Hochschule
Augsburg University of
Applied Sciences

Fakultät für Informatik

Diplomarbeit

Studienrichtung

Informatik

Matthias Kettl

Evaluation von Embedded-Linux auf dem i.MX287 Mikrocontroller von Freescale™

Erstprüfer: Prof. Dr. H. Högl

Zweitprüfer: Prof. Dr. W. Klüver

Abgabe: 28.09.2012

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Diplomarbeit:
Matthias Kettl
Von-Rehlingen-Straße 27
86356 Neusäß
Telefon:+49 821 2422166
matthias@kettl.com

Lizenz

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.

Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Neusäß, den 28.09.2012

Matthias Kettl

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ADC	Analog to Digital Converter
BCB	Boot Control Block
BGA	Ball Grid Array
BIOS	Basic Input/Output System
BSP	Board Support Package
CAN	Controller Area Network
CF	Compact Flash
CPU	Central Processing Unit
DCD	Device Configuration Data
DEB	Debian Pakete
DHCP	Dynamic Host Configuration Protocol
DVD	Digital Versatile Disc
eMMC	Embedded Multimedia Card
EXT (2/3)	Extended Filesystem (Version 2 / 3)
FAT	File Allocation Table
GDB	GNU Debugger
GND	Ground
GNU	GNU's Not Unix
GPIO	General Purpose Input/Output
HAB	High Assurance Boot
I^2C	Inter-Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IPK	Itsy Package Management System
IVT	Image Vector Table
KDB	Kernel Debugger
LAN	Local Area Network
LED	Light-Emitting Diode
MAC	Media-Access-Control Adresse
MBR	Master Boot Record
MMC	Multimedia Card

NAND	Not AND Gatter
NFS	Network File System
OCD	On-chip Debugger
OTG	On-The-Go
PC	Personal Computer
PWM	Pulse Width Modulation
QEMU	Quick EMUlator
RAM	Random-Access Memory
ROM	Read-Only Memory
RPM	RPM Package Manager
RSA	Rivest, Shamir und Adleman
RTC	Real Time Clock
SD	Secure Digital Memory Card
SSH	Secure Shell
SVN	Apache Subversion
TFTP	Trivial File Transfer Protocol
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VCC	Voltage at the common collector
Vref	Voltage Reference

1 Einführung

In vielen Bereichen unseres Alltags interagieren wir mit Embedded-Systemen, vielleicht sogar ohne davon wirklich bewusst Notiz zu nehmen. In einer Vielzahl von Teilgebieten der Technik werden diese heute eingesetzt. Sie kommen, um nur ein paar wenige zu nennen, in DVD-Playern, Kraftfahrzeugen, Flugzeugen, Waschmaschinen oder Mobiltelefonen vor. Embedded-Systeme stellen Schnittstellen für die Interaktion mit dem Benutzer bereit, messen, steuern, regeln und leisten noch einiges mehr. Dabei werden sie oft an ihre Aufgabenbereiche speziell angepasst.[1] Um eine solche Anpassung erst zu ermöglichen, bedarf es eines umfassenden Wissens über bereitgestellte Funktionen und Möglichkeiten. In dieser Arbeit wird eine Evaluierung eines solchen Systems im Bezug auf den Betrieb mit Embedded-Linux durchgeführt.
Linux erfreut sich in den letzten Jahren steigender Beliebtheit in diesem Bereich.[2] Darum richtet sich das Augenmerk der Evaluierung auf dieses Betriebssystem. Gerade im Bereich der Mobiltelefone und der Tablet-PCs ist hier ein deutlicher Trend zu erkennen.[3]
Im Verlauf der Arbeit werden darüber hinaus der Prozess zur Erstellung von Linux-Images für eine Ziel-Hardware, entsprechende Werkzeuge und Methoden dargestellt. Ein Augenmerk liegt dabei auch auf dem OpenSource Entwicklungszyklus, der bei der Entstehung von Software, für eine bestimmte Hardwareplattform eingesetzt wird. Es werden einige Werkzeuge, die in diesem Zusammenhang eingesetzt werden, vorgestellt.

1.1 Embedded Systems

Ein Beschreibungsversuch von eingebetteten Systemen (engl. embedded systems) gestaltet sich nicht ganz trivial. Versucht man eine Definition über grundlegende Eigenschaften eines solchen Systems, wie das Vorhandensein

1 vgl. [ES12]
2 vgl. [LD07]
3 vgl. [CGA11]

von Eingabe- / Ausgabemöglichkeiten und einer programmierbaren Kontrolllogik wie in [HOL05] beschrieben, so würde das ein eingebettetes System wohl beschreiben, jedoch genauso einen Standard Desktop PC. Diese werden aber nicht zwingend als eingebettete Systeme bezeichnet. Der Einsatzzweck ist eher universeller Art. Auf ihnen kann Software für unterschiedlichste Anwendungsbereiche betrieben werden. Eingebettete Systeme werden jedoch meist für einen speziellen Anwendungsfall konzipiert.

Unter Betrachtung des technischen Fortschrittes, immer leistungsfähigeren Prozessoren und fallender Preise für Speicher, lösen viele Embedded-Systeme heute nicht mehr nur einzelne Aufgabenstellungen. Heutige Set-Top-Boxen, beispielsweise aus dem Multimediabereich, stellen dem Benutzer eine Fülle von Diensten zur Verfügung. Sei es das Aufnehmen oder (zeitversetzte) Wiedergeben von TV-Sendungen, Wiedergabe von Musik oder das Surfen im Internet. Oftmals kommen multitaskingfähige Betriebssysteme zum Einsatz, ähnlich zu denen auf Standardrechnern.

In Hallinan [HAL07] ist eine Charakterisierung wie folgt zu finden:

- Beinhaltet eine Verarbeitungseinheit z.B. Mikroprozessor
- Typisch entworfen für eine spezielle Aufgabe oder einen Zweck
- Kein oder nur einfaches Benutzerinterface
- Oft mit begrenzten Ressourcen versehen
- Begrenzte Leistung, z.B. Betrieb mit Batterien
- Wird i.d.R nicht als Allzweckrechner verwendet
- Die Anwendungssoftware ist eingebettet und wird nicht vom Benutzer gewählt
- Wird komplett mit anwendungsspezifischer Hardware und vorinstallierter Software ausgeliefert
- Ist oft für Anwendungen ohne Benutzerinteraktion vorgesehen

Auch mit dieser Charakterisierung könnte man einen Rechner auf Basis von Desktop-Hardware u.U. als eingebettetes System verstehen. [HAL07] führt hier das Beispiel eines unbeaufsichtigten Data-Centers an, in dem ein Rechner die Überwachung übernimmt und Alarmmeldungen automatisch versendet. Solch ein System sollte auch ohne eine Interaktion mit dem Benutzer, z.B. nach einem Stromausfall, wieder funktionieren.

Heute verschwimmen die Grenzen zwischen Standardrechnern und eingebetteten Systemen zunehmend, was eine exakte Abgrenzung erschwert. Typisch sind häufig ein begrenzter Speicher, keine oder nur kleine Festplatten oder Flash-Speicher und manchmal auch keine Netzwerkfunktionalitäten. Oftmals besteht das Benutzerinterface nur aus einigen LEDs oder nur einem seriellen Port.[4]

1.2 Linux, GNU/Linux und Embedded Linux

In diesem Abschnitt sollen die Begrifflichkeiten Linux, GNU/Linux und Embedded Linux, deren Bedeutung und Verwendung, im Kontext dieser Arbeit gegeneinander abgegrenzt werden.

Linux

Ursprünglich wurde der Name Linux von Linus Torvalds für den von ihm geschriebenen Betriebssystem Kern, dem Linux Kernel, verwendet. Im heutigen Sprachgebrauch wird der Begriff für mehr als nur den Kernel verwendet. Häufig bezeichnet er den Kernel an sich, ein Linux System oder eine komplette Distribution. Eine Distribution besteht dabei meist aus Linux Kern und zusätzlicher Anwendungssoftware.[5]

4 vgl. [HOL05], [HAL07]

5 vgl. [YMBG08], S. 2 ff.

GNU/Linux

In der OpenSource-Szene wird eine Debatte um die Namensbezeichnung von Software-Distributionen auf Basis von Linux und GNU geführt. Ziel des GNU Projekt[6] ist es, ein vollständig freies Betriebssystem namens GNU zu entwickeln. Dem Projekt fehlte jedoch ein Betriebssystemkern. Diese Rolle nahm der Linux-Kern ein. Nun besteht Uneinigkeit über die Bezeichnung, eines aus GNU-Tools und Linux Kern bestehenden Systems. Nach Ansicht der Mitglieder des GNU Projektes müsste ein solches System als GNU/Linux und nicht nur als Linux bezeichnet werden.[7]

Um eventuelle Verwechslungen zwischen Betriebssystem und Betriebssystemkern auszuschließen, wird in dieser Arbeit Linux und GNU/Linux synonym für ein Betriebssystem verwendet. Handelt es sich um den Betriebssystemkern als solches, werden die Begriffe Linux Kern bzw. Linux Kernel verwendet.

Embedded Linux

In dieser Arbeit soll ein Embedded System gegen ein Embedded Linux evaluiert werden. Als Embedded Linux wird, im Kontext der Entwicklung für eingebettete Systeme, ein angepasster Linux Kern verstanden, der auf einem solchen System betrieben wird. Meist wird dieser von einer Vielzahl anderer, ebenfalls angepasster Software begleitet. Diese Kombination aus Kernel und Software wird auch als *embedded linux distribution* bezeichnet.[8]

Es soll nun herausgearbeitet werden, welche Gründe für oder gegen einen Einsatz von Linux sprechen. Linux wird heute schon in einer großen Zahl von Geräten eingesetzt. Die Gründe dafür sind sowohl technischer wie auch ökonomischer Art.[9]

6 vgl. [GNU12]

7 vgl. [RST07]

8 vgl. [YMBG08] S. 4 ff

9 vgl. [HAL07] S. 2 ff

In Hallinan [HAL07] sind dazu folgende Ausführungen zu finden:

- Linux ist eine ausgereifte, hochperformante, stabile Alternative zu traditionell proprietären Betriebssystemen für Embedded Systeme
- Linux unterstützt eine große Anzahl an Anwendungen und Netzwerkprotokollen
- Linux ist gut skalierbar
- Linux kann, im Gegensatz zu den meisten proprietären Betriebssystemen, ohne Lizenzgebühren ausgeliefert werden
- Linux hat eine große Anzahl aktiver Entwickler, was zu einer schnellen Unterstützung von neuen Hardwarearchitekturen, Plattformen und Geräten führt
- Eine große Zahl von Hardwareherstellern unterstützten Linux

Als Nachteile werden oftmals mangelnder Support und das Fehlen von Echtzeitfähigkeit genannt.
Diese Lücke schließen jedoch Unternehmen, die sich genau auf diesem Sektor spezialisiert haben. Über sie ist sowohl eine Anpassung von Linux an Hardwareplattformen, als auch ein entsprechender Support kommerziell zu beziehen.
Einige ambitionierte Projekte zielen auf eine Erweiterung um Echtzeitfähigkeit für Linux ab. So gibt es beispielsweise ein „CONFIG_PREEMPT_RT“ Patch für den Linux Kernel, der eingeschränkte Echtzeitfunktionalität bereitstellt. In diesem Zusammenhang sollte auch noch das, ursprünglich von Professor Victor Yodaiken und Michael Barabanov an der Universität von New Mexico initiierte, Projekt RTLinux[10] und das Projekt von Professor Paolo Mantegazza der Universität Mailand namens RTAI[11] genannt werden.

10 vgl. [RTL12] und [YMBG08] S. 5 – What Is Real-Time Linux?

11 vgl. [RTAI12]

2 Hardwarekit TQKa28-AA

In diesem Kapitel wird nun die verwendete Hardwareplattform näher beschrieben. Es handelt sich dabei um ein Hardwarekit der Firma TQ-Systems GmbH mit der Bezeichnung TQKa28-AA. Dieses Kit besteht aus zwei primären Baugruppen. Zum einen aus dem Evaluationsboard STK-MBa28, von TQ-Systems auch als Mainboard oder Baseboard bezeichnet, und zum anderen aus einem aufgesteckten Modul mit der Bezeichnung TQMa28. Das Evaluationsboard stellt für dieses Modul eine Vielzahl von externen Schnittstellen bereit. Das Kernstück auf dem TQMa28 Modul bildet eine Freescale™ ARM-CPU mit der Bezeichnung MCIMX287 (i.MX287).
Dieses Hardwarekit kann direkt in Applikationen oder Steuerungen integriert werden oder als Basis für eigene Produktideen dienen.[12]

Schlüsseleigenschaften[13] werden von TQ-Systems folgend beschrieben:

- Schnelles Time-to-Market
- Flexibel kombinierbare Displays
- Erweiterter Temperaturbereich
- Design-In Unterstützung
- Kosteneffizient
- Langzeitverfügbar
- Grundlage für kundenspezifische Lösungen

12 vgl. [TQKa28]

13 Quelle: [TQKa28]

Übersicht TQKa28-AA

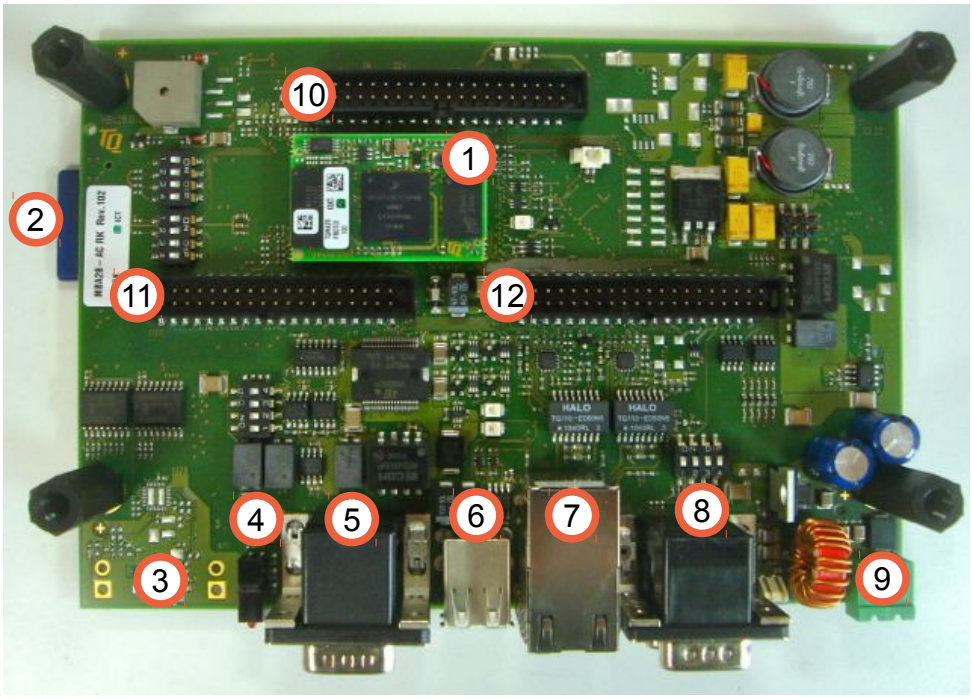

Abbildung 1: TQKa28-AA bestehend aus STK-MBa28 Board und TQMa28 Modul

ID	Bezeichnung	Beschreibung
1	TQMa28 Modul	Modul mit i.MX28 Prozessor, Flash, RAM
2	SD Karte	Kartenslot, Zugänglich von Unterseite
3	Audio	In Abbildung nicht bestückt
4	User LED	3x User LED, rot, gelb, grün
5	GPIO / CAN	oben CAN, unten GPIO
6	USB	2x USB
7	Ethernet	2x Ethernet
8	Serielle Ports	oben RS-232, unten RS-485
9	Power	Versorgungsspannung 18 V über Netzteil
10	Erweiterungsport	Vgl. Datenblatt STK-MBa28 Port X10
11	Erweiterungsport	Vgl. Datenblatt STK-MBa28 Port X15
12	Erweiterungsport	Vgl. Datenblatt STK-MBa28 Port X14

Tabelle 1: Übersicht TQKa28-AA

2.1 TQMa28 Modul

Abbildung 2: TQMa28 Modul

In den Produktinformationen[14] und dem User's Manual[15] zum TQMa28 Modul sind folgende beschreibende Informationen zu finden:

2.1.1 Aufbau des Moduls

Neben der CPU befinden sich auf dem Modul der Firma TQ-Systems weitere Basiskomponenten. Das komplette Modul beseht aus dem i.MX287 Prozessor, einem eMMC NAND Flash, einem DDR2 SRAM, EEPROM und einem Temperatursensor. Die genaue Lage der Komponenten sowie die exakte Bezeichnung derer, sind aus der folgenden Abbildung 3: – Komponenten TQMa28 Modul und der Tabelle 2: – Hauptkomponenten TQMa28 Modul, zu entnehmen.

14 vgl. [TQMaP10]

15 vgl. [TQMaUM11]

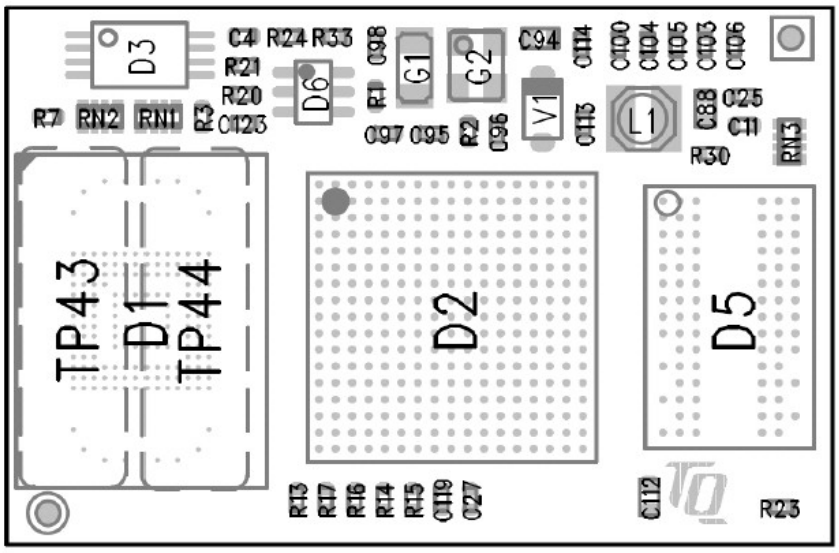

Abbildung 3: Komponenten TQMa28 Modul – Quelle: [TQMaUM11] S. 32 – Illustration 14

ID	Bezeichnung	Bemerkung
D1	THGBM1G4D1EBAI7	2 GB eMMC Flash-Speicher der Firma Toshiba
D2	MCIMX287CVM4B	i.MX 287 Prozessor der Firma Freescale
D3	464WK	64K I^2C EEPROM 64k der Firma TI
D5	IAHI2 D9MDK	128 MB DDR2 SRAM der Firma Micron
D6	T730	LM73 I^2C Temperatursensor der Firma National Semiconductor

Tabelle 2: Hauptkomponenten TQMa28 Modul

Blockdiagramm TQMa28

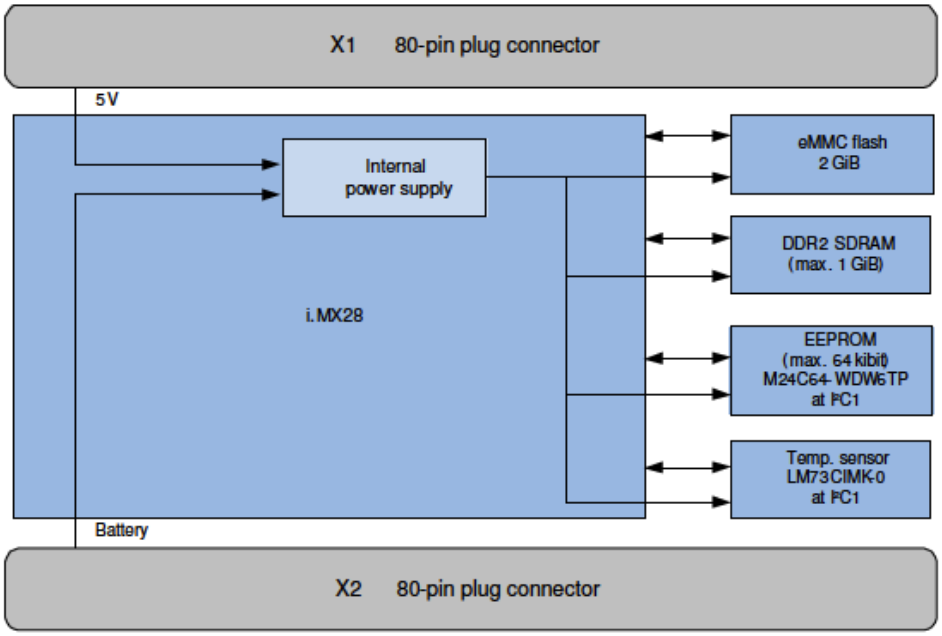

Abbildung 4: Blockdiagramm TQMa28 – Quelle: [TQMaUM11] S. 8:, 3.1.1 TQMa28 block diagram

Alle essentiellen Pins des Prozessors, außer denen für eMMC und DDR2 SRAM, sind auf die beiden 80 poligen Stecker X1 und X2 herausgeführt. Die exakte Belegung der beiden Steckverbindungen ist dem TQMa28 User's Manual [TQMaUM11] S. 10 ff. - 4.1.1.1 Module plug connector X1 und 4.1.1.2 Module plug connector X2 zu entnehmen. Die gesamten Abmessungen betragen dabei 40 x 26 mm.

2.1.2 Mikroprozessor

Als Prozessoren können auf dem Modul der i.MX 283 oder der i.MX 287 zum Einsatz kommen. Auf dem verwendeten Modul ist ein i.MX 287 integriert. Dabei handelt es sich um eine Freescale™ ARM CPU MCIMX287CVM4B in der Bauform BGA 289. Der in 90nm Technik gefertigte ARM 926 EJ-S Core arbeitet mit bis zu 454MHz und bietet einen breiten Funktionsumfang gerade im Hinblick auf Multimediaanwendungen.

Abgrenzung ARM und i.MX

Prozessoren die auf der ARM Technologie basieren, zeichnen sich besonders durch ihre Ausführungsgeschwindigkeit und ihre dabei geringe Stromaufnahme aus. Nahezu alle Betriebssysteme für Embedded Systeme sind auf ihnen lauffähig. Sie können mit Microsoft Windows Embedded und Windows Phone, Linux, Google Android und einigen Echtzeitbetriebssystemen betrieben werden. Durch diese Eigenschaften gewinnt die Architektur, gerade im Bereich der Smartphones und Tablet-PCs, immer mehr an Bedeutung gegenüber anderen Architekturen.

Das Design der ARM Architektur stammt von der ARM Holding Ltd.. Das Geschäftsmodell der ARM Holding Ltd. ist nicht die Herstellung und der Vertrieb von Prozessoren, sonder ist spezialisiert auf 32-bit-Chip-Design. Die Chips werden unter Lizenz von Halbleiterherstellern produziert. Zu den namhaften Partnern gehören u.a. Texas Instruments, Fujitsu und Freescale. In diesem konkreten Fall wird ein auf ARM926EJ-S Core basierender Chip von der Firma Freescale unter dem Namen i.MX 287 vertrieben.[16]

Im folgenden Blockdiagramm sind die Kernfunktionalitäten des i.MX-Chips zu sehen.

