OpenMoko

Schmidbauer, Mechtler, Täubl

17. Februar 2008

Zusammenfassung

Dieses Whitepaper soll eine kurze Einführung in das OpenMoko Projekt bieten. OpenMoko hat zum Ziel eine im Quellcode offene und frei zugängliche Softwareplattform für Mobiltelefonie zu entwickeln.

Kurzfassung

OpenMoko ist eine Linux Distribution die für Mobilfunkplattformen optimiert ist. Durch die vollständige Verfügbarkeit des Quellcodes, unterliegen die Softwareentwickler keiner Einschränkung in ihrer Innovationskraft. Die Verfügbarkeit des Quellcodes ermöglicht auch eine schnelle Portierung auf unterschiedliche Hardwareplattformen. Zu dem OpenMoko Projekt gehört auch eine Referenzhardwareimplementierung, bei welcher für alle Hardwarekomponenten eine öffentliche Dokumentation zur Verfügung steht.

Inhaltsverzeichnis

1 Das OpenMoko Projekt

FIC ist ein taiwanesischer OEM Hersteller von Notebooks und Desktops. Durch ihre Tocherfirma OpenMoko Inc. wollen sie nun in den stark wachsenden Bereich der mobilen Computer vordringen. Da ihr Fokus in der Herstellung und dem Vertrieb von Hardware liegt, haben sie sich entschlossen im Softwarebereich auf OpenSource zu setzen. Zu diesem Zweck sponsert die Firma OpenMoko Inc. die Entwicklergemeinde um OpenMoko.org.

Abbildung 1: OpenMoko Projekt

Die erste Version des Mobiltelefons wurde unter dem Namen Neo 1973 Anfang 2006 auf den Markt gebracht. Diese richtete sich haupstächlich an Entwickler und Enthusiasten, da die Software noch nicht die für einen Normalbenutzer notwendige Reife hatte.

Zur Consumer Electronic Show (CES) 2008 wurde die zweite Version der Hardware unter dem Namen Neo FreeRunner vorgestellt. Diese zeichnet sich gegenüber dem Vorgänger unter anderem durch leistungsfähigere CPU und Grafik sowie einen integrierten WLAN Chip aus. Dieses Gerät ist erstmals für den Endverbrauchermarkt konzipiert und ist die Hardwareplattform für zahlreiche Applikationen des OpenMoko Softwarestacks.

Durch die Förderung der Entwicklergemeinde sollen eine Vielzahl von Applikationen entstehen, die den Neo FreeRunner zu einem attraktiven Mobiltelefon machen. Durch die Kommunikationmöglichkeiten über GSM und WLAN, den integrierten GPS Empfänger und die vollständige Dokumentation der Hardwarekomponenten sind der Innovationskraft der Softwareentwickler keine Grenzen gesetzt.

1.1 Andere Softwareplattformen für Mobiltelefone

Durch die ständig steigende Leistungsfähigkeit der Mobiltelefonhardware und den attraktiven Einsatzmöglichkeiten die sich durch die Netzwerkfähigkeit ergeben, sind eine Vielzahl von Mitbewerbern in den Markt für Mobiltelefone eingetreten. Nachfolgend sollen die derzeit wichtigsten Plattformen kurz vorgestellt werden.

1.1.1 Qtopia

Qtopia ist eine Applikationsplattform für Linux-basierende Mobiltelefone und Embedded Devices der norwegischen Firma Trolltech. Trolltech ist am besten für ihr Qt-GUI Framework bekannt, welches die Grundlage für den KDE-Desktop bildet. Qtopia wird unter einer Dual-License vertrieben, welche die Wahl zwischen der GPL oder einen kommerziellen Lizenz ermöglicht. Da die GPL Lizenz dazu verpflichtet den Sourcecode der Applikationen zu veröffentlichen, wird die kommerzielle Version von vielen proprietären Softwaranbietern genutzt. Derzeit gibt es 11 Mobiltelefone, darunter Modelle von Motorola und Panasonic, welche auf Qtopia aufsetzen. Qtopia besteht aus einem Linux Kernel und Software Bibliotheken die die wichtigsten Funktionen eines Mobiltelefons unterstützen.

1.1.2 Android

Android ist eine Softwareplattform der Open Handset Alliance, in der Google eine führende Rolle einnimmt. Der gesamte Quellcode des Projektes wird unter der Apache Lizenz veröffentlicht. Die Plattform setzt auf den Linux Kernel auf und stellt eine Reihe von Bibliotheken zur Verfügung. Die Basisbibliothek wird als jar-File ausgeliefert, somit ist die bevorzugte Programmiersprache für Applikationen Java. Jedes dieser Java Programme wird in einer eigenen Java-VM, der Dalvik-VM, ausgeführt. Derzeit sind erst einige

Abbildung 2: Softwareplattformen für Mobiltelefone

Teile des Softwarestacks Source Codes veröffentlicht worden. Zu beachten ist das der Bytecode der Dalvik-VM und der Java-VM nicht kompatibel sind.

1.1.3 Maemo

Maemo ist eine von Nokia vorangetriebene Plattform für Handheld-Devices. Derzeit basieren die Nokia Internet Tablets N770 und N810 auf diesem Framework. Es besteht aus einem Linux Kernel und Teilen des Gnome Desktops. Die Internet Tablets verfügen derzeit über keine Telefoniefunktion, Geräte mit dieser Fähigkeit sind aber bereits in Planung. Maemo wird unter forum.nokia.com bereits als eingenständige Entwicklerplattform geführt.

1.1.4 Windows Mobile

Windows Mobile ist die Bezeichung für das Betriebsystem und die Basisapplikationen der Firma Microsoft für Mobiltelefone, PDA's und Embedded Devices. Die derzeit aktuelle Version ist Windows Mobile 6, die in Standard, Classic und Professional Abstufungen angeboten wird. Dabei wird bereits das .NET Compact Framework unterstützt und der Look and Feel von Windows Vista nachgebildet.

1.1.5 Symbian OS (UIQ)

Symbian ist ein proprietäres Betriebssystem und Applikationsframework der Firma Symbian. Symbian ist ein Gemeinschaftsunternehmen von Nokia, Ericsson, SonyEricsson, Panasonic, Siemens und Samsung. Symbian hat derzeit einen Marktanteil von über 60% bei Mobiltelefonen. Für Geräte mit Touchscreen werden die UIQ Bibliotheken verwendet. Diese stellen für Touchscreens optimierte Applikationen und Eingabemethoden zur Verfügung, welche auf den Symbian Betriebssystem aufsetzen.

1.1.6 iPhone

Das iPhone ist ein Mobiltelefon der Firma Apple, welches als Betriebssystem eine vereinfachte Version des Apple Betriebssystems OS X verwendet. Diese Plattform ist komplett proprietär und Applikationen können nur mit Einverständnis von Apple auf dem iPhone installiert werden. Apple hat versprochen in Zukunft ein Software Development Kit für Fremdanbieter von Software zu vermarkten.

2 OpenMoko Hardware Plattform

Die OpenMoko Hardwareplattform wird von der Firma OpenMoko Inc. entwickelt und vertrieben. Bei der Auswahl der Systemkomponenten wird darauf geachtet nur Bauteile mit öffentlich verfügbarer Dokumentation zu verwenden, um allen Entwicklern die Möglichkeit zu geben sämtliche Funktionen der Hardware benutzen zu können.

- Neo 1973 (GTA01)

  Der Neo 1973 ist intern unter dem Name GTA01 bekannt. Er war die erste Version der Hardware, welche über den Webshop an Entwickler und Enthusiasten verkauft wurde. Da der OpenMoko Softwarestack am Anfang noch einige Stabilitätsprobleme hatte, wurde Endusern vom Kauf dieser Hardware abgeraten. Das Gerät verfügt bereits über einen GPS Empfänger, aber keine WLAN Funktionalität.

  Funktionsumfang:

  - Touchscreen 2.8 Zoll (480x640)

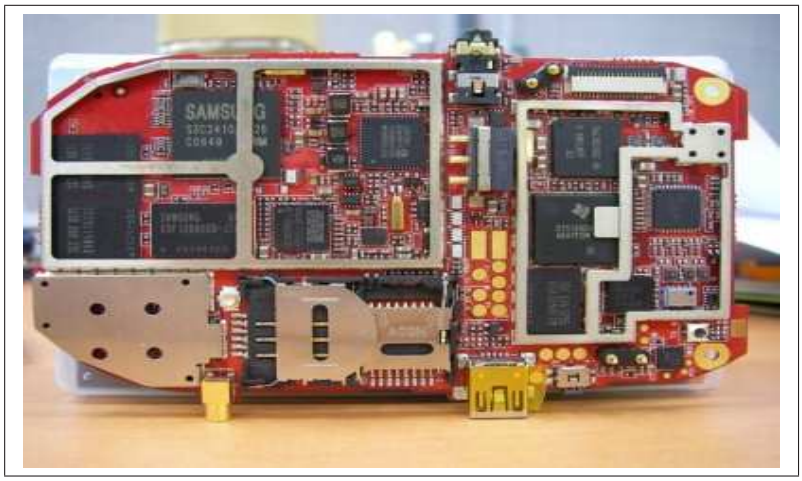

Abbildung 3: FIC Neo1973 GTA01

  - 266MHz Samsung S3C2410AL-26 128MB SDRAM
  - 64MB Flashspeicher
  - GSM, GPRS
  - USB 1.1
  - Bluetooth 2.0
  - Assisted GPS

- Neo FreeRunner (GTA02)

  Mit dem Neo FreeRunner wird erstmals der Enduser- und Massenmarkt angesprochen. Neben einer schnelleren 400 MHz CPU und einem auf 256 MB erweiterten Flashspeicher verfügt das Gerät nun auch über einen WLAN Chip. Diese Funktionalität ist besonders für Voice over IP und Webapplikationen äußerst wichtig. Durch die Verwendung eines Grafikchips mit Hardwarebeschleunigung können ansprechendere grafische Anwendungen geschaffen werden. Mit Hilfe der integrierten Beschleunigungssensoren kann mit alternativen Eingabemöglichkeiten expermentiert werden.

  Funktionsumfang:

  - Touchscreen 2.8 Zoll (480x640)
  - SMedia 2D/3D Grafikbeschleuniger
  - 400MHZ Samsung 2442 128MB SDRAM
  - 256MB Flashspeicher

- WLAN Atheros AR6K, 802.11 b/g
- Bluetooth 2.0
- Assisted GPS
- Beschleunigungssensoren
- Akkuaufladung über USB-Kabel

Die Samsung CPU implementiert die ARM Architektur und hat zahlreiche Funktionen wie USB-Controller, ADC-, TFT- und SD Controller sowie einen NAND Flash Controller auf einen Chip integriert. Eine vieldiskutierte Erweiterung für nächste Hardwareversionen wird ein Multitouch-fähiges Display sein, wie sie durch Apples Iphone populär wurden.

