

Entwicklung eines Linux Gerätetreibers am Beispiel eines e-Paper Displays

Anna-Lena Marx

25. Februar 2016

Inhaltsverzeichnis

1 Einleitung	1
2 Hardware	2
2.1 Beaglebone Black	2
2.2 Waveshare e-Paper-Display	2
3 Aufbau und Grundlagen	4
3.1 Aufbau der Hardware	4
3.2 Das UART-Kommunikationsprotokoll des Displays	6
3.3 Vorbereitungen zum Coding	7
3.3.1 Kernelquellen	7
3.3.2 Buildsystem	7
4 Das Kernelmodul - Geräteunabhängige Grundfunktionen	10
4.0.1 Kernelspace - eine Vorwarnung	10
4.0.2 Debugging	10
4.1 Das Grund-Modul - __init und __exit	11
4.1.1 Init des Waveshare e-Paper-Treibers	11
4.1.2 Fehlerbehandlung bei Initialisierungen	13
4.1.3 Alles hat ein Ende - Die exit-Funktion	14
4.2 Die applikationsgegetriggerten Treiberfunktionen	15
4.2.1 waveshare_driver_open()	16
4.2.2 waveshare_driver_close()	17
4.2.3 waveshare_driver_read - Kommunikation zwischen Userspace und Kernelspace	18
4.2.4 waveshare_driver_write - Kommunikation zwischen Userspace und Kernelspace	21
4.2.5 Testen der read und write Funktionen	22
4.2.6 waveshare_driver_poll	24
4.3 Das Sys-Filesystem	25
5 Das Kernelmodul - Der hardwarespezifische Treiber	29
5.1 Zuordnung zwischen Display und Treiber	29
5.1.1 Hotplugging	30
5.2 Die serielle Kommunikation über das UART-Interface im Kernel	31
5.2.1 Registrierung beim Serial Core Treiber	32
5.2.2 Probleme	33

Inhaltsverzeichnis

5.2.3	Die <code>probe()</code> -Funktion	34
5.2.4	Prototypische Steuerung des e-Paper Displays	37
6	Fazit	38
	Bibliografie	40

1 Einleitung

Im inzwischen all gegenwärtigen 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 Hardwaredtreiber im **Kernelspace** erfordern.

Zu letzterem zählen beispielsweise zeitkritische Treiber, die darauf angewiesen sind priorisiert und zu festen Zeitpunkten abgearbeitet zu werden, oder auch Hardwaredtreiber 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 Kerteltreiber 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 (zu sehen in Abbildung 2.1) 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.

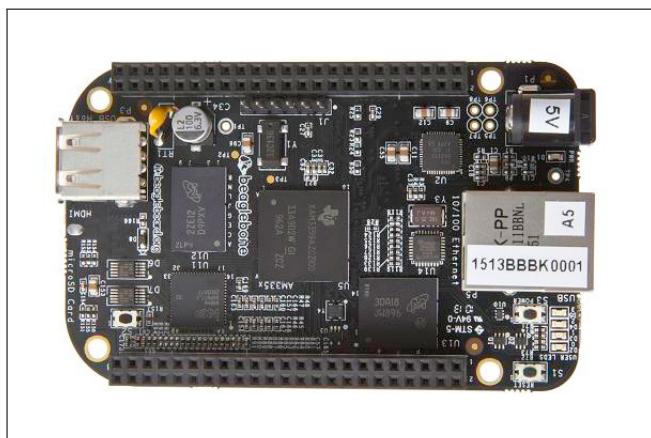
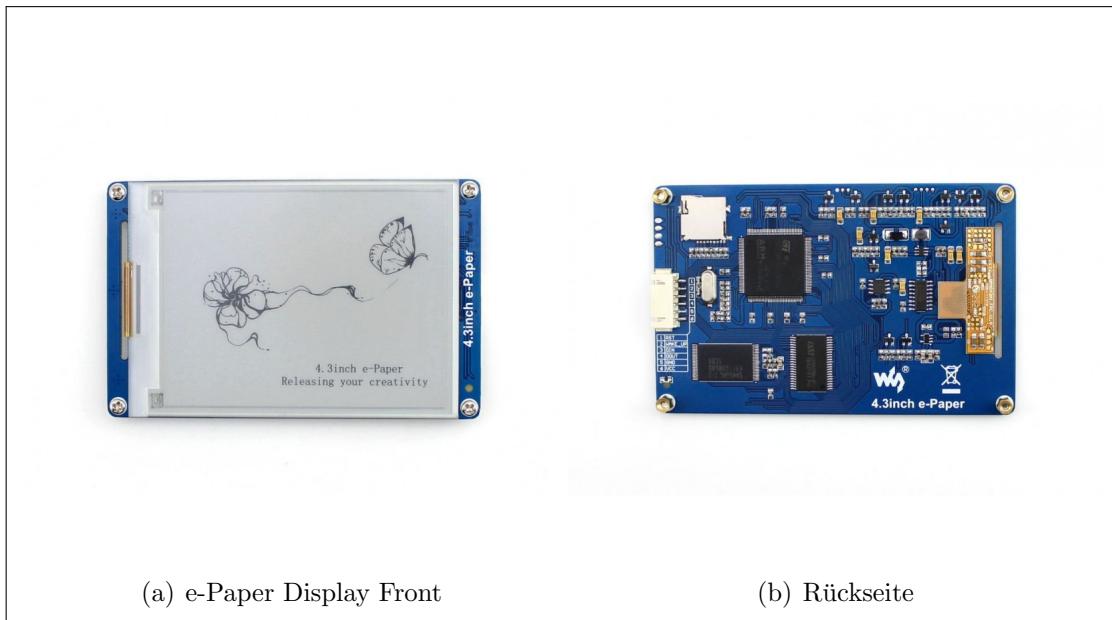