16 Vgl. [ARM12], [HAL07] S. 56 ff., [YMBG08] S. 57 ff.

Blockdiagramm i.MX28

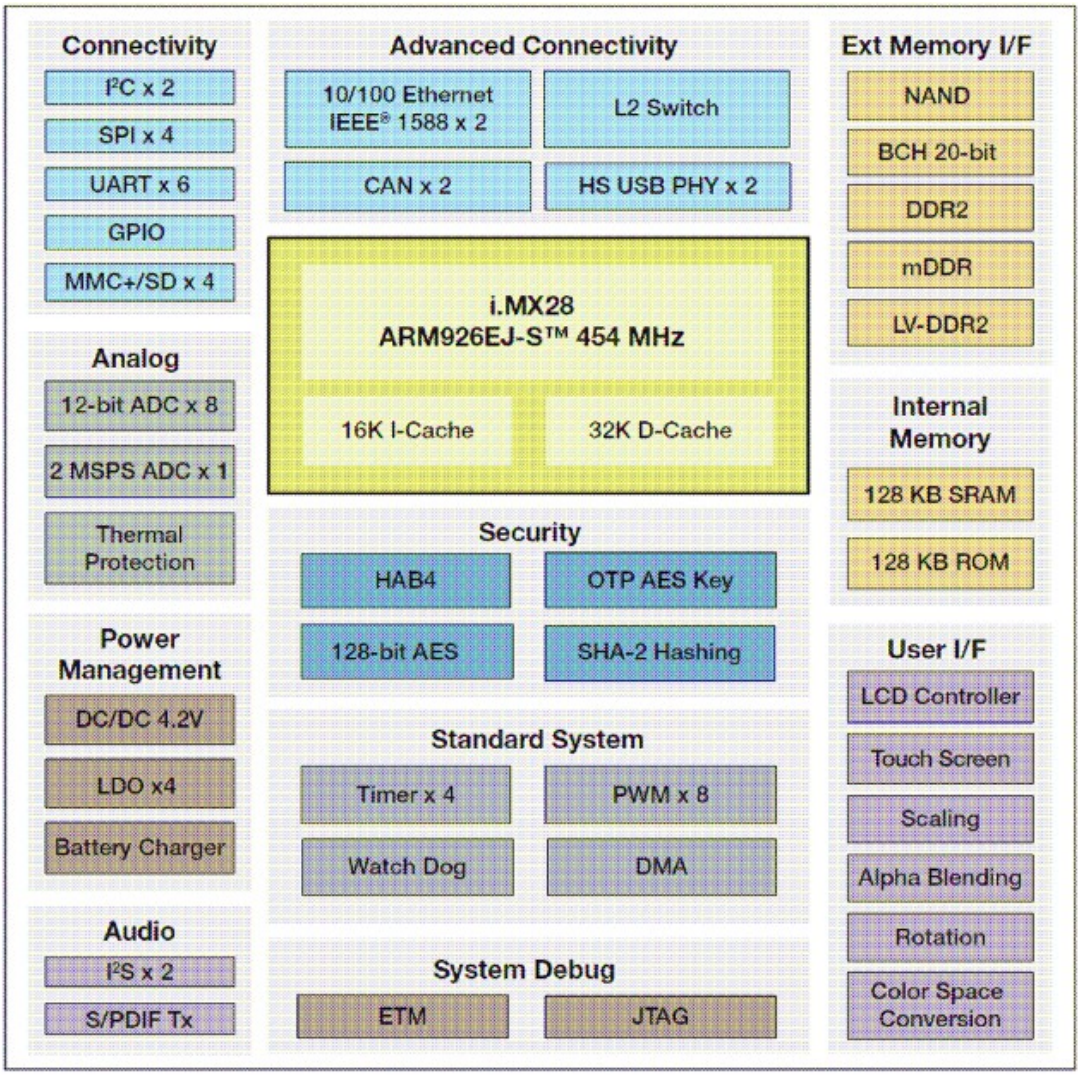

Abbildung 5: i.MX28 Blockdiagramm - Quelle: Freescale (TM)

Zu den Kernfunktionalitäten[17] des gerade einmal 14,0 x 14,0 x 0,8 mm großen Chips gehören:

ARM926EJ-S CPU:

- I-Cache, D-Cache, L2-Cache
- Integrated SRAM

17 Auszug aus dem TQMa28 User's Manual [TQMaUM11] S. 19

- CPU Clock: 454 MHz
- DDR2

User interfaces:

- LCD controller (up to 24-bit-per-pixel WVGA)
- Touch interface (4-wire / 5-wire)
- Graphics support (scaling, rotation, alpha blending, colour space conversion)

Interfaces:

- 2 × USB 2.0 Hi-Speed Host interface (USB0 is OTG capable)
- 2 × FlexCAN modules
- 2 × configurable SPI, 2 × SSI/I^2S, 5 × UART, MMC/SDIO, 2 × I^2C
- 2 × Ethernet (with IEEE® 1588 extension)

Package:

- BGA-289

Eine detaillierte Beschreibung aller Funktionen des Chips ist dem Datenblatt von Freescale™, i.MX28 Applications Processors for Consumer Products[18], zu entnehmen.

2.1.3 Speichermedien

Arbeitsspeicher DDR2 SRAM

Als Arbeitsspeicher des Systems ist ein maximal 1 GB großer DDR2 SRAM auf dem Modul vorgesehen. Das in dieser Arbeit verwendete Modul, ist jedoch nur mit 128 MB bestückt.

18 Vgl. [IMX28CEC] - 1.1 Device Features – S. 2 ff.

eMMC Flash

Der i.MX 287 Prozessor verfügt über einen eMMC Controller mit der Bezeichnung SSP0. Über diesen wird ein 2 GB großer eMMC NAND Flash der Firma Toshiba angesteuert. Der eMMC-Standard beschreibt eine Speicher-Architektur, bestehend aus MMC-Interface, Flash-Speicher und Controller. Diese Komponenten befinden sich zusammen in einem BGA Gehäuse[19]. Auf dem eMMC Flash kann das Betriebssystem gespeichert bzw. betrieben werden.

EEPROM

Zum permanenten speichern von Moduleigenschaften oder kundenspezifischen Parametern ist ein EEPROM mit max. 64kbit vorhanden. Das EEPROM wird über der I^2C Bus gesteuert.[20] Die genaue Adressierung kann dem *TQMa28 Users Manual [TQMaUM11] : S. 24 – 4.2.9.2 I^2C* entnommen werden.

2.1.4 Systemkomponenten

RTC (Real Time Clock)

Zur exakten Zeitbestimmung verfügt der Prozessor über eine interne Echtzeituhr (Real Time Clock). Die RTC wird, nach Verlust der Spannungsversorgung, von einem Li-Ion Akku direkt über den Prozessor weiter versorgt.

Temperatursensor

Über den I^2C Bus kann der, auf dem Modul bestückte, Temperatursensor gesteuert werden. Die genaue Adressierung kann dem *TQMa28 Users Manual [TQMaUM11]: S. 24 – 4.2.9.2 I^2C* entnommen werden.

SD-Karte

Als ein weiteres Speichermedium stellt das Modul einen SD-Controller (SPP1) bereit. Dieser wird an der Steckverbindung X1 bereitgestellt. Auf dem Modul

19 [EMMC07]

20 vgl. [TQMaUM11] S. 19 – 4.2.2.3 EEPROM S. 24 - 4.2.9.2 I^2C

selbst ist keine direkte Möglichkeit vorgesehen, eine SD Karte mit dem Controller zu verbinden.

Grafikinterface und LCD Bus

Für die Grafikausgabe wird ein LCD-Bus bereitgestellt. Es können an ihm parallele Displays mit einer Auflösung von maximal 800x480 Pixel und 24bit Farbtiefe betrieben werden. Im Rahmen der Arbeit stand kein entsprechendes LCD für Testzwecke bereit. Die Betrachtung dieser Funktionalität bleibt daher außen vor.

USB

Zur Verbindung von USB Geräten stehen zwei USB Ports zur Verfügung. Davon ist der Port USB0 OTG-fähig. On-the-go (kurz OTG) bezeichnet eine Erweiterung des USB Standards um die Möglichkeit, eine Kommunikation zwischen USB Geräten ohne einen Rechner zu ermöglichen. Beispielsweise können MP3 Player mit OTG-Funktionalität direkt und ohne einen PC Daten untereinander austauschen.[21]

Ethernet

Der i.MX 287 stellt dem Anwender zwei Ethernet-Schnittstellen an X1 und X2 zur Verfügung.

Serielle Schnittstellen

Auch die seriellen Schnittstellen des i. MX 287 werden über die beiden Steckverbindungen dem Baseboard bereitgestellt. In Summe besitzt der Prozessor sechs Stück. Davon können AUART0 bis AUART4 im High-Speed Modus betrieben werden. Der sechste Port, DUART, ist auf eine Geschwindigkeit von maximal 115,2 kbit beschränkt.

21 [OTG12]

PWM

Es werden acht PWM-Kanäle vom Modul bereitgestellt. In Verbindung mit dem STK-MBa28 werden sie jedoch in alternativer Konfiguration verwendet. Beispielsweise wird DUART über die Pins PWM0 und PWM1 bereitgestellt. Die Funktionen in Verbindung mit dem Baseboard sind dem Users Manual[22] *Table 17: PWM signals* zu entnehmen.

GPIO

Sie werden ebenfalls vom Modul bereitgestellt und sind über das Baseboard an X1B nutzbar.

JTAG

Zu Debugzwecken ist eine JTAG-Schnittstelle (X16) auf dem Baseboard vorgesehen. Alle notwendigen Pull-Up und Pull-Down Widerstände sind bereits auf dem TQMa28 Modul vorhanden.

ADC

Das TQMa28 Modul stellt acht Analog Digital Converter (kurz ADC) bereit. Vier von ihnen (LRADC2 bis LRADC5) können für die Auswertung eines Touchscreens (4-wire Touchscreen) verwendet werden.

Audio

Audiofunktionalitäten sind über ein serielles Audio Interface realisiert.

2.1.5 Spannungsversorgung des Moduls

Das Modul kann mit einer Spannung von 5 V oder 3.3 V betrieben werden. Beim Betrieb mit 3.3 V ist zu beachten, dass die USB-Schnittstellen nicht verwendet werden können.[23] Neben der Möglichkeit das Modul über eine Festspannungsquelle zu betreiben, besteht außerdem die Option, es über einen

22 vgl. [TQMaUM11] S. 25 - 4.2.10 PWM

23 vgl. [TQMaUM11] S. 7 - 2. BRIEF DESCRIPTION

Akku (Li-Ion) zu betreiben. Die Kontrolle über den Ladevorgang übernimmt die PMU (Power Management Unit) des i.MX selbst. Die PMU unterstützt u.a. einen linearen Ladevorgang der Li-Ion Akkus, die Kontrolle der Akkuspannung und regelt das Umschalten zwischen Festspannung und Akkuspannung.[24] Die einzelnen PINs zur Spannungsversorgung finden sich ebenfalls an den beiden Steckverbindungen wieder.

Typische Leistungsaufnahmen

Parameter	V_s	Typ.	Unit
Current consumption standby	3.3 V	11	mA
Current consumption standby	5.0 V	8,6	mA
Current consumption in reset	5.0 V	20,5	mA
Current consumption in Linux idle mode	5.0 V	140	mA
Current consumption in Linux idle mode	3.3 V	140	mA
Current consumption in Linux boot mode	5.0 V	220	mA
Current consumption in Linux boot mode	3.3 V	220	mA
Current consumption of internal RTC	> 1.3 V	10	µA

Tabelle 3: Current consumption - Quelle: [TQMaUM11] S. 28: 4.2.16 Power management – Table 20

Der maximal aufgenommene Strom, für den schlechtesten Fall, wird mit 400 mA angegeben, was einer rechnerisch maximalen Leistungsaufnahme von 1,32 W bei 3,3 V bzw. 2 W bei 5 V entspricht.

2.2 STK-MBa28 Board

In diesem Abschnitt soll das Mainboard oder Baseboard näher beschrieben werden. Nachdem ein Großteil der Funktionalität vom TQMa278 Modul bereitgestellt wird, beschränkt sich diese auf die bereitgestellten externen Schnittstellen und einige grundlegende Funktionen. Für weiterführende Informationen zum Baseboard kann das *STK-MBa28 Users Manual [STK11]* herangezogen werden.

24 vgl. [IMX28CEC] S. 9 – Table 4: i.MX28 Digital and Analog Modules (continued): PMU

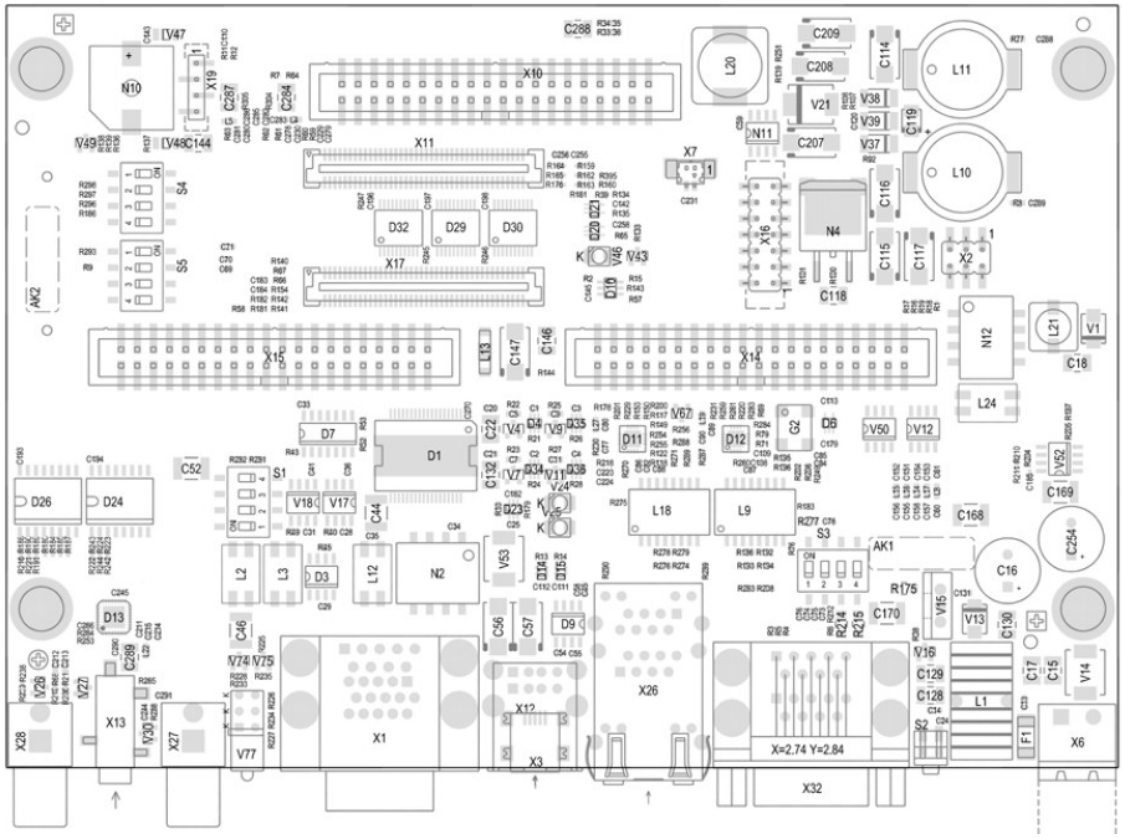

Abbildung 6: STK-MBa28 Übersicht - Quelle: [STK11] S. 61 - 6.1.2 STK-MBa28 top view

2.2.1 Spannungsversorgung des Boards

Die Spannungsversorgung erfolgt über das mitgelieferte Steckernetzgerät (18 V, max. 3,9 A) über den Anschluss X6. Eine durchgeführte Messung ergab eine Stromaufnahme von 230 mA während des Linux idle Betriebs. Board und Modul zusammen haben damit eine tatsächliche Leistungsaufnahme von 4,4 W. Der Wert erscheint, durch die zusätzliche Versorgung von Baugruppen auf dem Baseboard, plausibel.

2.2.2 Externe Schnittstellen

Die folgende Abbildung und die Tabelle bieten eine Übersicht über die externen Schnittstellen wie im *STK-MBA User's Manual [STK11] S. 9 ff.* beschrieben. Im User's Manual sind die Schnittstellen detailliert mit entsprechender Belegung dokumentiert.

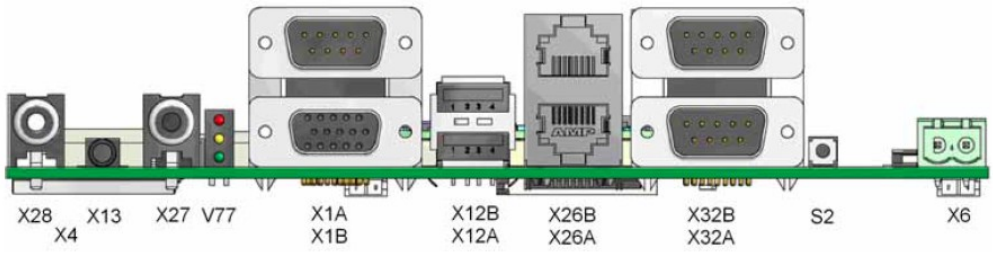

Abbildung 7: Externe Schnittstellen - Quelle: [STK11] S. 9 - Illustration 2: External interfaces of the STK-MBa28

ID	Bezeichnung	Bemerkung
X28	Audioausgang links	
X4	SD-Karte	von unten zugänglich
X13	Mikrofoneingang	
X27	Audioausgang rechts	
V77	Benutzer LED	rot, gelb und grün als Power LED
X1A	CAN1 / CAN2	
X1B	GPIO	
X12B	USB Port	OTG Funktionalität
X12A	USB Port	
X26B	Ethernet 2	IEEE-802.3, 10/100 MBit
X26A	Ethernet 1	IEEE-802.3, 10/100 MBit
X32B	RS 232	als AUART3 am TQMa28 Modul
X32A	RS 485	als AUART0 am TQMa28 Modul
S2	Reset Taster	
X6	Spannungsversorgung	Steckernetzgerät mit 18 V 3,9 A

Tabelle 4: Übersicht externer Schnittstellen STK-MBa28

2.2.3 Interne Schnittstellen

JTAG

Im Datenblatt ist der Anschluss für JTAG mit X16 bezeichnet. Die Belegung der 14-poligen Stiftleiste weicht dabei vom gängigen Pinout für ARM ab. Das Pinout bei ARM sieht an X16.13 Vref vor, am STK-MBa28 ist hier ein Pin mit der Bezeichnung DEBUG[25], zur Wahl des Debug-Verhaltens, zu finden. Der Debugmodus kann über den Schalter S3-2[26] beeinflusst werden. Es kann zwischen JTAG und Boundary Scan gewählt werden.

Erweiterungsports

Für Tests und Erweiterungen bietet das Baseboard drei Erweiterungsports. Hier sind nahezu alle angebotenen Funktionen an drei Wannensteckern komfortabel zu erreichen. Sie sind im *STK-MBa28 User's Manual* [STK11] unter der Bezeichnung X10, X15 und X14 ab der Seite 51 zu finden. Exemplarisch ist der AUART4 lediglich an diesen Ports verfügbar. AUART4, anders als AUART0 und AUART3, wird nicht an eine 9-polige Sub-D Buchse geführt, da er in alternativer Konfiguration als SPI-Interface arbeiten kann.

AUART2

Zu erwarten wäre in diesem Zusammenhang auch, dass der AUART2 an einem der Erweiterungsports zu finden ist. Dies ist jedoch nicht der Fall. Dieser UART ist über das Baseboard nicht nutzbar, auch wenn der Prozessor selbst über ihn verfügt.[27] Die Erklärung dafür lies sich erst im Verlauf der Arbeit finden. TQ-Systems nutzt sowohl den I²C-Bus 0 als auch den I²C-Bus 1 zur Ansteuerung von Peripherie auf dem Baseboard. Die dazu verwendeten Pins `i2c1_sda` und `i2c1_scl` sind dabei mit `auart2_rts` und `auart2_cts` gemultiplext und können nur entweder für AUART2 oder I²C 1 verwendet werden.

25 vgl. [TQMaUM11] S. 26 - 4.2.12 JTAG

26 vlg. [TQMaUM11] S. 26 - Table 18: DEBUG function

27 vgl. [IMX28CEC] S. 61 - 4.3 Signal Contact Assignments und [TQMaUM11] S. 23 - 4.2.9.1 UART

2.2.4 Benutzerinterface / Status LEDs

Im Auslieferungszustand stellt das Baseboard nur begrenzte Möglichkeiten zur Benutzerinteraktion bereit. Zwar können die internen und externen Schnittstellen genutzt werden, jedoch bietet das Board ohne Erweiterungen beispielsweise keine User-Taster oder ähnliches. Es existiert lediglich der Reset-Taster S2. Darüber hinaus bietet es noch drei LEDs die den Status des Boards widerspiegeln sollen. Von diesen können nur zwei als User-LEDs benutzt werden. Zur Parametrisierung des Basisverhaltens sind vier DIP-Schalter (S1,S3-S5) vorgesehen. In der folgenden Tabelle sind sie mit der jeweiligen Funktion beschrieben.

ID	Bezeichnung	Bemerkung
V77A	Benutzer LED (rot)	
V77B	Benutzer LED (gelb)	
V77C	Power LED (grün)	nicht als Benutzer LED nutzbar
V22	Power LED (blau)	Rückseite, VCC 5V OK
V24	USB Host Interface (rot)	
V25	USB Host Interface (rot)	
V46	Reset aktiv (grün)	
S1	Dip-Schalter CAN Terminierung	Default 1-4 OFF
S2	Reset-Taster	
S3	Dip-Schalter RS485 Terminierung	Default 2-4 OFF, 1 ohne Funktion
S4	Dip-Schalter Boot Mode	Default 1-4 OFF
S5	Dip-Schalter Boot Mode	Default 2-4 OFF, 1-2 ohne Funktion

Tabelle 5: Benutzerinterface / Status LEDs – Quelle: [STK11] S. 54 ff.

2.2.5 Bootoptionen

Über die Dip-Schalter S4 und S5 lässt sich das Bootverhalten beeinflussen. Mögliche Optionen sind:

S4 - 1	S4 - 2	S4 - 3	S4 -4	S5 - 1	S5 - 2	S5 - 3	S5-4	Boot from
OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	eMMC Flash
OFF	OFF	OFF	**ON**	**ON**	OFF	OFF	OFF	SD-Karte
OFF	**ON**	OFF	OFF	**ON**	OFF	OFF	OFF	Rescue Mode

Tabelle 6: Boot modes - vgl. [STK11] S. 57 - Table 70

Bei Auslieferung sind die Bootoptionen so eingestellt, dass der eMMC Flash Modus verwendet wird. Zur Option Rescue Mode sind im User's Manual keine weiteren Informationen zu finden. Hierzu gibt aber das *i.MX28 Applications Processor Reference Manual* Aufschluss. Auf Seite 997 wird die Funktion im Kapitel *12.13.4 USB Recovery Mode* beschrieben. Damit kann ein Firmware-Image per USB auf das Target geschrieben werden und dient als Rückfallebene zu den anderen Optionen.

3 Grundlegende Konzepte

3.1 Entwicklungskonzepte

Im Embedded Bereich ist oft die Rede von Hostsystem und Targetsystem (oder auch Zielsystem). Mit Target- / Zielsystem wird die eigentliche Embedded-Plattform bezeichnet. Das Hostsystem hingegen ist ein konventioneller Computer, an dem Entwickler mit entsprechenden Werkzeugen tätig sind. Software wird dort entwickelt bzw. angepasst und erst anschließend auf das Zielsystem übertragen.

Hostsysteme besitzen eine vergleichsweise deutlich höhere Leistung. Gerade wenn Quellcode für leistungsschwache Embedded Systeme übersetzt werden muss, wird diese Aufgabe vom Hostsystem erledigt. Unterscheiden sich die CPU-Architekturen, muss auf dem Hostsystem eine Cross-Compiler-Toolchain eingesetzt werden. Diese erzeugt den Maschinencode für die entsprechende Ziel-Architektur. Anschließend kann das Resultat dann auf dem eigentlichen System getestet werden.

Zwischen den beiden Systemen besteht meist eine permanente serielle Verbindung. Über sie kann mit dem Zielsystem interagiert werden. Dieser Weg wird gewählt, da es oft unmöglich ist, an Embedded-Systemen Peripherie wie Tastatur oder Bildschirm, für eine direkte Ein- oder Ausgabe anzuschließen.