2.1 Flash Layout

Die Informationen zum Flash Layout und NAND Flash wurden dem OpenMoko Wiki entnommen (`http://wiki.openmoko.org/wiki/NAND`).

OpenMoko verwendet NAND Flash um den Bootloader, den Linux Kernel und das Root Dateisystem zu speichern. NAND Flash hat eine Reihe von Eigenschaften auf die die Autoren kurz eingehen möchten.

Zuerst fällt auf, dass wenn der NAND Speicher gelöscht werden soll, alle Bits auf 1 gesetzt werden. Einzelne Bits können danach wieder auf 0 zurückgesetzt werden. Das Löschen einzelner Bits (setzen auf 1) ist **nicht** möglich. Es müssen ganze Blöcke, sogenannte `Erase Blocks` gelöscht werden. Wie oft ein `Erase Block` gelöscht werden kann ist begrenzt (bis zu 100.000 mal). Wird diese überschritten wird der gesamte Block als `Bad Block` markiert. Eine `Erase Block` besteht aus mehreren NAND Pages.

Ein sogenannte `NAND Page` besteht aus einer Anzahl von Bytes die für Daten verwendet werden können (z.b 512 oder 2048) plus einer bestimmten Anzahl von OOB (Out of Band) Bytes. Die Daten selbst werden **nicht** in den OOB Bytes abgelegt. Diese OOB Bytes werden dazu verwendet um

- defekte Blöcke zu markieren
- um ECC's abzulegen
- und um spezifische Filesystemdaten zu speichern (JFFS2)

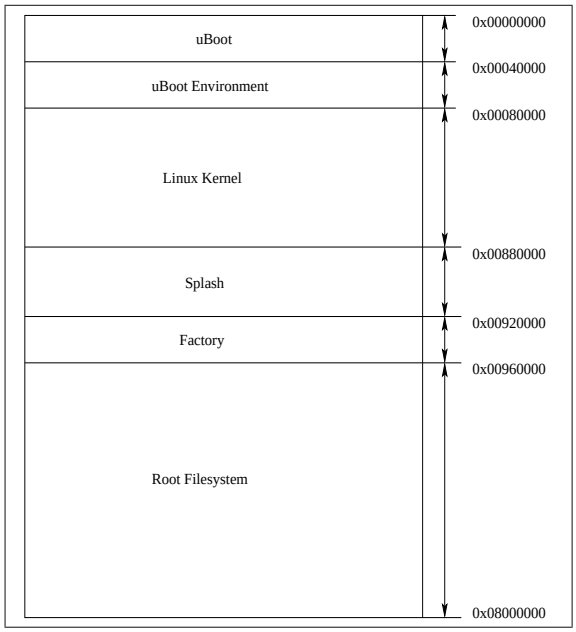

Abbildung 4: Flash Layout

In Abbildung 4 ist die Partitionierung des NAND Flashes dargestellt. In der ersten Partition von Adresse `0x00000000` bis `0x00040000` befindet sich der uBoot Bootloader. Dieser ist für das Laden des Kernels verantwortlich. Da der Bootloader einigen Einstellungsmöglichkeiten bietet, werden dies im sogenannten uBoot Environment abgelegt. Danach folgt der eigentliche Linux Kernel im Adressraum von `0x00080000` bis `0x00088000`. Der Kernel wird von uBoot geladen und zur Ausführung gebracht. Als naechstes ist das Splash Image abgelegt, welches beim Booten angezeigt wird. Der Bereich Factory ist den Autoren leider noch ein Rätsel und bedarf noch Nachforschung. Zum Schluss ist noch das Root Dateisystem gespeichert welches der Linux Kernel mounted und in dem sich alle OpenMoko Applikationen befinden.

2.2 Boot Ablauf

Die OpenMoko Hardware verwendet eine Samsung S3C2410X CPU. Diese CPU bietet die Möglichkeit direkt von NAND Flash zu booten.

Der Bootablauf ist schematisch in Abbildung 5 dargestellt. Beim Starten der CPU werden die ersten 4 Kilobyte des NAND Flashes in eine gleich grossen SRAM Buffer geladen. Diese 4 Kilobyte werden danach von der CPU ausgeführt. Somit dienen die ersten 4 Kilobytes des NAND Flashes als Stage

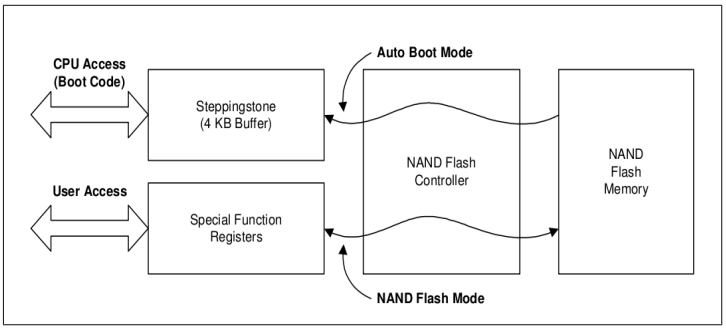

Abbildung 5: Steppingstone [Ele03]

1 Bootloader. Dieser kann dann etwaige weiter Bootloader nachladen. Im Fall von OpenMoko wird als Stage 2 Bootloader uBoot verwendet. uBoot ist dann wie in 2.1 beschrieben für das Laden des Linux Kernels verantwortlich. Des weiteren bietet uBoot auch die Möglichkeit das NAND Flash neu zu beschreiben. Es kann somit mit Hilfe von uBoot auch das Root Dateisystem oder der Linux Kernel neu geschrieben werden.

2.3 JFFS2

`JFFS2` ist ein von RedHat entwickeltes Dateisystem das speziell für den Einsatz auf NAND-Flash Speicher entwickelt wurde. Es ist eine Weiterentwicklung des von Axis Communications AB entwickelten `JFFS`.

Ursprünglich wurde ein sogenannter Flash Translation Layer (FTL) bzw. NAND Flash Translation Layer (NFTL) zum Speichern von Daten auf Flash Chips verwendet (siehe auch [Int98]. Der FTL emuliert dabei ein Standard Block Device auf dem das Filesystem abgelegt wird. In UNIX wird zwischen Block und Character Devices unterschieden. Auf Block Devices wird immer blockweise, z.b. in 512 Byte schritten, zugegriffen. Character Devices unterscheiden sich dadurch in dem sie einen byteweisen Zugriff ermöglichen. Der FTL versteckt daher die Eigenheiten des Flash Speicher (z.b. Löschen nur Blockweise siehe 2.1) und ermöglicht es daher Standard Dateisystem wie NTFS oder EXT3 zu verwenden.

Da dies aber nicht die optimalste Zugriffsform darstellt, wurde ein eigenes Dateisystem, welches auf die Eigenheiten des Flash Speicher rücksicht nimmt entwickelt. JFFS2 ist ein sogenanntes Log-Structured Filesystem (LFS) . Dabei werden alle Daten des Dateisystems sequentiell auf die Platte geschrieben. Dies entspricht der Struktur einer Log-Datei. Mehr dazu ist unter [MR91] zu

finden.

Da das orignale JFFS einige Nachteile aufwies wurde von RedHat eine zweite Version entwickelt. Verbessert wurden vorallem die Garbage Collection (kümmert sich um das Löschen von alten Log einträgen). Des weiteren unterstützt JFFS2 die Kompression von Dateisystemdaten. JFFS hatte auch keine Unterstützung von UNIX Hard Links, diese sind unter JFFS2 ebenfalls möglich. Mehr zum Thema JFFS2 ist in [Woo] beschrieben.

3 OpenMoko Software Platform

Die OpenMoko Software Plattform besteht aus vier funktional gegliederten Blöcken. Der erste Block ist für der Verwaltung der Hardware und das Powermanagement zuständig. Diese Aufgaben werden durch einen Linux 2.6 Kernel erledigt, der um einige spezielle Treiber für die OpenMoko Hardware ergänzt wurde. Ein wichtiges Grundprinzip ist, dass für alle Treiber der Sourcecode zur Verfügung steht.

Als nächste Block wird von Standard Linuxdiensten und Programmen abgebildet. Dazu gehören die von gängigen Linuxdistributionen bekannten Dienste wie sshd, udev, d-bus und die Bluetooth Utilities. Auch das X Window System mit der Grafikbibliothek GTK ist in diesem Block beinhaltet.

Der dritte Block wird durch das OpenMoko Application Framework gebildet, welche wichtige Applikationsbibliotheken für die OpenMoko Applikation zur Verfügung stellt. Somit wird es möglich dem Benutzer ein einheitliches Look and Feel über verschiedene Applikationen zu vermitteln. In Abbildung 6 sind die verschiedenen Einheiten der Softwareplattfom grafisch dargestellt.

Als letzter Block werden die eigentlichen OpenMoko Applikationen betrachtet. Diese setzen auf den von den anderen Blöcken zur Verfügung gestellten Services auf und bieten den Benutzer umfangreiche Funktionen. Auf einige Basisapplikationen wird in 3.5 näher eingegangen.

3.1 Linux Kernel

OpenMoko versucht aktuelle Linux Kernel zu verwenden, um von den dort stattfindenten Verbesserungen, insbesondere im Bereich Powermanagement und Treiber zu profitieren. Die nächste Release wird auf den neuen 2.6.24

Abbildung 6: OpenMoko Software Architektur

Kernel aufbauen, der zum Zeitpunkt der Erstellung dieses Dokuments noch nicht freigegeben wurde. Auf den Standard Linux Kernel aufsetzend werden nur wenige zusätzliche Erweiterungen, wie für den USB Controller und das GSM Modem benötigt.

3.2 uBoot

Zum Booten des Kernels wird der U-Boot Bootloader verwendet. U-Boot steht für Universal Bootloader und wird von einer Vielzahl von Linux Projekten im Embedded Bereich eingesetzt. Es stehen Version für PowerPC, ARM, XScale, MIPS und andere CPU Architekturen zur Verfügung. Zu den Aufgaben eines Bootloaders im Embedded Bereich zählen:

- Initialisierung der Hardware
- Übergabe von Boot Parametern an den Kernel
- Starten des Kernels
- Laden von Binärdateien über Ethernet oder serielle Schnittstelle

- Kopieren von Binärdateien von RAM zu Flash Speicher

U-Boot stellt über seine Command Line Interface (CLI) eine große Anzahl an Befehlen zur Bedienung und Konfiguration zur Verfügung.

Listing 1: u-boot

```
U-Boot 1.2.0-moko1 (Feb 16 2007 - 00:36:13)

DRAM:  128 MB
NAND:  64 MiB
Found Environment offset in OOB..
Video: 640x480x8 31kHz 59Hz
USB:   S3C2410 USB Deviced
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
GTA01Bv3 #
```

Listing 1 zeigt ein typisches Kommandoprompt eines OpenMoko Mobiltelefons welches den Bootloader geladen hat.

