control" unter der Bezeichnung "Build-Depends". Diese müssen entweder modifiziert, oder eben für die Crosscompiler-Umgebung aufgelöst werden.

```
Source: nano Section: editors
Priority: optional
Maintainer: Jordi Mallach <jordi@debian.org>
Uploaders: Steve Langasek <vorlon@debian.org>
Build-Depends: dpkg-dev (>= 1.13.9), debhelper (>= 7), libncurses5-dev,
libncursesw5-dev, libslang2-dev
Standards-Version: 3.8.3
Vcs-Svn: svn://svn.debian.org/svn/collab-maint/deb-maint/nano
Vcs-Browser: http://svn.debian.org/viewsvn/collab-maint/deb-maint/nano/
Homepage: http://www.nano-editor.org/
Package: nano
Architecture: any
Priority: important
Depends: ${shlibs:Depends}, dpkg (>= 1.15.4) | install-info, ${misc:Depends}
Provides: editor
Suggests: spell
Conflicts: nano-tiny (<= 1.0.0-1), pico, alpine-pico (<= 2.00+dfsg-5)
Replaces: pico
Description: small, friendly text editor inspired by Pico
 GNU nano is an easy-to-use text editor originally designed as a replacement
 for Pico, the ncurses-based editor from the non-free mailer package Pine
 (itself now available under the Apache License as Alpine).
 .
 However, nano also implements many features missing in pico, including:
  - feature toggles;
  - interactive search and replace (with regular expression support);
  - go to line (and column) command;
  - auto-indentation and color syntax-highlighting;
  - filename tab-completion and support for multiple buffers;
  - full internationalization support.

Package: nano-tiny
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Provides: editor
Description: small, friendly text editor inspired by Pico - tiny build
 GNU nano is an easy-to-use text editor originally designed as a replacement
 for Pico, the ncurses-based editor from the non-free mailer package Pine
 (itself now available under the Apache License as Alpine).
 .
 This package contains a build of GNU nano with many features disabled, for
 environments such as rescue disks where resources are limited.
```

Zunächst ist es also notwendig, die "ncurses"[42]-Bibliothek für die passende Architektur, hier PowerPC, zu erzeugen. Um ein Debian-Paket so anzupassen, dass es für Embedded Debian geeignet ist, und auch evtl. nötige Anpassungen vorzunehmen, damit es möglich ist, das Paket mittels eines Crosscompilers zu erzeugen, stellt Embedded Debian das Werkzeug "em_make"

[42] ncurses (Abk. für new curses) ist eine freie C-Programmbibliothek (engl. library), um zeichenorientierte Benutzerschnittstellen (Text User Interface, TUI) unabhängig vom darstellenden Textterminal bzw. Terminalemulator darzustellen.

bereit. Mit ihm kann man in sehr vielen Fällen die notwendigen Anpassungen vollautomatisch vornehmen:

```
root@excelsior:~/nano/ncurses-5.7+20100313# em_make -a powerpc -V emdebian-grip
Checking toolchains...
Setting emdebian version: 5.7+20100313-1em1 for 'Emdebian Grip'.
```

Nun wurden die im Debian-Paket hinterlegten Build-Prozesse auf die entsprechende Cross-Kompilierung angepasst, und das Paket kann mittels eines weiteren Programmaufrufs erzeugt werden. Aus der Ausgabe des Buildprozesses ist schon zu beginn ersichtlich, dass die notwendigen Anpassungen vorgenommen wurden. Hier ist nochmals ein letzter kontrollierender Blick möglich, ob wirklich ein Paket für die gewünschte Architektur erzeugt wird:

```
root@excelsior:~/nano/ncurses-5.7+20100313# emdebuild -a powerpc -V emdebian-gpip
Building 'ncurses' for powerpc on i386.
dpkg-architecture: Warnung: Angegebener GNU-Systemtyp powerpc-linux-gnu passt ni
cht auf gcc-Systemtyp i486-linux-gnu.
DEB_BUILD_ARCH=i386
DEB_BUILD_ARCH_OS=linux
DEB_BUILD_ARCH_CPU=i386
DEB_BUILD_ARCH_BITS=32
DEB_BUILD_ARCH_ENDIAN=little
DEB_BUILD_GNU_CPU=i486
DEB_BUILD_GNU_SYSTEM=linux-gnu
DEB_BUILD_GNU_TYPE=i486-linux-gnu
DEB_HOST_ARCH=powerpc
DEB_HOST_ARCH_OS=linux
DEB_HOST_ARCH_CPU=powerpc
DEB_HOST_ARCH_BITS=32
DEB_HOST_ARCH_ENDIAN=big
DEB_HOST_GNU_CPU=powerpc
DEB_HOST_GNU_SYSTEM=linux-gnu
DEB_HOST_GNU_TYPE=powerpc-linux-gnu
...
```

Die hierbei erzeugten Debian-Pakete lassen sich wie bereit zuvor beschrieben, wieder mittels "dpkg-cross" in Cross-Compiler-Pakete umwandeln, so dass sich diese in die Cross-Compiler-Umgebung integrieren. Wurden diese Cross-Compiler-Pakete installiert, lässt sich dich noch einmal mittels "dpkg -l" kontrollieren:

```
root@excelsior:~/nano# dpkg -l | grep curses | grep cross
ii  libncurses5-powerpc-cross                5.7+20100313-1
        shared libraries for terminal handling (for
ii  libncursesw5-dev-powerpc-cross           5.7+20100313-1
        developer's libraries for ncursesw (for cros
ii  libncursesw5-powerpc-cross               5.7+20100313-1
        shared libraries for terminal handling (wide
```

Da nun alle Abhängigkeiten von Bibliotheken für den Editor "nano" erfüllt sind, ist es nun möglich sich dem Editor als solches zuzuwenden. Die vorbereitenden Arbeiten an notwendigen Bibliotheken sind nun abgeschlossen.

Die Vorgehensweise ist der bei den Bibliotheken sehr ähnlich. Zunächst wird der Quellcode des Debian-Pakets, welches bereits im Vorfeld zur Ermittlung der notwendigen Abhängigkeiten heruntergeladen wurde, mittels "em_make" auf die Cross-Compiler-Umgebung angepasst:

```
root@excelsior:~/nano/nano-2.2.4# em_make -a powerpc -V emdebian-grip
Checking toolchains...
Setting emdebian version: 2.2.4-1em1 for 'Emdebian Grip'.
```

Im Anschluss kann der Editor ebenso wie zuvor die Bibliothek "ncurses" für die Architektur "powerpc" erzeugt werden.

```
Building 'nano' for powerpc on i386.
dpkg-architecture: Warnung: Angegebener GNU-Systemtyp powerpc-linux-gnu passt ni
cht auf gcc-Systemtyp i486-linux-gnu.
DEB_BUILD_ARCH=i386
DEB_BUILD_ARCH_OS=linux
DEB_BUILD_ARCH_CPU=i386
DEB_BUILD_ARCH_BITS=32
DEB_BUILD_ARCH_ENDIAN=little
DEB_BUILD_GNU_CPU=i486
DEB_BUILD_GNU_SYSTEM=linux-gnu
DEB_BUILD_GNU_TYPE=i486-linux-gnu
DEB_HOST_ARCH=powerpc
DEB_HOST_ARCH_OS=linux
DEB_HOST_ARCH_CPU=powerpc
DEB_HOST_ARCH_BITS=32
DEB_HOST_ARCH_ENDIAN=big
DEB_HOST_GNU_CPU=powerpc
DEB_HOST_GNU_SYSTEM=linux-gnu
DEB_HOST_GNU_TYPE=powerpc-linux-gnu
...
configure: loading site script /etc/dpkg-cross/cross-config.powerpc
configure: creating cache /root/nano/nano-2.2.4/config.cache
checking build system type... i486-pc-linux-gnu
checking host system type... powerpc-unknown-linux-gnu
checking target system type... powerpc-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for powerpc-linux-gnu-strip... powerpc-linux-gnu-strip
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes
checking for style of include used by make... GNU
checking for powerpc-linux-gnu-gcc... powerpc-linux-gnu-gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... yes
...
powerpc-linux-gnu-gcc -DHAVE_CONFIG_H -I. -I../../src -I.. -DLOCALEDIR=\"/usr/sh
are/locale\" -DSYSCONFDIR=\"/etc\" -I/usr/include/ncursesw -Wall -g -O2 -MT b
rowser.o -MD -MP -MF .deps/browser.Tpo -c -o browser.o ../../src/browser.c
mv -f .deps/browser.Tpo .deps/browser.Po
powerpc-linux-gnu-gcc -DHAVE_CONFIG_H -I. -I../../src -I.. -DLOCALEDIR=\"/usr/sh
are/locale\" -DSYSCONFDIR=\"/etc\" -I/usr/include/ncursesw -Wall -g -O2 -MT c
hars.o -MD -MP -MF .deps/chars.Tpo -c -o chars.o ../../src/chars.c
mv -f .deps/chars.Tpo .deps/chars.Po
powerpc-linux-gnu-gcc -DHAVE_CONFIG_H -I. -I../../src -I.. -DLOCALEDIR=\"/usr/sh
are/locale\" -DSYSCONFDIR=\"/etc\" -I/usr/include/ncursesw -Wall -g -O2 -MT c
olor.o -MD -MP -MF .deps/color.Tpo -c -o color.o ../../src/color.c
mv -f .deps/color.Tpo .deps/color.Po
...
rm -rf /root/nano/nano-2.2.4/debian/nano-udeb/usr
rm -f /root/nano/nano-2.2.4/debian/nano-udeb/bin/rnano
dh_testdir
dh_testroot
dh_installdocs -a
dh_installexamples -a
dh_installmenu -a
dh_installman -a
dh_installinfo -pnano doc/texinfo/nano.info
rm -f debian/nano/usr/share/info/dir \
                debian/nano/usr/share/info/dir.old
# We don't want info docs in -tiny
rm -r debian/nano-tiny/usr/share/info
dh_installchangelogs ChangeLog -a
dh_link -a
dh_strip -a
dh_compress -a
dh_fixperms -a
dh_installdeb -a
dh_shlibdeps -a
dh_gencontrol -a
```

```
dh_md5sums -a
dh_builddeb -a
dpkg-deb: Baue Paket »nano« in »../nano_2.2.4-1em1_powerpc.deb«.
dpkg-deb: Baue Paket »nano-tiny« in »../nano-tiny_2.2.4-1em1_powerpc.deb«.
dpkg-deb: Baue Paket »nano-udeb« in »../nano-udeb_2.2.4-1em1_powerpc.udeb«.
 dpkg-genchanges -sa >../nano_2.2.4-1em1_powerpc.changes
dpkg-genchanges: füge kompletten Quellcode beim Hochladen hinzu
dpkg-buildpackage: Alles hochzuladen (Originalquellen enthalten)
```

Hiermit ist der Build-Prozess für den Editor “nano” abgeschlossen, und es wurde ebenfalls ein Embedded Debian Paket für die angegebene Embedded Debian Distribution “grip” erstellt. Zur Sicherheit kann man sich die erzeugten Dateien und Pakete mittels des Befehls “ls -a -l” sehr anschaulich darstellen lassen.

```
root@excelsior:~/nano# ls -a -l
insgesamt 2296
drwxr-xr-x  4 root root    4096 26. Apr 09:50 .
drwx------  8 root root    4096  9. Apr 12:47 ..
-rw-r--r--  1 root root     373 26. Apr 09:50 emdebian-changelog.patch
drwxr-xr-x 10 root root    4096 26. Apr 09:50 nano-2.2.4
-rw-r--r--  1 root root   28891 15. Apr 23:02 nano_2.2.4-1.debian.tar.gz
-rw-r--r--  1 root root    1261 15. Apr 23:02 nano_2.2.4-1.dsc
-rw-r--r--  1 root root   28772 26. Apr 09:50 nano_2.2.4-1em1.debian.tar.gz
-rw-r--r--  1 root root    1027 26. Apr 09:50 nano_2.2.4-1em1.dsc
-rw-r--r--  1 root root   68099 26. Apr 09:50 nano_2.2.4-1em1_powerpc.build
-rw-r--r--  1 root root    2237 26. Apr 09:50 nano_2.2.4-1em1_powerpc.changes
-rw-r--r--  1 root root  539588 26. Apr 09:50 nano_2.2.4-1em1_powerpc.deb
-rw-r--r--  1 root root 1529085 15. Apr 23:02 nano_2.2.4.orig.tar.gz
drwxr-xr-x  3 root root    4096 26. Apr 09:50 nano.old
-rw-r--r--  1 root root   73056 26. Apr 09:50 nano-tiny_2.2.4-1em1_powerpc.deb
-rw-r--r--  1 root root   36392 26. Apr 09:50 nano-udeb_2.2.4-1em1_powerpc.udeb
```

Die Portierung des Editors “nano” auf die Embedded Debian Plattform ist nun erfolgreich abgeschlossen, und das entstandene Paket kann in eingebetteten Systemen eingesetzt werden.

6.6 Varianten von Embedded Debian

Embedded Debian existiert in 2 Varianten unter den Namen “grip” und “crush”. Beide Varianten sind für eingebettete Systeme geeignet, haben jedoch etwas andere Ziele, Voraussetzungen und Portierungsaufwand. Deshalb bietet sich eine genauere Betrachtung der beiden Varianten an.

