Entwicklung eines Linux Gerätetreibers am Beispiel eines e-Paper Displays

Anna-Lena Marx

12. Januar 2016

Abstract

Inhaltsverzeichnis

1 Einleitung

Im gerade aufstrebenden Gebiet der eingebetteten Systeme spielt Linux nicht nur als schlankes, gut portierbares Betriebssystem eine tragende Rolle. Auch die Möglichkeit neue Hardware, beispielsweise Sensorik einfach in bestehende Systeme integrieren zu können macht Linux zu einer interessanten Plattform. Während in es in vielen Fällen ausreicht solche Hardware über sehr gut dokumentierte und einfach zu verwendende Schnittstellen im `Userspace` anzusprechen, gibt es doch Anwendungsfälle die Hardwaretreiber im `Kernelspace` erfordern.

Zu letzterem zählen beispielsweise zeitkritische Treiber, die darauf angewiesen sind priorisiert und zu festen Zeitpunkten abgearbeitet zu werden, oder auch Hardwaretreiber im Android-Umfeld, die schon aufgrund des Designs des Systems genauer gesagt dessen Rechteverwaltung nicht im Userspace laufen dürfen.

Diese Projektarbeit soll sich mit solch einem Kerneltreiber auf einem eingebettetem System beschäftigen und exemplarisch darstellen, wie ein Linux-Treiber aufgebaut ist und funktioniert. Dazu wird an einem Beaglebone Black, einem eingebetteten Entwicklerboard, über die `UART`-Schnittstelle ein e-Paper-Display durch ein Linux-Kernel-Modul angebunden. Das Modul soll dem Nutzer die linuxtypischen Treiberschnittstellen bereitstellen und so die Kommunikation mit der Hardware zu ermöglichen.

Der Fokus soll hierbei vor allem auf allgemeinen Prinzipien und Funktionsweisen von Linux-Treibern liegen und erklären was in diesem komplexen, wenig bekannten Bereich ablaufen muss um Hardware so selbstverständlich verwenden zu können, wie dies in heutigen Linux-Systemen der Fall ist.

2 Hardware

2.1 Beaglebone Black

Zur Durchführung der Projektarbeit wird das Beaglebone Black, ein eingebettetes Entwicklerboard auf Basis des ARM-Prozessors AM335x von Texas Instruments eingesetzt. Das Beaglebone bietet mit einer Taktrate von 1GHz (Singlecore), 512MB RAM und vielen zugänglichen Hardware- und Debug-Schnittstellen, sowie einer sehr guten Dokumentation ideale Voraussetzungen für Hardwareintegration und Debugging.

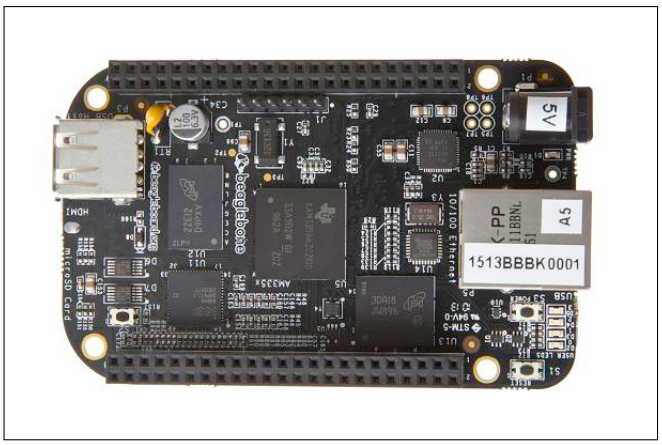
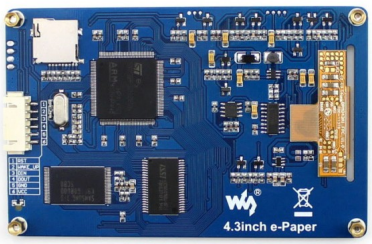


Abbildung 2.1: Beaglebone Black

2.2 Waveshare e-Paper-Display

Bei dem e-Paper-Display handelt es sich um ein Modell des Herstellers Waveshare mit einer Auflösung von 800 x 600 Pixeln und einer Größe von 4,3 inch, dass sehr einfach über die bekannte `UART`-Schnittstelle angesprochen werden kann. E-Paper-Displays sind vor allem dadurch interessant, dass einige Modelle wie auch das Gewählte, in der Lage sind die Anzeige auch ohne Spannungszufuhr aufrecht zu erhalten.