3.3 Linux Basisdienste

Die Linux Basisdienste sind die gleichen wie in jeder modernen Linux Distribution. Über sshd ist es möglich sich in das System über Netzwerk einzuloggen, wobei moderne Verfahren zur Verschlüsselung und Authentifizierung angewandt werden. Mittels udev werden die Devicenodes dynamisch erstellt und Zugriffsrechte sowie persistente Devicename verwaltet. Unter mehreren konkurierenden Bluetooth Stacks hat sich mittlerweile der bluez Bluetooth Stack bei allen Linux Distributionen durchgesetzt. Dieser wird auch für OpenMoko verwendet und stellt neben den Kerneltreiber zahlreiche Utilities zur Verwaltung von Bluetooth Verbindungen zur Verfügung. Über d-bus können den Applikationen Nachrichten über Kernel Events mittgeteilt werden. D-bus dient aber auch als flexible Kommunikationsschnittstelle zwischen einzelnen Applikationen.

Die GSM Funktionen des Mobiltelefons werden über Standard AT-Kommandos angesprochen. Der GSM Chip emuliert eine serielle Schnittstelle, welche die AT-Kommandos entgegennimmt und ausführt.

Der GPS Empfänger implementiert das Assisted-GPS (A-GPS) Verfahren. Dieses Verfahren ermöglicht es mit Hilfe von Assistence Servern die Zeit bis zur initialen Standortbestimmung zu verringern. Ausserdem hilft dieses Verfahren bei schlechtem oder nur teilweisem Satelitenempfang, dann können Standortdaten über GPRS aus dem Internet bezogen werden.

Die Basis für das grafische Userinterface bilden KDrive, ein minimaler X11 Server und die GTK Bibliothek. Als Window Manager, der für die Verwaltung der verschiedenen Applikationswindows verantwortlich ist, wird Matchbox verwendet.

3.4 OpenMoko Application Framework

Auf den bisher vorgestellten Komponenten setzt das OpenMoko Application Framework auf. Dieses Framwork wird durch vier Bibliotheken gebildet, die von den OpenMoko Applikationen benutzt werden. In Abbildung 7 wird der Zusammenhang dargestellt.

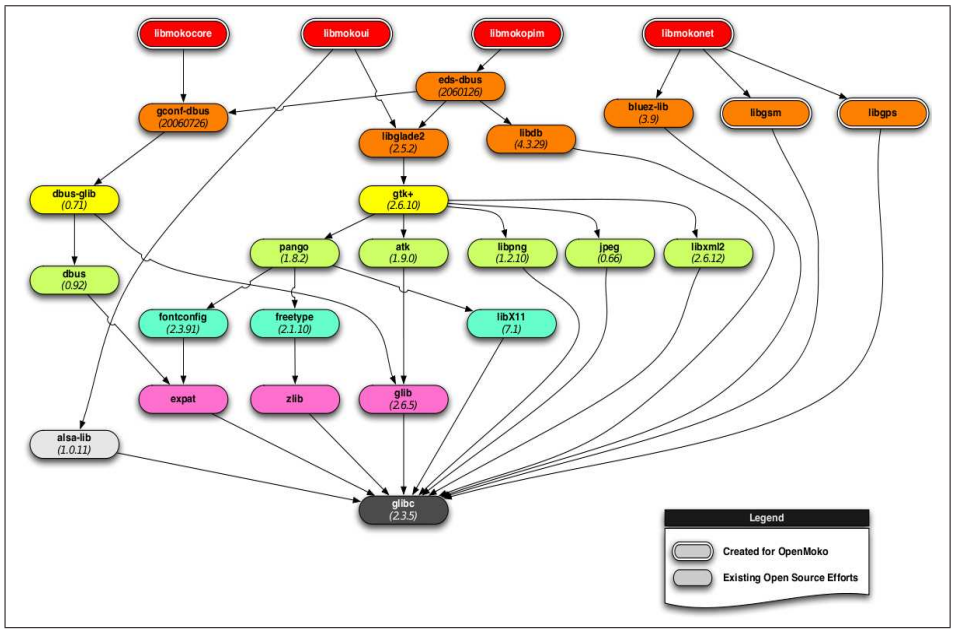

Abbildung 7: OpenMoko Application Framework

- libmokocore: Dient zum Senden von Nachrichten zwischen den Applikationen und zum Lesen und Speichern von Konfigurationsdaten.
- libmokoui: High Level API für das grafische Userinterface. Sorgt für einen einheitlichen Look and Feel der Applikationen.
- libmokopim: API zur Verwaltung des Personal Information Manager (PIM) der die Kontaktdaten speichert.

- libmokonet: Stellt eine einheitlich Schnittstelle für Netzwerkverbindungen über GPRS und Bluetooth zur Verfügung.

3.5 OpenMoko Applikationen

Applikationen können für Stift- oder Fingereingabe optimiert werden. Da der Benutzer des Mobiltelefons nicht für jede Interaktion den Stift benutzen möchte, werden einige Basisapplikationen für Fingereingabe optimiert. Für das Erstellen von Applikation für Mobiltelefone gelten eigene Styleguides die sich von klassischen Desktopapplikationen zum Teil erheblich unterscheiden. Im Folgenden werden einige der wichtigsten OpenMoko Applikationen vorgestellt.

3.5.1 Fingerbediente Applikationen

Zu den fingerbedienten Applikationen zählen unter anderem die Kontaktverwaltung, der Dialer und der Display-Locker. Dabei handelt es sich um zentrale Applikation die für die Bedienung eines Mobiltelefons unerlässlich sind. Beim Design dieser Applikation wurde darauf geachtet, dass sie über besonders grosse Bedienelemente verfügen um ein leichtes und fehlerfreies Auswählen der Bedienelemente zu ermöglichen.

In Abbildung 8 ist ein Screenshot der Kontaktverwaltung zu sehen, welche alle Informationen zu Kontakten in einer zentralen Datenbank abspeichert.

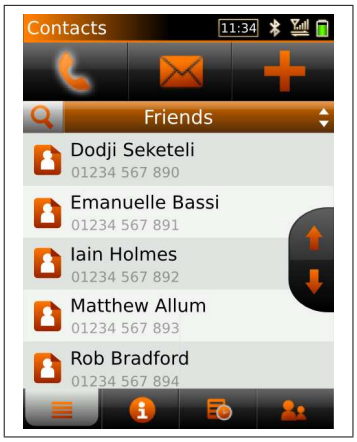

Abbildung 8: Kontaktverwaltung

In der OpenMoko Distribution befinden sich noch eine Vielzahl anderer fingerbedienter Applikation - darunter verschiedene Uhren, GPS Navigation und ein Rechner.

3.5.2 Stiftbediente Applikationen

Eine sehr wichtige stiftbediente Applikation ist die obere Menüleiste. Sie stellt auf sehr kleiner Fläche den Zustand wichtiger Subsysteme dar und ermöglicht die Auswahl und das Starten aller installierten Applikationen. In Abbildung 9 ist ein Screenshot der Menüleiste dargestellt.

Abbildung 9: Leiste

Dabei sieht man die beiden Bereiche der Menüleiste:

- 1) Hauptmenü - Dient zum Starten der Applikationen
- 2) Menüleistenapplikationen - Zeigen den Status der verschiedenen Subsysteme - wie Battery, Signalstärke, USB, WLAN und Bluetooth an.

Zu den stiftbedienten Applikation zählen weiters - Mediaplayer, Filemanager, Terminal, Webbrowser und zahlreiche andere in der OpenMoko Distribution verfügbaren Applikationen.

4 Softwareenwicklung mit OpenMoko

Durch die Verwendung von Standard Linux Komponenten ermöglicht OpenMoko eine sehr flexible Softwareentwicklung. Wenn kein Zugriff auf spezielle Hardware des Mobiltelefons notwendig ist, kann die Software auf einem normalen Linux PC entwickelt werden. Dies bietet die Vorteile einer schnellen CPU und somit kurzen Kompilierzeiten und der Möglichkeit auf seiner gewohnten Umgebung arbeiten zu können.

Um auch die niedrigere Bildschirmauflösung des Mobiltelefons am PC simulieren zu können, werden Projekte wie:

- Xoo (GTK2 Wrapper)
- Xephyr

verwendet. Beide Tools emuliern die Bildschirmauflösung des mobilen Endgeräts mit Hilfe eines "nested"X11 Servers (X11 Server der in einem anderen X11 Server läuft).

In weiterer Folge kann die Software mit einem Emulator getestet werden. Dazu wird QEMU verwendet. Mehr zu diesen Thema findet sich in Kapitel 5. QEMU kann mittlerweile die gesamte Hardware des OpenMoko Mobiltelefons emulieren, die einzige Ausnahme bildet der GPS Empfänger der derzeit noch nicht emuliert werden kann.

Als letzter und ultimativer Test von neuentwickelter Software ist eine Provisionierung auf die Zielhardware natürlich von zentraler Bedeutung. Mittels der integrierten USB Schnittstelle kann die Übertragung der Software auf das Mobiltelefon sehr schnell und bequem erfolgen.

Das folgende Kapitel soll eine kurze Einführung in die verschiedenen Tools bieten und somit einen schnelleren Einstieg in die Applikationsprogrammierung ermöglichen.

4.1 Einführung in Make und Makefiles

Make ist ein UNIX Kommando welches das Kompilieren von großen Programmen erleichtern soll. Mit der Hilfe von sogenannten Makefiles wird beschrieben, welche Teile des Programs wie zu kompilieren sind. Make kümmert sich dann automatisch um die Durchführung. Ein groszer Vorteil von Make ist, dass immer nur jene Dateien neu kompiliert werden die sich seit dem letzten Aufruf von Make geändert haben.

Es können auch Abhängigkeiten zwischen den einzelnen Dateien definiert werden. Zum Beispiel muss die Datei A neu kompiliert werden falls sich Datei B ändert.

Make ist aber nicht nur hilfreich wenn es um das kompilieren von Programmen geht, sondern viel allgemeiner wenn verschiedene Tätigkeiten ausgeführt werden müssen die in Abhähnigkeit zu einander stehen.

Hier ein kleines Beispiel das die Anwendung eines Makefiles veranschaulichen soll:

Listing 2: Makefile

```
1  FILE = moko_wp
2
3  .SUFFIXES:      .tex .dvi .ps
4
5  DVIFILE = $(FILE).dvi
6  PSFILE = $(FILE).ps
7  PDFFILE = $(FILE).pdf
8
9  TEX = latex
10 DVIPS = dvips
11 DVIOPT= -Ppdf -f
12 PS2PDF = ps2pdf14
13
14 .tex.dvi:
15         $(TEX) $<
16         $(TEX) $<
17
18 $(PSFILE): $(DVIFILE)
19         $(DVIPS) $(DVIOPT) $(DVIFILE) > $@
20
21 pdf: $(PSFILE)
22         $(PS2PDF) $(PSFILE)
```

Obiges Makefile wurde zum Erstellen dieses Dokuments verwendet. Zeile 1 weist den Dateinamen des Dokuments ohne Suffix der Varialbe `FILE` zu. Suffix bezeichnet hier die Endung einer Datei z.B. `.tex` oder `.c`. Zeile 3 definiert die für dieses Makefile gültige Suffixe. Make kennt nur eine bestimmte Anzahl von Suffixen, `.tex`, `.dvi` und `.ps` müssen erst bekannt gemacht werden. Die Zeilen 5-7 definiert weiter Variablen die weiter unten benötigt werden. Nun werden die Namen der aufzurufenden Kommandos ebenfalls Variablen zugewiesen (Zeile 9 - 12).