6.6.1 Embedded Debian Grip

“grip” stellt eine Anpassung an die Debian-Distribution “Lenny” oder vor allem auch für die zukünftige Debian-Distribution “Squeeze” dar. Die von Embedded Debian selbst gesteckten Vorgaben für “grip” waren:

- Teile von “squeeze” zu verwenden
- so wenig funktionale Änderungen vorzunehmen, wie möglich

- möglichst hohe funktionale Kompatibilität zu Debian "Squeeze"

Die Embedded Debian Variante "grip" sollte zwar einerseits eine "native" Buildumgebung bereitstellen, um auch auf den Zielsystemen direkt entwickeln zu können, inklusive der Unterstützung direkt auf dem Zielsystem eigene Embedded Debian Pakete zu erzeugen, als auch die Möglichkeit für den Entwickler offen halten, Embedded Debian Pakete und native Debian Pakete mit minimalem Aufwand zu verknüpfen.

Die Möglichkeit auch Cross-kompilieren zu können, wurde hierbei ebenfalls berücksichtigt.

Diese Kombination ist bisher einzigartig in der Welt der eingebetteten Systeme. Sie vereint, den Komfort eines "normalen" Linux-Systems mit der einfachen Möglichkeit selbst Pakete zu erzeugen und auch bereits existierende Pakete aus der Distribution Debian wiederzuverwenden.

Folgende Merkmale zeichnen Embedded Debian "grip" aus:

Embedded Debian "grip" kann Debian Pakete aus dem "normalen" Debian Repository auspacken, für den Einsatz in eingebetteten Systemen nicht relevante Dateien entfernen und daraus wieder ein neues Embedded Debian Paket erzeugen. Der Grund hierfür ist, die unmodifizierten Binärprogramme und Skripte der Paketbetreuer ohne Änderungen wiederzuverwenden, aber dennoch die Größe des Paketes, der Paket-Metadaten und natürlich letztendlich auch den Platzverbrauch des installierten Paketes zu reduzieren.

Embedded Debian "grip" kann Debian-Pakete jeder beliebigen, verfügbaren Architektur verarbeiten. Dies bietet die Möglichkeit auf einem System Pakete für Embedded Debian für unterschiedlichste Ziel-Architekturen zu erzeugen. Aktuell unterstützt Embedded Debian "grip" hierbei die Architekturen: arm, armel, i386, amd64, powerpc, mips, mipsel, source. Die Unterstützung für arm und armel wurde leider aus Debian nach der Veröffentlichung der Distribution "Lenny" entfernt, so dass es sehr wahrscheinlich ist, dass für die zukünftige Debian-Distribution "Squeeze" diese beiden Architekturen im Rahmen von Embedded Debian nicht mehr zur Verfügung gestellt werden können.

Embedded Debian "grip" nutzt die aus Debian bekannten Standardmethoden zur Erstellung eines Root-Dateisystems, für sog. "chroots" oder auch ganz gewöhnliche Installationen mittels der aus dem Desktopbereich bekannten Installationsroutinen.

Embedded Debian "grip" nutzt die sogenannten "TDebs". Da die mit der Software mitgelieferte Dokumentation bei Embedded Debian im Gegensatz zu "normalem" Debian nicht Bestandteil der Programmpakete sein sollte, um Speicherplatz zu sparen, entwickelte das Embedded Debian-Projekt die sogenannten "TDebs". Dies sind Debian-Pakete, die nur die Dokumentation

enthalten. So kann der Entwickler selbst entscheiden, ob er auf seinem eingebetteten System Dokumentation vorhalten möchte, oder sich doch lieber für mehr zur Verfügung stehenden Speicherplatz entscheidet. Alle Debian-Pakete können so umgewandelt werden, dass sie in ein Programm-Paket und ein Dokumentations-Paket für Embedded Debian aufgespalten werden können.

Hier nochmal die wichtigsten Eigenschaften von Embedded Debian "grip" im Überblick:

- Unterstützung für native Verwendung oder über eine Cross-Build-Umgebung
- Unterstützung für Compiler und Build-Werkzeuge auf der Zielplattform, inklusive interpretierter Sprachen
- Keine funktionalen Änderungen in Bibliotheken. Dies bedeutet konkret, dass keine Änderungen an den Parametern für "configure" in der Datei "debian/rules" nötig sind.
- Erweiterung der zukünftigen TDEB-Unterstützung in der Distribution Debian
- Nutzung der Option "nodocs" in den Einstellungen der Build-Optionen für Pakete. Ebenso die Erweiterung des Supports für die Option "nodocs" im Rahmen von Veränderungen an "dpkg" und der Erweiterung der "DEB-VENDOR"-Unterstützung.
- Optimierung des verwendeten Speicherplatzes zur Installation von Embedded Debian "grip". Geräte die unter Embedded Debian "grip" arbeiten sollen zwar wesentlich weniger Speicherplatz benötigen als eine "normale" Debian-Installation, allerdings natürlich mehr Speicherplatz benötigen, als das im Folgenden vorgestellte Embedded Debian "crush".
- Verwendung der "coreutils" aus der "normalen" Debian Distribution, anstatt einer auf Busybox basierenden Variante, um eine größere Kompatibilität zu gewährleisten.

6.6.2 Embedded Debian Crush

Embedded Debian "crush" geht noch einen Schritt weiter, als das zuvor beschriebene Embedded Debian "grip". Es ist wesentlich stärker auf Speicherplatz-Optimierung ausgelegt, und zielt damit vor allem auf eingebettete System mit sehr wenig Ressourcen. Jedoch bleibt hierbei natürlich Einiges an Komfort und Möglichkeiten für den Entwickler auf der Strecke. So ist es beispielsweise nicht möglich auf Embedded Debian "crush" nativ Programme zu übersetzen oder Pakete hierfür zu erstellen. Embedded Debian "crush"

ist komplett auf eine entsprechende Cross-Compiler-Umgebung angewiesen. Auch gerne für kleinere Aufgaben verwendete, interpretierte Sprachen stehen nicht zur Verfügung. Diese würden nicht nur den Speicherplatzverbrauch für diese Geräte stark in die Höhe treiben, sondern wären auch auf Grund ihrer Natur nur sehr langsam.

Die wichtigsten Eigenschaften von Embedded Debian "crush" im Überblick:

- keine native Kompilierung möglich. Embedded Debian "crush" ist für Geräte gedacht, die selbst auf Grund ihrer beschränkten Ressourcen nicht in der Lage wären, Pakete zu erzeugen.
- Konsequenterweise stehen für Embedded Debian "crush" auch keine Compiler- oder Build-Werkzeuge zur Verfügung. Da die Zielsysteme eh nicht in der Lage wären diese zu nutzen, gibt es auch keinen Grund, diese bereitzustellen.
- Für Embedded Debian "crush" sind viele funktionale Änderungen und Anpassungen notwendig, damit die Ziele erreicht werden können. Dies beinhaltet unter anderem, das Abschalten von Komponenten in Software, die schlicht nicht mit Crosscompilern zusammenarbeiten, das Anpassen der Paketabhängigkeiten, so dass keine unerwünschten Programme oder Bibliotheken erforderlich sind, oder gar das Aufteilen von Paketen in Unterpakete mit entsprechender Funktionalität und den damit einhergehenden Anpassen der Paket-Metadaten.
- Unterstützung von "TDeb" vergleichbar wie bei Embedded Debian "grip"
- Verwendung von "nodocs", ebenfalls wie in Embedded Debian "grip"
- Kompromisslose Optimierung zur Reduktion des notwendigen Speicherplatzes der Installation, der Paketgrößen und Abhängigkeitsbäume zwischen den einzelnen Paketen.
- Verwendung von Busybox anstelle der üblichen "coreutils", um eine noch größere Speicherplatzreduktion zu erreichen.

7 Zusammenfassung

Nach der intensiven Betrachtung der einzelnen bereits existierenden Build-Umgebungen für eingebettete System, kann man erkennen, dass es viele Ansätze gibt, das grundlegende Problem der beschränkten Ressourcen auf Hardware für eingebettete Systeme und deren Entwicklung anzugehen, aber keine alle Belange zur Zufriedenheit löst. Jedes der betrachteten System hat für sich gesehen auf seinem Gebiet entsprechende Stärken, aber universell und äußerst flexibel einsetzbar ist leider nur eines davon.

7.1 OpenWRT/FreeWRT

OpenWRT/FreeWRT bietet zwar an sich eine nette Build-Umgebung, in der sich auch zumindest mit akzeptablen Aufwand eigene Software oder Fremdsoftware integrieren lässt, allerdings kommt man um die Anpassung der eigenen Software, und sei es auch nur die Anpassung des Makefiles an die OpenWRT/FreeWRT Build-Umgebung, nicht herum. Die Praxis hat gezeigt, dass viele Softwarepakete, eben doch nur mit vielen Patches und Veränderungen auf OpenWRT/FreeWRT zu portieren sind.

Gerade für den Bereich der Datenanalyse oder Bildverarbeitung, gibt es für OpenWRT/FreeWRT nur wenige Pakete vom Distributor selbst. Das heißt, dass alle Pakete, die zum Betrieb einer Applikation aus dem Bereich der Bildverarbeitung nötig sind, zunächst selbst der OpenWRT/FreeWRT Build-Umgebung hinzugefügt werden müssen. Wenn man betrachtet, wofür OpenWRT/FreeWRT ursprünglich entwickelt worden war, verwundert es eben auch nicht, dass hier noch Defizite vorherrschen. OpenWRT/FreeWRT legt den Fokus sehr stark auf Netzwerkapplikationen, da es eben zunächst darum ging, die von der Firma Linksys angebotenen linuxbasierten Router, via Software mit erweiterten Funktionen auszustatten, und die Möglichkeiten dieser Geräte auszureizen.

Bis OpenWRT/FreeWRT also zu einer universellen Build-Umgebung für eingebettete Systeme werden würde, wäre noch sehr viel Arbeit nötig. Auch das Buildsystem als solches war schon mehrfach in der Diskussion, da es eben nicht so allgemeingültig gehalten ist, wie es sich einige Entwickler wünschen. Auch hier könnte man potentiell an Grenzen des auf Makefiles und Patches basierenden Systems stoßen.

7.2 Scratchbox

Scratchbox, bzw. Maemo, reiht sich hier etwas oberhalb vom zuvor erwähnten OpenWRT/FreeWRT ein. Es erleichtert dem Entwickler zwar die Kompilierung seiner Software, indem es ihn mit Hilfe von QEMU beim Umgang mit den "autotools" unterstützt, aber hier endet auch die Hilfestellung von Seiten Scratchbox.

Im weiteren Verlauf der Firmware-Erstellung stellt es den Entwickler vor die selben Probleme. Es ist nicht in eine Distribution integriert, und überlässt somit dem Entwickler die Aufgabe selbst für Abhängigkeiten von Bibliotheken zu sorgen. Des weiteren bietet es auch keine Unterstützung beim Erstellen der Firmware, bzw. des Firmware-Images. Der Entwickler muss selbst die zusammengehörigen Binärprogramme und Bibliotheken zusammenstellen, und in ein für das eingebettete System geeignetes Format bringen.

Weiterhin zeigt die Entwicklung, dass Scratchbox nur auf sehr wenigen Architekturen lauffähig ist. Primär ist Scratchbox auf die Architektur ARM ausgerichtet. Weitere Architekturen, wie beispielsweise PowerPC oder gar Mips sind schon seit Jahren von den Entwicklern als experimentell gekennzeichnet. Was auf Grund der Historie von Scratchbox nicht sehr verwundert, immerhin wird Scratchbox mittlerweile von Nokia stark gesponsert und unterstützt. Nokia interessiert sich natürlich primär dafür, dass Scratchbox, bzw. hier konkreter Maemo, perfekt auf die Mobiltelefone des Hauses ausgerichtet ist, welche ausschließlich unter der Architektur ARM laufen.

Hier zeigt sich auch schon ein weiteres Problem. Da Mobiltelefone eher zu den Geräten mit sehr wenig Ressourcen gehören, ist Scratchbox eher dafür ausgelegt, absolut minimalisiere Firmware zu erzeugen. Hier zählt geringer Speicherverbrauch wesentlich mehr, als möglicherweise Kompatibilität zu "normalen" Desktop-Systemen.

Insgesamt lässt sich sagen, dass Scratchbox sicher eine angenehme Art ist, für aktuelle und zukünftige Mobiltelefone aus dem Hause Nokia Software zu entwickeln. Aber für die im Rahmen dieser Arbeit interessanten etwas leistungsfähigeren eingebetteten Systeme, bietet es keine außerordentlich große Unterstützung an. Eine Erweiterung von Scratchbox auf eben solche Systeme wäre zwar möglich, aber selbst wenn man den sehr großen Aufwand betreiben möchte, wäre es fraglich, ob eine so große Erweiterung von Scratchbox jemals in den Hauptentwicklungszweig einfließen würde. Hier stehen eben zunächst die Interessen des Hauptsponsors im Vordergrund.

7.3 Embedded Debian

Embedded Debian stellt hier nun eine sehr gute Entwicklungsumgebung bereit. Wenn man sich nun noch vor Augen führt, dass Embedded Debian ja sogar noch in den Anfängen steckt, ist es erstaunlich, wie viel bereits erreicht wurde.

Cross-Kompilierung für eingebettete System in Verbindung mit einem sehr ausgereiften Paketmanagement, wie es Embedded Debian hat, ist ein sehr großer Gewinn und eine Erleichterung für den Entwickler. Es ermöglicht nicht nur die einfache Erstellung einer oder mehrerer Cross-Compiler-Umgebungen, dazu noch auf dem selben System, sondern bietet dem Entwickler auch noch die große Softwareauswahl des Debian Projektes zur Wiederverwendung an.