Linked Setup

Neben der seriellen Verbindung kann auch eine Netzwerkverbindung zwischen den Systemen bestehen. Über diese kann die erstellte Software oder ein ganzes Betriebssystem bereitgestellt werden. Es kommt zu keinem physischen Austausch eines Datenträgers bei diesem Prozess. Eine solche Entwicklungsumgebung wird als „**Linked Setup**“ bezeichnet. Beispielsweise kann der Linux-Kernel per TFTP (trivial file transfer protocol) und ein Root Filesystem per NFS (network file system) vom Host bereitgestellt werden. Ein entsprechender Bootloader auf dem Embedded-System stellt dann eine Verbindung zum Host her und lädt diese. Ein weiterer Vorteil dieser Methode ist ein schnelles und komfortables Debugging.[28]

Removable Storage Setup

Als eine zweite Variante ist es möglich auf die physische Verbindung beider Systeme zu verzichten. Auf dem Hostsystem wird dann ein Datenträger erzeugt, der alle notwendigen Informationen zum Betrieb des Zielsystems beinhaltet. Er wird anschließend entnommen und mit dem Zielsystem

28 [YMBG08] S. 39 ff.

verbunden. Diese Strategie wird „**Removable Storage Setup**“ genannt. Auf dem Zielsystem ist meist nur ein minimaler Bootloader vorhanden. Alle anderen Bestandteile des Betriebssystems werden vom Datenträger geladen. In der Praxis werden u.a. Medien wie Compact Flash, MMC oder SD Karten verwendet.
Nachvollziehbar verlängert sich der Entwicklungsprozess mit dieser Methode ungemein. Auch bei nur minimalen Änderungen z.B. am Root Filesystem muss der Datenträger neu erstellt und im Zielsystem getestet werden. Während der aktiven Entwicklung kommt sie daher eher selten zum Einsatz.[29]

Standalone Setup

Eine dritte Methode ist das native Kompilieren auf einem Embedded System selbst. Diese wird als „**Standalone Setup**“ bezeichnet. Je nach Leistung des Systems kann das durchaus eine Option sein. Hierbei werden die notwendigen Werkzeuge wie Compiler, Linker, Debugger usw. direkt auf der Embedded Plattform betrieben. Damit wird auch die Notwendigkeit eines Cross-Compilers umgangen. Der erzeugte Maschinencode ist direkt lauffähig.[30]

Alle drei Strategien werden in Kapitel 4.4 in Verbindung mit dem TQ Board durchgeführt.

3.2 Typischer Bootvorgang

Interessant ist welche Stufen, vom Einschalten eines Embedded Systems bis zu dessen Betrieb, durchlaufen werden. Um eine Vorstellung davon zu bekommen wird nun ein typischer Bootvorgang und dessen einzelne Bestandteile näher beschrieben. Jede einzelne Stufe spielt eine wesentliche Rolle in der Anpassung eines eigenen Linux für die Zielhardware.

29 [YMBG08] S. 40

30 [YMBG08] S. 40 ff.

Nachdem ein Board mit Spannung versorgt wurde, wird meist der Boot-Mode über die Einstellung der Boot-Pins ermittelt. Diese entscheiden darüber, von welchem Medium versucht wird etwas zu laden. Die möglichen Einstellungen für das STK-MBa28 Baseboard sind in *Tabelle 6: Boot modes* zu finden.
An welcher Adresse eines Speichermediums dies der Fall ist, kann meist in den Datenblättern der Hersteller gefunden werden. Diese Adresse ist zwischen verschiedenen Herstellern meist nicht identisch. Von dieser ersten Speicheradresse werden dann die Instruktionen eines dort abgelegten Programms geladen und von der CPU ausgeführt. Dieses, oft auch als Firmware bezeichnete Programm, kann bereits für den einfachen Betrieb des Systems ausreichen. Ein BIOS, wie es aus dem Bereich der Standard-PCs bekannt ist, existiert meist nicht.

Oft wird an dieser Startadresse ein Bootloader zu finden sein. Er ist ein spezielles Programm, welches weitere Teile eines komplexeren Betriebssystems nachlädt. Meist kopiert er sich selbst bei Aufruf aus dem vergleichsweise langsamen Flash-Speicher in den SRAM. Dadurch wird die Ausführungsgeschwindigkeit erhöht. Im Anschluss daran kann beispielsweise der Linux Kernel von ihm geladen werden.

Der Funktionsumfang eines Bootloaders sollte bereits in einer frühen Designphase festgelegt werden. Es existiert ein weites Feld an Bootloadern und sie können nicht nur Programme von Speichermedien nachladen, sondern auch erweiterte Aufgaben übernehmen. Eine davon ist beispielsweise ein Booten von einem Netzwerkserver. Es gibt auch Situationen in denen Daten nur verschlüsselt vorgehalten werden sollen. Wird eine solche Betriebsart als Teil des Entwicklungsprozesses oder des späteren produktiven Betriebs benötigt, ist auf eine Unterstützung zu achten.

Linux braucht neben dem Kernel noch einen Bereich, in dem es seine Programme, Daten, Konfigurationen u.v.m. findet. Der Bereich wird als Root

Filesystem bezeichnet. Seine Größe kann je nach Anforderungen stark variieren.

Die nachstehende Abbildung zeigt eine exemplarische Aufteilung eines Speicherbereiches, in die drei Teile Bootloader, Kernel und Root Filesystem. Wegen der nicht einheitlichen Adressierung wurde auf Speicheradressen verzichtet. Auch muss nicht immer eine Adressierung von links niedrig zu rechts hohen Adressen gegeben sein.[31]

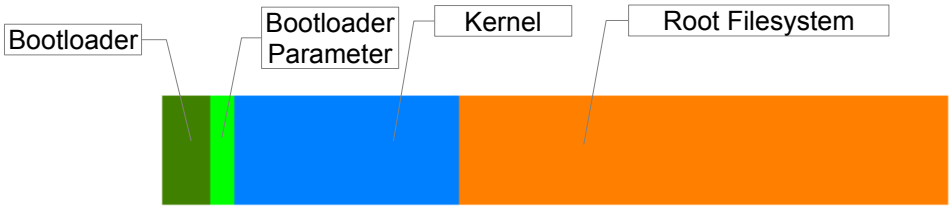

Abbildung 8: Typical solid-state storage device layout – Quelle: [YMBG08] S. 49 – Figure 2-5

Später muss der hier sehr grob beschriebene Startvorgang selbst noch in weitere Schritte untergliedert werden.

4 Inbetriebnahme

In diesem Kapitel befasst sich die Arbeit mit den Anforderungen an das Hostsystem und einer ersten seriellen Verbindungsaufnahme zum Embedded-System. Im Verlauf werden die drei Strategien Linked Setup, Removable Storage Setup und Standalone Setup gezeigt und die Möglichkeiten des Bootloaders erläutert.

4.1 Vorbereitung des Hostsystems

4.1.1 Hardwareanforderungen

Bei der Wahl des Hostsystems sollten im Vorfeld bereits einige Überlegungen

31 vgl. [YMBG08] S. 48 ff

angestellt werden. Mit dem Wissen über die grundlegenden Konzepte aus dem vorigen Kapitel lassen sich bereits einige Anforderungen an die Hardware ableiten. Es sollte bei der Wahl des Hostsystems auf entsprechende Kommunikationsschnittstellen geachtet werden.

Das System sollte mindestens über einen seriellen Anschluss verfügen. Ohne diesen ist eine Parametrisierung, Konfiguration oder Interaktion mit dem Embedded-System zwar nicht ausgeschlossen, jedoch schwierig. Für eine effektive Entwicklung ist er nahezu unumgänglich. Bereits diese erste Anforderung kann sich jedoch als problematisch herausstellen. Viele aktuelle Notebooks und Desktop-PCs besitzen keine serielle Schnittstelle mehr. Ein Ausweg kann hier ein USB zu RS232 Konverter sein.

Des Weiteren sollte das Hostsystem über mindestens einen Netzwerkanschluss verfügen. Über diesen kann im weiteren Verlauf der Linux-Kernel und das Root Filesystem bereitgestellt werden. Dies gestaltet sich deutlich einfacher. Netzwerkanschlüsse gehören heute zum Standard.
Je nach Netzwerkumgebung, in dem das Hostsystem betrieben wird, kann es durchaus sinnvoll sein, über einen zweiten Anschluss nachzudenken. So kann das Hostsystem über eine Netzwerkverbindung am Produktions-LAN, z.B. einem Firmennetz, angeschlossen sein und über eine Zweite eine Verbindung zum Targetsystem aufgebaut werden. Der Vorteil kann darin gesehen werden, dass ein System, auf dem aktiv entwickelt wird, nicht im Produktions-LAN betrieben werden muss. Gerade in der Entwicklungsphase kann es zu Fehlern kommen, die u.U. Auswirkungen auf ein komplettes Netzwerk haben können. Von der anderen Seite betrachtet, können auch Einflüsse aus einem Netzwerk z.B. Ergebnisse von Tests verfälschen. Eine sichere Methode, solche Situationen zu umgehen, ist der direkte Anschluss des Embedded-Systems mittels eines Cross-Kabels über einen zweiten Netzwerkanschluss.

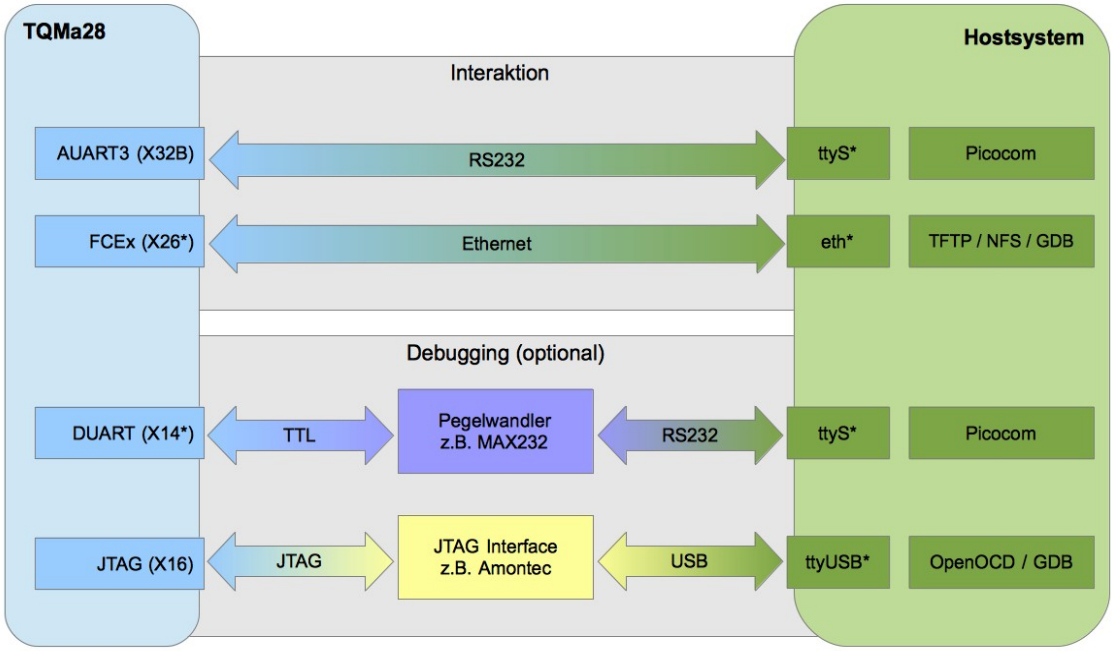

Abbildung 9: Verbindungen zwischen TQMa28 und Hostsystem

Abbildung 9 zeigt die physischen Verbindungen zwischen dem STK-MBa28 (TQMa28) und dem Entwicklersystem (Hostsystem). Dabei sind in diesem Schema auch die optionalen Verbindungen für ein späteres Debugging zu finden. Für die nachfolgenden Schritte sind sie zwar unerheblich, spielen aber im Kapitel 7 eine wichtige Rolle.

Wird mit externen Medien wie SD oder CF Karten gearbeitet, sollte für einen raschen Medienwechsel ein entsprechender interner oder externer Kartenleser vorhanden sein.

Häufig werden auf dem Hostsystem große Mengen an Quellcode übersetzt. Um hier effizient arbeiten zu können, sollte die Leistungsanforderung an das System nicht unterschätzt werden. Zum Teil kann das Compilieren einer kompletten Linux-Distribution, je nach Umfang, mehrere Stunden oder gar Tage in Anspruch nehmen.

4.1.2 Softwarevoraussetzungen

Innerhalb dieser Arbeit wird Linux auf dem Hostsystem betrieben. Selbstverständlich würden auch andere Betriebssysteme in Betracht kommen. Für alle namhaften Betriebssysteme sind entsprechende Werkzeuge vorhanden und können zu einer Umgebung eingerichtet werden, die für die selben Aufgaben geeignet wären. Solche Umgebungen werden in dieser Arbeit nicht weiter beleuchtet. Aus der Nähe zum Zielsystem ist die Entscheidung für das Betriebssystems des Hosts auf Debian GNU/Linux 6.0 gefallen. Diese Entscheidung basiert auch darauf, dass die meisten Werkzeuge direkt aus dem Paket-Repository installiert werden können. Diese Werkzeuge sind zwar für andere Betriebssysteme auch verfügbar, müssen aber zum Teil über Umwege zugänglich gemacht werden.

Ein Weg für Microsoft Windows wäre der Einsatz von Cygwin der Firma Red Hat. Mit Cygwin ist es möglich, Programme die eigentlich unter Linux, BSD, usw. betrieben werden, dann unter Windows zu betreiben. Schon aus dieser kurzen Beschreibung ist ersichtlich, dass die Einführung einer weiteren Zwischenschicht, in Form von Cygwin auch eine weitere Quelle für Probleme darstellen kann. Aber auch solche Entwicklungsumgebungen haben ihre Berechtigung. Einige spezielle Werkzeuge der Hersteller von Embedded Systemen sind allerdings nur für Windows verfügbar.

4.2 Vorbereitung Netzwerkbooten

Zum Booten über ein Netzwerk werden auf dem Hostsystem im Voraus spezielle Server-Dienste benötigt. Mindestens ist ein TFTP- und ein NFS-Server notwendig. Vom TFTP-Server wird ein für das Target passender Kernel und über den NFS-Server ein Root Filesystem bereitgestellt. Zusätzlich kann es sinnvoll sein, einen DHCP-Server zu installieren. Gerade wenn für mehrere Targets entwickelt wird, kann anhand der MAC-Adresse eine entsprechende Konfiguration übertragen werden, ohne in die Konfiguration des Targets selbst

eingreifen zu müssen. Der genaue Ablauf des Prozesses wird in nachstehender Grafik verdeutlicht.

4.2.1 Prozess des Netzwerkbootens

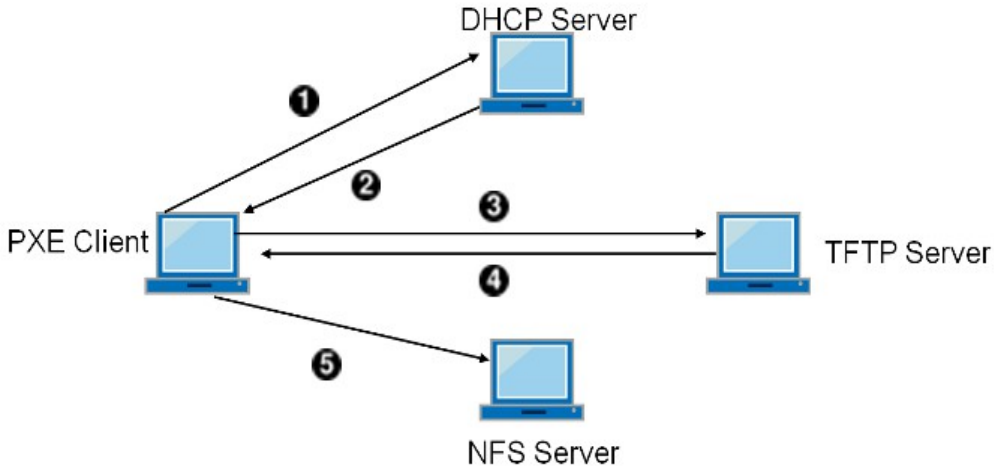

Abbildung 10: PXE Booting process with NFS root mount – Quelle: [FBSDPXE]

In obiger Grafik würde es sich beim „PXE Client“ um das Embedded-System handeln. DHCP-, TFTP- und NFS-Server können wahlweise auf verschiedenen oder dem selben Rechner installiert sein.

Beschreibung des Prozesses

1. Der Bootloader des Embedded-Systems führt einen DHCPDISCOVER durch und sucht nach einem DHCP-Server
2. Der DHCP-Server antwortet mit einer IP-Adresse und den Parametern `next-server`, `filename`, und `root-path`
3. Der Boot-Loader sendet eine Anfrage an den TFTP-Server (`next-server`) und fordert `filename` von diesem an
4. Der TFTP-Server antwortet und sendet `filename`
5. Der Bootloader führt `filename`, z.B. den Kernel, aus. Der Kernel versucht dann das Root Filesystem vom NFS-Server (`root-path`) zu mounten.[32]

32 [FBSDPXE]

4.2.2 Installation der Serverdienste

Alle benötigten Dienste sind im Paket-Repository von Debian vorhanden. Die Installation erfolgt mittels des Paketmanagers apt-get und benötigt root-Rechte auf dem Hostsystem. Benötigt werden die Pakete `isc-dhcp-server`, `tftpd-hpa` und `nfs-kernel-server`.

```
$ apt-get install isc-dhcp-server tftpd-hpa \
     nfs-kernel-server
```

4.2.3 Vorbereitung DHCP-Server

Als Erstes muss die Konfiguration des DHCP-Servers unter `/etc/dhcp/dhcpd.conf` vorgenommen werden. In einem Netzwerksegment kann nur ein DHCP-Server betrieben werden. Ist dies bereits der Fall, sollte ein zweites Netzwerkinterface zur direkten Verbindung von Target und Hosts-System verwendet werden. Auch sollte die MAC-Adresse des Targets, der Name des zu verwendenden Host-Interfaces und die Netzklasse bekannt sein.[33] Die Basiskonfiguration sieht dann wie folgt aus:

```
$ cat /etc/dhcp/dhcpd.conf

allow bootp;
default-lease-time 600;
max-lease-time 7200;

subnet 192.168.1.0 netmask 255.255.255.0 {
                range 192.168.1.250 192.168.1.253;
                option routers 192.168.1.1;
                option domain-name "foo.com";
                option domain-name-server 192.168.1.1;
```

33 [DEBNB]

```
}

host tqma28 {
                hardware ethernet 00:D0:93:25:3A:10;
                fixed-address 192.168.1.200;
                next-server 192.168.1.77;
                filename "tqma28/uImage-tqma28";
                option root-path "/exports/rootfs/tqma28";
}
```

Werden im Hostsystem mehrere Netzwerkkarten verwendet, muss unter `/etc/default/isc-dhcp-server` die Variable `INTERFACES=""` auf das entsprechende Netzwerkinterface, z.B. `eth1`, eingestellt werden.
Im Anschluss kann der Server gestartet werden. Dies erfolgt mit dem Kommando:

```
:# /etc/init.d/isc-dhcp-server start
```

Die Ausgabe sollte einen fehlerfreien Start melden:

```
Starting ISC DHCP server: dhcpd.
```

Sollten beim Start Fehlermeldungen auftreten, kann die Log-Datei `messages` unter `/var/log` Aufschluss geben.

4.2.4 Vorbereitung TFTP-Server

Vom TFTP-Server wird das Verzeichnis `/srv/tftp/` genutzt. Um auf einfache Art zwischen verschiedenen Targets zu unterscheiden, wird hier ein Unterverzeichnis mit der Bezeichnung des Targets angelegt. Hier wird später der Linux-Kernel zu finden sein, der vom Embedded-System gestartet werden soll:

```
:# mkdir -p /srv/tftp/tqma28
```

Ein einfacher Test des Dienstes kann mit dem Befehl `tftp` auf dem Hostsystem durchgeführt werden. Dazu wird im Verzeichnis `/srv/tftp/tqma28` eine Datei mit z.B. `touch` erstellt. Diese wird dann versucht vom Hostsystem mittels TFTP wieder auf sich selbst zu übertragen:

```
:# touch /srv/tftp/tqma28/test
:# echo "Test" > /srv/tftp/tqma28/test
:# cd /tmp
:# tftp
(to) 127.0.0.1
tftp> get tqma28/test
tftp> quit
```

War die Übertragung erfolgreich, kann von einer korrekten Funktion des TFTP-Servers ausgegangen werden. Später sollte zusätzlich ein manueller Test vom Target aus erfolgen.

4.2.5 Vorbereitung NFS-Server

Nun muss noch die Auslieferung des Root Filesystems vorbereitet werden. Dazu wird ein Verzeichnis unter `/exports/rootfs/tqma28` erstellt und der NFS-Server über die Konfigurationsdatei `/etc/exports` angewiesen, dieses bereitzustellen:

```
$ mkdir -p /exports/rootfs/tqma28
$ echo " /exports/rootfs/tqma28 \
     *(rw,no_root_squash,async,no_subtree_check)" \
     >> /etc/exports
```

Der NFS-Server wird anschließend einem Neustart unterzogen:

```
$ /etc/init.d/nfs-kernel-server restart
```

und mit dem Kommando `exportfs` überprüft. Als ein weiterer Test kann vom Hostsystem versucht werden, diesen NFS-Share zu mounten:

```
$ mount -t nfs 127.0.0.1:/exports/rootfs/tqma28 /mnt
$ mount
[…]
127.0.0.1:/exports/rootfs/tqma28 on /mnt type nfs
(rw,addr=127.0.0.1)
[…]
```

Die Vorbereitungen sind damit abgeschlossen. Die Verwendung im Zusammenspiel mit dem Target und dessen Bootloader wird im Kapitel 4.4.3 - Netzwerkbooten genauer beschrieben.

4.3 Serielle Verbindung und Konsole

Für den Verbindungsaufbau mit dem Baseboard wird ein serieller Port am Hostrechner, ein serielles Kabel und ein Terminalprogramm verwendet. Unter Linux existiert eine Vielzahl von verschiedenen Terminalprogrammen. Es ist dem Anwender selbst überlassen, ob er nun eine Konsolenversion oder eine grafische Variante verwenden möchte. Beschrieben wird hier nur das textorientierte Picocom. Alternativen wären z.B. Minicom oder mit grafischer Unterstützung Cutecom. Alle drei können einfach aus dem Paket-Repository von Debian installiert werden. Zur Installation werden auf dem System root-Rechte benötigt. Die Installation erfolgt mit dem Kommando:

```
:# apt-get install picocom
```

Der Paketmanager lädt darauf hin das Programm mit allen benötigten Abhängigkeiten von den Debian-Servern herunter und installiert sie.

Für den Verbindungsaufbau müssen an Picocom einige Parameter übergeben werden. Dazu gehören die zu verwendende serielle Schnittstelle und die Baudrate. Im folgenden Fall besaß das Hostsystem eine on-Board serielle Schnittstelle mit dem Device-Namen `/dev/ttyS0`. Sollte ein USB-RS232-Converter eingesetzt werden, kann diese Bezeichnung abweichen. Typisch wäre beispielsweise `/dev/ttyUSB0`. Das resultierende Kommando lautet:

```
:# picocom -b 115200 /dev/ttyS0
```

Nachdem die physische Verbindung zwischen Host- und Zielsystem (X32B) besteht und das Terminalprogramm gestartet ist, kann nun die Spannungsversorgung des Baseboards eingeschaltet werden. Es meldet sich der Bootloader mit nachfolgender Ausgabe:

```
Terminal ready
?
U-Boot 2009.08 (Jun 24 2011 - 15:26:35)

Freescale i.MX28 family
CPU:   454 MHz
BUS:   151 MHz
EMI:   205 MHz
GPMI:   24 MHz
DRAM:  128 MB
MMC:   IMX_SSP_MMC: 0, IMX_SSP_MMC: 1
In:    serial
Out:   serial
Err:   serial
Board: TQMa28
        on an MBa28 carrier boardSer#:  TQMa28-PROTO1.100
33163188
Net:   FEC0, FEC1
Hit any key to stop autoboot:  0
=>
```

Tabelle 7: Ausgabe Bootloader

Diese Ausgabe in Tabelle 7 lässt bereits einige Rückschlüsse zu, die für den weiteren Verlauf von Interesse sind.