Zeile 14 definert ein sogenanntes `Target`. Dieses Target beschreibt wenn eine Datei mit der Endung `.dvi` erstellt werden soll, muss zuerst die Datei mit der Endung `.tex` zwei mal mit dem Kommando `latex` kompiliert werden (Zeilen 15 und 16).

Um eine Datei `$(PSFILE)` erstellen zu können (Zeile 18), muss zuerst eine Datei `$(DVIFILE)` vorhanden sein. Dieses wird durch den Befehl in Zeile 19 erzeugt.

Das Target in Zeile 21 `pdf` beschreibt wie eine PDF Datei erzeugt werden kann. Dazu muss zuerst eine Datei `$(PSFILE)` vorhanden sein. Ist dies der Fall kann durch den Befehl in der nächsten Zeile das PDF erstellt werden.

Das oben angeführte Makefile beschreibt sehr anschaulich die Abhängigkeiten zwischen den einzelnen zu erstellenden Dateien. Um eine PDF zu erzeugen, muss zuerst eine Postscript Datei erstellt werden. Ist diese nicht vorhanden, muss eine DVI Datei erzeugt werden. Ist auch diese nicht vorhanden muss aus der Latex Source eine ebensolche kompiliert werden.

Mehr Informationen zu Make findet man in [Mak07].

Das Listing 2 wird in einer Datei mit dem Namen `Makefile` abgelegt. Ist nun im selben Verzeichnis eine Datei mit dem Namen `moko_wp.tex` vorhanden kann durch den Aufruf von

```
make pdf
```

eine PDF Datei mit dem Namen `moko_wp.pdf` einfach erstellt werden. Dies geschieht allerdings nur dann, falls die vorhanden Quelldatei `moko_wp.tex` einen jüngeren Zeitstempel als die PDF Datei hat.

4.2 Einführung in Cross Compiler

Ein Cross Compiler erzeugt ausführbaren Code für eine Platform, wird dabei aber nicht auf dieser Platform ausgeführt. Zum Beispiel wird im Fall von OpenMoko die Software auf dem Notebook des Autors mit einem Intel Prozessor kompiliert, der Compiler erzeugt dabei aber ausführbaren Code der auf einer ARM Platform läuft.

Beim Cross kompilieren wird zwischen Host, Build und Target Platform unterschieden. Als Host wird die Platform verstanden auf der der Compiler selbst erstellt wird. Build bezeichnet die Platform auf der der Compiler zum Erzeugen des ausführbaren Codes aufgerufen wird. Target ist die Zielplatform auf der der ausführbare Code läuft. Ein kleines Beispiel soll dies verdeutlichen.

Nehmen wir drei Rechner A, B und C an. Alle drei System basieren auf unterschiedlichen Platformen. Auf Rechner A wird mit Hilfe des dort installieren Kompilers ein nativer Kompiler für den Rechner B erstellt. Schliezlicher wird auf System B ausführbarer Code für Rechner C erzeugt. Sind wie hier beschrieben die Platformen der drei Systeme unterschiedlich sprich man von einem `Canadian Cross`. Meistens sind die System A und B aber ident.

Mehr zum Thema Cross Compiler findet sich in [Yag03] und [vH04].

4.3 OpenEmbedded

OpenEmbedded (http://www.openembedded.org) ist ein Development Environment das speziell für Embedded Linux System entwickelt wurde. Ziel ist es dem Embedded Linux Entwickler eine Toolchain zur Verfügung zu stellen, mit der er einfach für unterschiedlichste Hardwareplatformen Softwarepakete

erstellen kann. Mit Hilfe von OpenEmbedded lassen sich auch ganze Linux Distributionen erzeugen, wie das zum Beispiel auch OpenMoko vorzeigt.

Um ein neues Paket zu erstellen braucht eine Entwickler nur eine Konfigurationsdatei zu erstellen. In dieser enthalten sind die Informationen von wo der Sourcecode für das zu erstellende Paket bezogen werden kann (z.B. download aus dem Internet), wie das Paket zu kompilieren ist (z.B. GNU Autotools oder Makefile). OpenEmbedded kümmert sich dann selbstständig um den Download, das Cross kompilieren auf die Zielplatform und das Erstellen des Paketes. Jeder dieser einzelnen Schritte wird als `Task` bezeichnet.

Die Pakete liegen nach einer erfolgreichen Erstellung entweder im tar oder ipkg Format vor. Dies kann vor dem Erstellen des Paketes konfiguriert werden (Variable `INHERIT`).

OpenEmbedded besteht eigentlich nur aus einer Reihe von Metadaten die das Erstellen der Pakete bzw. der Rootfilesystem beschreiben.

Es wird dabei zwischen den folgenden Arten von Metadaten unterschieden:

- Konfigurationsdateien
- .bb Dateien
- Klassen

In Konfigurationsdateien befinden sich Einstellungen die für alle Pakete und Tasks (einzelne Arbeitsschritte) die ausgeführt werden. Die wichtigste Konfigurationsdatei ist `conf/bitbake.conf`. In dieser Datei werden alle anderen Konfigurationsdateien die für eine spezielle Hardwarearchitektur erforderlich sind inkludiert.

Dateien mit der Endung `.bb` enthalten die sogenannten BitBake Recipes. In diesen wird beschrieben wir eine bestimmte Software zu kompilieren und installieren ist. BitBake erstellt das aus der installieren Software ein Paket, welches auf der Zielplatform leicht installiert werden kann. Das Format der Pakete kann entweder `tar.gz`, also ein komprimiertes Tar Archive sein oder ein sogenanntes IPKG sein. IPKG ist ein besonders schlankes Paketmanagementsytem das speziell für Embedded Plattformen entwickelt wurde.

Klassen beschreiben welche Schritte BitBake ausführen muss, um verschiedenste Software kompilieren zu können. So gibt es z.B. eine Klasse `autotools` die sich um das Erzeugen eines Paketes kümmert welches mit Hilfe der GNU Autotools erstellt wurde.

Konkrete Beispiele der Anwendung finden sich in den Kapiteln 4.4.6 und 4.4.7.

4.3.1 BitBake

BitBake ist der sogenannte "Task Executor" von OpenEmbedded. Er liest die vom Entwickler erstellten Konfigurationsdateien und führt die darin angeführten Schritte aus. Es kann mit GNU make verglichen werden. BitBake ist an dem Tool "Portage" angelehnt, welches in der Linux Distribution Gentoo für das Erstellen von Paketen zuständig ist.

BitBake kümmert sich um folgende Dinge:

- Cross Compilation der Softwarepakete
- Auflösen von Abhängigkeiten zwischen Paketen
- Download, Patchen, Kompilierendes Source Codes

So kann zum Beispiel mit dem Kommando

```
bitbake -c build-package-<packagename>
```

ein Pakete einer bestimmten Software erstellt werden. BitBake erkennt anhand von `packagename`, welche Konfigurationsdateien für die angegebene Software zu verwenden sind. Es folgt das Kompilieren, Installieren und das Erzeugen eines Paketes.

Des weiteren stellt BitBake auch Tasks zum Erstellen ganzer Distributionen oder Root Dateisystemen zur Verfügung. Welche Distribution (z.b. OpenMoko) dabei erzeugt wird, hängt auch hier von den Konfigurationsdateien ab.

4.4 OpenMoko und OpenEmbedded

Das OpenMoko Projekt verwendet wie schon weiter oben beschrieben (Kapitel 4.1) OpenEmbedded zum Erstellen der OpenMoko Linux Distribution und deren Pakete. Es wird dafür aber ein eigener OpenEmbedded Zweig zur Verfügung gestellt. D.h. das OpenMoko Projekt nimmt die vorhanden OpenEmbedded Sourcen, prüft diese sicherheitshalber auf deren funktionalität

und stellt diese unter eine eignen URL wieder zur Verfügung. Dies soll die Entwicklung von Anwendung für Projekte Dritter vereinfachen und stabiler gestalten.

4.4.1 MokoMakefile

MokoMakefile ist ein GNU Makefile (siehe 4.1) welches dem Softwareentwickler einen relativ einfachen Einstieg in den Buildprozess von OpenMoko ermöglichen soll. Es kümmert sich um den Download der aktuellen OpenEmbedded Version und checkt die aktuellste OpenMoko Version aus dem Subversion Repository aus. Des weiteren werden Make Targets für das Erstellen des QEMU Emulators (siehe 5) und zum Erstellen des Images für den Bootloader, das Root Dateisystem und den Linux Kernel angeboten.

Folgende Befehle sollen die Anwendung von MokoMakefile verdeutlichen:

- Erstellen des Build Environments: `make setup`
- Update auf die letzte Version von MokoMakefile: `make update-makefile`
- Erstellen des QEMU Emulators: `make build-qemu`
- Erstellen des Flash Images zum Booten des Emulators: `make openmoko-devel-image`

MokoMakefile vereinfacht vorallem den Umgang mit den zahlreichen Versionskontrollsystemen die die verschiedenen Tools verwenden. So wird für die Versionskontrolle von BitBake Subversion [Sub07] verwendet, die einzelne BitBake Recipes (siehe 4.3.1) werden allerdings mit Hilfe von Monotone [Mon07] verwaltet.

4.4.2 Erstellen einer OpenMoko Entwicklungsumgebung

Um mit der Entwicklung eines eignen Softwarepakets zu beginnen muss zuerst die Entwicklungsumgebung erstellt werden. Das folgende Kapitel beschreibt das Erstellen dieser Umgebung und wurde hautpsächlich dem OpenMoko Wiki entnommen ([Var07]). Wichtig ist eine relativ performante Internetanbindung, da ein vielzahl von Dateien unterschiedlicher Grösse heruntergeladen werden muss.