Der Entwickler kann sich also auf seine selbst erstellten Programme konzentrieren, und muss sich nicht mit den Unwägbarkeiten der Portierung und Cross-Kompilierung von Bibliotheken und fremder Software beschäftigen. Dies stellt einen enormen Gewinn für den Entwicklungsprozess dar, vor allem wenn der Entwickler bereits zuvor mit der Distribution Debian vertraut ist.

Zur Erstellung der letztendlich notwendigen Firmware, stehen hier dem Entwickler auch alle von der Distribution Debian gegebenen Möglichkeiten offen. Es findet keine Spezialisierung auf nur wenige Methoden oder Verfahren statt.

Darüber hinaus kann sich der Entwickler je nach den Anforderungen des eingebetteten Systems noch für ein sehr ressourcenschonendes und platzsparendes Basissystem (Embedded Debian "crush") oder für ein etwas komfortableres, allerdings auch leicht ressourcenhungrigeres Basissystem (Embedded Debian "grip") entscheiden.

Embedded Debian stellt dem Entwickler von eingebetteten Systemen alles Notwendige in einer sehr komfortablen Art und Weise zur Verfügung.

Teil III
Firmware Entwicklung

Auf Grund der Betrachtung der Buildumgebungen OpenWRT/FreeWRT, Scratchbox und Embedded Debian, wird hier nun die eigentliche Erstellung der Firmware für ein eingebettetes System beschrieben.

Im ersten Schritt wird dir Firmware für ein eingebettetes System auf PC-Basis erläutert. Allerdings wird hierbei bereits darauf geachtet, dass es später möglich sein soll, die hier entwickelten und angewandten Prinzipien auch auf die Firmware-Entwicklung auf einer anderen Prozessor-Architektur zu übertragen.

Basierend auf der im vorangegangenen Teil vorgenommenen Betrachtung der Systeme, der Beurteilung und Kategorisierung selbiger, kommt im Folgenden ausschließlich Embedded Debian zum Einsatz.

Auf Grund der jetzt schon großen Möglichkeiten, die Embedded Debian bietet, und dem im Fraunhofer IOSB bereits existierenden Wissen und der praktischen Anwendung von Debian als Linux Distribution, stellt Embedded Debian hier die beste Ausgangssituation dar.

8 Allgemeine Richtlinien

Die Konzeption und Erstellung einer Firmware für eingebettete Geräte auf Linux-Basis, folgt einigen allgemeinen Richtlinien, die generell bei der Entwicklung von eingebetteten Systemen zu berücksichtigen sind. Des weiteren gab es aber auch Wünsche und Neigungen des Fraunhofer IOSB[43], bzw. der

[43] http://www.iosb.fraunhofer.de

Abteilung IAD[44], welche Voraussetzungen die zu entwickelnde Firmware zu erfüllen hat. Da diese für später in der Arbeit zu treffende Entscheidungen einige Relevanz besitzen, bietet es sich an, diese hier detaillierter aufzuführen:

- flexible Build-Umgebung, d.h. anpassbar auf evtl. zukünftige Entwicklungen
- sehr gute Debian-Kenntnisse vorhanden, d.h. Firmware basierend auf der Distribution Debian und deren Werkzeuge wünschenswert. Dies erleichtert die evtl. spätere Weiterentwicklung.
- Software des Fraunhofer IOSB liegt bereits vielfach als Debian-Paket vor.
- Speicherplatzverbrauch der Firmware des eingebetteten Systems ist nicht als extrem kritischer Faktor anzusehen. Primär wird die Firmware auf USB-Sticks ausgeliefert werden.
- weitgehende Unabhängigkeit der Firmware vom Host-System auf dem diese erstellt wird.
- einfache Updatefähigkeit durch den Kunden (Austausch von Dateien)

9 Firmware für i386

Unter den soeben genannten Voraussetzungen gilt es nun im weiteren Verlauf dieser Arbeit eine Firmware für ein PC-basiertes eingebettetes System auf Debian-Basis, hier konkret Embedded Debian zu erstellen.

Die grundlegenden Schritte unterschieden sich hierbei kaum von der Installation eines "normalen" Debian-System.

1. Erstellen des Basissystems, evtl. schon minimiert im Speicherplatzverbrauch. (debootstrap)
2. Erzeugen und Verwalten eines eigenen Debian Paketrepositories, für die benötigten Embedded Debian Pakete. (reprepro)
3. Konzeption der Dateisysteme. (aufs)
4. Erzeugung des Linux-Kernels. (Linux-Kernel)
5. Integration sonstiger gewünschter oder benötigter Software, sowohl Freie Software als auch Software des Fraunhofer IOSB.

[44] http://www.iosb.fraunhofer.de/servlet/is/8439/

6. Anpassung des Systems durch selbst erstellte Skripte und Programme.
7. Erzeugen der notwendigen Firmware-Dateien.
8. Test der Firmware in einer simulierten Umgebung.

9.1 debootstrap

"debootstrap" kann verwendet werden, um ein Debian System von Grund auf neu zu installieren. Dies kann aus einem laufenden System heraus auf einer anderen Partition oder auch in einem Verzeichnis auf dem aktuellen System geschehen, beispielsweise um eine andere Release-Version von Debian zu testen, oder wie in diesem Fall, um ein Basis-System für ein eingebettetes System zu erstellen. Auch ist es möglich, mittels debootstrap eine Installation von Debian von einer anderen Linux-Distribution aus vorzunehmen.

"debootstrap" nutzt die folgende Syntax:

```
debootstrap [OPTION]... SUITE TARGET [MIRROR [SCRIPT]]
```

debootstrap benötigt neben den im Folgenden beschriebenen Optionen noch Informationen über die zu installierende Debian Version (SUITE) und das Verzeichnis (TARGET), in dem das neue System erzeugt werden soll. Weiterhin kann ein Debian Spiegel (MIRROR) angegeben werden, aus dem die notwendigen Pakete bezogen werden sollen. Optional kann auch ein Skript übergeben werden, dies wird während der Installation ausgeführt.

- –arch ARCH - Architektur, für die das System erzeugt werden soll.
- –download-only - Pakete werden nur vom Server geholt, aber nicht installiert.
- –include=a,b,c... - Lädt die zusätzlich angegebenen Pakete vom Server und installiert diese. Beachten Sie, dass Abhängigkeiten von Anwender aufzulösen sind, es müssen also alle notwendigen Pakete aufgeführt werden.
- –exclude=a,b,c... - Die aufgeführten Pakete werden nicht installiert und auch aus den internen Listen entfernt. Achtung: Dies wird wahrscheinlich essenziell wichtige Pakete löschen.
- –variant=buildd - Installiert ein System mit allen für das Programm buildd notwendigen Paketen.
- –verbose - Gibt mehr Informationen während der Installation aus.

- –print-debs - Erzeugt eine Liste der zu installierenden Pakete und beendet dann das Programm.
- –unpack-tarball DATEI - Entpackt die notwendigen Pakete aus dem Tar-Archiv DATEI. Es werden keine Pakete von einem Server geholt.
- –boot-floppies - Nur für die internen Tests beim Erstellen von eigenen Boot-Disketten.
- –debian-installer - Für interne Tests des Debian Installers.

Um nun das Basissystem für das zu erzeugende eingebettete System auf der Basis von Embedded Debian Grip zu erzeugen, ist folgender Befehlsaufruf nötig:

```
debootstrap --arch i386 --exclude="aptitude,dmidecode,iptables,tasksel,tasksel-da
ta,vim-common,vim-tiny" --include="less" lenny /tmp/image.tmp http://www.emdebian
.org/grip/
```

Aus dem konkreten Befehlsaufruf werden verschiedene Dinge ersichtlich. Zum einen wird mittels des Parameters "–arch i386" erzwungen, dass ein Basis-System für die PC-Architektur erstellt wird. Diese Angabe kann natürlich weggelassen werden, wenn das Host-System ebenfalls ein System auf PC-Basis darstellt.

Des weiteren kann man erkennen, dass Einiges an Software, welche normalerweise zum Basis-System gehören würde, weggelassen wird, um den Speicherplatzverbrauch zu minimieren. Diese Software wird für ein eingebettetes System auf der Basis von Embedded Debian "grip" nicht benötigt, und würde nur unnötig Speicherplatz verbrauchen, der anderweitig besser genutzt werden kann.

Durch die Angabe von "lenny" als "Suite", wird ein Basissystem auf der Basis des aktuellen stabilen Debian Lenny generiert.

Als Zielangabe wird hierbei ein Verzeichnis unter "/tmp" verwendet. Dies ist natürlich nicht unbedingt notwendig, hier kann auch ein beliebiger anderer Pfad genutzt werden. Allerdings bietet sich das Verzeichnis für temporäre Dateien bereits hier an, vor allem da später ja eine weitgehende Automatisierung des Erstellungsprozesses vorgesehen ist.

Zuletzt wird dem Kommando "debootstrap" noch der Spiegelserver mitgeteilt, von dem es die Pakete für das Basissystem beziehen soll. Würden wir ein "normales" Debian-System erzeugen wollen, wäre hier die Angabe eines offiziellen Debian-Servers notwendig. Da hier jedoch ein System auf der Basis von Embedded Debian "grip" erzeugt werden soll, wird hier nun der Server des Embedded Debian Projekts angegeben.

"debootstrap" arbeitet nun alle notwendigen Befehle ab, und stellt zum Ende seiner Tätigkeit ein minimales Basissystem in einem Verzeichnis bereit:

```
root@excelsior:~# debootstrap --arch i386 --exclude="aptitude,dmidecode,iptables
,tasksel,tasksel-data,vim-common,vim-tiny" --include="less" lenny /tmp/image.tmp
 http://www.emdebian.org/grip/
I: Retrieving Release
I: Retrieving Packages
I: Validating Packages
I: Resolving dependencies of required packages...
I: Resolving dependencies of base packages...
I: Found additional required dependencies: apt debconf debconf-i18n debian-archi
ve-keyring dhcp3-client dhcp3-common gnupg gpgv ifupdown libbz2-1.0 libdb4.6 lib
ncursesw5 libnewt0.52 libpopt0 libreadline5 libssl0.9.8 libusb-0.1-4 lzma module
-init-tools nano net-tools netbase ntpdate readline-common udev wget whiptail
I: Found additional base dependencies: libgdbm3 libsigc++-2.0-0c2a
I: Checking component main on http://www.emdebian.org/grip...
I: Retrieving adduser
I: Validating adduser
...
I: Retrieving zlib1g
I: Validating zlib1g
I: Chosen extractor for .deb packages: dpkg-deb
I: Extracting apt...
I: Extracting base-files...
...
I: Extracting whiptail...
I: Extracting zlib1g...
I: Installing core packages...
I: Unpacking required packages...
I: Unpacking apt...
I: Unpacking base-files...
...
I: Unpacking whiptail...
I: Unpacking zlib1g...
I: Configuring required packages...
I: Configuring sysv-rc...
I: Configuring libpam-runtime...
...
I: Configuring libcwidget3...
I: Configuring libgnutls26...
I: Base system installed successfully.
```

Dass nun das Basissystem für das eingebettete System erstellt wurde, lässt ja schon die letzte Meldung von "debootstrap" erahnen. Um dies zu überprüfen, bietet es sich jedoch an, nochmals einen kurzen Blick in das Verzeichnis zu werfen.

```
root@excelsior:~# du -sh /tmp/image.tmp
83M     /tmp/image.tmp
root@excelsior:~# ls -a -l /tmp/image.tmp
insgesamt 80
drwxr-xr-x 20 root root 4096 26. Apr 12:40 .
drwxrwxrwt  6 root root 4096 26. Apr 12:39 ..
drwxr-xr-x  2 root root 4096 26. Apr 12:40 bin
drwxr-xr-x  2 root root 4096 11. Apr 2009  boot
drwxr-xr-x  5 root root 4096 26. Apr 12:40 dev
drwxr-xr-x 39 root root 4096 26. Apr 12:40 etc
drwxr-xr-x  2 root root 4096 11. Apr 2009  home
drwxr-xr-x 10 root root 4096 26. Apr 12:40 lib
drwxr-xr-x  2 root root 4096 26. Apr 12:39 media
drwxr-xr-x  2 root root 4096 11. Apr 2009  mnt
drwxr-xr-x  2 root root 4096 26. Apr 12:39 opt
drwxr-xr-x  2 root root 4096 11. Apr 2009  proc
drwxr-xr-x  2 root root 4096 26. Apr 12:39 root
drwxr-xr-x  2 root root 4096 26. Apr 12:40 sbin
drwxr-xr-x  2 root root 4096 16. Sep 2008  selinux
drwxr-xr-x  2 root root 4096 26. Apr 12:39 srv
drwxr-xr-x  2 root root 4096 12. Aug 2008  sys
drwxrwxrwt  2 root root 4096 26. Apr 12:40 tmp
drwxr-xr-x 10 root root 4096 26. Apr 12:39 usr
drwxr-xr-x 13 root root 4096 26. Apr 12:39 var
```

Hier lässt sich erkennen, dass das komplette Basissystem erzeugt wurde, alle für ein minimales Linux-System erforderlichen Verzeichnisse existieren, mit Inhalt gefüllt sind und das Basis-System im aktuellen Zustand auf der Festplatte des HOST-Rechners 83MB Speicherplatz belegt.

9.2 reprepro

Es ist im Laufe dieser Arbeit notwendig viele Debian-Pakete oder Embedded Debian Pakete zu erzeugen. Diese manuell in evtl. verschiedenen Versionsnummern zu verwalten, und immer an die passenden Stellen zu kopieren stellt einen nicht zu verachtenden Aufwand dar.