Abbildung 2.1: Beaglebone Black

2.2 Waveshare e-Paper-Display

Bei dem e-Paper-Display (Abbildung 2.2) 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.

2 Hardware



(a) e-Paper Display Front

(b) Rückseite

Abbildung 2.2: Waveshare e-Paper-Display

3 Aufbau und Grundlagen

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 serielle 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. Der Aufbau ist in Abbildung 3.1 zu sehen.

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**, wie in Abbildung 3.2 zu sehen, ausgelesen.

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

3 Aufbau und Grundlagen

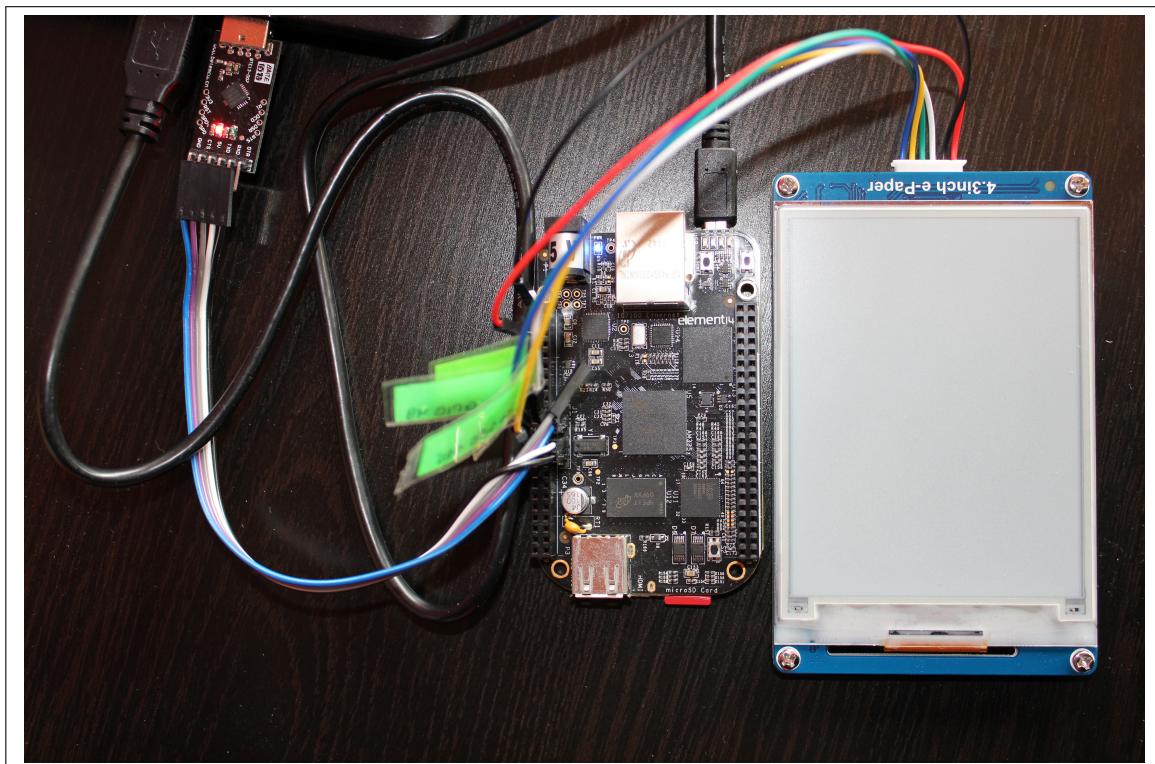


Abbildung 3.1: Aufbau der Hardware

The screenshot shows a terminal window titled "anna:minicom — Konsole". The window displays the boot log of the BeagleBoard. The log starts with "Willkommen zu minicom 2.7" and continues through various initialization steps, including loading the kernel and rootfs, and performing device tree blob operations. The log ends with the message "Starting initialze of finalize resolvconf[OK]". The terminal window has a standard Linux desktop interface at the bottom, including icons for file, copy, paste, and browser, along with a system tray showing the date and time (21:38).

```
anna:minicom — Konsole
Datei Bearbeiten Ansicht Bookmarks Einstellungen Hilfe

Willkommen zu minicom 2.7
Optionen: II8n
Übersetzt am Sep 6 2015, 19:49:19.
Port /dev/ttyUSB0, 21:36:40

Drücken Sie CTRL-A Z für Hilfe zu speziellen Tasten
0250120 bytes read in 723 ms (10.9 MB/s)
Debug: [/boot/initrd.img-4.1.12-ti-r29]...
4075082 bytes read in 365 ms (10.6 MB/s)