Alle Dateien die zur Entwicklungsumgebung gehören sollten in einem gemeinsamen Unterverzeichnis abgelegt werden. Da die meisten Beispiele auf das Verzeichnis `$HOME/moko` verweisen, hat der Author dieses auch übernommen:

```
mkdir ~/moko && cd ~/moko
```

Nun kann die aktuelle Version des MokoMakefiles heruntergeladen werden:

```
wget http://www.rwhitby.net/files/openmoko/Makefile
```

Mit Hilfe von

```
make setup
```

wird die Entwicklungsumgebung vorbereitet. `make setup` führt eine Vielzahl von Kommandos aus, die Anhand deren Ausgabe beschrieben werden sollen. Dies soll auch zu einem besseren Verständnis der Entwicklungsumgebung beitragen:

Listing 3: make setup Teil 1

```
   ___                                    _          ____   ___   ___ _____ ____
  /---\- __   ___ - __   /\/\   ___ | | _____   |--- \ / - \ / - \---  |--- \
 //  // '_ \ / _ \ '_ \ /    \ / _ \| |/ / _ \    --) | | | | | | | / /  --) |
/ \-//| |-) |  __/ | | / /\/\ \ (-) |   < (-) |  / --/| |-| | |-| |/ /- / --/
\---/ | .--/ \___|_| |_\/    \/\___/|_|\_\___/  |_____|\___/ \___//_/(-)_____|
      |_|

[ ! -e stamps/bitbake ] || \
        ( grep -e 'tags/bitbake-1.8.8' bitbake/.svn/entries > /dev/null ) || \
        ( rm -rf bitbake stamps/bitbake patches stamps/patches )
[ -e stamps/patches ] || \
        ( svn co http://svn.nslu2-linux.org/svnroot/mokomakefile/trunk/patches patches )
A    patches/bitbake-HEAD
A    patches/openmoko-HEAD
A    patches/openembedded-HEAD
Checked out revision 175.
[ -d stamps ] || mkdir stamps
touch stamps/patches
[ -e stamps/bitbake ] || \
        ( svn co -r HEAD \
                http://svn.berlios.de/svnroot/repos/bitbake/tags/bitbake-1.8.8 bitbake )
```

In den Zeilen 8-10 wird festgestellt ob eine bestimmte Version von BitBake bereits vorhanden ist, ist dies nicht der Fall, wird die bereits vorhanden Version von BitBake gelöscht. Das Verzeichnis `stamps` dient dazu um den Status des Erstellens der Entwicklungsumgebung festzuhalten. Ist zum Beispiel die Datei `stamps/bitbake` vorhanden, bedeutet dies das BitBake bereits installiert wurde.

Danach erfolgt in den Zeilen 11 - 16 das Auschecken einiger Patches die für die Installation benötigt werden.

Nun folgt das Erstellen des Verzeichnisses `stamps`, das Anlegen der Datei `patches`, die anzeigt das die erforderlichen Patches bereits vorhanden sind.

Als nächstes folgt das auschecken der Version 1.8.8 von bitbake aus dem BitBake Subversion [Sub07] (Zeile 20 und 21).

Es kann mit der Installation von OpenEmbedded begonnen werden.

```
1  [ -e OE.mtn ] || \
           ( ( version=`mtn --version | awk '{ print $2; }'` ; \
3              wget -c -O OE.mtn.bz2 \
        http://downloads.openmoko.org/OE/snapshots/OE-this-is-for-mtn-$version.mtn.bz2 || \
5              wget -c -O OE.mtn.bz2 \
                   http://downloads.openmoko.org/OE/snapshots/OE.mtn.bz2 ) && \
7            bunzip2 -c OE.mtn.bz2 > OE.mtn.partial && \
             mv OE.mtn.partial OE.mtn )
9  make[1]: Leaving directory `/home/pinhead/tmp/moko'
   [ -e stamps/OE.mtn ] || \
11          ( mtn --db=OE.mtn db migrate && \
              mtn --db=OE.mtn pull monotone.openmoko.org org.openembedded.dev )
13 [ -d stamps ] || mkdir stamps
   touch stamps/OE.mtn
15 [ -e stamps/openembedded ] || \
           ( mtn --db=OE.mtn checkout --branch=org.openembedded.dev \
17           openembedded ) || \
           ( mtn --db=OE.mtn checkout --branch=org.openembedded.dev \
19           -r `mtn --db=OE.mtn automate heads org.openembedded.dev | head -n1` openembedded )
   rm -f openembedded/patches
21 [ ! -e patches/openembedded-HEAD/series ] || \
           ( ln -sfn ../patches/openembedded-HEAD openembedded/patches )
23 [ ! -e openembedded/patches/series ] || \
           ( cd openembedded && quilt push -a )
25 [ -d stamps ] || mkdir stamps
   touch stamps/openembedded
```

In den Zeilen 1 - 15 folgt der Download und die Installation von OpenEmbedded. Die erstellte Datei OE.mtn im Verzeichnis `stamps` (Zeile 14) zeigt an das das OpenEmbedded Repository erfolgreich heruntergeladen wurde. Zu beachten ist, dass das OE Repository nicht von `http://openembedded.org` bezogen wird, sondern von `monotone.openmoko.org`. Dies wurde bereits in Kapitel 4.4 beschrieben.

Nun kann mit dem Checkout des OpenMoko Source Codes begonnen werden:

```
   [ -e stamps/openmoko ] || [ -e openmoko/trunk/.svn/entries ] || \
2            ( cd openmoko && svn co -r HEAD http://svn.openmoko.org/trunk )
   A    trunk/doc
4  A    trunk/doc/hardware
   A    trunk/doc/hardware/HXD8v011
6  A    trunk/doc/hardware/HXD8v011/gpio.txt
   A    trunk/doc/hardware/HXD8v011/pmu-voltage.txt
8  .
   .
10 .
   Checked out revision 3777.
12 rm -f openmoko/patches
   [ ! -e patches/openmoko-HEAD/series ] || \
14          ( ln -sfn ../patches/openmoko-HEAD openmoko/patches )
   [ ! -e openmoko/patches/series ] || \
16          ( cd openmoko && quilt push -a )
   [ -d stamps ] || mkdir stamps
18 touch stamps/openmoko
```

In Zeile 2 wird das Subversion Kommando `svn checkout (co)` ausgeführt. Dieses verbindet sich zum OpenMoko Subversion Server `http://svn.openmoko.org` und lädt die aktuellsten Source Dateien herunter (Revision HEAD). Auch hier wird wiederum eine Datei `stamps/openmoko` angelegt, die anzeigt das dieser Schritt ebenfalls erfolgreich beendet wurde.

```
mkdir -p build/conf
[ -e build/conf/local.conf ] || \
        ( echo 'MACHINE = "fic-gta01"' > build/conf/local.conf ; \
          echo 'DISTRO = "openmoko"' >> build/conf/local.conf ; \
          echo 'BUILD_ARCH = "'`uname -m`'"' >> build/conf/local.conf ; \
          echo 'INHERIT += "rm_work"' >> build/conf/local.conf )
rm -f build/conf/site.conf
( ln -sfn ../../openmoko/trunk/src/host/openembedded/site.conf build/conf/site.conf )
[ -e setup-env ] || \
        echo 'export OMDIR="'`pwd`'"' > setup-env
echo \
        'export BBPATH="${OMDIR}/build:${OMDIR}/openembedded"' \
                >> setup-env
echo \
        'export PYTHONPATH="${OMDIR}/bitbake/libbitbake"' \
                >> setup-env
echo \
        'export PATH="${OMDIR}/bitbake/bin:${PATH}"' \
                >> setup-env
```

Zum Schluß wird noch die Konfigurationsdatei `build/conf/local.conf` angelegt. Diese wird von BitBake verwendet und beschreibt die Maschine für die Distribution bzw. die einzelnen Pakete erstellt werden sollen (`fix-gta01` Zeile 3), den Namen der Distribution die erstellt werden soll `openmoko` (Zeile 4) und die Architektur auf der der Build erstellt wird (hier wird das UNIX Kommando `uname -m` (Zeile 5) aufgerufen, im Falle des Autors ist die Ausgabe `i686`). Der Parameter `INHERIT += rm_work` (Zeile 6) bewirkt das beim Erstellen der OpenEmbedded Pakete das Arbeitsverzeichnis, in dem temporäre Dateien, die für das Kompilierendes Paketes notwendig sind wieder gelöscht werden. Diese würden unnötigen Platz benötigen.

Zusätzlich wird noch eine Datei `setup-env` erstellt, die wichtige Environment Variablen für BitBake enthält. Vor jedem Aufruf von BitBake wird diese Datei von MokoMakefile gesourced. `OMDIR` in Zeile 10 enthält das aktuelle Verzeichnis. Im Falle des Autors `$HOME/moko`. Mithilfe von `BBPATH` findet BitBake die oben erstellte Konfigurationsdatei `build/conf/local.conf`. Im Verzeichnis `build` landen auch alle erstellten Pakete. `PYTHONPATH` wird von BitBake verwendet (BitBake wurde in Python [Pyt07] entwickelt) um diverse Python Bibliotheken zu finden. Zu guter Letzt wird `PATH` noch richtig gesetzt damit BitBake ohne die Angabe des kompletten Installationspfades `$HOME/moko/bitbake/bin` aufgerufen werden kann.

4.4.3 Erstellen des OpenMoko Images

Nun kann mit dem Build der OpenMoko Distribution begonnen werden, dazu ist das Kommando

```
make openmoko-devel-image
```

notwendig. Es werden jetzt alle Softwarepakete und der Linux Kernel kompiliert und danach die verschiedenen Images (Kernel, uBoot, RootFS) erstellt. Dies dauert auch auf schnellen Rechnern mehrere Stunden.

Wurden alle Pakete erfolgreich erstellt, befinden sich die fertigen Images unter `images/openmoko`. Der nächste Schritt wäre zum Beispiel die Images im Emulator auszuprobieren. Mehr dazu siehe Kapitel 5.

4.4.4 Update der OpenMoko Umgebung

Da sich die OpenMoko Software ständig weiterentwickelt, sollte von Zeit zu Zeit ein Update des MokoMakefiles und der OpenMoko Distribution erfolgen. Zuerst muss ein Update des Makefiles durchgeführt werden.

```
make update-makefile
```

lädt via `wget` das aktuelle MokoMakefile und installiert es.

```
make setup
```

passt die vorhanden Verzeichnisstruktur eventuellen Änderungen an.

```
make update
```

Holt die letzte Version des Sourcecodes und der Patches aus den verschiedenen Repositories. Schliesslich müssen noch via

```
make openmoko-devel-image
```

die Kernel-, uBoot- und Filesystemimages neu erstellt werden. BitBake kompiliert bei diesem Schritt **nicht** alle Pakete neu, sondern nur diejenigen die sich seit dem Update verändert haben.

4.4.5 Überblick der Verzeichnisstruktur

Da sich die vollständige Entwicklungsumgebung auf mehrere Untervereichnisse aufteilt und es nicht immer einfach ist den Überblick zu bewahren soll in diesem Abschnitt ein kurzer Überblick Über die wichtigsten Verzeichnisse und deren Inhalt gegeben werden. Diese sind Tabelle 1 dargestellt.

4.4.6 Erstellen eines OpenEmbedded Paketes für PJSIP

Für unser Projekt verwenden wir eine Bibliothek names PJSIP (siehe Kapitel 6). Um die Installation der dafür benötigten Dateien möglichst einfach zu gestalten wurde ein OpenEmbedded Paket erstellt und diese der OpenMoko Distribution hinzugefügt.