Eine wichtige Information ist, welcher Bootloader verwendet wird. In diesem Fall ist es der U-Boot Loader (Das U-Boot) der Firma DENX Software Engineering in der Version 2009.08. Er spielt eine wichtige Rolle und findet in einem eigenen Unterpunkt gesonderte Betrachtung. Der Bootloader-Version folgen einige Kerndaten des Boards.

Nach einer kurzen Wartezeit beginnt der Bootloader einen Linux-Kernel zu

starten. Letztlich endet die Ausgabe bei einem Login-Shell. Der Login ist mit dem Benutzernamen „root“ und dem Passwort „root“ möglich.

4.4 Bootprozess

4.4.1 Bootmedium eMMC Flash

Im Auslieferungszustand befindet sich auf dem eMMC-Flash des Moduls bereits ein vorinstalliertes Linux System. Auch die Booteinstellungen des Baseboards sind auf das Booten von eMMC voreingestellt (vgl. Tabelle 6). Nach Inbetriebnahme startet das System den Bootloader (U-Boot) aus dem eMMC-Flash, dieser wiederum lädt anschließend den Linux-Kernel in den Speicher und führt ihn aus. Dies wird auch als Standalone Setup bezeichnet, verlangsamt jedoch den Entwicklungszyklus durch fortwährendes beschreiben des eMMC Speichers. Bei jeder neuen Softwareversion und jedweder Softwareänderung muss der Speicher neu beschrieben werden. Zudem werden spezielle Tools auf Hostsystem und / oder dem Board selbst benötigt um diesen Vorgang durchzuführen. Über diese Tools und die Funktionsweise des Boards selbst müssen zur Durchführung weitreichende Kenntnisse vorhanden sein.

4.4.2 Bootmedium SD Karte

Zum Betrieb von einer extern eingesteckten SD-Karte muss diese entsprechend vorbereitet sein. Bei Auslieferung befindet sich eine solche Karte im Lieferumfang. Diese wird wird in den Kartenslot auf der Rückseite des STK-MBa28 Baseboards eingesteckt. Zudem muss die Konfiguration der Bootpins (entsprechend Tabelle 6) geändert werden, bevor das Board wieder mit Spannung versorgt wird.

Diese Betriebsart bietet den Vorteil, verschiedene Systeme schnell austauschen und testen zu können, was sich, im Gegensatz zum beschreiben des eMMC, deutlich einfacher gestaltet. Jedoch wird bei dieser Methode der Entwicklungszyklus, ebenfalls durch wiederkehrendes

beschreiben der SD-Karte, verlangsamt. Die Methode zeigt ihre Vorteile bei einem Deployment nach der Entwicklungsphase. Durch einfachen Austausch des Bootmediums kann ein Update der Systemsoftware durchgeführt werden ohne das weitreichende Systemkenntnisse notwendig sind. Diese Methode wird als Removable Storage Setup bezeichnet.
Wie eine SD-Karte für den Betrieb vorbereitet werden muss, wird in Kapitel 6 und 7 detailliert beschrieben.

4.4.3 Netzwerkbooten

Eine weitere Möglichkeit das Board mit Software zu versorgen bietet das Booten über Netzwerk. Hierzu wird eine Kombination aus DHCP-, TFTP- und NFS-Server verwendet. Für diese Betriebsart ist Voraussetzung, dass bereits ein lauffähiger U-Boot, entweder von eMMC oder SD-Karte, gestartet wird. Dem Bootloader muss dann der entsprechende Server, in diesem Fall das Hostsystem, angegeben werden, von dem Kernel und Root Filesystem geladen werden sollen. Alternativ besteht die Möglichkeit auf eine explizite Spezifikation des Servers zu verzichten und die Weitergabe der Informationen einem DHCP-Server zu überlassen.
Auf den ersten Blick scheint diese Methode durch das Zusammenspiel verschiedener Dienste auf dem Hostsystem und entsprechender Konfiguration aufwendiger. Dies gilt jedoch nur für die Ersteinrichtung. Sind die Serverdienste einmal korrekt konfiguriert, lässt sich die Entwicklung deutlich beschleunigen. Fortwährendes beschreiben des eMMC- oder SD-Speichers entfällt. Alle Anpassungen an Kernel und Root Filesystem lassen sich auf dem Hostsystem durchführen und an das Board weiterreichen. Dies kann zu einer Beschleunigung des Entwicklungszyklus maßgeblich beitragen. Für den produktiven Einsatz beim Endkunden ist sie jedoch weniger geeignet, da das System nur in Verbindung mit einem entsprechenden Server zu verwenden ist. Ein Standalone Betrieb ist nicht möglich. Diese Betriebsart wird als Linked Setup bezeichnet.

Im weiteren Verlauf werden die im TQ Support-Wiki bereitgestellten Prebuild-Archive verwendet. Benötigt wird das Archiv `tqma28-mfg-linux-update-20110911-01.zip` und `rootfs-squeeze-20110911-01.tgz`. Beide sind auf der Seite Download-Links[34] zu finden.

Im Archiv `tqma28-mfg-linux-update-20110911-01.zip` findet sich nach dem Entpacken ein Kernel-Image und ein minimales Root Filesystem. Alternativ kann auch das in `rootfs-squeeze-20110911-01.tgz` enthaltene Root Filesystem verwendet werden. Hierbei handelt es sich um ein Debian Squeeze for ARM.

Das Kernel-Image findet sich unter `TQMa28-Linux-Update/Linux/files/uImage` und muss auf dem Hostsystem in das vorbereitete TFTP-Verzeichnis `/srv/tftp/tqma28/` kopiert werden. Dabei ist darauf zu achten, dass der Dateiname korrekt ist. Je nach Konfiguration des Bootloaders muss dieser von `uImage` mittels `mv` in `uImage-tqma28` umbenannt werden.

Nun ist es bereits möglich den Kernel über das Netzwerk zu laden und auszuführen. Dazu wird eine Terminalsitzung zum Target aufgebaut und die Kommandofolge `run addtty; dhcp; bootm;` eingegeben. Die Ausgabe sieht dann wie folgt aus:

```
=> run addtty
=> dhcp
BOOTP broadcast 1
DHCP client bound to address 192.168.1.200
Using FEC0 device
TFTP from server 192.168.1.77; our IP address is
```

34 http://support.tq-group.com/doku.php?id=sys:downloads#mfg_profil_linux_installation

```
192.168.1.200
Filename 'tqma28/uImage-tqma28'.
Load address: 0x42000000
Loading:
#################################################################
#################################################################
################################
done
Bytes transferred = 2259356 (22799c hex)

=> bootm
## Booting kernel from Legacy Image at 42000000 ...
   Image Name:   Linux-2.6.35.3
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2259292 Bytes =  2.2 MB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK
Starting kernel ...
[...]
```

run addtty	Setzt Kernel-Parameter für die Ausgabe auf serieller Schnittstelle ttySP3
dhcp	Baut Verbindung zum DHCP-Server auf und lädt den Kernel uImage-tqma28 per TFTP in den Speicher
bootm	Führt den Kernel aus dem Speicher aus

Nachdem noch kein passendes Root Filesystem verfügbar ist, führt dies zu einer Kernel Panic und der Startvorgang wird abgebrochen. Zum vollständigen Netzwerkboot muss dieses nun noch in den Exportpfad des NFS-Servers kopiert werden. Es besteht die Auswahl zwischen dem minimalen System oder dem Debian System. Sie sind unter TQMa28-Linux-Update/Linux/files zu finden:

rootfs.tar.bz2	Minimales System
rootfs-debian.tar.bz2	Debian System

Die Installation kann im Export-Verzeichnis in zwei Unterverzeichnisse geschehen. Damit kann einfach, mittels Änderung des root-path, zwischen den beiden gewechselt werden.

```
:# mkdir -p /exports/rootfs/tqma28/minimal
:# mkdir -p /exports/rootfs/tqma28/debian
:# tar xfvj rootfs.tar.bz2 \
     -C /exports/rootfs/tqma28/minimal/
:# tar xfvj rootfs-debian.tar.bz2 \
     -C /exports/rootfs/tqma28/debian/
```

Wiederum wird eine Terminalverbindung zum Target aufgebaut und die folgende Kommandofolge abgesetzt:

```
=> dhcp
=> setenv rootpath /exports/rootfs/tqma28/debian
=> setenv bootargs $bootargs ip=dhcp
=> run addtty
=> run addnfs
=> bootm
```

Um den `rootpath` zu setzten und die Kernel-Parameter um die

entsprechenden Netzwerk- und NFS-Einstellungen zu ergänzen, kommen die Kommandos `setenv` und `run addnfs` zum Einsatz.

Mit `setenv` können im U-Boot Umgebungsvariablen gesetzt werden. In diesem Fall sind das die beiden Variablen `rootpath` und `bootargs`.
`run addnfs` führt ein Makro-Kommando aus, dass die Kernel-Parameter um die entsprechenden Informationen aus dem Environment ergänzt. Selbiges führt `run addtty` im Bezug auf die Konsolenausgabe durch. Beide Makro-Kommandos sind in der TQ-Umgebung bereits definiert. Deren konkrete Bedeutung kann aus dem Listing des *Anhang A – U-Boot Environment* entnommen werden.

Vor dem Kommando `bootm` kann mittels `printenv bootargs` die Parameterübergabe an den Linux-Kernel überprüft werden:

```
=> printenv bootargs
bootargs=console=tty0 console=ttySP3,115200 ip=dhcp
root=/dev/nfs rw ramdisk_size=16384
nfsroot=192.168.1.77:/exports/rootfs/tqma28/debian,v3,tcp
```

Der Kernel wird angewiesen, `ttySP3` als Ausgabe mit `115200` Baud zu verwenden. Zudem soll die IP-Addresse vom DHCP-Server bezogen werden. Über `root=/dev/nfs` wird vom Kernel gefordert, das Root Filesystem über NFS zu beziehen und der explizite NFS-Pfad zum Server wird mit `nfsroot=[…]` übergeben.

Eine Übersicht der in U-Boot verfügbaren Kommandos ist über die Konsole mit dem Kommando `help` zu betrachten. Zudem ist eine Beschreibung zu jedem Befehl in der U-Boot Dokumentation, im Quellcodeverzeichnis in der Datei README, zu finden.

5 Analyse des TQ Systems

5.1 Quellcode und SVN-Repository

Zur Analyse des Systems, sowie für den folgenden Nachbau dieses, werden die Quellen und Anpassungen von TQ-Systems herangezogen. Einige Informationen dazu sind im TQ-Support Wiki[35] zu finden.
TQ verwendet zur Versionskontrolle Subversion. Dabei handelt es sich um ein Opensource Versionskontroll-System. Einführend wird hier nur eine kurze Übersicht über die verwendeten Befehle gegeben. Eine detaillierte Dokumentation zur Verwendung ist auf der Webseite[36] des Projektes zu finden. Die Installation von Subversion kann unter Debian über das Paketrepository erfolgen:

```
$ apt-get install subversion
```

Übersicht verwendeter Subversion Kommandos:

`svn help`
: Hilfe zu Subversion

`svn checkout http://projekt1/trunk/ /home/user/projekt1`
: erstellt eine lokale Arbeitskopie

`svn diff`
: zeigt Unterschiede seit dem letzten Commit

`svn update`
: Arbeitskopie auf die Serverversion aktualisieren

Zum Erstellen einer Arbeitskopie wird das Kommando `svn checkout` verwendet. Für das benötigte TQ-Repository wird im Detail folgendes

35 [TQWIKI]

36 http://subversion.apache.org/docs/

Kommando verwendet:

```
svn co --username <user> \
https://svn.reccoware.de/tq/system/trunk tq
```

Dieses Repository ist nicht öffentlich zugänglich, weshalb ein Benutzername und ein Passwort für diesen Vorgang benötigt wird. TQ-Systems weist im Support-Wiki auf diesen Umstand hin. Somit ist eine Kontaktaufnahme per E-Mail notwendig um einen entsprechenden Zugang zu bekommen.
Nach erfolgreichem Checkout finden sich die Quellen im Unterverzeichnis `tq/`.

Das `tq/` Verzeichnis weist folgende Struktur auf:

`bin`	Tools zum erstellen eines bootbaren Images
`boot`	Quellen, Patches und Tools zum U-Boot
`build`	Skripte zum Einrichten der Build-Umgebung
`Documentation`	Dokumentationen
`flash`	Tools zur Installation der Images
`hosttools`	Debugger und Tools zum erstellen des Images
`images`	gepackte Root Filesysteme
`install`	Beispiele
`kernel`	Kernel und Kernel-Quellen
`local`	Konfigurationen für das Root Filesystem
`Makefile`	Makefile
`programs`	Beispiele QT und GPIO
`rootfs`	Debian squeez
`tmp`	tmp-Verzeichnis

Somit sind alle zum Betrieb notwendigen Bestandteile in diesem Arbeitsverzeichnis vorhanden.

5.2 U-Boot Loader

Von TQ-Systems wird die U-Boot Version 2009.08 (Sep 11 2011 – 17:36:38), sowohl auf eMMC als auch auf der SD-Karte, eingesetzt. Es handelt sich dabei nicht um eine unveränderte Mainlineversion. Zahlreiche Änderungen und Anpassungen wurden von TQ-Systems daran vorgenommen. Insgesamt wurden 223 Patches auf den Quellcode angewendet, welche im SVN-Repository unter `boot/patches` zu finden sind. Der überwiegende Teil, ca. 200, sind Patches der Firma Freescale. Hierbei handelt es sich größtenteils um Anpassungen, die die i.MX-Prozessoren im Allgemeinen betreffen. Weitere 20 Patches stammen von TQ und sind Anpassungen an deren Hardwareplattform. Weitere 33 Patches sind im Verzeichnis `boot/u-boot/` zu finden und betreffen das Makefile von U-Boot.

Aus dem SVN-Repository von TQ-Systems lässt sich schließen, dass die Anpassungen auf der Implementierung des Freescale MX28EVK-Boards basieren. Die beiden Boards weichen jedoch im Design in manchen Punkten voneinander ab. So verwendet das Freescale MX28EVK z.B. DUART für die serielle U-Boot Ausgabe, während TQ-Systems bei ihrem Board AUART3 vorsieht. Zudem verwendet Freescale zwei externe SD-Karten, hingegen benutzt das TQ-Board eine externe SD-Karte und einen auf dem Modul integrierten eMMC-Speicher.

Diese und einige andere Unterschiede im Design der Hardware, spiegeln sich in den verwendeten Patches wieder und werden beim nachfolgenden Bau einer aktuellen Mainlineversion von U-Boot eine Rolle spielen.
Bei der kompilierten Version von diesem U-Boot handelt es sich um eine *ELF 32-bit LSB shared object, ARM, version 1 (SYSV), statically linked, not stripped* und weist eine Größe von 565 kB auf. Übersetzt wurde er mittels *ELDK 4.2 arm: Build 2008-11-24 armVFP: Build 2008-11-24*.

5.3 Linux Kernel

Die ab Werk verwendete Linux Kernelversion ist 2.6.35.3. Das Kommando `file` gibt dazu näheren Aufschluss. Eine ähnliche Ausgabe liefert auch das Programm `mkimage -l uImage-tqma28`, welches im Quellcode-Verzeichnis des U-Boot Loaders (unter `tools/`) zu finden ist.

```
$ file uImage-tqma28
uImage-tqma28: u-boot legacy uImage, Linux-2.6.35.3,
Linux/ARM, OS Kernel Image (Not compressed), 2259292 bytes,
Sun Sep 11 17:40:13 2011, Load Address: 0x40008000, Entry
Point: 0x40008000, Header CRC: 0x9BEC98D1, Data CRC:
0x8F90D56C
```

Diese Ausgabe kann verwendet werden um eine erste Einschätzung des verwendeten Kernels zu treffen. Anhand der Versionsnummer kann bereits darauf geschlossen werden, dass die verwendete Version bereits etwas antiquiert ist. Erschienen ist sie (laut kernel.org) am 20.08.2010 – aktuell (August 2012) ist die Version 3.5.0. Dieser Umstand könnte sich, je nach Einsatzzweck und Zugänglichkeit des Systems, z.B. im Bezug auf die Systemsicherheit als nachteilig herausstellen. Es liegt die Vermutung nahe, dass zahlreiche, bekannte Sicherheitslücken in dieser Version bestehen und nicht behoben sind. Bei einer kurz angelegten Suche über www.cvedetails.com lassen sich mit den Suchbegriffen „Linux Kernel“ und „2.6.35.3“ bereits 124 Vulnerabilities finden, von denen jedoch sicher nicht alle eine relevante Bedrohung darstellen.

Aus obiger Ausgabe lässt sich zudem die Größe des Kernel mit 2,25 MB und die Ziel-Architektur ARM bestimmen. Zudem ist das Kernel-Image nicht gepackt (Not compressed). In manchen Fällen ist eine Kompression des Kernel-Images aus Platzgründen sinnvoll, jedoch wird die für den Startvorgang benötigte Zeit

um die des Entpackens verlängert. Auch sollte die Systemperformance ausreichend sein. Bei Targets mit sehr schwacher CPU kann der Systemstart durch das Entpacken enorm verzögert werden. Wird die Start-up Zeit als kritisch betrachtet und ist ausreichend Speicherplatz vorhanden, kann ein nicht komprimiertes Image einen Zeitvorteil darstellen.

Weitere Informationen lassen sich über das System auf dem Target selbst bestimmen. Die Ausgabe des Kommandos `cat /prov/version` liefert weitere Informationen:

```
cat /proc/version
Linux version 2.6.35.3 (weo@turing.recco.de) (gcc version
4.2.2) #1 PREEMPT Fri Jun 24 15:30:38 CEST 2011
```

Hieraus lässt sich der verwendete Compiler, hier gcc, und dessen Version 4.2.2 schließen.

Bei der Durchsicht des im SVN-Repository von TQ-Systems verwendeten Linux Quellcode fällt auf, dass vor dem Bau des Kernels ca. 600 Patches auf diesen angewendet werden. Ähnlich wie beim U-Boot Loader, sind von diesen ca. 600 Patches ebenfalls der Großteil von der Firma Freescale. Rund 30 der Patches wurden von TQ-Systems erstellt. Diese betreffen u.a. die I^2C Initialisierung, die Netzwerkinterfaces, Unterstützungen für einige LCD Displays und Fixes im Bereich der MMC Ansteuerung sowie der AUART Schnittstellen.

5.4 Root Filesysteme

Für das Board werden von TQ-Systems auch im SVN-Repository zwei verschiedene Root Filesysteme angeboten. Ein minimales System und ein Debian System. Diese beiden werden im Folgenden auf ihre Eigenschaften und den Leistungsumfang hin untersucht. Es wird jeweils dargestellt, mit welchen

Methoden diese erstellt wurden, welche Größe sie aufweisen und welchen Leistungsumfang sie bieten. Zudem wird jeweils die Verfügbarkeit von *bash* zur Konsolenbedienung, *Xorg* für grafische Darstellung, *python* und *gcc* zur Programmierung und zum nativen Kompilieren untersucht.

5.4.1 Minimal System

Folgt man den Ausführungen in TQ Support Wiki[37] ist festzustellen, dass dieses minimale System mit ELDK 4.2 erstellt wurde. Der vom System belegte Speicherplatz wurde auf dem Target mit dem Kommando `df -h` ermittelt und beläuft sich auf rund 66 MB. Zur Laufzeit werden 12 MB Arbeitsspeicher belegt.

```
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/root             1.4G   66M  1.2G   6% /
[…]

$ free -m
       total  used  free  shared  buffers  cached
Mem:   121    12    108   0       0        5
[…]
```

Verfügbare Dienste u.a.:

dropbear	SSH-Server
rsyslogd	Server für Nachrichtenlogging
udevd	Device Manager
dhclient	DHCP Client Tools

37 [TQWIKI]

Kommandos

Dem Anwender stehen rund 120 Kommandos zur Verfügung. Diese gliedern sich in:

/bin	ca. 40 Kommandos
/sbin	ca. 30 Kommandos
/usr/bin	ca. 30 Kommandos
/usr/sbin	ca. 5 Kommandos
/usr/local/bin	ca. 10 Kommandos
/usr/local/sbin	2 Kommandos

Verfügbarkeit essentieller Programme u.a.:

bash	Version 4.1.5
Xorg	Version 1.1.7
python	N/A
gcc	N/A

Paketrepository

Ein Paketrepository, zur einfachen Installation zusätzlicher Software, ist nicht vorhanden.

Zusammenfassend erscheint dieses Root Filesystem als eine minimale Basisausstattung. Ein Betrieb damit ist möglich, jedoch lädt es nicht zu weitreichenden Experimenten ein. Natives Kompilieren von Programmen ist bedingt durch das Fehlen eines C-Compilers nicht ohne weiteres möglich. Auch gestaltet sich die Installation zusätzlicher Software durch das Fehlen eines Paketmanagers und -repository schwierig. Eine Erweiterung ist nur

über den Weg des crosscompilierens auf dem Hostsystem und anschließender Übertragung auf das Target möglich.

5.4.2 Debian System

Explizit handelt es sich bei dem Debian Root Filesystem um Debian GNU/Linux for ARM in der Version 6.0 (Squeeze). Der genaue Erstellungsprozess ist anhand des Root Filesystems nicht zu identifizieren. Es liegt die Vermutung nahe, dass die Tools `debootstrap`[38] bzw. `multistrap`[39] verwendet wurden. Der belegte Speicherplatz beläuft sich auf 736 MB und die Auslastung des Hauptspeichers auf 74 MB.

Verfügbare Dienste u.a.:

OpenBSD Secure Shell	SSH-Server
udevd	Device Manager
nfs-kernel-server	NFS-Server
cron	Zeitgesteuerte Tasks
ISC DHCP server	DHCP-Server
ALSA	Soundausgabe

Kommandos

/bin	ca. 90 Kommandos
/sbin	ca. 120 Kommandos
/usr/bin	ca. 650 Kommandos
/usr/sbin	ca. 120 Kommandos
/usr/local/bin	ca. 20 Kommandos
/usr/local/sbin	2 Kommandos

38 http://wiki.debian.org/Debootstrap

39 http://wiki.debian.org/Multistrap

Verfügbarkeit essentieller Programme:

bash	Version 4.1.5
Xorg	Version 1.7.7
python	Version 2.6.6
gcc	Version 4.4.5

Paketrepository

Sowohl Paketmanager dpkg wie auch apt und aptitude sind vorhanden. Bei apt handelt es sich um das Advanced Packaging Tool das zur eigentlichen Paketverwaltung dpkg heranzieht. Aptitude wiederum bietet ein interaktives Frontend für apt. Das Paketrepository umfasst rund 30.000 Pakete. Diese können beispielsweise mit dem Kommando `apt-get install <PAKET>` installiert werden. Ca. 350 Pakete sind innerhalb des Systems bereits installiert.

Bei dem Debian Root Filesystem handelt es sich um ein nahezu vollwertiges Debian System. Sämtliche essentielle Dienste und Programme sind bereits vorhanden oder können über das Paketsystem nachinstalliert werden. Zudem ist es möglich, nativ auf dem Target zu kompilieren.

5.5 Bewertung

Umfassend betrachtet bietet TQ-Systems mit der Kombination aus U-Boot Loader, Linux-Kernel und zwei Root Filesystemen eine sofort einsetzbare Softwareumgebung für das STK-MBa28. Beide bereitgestellten Systeme können für einen produktiven Einsatz genutzt werden. Im Hinblick auf die OpenSource-Philosophie zeigen sich jedoch gewisse Defizite.
So ist z.B. nicht erkennbar, ob die durchgeführten Änderungen an U-Boot Loader und Linux Kernel wieder in deren Mainlineversionen zurückfließen.

Weder in den aktuellen Versionen von U-Boot, noch in denen vom Linux Kernel sind Konfigurationen für das TQ-Board zu finden. Damit liegt der Verdacht nahe, dass eine Rückkopplung zu den Hauptentwicklungszweigen fehlt. Das hat zur Folge, dass keine anderen oder neueren Versionen der Kernelemente Bootloader und Kernel, ohne umfangreiche Anpassungen an ihnen durchzuführen, verwendet werden können. Eine einfache Versionsanhebung der beiden Komponenten ist somit nicht möglich. Weniger kritisch ist die Situation bei den Root Filesystemen. Hier können durchaus auch aktuellere oder eigene Varianten eingesetzt werden.