Debug: [/boot/dtbz/4.1.12-ti-r29/am335x-boneblack.dtb]...
59295 bytes read in 130 ms (445.3 kB/s)
Debug: [console=tty0 console=tty0,115200n root=/dev/mmcblk0p1 rootfstype=ext4 rootwait coherent_pool=1M quiet cape_universal=enable]...
Debug: [bootz 0xe2000000 0x80000000:3e31ce 0x80000000]...
Kernel image @ 0x82000000 [ 0x0000000 - 0x7de308 ]
## Flattened Device Tree blob at 88000000
  Booting using the fdt blob at 0x88000000
    Using Device Tree in place at 88000000, end 8801179e

Starting kernel ...
[ 3.262085] whup_m3 ipc 4d511224.wkup_m3 ipc: could not get rproc handle
[ 3.664572] omap-sham 53100000:sham: Initialization failed
[ 3.664616] cpus cpuid: cpuid clock notification failed
[ 3.818148] bone_capemgr bone_capemgr: slot #0: No cape found
[ 3.870137] bone_capemgr bone_capemgr: slot #1: No cape found
[ 3.938137] bone_capemgr bone_capemgr: slot #2: No cape found
[ 3.998136] bone_capemgr bone_capemgr: slot #3: No cape found
[11.996534] remoteproc1: failed to load am335x-pru0.fw
[12.012680] remoteproc1: request_firmware failed: -2
[12.017985] pru_rproc 4a334000.pru0: rproc_boot failed
[12.265144] remoteproc1: failed to load am335x-pru1.fw
[12.296259] remoteproc1: request_firmware failed: -2
[12.399616] pru_rproc 4a338000.pru1: rproc_boot failed
* Stopping Send an event to indicate plymouth is up[ OK ]
* Starting Mount filesystems on boot[ OK ]
* Starting Populate and link to /run/filesystem[ OK ]
* Stopping Populate and link to /run/filesystem[ OK ]
* Stopping track if upstart is running in a container[ OK ]
* Starting Signal sysvinit that the rootfs is mounted[ OK ]
* Starting initialize of finalize resolvconf[ OK ]
```