Bei allen unten angeführten Befehlen wird davon ausgegangen das der Anwender ein Verzeichnis `moko` in seinem Heimatverzeichnis angelegt hat und sich in diesem befindet (`cd $HOME/moko`). In diesem Verzeichnis sollte sich bereits Version des OpenMoko OpenEmbedded Zweigs befinde, wie in Kaptitel 4.4 beschrieben.

Zuerst muss ein sogenanntes "Local Overlay" erzeugt werden. OpenEmbedded verwendet eine spezielle Verzeichnisstruktur um Ordnung in die Vielzahl von Paketen zu bringen. Beim Erstellen eines "Local Overlays" legt man nun einige Verzeichnis selbst so an, dass OpenEmbedded diese beim Erstellen der Distribution berücksichtigen kann. In diesen Verzeichnissen erstellt der Benutzer dann seine eigenen BitBake Rezepte (siehe 4.3.1). Diese gesonderte Struktur wird benötigt, um bei einem eventuellen Update lokale Änderungen nicht zu verlieren.

Folgende Verzeichnisstruktur muss daher angelegt werden:

```
mkdir local local/conf local/classes \
      local/packages local/packages/images \
      local/packages/tasks -p
```

Wurden die oben angeführten Verzeichnisse erfolgreich im Pfad `$HOME/moko` angelegt, kann nun das globale Konfigurationsfile von OpenEmbedded `build/conf/site.conf` umkopiert werden:

```
cp build/conf/site.conf local/conf/site.conf
```

Tabelle 1: Überblick Verzeichnisstruktur

Verzeichnis	Beschreibung
`$HOME/moko/`	Hier befindet sich die komplette Entwicklungsumgebung und alle weiteren Unterverzeichnisse
`bitbake/`	Enthält BitBake und den OpenEmbedded TaskExecutor
`build/`	Alle Pakete, Images und Objekt Dateien werden hier abgelegt
`build/qemu/`	Objekt Dateien und Binaries die durch das Kompilieren von QEMU erstellt werden
`build/qemu/openmoko/`	Images (Root Filesystem, Kernel und uBoot) die durch OpenEmbedded erstellt werden. Dies sind allerdings Symbolische Links nach `images/openmoko/`
`build/tmp/staging/`	Objekt Dateien die OpenEmbedded beim Erstellen der einzelnen Pakete erzeugt
`build/tmp/work/`	Source Code der einzelnen Pakete, wird aber nach erfolgreichem Kompilieren gelöscht. Außerdem befinden sich hier die `make install` Targets
`build/tmp/rootfs/`	Template des Root Filesystems. Aus diesem Verzeichnis wird das Root Filesystem Image erstellt.
`build/tmp/cross/`	Cross Compiler der zum Kompilieren benötigt wird
`build/tmp/stamps/`	Dateien die Anzeigen was bereits kompiliert wurde
`build/tmp/deploy/`	Enthält die erstellten ipkg-Dateien
`build/conf/`	Konfigurationsdateien für OpenEmbedded
`images/openmoko/`	Enthält die erstellten Kernel, Root Filesystem und uBoot Images
`local/`	Das lokale Overlay siehe 4.4.6
`openembedded/`	Enthält alle Metadaten (BitBake Recipes, M4 Makros) von OpenEmbedded)
`openmoko/`	Alle Dateien die aus dem OpenMoko Subversion Repository ausgecheckt wurden. Dies sind vorallem Kernel Patches, Patches für uBoot, Source Code der OpenMoko Applikationen und BitBake Recipes
`openmoko/developers/`	Diverse Dateien von Core Entwicklern. Hier findet sich oft interessante Dokumentationen
`openmoko/patches`	Patches für OpenEmbedded, BitBake usw. in der Version des Autors allerdings leer
`sources/`	Source Code der OpenMoko und OpenEmbedded Applikationen
`stamps/`	Dateien die MokoMakefile anzeigen welche Schritte bereits durchgeführt wurden. Siehe dazu auch 4.4.2

Dies ist notwendig da dem Build Prozess von OpenEmbedded ja noch nichts von unserem soeben erstellen "Local Overlay" bekannt ist. Wir wollen aber nicht die globale Konfiguration ändern, diese könnte bei einem Update von OpenEmbedded überschrieben werden. Unsere Änderungen gingen dadurch verloren. Also kopieren wir die Datei wie oben beschrieben um und tragen folgende Anpassungen ein:

```
BBFILES := "${OMDIR}/openembedded/packages/*/*.bb \
   ${OMDIR}/local/packages/*/*.bb"
BBFILE_PATTERN_overlay = "^${OMDIR}/local/"
BBFILE_PRIORITY_overlay = "20"
```

In der Konfigurationsdatei des Authors mußte die Variable `BBFILES` verändert und die beiden anderen Variablen `BBFILE_PATTERN_overlay` und `BBFILE_PRIORITY_overlay` zusätzlich eingetragen werden.

Damit beim Aufruf von BitBake durch MokoMakefile auch die richtige Konfigurationsdatei verwendet wird, muss noch die Datei `setup-env` angepasst werden. Diese befindet sich im Verzeichnis `$HOME/moko`. Die Variable `BBPATH` muss angepasst werden und wie folgt aussehen

```
export BBPATH="${OMDIR}/local:${OMDIR}/build:${OMDIR}/openembedded"
```

Es sind nun alle erforderlichen Änderungen an OpenEmbedded vorgenommen worden damit lokale Pakete beim Erstellen der Distribution berücksichtigt werden können. Der nächste Schritt ist das Erstellen der notwendigen BitBake Rezepte für die PJSIP Bibliothek.

Dazu wird die Datei `packages/pjproject/pjproject_0.8.0.bb` mit folgendem Inhalt erstellt:

Listing 4: pjproject_0_8_0.bb

```
DESCRIPTION = "PJSIP Library"
AUTHOR = "pjsip.org"
HOMEPAGE = "http://pjsip.org"
SECTION = "libs"
MAINTAINER = "Toni Schmidbauer"
PRIORITY = "optional"
LICENSE = "GPL"

# where can we find the source package for downloading?
SRC_URI = "http://pjsip.org/release/0.8.0/pjproject-0.8.0.tar.gz"

# pjsip does not work with parallel make, so disable it
PARALLEL_MAKE = ""

# we just want the archive files included in our package
# headers and so on are not required for running our application
FILES_${PN} = "/usr/local/lib/lib*.a"

# pjsip uses ./configure aka autotools, so tell bitbake to use it
inherit autotools
```

Die Variable `SRC_URI` gibt den URL an von der BitBake das Source Paket der PJSIP Bibliothek finden kann. Der erste Schritt beim Erstellen des Pakets ist das Herunterladen des Source Codes. `FILES_${PN}` spezifiziert anhand eines regulären Ausdrucks welche Dateien in das zu erstellende Paket aufgenommen werden sollen. Da der PJSIP Source Code mit Hilfe der GNU Autotools konfiguriert und compiliert wird, teilt `inherit autotools` BitBake mit, dass das Paket durch einen Aufruf von `./configure && make && make install` konfiguriert, kompiliert und installiert werden kann.

Nun muß OpenEmbedded mitgeteilt werden, dass beim Erstellen der OpenMoko Distribution ein zusätzliches Paket mit installiert werden soll. Dies geschieht in der Datei `build/conf/local.conf`. Es muß folgende Zeile am Schluß der Datei angehängt werden:

```
DISTRO_EXTRA_RDEPENDS += "pjproject"
```

Somit ist dem Buildprozess der OpenMoko-Distribution bekannt das es ein Softwarepaket names "pjproject" gibt, dass beim Erstellen der Distribution mit aufgenommen wird.

Nun kann das neue Softwarepakete mit Hilfe des Kommandos

```
make build-package-pjproject
```

kompiliert werden.

Nun kann ein neues Root Dateisystem erstellt werden.

```
make openmoko-devel-image
```

erledigt diese. Selbstverständlich muss das Dateisystem auch in das virtuelle Flash kopiert werden

```
make flash-qemu-local
```

erledigt dies. Mehr zur Bedienung des Emulators findet sich in 5.

4.4.7 Hello World mit OpenEmbedded

Das hier beschrieben Beispiel wurde dem OpenMoko Wiki (siehe [Var07]) entnommen.

In diesem Kapitel soll eine kleine Beispielapplikation entwickelt werden. Es handelt sich dabei um eine einfaches C Programm das in einem Terminal

“Hello World” ausgibt. Diese Applikation soll mit BitBake cross kompiliert (siehe 4.2) werden und am Emulator provisioniert werden. Die Dateien werden dabei direkt im Local Overlay (siehe 4.4.6) abgelegt. Dies sollte nach Meinung des Autor normalerweise vermieden werden. Wird ein eigenes Projekt entwickelt, ist es besser die Dateien in einem separaten Repository aufzubewahren und dort sicherzustellen das sich die Applikation sauber kompilieren lässt. Dies hat den Vorteil unabhängig von OpenEmbedded und BitBake zu sein. Damit ist eine einfache Portierung auf andere Buildsystem und Platformen gewährleistet.

Als erster Schritt wird ein Untervereichnis im Local Overlay (siehe 4.4.6) angelegt in dem sich alle Dateien des Projekts befinden. Wie immer wird vom Verzeichnis `$HOME/moko` ausgegangen.

```
mkdir -p local/myhelloworld/files && cd local/myhellworld/files
```

Nun können dort die notwendigen Quelldateien abgelegt werden

Listing 5: myhelloworld.c

```
#include <stdio.h>

int main(int argc, char** argv)
{
        printf("Hello World!\n");
        return 0;
}
```

Listing 5 beschreibt ein kleines C Programme, welches auf `STDOUT` “Hello World” ausgibt.

Wurde obige Datei erstellt, kann ein BitBake Recipe zum kompilieren des Paketes erstellt werden. Dieses befindet sich im Verzeichnis `local/myhelloworld/`

Listing 6: myhellworld.bb

```
1  DESCRIPTION = "A killer hello world application"
   AUTHOR = "Toni Schmidbauer"
3  HOMEPAGE = ""
   SECTION = "console/applications"
5  PRIORITY = "optional"
   LICENSE = "BSD"
7  SRC_URI = "file://myhelloworld.c \
   file://README.txt"
9
   S = "${WORKDIR}/myhelloworld/"
11
   do_compile() {
13         ${CC} ${CFLAGS} ${LDFLAGS} ${WORKDIR}/myhelloworld.c -o myhelloworld
   }
15
   do_install() {
17         install -m 0755 -d ${D}${bindir} ${D}${docdir}/myhelloworld
           install -m 0755 ${S}/myhelloworld ${D}${bindir}
19         install -m 0644 ${WORKDIR}/README.txt ${D}${docdir}/myhelloworld
   }
```

Zeilen 1 - 6 sind allgemeine Beschreibung die die Applikation betreffen. Interessant ist hier die Variable `SECTION`. Diese stellt eine allgemeine Beschrei-

bung dar und hält sich an den Debian (`http://debian.org` Linux Standard. Welche Varianten von `SECTION` zulässig sind kann unter

`http://www.debian.org/doc/debian-policy/ch-archive.html#s-subsections`

nachgelesen werden.