6 Reproduktion des TQ-Systems

Im folgenden Abschnitt wird nun versucht alle zum Betrieb nötigen Bestandteile, ohne Rückgriff auf die Prebuild-Archive, zu reproduzieren. Es wird die Vorbereitung des Hostsystems und der Bauvorgang beschrieben. Zuletzt wird die Erstellung einer SD-Karte als bootbares Medium und die Installation der einzelnen Bestandteile auf dieser beschrieben. Die nachfolgenden Ausführungen folgen in weiten Teilen der Installationsanleitung von TQ-Systems[40].

6.1 Vorbereitung des Hostsystems

Zuerst werden auf dem Hostsystem Cross Development Tools benötigt. Benutzt wird ELDK 4.2 der Firma Denx Software Engeneering. Das Embedded Linux Development Kit (kurz ELDK) bietet eine Zusammenstellung von GNU Cross Development Tools wie Cross-Compiler, Debugger, Pre-Builds, etc. für eine Target-Architektur[41]. Diese Tools sind notwendig, da die Architektur der Hostplattform und die des Targetsystems meist voneinander abweichen. Beispielsweise kann das Hostsystem auf einer x86-Architektur basieren, während das Targetsystem eine ARM-Architektur nutzt. Ein für x86-Architektur

40 http://support.tq-group.com/doku.php?id=tqma28:build:installation

41 [ELDK12] - Kapitel: 3. Embedded Linux Development Kit

übersetztes Programm ist, bedingt durch sich z.B. unterscheidende Befehlssätze, nicht auf einem ARM Prozessor lauffähig. Diese Zusammenstellung wird auch als Cross Toolchain oder nur Toolchain bezeichnet und besteht traditionell aus Linker, Assembler, C-Compiler und C-Library mit Headern. Eine geeignete Toolchain selbst zu konfigurieren und zu erstellen stellt eine komplexe Aufgabe dar und erfordert ein genaues Verständnis über das Zusammenspiel der einzelnen Softwarepakete[42]. Es bietet sich daher an, die von DENX Software Engineering vorbereitete Toolchain zu verwenden.

Die Installation der ELDK Toolchain erfolgt in drei Schritten. Als erstes müssen die zum Betrieb benötigten Pakete installiert werden. Darauf folgt die Einrichtung von ELDK selbst und letztlich muss die Toolchain auf der Hostplattform bekannt gemacht werden.

6.1.1 Paketabhängigkeiten

Benötigte Pakete können ebenfalls aus dem Paketrepository von Debian installiert werden. Verwendet wird hier wiederum APT.

```
$ apt-get install patch libncurses5-dev tftpd-hpa \
          subversion nfs-kernel-server fakeroot tofrodos
```

Weiter wird von TQ-Systems empfohlen, unter Ubuntu bzw. Debian, folgende Pakete hinzuzufügen:

```
$ apt-get install meld geany geany-plugins tftp-hpa \
          glibc-doc manpages-posix manpages-posix-dev
```

42 [YMBG08] - S. 91 ff

6.1.2 Installation der Buildumgebung

ELDK wird von TQ-Systems als IOS-Image bereitgestellt. Vor dem Herunterladen und der Konfiguration sollte sichergestellt werden, dass genügend freier Speicherplatz auf dem Hostsystem vorhanden ist. TQ gibt als Speicherplatzbedarf mindestens 2 GB in `/opt` und 2 GB in `/home` an. Die Installation erfolgt im Verzeichnis `/opt/eldk`.

Download und anschließende Installation erfolgen mit der Kommandofolge:

```
$ pushd /tmp
$ wget http://repository.reccoware.de/stuff/tq0412/arm-2008-11-24.iso
$ popd
$ sudo mount -o loop,ro /tmp/arm-2008-11-24.iso /mnt
$ pushd /mnt
$ sudo ./install -d /opt/eldk
$ popd
$ sudo umount /mnt
```

Während der Installation tritt die Fehlermeldung „*error: failed to stat …/.gvfs: Permission denied*“ mehrfach auf. Im TQ Support Wiki wird auf diesen Umstand hingewiesen und auch darauf, dass diese wohl „unproblematisch“ sei.

6.1.3 Präsentation der Toolchain im Environment

Zur Verwendung der Toolchain muss diese in der Arbeitsumgebung bekannt gemacht werden. Durch die Installation nach `/opt/eldk` werden die in ELDK enthaltenen Programme nicht automatisch in den Suchpfad eingebunden. Zum Abschluss der Konfiguration wird auch das in 5.1 bereits beschriebene SVN-Repository benötigt.

Zur Vereinfachung und für nachfolgende Schritte wird eine Shell-Variable mit dem absoluten Pfad zur Arbeitskopie des TQ-Repository definiert:

```
$ cd tq; TOPDIR=$(pwd)
```

In der Arbeitsumgebung muss nun die Toolchain selbst präsentiert werden. Dazu werden zwei Skripte verwendet.

```
cd $TOPDIR/build
source setenv-eldk
sudo ./setup-eldk-toolchain.sh
```

Mit `source` wird das Shell-Skript `setenv-eldk` in die aktuelle Shell-Sitzung geladen. Das Skript setzt einige Umgebungsvariablen zur Nutzung der Toolchain. Unter anderem wird der Suchpfad `$PATH` um den Installationspfad von ELDK ergänzt und die für den späteren Bauvorgang notwendige Variable `CROSS_COMPILE` entsprechend gesetzt. Anschließend wird die Toolchain mit `setup-eldk-toolchain.sh` eingerichtet.

Auszug aus setenv-eldk:

```
[…]
export ELDK
export GCC_PATH=$ELDK/usr/bin
export LIB_BASE=$ELDK/arm

export ELDK_ORIG_PATH=$PATH
export PATH=$ELDK/usr/bin:$ELDK/bin:$PATH
export CROSS_COMPILE=arm-linux-gnueabi-
[…]
```

Nach erfolgreicher Installation und Präsentation innerhalb der Arbeitsumgebung kann nun überprüft werden, ob die Toolchain nutzbar ist. Dies kann mit dem Kommando `` `$CROSS_COMPILE`gcc --version `` überprüft werden. Die Ausgabe dieses Kommandos sollte in etwa so aussehen:

```
arm-linux-gnueabi-gcc (GCC) 4.2.2
[...]
```

Anschließend müssen noch einige für den Bauvorgang notwendige Verzeichnisse innerhalb der Arbeitskopie erstellt werden. Dazu wird der Befehl `cd $TOPDIR; make prepare` verwendet.

6.2 Erstellungsprozess

Innerhalb des Arbeitsverzeichnis ist ein komfortables Makefile bereits vorbereitet. Informationen zu den einzelnen Optionen können mit `make help` abgerufen werden. Ein simples absetzten von `make` führt dazu, dass die enthaltenen Archive, z.B. von U-Boot und Linux Kernel, entpackt und anschließend die entsprechenden Patches auf den Quellcode angewendet werden, bevor die Übersetzung gestartet wird. Dieser Prozess nimmt einige Zeit in Anspruch.

Die Ergebnisse dieses Prozesses sind unter `install/tftpboot` zu finden:

rootfs-squeeze.tgz	Root Filesystem Archiv
tqma28-rootfs-debian.tar.bz2	Root Filesystem Archiv
tqma28-rootfs.tar.bz2	Root Filesystem Archiv
u-boot	U-Boot Loader
boot.bin -> u-boot_tqma28.bin	Symbolischer Link
u-boot_tqma28.bin	U-Boot Loader

uImage-tqma28	Kernel Image als uImage
zImage-tqma28	Kernel Image als zImage

Anzumerken ist noch, dass darüber hinaus einige Host-Tools übersetzt werden. Im Speziellen ist hier „elftosb“ zu nennen, da es zur Erstellung des späteren Bootmediums eine Schlüsselrolle einnehmen wird.

6.3 Vorbereitung des Bootmediums

Exemplarisch wird das Vorgehen anhand einer SD-Karte als Bootmedium gezeigt. Das Netzwerkbooten bleibt in diesem Kontext außen vor, da die nachfolgenden Schritte dabei nicht notwendig sind. Nachdem sich die Methoden zur Verwendung des eMMC-Speichers und der SD-Karte ähneln, wird hier der Fokus auf die Benutzung einer SD-Karte gelegt. Es wird in diesem Abschnitt bewusst auf die Verwendung von fertigen Skripten zur Erstellung verzichtet um ein Verständnis über den Vorgang zu erlangen.

6.3.1 Bootvorgang des i.MX28

Notwendig zur Vorbereitung des Bootmediums ist nun eine konkretere Betrachtung des in Abschnitt 3.2 bereits allgemein ausgeführten Bootvorgangs. Dazu wird das i.MX28 Applications Processor Reference Manual[43] herangezogen. Die hier dargestellten Konzepte stammen aus *Kapitel 12 – Boot Modes*. Zur Veranschaulichung dient Abbildung 11. In dieser sind schematisch die Startvorgänge von einem x86- und einem i.MX28-System, vom Einschalten bis zu einer Shell-Sitzung, dargestellt.

43 [IMX28RM]

Bereits aus der Grafik wird deutlich, dass der Prozess auf dem i.MX-Prozessor von dem eines Standard-PCs abweicht.

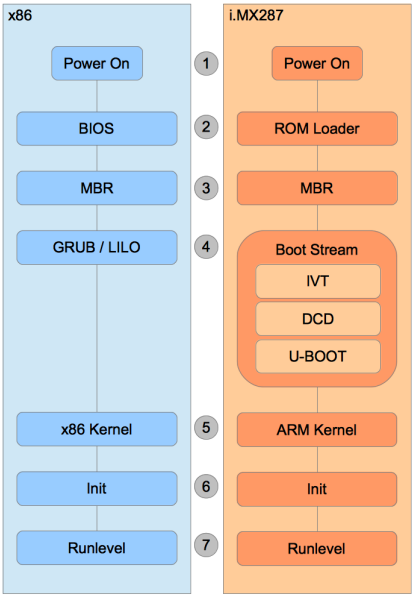

Abbildung 11: Gegenüberstellung Bootprozess x86 - i.MX287

Nach dem Power On (1) wird ein on-Chip Boot ROM (2) gestartet. Dieses ROM unterstützt das Booten von verschiedenen Quellen. Es liest als erstes die Boot-Pins um festzustellen welches Bootmedium verwendet werden soll. Zudem stellt es den USB-Recovery Modus zur Verfügung.

Darüber hinaus wird sekundäres Booten von NAND oder SD/MMC Geräten und eine DCD (Device Configuration Data) Unterstützung angeboten. Als Sicherheitsmechanismen sind verschlüsselte Boot Images und signaturbasiertes HAB (High Assurance Boot) möglich.

DCD (Device Configuration Data)

Bei der DCD handelt es sich um Konfigurationsinformationen für verschiedene Peripherien. Diese Informationen sind im Programm-Image oder auch Boot Stream (4) enthalten. DCD kann beispielsweise verwendet werden um vor der Benutzung des SRAM diesen zu initialisieren.

Verschlüsselte Boot Images

Je nach Anwendungsbereich kann eine der Sicherheitsanforderungen die Verwendung von verschlüsselten Boot Images sein. Der Anwender hat dabei volle Kontrolle über die zu verwendenden Schlüssel. Ein Reverse Engineering, die Analyse des Boot Images von außen oder der Einbruch in das verschlüsselte System wird dadurch erschwert[44].

44 Vgl. [IMX28RM] - 12.8 Constructing Boot Image (SB Files) to Be Loaded by ROM

Signaturbasiertes HAB (High Assurance Boot)

Dieser Mechanismus schützt gegen die Gefahr, einer Veränderung von Systemcode oder Speicher durch einen Angreifer. Zudem verhindert er den Zugriff auf Funktionen, die nicht verfügbar sein sollten. Um diese Richtlinien zu erreichen verwendet HAB digitale RSA Signaturen[45].

Die beiden letzten Verfahren werden von Freescale unter der Bezeichnung „Secure Boot“ geführt.

Auf dem ausgewählten Bootmedium wird als nächstes ein gültiger MBR (Master Boot Record) erwartet. Der MBR wird anhand seiner Signatur (0x55AA) bei der Offset-Adresse 0x1FE des ersten Sektors identifiziert. Die Partitionstabelle wird bei 0x1BE erwartet. Für die Firmware (das Boot Stream Image) wird eine eigene Partition verwendet. Sie wird anhand der MBR_SIGMATEL_ID 'S' identifiziert.
Aus dieser Ausführung des Datenblattes wird nur bedingt klar welche praktische Bedeutung die MBR_SIGMATEL_ID 'S' hat. Konkret handelt es sich dabei um eine Partition vom Typ 0x53. Dies ergibt sich aus der Tatsache, dass 'S' den ASCII-Wert 83 bzw. 0x53 besitzt. Bei den meisten Programmen zur Partitionierung wird dieser Typ unter der Bezeichnung "OnTrack DM6 Aux3" geführt. Für diese Partition wird nun der Begriff „Firmware-Partition“ zur Abgrenzung verwendet.

Im ersten Block dieser Firmware-Partition wird ein BCB[46] (Boot Control Block) erwartet. Durch den BCB wird es möglich, mehr als nur eine Firmware innerhalb der Firmware-Partition vorzuhalten. Für weitere Informationen zum BCB, siehe *[IMX28RM] - 977 ff.*

45 vgl. [IMX28RM] - 12.7 High Assurance Boot (HAB)

46 vgl. [IMX28RM] - 12.11.1 Boot Control Block (BCB) Data Structure

Die so identifizierte Startadresse des User-Codes (Bootloader oder Kernel-Image), in diesem Fall U-Boot (4), wird in den Speicher geladen. Das Boot ROM übergibt die Kontrolle an den User-Code und verlässt den ROM-Kontext.

Vom Bootloader wird der Linux-Kernel (5) aus einer weiteren Partition geladen und die Kontrolle an diesen übergeben. Letztlich wird vom Kernel jegliche weitere Verwaltung der Hardware übernommen und der erste Prozess, der Init-Prozess (6), gestartet. Dieser Prozess orientiert sich nach dem anzustrebenden Runlevel (7) und startet die für den gewünschten Betriebszustand benötigten weiteren Prozesse.

Boot Stream oder Boot Image

Nun müssen noch die eigentlichen Bestandteile des Boot Stream (4) vor der Erstellung der SD-Karte näher betrachtet werden. Ein zur Ausführung geeignetes Boot-Image muss neben dem User-Code drei weitere Bereiche enthalten. Es besteht aus:

- IVT (Image Vector Table)[47]
  Liste von Zeiger, die vom ROM gelesen wird, um festzustellen wo sich die Komponenten innerhalb des Programm-Image befinden
- Boot Data
  Wird beim i.MX28 nicht verwendet.
- DCD (Device Configuration Data)[48]
  Konfigurationsinformationen für den Betrieb
- eigentlicher User-Code
  U-Boot oder Linux Kernel

47 Siehe dazu [IMX28RM] - 12.6.1 Image Vector Table (IVT)

48 Siehe dazu [IMX28RM] - 12.6.2 Device Configuration Data (DCD)

Zur Konstruktion eines solchen Images stellt Freescale ein Tool namens „*elftosb*“ bereit (vgl. Abbildung 12). Es dient auch zur Einrichtung von HAB und Verschlüsselung. Beides bleibt jedoch im weiteren außen vor, um während der Entwicklung zusätzliche Fehlerquellen auszuschließen. Eine nachträgliche Verschlüsselung und / oder Signatur des Images ist problemlos möglich.

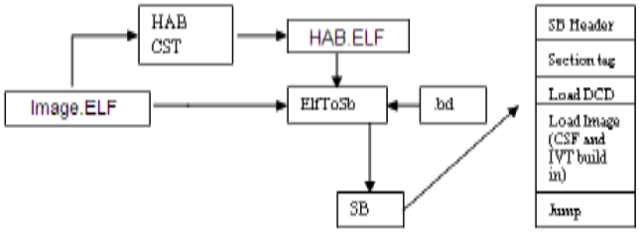

Abbildung 12: i.MX28 Boot Image Generation – Quelle: [IMX28RM] S. 970

Beschrieben werden die oben genannten Bestandteile in einem sogenannten ROM Command Script mit der Dateiendung .bd. In ihm wird festgelegt, in welcher Abfolge die im Image enthaltenen Programmteile geladen und ausgeführt werden. Deutlicher wird dies bei der Betrachtung des `uboot_ivt.bd` ROM Command Script:

```
sources {
     power_prep="./power_prep/power_prep";
     sdram_prep="./boot_prep/boot_prep";
     u_boot="./u-boot";
}
section (0) {

     load power_prep;    // Power Supply initialization
     load ivt (entry = power_prep:_start) > 0x8000;
     hab call 0x8000;

     load sdram_prep;    // SDRAM initialization
     load ivt (entry = sdram_prep:_start) > 0x8000;
     hab call 0x8000;
```

```
    load u_boot;        //  Load and call u_boot ELF ARM
    load ivt (entry = u_boot:_start) > 0x8000;
    hab call 0x8000;
}
```

Aus dem kompilierten U-Boot im ELF-Format und der .bd-Datei wird das SB-Image gebaut. Es enthält nun alle Bestandteile um vom Target ausgeführt zu werden. Eine detaillierte Beschreibung zum .sb Dateiformat und einige Informationen zum Ver- und Entschlüsseln des Images finden sich im Dokument *iMX28_SB_Description*[49].

6.3.2 Einrichtung der SD-Karte

Alle notwendigen Komponenten werden beim Durchlauf des make Prozesses innerhalb der TQ-Arbeitskopie bereitgestellt. Des weiteren werden die Programme `fdisk`, `dd`, `mkfs.ext2` und `mkfs.vfat` verwendet. Zur Erstellung sind fünf Einzelschritte notwendig.

1. ELF in SB-Image umwandeln

   Dazu wird innerhalb der Arbeitskopie in das Verzeichnis `boot/bootlets/imx-bootlets` gewechselt. Hier sind bereits der U-Boot im ELF-Format und das ROM Command Script sowie dessen Abhängigkeiten vorhanden. Mit Hilfe von `elftosb` aus dem Unterverzeichnis `hosttools/` wird nun das eigentliche SB-Image erstellt:

   ```
   $ hosttools/elf2sb/elftosb-10.12.01/bld/linux/elftosb
   -z -f imx28 -c  ./linux_ivt.bd -o imx28_ivt_linux.sb
   ```

49 http://ebookbrowse.com/imx28-sb-description-pdf-d221013169

Beschreibung der Optionen:

-z	Keine Verschlüsselung, darum Zero-Key verwenden
-f	Target-Plattform
-c	ROM Command Script
-o	Dateiname für die Ausgabe des SB-Image

2. Partitionierung der SD-Karte

   Verwendet wird `fdisk`. Wichtig bei der Benutzung ist die Angabe des Parameters `-u` um die Größen in Sektoren statt in Zylindern darzustellen.

```
$ fdisk -u /dev/sdc
```

   Mit sdc ist die SD-Karte bezeichnet. Erstellt werden drei primäre Partitionen. Die erste beginnt bei Sektor 2048 und endet bei Sektor 4096. Der Typ wird auf 0x53 „OnTrack DM6 Aux3“ festgelegt. Diese stellt die Firmware-Partition dar. In die zweite primäre Partition wird das Kernel-Image kopiert. Sie wird vom Typ E „W95 FAT16 (LBA)“ gewählt und muss minimal die Größe des Kernel-Images besitzen.
   Innerhalb der Dritten wird das Root Filesystem installiert. Sie kann den gesamten verbleibenden Platz einnehmen. Als Typ wird 0x83 „Linux“ verwendet.

   Die fertige Partitionierung sollte so aussehen:

```
[…]
Device Boot    Start    End    Blocks    Id  System
/dev/sdc1     2048     4096    1024+     53  OnTrack ...
/dev/sdc2     8000     28480 10240+       e  W95 FAT16 ...
/dev/sdc3    30000     2012159 991080    83  Linux
```

```
[…]
```

Entscheidend ist nur der Startsektor der Firmware-Partition und deren Größe. Sie muss mindestens die Größe der Firmware plus vier Sektoren aufweisen. In den ersten vier Sektoren werden Informationen von der Firmware selbst gespeichert.

3. Schreiben der Firmware

   Das erste Byte des SB-Images wird am Anfang des fünften Sektors erwartet. Es müssen also die ersten vier vor dem Schreiben übersprungen werden. Verwendet wird das Kommando `dd`, um direkt in die erste Partition zu schreiben:

   ```
   $ dd if=imx28_ivt_uboot.sb bs=512 seek=4 of=/dev/sdc1
   ```

   Nach diesem Schritt kann bereits die Funktion der Karte überprüft werden. Es sollte nun der U-Boot Loader gestartet werden.

4. Installation des Linux Kernel-Image

   Die zweite Partition muss vor der weiteren Verwendung mit einem FAT16 Dateisystem versehen werden. Dies geschieht mit dem Kommando `mkfs.vfat -F 16 /dev/sdc2`. Anschließend wird die Partition mit `mount /dev/sdc2 /mnt` gemountet und der Linux Kernel „uImage-tqma28“ aus dem `install/tftpboot` Verzeichnis nach `/mnt/uImage` kopiert. Danach wird `/mnt` aus dem System wieder ausgehängt. Die FAT16 Partition wird benötigt, um im U-Boot komfortabel mit den FAT-Kommandos den Kernel lokalisieren und laden zu können. Sofern U-boot mit einer Unterstützung für EXT-Dateisysteme übersetzt wurde, wäre auch dies möglich.

5. Installation des Root Filesystems

Zuletzt wird das Root Filesystem installiert. Dazu wird die dritte Partition mit ext2 formatiert, gemountet und das Root Filesystem Archiv nach `/mnt` entpackt:

```
$ mkfs.ext2 /dev/sdc3
$ mount /dev/sdc3 /mnt
$ tar xfvj tqma28-rootfs.tar.bz2 -C /mnt/
$ umount /mnt
```

Damit ist die SD-Karte für den Betrieb im STK-MBa28 Board bereit.

6.4 Booten des Systems

Vor dem Start des Systems müssen die Bootpins entsprechend Tabelle 6 geändert werden und die SD-Karte in das Baseboard eingesteckt werden. Nach dem Einschalten der Spannungsversorgung startet der U-Boot Loader. Der Bootvorgang wird dann mit folgenden Kommandos gestartet:

```
=>setenv bootargs 'fec_mac=02:00:01:00:00:01 ip=dhcp
root=/dev/mmcblk1p3 rw console=ttySP3,115200
lcd_panel=fg0700 ssp1 panic=1'
=> fatload mmc 1:2 0x42000000 /uImage
=> bootm
```

Um automatisch diese Kommandos auszuführen, können die drei Kommandos zusammen als Variable in der U-Boot Umgebung gespeichert werden.

```
=> setenv boot_mymmc 'setenv bootargs
'fec_mac=02:00:01:00:00:01 ip=dhcp root=/dev/mmcblk1p3 rw
console=ttySP3,115200 lcd_panel=fg0700 ssp1 panic=1';
fatload mmc 1:2 0x42000000 /uImage; bootm'
```

U-Boot ruft nach seinem Start `bootcmd` auf. Um diese auf `boot_mymmc` zu setzten muss die Umgebungsvariable zuerst mit `setenv bootcmd` gelöscht werden und kann dann mittels `savenv bootcmd 'run boot_mymmc'` neu gesetzt werden. Anschließend werden die neuen Werte mit `savenv` dauerhaft gespeichert.

7 Erstellen eines eigenen Systems

Im diesem Kapitel wird nun versucht ein komplettes System für das STK-MBa28, losgelöst von den TQ Quellen, auf Basis der jeweiligen Mainline-versionen und unter Verwendung von ELDK in der Version 5.2 zu konstruieren.

7.1 Vorbereitung des Hostsystems

Auf explizite Paketabhängigkeiten wird in diesem Kapitel nicht mehr eingegangen. Sie erschließen sich meist aus den Installations- und Konfigurationsprozessen selbst. Zudem wird im Verlauf immer wieder auf entsprechende Dokumentationen hingewiesen. Eingeführt und betrachtet werden nur solche Bestandteile, die maßgeblich am Erstellungsprozess beteiligt sind.