Abbildung 3.2: Minicom Ausgabe während des Boot-Vorgangs

3.2 Das UART-Kommunikationsprotokoll des Displays

Das UART²-Protokoll spezifiziert eine Hardware-Schnittstelle zur seriellen, asynchronen und bidirektionalen Datenübertragung. Für die UART-Spezifikation existieren verschiedene Hardware-Interfaces, die beispielsweise Handshakes zwischen den Kommunikationspartnern ermöglichen. Das Waveshare-Display verwendet hingegen ein einfacheres UART-Interface, dass nur Leitungen für Rx (Receive, Empfangen) und Tx (Transmit, Übertragen) anbietet und auf TTL³-Pegel (5V) arbeitet, TTL-UART genannt. Die Baudrate⁴ mit der das Display Daten überträgt beziehungsweise empfängt ist anpassbar, beträgt aber standardmäßig 115200 Baud.

Befehle an das Display werden in einem spezifizierten Befehlsrahmen übertragen und interpretiert. Um einen plattformübergreifenden Datenaustausch sicher zu stellen, werden die Daten in der sogenannten Network Byte Order, also dem Big Endian Format übertragen. Dabei werden zuerst die höherwertigen Bytes gesendet (Most Significant Byte, MSB), dann die Niederwertigen (Least Significant Byte, LSB). Der Befehlsrahmen umfasst einen Befehlsrahmenkopf (frame header), die Länge des Rahmens, den Typ des Befehls, Befehlsparameter oder Nutzdaten, ein festgelegtes Befehlsrahmenende und ein Parity-Byte. Die Tabelle 3.1 zeigt den Aufbau dieses Befehlsrahmens.

Befehl	0xA5	0xXX XX	0xXX	0xXX	0xCC 33C3 3C	0xXX
Beschreibung	Befehlsrahmenkopf (1 Byte)	Befehlsrahmenlänge (2 Bytes)	Befehlstyp (1 Byte)	Befehlsparameter oder Nutzdaten (0 - 1024 Bytes)	Befehlsrahmenende (4 Bytes)	Parity-Byte (1 Byte)

Tabelle 3.1: Befehlsrahmen Format für das Waveshare Display

Beispielsweise kann mit dem Befehlsrahmen A5 00 09 0A CC 33 C3 3C A6, übertragen über das UART Interface, und dem enthaltenen Befehl 0x0A die Anweisung zum Neuzeichnen des Display Inhaltes übermittelt werden.

In der Dokumentation des Displays⁵ sind alle verfügbaren Befehle aufgelistet und detailliert erklärt, weshalb darauf hier nicht weiter auf diese eingegangen werden soll.

²UART - Universal Asynchronous Receiver Transmitter

³TTL - Transistor-Transistor-Logik

⁴Baud - Einheit für Datenübertragungsrate in der Nachrichtentechnik, gibt an wie viele Symbole (hier Bits) pro Sekunde übertragen werden

⁵<http://www.waveshare.com/w/upload/7/71/4.3inch-e-Paper-UserManual.pdf>

3.3 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.

3.3.1 Kernelquellen

Für die hier verwendete Kernelversion können die Quelltexte wie im folgenden Listing 1 gezeigt, geladen und kompiliert werden. `CROSS_COMPILE` gibt dabei an, welcher Cross-Compiler verwendet werden soll. Liegt dieser außerhalb der von `PATH`⁶ 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

3.3.2 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

⁶Linux Umgebungsvariable, die alle Pfade enthält unter denen ausführbare Befehle gesucht werden

3 Aufbau und Grundlagen

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**, zu sehen in Listing 2, 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.

```
1 # Pfad zu den Kernelquellen
2 KERNEL_DIR=~/Development/linux
3
4 # Objektdatei
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

3 Aufbau und Grundlagen

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 Das Kernelmodul - Geräteunabhängige Grundfunktionen

Am Beispiel des Treibers für das Waveshare e-Paper-Display sollen nun der Aufbau und die prinzipielle Funktionsweise eines Linux-Treibers sowie auftretende Probleme bei der Umsetzung erläutert werden.

Im ersten Teil sollen die grundlegenden Funktionen und üblichen Schnittstellen eines Kernelmoduls anhand des Waveshare Treibers genauer betrachtet werden, während sich der zweite Teil auf die hardwarespezifische Implementierung für das Display konzentriert.

4.0.1 Kernelspace - eine Vorwarnung

Zu beachten ist, dass die C-Standardbibliothek 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 Symbolisten 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-Standardbibliothek 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. Um nicht bei jeder Debug-Ausgabe eine Priorität setzen und ein Kürzel für das eigene Modul einzufügen zu müssen, wird im Treiber hierfür zuerst ein Makro (Listing 3) definiert:

```

1 #ifdef DEBUG
2 #define PRINT(msg)           do { printk(KERN_INFO "waveshare - %s \n", msg); }
3   ↵   while (0)
4 #endif

```

Listing 3: Debug-Macro

4.1 Das Grund-Modul - `__init` und `__exit`

Ein minimales Modul braucht nicht viel mehr als eine `__init`, eine `__exit` Funktion und ein Macro, dass dessen Lizenz angibt. Nur Module, die einer Form der GPL¹ unterliegen, können den vollständigen Umfang der Funktionen des Linux-Kernels benutzen. Die `__init` Funktion wird aufgerufen, sobald das Modul mit `insmod` oder `modprobe` zum Kernel hinzugeladen und so gestartet wird. Sie initialisiert den Treiber und ist abzugrenzen von der `probe` Funktion, die normalerweise dazu genutzt wird nach der grundlegenden Treiberinitialisierung hardwarespezifische Bestandteile des Treibers anzulegen. Die `__exit` Funktion wird aufgerufen, wenn der Treiber entladen werden soll. Alle Initialisierungen und Reservierungen, vor allem reservierter Speicher müssen freigegeben werden.

4.1.1 Init des Waveshare e-Paper-Treibers

Der Name der Init-Funktion ist frei wählbar. Über das Macro `module_init(waveshare_init);` wird definiert, dass hier die parameterlose Funktion mit dem Namen `waveshare_init` und dem Rückgabetyp `int` verwendet werden soll.

Die Initialisierung beginnt mit der Anforderung von Gerätenummern für ein Character basiertes Gerät, die dazu genutzt werden den zugehörigen Treiber zu identifizieren sowie die einzelnen Geräte auseinander zu halten. Sie lösen das Konzept von Major-Nummer (Treiberidentifikation) und Minor-Nummer (Geräte auseinander halten) ab und sind in der Lage mehr physikalische bzw. logische Geräte zu unterscheiden. Gerätenummern und Major-/Minor-Nummern können in einander überführt werden. Zur Anforderung der Gerätenummern wird die Funktion `alloc_chrdev_region()` (Listing 4, Zeile 15) verwendet. Dabei gibt der ist der erste Parameter eine Referenz auf eine Struktur vom Typ `dev_t`, eine Gerätenummer, dort wird die erste angeforderte Nummer abgelegt. Mit dem zweiten wird angegeben bei welchem Wert die Minor-Nummern starten, mit dem dritten, wie viele verschiedene Geräte unterschieden werden sollen. Der letzte Parameter gibt den Namen des Treibers an, der mit der Gerätenummer assoziiert wird. Dabei wird der Treiber für das Gerät beim Kernel angemeldet. Im Fehlerfall springt das Programm zur Sprungmarke `free_device_number`. Die Fehlerbehandlung wird im Folgenden näher erläutert.

¹GNU General Public License; verlangt, dass jedes Programm das in irgendeiner Art und Weise von einem unter GPL lizenzierten Programm abgeleitet wird selbst unter diese Lizenz gestellt und offengelegt werden muss

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

Mit `cdev_alloc()` wird Speicher für ein Objekt des Typs `struct cdev *` alloziert, durch das der hier benötigte zeichenorientierte Gerätetreiber repräsentiert wird. Wird hier nicht der geforderte Pointer auf das instantiierte Objekt zurückgegeben, wird die Fehlerbehandlung bei der Sprungmarke `free_device_number` begonnen.

Dem nun angelegten Treiberobjekt, in C durch ein `struct` repräsentiert, wird im Element `owner` der Besitzer des Treiber mitgeteilt. Im Element `ops` wird ein Verweis auf die `struct fileoperations` angegeben. In dieser `struct` (Listing 4, Zeile 1) sind Pointer auf Funktionen gespeichert, die Interaktion des Treibers mit dem Betriebssystem zur Verfügung stellen. Darunter fallen beispielsweise Funktionen zum Öffnen einer Treiberinstanz, Daten aus dem Treiber auslesen, Daten an den Treiber senden, sowie für das Schließen (Freigeben).

Anschließend wird mit `cdev_add(waveshare_obj, waveshare_dev_number, 1)` das instantiierte Treiberobjekt (`waveshare_obj`) beim Kernel registriert. Mit `waveshare_dev_number` wird die erste Gerätenummer angegeben, mit der Zahl, wie viele Gerätenummern mit dem Treiber verwaltet werden sollen.

Bevor nun auf den hardwarespezifischeren Teil der `init`-Methode eingegangen wird, sollen zuerst die weiteren grundlegenden Treiberbestandteile betrachtet werden.