Da Debian und auch Debian Embedded in der Lage sind, Pakete nicht nur von der lokalen Festplatte des Host-Systems, sondern auch von beliebigen Systemen aus dem Internet vollautomatisch herunterzuladen und zu verarbeiten, drängt es sich in diesem Zusammenhang geradezu auf, ein eigenes Paket-Repository für die notwendigen Pakete zu erzeugen. Dies erleichtert die Verwaltung und den Umgang mit diesen ungemein. Des weiteren stellt dies auch sicher, dass in Zukunft mehrere Personen zeitgleich auf immer die aktuellsten Versionsstände der Pakete zurückgreifen, und nicht aus Versehen veraltete, fehlerbehaftete oder gar schädliche Software zur Erzeugung der Firmware für eingebettete Systeme verwenden.

Ein Werkzeug hierfür ist das Programm "reprepro"[45]. "reprepro" kann nicht nur sehr große Paketverzeichnisse verwalten, es bietet sich auch durch seine einfache Handhabung für kleine bis mittlere Paket-Repositories an.

Zur Ablage des Paket-Repositories bot sich der Server "wsd.iitb.fhg.de"[46] des Fraunhofer IOSB an. Dieser Server ist sowohl vom Fraunhofer, als auch vom ganzen Internet aus zu erreichen. Als Zielverzeichnis für das Paket-Repository eignet sich zunächst für die Entwicklung der Firmware für das eingebettete System, das Home-Verzeichnis meines dortigen Benutzers. Für eine spätere Verwendung im Produktivbetrieb ist es allerdings geeigneter, das Paket-Repository unter einer anderen nicht benutzerspezifischen Internet-Adresse anzubieten.

Um das Paket-Repository mit "reprepro" aufzubauen, ist es zunächst nötig, sowohl die Verzeichnisstruktur, als auch eine grundlegende Konfigurationsdatei anzulegen.

Mittels des Kommandos "mkdir" muss das Verzeichnis "conf" unterhalb des Basis-Verzeichnisses für das Paket-Repository angelegt werden.

```
frank@wsd.iitb.fhg.de:~/public_html# mkdir conf
```

In dieses neu angelegte Verzeichnis muss nun die Konfigurations-Datei für "reprepro" gelegt werden. Für ein einfaches Test-Repository reichen relativ wenig Angaben:

- Origin - Quelle der Pakete

[45] http://mirrorer.alioth.debian.org/
[46] http://wsd.iitb.fhg.de/

- Label - Bezeichnung des Paket-Repositories
- Suite - stable, testing oder unstable, je nachdem für welche Version von Debian oder Embedded Debian das Repository verwendet werden soll.
- Codename - lenny, squeeze oder sid. Diese Angabe entspricht einer Art Alias, da häufig nicht die Bezeichnung für die Suite, sondern auch der direkte Codename der Debian-Version verwendet wird.
- Version - Die Versionsnummer des Paket-Repositories. Idealerweise verwendet man hier die Versionnummer der zugehörigen Debian-Version.
- Architectures - die in diesem Repository erlaubten System-Architekturen
- Components - Angabe, welche Komponenten in diesem Repository verwaltet werden. Da hier auch sog. "unfreie" Software direkt aus dem Fraunhofer IOSB verwaltet werden soll, ist hier zusätzlich zu "main" auch die Angebe von "contrib" und "non-free", also unfreier und damit zusammenarbeitender Software nötig.
- Description - Eine freie textuelle Beschreibung des Paket-Repositories

Die hier konkret verwendete Datei "distribution" lautet wie folgt:

```
frank@wsd.iitb.fhg.de:~/public_html# cat conf/distributions
Origin: Fraunhofer
Label: Fraunhofer
Suite: stable
Codename: lenny
Version: 5.0
Architectures: i386 powerpc
Components: main contrib non-free
Description: Test Repository
```

Die Aufnahme von Paketen in das Repository erfolgt nun mittels des Befehls "reprepro". Um beispielsweise das Paket des Editors "joe"[47] aufzunehmen, welches im Home-Verzeichnis des Benutzers liegt, reicht folgender Befehl:

```
frank@wsd.iitb.fhg.de:~/public_html# reprepro -b ~/public_html includedeb \
~/joe_3.5-2eml_i386.deb
```

Nun kann man sich zur Sicherheit noch mittels "reprepro" anzeigen lassen, ob und in welcher Version der Editor "joe" im Repository vorhanden ist:

```
frank@wsd.iitb.fhg.de:~/public_html# reprepro -b ~/public_html list stable joe
lenny|main|i386: joe 3.5-2eml
```

[47] Joe oder „Joe's Own Editor" ist ein freier (unter der GPL) Konsolen-basierter Texteditor für Unix-Systeme. Er ist auf vielen Unix-Distributionen als Standard-Anwendung beigefügt.

Um den Editor "joe" wieder aus dem Repository zu entfernen, sollte dies nötig werden, kann dies ebenfalls mittels "reprepro" bewerkstelligt werden:

```
frank@wsd.iitb.fhg.de:~/public_html# reprepro -b ~/public_html remove joe
```

"reprepro" bietet selbstverständlich noch sehr viel mehr Optionen und Kommandos. Für die Funktionalität, die im Rahmen dieser Arbeit benötigt wird, sind jedoch die oben genannten Möglichkeiten ausreichend. Weitere Optionen und Konfigurationen dienen zum Verwalten, sehr großer Repositories mitsamt Quellcode oder evtl. kryptographischer Signierung der Metadaten. Jedoch werden diese Möglichkeiten hier nicht benötigt.

9.3 aufs

aufs[48] (Another Unionfs) ist ein Overlay-Dateisystem, welches zum (scheinbaren) Schreiben von Daten auf nicht beschreibbaren Datenträgern (wie z. B. CD-ROMs[49] und DVDs[50]) benötigt wird. Dazu werden mindestens zwei Dateisysteme übereinander gelegt. Dabei wird ein beschreibbares Dateisystem über das nicht beschreibbare gelegt. Soll nun eine Datei gelesen werden, wird zunächst versucht, es auf dem beschreibbaren Dateisystem zu lesen. Ist diese dort nicht vorhanden, wird diese aus dem darunter liegenden nicht beschreibbaren Dateisystem gelesen. Ein Schreibzugriff erfolgt immer auf das beschreibbare Dateisystem.

aufs ist eine Abspaltung von UnionFS, welches aber von Junjiro Okajima, der auch schon für UnionFS[51] viele Bugfixes geliefert hat, komplett neu entwickelt wurde.

Für das folgende Beispiel wird davon ausgegangen, dass man ein Nur-Lesen Dateisystem habt, in dem man aber gerne trotzdem etwas schreiben möchte, ohne erst das gesamte Dateisystem auf die Festplatte zu kopieren bzw. auch um nach einer Reihe von Änderungen nachvollziehen zu können, was genau geändert wurde. Hierzu geht das Beispiel von einer CD aus, die unter "/mnt/cdrom" gemountet wurde und einem Verzeichnis "/branches/tmp" in welches Änderungen geschrieben werden sollen:

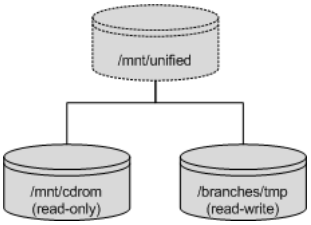
Vor dem ersten Mounten sollte sichergestellt sein, dass das Verzeichnis "/branches/tmp" leer ist. Um nun aufs zu mounten wird folgender Befehl ausgeführt:

[48] http://aufs.sourceforge.net/

[49] CD-ROM ist die Abkürzung für Compact Disc Read-Only Memory, ein physikalischer Permanentspeicher für digitale Daten.

[50] Die DVD ist ein digitales Speichermedium, das im Aussehen einer CD ähnelt, aber über eine deutlich höhere Speicherkapazität verfügt. Sie zählt zu den optischen Datenspeichern. Das Backronym „DVD" geht auf die Abkürzung von Digital Versatile Disc (engl. für digital vielseitige Scheibe) zurück.

[51] http://www.fsl.cs.sunysb.edu/project-unionfs.html

Abbildung 14: 


```
mount none /mnt/unified -t aufs -o br:/mnt/cdrom=ro:/branches/tmp=rw
```

Interessant ist der letzte Teil, der hinter dem "-o" folgt und das aufs konfiguriert. Hier wird mit "br:" die Liste der durch Doppelpunkt getrennten Branches eingeleitet:

br:/mnt/cdrom=ro:/branches/tmp=rw => es werden zwei Branches definiert:

- /mnt/cdrom=ro - Ist der Read-Only Branch. Das =ro markiert den Branch als read-only
- /branches/tmp=rw - Ist der Branch in dem die Änderungen geschrieben werden, markiert durch =rw für read-write

Unter "/mnt/unified" können nun Änderungen an dem Dateisystem auf der CD durchgeführt werden. Die geänderten Dateien finden sich anschließend alle unter "/branches/tmp". Im Falle einer CD könnte man nun zum Beispiel ein neues ISO Image direkt aus "/mnt/unified" erstellen. Eine andere Verwendungsmöglichkeit ist zum Beispiel das Speichern der Kernel-Quellen in einem kompakten SquashFS mit einem aufs als zusätzlichen Layer der ein Schreiben (und damit Kompilieren) ermöglicht.

Dieses Verfahren bietet sich auch im Bereich der eingebetteten System an. Es ermöglicht es ein nur lesbares Firmware-Image dennoch zur Laufzeit zu beschreiben. Firmware-Images werden meist in einem schreibgeschützten Format ausgeliefert. Dies bietet mehrere Vorteile:

- stärkere Komprimierung möglich (Speicherplatz-Optimierung)
- Basis-System ist geschützt vor Veränderungen zur Laufzeit durch den Benutzer (rescue-Modus)

Durch die Verwendung von "aufs" lassen sich hier alle positiven Eigenschaften verbinden. Ein hoch komprimiertes, nur lesbares Basis-System (Firmware-Image) lässt sich hiermit mit einem beschreibbaren Dateisystem überlagern,

so dass es zur Laufzeit das System dennoch verändert und angepasst werden kann.

Leider existieren von "aufs" keine offiziellen Veröffentlichungen, die sich einfach herunterladen lassen würden, stattdessen, muss man sich des Versionsverwaltungs-Werkzeugs "git"[52] bedienen.

Um nun "aufs" für den im nachfolgenden verwendeten Linux-Kernel zu erhalten, sind mehrere Befehle notwendig:

```
florian@excelsior:~$ git clone http://git.c3sl.ufpr.br/pub/scm/aufs/aufs2-standalone.git
florian@excelsior:~$ cd aufs2-standalone
florian@excelsior:~$ git checkout origin/aufs2-33
```

Danach befindet sich die Version "aufs2-33" auf der Festplatte im Verzeichnis "aufs2-standalone":

```
florian@excelsior:~/compile/aufs2-standalone$ ls -a -l
insgesamt 240
drwxr-xr-x 7 florian florian   4096 2010-03-29 15:50 .
drwxr-xr-x 5 florian florian   4096 2010-03-29 15:19 ..
-rw-r--r-- 1 florian florian   3049 2010-03-29 15:50 aufs2-base.patch
-rw-r--r-- 1 florian florian    977 2010-03-29 15:50 aufs2-kbuild.patch
-rw-r--r-- 1 florian florian   8629 2010-03-29 15:50 aufs2-standalone.patch
-rw-r--r-- 1 florian florian 150944 2010-03-29 15:50 ChangeLog
-rw-r--r-- 1 florian florian   2436 2010-03-29 15:50 config.mk
-rw-r--r-- 1 florian florian  17990 2010-03-29 15:19 COPYING
drwxr-xr-x 2 florian florian   4096 2010-03-29 15:19 design
drwxr-xr-x 3 florian florian   4096 2010-03-29 15:19 Documentation
drwxr-xr-x 3 florian florian   4096 2010-03-29 15:19 fs
drwxr-xr-x 8 florian florian   4096 2010-03-29 15:50 .git
drwxr-xr-x 3 florian florian   4096 2010-03-29 15:19 include
-rw-r--r-- 1 florian florian    823 2010-03-29 15:19 Makefile
-rw-r--r-- 1 florian florian  13231 2010-03-29 15:19 README
```

Wie nun erkennbar ist, wurde "aufs" in der gewünschten Version aus dem "git"-Repository ausgecheckt. Nun stehen diverse Patches und der Programmcode zur Integration von "aufs" in den Linux-Kernel zur Verfügung.

9.4 Linux-Kernel

Der Kernel eines Betriebssystems bildet die hardwareabstrahierende Schicht (Hardwareabstraktionsschicht), das heißt, er stellt der auf dieser Basis aufsetzenden Software eine einheitliche Schnittstelle (API) zur Verfügung, die unabhängig von der Rechnerarchitektur ist. Die Software kann so immer auf die Schnittstelle zugreifen und braucht die Hardware selbst, die sie nutzt, nicht genauer zu kennen. Linux ist dabei ein modularer monolithischer Betriebssystemkern und zuständig für Speicherverwaltung, Prozessverwaltung, Multitasking, Lastverteilung, Sicherheitserzwingung und Eingabe/Ausgabe-Operationen auf verschiedenen Geräten.

[52] Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien. Es wurde ursprünglich für die Quellcode-Verwaltung des Linux-Kernels entwickelt.

Da die bei den "normalen" Distributionen mitgelieferten Linux-Kernel[53] sehr groß sind, um auch eine Vielzahl an Hardware- und Systemkonfigurationen abzudecken, wäre ein so konfigurierter Linux-Kernel für ein eingebettetes System völlig unnötig, und würde auch dazu führen, den Speicherplatz zu sehr großen Teilen in Anspruch zu nehmen.