(a) e-Paper Display Front



(b) Rückseite

Abbildung 2.2: Waveshare e-Paper-Display

3 Aufbau und Vorbereitungen

Das Beaglebone wird mit Ubuntu-Linux und dem Kernel 4.1.12-ti-r29 betrieben. Allerdings ist der Treibercode nicht von einer speziellen Kernelversion, einer Linux-Distribution oder einem spezifischen Development-Board abhängig.

Zuerst wurde für diese Arbeit zu Beginn ein Arch-Linux System mit einem Mainline-Kernel der Version 4.3 benutzt, da dieses Setup größtmögliche Freiheit in der Konfiguration des Systems und der Verwendung von Custom-Kernels bot. Leider gab es in dieser Konfiguration ein Problem bei der Verwendung der benötigten `UART`-Schnittstellen dessen Lösung den Rahmen dieser Arbeit sprengen würde und einen Wechsel unvermeidlich werden lies.

Der Treiber wird für die Kernelversionen 4.x der ARM-Plattform geschrieben und ist, solange es keine größeren Änderungen der verwendeten APIs gibt, für jeden entsprechenden Kernel kompilierbar.

3.1 Aufbau der Hardware

Das e-Paper Display besitzt ein `UART`-Interface, welches die Kommunikation zwischen Beaglebone und Display über zwei Pins ermöglicht. Dazu wird die `DIN`-Leitung des Displays an den `TXD`-Pin des `UART-1`-Interface des Beaglebones angeschlossen. Ebenso wird mit der `DOUT`-Leitung und dem `RXD`-Pin des gleichen Interfaces verfahren. Die Leitungen `RST`, der Reset und `WAKEUP` des Displays werden an den `GPIO`[1]-Schnittstellen des Beaglebone angelegt und sorgen dafür, dass der Displayinhalt gelöscht, bzw. das Display aus einem Ruhezustand geholt werden kann. Zuletzt müssen noch Versorgungsspannung (`VCC`, 5V) und Erdung (`GND`) an die entsprechenden Pins des Beaglebone angeschlossen werden.

Das Beaglebone selbst wird über ein USB-Kabel an den Hostrechner angeschlossen und kann darüber über das `ssh`-Protokoll erreicht werden. Um die zusätzliche Hardware versorgen zu können, muss das Board zusätzlich über ein 5V-Netzteil mit Strom versorgt werden.

Um schon ab dem Bootvorgang Kernellogs zeitgleich lesen zu können wird die serielle Debugging-Schnittstelle des Beaglebone mithilfe eines USB-Serial-Wandlers und dem Programm `Minicom` ausgelesen.

[1] General Purpose Input/Output (Allzweck Ein-/Ausgabe), ein Pin dessen Verhalten frei programmiert werden kann