Die Zeilen 7 und 8 beschreiben die Quelldateien für das Projekt. Diese werden zum Kompilieren und Installieren benötigt.

In Zeile 12 findet sich die Funktion `do_compile()` welche von BitBake aufgerufen wird wenn das Paket im Task Compile kompiliert werden soll. In der nächsten Zeile findet sich der notwendige Compileraufruf. Die Variablen `CC`, `CFLAGS` und `LDFLAGS` werden von BitBake gesetzt und betreffen Optionen mit denen der Compiler bzw. der Linker aufgerufen wird.

Die Funktion `do_install()` in Zeile 17, beschreibt wie die kompilierten Dateien installiert werden sollen. Zeile 18 legt ein Verzeichnis `bindir` und `$(docdir)/myhelloworld` an falls diese nicht schon vorhanden sind. Die Variablen `D`, `bindir` und `docdir` werden wieder von BitBake gesetzt. Als nächstes wird die ausführbare Datei aus dem Verzeichnis `$(S)/myhelloworld/` in das Zielverzeichnis `$(D)/bindir/` kopiert. Dies geschieht ebenfalls mit der Datei `README.txt` nur landet diese eben im Verzeichnis `$(D)/docdir`.

Eine Anmerkung zur von BitBake gesetzen Variablen `D` und `S`. Diese beschreiben ein temporäre Verzeichnisse in denen alle Applikationen installiert werden. `D` stellt dabei die Wurzel des Zieldateisystems dar. Dies bedeutet das alle Dateien die sich unterhalb des Verzeichnisses `$D/<pfad>` befinden, im Zieldateisystem (in diesem Fall das Root Dateisystem der OpenMoko Distribution) im Verzeichnis `/<pfad>` abgelegt werden. `S` gibt die Wurzel des Verzeichnisses an in dem das Softwarepaket kompiliert wurde.

Nun kann mit Hilfe von BitBake ein Paket erstellt werden. Dies geschieht durch den Aufruf von

```
make build-package-myhelloworld
```

oder soll BitBake direkt aufgerufen werden durch

```
. ./setup-env && cd build && bitbake -c build myhellworld )
```

Listing 7: build-package-myhelloworld

```
NOTE: Handling BitBake files: | (4882/4882) [100 %]
NOTE: Parsing finished. 4646 cached, 0 parsed, 236 skipped, 0 masked.
NOTE: build 200801142040: started

OE Build Configuration:
BB_VERSION     = "1.8.8"
```

```
OE_REVISION     = "<unknown>"
TARGET_ARCH     = "arm"
TARGET_OS       = "linux-gnueabi"
MACHINE         = "fic-gta01"
DISTRO          = "openmoko"
DISTRO_VERSION = "P1-Snapshot-20080114"
TARGET_FPU      = "soft"

NOTE: Resolving missing task queue dependencies
NOTE: preferred version 2.5 of glibc not available
        (for item virtual/arm-angstrom-linux-gnueabi-libc-for-gcc)
NOTE: Preparing runqueue
NOTE: Executing runqueue
.
.
.
NOTE: Running task 270 of 270 (ID: 4,
      /home/pinhead/moko/local/packages/myhelloworld/myhelloworld.bb, do_build)
NOTE: package myhelloworld-1.0: started
NOTE: package myhelloworld-1.0-r0: task do_build: started
NOTE: package myhelloworld-1.0-r0: task do_build: completed
NOTE: package myhelloworld-1.0: completed
NOTE: Tasks Summary: Attempted 270 tasks of which
      257 didn't need to be rerun and 0 failed.
NOTE: build 200801142040: completed
```

Listing 7 stellt die Ausgabe des Befehls `bitbake -c build` dar. Man erkennt in den Zeilen 5 - 13 die Zielplatform `ARM` und welche Distribution erstellt werden soll(`openmoko`). Danach folgt ein Darstellung aller Tasks die BitBake ausführt. In Zeile 28 erhält man noch eine Zusammenfassung. Wichtig ist hier die Ausgabe `0 failed`.

Danach folgt ein

```
make rebuild-package-task-base
```

Diese Kommando übersetzt alle Pakete die zum Erzeugen des Root Dateisystems notwendig sind. Nun kann das neue Root Dateisystem selbst erzeugt werden.

```
make openmoko-devel-image
```

erstellt dieses. Selbstverständlich muss das Dateisystem auch in das virtuelle Flash kopiert werden

```
make flash-qemu-local
```

erledigt dies. Mehr zur Bedienung des Emulators findet sich in 5. Nach dem Starten des Emulators und dem öffnen des Terminals kann `myhelloworld` aufgerufen werden, wie in Abbildung 10 dargestellt.

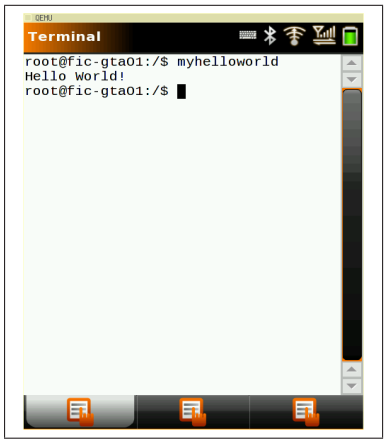

Abbildung 10: Boot Screen

5 OpenMoko Emulator

QEMU ist ein unter einer Open Source Lizenz (GNU General Public Licence GPL) verfügbarer Emulator und kann mit VMWARE verglichen werden. QEMU kann aber eine Vielzahl von Hardwarearchitekturen emulieren. Zu den Unterstützen Plattformen zählen zum Beispiel

- Intel x86 und x86_64
- ARM
- SPARC und SPARC64
- PowerPC und PowerPC64
- MIPS
- m68k
- SH-4
- Alpha
- CRIS

Das OpenMoko Projekt verwendet QEMU um die verwendete Hardware zu emulieren und den Entwicklern somit ein einfaches Werkzeug zum Testen ihrer Software zur Verfügung zu stellen.

QEMU wurde dabei um einige Komponenten erweitert um ein vollwertiges Mobiltelefon simulieren zu können. So wird zum Beispiel auch der im OpenMoko Telefon vorhandene GPS Chip emuliert. Selbiges gilt auch für die GSM Hardware. Dazu ist es allerdings notwendig nicht den offizielle QEMU Quellcode zu verwenden (dieser ist unter `http://fabrice.bellard.free.fr/qemu/` zu finden), sonder eine von OpenMoko bereitgestellte Version. Mehr dazu findest sich im Kapitel 5.3.

5.1 QEMU mit MokoMakefile

MokoMakefile stellt einige Make Targets (siehe auch 4.1) zur Verfügung um das Kompilieren und Starten von QEMU zu vereinfachen. Der Emulator wird automatisch beim Erstellen der Entwicklungsumgebung kompiliert (siehe 4.4.2). Alle ausführbaren Dateien finden sich im Verzeichnis `build/qemu`, wie in 4.4.5 beschrieben.

Zuerst muss ein Flash File erstellt werden. Dieses wird von QEMU verwendet um die Distribution zu booten

```
make flash-qemu-local
```

Wurde das Kommando erfolgreich ausgeführt, finden sich die fertigen Flash Dateien im Verzeichnis `build/qemu/openmoko`

- `openmoko-kernel.bin`: der OpenMoko Linux Kernel
- `openmoko-flash.base`: das Flash Image

Nun kann der Emulator mit dem aktuell geflashten Image gestartet werden.

```
make run-qemu
```

Es erscheint sofort das uBoot Boot Menü welches in Abbildung 11 dargestellt ist.

Mit Hilfe der Leertaste kann das Booten des Betriebssystems fortgesetzt werden. Nach einigen Meldungen des Linux Kernels erscheint der OpenMoko Splashscreen (Abbildung 12). Die Geschindigkeit des Emulator ist stark von der CPU Perfomance des Host Systems abhängig. Auf dem Intel Core 2 System des Autors läuft der Emulator mit anehmbarer Geschwindigkeit.

Das Laden des Betriebssystem nimmt ein paar Minuten in anspruch. Danach erscheint das Hauptmenü wie in Abbildung 13 dargestellt.

Abbildung 11: Boot Screen

Abbildung 12: Splash Screen

Abbildung 13: Hauptmenü

5.2 Offizielle OpenMoko Images im Emulator

OpenMoko stellt auch offizelle Images zur Verfügung. Mit der in 5.1 vorgestellten Methode werden ja die in 4.4.2 erstellten Images geladen.

```
make download-images
```

lädt die letzten verfügbaren offiziellen OpenMoko Images von der OpenMoko Webseite und legt diese im Verzeichnis `images/` ab. Nun kann mit

```
make flash-qemu-offical
```

aus diesem Image eine Flash Datei erstellt werden. `make run-qemu` startet danach wieder den Emulator.

5.3 QEMU ohne MokoMakefile

Es gibt auch die Möglichkeit QEMU ohne die Hilfe von MokoMakefile zu kompilieren und die von OpenMoko bereitgestellten Images zu flashen. Dies hat den Sinn ein schnelles Ausprobieren der OpenMoko Plattform zu ermöglichen, ohne die komplette OpemEmbedded Umgebung erstellen zu müssen. Folgende Schritte sind dazu notwendig (diese Anleitung wurde von `http://wiki.openmoko.org/wiki/OpenMoko_under_QEMU` übernommen und getestet).

Zuerst muss der Source Code des Emulator aus dem OpenMoko Subversion Repository ausgecheckt werden. Es muss die von OpenMoko bereitgestellte und gepatchte QEMU Version verwendet werden, da sonst einige Hardwareteile (vorallem GPS und GSM Chip) nicht emuliert werden.

```
svn checkout https://svn.openmoko.org/trunk/src/host/qemu-neo1973
```

Diese Kommando verbindet sich zum Subversion Server und checkt den aktuellen Source Code aus. Dieser befindet sich danach im Verzeichnis `qemu-neu1973/`. Nun kann mit

```
cd qemu-neo1973 && ./configure --target-list=arm-softmmu && make
```

QEMU kompiliert werden. Zum erfolgreichen Kompilieren ist allerdings die GCC Version 3.x notwendig. Mit Hilfe von

```
openmoko/download.sh
```

können die aktuellen, von OpenMoko bereitgestellten, Images heruntergе-

laden werden.

`openmoko/flash.sh`

Erstellt das virtuelle Flash Device.