Aus diesem Grund ist es für eingebettete Systeme immer ratsam, selbst einen angepassten und auf das eingebettete System zugeschnittenen Linux-Kernel zu erstellen.

Zunächst ist es aber noch notwendig, die im vorangegangenen Abschnitt ausgecheckten Änderungen für das Dateisystem "aufs" in den Linux-Kernel einzuarbeiten. Die bei "aufs" mitgelieferte Datei "README" beschreibt diesen Vorgang und welche Schritte hierfür notwendig sind sehr ausführlich. Im Wesentlichen gilt es, die Patches und Quellcode-Dateien in den Quellcode-Baum des Linux-Kernels zu integrieren. Dies wird mit folgenden Befehlen vorgenommen:

```
tar -jxf ../linux-2.6.33.1.tar.bz2
cd linux-2.6.33.1/
patch -p1 < ~/aufs2-standalone/aufs2-kbuild.patch
patch -p1 < ~/aufs2-standalone/aufs2-base.patch
cp ~/aufs2-standalone/fs/aufs fs/ -R
cp ~/aufs2-standalone/Documentation/* Documentation/ -R
cp ~/aufs2-standalone/include/linux/aufs_type.h include/linux/
```

Nach diesen Vorarbeiten kann die eigentliche Konfiguration des Linux-Kernels von statten gehen.

Mittels des Kommandos "make menuconfig" ist es möglich die Einstellungen und Optionen des Linux-Kernels vollständig über ein textbasiertes Curses-Interface vorzunehmen. Die Konfigurationsoberfläche präsentiert sich nach dem Starten wie in der Abbildung auf der nächsten Seite dargestellt.

Alle vorgenommenen Einstellungen darzulegen und zu präsentieren würde den Rahmen dieser Arbeit sprengen, deshalb werden im Folgenden nur die absolut wichtigsten Einstellungen dargestellt. Des weiteren sind viele Optionen, wie beispielsweise verschiedene Treiber, sehr davon abhängig, auf welcher Hardware das eingebettete System eingesetzt werden soll.

Als außerordentlich wichtig sind die Einstellungen für die Dateisysteme zu sehen. Diese bewirken, dass das zuvor geschilderte Konzept mittels des Dateisystems "aufs" und der Nur-Lese-Firmware in einem anderen Dateisystem (SquashFS) umgesetzt werden kann.

Diese Einstellungen finden sich allesamt, wie in der Abbildung auf der nächsten Seite dargestellt unter dem Menüpunkt "Miscellaneous filesystems", einem Untermenü von "File systems".

[53] http://www.kernel.org

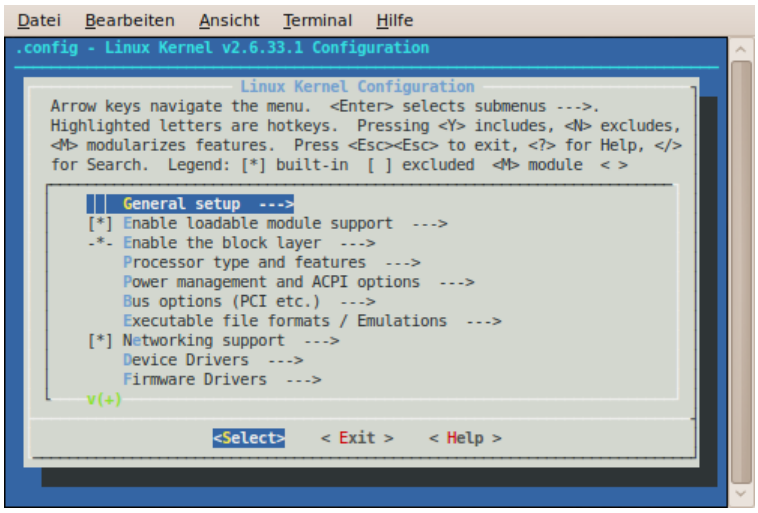


Abbildung 15:

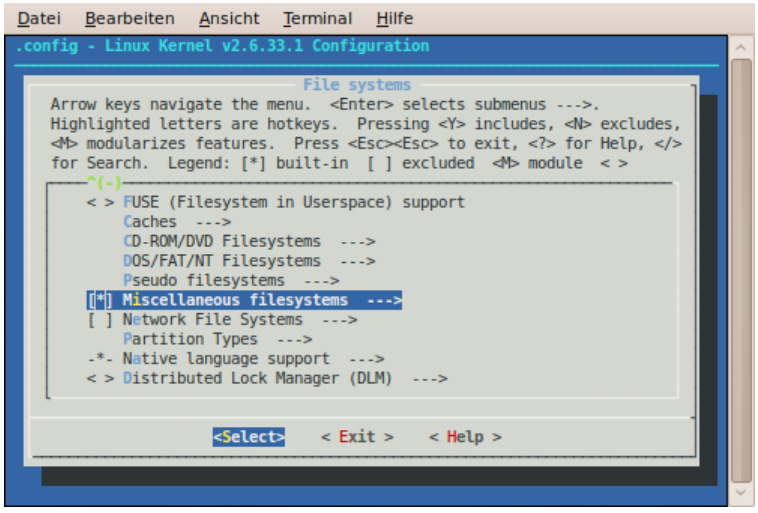


Abbildung 16:

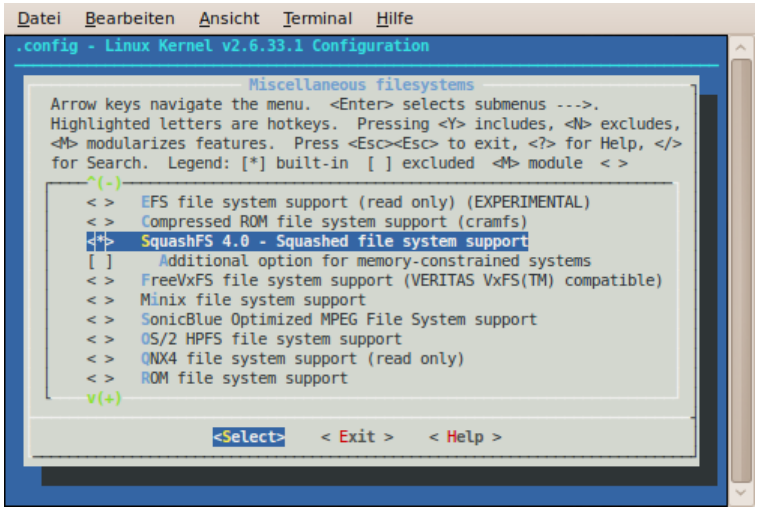


Abbildung 17:

Zunächst ist es notwendig hier die Unterstützung für das Dateisystem wie auf dieser Seite gezeigt zu aktivieren. Hierzu muss der Menüpunkt "SquashFS 4.0 - Squashed file system support" aktiviert werden und fest in den Linux-Kernel eingebaut werden. Somit steht dem späteren eingebetteten System der Treiber für dieses komprimierende Dateisystem direkt während des Startvorgangs zur Verfügung.

Des weiteren ist es ebenso notwendig, das zuvor in den Linux-Kernel integrierte Overlay-Dateisystem "aufs" zu aktivieren. Hierzu ist wie in der Abbildung auf der nächsten Seite dargestellt, die Option "Aufs (Advanced multi layered unification filesystem) support" zu aktivieren, und ebenfalls fest in den Linux-Kernel zu integrieren.

Nach der erfolgten Konfiguration und der Beendigung des Konfigurationswerkzeugs, muss der Linux-Kernel kompiliert und paketiert werden. Hierzu stellt Debian auch direkt mit "make-kpkg" aus dem Paket "kernel-package" ein entsprechendes Werkzeug bereit:

```
fakeroot make-kpkg --append-to-version=-mcmxt --revision=01 kernel_image
```

Hierdurch wurde nun ein Linux-Kernel-Paket erstellt, wie sich leicht durch Auflisten des Verzeichnisinhaltes erkennen lässt.

```
florian@excelsior:~/kernel/mcmxt$ ls -a -l
insgesamt 3748 drwxr-xr-x  3 florian florian     4096 2010-03-29 18:43 .
drwxr-xr-x  5 florian florian     4096 2010-03-29 13:56 ..
-rw-r--r--  1 florian florian    46810 2010-03-29 15:51 config
drwxr-xr-x 25 florian florian     4096 2010-04-26 16:14 linux-2.6.33.1
-rw-r--r--  1 florian florian 3775608 2010-03-29 18:43 linux-image-2.6.33.1-mcmxt_01_i386.deb
```

Dieses Linux-Kernel-Paket sollte nun ebenfalls, wie im Kapitel "reprepro" beschreiben in das Paket-Repository integriert werden. Dies ist zwar nicht zwingend nötigt, erleichtert aber die spätere Arbeit mit diesem Paket.

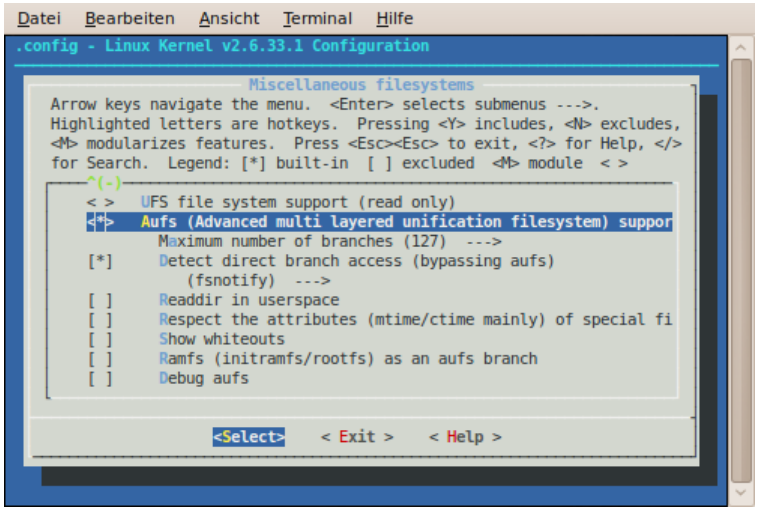


Abbildung 18:

9.5 zusätzliche Basissoftware

Um einige grundlegende Funktionen des eingebetteten Systems zu ermöglichen wird jedoch außer dem Basissystem noch weitere Software benötigt. Diese Software ist allesamt direkt von Debian erhältlich. Um diese jedoch im eingebetteten System auf der Basis vom Embedded Debian nutzen zu können, ist es nötig diese Software mittels der weiter oben beschriebenen Verfahren in Pakete für Embedded Debian umzuwandeln und diese dann in das erstellte Paketrepository mittels “reprepro” zu integrieren.

9.5.1 dropbear

Dropbear stellt eine (ebenfalls freie) Alternative zum OpenSSH[54]-Server und -Client dar. Dropbear wurde von Matt Johnston speziell für Arbeitsumgebungen ausgelegt, die wenig Speicher- und/oder Prozessorresourcen zur Verfügung stellen.

Dropbear stellt eine vollständige Implementierung des SSH2-Protokolls dar – sowohl die des Clients als auch des Servers. Die Version 1 des SSH-Protokolls wurde nicht implementiert: In erster Linie dient dies um einen schlanken Ausführungsprogrammcode zu erhalten, Betriebsmittel zu sparen und um bekannte Sicherheitsrisiken[55] in SSH Version 1 zu umgehen.

Dropbear ist für folgende Betriebssysteme verfügbar:

[54] OpenSSH (Open Secure Shell) ist eine Zusammenstellung von Computerprogrammen, die verschlüsselte Verbindungen über ein Computernetzwerk mittels des SSH-Protokolls ermöglichen.

[55] Die von SSH-1 verwendete Integritätsprüfung weist Schwachstellen auf, die es einem Angreifer ermöglichen, eine SSH-1-Sitzung auszuspähen.

- Linux
- Mac OS X
- FreeBSD, NetBSD and OpenBSD
- Solaris
- IRIX 6.5
- Tru64 5.1
- AIX 4.3.3
- HPUX 11.00
- Cygwin

In der Firmware des eingebetteten System übernimmt Dropbear die Aufgabe des SSH-Servers, so dass ein Shell-Zugriff zur Administration und Wartung über Netzwerk möglich ist.

9.5.2 mtools

Die mtools sind eine Sammlung von Public-Domain-Programmen um mit einem UNIX-Rechner auf DOS[56]-Dateisysteme (in der Praxis: DOS-formatierte Disketten oder USB-Sticks) zuzugreifen. Die Kommandos emulieren weitestgehend die entsprechenden DOS-Befehle, es gibt jedoch einige (kleine und teilweise unvermeidliche) Unterschiede.

Es gibt praktisch alle Diskettenzugriffe, die es auch unter MSDOS gibt. Im Einzelnen sind das folgende Befehle:

- mattrib - DOS-Dateiattribute verändern
- mcd - in ein DOS-Verzeichnis wechseln
- mcopy - Datei(en) von einer DOS-Diskette auf einen UNIX-Rechner (und umgekehrt) kopieren
- mdel - DOS-Datei(en) löschen.
- mdir - DOS-Verzeichnis auflisten lassen.
- mlabel - DOS-Diskette mit einem Namen (einem Label) versehen

[56] Als Disk Operating System (englische Aussprache: [dɪsk ˈɒpəɹeɪtɪŋ ˈsɪstəm]; kurz DOS) werden kleine und einfache Betriebssysteme für Computer bezeichnet, deren Hauptaufgabe die Verwaltung von magnetischen Speichermedien wie Disketten und Festplatten ist.