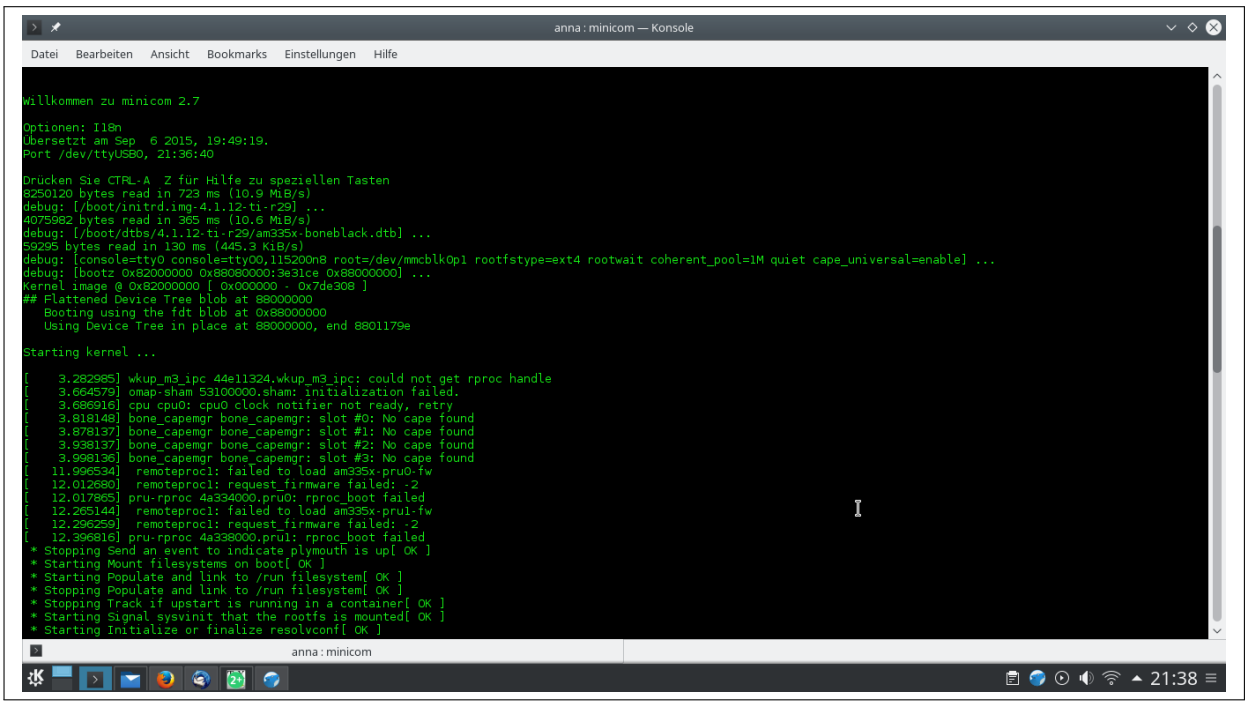


Abbildung 3.1: Minicom Ausgabe während des Boot-Vorgangs

3.2 Vorbereitungen zum Coding

Kernelmodule müssen genau zu der Kernelversion, also den Schnittstellen des Kernels passen, auf dem sie ausgeführt werden sollen. Daher müssen die Quelltexte des Kernels vorliegen um dafür ein Kernelmodul zu erstellen. Sind Zielsystem des Treibers und dass auf dem das Modul kompiliert werden soll identisch, werden die Quelltexte traditionell unter `/usr/src` abgelegt und mit dem `GCC`-Kompiler kompiliert. Handelt es sich wie im Fall dieser Arbeit um unterschiedliche Plattformen und Kernelversionen, muss der Quelltext der genau passenden Kernelversion, sowie ein `GCC`-Compiler für die Zielplattform, ein `Cross-Compiler`, geladen werden. Natürlich kann auch direkt auf der Zielplattform, dem Beaglebone entwickelt werden, allerdings aufgrund dessen geringer Leistungsfähigkeit nicht empfehlenswert.

3.2.1 Kernelquellen

Für die hier verwendete Kernelversion können die Quelltexte wie folgende geladen und kompiliert werden. `CROSS_COMPILE` gibt dabei an, welcher Cross-Compiler verwendet werden soll. Liegt dieser außerhalb der von `PATH`[2] erfassten Pfade, muss hier der vollständige Pfad zum Cross-Compiler angegeben werden.

[2]Linux Umgebungsvariable, die alle Pfade enthält unter denen ausführbare Befehle gesucht werden

```
# Laden der Kernel-Quellen mit dem Versionsverwaltungssystem Git
git clone git@github.com:beagleboard/linux.git
cd linux
git checkout 4.1.12-ti-r29

# Laden der passenden Kernelkonfiguration für das Beaglebone
make ARCH=arm CROSS_COMPILE=arm-none-eabi- bb.org_defconfig
# Bau des Kernels und aller fest integrierter Module auf 4 Threads
make ARCH=arm CROSS_COMPILE=arm-none-eabi- -j4
# Bau der anderen Module (In-Tree)
make ARCH=arm CROSS_COMPILE=arm-none-eabi- modules -j4

# Soll der Kernel im Ganzen eingesetz werden, muss er mit folgenden Befehlen
# auf die  SD-Karte des Beaglebone kopiert werden
make ARCH=arm CROSS_COMPILE=arm-none-eabi- INSTALL_MOD_PATH=/path/to/sdcard/usr modules_install
cp arch/arm/boot/zImage /path/to/sdcard/boot/
```

Listing 1: Laden und Kompilieren der Kernelquellen

3.2.2 Buildsystem

Kernelmodule können entweder als fester Bestandteil des Kernels (In-Tree) oder als dynamisch ladbares Modul außerhalb des Kernels kompiliert werden. Im ersten Fall muss bei jeder Änderung der gesamte Kernel neu kompiliert und auf die SD-Karte des Beaglebone geladen werden, was aufgrund der Größe der Linux-Kernel-Quellen für Entwicklung und Testen eines neuen Treibers nicht zielführend ist. Treiber die in dieser Form in den Linux-Kernel eingebunden sind meist für Hardwarezugriffe verantwortlich die geschehen müssen, bevor der Kernel soweit hochgefahren ist um externe Module laden zu können. Für diese Projektarbeit soll mit der zweiten Möglichkeit, dem dynamisch ladbaren Modul, dass nur gegen die Kernelquellen gelinkt wird gearbeitet werden. Der Vorteil hierbei liegt zum einen darin, dass wirklich nur das Modul laufend neu kompiliert wird, zum anderen ist es so möglich im laufenden Betrieb das Modul zu entladen, gegen eine neuere Variante auszutauschen und wieder zum Kernel zu laden.