7.1.1 Bedeutung von Git

Git spielt eine wichtige Rolle in der OpenSource-Software-Entwicklung. Gerade weil die beiden Hauptbestandteile, U-Boot und der Linux Kernel, Git zur eigenen Versionskontrolle und kollaborativen Zusammenarbeit der Entwickler nutzt, soll es nicht versäumt werden hier ebenfalls eine Einführung zu geben. Natürlich ist dies kein Ersatz für ein Studium der Dokumentation. Es sollen aber einige grundlegende Konzepte beleuchtet werden. Eine kurze Einführung in die für nachfolgende Schritte verwendeten praktischen Kommandos soll zudem vermittelt werden.

Versionskontrolle ist ein wichtiger Aspekt in der Softwareentwicklung. Mit Git wird es möglich, alle Änderungen an Dateien über die Zeit hinweg zu verfolgen.[50]

Für die nachfolgenden Schritte wird in erster Linie der Quellcode der verschiedenen Projekte benötigt. Der Quellcode von U-Boot bzw. vom Linux Kernel befindet sich in öffentlichen Git-Repositories. Um nun mit dem Quellcode arbeiten zu können, wird eine lokale Kopie benötigt. Diese Kopie wird in Git als *clone* bezeichnet. Dieser lokale Klon steht dann wiederum unter der Versionskontrolle von Git, jedoch dann lokal.

Änderungen werden, im Gegensatz zu anderen Versionskontrollsystemen, nicht als Liste von Änderungen einer Datei gespeichert, sondern als kompletter Schnappschuss des Projektstandes[51]. Bei jedem *commit* wird solch ein neuer Snapshot erstellt. Commit bedeutet dabei Änderungen innerhalb des von Git verwalteten Projekts aufzuzeichnen. Zu jeder Zeit kann auf ältere Commits zurückgegriffen oder die Unterschiede zwischen ihnen betrachtet werden.

Daneben kennt Git die Konzepte *branch* und *merge*. Ein Branch ist ein Abzweig von der Hauptversion (*master*). An ihm kann ohne Einfluss auf die Hauptversion entwickelt werden. Beispielsweise wäre ein testing-Branch denkbar, an dem neue Features getestet werden. Sind die Anpassungen abgeschlossen und sollen sie in die Hauptversion des Projektes übertragen werden, wird ein Merge durchgeführt. Dabei werden die Änderungen aus einem Zweig mit dem Hauptzweig verschmolzen.

Eine weitere wichtige Funktion ist das Erstellen von Patches. Ein Patch enthält nur Änderungen an Quellcodezeiten. Patches dienen der Dokumentation von Anpassungen, aber vor allem der Kommunikation mit einem Projekt. So sind die meisten öffentlichen Repositories nur öffentlich lesbar, jedoch nicht zu beschreiben. Damit könnten keine Anpassungen zurück an das eigentliche Projekt gegeben werden. Hierbei kommt das Patch ins Spiel. Nach seiner Erstellung kann es an den Verwalter des Projektes

50 Vgl. [PROGIT] - 1.1 - About Version Control

51 Vgl. [PROGIT] - 1.3 - Getting Started - Git Basics

gesendet werden. Er entscheidet dann über die Integration des Patches.

In diesem Fall kommt Git hauptsächlich zur Beschaffung des Quellcodes und zur Dokumentation der Änderungen zum Einsatz. Die während dieser Arbeit verwendeten Git-Kommandos sind:

`git clone`	Lokale Kopie des Quellcodes erzeugen
`git branch`	Entwicklungszweig erstellen
`git merge`	Entwicklungszweig mit Hauptversion zusammenführen
`git commit`	Änderungen aufzeichnen
`git format-patch`	Patch erstellen
`git log`	Verlauf der Änderungen anzeigen

Vor den Änderungen, z.B. an U-Boot, wurde ein Branch erstellt, an dem die Anpassungen durchgeführt wurden. Nach Abschluss der Arbeiten und Tests wurde der Branch in den Master-Zweig gemerged und ein Patch davon erstellt.

7.1.2 Installation von ELDK 5.2

Für den Bau einer eigenen Umgebung wird die aktuelle Version 5.2.1 von ELDK verwendet. Der Installationsprozess wurde der Dokumentation zu ELDK entnommen und entsprechend den Gegebenheiten angepasst.

```
$ mkdir eldk-download
$ cd eldk-download
$ mkdir -p targets/armv5te
$ wget ftp://ftp.denx.de/pub/eldk/5.2.1/install.sh
$ cd targets/armv5te
$ wget ftp://ftp.denx.de/pub/eldk/5.2.1/targets/\
```

```
    armv5te/target.conf
$ wget ftp://ftp.denx.de/pub/eldk/5.2.1/targets/armv5te/\
    eldk-eglibc-i686-arm-toolchain-gmae-5.2.1.tar.bz2
$ wget ftp://ftp.denx.de/pub/eldk/5.2.1/targets/armv5te/\
    core-image-minimal-generic-armv5te.tar.gz
$ wget ftp://ftp.denx.de/pub/eldk/5.2.1/targets/armv5te/\
 core-image-sato-sdk-generic-armv5te.tar.gz

$ cd ..
$./install.sh -r "minimal" armv5te
```

Die Bekanntmachung der Toolchain erfolgt unter Zuhilfenahme des enthaltenen Skriptes `environment-setup-armv5te-linux-gnueabi`. Zusätzlich muss noch die Umgebungsvariable `CROSS_COMPILE` entsprechend gesetzt werden.

```
$ source /opt/eldk-5.2.1/armv5te/\
    environment-setup-armv5te-linux-gnueabi
$ export CROSS_COMPILE=arm-linux-gnueabi-
```

Damit ist die Vorbereitung von ELDK abgeschlossen und kann in den folgenden Schritten verwendet werden.

7.2 Erstellungsprozess U-Boot

7.2.1 Beschaffung des Quellcodes

Wie bereits erwähnt, befindet sich der Quellcode zu U-Boot in einem Git-Repository. Für den weiteren Anpassungs- und Erstellungsprozess wird dieser nun benötigt. Die aktuelle Version ist auf den Seiten von DENX Software Engineering verfügbar und kann durch einen einfachen `git clone` lokal verfügbar gemacht werden.

```
$ git clone git://git.denx.de/u-boot.git u-boot
```

Anschließend ist er im Unterverzeichnis `u-boot/` zu finden. Zum Zeitpunkt der Arbeit handelte es sich um die Version 2012.07.

7.2.2 Erstellen eines neuen Boards auf Basis des mx28evk

Aus der Analyse des TQ SVN-Repository ist bereits bekannt, dass zwischen dem TQMA28-Modul bzw. dem STK-MBa28 Baseboard starke Ähnlichkeit zu dem Freescale MX28EVK besteht. Es liegt die Vermutung nahe, dass das MX28EVK eine grundlegende Rolle bei der TQ Entwicklung spielte. Daher wird die Konfiguration dieses, bereits im U-Boot vorhandenen Boards, als Grundlage herangezogen. Eine direkte Konfiguration für das TQMa28 Modul existiert in der Mainlineversion von U-Boot nicht.

Im ersten Schritt wird U-Boot für die Verwendung mit einem Board vorbereitet. Dazu dienen die Kommandos `make mrproper` und `make <BOARDNAME>_config`.

Um nun eine zusätzliche Konfiguration für das TQ-Board zu schaffen, werden die boardspezifischen Teile im Verzeichnis `board/freescale/mx28evk` unter dem Namen *tqma28* nach `board/tqc/` kopiert und die Datei `mx28evk.c` umbenannt in `tqma28.c`.
Ebenso muss die Boardkonfiguration von `include/configs/mx28evk.h` nach `include/configs/tqma28.h` kopiert werden.

```
$ cp -a board/freescale/mx28evk board/tqc/tqma28
$ mv board/tqc/tqma28/mx28evk.c board/tqc/tqma28/tqma28.c
$ cp include/configs/mx28evk.h include/configs/tqma28.h
```

Zusätzlich muss das Makefile entsprechend angepasst werden. Dazu wird die folgende Zeile 29 ausgetauscht:

```
cat /board/tqc/tqma28/Makefile
[…]
#COBJS    := mx28evk.o
COBJS    := tqma28.o
[…]
```

Dieses neue Board muss in der `boards.cfg`, im `u-boot/` Verzeichnis, noch bekannt gemacht werden. Folgende Zeile muss ergänzt werden:

```
tqma28      arm         arm926ejs   -      tqc      mx28
```

Eine Übersicht der angepassten Dateien ist dem *Anhang B – U-Boot Support für TQMa28* zu entnehmen.

U-Boot kann nun mit `make mrproper; make tqma28_config` für das Board vorbereitet werden und sollte ohne Ausgabe einer Fehlermeldung abschließen.
Ohne weitere Anpassungen am Quellcode ist jetzt ein Build des U-Boot Loaders bereits möglich. Dies wird mit `make u-boot.sb` überprüft. Der Erstellungsprozess sollte ohne Abbruch beendet werden und das bereits bekannte SB-Image (`u-boot.sb`) liefern.

7.2.3 Allgemeine Konfiguration

Konfiguriert wird das neu erstellte tqma28 Board über die Board-Konfiguration `tqma28.h` im `include/configs` Verzeichnis. Hier werden die Anpassungen an spezifische Eigenschaften des Boards vorgenommen. Beispielsweise müssen einige Einstellungen zur korrekten Verwendung des

SRAMS durchgeführt werden. Das MX28EVK verwendet 1 GB SRAM, wohin gegen das TQ-Modul nur über 128 MB verfügt. Diese Anpassung, im Bereich der Zeilen 85 - 88 sieht wie folgt aus:

```
[…]
#define CONFIG_NR_DRAM_BANKS 1          /*1 bank of DRAM*/
#define PHYS_SDRAM_1 0x40000000         /*Base address*/
-#define PHYS_SDRAM_1_SIZE 0x40000000  /*Max 1 GB RAM*/
+#define PHYS_SDRAM_1_SIZE 0x08000000  /*Max 128M RAM*/
#define CONFIG_STACKSIZE (128 * 1024)  /*128 KB stack*/
[…]
```

Auch wird die Konsolenbezeichnung in Zeile 106 geändert:

```
-#define CONFIG_SYS_PROMPT       "MX28EVK U-Boot > "
+#define CONFIG_SYS_PROMPT       "TQMA28 U-Boot > "
```

Bei der Durchsicht der Konfiguration wurde klar, dass vom MX28EVK der DUART Port für die Ausgabe des U-Boot Loaders verwendet wird. Ersichtlich aus Zeile 124:

```
[…]
#define CONFIG_PL01x_PORTS  { (void *)MXS_UARTDBG_BASE }
[…]
```

Von TQ wird auf dem STK-MBa28 der APPUART3 Port für diese Aufgabe vorgesehen. Eine simple Anpassung der `CONFIG_PL01x_PORTS` von `MXS_UARTDBG_BASE` zu `MXS_UARTAPP3_BASE` kann in diesem Fall nicht durchgeführt werden. Der einzige PrimeCell kompatible Port ist DUART[52]. Zur Verwendung der APPUARTs muss ein eigener Treiber

52 vgl. [IMX28RM] - S. 1534 ff.

verwendet werden. Diesem Umstand wird in Abschnitt 7.2.4 separat Rechnung getragen. Auch fällt auf, dass nur eine MMC-Gerät für die Verwendung vorgesehen ist. Dies wird im Abschnitt 7.2.5 gesondert behandelt.

Ein erster Test konnte trotzdem durchgeführt werden. DUART ist über den Extension-Port herausgeführt und kann durch verbinden eines externen Pegelwandlers, z.B. MAX232, verwendet werden. Die benötigten Pins sind am Anschluss X14.29 DUART_TX und X14.32 DUART_RX zu finden[53]. Die Spannungsversorgung wird dabei über X15.39 5 V und X15.1 GND hergestellt.

U-Boot kann somit in herkömmlicher Weise in Verbindung mit Picocom bedient werden.

7.2.4 Anpassung AUART3 Treiber

Die Verwendung des DUART Ports für die Bedienung des U-Boot Loaders ist zwar prinzipiell möglich, jedoch nicht zufriedenstellend. Auf dem STK-MBa28 ist APPUART3 über einen 9-poligen Sub-D-Stecker, der genau diesem Zweck dienen kann, herausgeführt. Schon allein aus praktischen Gründen und um auf den externen Pegelwandler zu verzichten, ist eine Nutzung des APPUART3 Ports anzustreben. Nachteilig ist, dass in der Mainlineversion des U-Boots kein passender Treiber zur Nutzung vorhanden ist und dieser selbst implementiert werden müsste.
Im Linux-Kernel ist ein entsprechender Treiber für APPUART unter `drivers/tty/serial/mxs-auart.c` verfügbar. Dieser könnte die Grundlage für eine Implementierung auch im U-Boot Loader sein.

Ein deutlich schnellerer Weg um eine Unterstützung zu erlangen, wäre den bereits im TQ Repository vorhandenen Treiber in die neue U-Boot Version zu

53 [STK11] - S. 52 Table 58: Plug connector Starterkit interface (X14)

portieren. Dabei ist auf die Lizenz zu achten. Nachdem die Implementierung von TQ allerdings unter der GPL steht, ist dies möglich und wird im Folgenden dargestellt.

Im ersten Schritt muss dazu untersucht werden, wie U-Boot verschiedene Treiber verwaltet und wie diese eingebunden werden. Einen Anhaltspunkt dazu liefert bereits die Ordnerstruktur innerhalb der Arbeitskopie. Im `u-boot/` Verzeichnis findet sich ein Ordner mit der Bezeichnung `drivers/`. Bei dessen Durchsicht ist ein weiterer Unterordner `serial/` zu finden. In ihm befinden sich die Implementierungen von verschiedenen Treibern und ein Makefile. Bei der Analyse des Makefiles ist festzustellen, dass U-Boot die Verwendung der Treiber über Defines in der Board-Konfiguration steuert. Aus dieser Information wird dann der entsprechende Treiber übersetzt und verwendet.

Der portierte Treiber soll über eine eigene Präprozessordirektive ausgewählt werden können. Dazu wird im `Makefile` unter `drivers/serial/` eine neue Zeile eingefügt. Die Steuerung soll über `CONFIG_MXAUART_SERIAL` erfolgen und die Implementierung den Namen `serial_mxauart.c` tragen. Die einzufügende Zeile sieht dann wie folgt aus:

```
COBJS-$(CONFIG_MXAUART_SERIAL) += serial_mxauart.o
```

Im nächsten Schritt werden die aus dem TQ Repository benötigten Dateien `serial_auart.c` und `regs-uartapp.h` nach `drivers/serial` kopiert.

```
$ cp -a tq/boot/u-boot/cpu/arm926ejs/mx28/serial_auart.c\
     u-boot/drivers/serial/serial_mxauart.c
$ cp -a tq/boot/u-boot/include/asm-arm/arch-mx28/\
     regs-uartapp.h\
     u-boot/drivers/serial/serial_mxauart.h
```

Zudem müssen in der `serial_auart.c` Anpassungen vorgenommen werden.

In Zeile 35:

```
-#include <asm/arch/regs-uartapp.h>
+#include "serial_mxauart.h"
```

In Zeile 39:

```
-#define REGS_UARTAPP_BASE REGS_UARTAPP3_BASE
+#define REGS_UARTAPP_BASE MXS_UARTAPP3_BASE
```

In Zeile 40:

```
+#define REG_RD(base, reg) \
+(*(volatile unsigned int *)((base) + (reg)))
+ #define REG_WR(base, reg, value) \
+((*(volatile unsigned int *)((base) + (reg))) = (value))
```

In Zeile 86:

```
-auart_board_init();
+//auart_board_init();
```

Zuletzt muss U-Boot in der Board-Konfiguration `tqma28.h` unter `includes/configs` angewiesen werden, den neuen Treiber anstelle des PrimeCell Treibers zu verwenden.

In Zeile 122:

```
-#define CONFIG_PL011_SERIAL
-#define CONFIG_PL011_CLOCK           24000000
-#define CONFIG_PL01x_PORTS { (void *)MXS_UARTDBG_BASE }
+#define CONFIG_MXAUART_SERIAL
+#define CONFIG_UARTAPP_CLK           24000000
```

Anschließend muss das Pin-Multiplexing zur Verwendung der Pins für APPUART gesetzt werden. Dazu wird `board/tqc/tqma28/iomux.c` um folgendes ergänzt:

In Zeile 39:

```
+/*AUART*/
+MX28_PAD_AUART3_RX__AUART3_RX | (MXS_PAD_4MA |
+MXS_PAD_3V3 | MXS_PAD_NOPULL),
+MX28_PAD_AUART3_TX__AUART3_TX  | (MXS_PAD_4MA |
+MXS_PAD_3V3 | MXS_PAD_NOPULL),
+MX28_PAD_AUART3_RTS__AUART3_RTS | (MXS_PAD_4MA |
+MXS_PAD_3V3 | MXS_PAD_NOPULL),
+MX28_PAD_AUART3_CTS__AUART3_CTS | (MXS_PAD_4MA |
+MXS_PAD_3V3 | MXS_PAD_NOPULL),
```

U-Boot kann nun erneut übersetzt werden und die Konsolenausgabe ist an X32B des Boards verfügbar. Das komplette Patch zur Anpassung ist im *Anhang C – U-Boot AUART3 Patch* zu finden.

7.2.5 Anpassung für MMC0 und MMC1 Benutzung

Der Zugriff auf MMC0, den internen eMMC Speicher, war sofort möglich. Jedoch konnte nicht auf MMC1, die externe SD-Karte, zugegriffen werden. Grund hierfür ist, dass für das MX28EVK nur der Betrieb von MMC0 vorgesehen ist. Bestätigen lässt sich das anhand der Ausgabe des U-Boot Loaders beim Start. Bereits hier wird nur `MMC:  MXS MMC: 0` genannt. Das Kommando `mmc list` bestätigt diesen Umstand zusätzlich.

Für eine Nutzung von MMC1 ist eine Anpassung in der Datei `tqma28.c` und `iomux.c` notwendig. Initialisiert wird in der Funktion `board_mmc_init` nur die MMC-Unterstützung für die ID 0, jedoch nicht für die ID 1.

Die Anpassung um MMC1 sieht wie folgt aus:

```
int board_mmc_init(bd_t *bis)
{
  /* Configure MMC0 WP as input */
  gpio_direction_input(MX28_PAD_SSP1_SCK__GPIO_2_12);
  /* Configure MMC0 Power Enable */
  gpio_direction_output(MX28_PAD_PWM3__GPIO_3_28, 0);

 +/* Configure WP as input MMC1*/
 +gpio_direction_input(MX28_PAD_SSP1_SCK__GPIO_2_12);
 +/* Configure MMC0 Power Enable MMC1*/
 +gpio_direction_output(MX28_PAD_PWM3__GPIO_3_28, 0);

 -return mxsmmc_initialize(bis, 0, mx28evk_mmc_wp);
 +return  mxsmmc_initialize(bis,  0,  mx28evk_mmc_wp)  |
mxsmmc_initialize(bis, 1, mx28evk_mmc_wp);
}
```

Es wird die Datenrichtung der GPIOs für die Verwendung von MMC1 entsprechend dem Datenblatt gesetzt. Dabei handelt es sich bei `MX28_PAD_SSP1_SCK__GPIO_2_12` um die Write Protection, die als Eingang gelesen werden muss und bei `MX28_PAD_PWM3__GPIO_3_28` um die Spannungsversorgung der Karte als Ausgang. Anschließend wird `mxsmmc_initialize` zusätzlich mit der ID 1 aufgerufen.

Damit MMC1 erfolgreich verwendet werden kann, muss das entsprechende Multiplexing der Pins in `iomux.c` erweitert werden. Dazu werden folgende Zeilen hinzugefügt:

```
/* MMC1 */
MX28_PAD_GPMI_D00__SSP1_D0 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D01__SSP1_D1 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D02__SSP1_D2 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D03__SSP1_D3 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D04__SSP1_D4 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D05__SSP1_D5 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D06__SSP1_D6 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_D07__SSP1_D7 | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_RDY1__SSP1_CMD | MUX_CONFIG_GPMI,
MX28_PAD_GPMI_RDY0__SSP1_CARD_DETECT |
      (MXS_PAD_8MA | MXS_PAD_3V3 | MXS_PAD_NOPULL),
MX28_PAD_GPMI_WRN__SSP1_SCK |
      (MXS_PAD_12MA | MXS_PAD_3V3 | MXS_PAD_NOPULL),
 /* write protect */
 MX28_PAD_GPMI_RESETN__GPIO_0_28,
  /* MMC1 slot power enable */
  MX28_PAD_PWM4__GPIO_3_29 |
      (MXS_PAD_12MA | MXS_PAD_3V3 | MXS_PAD_PULLUP),
```

Abgeleitet wurde dies von MMC0 und der Beschreibung im Referenz Manual des i.MX28 *[IMX28RM] - 9.2.2 Pin Interface Multiplexing*. Die Bezeichner der Pins stammen aus `arch/arm/include/asm/arch/iomux-mx28.h`. Eine komplette Liste der Änderungen ist wiederum dem *Anhang C – U-Boot AUART3 Patch* zu entnehmen.

Beim Start des U-Boot werden dadurch nun `MMC: MXS MMC: 0, MXS MMC: 1` angezeigt. Auch `mmc list` führt nun zwei MMC-Geräte an. Beide Geräte können nun in herkömmlicher Art und Weise verwendet werden.

Der Boot Loader kann mit der bereits beschriebenen Methode auf die SD-

Karte installiert werden. U-Boot bietet in neueren Versionen eigene Tools an. Folgt man den Ausführungen in `doc/README.mx28_common`, kann die Installation auch mit dem Werkzeug `tools/mxsboot` durchgeführt werden. Aus dem SB-Image wird ein SD-Image erstellt, welches direkt in die erste Partition geschrieben wird. Bei dieser Methode ist zu beachten, dass der Startsektor der Firmware-Partition 2048 sein muss. Anderenfalls kann er über die Option `-p` angepasst werden.

```
$ ./tools/mxsboot sd u-boot.sb u-boot.sd
$ dd if=u-boot.sd of=/dev/sdc1
```

7.3 Erstellungsprozess Linux Kernel

Im Folgenden wird nun ein eigener Kernel konfiguriert und übersetzt. Zur Zeit der Durchführung war die Kernel-Version 3.5.0 aktuell.

7.3.1 Beschaffung des Quellcodes

Der Quellcode des Kernels wird, wie auch schon bei U-Boot, mit Git beschafft. Als Bezugsquelle dient www.kernel.org. Ausgeführt wird folgendes Git-Kommando:

```
git clone git://git.kernel.org/pub/scm/linux/\
     kernel/git/stable/linux-stable.git linux
```

Er ist anschließend im Unterverzeichnis `linux/` verfügbar. Soll kein 3.x Kernel verwendet werden, bietet auch Freescale eine alternative Bezugsquelle[54] an.

7.3.2 Kernelkonfiguration und -übersetzung

Ebenso wie bei der Übersetzung von U-Boot muss die Toolchain in der Shell-

54 git://git.freescale.com/imx/linux-2.6-imx.git

Sitzung bekannt sein. Zusätzlich muss die Umgebungsvariable `ARCH` gesetzt werden. Sie gibt an, für welche Zielarchitektur der Linux Kernel konfiguriert und übersetzt werden soll. Alternativ kann die Architektur auch direkt an `make` übergeben werden.

Ein exemplarischer Aufruf würde in etwa so aussehen :

```
$ make ARCH=arm <...>
```

Nach dem dies jedoch bei jedem Aufruf durchgeführt werden müsste, bietet es sich an, die Variable in der Umgebung zu setzten und im Kommando darauf zu verzichten.

```
$ export ARCH=arm
```

Im nächsten Schritt muss die Kernelkonfiguration für das Target angepasst werden.

```
$ make mxs_defconfig
```

Zusätzlich sind einige Anpassungen an die abweichenden Hardware-gegebenheiten notwendig. Dazu wird die Kernelkonfiguration mit `make menuconfig` aufgerufen.

Anfangs sollte überprüft werden, ob die Unterstützung für das MX28EVK aktiviert ist. Unter der Rubrik `System Type` sollte `Support MX28EVK Platform` ausgewählt sein.