- mmd - DOS-(Unter-)Verzeichnis erzeugen
- mrd - DOS-(Unter-)Verzeichnis löschen
- mread - Datei(en) von einer DOS-Diskette auf einen UNIX-Rechner kopieren
- mren - eine bereits vorhandene DOS-Datei umbenennen
- mtype - Inhalt einer DOS-Datei anzeigen lassen
- mwrite - Datei(en) von einem UNIX-Rechner auf eine DOS-Diskette kopieren

Die mtools können verwendet werden, um auf Dateien auf DOS-Disketten oder wie in diesem Fall, mit FAT[57] formatierten USB-Sticks zusätzliche Dateien zur Laufzeit in das eingebettete System zu integrieren, ohne jedoch das entsprechende Dateisystem direkt zu mounten.

9.5.3 tcl8.4 und tcllib

Tcl[58] (Aussprache engl. tickle oder auch als Abkürzung für Tool command language) ist eine Open Source-Skriptsprache.

Tcl wurde ursprünglich ab 1988 von John Ousterhout an der University of California, Berkeley als Makrosprache für ein experimentelles CAD-System entwickelt. Aus dieser Zeit stammt das Konzept, den Tcl-Interpreter als Bibliothek in z. B. ein C-Programm einzubinden, was auch noch heute möglich ist.

Die Wahlsprüche von Tcl lauten: „radically simple“, also „radikal einfach“, was sich insbesondere auf die Syntax der Sprache bezieht und "everything is a string", "Alles ist Text", was sich auf den Umgang mit Befehlen und Daten in Tcl bezieht.

Diese Syntax folgt der polnischen Notation. Die einzigen wirklich reservierten Wörter sind die geschweiften Klammern zur Festlegung von Blöcken, die eckigen Klammern zur Evaluation von Ausdrücken, die Anführungszeichen zur Abgrenzung von Strings, der Backslash und das Zeilenende. Alle anderen Bestandteile der Sprache können umdefiniert werden. Zwischen eingebauten und von Programmen oder Tcl-Libraries hinzugefügten Funktionen besteht kein Unterschied.

Zur Einbindung externer Bibliotheken besitzt Tcl ein eigenes Paketsystem, das diese auch bei Bedarf automatisch nachladen kann. Weiterhin ist es

[57] Das File Allocation Table (FAT [fæt]; auf Deutsch etwa „Dateizuordnungstabelle“) ist ein von Microsoft entwickeltes Dateisystem.

[58] http://www.tcl.tk

möglich, Tcl-Programme um in C oder einer anderen kompilierten Sprache geschriebene Bibliotheken zu erweitern; hierfür existiert in Form der TclStubs eine standardisierte Schnittstelle. Außerdem können mithilfe der CriTcl[59]-Erweiterung zeitkritische Programmteile in C-Quellcode innerhalb des Tcl-Quellcodes notiert werden. Diese werden automatisch kompiliert und eingebunden.

Tcllib[60] ist eine der populärsten Bibliotheken zur Programmiersprache Tcl. Sie enthält eine Reihe Funktionen unter anderem für Internet-bezogene, mathematische und programmiertechnische Alltagsaufgaben und ist selbst vollständig in Tcl geschrieben.

Da Tcl eine in der Abteilung IAD des Fraunhofer IOSB sehr beliebte Skriptsprache ist, benötigen sehr viele Anwendungsprogramme dieser Abteilung einen installierten und lauffähigen Tcl-Interpreter. Für die eigentliche Funktion des eingebetteten Systems ist er nicht zwingend notwendig, wohl aber für viele Anwendungen des Fraunhofer IOSB.

9.5.4 thttpd

Ein Webserver ist ein Dienst, der Dokumente an Clients wie z. B. Webbrowser überträgt. Als Webserver bezeichnet man den Computer mit Webserver-Software oder nur die Webserver-Software selbst. Webserver werden lokal, in Firmennetzwerken und überwiegend als WWW-Dienst im Internet eingesetzt. Dokumente können somit dem geforderten Zweck lokal, firmenintern und weltweit zur Verfügung gestellt werden.

Für das eingebettete System stellt der minimalistische Webserver "thttpd"[61] eine einfache, auf Webseiten basierende Konfigurationsschnittstelle bereit, mittels der sich der Endkunde sehr einfach das eingebettete System einrichten und konfigurieren kann.

9.5.5 ghostscript

Ghostscript ist ein kostenloser Interpreter der Seitenbeschreibungssprachen PostScript und Portable Document Format (PDF).

Es besteht aus einem Softwarepaket, das eine API mit Funktionen bereitstellt, um PostScript und PDF auf Druckern oder Bildschirmen darzustellen. Es bietet einen hohen Grad an Kompatibilität mit dem proprietären „Original“ von Adobe.

Die wichtigsten Einsatzmöglichkeiten von Ghostscript sind:

[59] http://www.equi4.com/starkit/critcl.html
[60] http://tcllib.sourceforge.net/
[61] http://www.acme.com/software/thttpd/

- Rasterung von PostScript[62]- oder PDF[63]-Dateien. Durch diesen Prozess wird die als PostScript- oder PDF-Code vorliegende Beschreibung einer Seite für den Drucker oder den Monitor aufbereitet und als Abbildung sichtbar. Man spricht von einem Software-RIP.
- Konvertierung von PostScript- und PDF-Dateien in andere Dateiformate. Überwiegend werden PostScript-Dateien in PDF-Dateien oder reine Rastergrafikformate umgewandelt. Bei der Konvertierung nach PDF bleiben alle Informationen über Vektor- und Pixel-Elemente erhalten, während bei der Ausgabe von Rastergrafiken alle Elemente in einer bestimmten Auflösung gerastert werden.
- PostScript-Code zum Entwickeln und Debuggen von PostScript-Programmen ausführen lassen.

Ghostscript ist ebenfalls nicht zwingend für die Grundfunktionalität des eingebetteten System notwendig. Jedoch ist er für die korrekte Funktion des Programms MCMXT des Fraunhofer IOSB notwendig. Ghostscript wird hier zur Erzeugung von PDF-Dokumenten direkt auf dem eingebetteten System eingesetzt.

9.6 zusätzliche Konfigurationen und Skripte

Da die Standard-Konfigurationen, die mit den Debian-Paketen mitgeliefert werden, nur sehr rudimentär sind, ist es nötig diese noch teilweise anzupassen, sowohl um die erwünschte Funktionalität zu erreichen, als auch diese auf die Besonderheiten eines eingebetteten Systems anzupassen.

9.6.1 /etc/mtools.conf

Die Konfigurationsdatei der "mtools" ist per Default darauf ausgelegt, Dateien von Disketten oder IDE-Festplatten zu kopieren. Da hier jedoch primär Dateien von USB-Sticks kopiert werden sollen, ist es notwendig, die Konfiguration entsprechend anzupassen.

Da das eingebettete System von USB starten soll, ist die Unterstützung von USB-Sticks oder -Festplatten fest in den Kernel integriert. Die Unterstützung von evtl. vorhandenen IDE- oder SATA-Festplatten, die möglicherweise im eingebetteten System zusätzlich vorhanden sind, liegt hingegen nur modular

[62] PostScript ist eine Seitenbeschreibungssprache, die unter diesem Namen seit 1984 vom Unternehmen Adobe entwickelt wird.

[63] Das Portable Document Format (PDF; deutsch: (trans)portables Dokumentenformat) ist ein plattformunabhängiges Dateiformat für Dokumente, das vom Unternehmen Adobe Systems entwickelt und 1993 veröffentlicht wurde.

vor. Dies bedeutet, dass USB-Sticks immer vor den internen Festplatten initialisiert werden, und deshalb im System als erste Festplatte sichtbar wird. Diese ist immer unter der Bezeichnung "/dev/sda" erreichbar.

```
# Debian default mtools.conf file.
# "info mtools" or "man mtools.conf" for more detail.

# # Linux floppy drives
drive a: file="/dev/fd0" exclusive
drive b: file="/dev/fd1" exclusive

# # First SCSI hard disk partition
drive c: file="/dev/sda1"

# # First IDE hard disk partition
# drive c: file="/dev/hda1"

# # dosemu hdimage.
drive m: file="/var/lib/dosemu/hdimage.first" partition=1 offset=128

# # dosemu floppy image
drive n: file="/var/lib/dosemu/fdimage"

# # SCSI zip disk
# drive z: file="/dev/sda4"

# # uncomment the following line to display all file names in lower
# # case by default
mtools_lower_case=1
```

9.6.2 /etc/apt/sources.list.d/fhg.sources.list

Damit das eingebettete System Zugriff auf das weiter oben erstellte Paket-Repository erhält, muss dem Paketmanagement "apt" via Konfigurationsdatei der Pfad zu diesem Repository bekannt gemacht werden. Nur so ist es möglich, die selbst erstellen Pakete für Embedded Debian zu verwenden.

```
deb http://wsd.iitb.fhg.de/~frank/ lenny main contrib non-free
```

9.6.3 /etc/default/rcS

Hier werden die ganz allgemeinen Konfigurationseinstellungen für die Startskripte des SystemV Init Systems eingestellt.

Da das Nur-Lese-Root-Dateisystem zur temporären Beschreibbarkeit während des Betriebs mit einem sogenannten "tmpfs" über das Overlay-Dateisystem "aufs" verknüpft wird, und hierzu das bereits existierende "tmpfs" unter dem Verzeichnis "/tmp" genutzt wird, muss durch den Parameter "TMPTIME=infinite" dafür gesorgt werden, dass hier nicht zu einem späteren Zeitpunkt des Startvorgangs wieder wichtige und benötigte Dateien und Verzeichnisse gelöscht werden.

Des Weiteren wird durch die beiden Parameter "RAMRUN=yes" und "RAMLOCK=yes" dafür gesorgt, dass unter den Pfaden "/var/run" für PID-Dateien und "/var/lock" für Lock-Dateien, ebenfalls ein "tmpfs" gemountet und verwendet wird. Wäre dies nicht der Fall, würde das System nicht funktionieren,

da auf dem Nur-Lese-Root-Dateisystem ein Erstellen dieser Dateien nicht möglich wäre.

```
#
# /etc/default/rcS
#
# Default settings for the scripts in /etc/rcS.d/
#
# For information about these variables see the rcS(5) manual page.
#
# This file belongs to the "initscripts" package.

TMPTIME=infinite
SULOGIN=no
DELAYLOGIN=no
UTC=yes
VERBOSE=no
FSCKFIX=no
RAMRUN=yes
RAMLOCK=yes
```

9.6.4 /etc/hostname

Um das eingebettete System wieder zuerkennen, ist es von Vorteil einen eindeutigen Namen hierfür zu vergeben. Der auch beim Endkunden als solcher ersichtlich wird. Damit ist es dem Endkunden sehr einfach möglich nachzusehen, welches System er zur Zeit einsetzt.

```
mcmxt
```

9.6.5 /etc/thttpd/thttpd.conf

Der Webserver "thttpd" muss ebenfalls noch korrekt eingestellt werden. Zunächst muss dem Webserver "thttpd" mitgeteilt werden, dass nicht die Dateien unter "/var/www", wie voreingestellt, sondern die Dateien und Skripte unter dem Verzeichnis "/usr/share/MCMXT2D/cgi" ausgeliefert oder ausgeführt werden sollen. Hier befindet sich das Webfrontend des Markenservers MCMXT, womit dieser komfortabel gesteuert werden kann. Dies geschieht über den Parameter "dir=/usr/share/MCMXT2D/cgi".

Da der Markenserver MCMXT unter seinem eigenen Benutzer Namens "mcmxt" laufen wird, und der Webserver, bzw. dessen CGI-Skripte, mittels Shared Memory direkt mit dem Markenserver kommunizieren müssen, ist es notwendig, sowohl den Markenserver MCMXT als auch den Webserver unter dem selben Benutzer laufen zu lassen. Hierfür wird der Webserver unter dem Benutzer "mcmxt" betrieben, dies erfolgt mittels des Parameters "user=mcmxt".

```
# /etc/thttpd/thttpd.conf: thttpd configuration file

# This file is for thttpd processes created by /etc/init.d/thttpd.
# Commentary is based closely on the thttpd(8) 2.25b manpage, by Jef Poskanzer.

# Specifies an alternate port number to listen on.
port=80
```

```
# Specifies a directory to chdir() to at startup. This is merely a convenience -
# you could just as easily do a cd in the shell script that invokes the program.
dir=/usr/share/MCMXT2D/cgi

# Do a chroot() at initialization time, restricting file access to the program's
# current directory. If chroot is the compiled-in default (not the case on
# Debian), then nochroot disables it. See thttpd(8) for details.
#nochroot
#chroot

# Specifies a directory to chdir() to after chrooting. If you're not chrooting,
# you might as well do a single chdir() with the dir option. If you are
# chrooting, this lets you put the web files in a subdirectory of the chroot
# tree, instead of in the top level mixed in with the chroot files.
#data_dir=

# Don't do explicit symbolic link checking. Normally, thttpd explicitly expands
# any symbolic links in filenames, to check that the resulting path stays within
# the original document tree. If you want to turn off this check and save some
# CPU time, you can use the nosymlinks option, however this is not
# recommended. Note, though, that if you are using the chroot option, the
# symlink checking is unnecessary and is turned off, so the safe way to save
# those CPU cycles is to use chroot.
#symlinks
#nosymlinks

# Do el-cheapo virtual hosting. If vhost is the compiled-in default (not the
# case on Debian), then novhost disables it. See thttpd(8) for details.
#vhost
#novhost

# Use a global passwd file. This means that every file in the entire document
# tree is protected by the single .htpasswd file at the top of the tree.
# Otherwise the semantics of the .htpasswd file are the same. If this option is
# set but there is no .htpasswd file in the top-level directory, then thttpd
# proceeds as if the option was not set - first looking for a local .htpasswd
# file, and if that doesn't exist either then serving the file without any
# password. If globalpasswd is the compiled-in default (not the case on Debian),
# then noglobalpasswd disables it.
#globalpasswd
#noglobalpasswd

# Specifies what user to switch to after initialization when started as root.
user=mcmxt

# Specifies a wildcard pattern for CGI programs, for instance "**.cgi" or
# "/cgi-bin/*". See thttpd(8) for details.
cgipat=*.cgi

# Specifies a file of throttle settings. See thttpd(8) for details.
throttles=/etc/thttpd/throttle.conf

# Specifies a hostname to bind to, for multihoming. The default is to bind to
# all hostnames supported on the local machine. See thttpd(8) for details.
#host=

# Specifies a file for logging. If no logfile option is specified, thttpd logs
# via syslog(). If logfile=/dev/null is specified, thttpd doesn't log at all.
logfile=/var/log/thttpd.log

# Specifies a file to write the process-id to. If no file is specified, no
# process-id is written. You can use this file to send signals to thttpd. See
# thttpd(8) for details.
#pidfile=

# Specifies the character set to use with text MIME types.
#charset=iso-8859-1

# Specifies a P3P server privacy header to be returned with all responses. See
# http://www.w3.org/P3P/ for details. Thttpd doesn't do anything at all with the
# string except put it in the P3P: response header.
#p3p=

# Specifies the number of seconds to be used in a "Cache-Control: max-age"
# header to be returned with all responses. An equivalent "Expires" header is
# also generated. The default is no Cache-Control or Expires headers, which is
# just fine for most sites.
#max_age=
```