Der Build des Linux-Kernels wird über das `GNU Make` Build-Management-System gesteuert, dass auch für das hier behandelte einzelne Kernel-Modul verwenden werden soll. Hierfür muss ein `Makefile` (M zwingend groß geschrieben) erstellt werden. Dass verwendete `Makefile` kann sowohl für ein In-Tree-Build, als auch wie in diesem Fall für ein konkretes Modul außerhalb der Kernelquellen verwendet werden. Um das Modul mit der richtigen Kernelversion zu verknüpfen, wird im Makefile der Pfad zu den Quelltexten angegeben. Stimmt die Kernelversion nicht genau überein, kann das Modul nicht geladen werden!

Mit dem Aufruf von Make wird eine Objektdatei (`waveshare.o`) erzeugt die zusammen mit der Information über die Kernel-Version (`init/vermagic.o`) zum ladbaren Kernelmodul `waveshare.ko` gelinkt wird.

```
# Pfad zu den Kernelquellen
KERNEL_DIR=~/Development/linux

# Objektdatei
obj-$(CONFIG_WAVESHARE) += waveshare.o
obj-m := waveshare.o

# Speichert den Pfad von wo aus make aufgerufen wurde in PWD
PWD := $(shell pwd)

# Ziel für make (all)
all:
	$(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD)

# Ziel für make modules
modules:
	$(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules

# Ziel für make clean
clean:
	$(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD) clean
```

Listing 2: Makefile

Das kompilierte Modul kann an jedem Ort liegen und mit `Root`-Rechten über den Befehl `insmod waveshare.ko` (Pfadangabe ist erforderlich, falls das Modul nicht im aktuellen Verzeichnis liegt) zum Kernel geladen werden. Sollen auch eventuelle Abhängigkeiten des Moduls beim Laden beachtet werden oder das Modul Hotplugging unterstützen, muss das Modul im Zielsystem unter `lib/modules/4.1.12-ti-r29/kernel/drivers/treibergattung` abgelegt und mit dem Befehl `depmod -a` Abhängigkeiten aufgelöst sowie Map-Dateien für die Hotplugging-Infrastruktur erzeugt werden. Danach kann das Modul mit `Root`-Rechten und dem Befehl `modprobe waveshare` geladen werden. Ist das Modul geladen, erscheint es in der Ausgabe von `lsmod`.

4 Kernelmodul

Der Treiber für das e-Paper-Display wird kein von Grunde auf neu geschriebener Treiber werden, sondern soll als sogenannter „Stacked Driver“ gestapelt auf schon vorhandenen Low-Level-Treibern des Linux-Kernels aufgebaut werden. Um das `UART`-Interface im Kernel anzusprechen wird auf dem `serial-core` Low-Level-Treiber für serielle Kommunikation aufgesetzt.

4.0.1 Kernelspace - eine Vorwarnung

Zu beachten ist, dass die C-Standartbibliothek im Kernel nicht zur Verfügung steht. Viele oft benötigte Funktionen werden allerdings als leicht gewichtigere Funktionen angeboten. Ebenso gibt es keinen Speicherschutz wie zwischen Anwendungen im Userspace. Fehler in einem Modul können sich auf den gesamten Kernel auswirken und diesen zum Absturz führen.

4.0.2 Debugging

Debugging ist im Kernel nicht ohne weiteres möglich. Debugger, wie aus modernen IDEs für viele Programmiersprachen bekannt, gibt es so nicht. Dem am nächsten kommt der Kerneldebugger `kgdb`, der es möglich macht auf Hochsprachenniveau Kernelcode zu betrachten. Allerdings sind dazu zwei Rechner mit einem komplexen Aufbau nötig, es dauert sehr lange bis der noch experimentelle `kgdb` lange Symbollisten auflöst und noch nicht alle Hardwareplattformen werden unterstützt. Im Rahmen dieser Arbeit wird daher auf die im Kernel alt hergebrachte Art der Fehlersuche mit `printk` gesetzt, das mit `printf` in der C-Standartbibliothek vergleichbar ist. Über `printk` können mit einer Priorität versehene Kernellogs geschrieben werden, die sich beispielsweise mit `dmesg` auslesen lassen. An relevanten Stellen lassen sich so Ausgaben, zum Beispiel von Variablenwerten generieren.

4.1 Der Treiber

List of Listings

Abbildungsverzeichnis