Listing 8: make setup Teil 1

```
Formatting 'openmoko-flash.image', fmt=qcow2, backing_file=openmoko-flash.base, size=67584kB

    All done.

    Read the qemu manual and use a commandline like the following to boot:
$ arm-softmmu/qemu-system-arm -M gta01 -m 130 -mtdblock \
  openmoko/openmoko-flash.image -kernel openmoko/openmoko-kernel.bin -usb -show-cursor

    Append '-snapshot' to make the flash image read-only so that every
    time emulation starts in the original unmodified state.
    Use [Enter] for AUX button, [Space] for POWER.
    Append '-usbdevice keyboard' to attach a standard keyboard.

    If you've configured qemu with --enable-phonesim (requieres Qt4 and
    a C++ toolchain), use the following commandline to run phonesim:
$ (cd phonesim; LD_LIBRARY_PATH=lib ./phonesim -gui ../openmoko/neo1973.xml)&
    The GUI is optional. When phonesim is running, append
    '-serial tcp:localhost:12345' to qemu invocation. Security
    note a la GLSA: phonesim listens on 0.0.0.0.
```

Wurde obiges Kommando wie in 8 dargestellt erfolgreich beendet kann mit folgenden Kommando QEMU gestartet werden.

```
arm-softmmu/qemu-system-arm -M neo -m 130 -mtdblock \
  openmoko/openmoko-flash.image -kernel \
  openmoko/openmoko-kernel.bin -usb \
  -show-cursor
```

5.4 Netzwerkverbindung

QEMU bietet auch die Möglichkeit eine Netzwerkverbindung zwischen Emulation und Host System aufzubauen. Da QEMU im Falle von OpenMoko kein Netzwerkinterface zur Verfügung stellt, muss hier etwas getrickst werden.

Ein funktionierende Netzwerkverbindung zwischen Emulation und Host System ist besonders für die Fehlersuche in Anwendungen hilfreich. Der Bildschirm wird zwar schön dargestellt, ist aber etwas umständlich zu bedienen. Ein Login am Endgerät vereinfacht dies enorm und erleichtert die Fehlersuch in Applikationen.

Des Weiteren könne Anwendung einfach auf das emulierte Telefon kopiert werden. Ein schneller Compile-Run-Debug Zyklus, der bei der Entwicklung von Anwendungen notwendig ist, lässt sich dadurch einfach verwirklichen.

5.4.1 Netzwerk mit PPP

Diese Anleitung wurde dem OpenMoko Wiki

`http://wiki.openmoko.org/wiki/OpenMoko_under_QEMU` entnommen.

Dies stellt die einfachere der beiden Möglichkeiten für eine Netzwerkverbindung zwischen Hostsystem und Emulation dar. QEMU muss dazu händisch mit den zusätzlichen Optionen `-serial vc -serial pty` gestartet werden. Dadurch wird ein serieller Port emuliert über den mit Hilfe von `PPP` eine IP Verbindung aufgebaut werden kann. Listing 9 stellt das Notwendige Kommando dar

Listing 9: Aufruf QEMU

```
( cd build/qemu && arm-softmmu/qemu-system-arm \
    -M neo -m 130 -usb -show-cursor \
    -usbdevice keyboard \
    -mtdblock openmoko/openmoko-flash.image \
    -sd openmoko/openmoko-sd.image \
    -kernel openmoko/openmoko-kernel.bin \
  -serial vc -serial pty )
```

wichtig ist es beim Starten von QEMU auf folgende Ausgabe zu achten

```
char device redirected to /dev/pts/5
```

da das Gerät `/dev/pts/5` später zum Starten des PPPD's am Host System benötigt wird. Als nächsten Schritt öffnet man die Terminal Appliktion und setzt folgendes Kommando ab

```
pppd nodetach debug /dev/ttySAC1
```

Es wird hier der PPPD Daemon gestartet. Dieser bindet sich an das serielle Interface `/dev/ttySAC1` welches von QEMU ja emuliert wird. Nun kann auch am Host System der PPPD gestartet werden

```
sudo pppd nodetach debug 192.168.68.1:192.168.68.2 noauth /dev/pts/5
```

`sudo` ist notwendig da hier `root` Rechte benötigt werden.

Mit einem einfachen `ping` Kommando wird die Verbindung getestet

Listing 10: ping Kommando

```
% ping 192.168.68.2
PING 192.168.68.2 (192.168.68.2) 56(84) bytes of data.
64 bytes from 192.168.68.2: icmp_seq=1 ttl=64 time=19.1 ms
64 bytes from 192.168.68.2: icmp_seq=2 ttl=64 time=8.80 ms

--- 192.168.68.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 4.518/10.209/19.182/5.440 ms
```

Ein Login mit Hilfe von `ssh` ist nun ebenfalls möglich

Listing 11: ssh Aufruf

```
% ssh root@192.168.68.2
root@192.168.68.2's password:
root@fic-gta01:~$ uname -a
Linux fic-gta01 2.6.22.5-moko11 #1 PREEMPT Sat Dec 1 23:58:28 CET 2007 armv4tl unknown
root@fic-gta01:~$
```

Es ist kein `root` Passwort vergeben, die Abfrage nach dem Passwort kann einfach mit der Eingabetaste bestätigt werden.

6 PJSIP

PJSIP ist eine Opensource Implementierung der für SIP basierende VoIP Kommunikation notwendigen Protokolle und Multimediadienste. Auch neuere Protokollerweiterungen wie Presence und Instant Messaging werden unterstützt.

Zu den größten Vorteilen des unter GPL Lizenz entwickelten Frameworks gehören seine extreme Portabilität, die geringen Systemanforderungen und die hervorrgende Dokumentation. In Abbildung 14 ist die Architektur des gesamten Frameworks dargestellt.

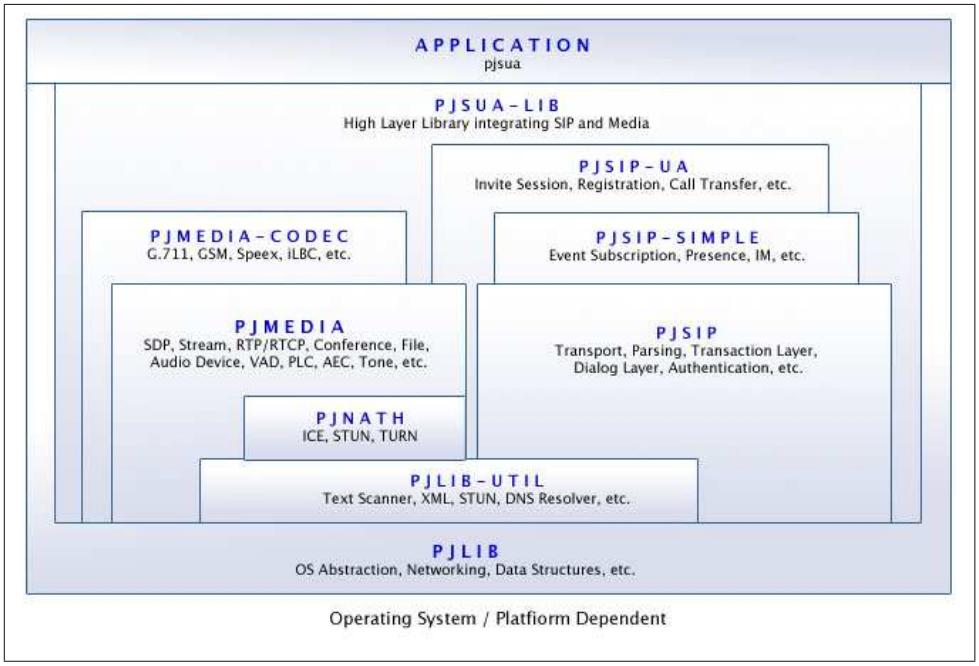

Abbildung 14: PJSIP Stack

In dieser Abbildung erkennt man den geschichteten Aufbau der verschiedenen Bibliotheken. Die PJLIB stellt Basisfunktionen zur Verfügung und soll anstelle der plattformabhängigen C-Bibliotheken und System-Calls verwendet werden.

PJLIB-UTIL stellt nützliche Funktionen, wie XML-Parsing, STUN-Client und DNS-Resolver zur Verfügung. In der PJNATH Bibliothek sind die, für die Überbrückung von NAT notwendigen Verfahren implementiert. Neben STUN und TURN ist auch der vollständige ICE-Algorithmus implementiert.

Der auf den oben genannten Bibliotheken aufsetzende PJMEDIA Stack implementiert alle für VoIP eingesetzten Audio Codecs. Darüber hinaus übernimmt er das Parsen der SDP Nachrichten, sowie das Management des gesamten RTP Protokolls.

Die eigentlich PJSIP Bibliothek kümmert sich um den Aufbau und die Verwaltung der SIP Sessions. Auch das Event und Presence Framework befindet sich in dieser Funktionseinheit.

Als oberste Schicht wird eine PJSUA API für C/C++ und Python zur Verfügung gestellt. Damit lassen sich vollständige SIP User Agents mit relativ geringen Aufwand implementieren. Sollte man spezielle Funktionen benötigen ist es allerdings erforderlich die PJSIP Bibliothek direkt zu verwenden.

PJSIP wurde bereits erfolgreich auf Linux, BSD, MacOS, Solaris, Symbian und Windows portiert. Der SIP Client für die Nintendo DS wurde mit dem PJSIP Framework erstellt.

Derzeit wird in der Entwicklerversion die Unterstützung für IPv6 hinzugefügt, somit stellt PJSIP auch für zukünftige Anforderungen eine solide Basis zur Verfügung.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ECC ..... Error Correction Codes
FTL ..... Flash Translation Layer
GPL ..... GNU General Public License
LFS ..... Log Structured Filesystem
NFTL ..... NAND Flash Translation Layer
OOB ..... Out of Band Bytes

Literaturverzeichnis

Literatur

[Ele03] Samsung Electronics. *USER'S MANUAL S3C2410X 32-Bit RISC Microprocessor Revision 1.2*. Samsung Electronics, 2003.

[Int98] Intel. *Understanding the Flash Translation Layer (FTL) Specification*. Intel, 1998.

[Mak07] Makefiles. `http://gnu.org/software/make/manual/make.html`, 2007.

[Mon07] Monotone. Monotone. `http://www.monotone.ca/`, 2007.

[MR91] John K. Ousterhout Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System*. Technical report, Electrical Engineering and Computer Sciences, Computer Science Division, University of California, 1991.

[Pyt07] Python. `http://www.python.org/`, 2007.

[Sub07] Subversion. Subversion. `http://subversion.tigris.org/`, 2007.

[Var07] Various. Application development crash course. `http://wiki.openmoko.org/wiki/Application_Development_Crash_Course`, 2007.

[vH04] Kurt Hall / William von Hagen. *The Definitiv Guide to GCC*. Apress, 2004.

[Woo] David Woodhouse. *JFFS : The Journalling Flash File System*. Red Hat, Inc.

[Yag03] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly Media, 03 2003.