9.6.6 /etc/network/interfaces

Damit das eingebettete System Zugang zu einem Netzwerk erhält, muss über die Datei "/etc/network/interfaces" noch die Konfiguration der Netzwerkschnittstellen vorgenommen werden.

Die Konfiguration des sogenannten "loopback"-Gerätes ist hierbei immer zwingend vorzunehmen, auch wenn keine echte physikalische Anschlussmöglichkeit für ein Netzwerk am eingebetteten System vorhanden ist.

Für den Fall unseres eingebetteten Systems bietet es sich an, die erste Netzwerkschnittstelle "eth0" mittels DHCP konfigurieren zu lassen. Natürlich wäre es auch möglich hier bereits feste Vorgaben vorzunehmen.

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

9.6.7 /etc/rc.local

Da sich einige Skripte aus der Embedded Debian Distribution darauf verlassen, dass bestimmte Verzeichnisse und/oder Dateien existieren, ist es notwendig, diese nach Abschluss der grundlegenden Initialisierung des Systems manuell anzulegen. Bei einem "normalen" Debian-System, würden diese Verzeichnisse und Dateien einen Neustart im persistenten Speicher der Festplatte überdauern. Da aber das eingebettete System keinen solchen persistenten Speicher besitzt, wird es notwendig diese manuell anzulegen.

Das Verzeichnis "/var/log/apt" ist notwendig, damit die Werkzeuge zur Paketverwaltung funktionieren. Diese erwarten dieses Verzeichnis, um dort ihre Protokolldateien abzulegen.

Das Verzeichnis "/var/log/fsck" und die Datei "/var/log/dmesg" erfüllen keinen praktischen Zweck. Ihre Existenz verhindert jedoch, dass die Startskripte eine Fehlermeldung auf der Konsole während des Startvorgangs ausgeben. Um dies zu verhindern, was keinen funktionalen sondern nur rein optischen Gesichtspunkten dient, werden diese ebenfalls bei jedem Start von Neuem angelegt.

```
#!/bin/sh -e
#
# rc.local
#
# This script is executed at the end of each multiuser runlevel.
# Make sure that the script will "exit 0" on success or any other
# value on error.
#
# In order to enable or disable this script just change the execution
# bits.
#
# By default this script does nothing.
```

```
# /var/log
mkdir -p /var/log/apt
mkdir -p /var/log/fsck
touch /var/log/dmesg

exit 0
```

9.6.8 /etc/init.d/embedded

In einem normalen Debian-System werden die Dateisysteme außer dem Root-Dateisystem sehr spät im Startprozess aktiviert und gemountet. Es wird davon ausgegangen, dass das Root-Dateisystem bereits vom Kernel selbst gemountet wurde, und auch beschreibbar ist. Dies ist jedoch hier beim eingebetteten System mit seinem Nur-Lesen-Root-Dateisystem nicht gegeben. Deshalb ist es notwendig die Schritte zur Aktivierung des Pseudo-Schreibzugriffs, möglichst früh im Startvorgang vorzunehmen, so dass Skripte und Programme, die sich darauf verlassen ein beschreibbares Root-Dateisystem vorzufinden, keine Probleme bekommen.

Zunächst ist es notwendig, das "tmpfs" unter dem Verzeichnis "/tmp" einzuhängen, und die Zielverzeichnisse für das spätere Verknüpfen mit dem Nur-Lese-Root-Dateisystem zu erstellen.

War dies erfolgreich, kann nun das Root-Dateisystem unter "/etc" und das "tmpfs" unter "/tmp/etc" mittels "aufs" verknüpft werden, und wieder unter "/etc" zur Verfügung gestellt werden. Dies geschieht mittels des Befehls "mount -n -t aufs -o br:/tmp/etc:/etc none /etc".

Das selbe kann nun auch für das Verzeichnis "/var" vorgenommen werden.

```
#!/bin/sh

echo -n "Mounting /tmp (tmpfs) ... "
if mount -n -t tmpfs -o mode=1777 tmpfs /tmp
then
  mkdir -m 0755 /tmp/etc
  mkdir -m 0755 /tmp/var
  echo "done"
else
  echo "failed"
fi

echo -n "Mounting /etc (aufs) ... "
if mount -n -t aufs -o br:/tmp/etc:/etc none /etc
then
  echo "done"
else
  echo "failed"
fi

echo -n "Mounting /var (aufs) ... "
if mount -n -t aufs -o br:/tmp/var:/var none /var
then
  touch /var/log/dmesg
  echo "done"
else
  echo "failed"
fi

exit 0
```

9.7 Multi-Cursor-MarkerXtrackT (MCMXT)

Multi-Cursor-MarkerXtrackT (MCMXT) ist ein Programm zur automatischen Vermessung von Marken über ein Videosystem. Es kann mit einer Kamera eine Reihe von Objekten schritt haltend identifizieren und diese genau vermessen. Die Objekte werden hierbei mit sogenannten Markern versehen, welche eine eindeutige Identifizierung des Objekts ermöglichen. Auch in natürlicher störbehafteter Umgebung hat sich das Verfahren hervorragend bewährt. Die Messmarken identifizieren sich mittels eines ringförmig angeordneten Barcodes. Es können bis zu 100 verschiedene Marker gleichzeitig erkannt werden.

Die Messgenauigkeit bezüglich der Marken-Koordinaten ist besser als 0,1 Pixel, deren Drehlage wird absolut mit einer Genauigkeit besser als 2 Grad gemessen.

MCMXT arbeitet als abgeschlossene Kamera-Rechner-Einheit mit Ethernet-Schnittstelle. Eine interaktive Bedienung des Systems während des laufenden Betriebes ist nicht erforderlich. Die Applikation erhält ihre benötigten Messwerte als Client von dem MCMXT-Server über eine TCP/IP-Schnittstelle. Das System ist darüber hinaus als Linux-basiertes System über Netzwerk und Internet fern wartbar.

9.8 MCMXT Integration

Um MCMXT in das eingebettete System zu integrieren ist es zunächst notwendig, die nötigen Pakete für Embedded Debian zu erzeugen. Erfreulicherweise liegen alle hierfür notwendigen Pakete vom Entwickler bereits als Debian Pakete vor, so ist nur eine Umwandlung in Pakete für Embedded Debian, wie in Teil II beschreiben vorzunehmen.

Da hierbei allerdings auch die Paketabhängigkeiten zu beachten sind, ergab sich eine ganze Hierarchie von Paketen, die umgewandelt werden mussten:

- libdc1394-22_2.0.2-1em1_i386.deb
- libraw1394-8_1.3.0-4em1_i386.deb
- librcp-1_1.4em1_i386.deb
- libgrab-2_3.6em1_i386.deb
- libgd2-noxpm_2.0.36~rc1~dfsg-3+lenny1em1_i386.deb
- libfftpack1_1.0-1em1_i386.deb
- libmcmxt-2_5.3em1_i386.deb

- libmm14_1.4.2-3em1_i386.deb
- libu3d-2_2.6em1_i386.deb
- clig_1.9.11.1-3em1_all.deb
- libiitb-3_1.13em1_i386.deb
- libu2d-1_3.17em1_i386.deb
- libpm-1_1.5em1_i386.deb
- libpof-1_1.1em1_i386.deb
- libqc-1_1.1.4em1_i386.deb
- libsmphr-1_2.2em1_i386.deb
- mcmxt2ds-2_3.27em1_i386.deb

Im Laufe der Portierung all der soeben genannten Bibliotheken auf Embedded Debian, stellte sich heraus, dass einige Abhängigkeiten von Bibliotheken oder auch vom Hauptprogramm MCMXT falsch oder zu großzügig gewählt wurden. Erfreulicherweise war der Entwickler von MCMXT sehr offen und hilfsbereit, so dass die notwendigen Änderungen und Anpassungen zeitnah vorgenommen wurden.

Nachdem all diese Pakete auf Embedded Debian portiert sind, ist es noch nötig diese, wie alle anderen Pakete bisher auch, dem Paket-Repository hinzuzufügen, so dass sie den Paketverwaltungs-Werkzeugen des eingebetteten System zur Verfügung stehen.

9.9 Firmware erstellen

Noch befindet sich die Firmware nur als Dateien in einem Verzeichnis unter "/tmp". Um hieraus nun ein komprimiertes Firmware-Image zu erzeugen, das zusammen mit dem zuvor erstellten Linux-Kernel das System des eingebetteten Systems darstellt, findet das Werkzeug "mksquashfs" aus dem Programmpaket "squashfs-tools" Verwendung.

```
mksquashfs source1 source2 ... dest [options] [-e list of exclude dirs/files]
```

Der konkrete Befehl zum Erstellen des Firmware-Images gestaltet sich wie folgt:

```
root@excelsior:/tmp# mksquashfs /tmp/image.10198 image
Parallel mksquashfs: Using 1 processor
Creating 4.0 filesystem on image, block size 131072.
[=============================================================\] 7804/7804 100%
Exportable Squashfs 4.0 filesystem, data block size 131072
        compressed data, compressed metadata, compressed fragments
        duplicates are removed
Filesystem size 48660.07 Kbytes (47.52 Mbytes)
        37.54% of uncompressed filesystem size (129634.77 Kbytes)
Inode table size 92388 bytes (90.22 Kbytes)
        30.36% of uncompressed inode table size (304317 bytes)
Directory table size 90660 bytes (88.54 Kbytes)
        49.88% of uncompressed directory table size (181766 bytes)
Number of duplicate files found 742 Number of inodes 9029
Number of files 7342 Number of fragments 525
Number of symbolic links  536
Number of device nodes 80
Number of fifo nodes 2
Number of socket nodes 0
Number of directories 1069
Number of ids (unique uids + gids) 14
Number of uids 3
        root (0)
        man (6)
        libuuid (100)
Number of gids 13
        root (0)
        video (44)
        audio (29)
        tty (5)
        kmem (15)
        disk (6)
        adm (4)
        shadow (42)
        crontab (102)
        staff (50)
        libuuid (101)
        mail (8)
        utmp (43)
```

Nun wurde unter dem Pfad "/tmp/image" ein Root-Dateisystem erstellt, welches zusammen mit dem Linux-Kernel die Firmware des eingebetteten Systems darstellt.

```
root@excelsior:/tmp# ls -a -l
insgesamt 50232
drwxrwxrwt  6 root root     4096 28. Apr 13:18 .
drwxr-xr-x 20 root root     4096 23. Mär 20:07 ..
-rwx------  1 root root 49831936 28. Apr 13:19 image
drwxr-xr-x 20 root root     4096 31. Mär 12:09 image.10198
-rw-r--r--  1 root root  1573168 29. Mär 16:42 vmlinuz
```

Die beiden Dateien "image" und "vmlinuz" können nun auf das Speichermedium des eingebetteten Systems kopiert werden, und in den für das eingebettete System verfügbaren Bootloader integriert werden, oder wie im nächsten Abschnitt beschrieben mittels einer Emulation zunächst noch einmal getestet werden.

9.10 Firmware testen

Um die soeben erstelle Firmware nun zu testen, bietet sich das schon einige Male erwähnte QEMU an. Welches ja nicht nur, wie bisher verwendet, eine Prozessor-Emulation bereitstellt, sondern auch ein komplettes virtuelles PC-System emulieren kann.