Der Kernel muss in der Lage sein, die Application UARTs zu benutzen. Es müssen die Optionen `MXS AUART support` und `MXS AUART console support` aktiviert werden. Zu finden sind diese Optionen unter `Device`

`Drivers ---> Character devices ---> Serial drivers`. Ohne diese beiden startet das System zwar, jedoch endet die Konsolenausgabe bei `Starting kernel`.

In der Defaultkonfiguration ist keine Unterstützung für EXT2 enthalten. Unter `File systems` kann die Option `Second extended fs support` aktiviert werden. Ist absehbar, dass lediglich eine EXT3-Partition als Root-Partition eingesetzt werden soll, kann auf diesen Schritt verzichtet werden.

Die Real Time Clock wurde bei ersten Tests nicht korrekt erkannt. Hier fehlte eine entsprechende Unterstützung der `Freescale STMP3xxx/i.MX23/ i.MX28 RTC`. Diese Option kann unter `Device Drivers ---> Real Time Clock` aktiviert werden.

GPIOs werden vom STK-MBa28 über zwei 8-Bit I²C IO Expander vom Typ PCA9554D bereitgestellt. In der Konfiguration des MX28EVK sind diese so nicht vorgesehen und müssen manuell integriert werden. Für diese, den Temperatursensor LM73, das Serielle EEPROM 24c64 und den SGTL5000 Audio-Codec sieht TQ-Systems nicht die Verwendung des I²C0-Bus, sondern die des I²C1-Bus vor. Hauptsächlich betreffen die Anpassungen die Datei `arch/arm/mach-mxs/mach-mx28evk.c` und wurden aus den Anpassungen von TQ-Systems für den Kernel 2.6 portiert. Der komplette Kernel-Patch sowie ein kurzer Funktionstest der GPIOs ist im *Anhang E – GPIO Kernel Patch* zu finden.
Zusätzlich muss bei der Kernelkonfiguration die Option `PCA953x, PCA955x, PCA957x, TCA64xx, and MAX7310 I/O ports` unter `Device Drivers ---> GPIO Support --->` aktiviert werden. Dabei darf jedoch nicht die Zusatzoption `Interrupt controller support for PCA953x` benutzt werden. Ist diese aktiv, kommt es bei der Initialisierung des zweiten PCA9554D zu einem IRQ-Konflikt.

Nach den Anpassungen wird die Übersetzung mit `make uImage` gestartet und liefert ein Kernel-Image unter `arch/arm/boot/uImage`.

Zum Test wurde das minimale TQ Root Filesystem auf der dritten Partition belassen. Für den Start müssen die übergebenen Kernel-Parameter angepasst werden. Aus der Sichtung des Kernel-Treibers für die Application UARTs ist bereits bekannt, dass der Name des Devices sich von Kernel-Version 2.6.35 zu 3.5.0 geändert hat. Im Kernel 3.5.0. werden die Application UARTS nicht mit /dev/ttySPx, sondern als /dev/ttyAPPx bezeichnet. Diesem Umstand wird im U-Boot mit dem Parameter `console` Rechnung getragen. Die Kommandofolge sieht dann wie folgt aus:

```
=> setenv bootargs 'root=/dev/mmcblk1p3 rw console=ttyAPP3,
115200'
=> fatload mmc 1:2 $loadaddr /uImage35
=> bootm
```

Trotz der Angabe der Root-Partition ist der Kernel nicht in der Lage diese korrekt in das System einzubinden. Der Kernel bricht mit einer Kernel Panic ab.

```
[1.230000]     VFS: Cannot open root device "mmcblk1p3"
               or unknown-block(0,0): error -6
[1.230000]     Please append a correct "root=" boot option
[1.240000] b300         1955840 mmcblk0  driver: mmcblk
[1.250000]   b301          524288 mmcblk0p1 00000000-0[...]
[1.260000]   b302            8192 mmcblk0p2 00000000-[...]
[1.260000]   b303         1415167 mmcblk0p3 00000000-0[...]
[1.270000] b310             512 mmcblk0boot1  (driver?)
[1.280000] b308             512 mmcblk0boot0  (driver?)
[1.280000]     Kernel panic - not syncing: VFS: Unable to
               mount root fs on unknown-block(0,0)
```

Versuche haben gezeigt, dass der Kernel früher versucht das Root Filesystem zu mounten, als der Treiber für das MMC1-Gerät bereit ist. Werden die Kernel-Parameter um `rootdelay=1` ergänzt, tritt dieser Fehler nicht mehr auf. Die Wartezeit von einer Sekunde reicht aus, dass der Treiber bereit ist und die notwendigen Devices dem System bereitgestellt werden.

```
=> setenv bootargs 'root=/dev/mmcblk1p3 rw rootdelay=1
console=ttyAPP3,115200'
```

7.4 Cross-Installation von Debian

In diesem Abschnitt wird beschrieben, wie ein Debian-System erstellt wird, wenn die Architektur des Hostsystems und die des Targets unterschiedlich sind. Es wird multistrap verwendet. Damit lassen sich Debian-Systeme mit unterschiedlichen Ausprägungen zusammenstellen[55]. Es ist für die Architekturen amd64, armel, i386, mips, mipsel und powerpc geeignet. Für den Testbetrieb wird QEMU verwendet und das fertige System anschließend auf das Target übertragen.

7.4.1 Verwendung von Multistrap

Multistrap ist ein Werkzeug um Debian Root Filesysteme zu erstellen. Dabei setzt es, anders als debootstrap, APT und DPKG voraus. Eine Debian oder auf Debian basierende Distribution ist also Voraussetzung. Es stellt eine Weiterentwicklung gegenüber debootstrap dar, da damit auch Filesysteme für fremde Architekturen erstellt werden können. [56] Es kann aus den Debian-Paketquellen installiert werden.

```
$ apt-get install multistrap
```

55 Quelle: [DEBARM] - Cross-installing Debian using debootstrap/multistrap

56 http://packages.debian.org/de/squeeze/multistrap

Vor dem Start muss eine entsprechende Konfigurationsdatei `multistrap.conf` angelegt werden. In ihr werden die Basiseinstellungen für das zu erstellende System spezifiziert. Für eine genaue Beschreibung aller Optionen sollte die Manual Page zu multistrap konsultiert werden. Hierbei handelt es sich lediglich um ein minimales Beispiel[57]. Ergänzt werden sollte in der packages-Sektion in jedem Fall udev, apt, dialog, net-tools und iputils-ping. Ohne diese fünf Pakete ist das System kaum praktisch nutzbar. Udev kümmert sich um die Geräteverwaltung, APT und dialog wird zur Paketverwaltung installiert und die net-tools / iputils-ping werden für die Netzwerkkonfiguration benötigt. Optional, aber sinnvoll, sind die Pakete dhcp3-client um die Netzwerkkonfiguration vom DHCP-Server zu beziehen, SSH für Remote-Verbindungen und vim als Editor.

```
$ cat multistrap.conf
General]
noauth=true
unpack=true
debootstrap=Squeeze
aptsources=Squeeze
arch=armel

[Squeeze]
packages=udev apt dialog net-tools iputils-ping dhcp3-
client ssh vim
source=http://ftp.au.debian.org/debian/
keyring=debian-archive-keyring
components=main non-free
suite=squeeze
```

57 Quelle: [DEBARM] - Basic usage

Erstellt wird das Root-Filesystem mit dem Kommando:

```
$ multistrap -a armel -d squeeze -f multistrap.conf
```

Multistrap lädt die benötigten binären Pakete von den angegebenen Quellen herunter und stellt das System im Unterverzeichnis `squeeze/` zusammen. Bei der minimalen Konfiguration, wie oben, erhält man ein 240 MB großes Root Filesystem.
Abschließend müssen die Post-Install Scripts aufgerufen und einige vom Target abhängige Anpassungen durchgeführt werden. Dazu müsste in das Root-Verzeichnis mit `chroot` gewechselt werden. Dies ist auf Grund der unterschiedlichen Architekturen von Host und Target nicht möglich. Abhilfe schafft hier QEMU.

7.4.2 Emulation mittels QEMU

QEMU ist eine freie virtuelle Maschine, die die komplette Hardware eines Computers emuliert.[58] Die Installation erfolgt ebenfalls aus den Debian-Paketquellen.

```
$ apt-get install binfmt-support qemu qemu-user-static
$ cp /usr/bin/qemu-arm-static squeeze/usr/bin
```

Nun kann mit `chroot` in das Verzeichnis `squeeze/` gewechselt werden und die ausstehende Konfiguration durchgeführt werden.

```
$ chroot squeeze
$ dpkg --configure -a
```

Vor dem ersten Start muss nun noch die Konsolenausgabe von getty auf das richtige Gerät geleitet werden. Dazu muss `/etc/inittab` des neuen Root

58 vgl. http://de.wikipedia.org/wiki/QEMU

Filesystems entsprechend angepasst werden. Zusätzlich muss mindestens das Device `/dev/console` erstellt werden. Es sollte darüber hinaus ein Passwort für den root-Account mit `passwd` und ein Hostname gesetzt werden. Um Fehlermeldungen beim Booten bzw. bei der DHCP Abfrage zu verhindern, werden die Dateien `/etc/fstab` und `/etc/resolv.conf` erstellt.

```
$ chroot squeeze
$ mknod -m 622 /dev/console c 5 1
$ passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
$ echo "T0:2345:respawn:/sbin/getty -L console 115200
vt100" >> /etc/inittab
$ echo "tqma28" > /etc/hostname
$ touch /etc/fstab
$ touch /etc/resolv.conf
```

Das neue Debian Squeeze kann nun in die dritte Partition der SD-Karte kopiert und gestartet werden.

7.5 Debugging Technik

Eine zentrale Rolle bei der Anpassung des U-Boot Loader spielte das Debugging. Ohne Möglichkeit den veränderten Quellcode auf dem Target selbst zu untersuchen und dessen Verhalten zu analysieren ist eine erfolgreiche Anpassung nahezu unmöglich.
Moderne Mikrocontroller sind zu diesem Zweck meist JTAG-fähig. Damit lassen sich Programme, die sich in ihrer eigentlichen Arbeitsumgebung befinden, testen. Über die JTAG-Schnittstelle lässt sich das Verhalten des

Mikrocontrollers von außen beeinflussen. Im normalen Betrieb beeinträchtigen die JTAG-Komponenten den Betrieb nicht.[59]

7.5.1 JTAG Interface – Amontec

Hostsysteme, auf denen ein Debugger betrieben werden soll, besitzen meist keine passende Schnittstelle um direkt mit dem Target über JTAG zu kommunizieren. Zur Nutzung der JTAG-Funktionalität wird deshalb ein entsprechendes Interface benötigt. Ein Amontec JTAGkey-tiny wurde hierzu verwendet. Es zeichnet sich vor allem durch niedrige Anschaffungskosten, freie Treiber für Linux und die Kompatibilität zu OpenOCD aus. Der Anschluss an das Hostsystem erfolgt per USB, sowie am Target über einen ARM Multi-ICE 20-pin Stecker.

7.5.2 Verbindung JTAG-Interface und STK-MBa28

Auf dem STK-MBa28 sind die JTAG Pins an der Stiftleiste X16 verfügbar. Leider ist der Anschluss nicht kompatibel mit dem ARM Multi-ICE 20 Pin. Um eine Verbindung herzustellen musste ein Adapterkabel (vgl. Abbildung 13) gefertigt werden.

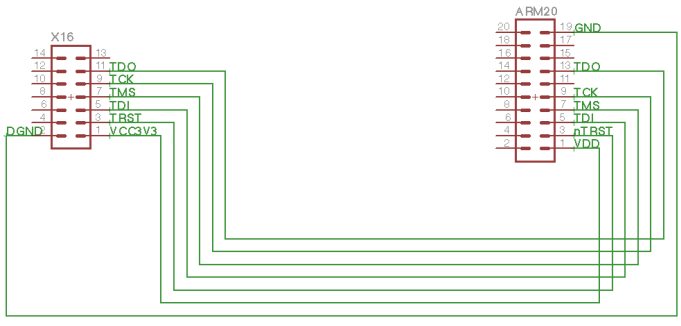

Abbildung 13: JTAG Verbindung STK-MBa28 - Amontec JTAGkey-Tiny

Die Belegung des X16 Anschlusses kann dem *User's Manual [TQMaUM11] S. 45 - 4.2.3.2 JTAG (X16)* entnommen werden.

59 http://de.wikipedia.org/wiki/Joint_Test_Action_Group

7.5.3 OpenOCD

Die Bereitstellung von Debugging-Funktionalität, In-System Programmierung und Boundary-Scan Test auf dem Hostsystem übernimmt OpenOCD.[60] Es arbeitet eng mit dem JTAG-Interface zusammen um die Kontrolle über das Target zu übernehmen. Eine Liste der unterstützten Interfaces[61] und Zielarchitekturen[62] ist auf den Internetseiten des Projektes verfügbar. Dem Hostsystem werden wiederum Schnittstellen zur Bedienung per Telnet oder über einen Debugger bereitgestellt. Die Installation von OpenOCD 0.5 kann der Dokumentation entnommen werden und wird hier nicht weiter beschrieben.

Gestartet wird OpenOCD mit mindestens zwei Aufrufparametern. Der erste spezifiziert das zu verwendende JTAG-Interface, der zweite die Zielplattform. Entsprechende Skripte sind für das Interface und das Board unter `share/openocd/scripts/interface/jtagkey-tiny.cfg` und `share/openocd/scripts/board/imx28evk.cfg` bereits vorhanden.

`jtagkey-tiny.cfg` muss um den Eintrag `jtag_rclk 1000` ergänzt werden. OpenOCD wird dann mit folgendem Aufruf gestartet:

```
$ ./bin/openocd -f share/openocd/scripts/interface/jtagkey-
tiny.cfg -f share/openocd/scripts/board/imx28evk.cfg
```

Für eine simple Steuerung des Targets kann nun vom Hostsystem mittels Telnet eine Verbindung zum OpenOCD-Server, auf Port 4444, aufgebaut werden. Die Eingabe des Kommandos `help` bringt Aufschluss über die Funktionen.
Bessere Debug-Möglichkeiten bietet der Einsatz des GDB-Debuggers. Er ist in der ELDK Toolchain bereits unter dem Namen `arm-linux-gnueabi-gdb`, enthalten. Zur Parameterübergabe für den Verbindungsaufbau zu OpenOCD

60 http://openocd.sourceforge.net/doc/html/About.html#About

61 http://openocd.sourceforge.net/doc/html/Debug-Adapter-Hardware.html#Debug-Adapter-Hardware

62 http://openocd.sourceforge.net/doc/html/About.html#About

bietet es sich an, ein .gdbinit Skript für GDB zu erstellen.

```
$ cat .gdbinit
target remote localhost:3333

symbol-file
add-symbol-file u-boot 0x47F83000

monitor gdb_breakpoint_override hard
monitor imx28.cpu arm7_9 fast_memory_access enable
```

GDB wird bei seinem Aufruf damit angewiesen, eine Verbindung zum OpenOCD-Server auf Port 3333 herzustellen.
In diesem Beispiel wurde U-Boot untersucht. U-Boot kopiert sich selbst nach seinem Aufruf in den schnelleren RAM-Speicher um seine Ausführung zu beschleunigen. Wegen dieser Relocation muss die Symboltabelle verworfen und neu mit der entsprechenden Adresse aufgerufen werden.
Sofern ein Zugriff auf U-Boot per Konsole möglich ist, kann die Adresse mit `bdinfo` ermittelt werden. Bezeichnet wird sie als `relocaddr`. Ist kein Zugriff per Konsole auf den U-Boot möglich, kann die Adresse auch aus der Konfiguration des Targets berechnet werden. Die genaue Berechnung kann der U-Boot Dokumentation[63] im Kapitel *10.1.2. Debugging of U-Boot After Relocation* entnommen werden. Dort ist auch eine Beispielberechnung dargestellt. Eine dritte Möglichkeit ist die Präprozessordirektive DEBUG in `arch/arm/lib/board.c` zu definieren. U-Boot wird damit angewiesen erweiterte Debug-Meldungen, u.a. auch die Meldung `relocation Offset is:`, auszugeben.
Die vorgestellte Methode kann natürlich auch verwendet werden um den Linux Kernel zu debuggen. Sie kann für Fehler in einer frühen Startphase des Kernel eingesetzt werden, bevor ein KDB (Kernel Debugger) zur Analyse bereit steht.

63 http://www.denx.de/wiki/DULG/DebuggingUBoot

7.6 Bewertung

Kapitel 7 zeigt, dass es möglich ist, ein komplettes System aus Mainlineversionen zu erstellen. Dieser Weg ist, bedingt durch die fehlende Unterstützung für das TQ-Modul / Board, natürlich durchzogen von manuellen Anpassungen.

Die hier veränderten Mainlineversionen sollen sicher nicht dem produktiven Einsatz, sondern der Darstellung einer exemplarischen Vorgehensmethode dienen. Um eine Reife für den erfolgreichen Einsatz in der Praxis zu erreichen, fehlen Tests dieser Änderungen und die Verifikation einer stabilen Funktion. Auch an den Konfigurationen des Root Filesystems wären zusätzliche Optimierungen notwendig. Jedoch werden die generellen Schritte innerhalb der Portierung von Linux auf ein Zielsystem sichtbar.

Es wird deutlich, wie wichtig eine Rückführung von Änderungen an die einzelnen Projekte ist. In einem optimalen Fall wären alle Anpassungen an das eigentliche Projekt zurückgeführt worden. Ein großer Teil der hier dargestellten Anpassungen wäre hinfällig gewesen, wenn dies geschehen wäre. Es darf jedoch nicht vergessen werden, dass die Pflege solcher Anpassungen innerhalb der Mainlineversionen einen nicht zu unterschätzenden Aufwand für den Maintainer darstellt.

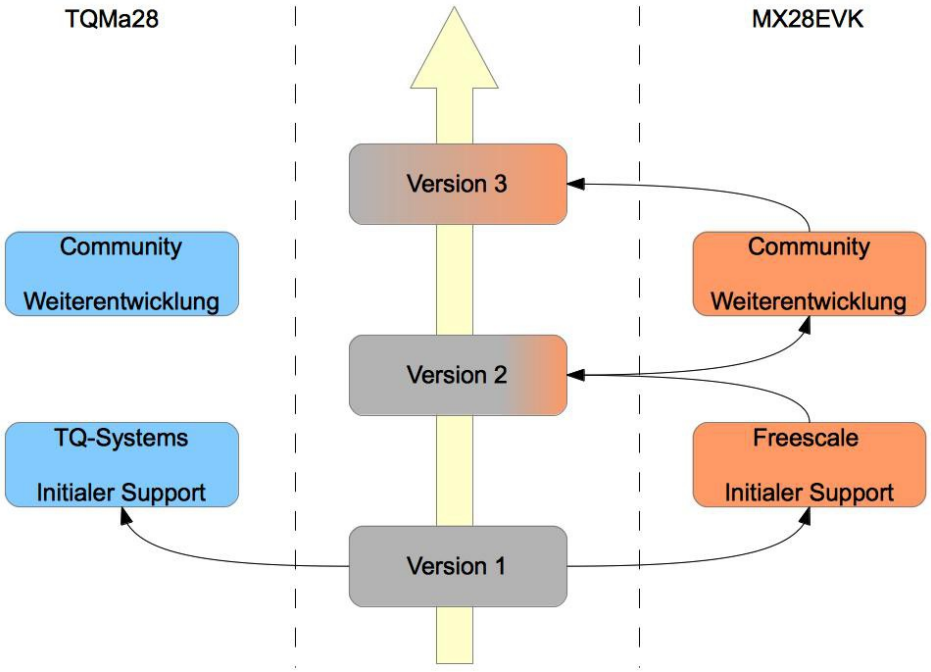

Abbildung 14: OpenSource Software-Entwicklung

Soll eine breite Unterstützung der OpenSource-Community erreicht werden, ist aber die Rückführung eine zwingende Voraussetzung. Sind die Einstiegshürden für Anwender und Entwickler zu hoch oder zu unbequem, wird diese zwangsläufig gering ausfallen oder gänzlich ausbleiben.

8 Yocto Project

Die bisher dargestellten Verfahren sind mit einer Vielzahl von manuellen Anpassungen übersät. Nun steht für viele Anwender von Embedded-Systemen nicht die Anpassung der Systemsoftware, sondern die Zusammenstellung benötigter Anwendungspakete für einen bestimmten Einsatzfall im Vordergrund. Grundsätzlich ist das mit den vorgestellten Methoden zwar möglich, es ist jedoch ein sehr zeitaufwändiges Unterfangen. Wenn sich beispielsweise während der Entwicklung die Hardware-Plattform ändert oder in einer neueren

Produktversion zusätzliche Plattformen integriert werden sollen. Der eigentliche Einsatzzweck ändert sich damit nicht, jedoch müssen alle Schritte erneut durchlaufen werden. Nun wäre ein Konzept wünschenswert, bei dem die Zusammenstellung eines Linux-Systems losgelöst von der eigentlichen Hardware-Plattform wäre.

Einen solchen Ansatz verfolgt das Yocto-Project[64]. Es bietet die Möglichkeit eine komplette Linux-Distribution nach den Vorstellungen des Entwicklers zu generieren. Dabei ist das Yocto Projekt keine weitere Linux-Distribution, sondern ein Build-System mit dem, auf Basis von Metadaten, ein Linux aus den Quellen (from scratch) gebaut werden kann. Dies steht im Gegensatz zum Ansatz von Debian for ARM. Hier wird eine Linux-Distribution aus binären Paketen zusammengestellt.
Es werden die gängigen Embedded Architekturen wie ARM, MIPS, PowerPC und x86 unterstützt. Verwendet wird dazu das Poky Build System. Poky besteht wiederum aus BitBake und Metadaten. Während sich BitBake um den eigentlichen Bau von Paketen kümmert, wird mit den Metadaten beschrieben, was und wie es gebaut werden soll.
Der große Vorteil der Entkoppelung des Systemdesigns von der Zielhardware ist darin zu sehen, dass die Entwicklungsarbeit, auch für andere Hardwarearchitekturen und Embedded Boards wieder verwendet werden kann. Dabei reicht es meist aus eine einzige Zeile in der Konfiguration anzupassen. Dadurch wird die Zeit bis zur Marktreife verkürzt und Entwicklungskosten eingespart.[65]

Um diese hohe Flexibilität und Portabilität zu erreichen spielen zwei Konzepte innerhalb von Yocto eine Schlüsselrolle.
Es werden sogenannte „Rezepte“ (recipe) verwendet, die Angaben darüber enthalten, von wo einzelne Pakete bezogen und wie diese integriert werden sollen. Diese Recipes dienen zur Steuerung von BitBake.

