

Entwicklung eines Linux Gerätetreibers am Beispiel eines e-Paper Displays

Anna-Lena Marx

11. Januar 2016

Abstract

Inhaltsverzeichnis

1	Einleitung	1
2	Hardware	2
2.1	Beaglebone Black	2
2.2	Waveshare e-Paper-Display	2
3	Aufbau und Vorbereitungen	3
3.1	Aufbau der Hardware	3
3.2	Vorbereitungen zum Coding	4
3.2.1	Kernelquellen	4
3.2.2	Buildsystem	4
	Bibliografie	7

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 Kernaltreiber 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.

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.

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 GPIO¹-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.

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

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.

```

1  # Laden der Kernel-Quellen mit dem Versionsverwaltungssystem Git
2  git clone git@github.com:beagleboard/linux.git
3  cd linux
4  git checkout 4.1.12-ti-r29
5
6  # Laden der passenden Kernelkonfiguration für das Beaglebone
7  make ARCH=arm CROSS_COMPILE=arm-none-eabi- bb.org_defconfig
8  # Bau des Kernels und aller fest integrierter Module auf 4 Threads
9  make ARCH=arm CROSS_COMPILE=arm-none-eabi- -j4
10 # Bau der anderen Module (In-Tree)
11 make ARCH=arm CROSS_COMPILE=arm-none-eabi- modules -j4
12
13 # Soll der Kernel im Ganzen eingesetzt werden, muss er mit folgenden Befehlen
14 # auf die SD-Karte des Beaglebone kopiert werden
15 make ARCH=arm CROSS_COMPILE=arm-none-eabi- INSTALL_MOD_PATH=/path/to/sdcard/usr modules_install
16 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

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

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 verwendet 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 Objektdaten (waveshare.o) erzeugt die zusammen mit der Information über die Kernel-Version (init/vermagic.o) zum ladbaren Kernelmodul waveshare.ko gelinkt wird.

```
1  # Pfad zu den Kernelquellen
2  KERNEL_DIR=~/.Development/linux
3
4  # Objektdaten
5  obj-$(CONFIG_WAVESHARE) += waveshare.o
6  obj-m := waveshare.o
7
8  # Speichert den Pfad von wo aus make aufgerufen wurde in PWD
9  PWD := $(shell pwd)
10
11 # Ziel für make (all)
12 all:
13     $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD)
14
15 # Ziel für make modules
16 modules:
17     $(MAKE) -C $(KERNEL_DIR) M=$(PWD) modules
18
19 # Ziel für make clean
20 clean:
21     $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD) clean
```

Listing 2: Makefile

Das kompilierte Modul kann an jedem Ort liegen und mit Root-Rechten über den Be-

3 Aufbau und Vorbereitungen

fehl `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`.

List of Listings

1	Laden und Kompilieren der Kernelquellen	4
2	Makefile	5

Abbildungsverzeichnis