Abbildung 19:

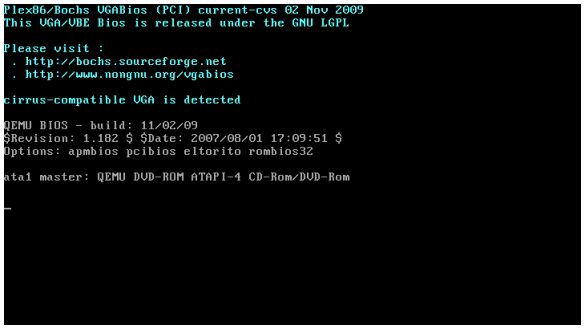
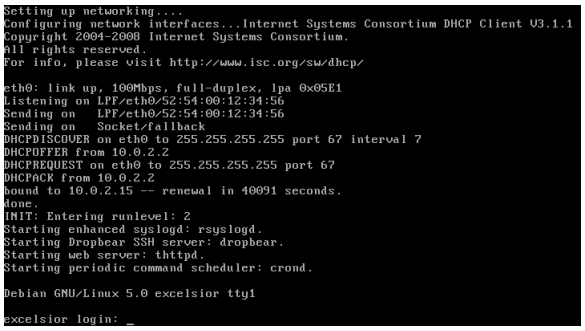


Abbildung 20:



Um nun die Firmware mittels QEMU zu testen, und auch Netzwerkzugriff bereitzustellen ist folgender Programmaufruf von Nöten:

```
florian@excelsior:~$ qemu -net nic -net user -kernel vmlinuz -appen
d "root=/dev/ram ramdisk_size=51200" -initrd image
```

Nachdem QEMU wie in der Abbildung auf dieser Seite gestartet ist, die angegebenen Dateien “vmlinuz” und “image” geladen hat, und das eigentliche Linux-System gestartet hat, präsentiert es sich ohne Fehlermeldung dem Benutzer, und erlaubt diesem einen Login auf dem System, wie auf dieser Seite dargestellt.

Die Firmware wurde also korrekt erstellt und funktioniert einwandfrei ohne jegliche Fehlermeldung. Selbst der Netzwerkzugriff ist bereits möglich, wie in der Abbildung auf der nächsten Seite anschaulich dargestellt.

9.11 Automatisierung

Um die einzelnen Schritte der Firmware-Erstellung zu automatisieren, ist es geboten ein kleines Programm zu schreiben, das diese einzelnen Schritte zusammenfasst und automatisiert abarbeitet.

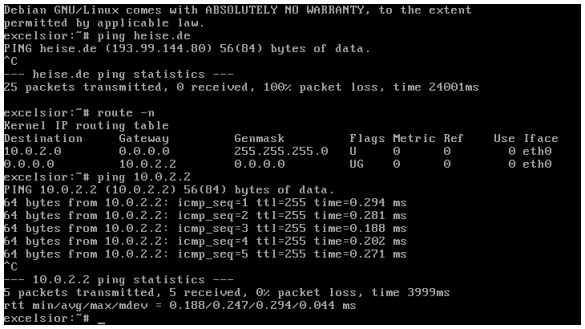


Abbildung 21:

Um die Firmware-Erstellung zu steuern, wird hierbei ein sogennantes Profil-Verzeichnis eingesetzt. Dies ist ein Verzeichnis, das dem eigentlichen Erstellungs-Werkzeug ("genemdebian") als Parameter zu übergeben ist, und alle notwendigen Einstellungen beinhaltet, um flexibel aber dennoch automatisiert eine Firmware für ein eingebettetes System zu erzeugen.

Das Profil-Verzeichnis, hat folgenden Inhalt:

```
root@excelsior:~# ls emprofile-mcmxt -a -l
insgesamt 20 drwxr-xr-x 3 root root 4096 31. Mär 14:55 .
drwx------ 8 root root 4096 26. Apr 12:57 ..
-rw-r--r-- 1 root root  224 29. Mär 17:08 config
drwxr-xr-x 4 root root 4096 31. Mär 15:01 files
-rw-r--r-- 1 root root   98 31. Mär 14:55 packages.list
```

Die einzelnen Dateien und Verzeichnis stellen hierbei unterschiedliche Informationen zur Verfügung, die im Laufe des Erstellungsprozesses benötigt werden.

9.11.1 config

Die Konfigurationsdatei "config" setzt wichtige Variablen für die Erstellung der Firmware des eingebetteten Systems. Diese sind:

- ARCH - die Prozessorarchitektur des zu erzeugenden Systems (i386)
- DIST - Name der zu verwendenden Distribution (lenny)
- SOURCE - Pfad zum Paket-Repository für das Basis-System ("grip"-Repository)
- BOOTSTRAP_EXCLUDE - Auflistung der aus dem Basissystem zu entfernenden Pakete
- BOOTSTRAP_INCLUDE - Auflistung der zusätzlich zu installierenden Basispakete

- ROOTPW - Angabe des Root-Passwortes des eingebetteten Systems.

Diese Variablen und Angaben werden an vielfältigen Stellen im Programm "genendebian" verwendet und werden in dieses zur Laufzeit importiert.

```
#!/bin/sh

ENAME="mcmxt"

ARCH="i386"
DIST="lenny"
SOURCE="http://www.emdebian.org/grip/"

BOOTSTRAP_EXCLUDE="aptitude,dmidecode,iptables,tasksel,tasksel-data,vim-common,v
im-tiny"
BOOTSTRAP_INCLUDE="less"

ROOTPW="hallo123"
```

9.11.2 packages.list

Über die Datei "packages.list" wird eine Liste von Paketen angegeben, die nach der Erstellung des Basis-Systems noch zusätzlich installiert werden sollen. Die Pakete werden alle unter Berücksichtigung der in den Paketen selbst vorgegebenen Abhängigkeiten installiert. Sollte ein weiteres Paket nötig werden, so kann dieses einfach an das Ende der Datei "packages.list" hinzugefügt werden, die Reihenfolge, in der die Pakete in der Datei stehen, hat hierbei keinerlei Einfluß.

```
linux-image-2.6.33.1-mcmxt
aufs-tools
dropbear
mtools
tcl8.4
tcllib
thttpd
ghostscript
mcmxt2ds-2
```

9.11.3 files

Das Verzeichnis "files" beinhaltet alle Dateien und Konfigurationsdateien, die nach der Installation des Basis-Systems und der zusätzlich über die "packages.list" angeforderten Pakete, noch der Firmware hinzugefügt werden sollen. Hier können vor dem eigentlichen Erstellen der Firmware-Datei noch letzte, beliebige Änderungen an Konfigurationsdateien und auch dem gesamten System vorgenommen werden.

Der Inhalt des Verzeichnisses, der zur Erstellung der Firmware im Rahmen dieser Arbeit genutzt wurde, und weiter oben bereits ausführlich erklärt wurde, lautet:

```
./etc
./etc/mtools.conf
./etc/apt
./etc/apt/sources.list.d
./etc/apt/sources.list.d/fhg.sources.list
./etc/default
./etc/default/rcS
./etc/rcS.d
./etc/rcS.d/S02embedded
./etc/hostname
./etc/thttpd
./etc/thttpd/thttpd.conf
./etc/network
./etc/network/interfaces
./etc/rc.local
./etc/init.d
./etc/init.d/embedded
./usr
./usr/share
./usr/share/tcltk
./usr/share/tcltk/clig
```

9.11.4 genemdebian

Das eigentliche Programm zur Erstellung der Firmware, setzt nun aus den oben genannten Konfigurationsdateien und Einzelschritten automatisiert eine den Vorgaben entsprechende Firmware für ein eingebettetes System zusammen. Das Programm akzeptiert hierbei zwei Parameter:

- -p <path>: Angabe des Profilverzeichnisses
- -v: Ausgabe von erweiterten Statusinformationen

```
#!/bin/bash

NAME="genemdebian"
VERSION="0.9"

VERBOSE="0"
PID=$$

function gethelp () {
  echo "${NAME} version ${VERSION}"
  echo
  echo "Usage:"
  echo "${NAME} -p <path to profile>"
  echo
  echo "Options:"
  echo " -p PATH: set profile path (no default)"
  echo " -v: set verbose mode"
  echo
```

```
  exit 2
}

ARCH="i386"
while getopts 'p:hv?' OPTION
do
  case $OPTION in
        p)      PROFILE_PATH="$OPTARG"
                ;;
        h)      gethelp
                ;;
        v)      VERBOSE="1"
                ;;
        ?)      gethelp
                ;;
  esac
done

if [ -d ${PROFILE_PATH} ]
then
  if [ -f ${PROFILE_PATH}/config ]
  then
    . ${PROFILE_PATH}/config
  else
    echo "${PROFILE_PATH}/config not found."
    exit 1
  fi
else
  echo "${PROFILE_PATH} not found."
  exit 1
fi

echo -n "Generate base system..."
if [ ${VERBOSE} = "1" ]
then
  echo
  debootstrap --exclude=${BOOTSTRAP_EXCLUDE} --include=${BOOTSTRAP_INCLUDE} \
  ${DIST} /tmp/image.$PID ${SOURCE}
  echo
else
  debootstrap --exclude=${BOOTSTRAP_EXCLUDE} --include=${BOOTSTRAP_INCLUDE} \
  ${DIST} /tmp/image.$PID ${SOURCE} &> /dev/null
  if [ $? -eq 0 ]
  then
    echo " OK."
  else
    echo " FAILED."
    exit 1
  fi
fi

if [ -f ${PROFILE_PATH}/packages.list ]
then
  ADD_PACKAGES=$(cat ${PROFILE_PATH}/packages.list)
  cp -R ${PROFILE_PATH}/files/etc/apt/sources.list.d/* \
  /tmp/image.$PID/etc/apt/sources.list.d/
  echo -n "Install additional packages..."
  if [ ${VERBOSE} = "1" ]
  then
    echo
    chroot /tmp/image.$PID apt-get -y update
    chroot /tmp/image.$PID apt-get -y --force-yes install $ADD_PACKAGES
    chroot /tmp/image.$PID apt-get -y clean
    echo
  else
    chroot /tmp/image.$PID apt-get -y update &> /dev/null && chroot \
    /tmp/image.$PID apt-get -y --force-yes install $ADD_PACKAGES \
    &> /dev/null && chroot /tmp/image.$PID apt-get -y clean &> /dev/null
    if [ $? -eq 0 ]
    then
      echo " OK."
    else
      echo " FAILED."
      exit 1
    fi
  fi
fi

if [ -d ${PROFILE_PATH}/files ]
then
  echo -n "Install additional files..."
  if [ ${VERBOSE} = "1" ]
  then
    echo
```

```
    cp -vR ${PROFILE_PATH}/files/* /tmp/image.$PID
    echo
  else
    cp -R ${PROFILE_PATH}/files/* /tmp/image.$PID
    if [ $? -eq 0 ]
    then
      echo " OK."
    else
      echo " FAILED."
      exit 1
    fi
  fi
fi

if [ "$ROOTPW" != "" ]
then
  echo -n "Setting new root password..."
  echo "root:$ROOTPW" | chroot /tmp/image.$PID chpasswd -m
  if [ $? -eq 0 ]
  then
    echo " OK."
  else
    echo " FAILED."
    exit 1
  fi
  rm -Rf /tmp/image.$PID/etc/*-
fi

rm -Rf /tmp/image.$PID/dev/.udev
rm -Rf /tmp/image.$PID/var/log/*
rm -Rf /tmp/image.$PID/var/tmp
ln -s /tmp /tmp/image.$PID/var/tmp

mv /tmp/image.$PID/boot/vmlinuz* /tmp/vmlinuz
mksquashfs /tmp/image.$PID /tmp/image

exit 0
```

Wird das Programm "genemdebian" nun mit dem genannten Profil-Verzeichnis gestartet, gestaltet sich der Aufruf wie folgt:

```
root@excelsior:~# ./genemdebian -p /root/emprofile-mcmxt
Generate base system... OK.
Install additional packages... OK.
Install additional files... OK.
Setting new root password... OK.
Parallel mksquashfs: Using 1 processor
Creating 4.0 filesystem on /tmp/image, block size 131072.
[==========================================================/] 7801/7801 100%
Exportable Squashfs 4.0 filesystem, data block size 131072
        compressed data, compressed metadata, compressed fragments
        duplicates are removed Filesystem size 48514.00 Kbytes (47.38 Mbytes)
        37.48% of uncompressed filesystem size (129445.23 Kbytes)
Inode table size 92526 bytes (90.36 Kbytes)
        30.41% of uncompressed inode table size (304245 bytes)
Directory table size 90701 bytes (88.58 Kbytes)
        49.91% of uncompressed directory table size (181722 bytes)
Number of duplicate files found 741
Number of inodes 9027 Number of files 7340
Number of fragments 525
Number of symbolic links 536
Number of device nodes 80
Number of fifo nodes 2
Number of socket nodes 0
Number of directories 1069
Number of ids (unique uids + gids) 14
Number of uids 3
        root (0)
        man (6)
        libuuid (100)
Number of gids 13
        root (0)
        video (44)
        audio (29)
        tty (5)
        kmem (15)
        disk (6)
        adm (4)
        shadow (42)
```

```
crontab (102)
staff (50)
libuuid (101)
mail (8)
utmp (43)
```

Die Firmware wurde nun nach den Vorgaben aus dem Profil-Verzeichnis erzeugt, wie sich im Verzeichnis "/tmp" sehen lässt:

```
root@excelsior:/tmp# ls -a -l
insgesamt 50080
drwxrwxrwt  5 root root     4096 28. Apr 14:25 .
drwxr-xr-x 20 root root     4096 23. Mär 20:07 ..
drwxrwxrwt  2 root root     4096 24. Mär 12:12 .ICE-unix
-rwx------  1 root root 49680384 28. Apr 14:25 image
drwxr-xr-x 20 root root     4096 28. Apr 14:19 image.16226
-rw-r--r--  1 root root  1573168 29. Mär 16:42 vmlinuz
drwxrwxrwt  2 root root     4096 24. Mär 12:12 .X11-unix
```

Abbildungsverzeichnis