64 http://www.yoctoproject.org

65 Quelle: http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html

Als zweites verwendet Yocto ein Schichten-Modell, sogenannte Layer. Unterstützung für eine spezielle Hardware-Plattform wird als eigene Schicht realisiert. Die Schichten für Hardware-Plattformen (BSP-Layer oder Hardware-Specific BSP) können somit ausgetauscht werden, ohne die eigentliche Konfiguration des Systems ändern zu müssen. In Abbildung 15 sind die einzelnen Schichten grafisch dargestellt.
Dazu kommt der Vorteil, dass solch ein System-Image auch für den Betrieb mit einem Emulator benutzt werden kann. Tests können damit direkt auf dem Hostsystem, auch ohne Target, durchgeführt werden.[66]

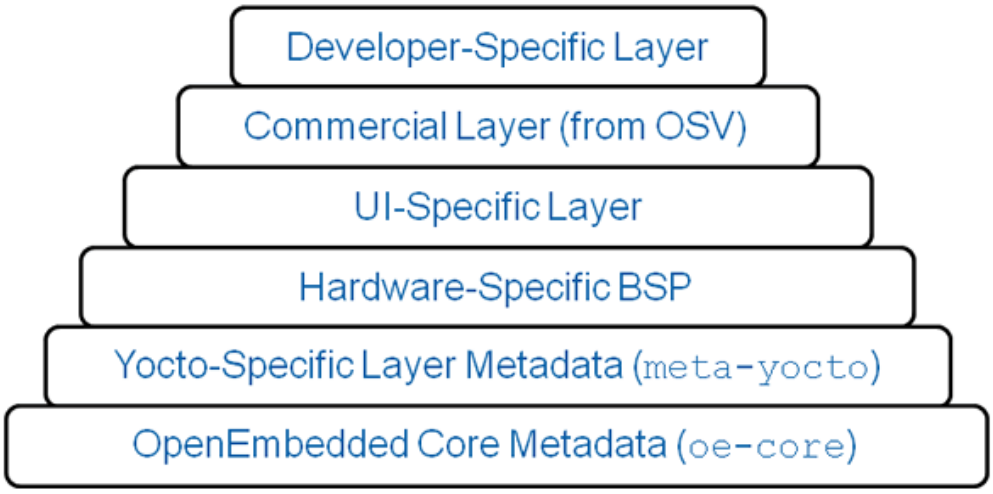

Abbildung 15: Yocto Layers - Quelle: http://www.yoctoproject.org/projects/openembedded-core

BSP-Layer können in der Regel von der Projektseite heruntergeladen werden. Leider sind weder für das TQMa28-Modul noch für das MX28EVK entsprechende Layer dort zu finden. Zumindest Freescale bietet in ihrem Git-Repository[67] den Layer meta-fsl-arm an. Darin werden die Plattformen imx23evk, imx51evk, imx53ard, imx53qsb, imx53qsb, imx6qsabrelite und auch imx28evk unterstützt. Dieser Layer könnte nun, unter Beachtung einiger

66 Quelle: http://www.yoctoproject.org/docs/current/yocto-project-qs/yocto-project-qs.html

67 https://github.com/Freescale/meta-fsl-arm

Abhängigkeiten zu openembedded-core, in das Poky Build System integriert werden.
Eine weiter elegante Variante stellt Freescale in seinem Git-Repository bereit. Unter der Bezeichnung *Freescale's Community Yocto BSP*[68] wird bereits ein von Freescale vorbereitetes, komplettes Yocto/Poky angeboten.

Die Installation erfolgt wie in der README beschrieben:

```
$ mkdir ~/bin
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ mkdir fsl-community-bsp
$ cd fsl-community-bsp
$ repo init -u https://github.com/Freescale/fsl-community-bsp-platform -b denzil
$ repo sync
```

Bei `repo` handelt es sich um ein Python-Skript, dass von Google für Android, zur vereinfachten Verwendung von Git, erstellt wurde.[69]

In der Datei `setup-environment` muss in Zeile 26 das entsprechende Target angegeben werden.

Zeile 26:
```
MACHINE='imx28evk'
```

Soll das Image für die Emulation mit QEMU gebaut werden, muss die Buildarchitektur `imx28evk` durch `qemux86` ersetzt werden. Nun kann der Übersetzungsvorgang gestartet werden:

68 https://github.com/Freescale/fsl-community-bsp-platform

69 Quelle: http://code.google.com/p/git-repo/

```
$ . ./setup-environment build
$ bitbake core-image-minimal
```

Es werden ca. 2300 Pakete heruntergeladen und aus den Quellen übersetzt. Der Zeitaufwand lag bei einem AMD Athlon(tm) 64 X2 Dual Core 4200+ mit 2GB RAM bei mehreren Stunden. Dies kann durchaus als Nachteil gesehen werden, betrifft jedoch nur den initialen Bauvorgang. Sofern die Möglichkeit besteht, sollte das Hostsystem entsprechend performant dimensioniert werden. In diesem Zusammenhang sollten die Variablen `BB_NUMBER_THREADS` und `PARALLEL_MAKE` unter `build/conf/local.conf` entsprechend dem verwendeten Hostsystem angepasst werden. Mit ihnen lässt sich eine optimalere Auslastung erreichen.

Das Ergebnis des Bauvorgangs ist anschließend im Unterverzeichnis `tmp/deploy/images` zu finden. Bereitgestellt werden darin:

- core-image-minimal-imx28evk.ext3
- core-image-minimal-imx28evk.sdcard
- core-image-minimal-imx28evk.tar.bz2
- u-boot-imx28evk.sb
- uImage-imx28evk.bin

Zwar sind U-Boot und Linux Kernel wiederum nicht ohne die entsprechenden Anpassungen an das TQ-Board verwendbar, jedoch kann das Root Filesystem ohne weiteres verwendet werden. Lediglich in der Datei `etc/inittab` muss der serielle Port wieder von `ttyAMA0` auf `ttyAPP3` geändert werden.

Das Image `core-image-minimal-imx28evk.ext3` kann direkt mit `dd` in die dritte Partition geschrieben und verwendet werden.

Das Yocto Project bietet neben dem textorientierten Weg zur Konfiguration des Images eine grafische Oberfläche namens HOB. Zum Start dieser wird im Verzeichnis `build/` das Kommando `hob` ausgeführt. Durch eine in Wizzard-Art geführte Konfiguration des Images setzt es die Einstiegshürde zu Yocto deutlich herab.

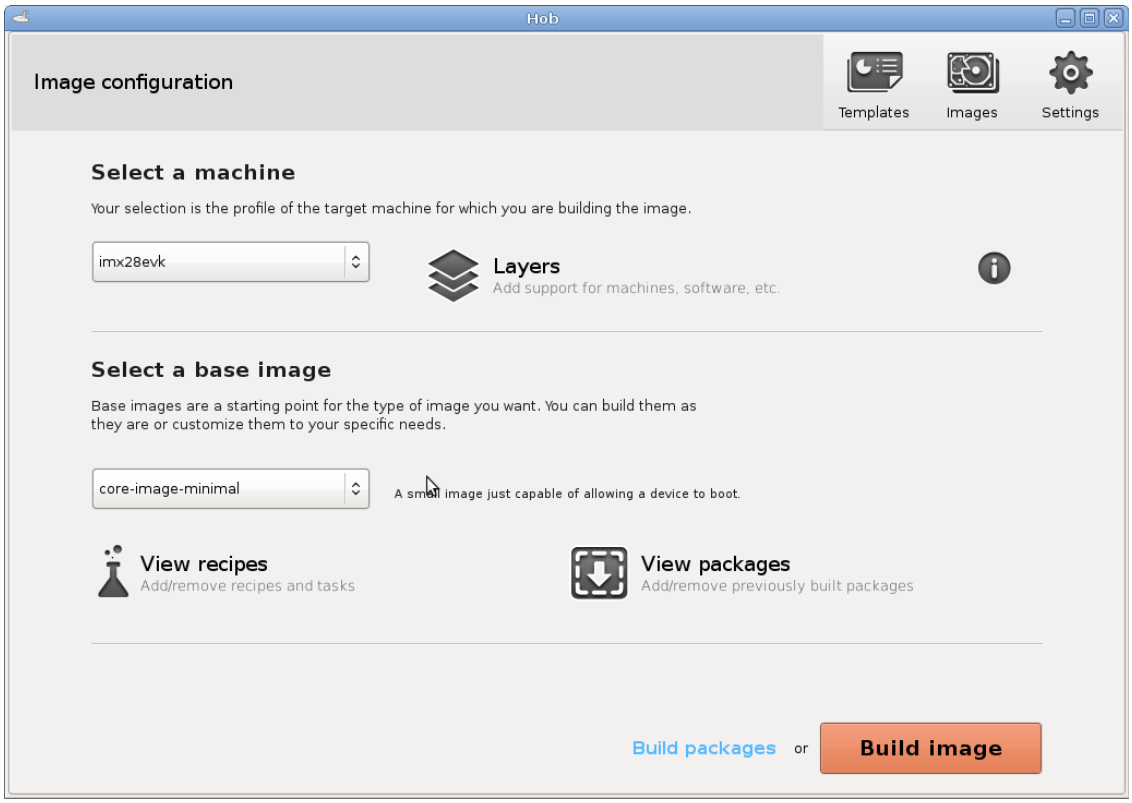

Abbildung 16: HOB Image Konfiguration

Auf einfach Art und Weise lassen sich benutzerdefinierte Images zusammenstellen. „Rezepte“ für zusätzliche Pakete lassen sich einfach aktivieren und werden in das Image integriert.

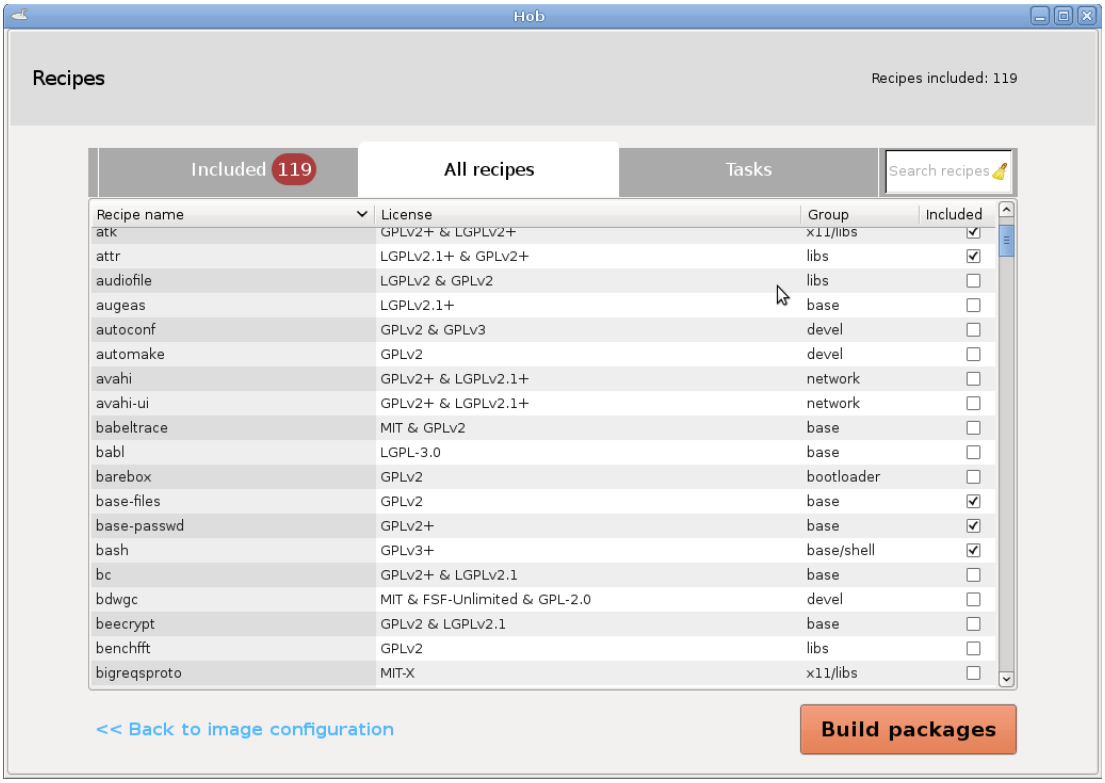

Abbildung 17: HOB Recipes Konfiguration

Für die Paketverwaltung kann zwischen RPM, DEB oder IPK gewählt werden.

Yocto unterstützt rund zwanzig verschiedene Imageformate, darunter jffs2, cramfs, ext2, ext3, btrfs, squashfs und vmdk.

9 Zusammenfassung

Diese Arbeit bietet eine Analyse zum Betrieb von Linux auf der Embedded-Plattform von TQ-Systems und damit auf der i.MX28 Prozessorfamilie. Verschiedene Möglichkeiten zur Anbindung von Zielsystemen an das Entwicklungssystem wurden dargestellt und deren Relevanz im Bezug auf Entwicklungsarbeit und den späteren Produktivbetrieb aufgezeigt.
Sowohl der Funktionsumfang der bereits vorbereiteten Root Filesysteme, wie auch des Kernel-Images wurden dargestellt. Des Weiteren wurde herausgearbeitet, wie eine Portierung von Linux auf eine Zielplattform praxisnah erfolgen kann. Vor allem lag der Fokus dabei darauf, dies auf Basis von öffentlich zugänglichen Quellen und freier Software durchzuführen. Es wurde ein Überblick über die verwendeten Werkzeuge in Verbindung mit den Übersetzungsaufgaben, der Versionskontrolle und der kollaborativen Zusammenarbeit gegeben.
Ein theoretisch optimaler Entwicklungszyklus unter Einbezug der Community wurde dargestellt und nach Parallelen und Abweichungen zur gegebenen Situation gesucht.
Die entsprechenden Anpassungen am U-Boot Loader wie auch Linux-Kernel stellen ein exemplarisches Vorgehen dar und sollen einen Eindruck von den Prozessen zur Portierung bieten. In diesem Zusammenhang wurden auch die notwendigen Debug-Techniken beleuchtet.
Mit dem Yocto-Projekt wurde ein alternatives und modernes Verfahren zur Bereitstellung von Linux auf unterschiedlichsten Zielplattformen dargestellt. Von Interesse in diesem Zusammenhang ist die Verringerung des Aufwandes für den Entwickler, sowohl in zeitlicher Hinsicht wie auch im Bezug auf die Wiederverwendbarkeit seiner Arbeit.

Eine tabellarische Zusammenfassung der Ergebnisse wird in den beiden nachfolgenden Tabellen 8 und 9 gegeben.

Vergleich der Root-Filesysteme

	TQ minimal	TQ Debian	Custom Debian	Yocto minimal
Kernel Version	2.6.35.3	2.6.35.3	3.5.0	2.6.35.3
Buildsystem	ELDK	Multistrap oder debootstrap	Multistrap	Yocto / Poky
Buildsystem Version	4.2	-	2.1.7	1.2 / 7.0.1
Größe	66 MB	736 MB	240 MB	10 MB
Belegter RAM	14 MB	74 MB	14 MB	12 MB
Coreutils	J	J	J	N[2)]
Busybox	N	N	N	J[2)]
Paketmanager	N	DEB	DEB	DEB, RPM, IPK
Repository im Netz vorhanden	N	J	J	(J)
Natives Compilieren	N	J	N[1)]	N[1)]
Shell	bash 4.1.5	bash 4.1.5	bash 4.1.5	bash 4.2.10
Programme und Tools	~120	~1000	~500	~200
Man-Infopages	N	J	N	N
X Unterstützung	J	J	N[1)]	N[1)]

Tabelle 8: Vergleich Rootfile-Systeme

[1)] Nachinstallation möglich

[2)] auch Coreutils möglich

Vergleich der Kernel-Versionen

	Kernel 2.6.35.3	Kernel 3.5.0
Funktion		
USB0	J	J
USB1	J	J
CAN0	J [1)]	J [1)]
CAN1	J [1)]	J [1)]
FEC0 (eth0)	J	J
FEC1 (eth1)	J	J
GPIO (I^2C 0x20)	J	J
GPIO (I^2C 0x21)	J	J
EEPROM (I^2C 0x50)	J	J
LM73 (I^2C 0x4A)	J	J
Audio (I^2C 0x0A)	J	N [2)]
DUART	J (ttyAM0)	J (ttyAMA0)
AUART0 (RS485)	J (ttySP0)	J (ttyAPP0)
AUART1	J (ttySP1)	N [2)]
AUART2	N [4)]	N [4)]
AUART3 (RS232)	J (ttySP3)	J (ttyAPP3)
AUART4	N [5)] (ttySP4)	N [2)]
USER LED1 (I^2C 0x21+4)	J	J
USER LED1 (I^2C 0x21+5)	J	J
LCD	- [3)]	- [3)]
TOUCH (ADCs)	- [3)]	- [3)]

Tabelle 9: Hardwareunterstützung der Kernel

[1)] Keine Testhardware; Device im System vorhanden

[2)] Wird nicht korrekt erkannt; u.U. manuelle Anpassung nötig

[3)] Keine Testhardware vorhanden

[4)] Bedingt durch Hardwaredesign nicht nutzbar

[5)] Device vorhanden, jedoch Kernelfehler bei Zugriff (mxs-auart mxs-auart.4: Unhandled status 52028d)

10 Anhang

Anhang A – U-boot Environment

```
=> printenv
bootcmd=run boot_mmc
bootdelay=3
baudrate=115200
ipaddr=172.16.135.170
serverip=172.16.135.107
gatewayip=172.16.135.254
netmask=255.255.255.0
bootfile="uImage"
loadaddr=0x42000000
rd_size=16384
netdev=eth0
console="ttySP3"
lcdpanel=fg0700
kernel=uImage
rootpath=/exports/nfsroot
ipmode=static
addnfs=setenv bootargs $bootargs root=/dev/nfs rw ramdisk_size=$rd_size
nfsroot=$serverip:$rootpath,v3,tcp rw
addether=setenv bootargs $bootargs fec_mac=$ethaddr
addip_static=setenv bootargs $bootargs ip=$ipaddr:$serverip:$gatewayip:
$netmask:$hostname:$netdev:off
addip_dyn=setenv bootargs $bootargs ip=dhcp
addip=if test "$ipmode" != static; then run addip_dyn; else run
addip_static; fi
addtty=setenv bootargs $bootargs console=$console,$baudrate
addlcd=setenv bootargs $bootargs lcd_panel=$lcdpanel
addmisc=setenv bootargs $bootargs panic=1
boot_nfs=run addether addip addnfs addtty addlcd addmisc; tftp
$loadaddr_kernel $ipaddr/$kernel; bootm
boot_net_debian=setenv rootpath /exports/rootfs/$ipaddr/debian; run boot_nfs
addmmc=setenv bootargs $bootargs root=/dev/mmcblk0p3 ro
boot_mmc=run addether addip addmmc addtty addlcd addmisc; mmc read 0
$loadaddr 104101 3000; bootm
erase_env=mw.b $loadaddr 0 512; mmc write 0 $loadaddr 2 1
n=run boot_net_debian
m=run boot_mmc
p=ping $serverip
ethact=FEC0
```

```
ethaddr=00:D0:93:25:3A:10
eth1addr=00:D0:93:25:3A:11
serial#=TQMa28-PROTO1.100 33163188
stdin=serial
stdout=serial
stderr=serial
ver=U-Boot 2009.08 (Jun 24 2011 - 15:26:35)

Environment size: 1468/130044 bytes
```

Anhang B – U-Boot Support für TQMa28

```
$ git status
# Not currently on any branch.
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      new file:   board/tqc/tqma28/Makefile
#      new file:   board/tqc/tqma28/iomux.c
#      new file:   board/tqc/tqma28/tqma28.c
#      new file:   board/tqc/tqma28/u-boot.bd
#      new file:   include/configs/tqma28.h
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#      modified:   boards.cfg
#
```

Anhang C – U-Boot AUART3 Patch

Siehe CD:
`source/u-boot-patch/0001-Added-AUART-Console.patch`

Anhang D – U-Boot MMC1 Patch

Siehe CD:
`source/u-boot-patch/0002-Added-MMC1-support.patch`

Anhang E – GPIO Kernel Patch

Siehe CD:
`source/linux-3.5-patches/0001-PCA955x-support-added-to-MX28EVK-for-TQMa28.patch`

Funktionstest der GPIOs – User-LEDs

```
root@tqma28:/sys/class/gpio# echo 245 > export
root@tqma28:/sys/class/gpio# echo out > gpio245/direction
root@tqma28:/sys/class/gpio# echo 1 > gpio245/value
root@tqma28:/sys/class/gpio# echo 0 > gpio245/value
```

Literaturverzeichnis

[ARM12] ARM Holding, UK (2012): Company Profile, [online] http://www.arm.com/about/company-profile/index.php (Abgerufen: 23.05.2012)

[CGA11] Canalys (2011): Google's Android becomes the world's leading smart phone platform, [online] http://www.canalys.com/static/press_release/2011/r2011013.pdf (Abgerufen: 04.05.2012)

[DEBARM] debian.org (2012): EmDebian CrossDebootstrap, [online] http://wiki.debian.org/EmDebian/CrossDebootstrap (Abgerufen: 10.07.2012)

[DEBNB] debian.org (2012): Debian GNU/Linux – Installationsanleitung - 4.5. Dateien vorbereiten für TFTP-Netzwerk-Bo, [online] http://www.debian.org/releases/stable/i386/ch04s05.html.de (Abgerufen: 07.2012)

[ELDK12] DENX Software Engineering (2012): DULG, [online] http://www.denx.de/wiki/view/DULG/Manual (Abgerufen: 07.2012)

[EMMC07] IHS (2007): MultiMediaCard Association, JEDEC Adopt eMMC Trademark, [online] http://www.ihs.com/news/jedec-mmca-emmc.htm (Abgerufen: 18.05.2012)

[ES12] Wikipedia (2012): Eingebettetes System, [online] http://de.wikipedia.org/wiki/Eingebettetes_System (Abgerufen: 04.05.2012)

[FBSDPXE] freebsd.org (2012): FreeBSD Handbook - Chapter 32 Advanced Networking: Figure 32-1. PXE Booting process with NFS root mount, [online] http://www.freebsd.org/doc/handbook/network-pxe-nfs.html (Abgerufen: 07.2012)

[GNU12] Free Software Foundation (2012): Was ist GNU?, [online] http://www.gnu.org/ (Abgerufen: 06.05.2012)

[HAL07] Hallinan, Christopher (2007): Embedded Linux Primer, Auflage: Second Printing, Prentice Hall, ISBN: 0-13-167984-8

[HOL05] Craig Hollabaugh, Ph.D. (2005): Embedded Linux, Auflage: 6th Printing, Addison Wesley, ISBN: 0-672-32226-9

[IMX28CEC] Freescale Semiconductor (2012): i.MX28 ApplicationsProcessors for ConsumerProductsSilicon Version 1.2, [online] http://cache.freescale.com/files/32bit/doc/data_sheet/IMX28CEC.pdf (Abgerufen: Rev. 2, 03/2012)

[IMX28RM] Freescale Semiconductor (2010): i.MX28 Applications ProcessorReference Manual, [online] http://cache.freescale.com/files/dsp/doc/ref_manual/MCIMX28RM.pdf?fpsp=1 (Abgerufen: Rev. 1, 2010)

[LD07] Linux Devices (2007): Snapshot of the embedded Linux market, [online] http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Snapshot-of-the-embedded-Linux-market-April-2007/ (Abgerufen: 04.05.2012)

[OTG12] usb.org (2012): USB On-The-Go and Embedded Host, [online] http://www.usb.org/developers/onthego (Abgerufen: 18.05.2012)

[PROGIT] Scott Chacon (2012): Pro Git, [online] http://git-scm.com/book (Abgerufen: 07.2012)

[RST07] Richard Stallman (2007): GNU Users Who Have Never Heard of GNU, [online] http://www.gnu.org/gnu/gnu-users-never-heard-of-gnu.en.html (Abgerufen: 04.05.2012)

[RTAI12] Wikipedia (2012): RTAI, [online] http://de.wikipedia.org/wiki/RTAI (Abgerufen: 07.05.2012)

[RTL12] Wikipedia (2012): RTLinux, [online] http://de.wikipedia.org/wiki/RTLinux (Abgerufen: 07.05.2012)

[STK11] TQ Systems GmbH (2011 - Rev. 102): STK-MBa28User's Manual, [online] http://www.tq-group.com/uploads/tx_abdownloads/files/STK-MBa28.UM.102.pdf (Abgerufen: 18.05.2012)

[TQKa28] TQ Systems GmbH (2012): Hardwarekit zur Systemintegration TQKa28, [online] (Abgerufen: 13.05.2012)

[TQMaP10] TQ Systems GmBH (2010): TQMa28 Multifunktionstalent für den universellen Einsatz, [online] http://www.tq-group.com/fileadmin/web_data/tx_datamintstqproducts/downloads/TQMa28_Produktsteckbrief_Rev.014a.pdf (Abgerufen: 2010 Rev. 014)

[TQMaUM11] TQ Systems GmbH (2011): TQMa28 User's Manual, [online] http://www.tq-group.com/uploads/tx_abdownloads/files/TQMa28.UM.102.pdf (Abgerufen: 2011 Rev. 102)

[TQWIKI] TQ-Systems GmbH (2012): TQ Support Wiki, [online] http://support.tq-group.com (Abgerufen: 2012)

[YMBG08] Karim Yoghmour, Jon Masters, Gilad Ben-Yossef, Philippe Gerum (2008): Building Embedded Linux Systems, Auflage: Second Edition, O'Reilly, ISBN: 978-0-596-52968-0