```

1  static struct file_operations fops = {
2      .owner = THIS_MODULE,
3      .open = waveshare_driver_open,
4      .release = waveshare_driver_close,
5      .read = waveshare_driver_read,
6      .write = waveshare_driver_write,
7      .poll = waveshare_driver_poll,
8  };
9
10
11 static int __init waveshare_init (void) {
12
13 [...]
14
15     if (alloc_chrdev_region (&waveshare_dev_number, 0, 1, WAVESHARE) < 0 ) {
16         goto free_device_number;
17     }
18
19     waveshare_obj = cdev_alloc();
20
21     if (waveshare_obj == NULL) {
22         goto free_device_number;
23     }
24
25     waveshare_obj->owner = THIS_MODULE;
26     waveshare_obj->ops = &fops;
27
28     if (cdev_add (waveshare_obj, waveshare_dev_number,1)) {
29         goto free_cdev;
30     }
31
32 [...]
33
34 }
```

Listing 4: waveshare_init

4.1.2 Fehlerbehandlung bei Initialisierungen

Wie schon erwähnt wird die Fehlerbehandlung bei Treiberinitialisierungen im Kernel über `goto` Sprungmarken realisiert. Während `gotos` in der Applikationsprogrammierung nicht gerne gesehen sind, sind sie im Kernel für das Aufräumen in der korrekten Reihenfolge im Fehlerfall das Mittel der Wahl, da sie hier sehr effizient und gut lesbar eingesetzt werden können.

Die einzelnen Ressourcen des Treibers werden aufeinander aufbauend alloziert und beim Kernel registriert. Geht bei einem Bestandteil etwas schief, müssen alle Ressourcen in umgekehrter Reihenfolge der Initialisierung wieder freigegeben werden, um Speicherlecks zu vermeiden. Geht bei der letzten Initialisierung etwas schief, springt der

Programmlauf zur ersten Sprungmarke und läuft seriell durch die später folgenden der früher Durchgeführten.

Für die komplette `init`-Methode wird eine solche Fehlerbehandlung in Listing 5 dargestellt.

```

1 static int __init waveshare_init (void) {
2
3 [...]
4
5 free_cdev:
6     gpio_set_value(resetPin, !val);
7     gpio_set_value(wakeupPin, !val);
8     gpio_free(resetPin);
9     gpio_free(wakeupPin);
10    PRINT ("adding cdev failed");
11    kobject_put (&waveshare_obj->kobj);
12
13 free_platform:
14    PRINT ("register_platform failed");
15    platform_driver_unregister(&waveshare_serial_driver);
16
17 free_uart:
18    PRINT ("register_uart failed");
19    uart_unregister_driver(&waveshare_uart_driver);
20
21 free_device_number:
22    PRINT ("alloc_chrdev_region or cdev_alloc failed");
23    unregister_chrdev_region (waveshare_dev_number, 1);
24    return -EIO;
25
26 }
```

Listing 5: `waveshare_init` Fehlerbehandlung

4.1.3 Alles hat ein Ende - Die `exit`-Funktion

Die `exit`-Funktion `waveshare_exit()`, zu sehen in Listing 6, wird beim Entladen des Treibers vom Kernels aufgerufen und unterscheidet sich nur unwesentlich von den Aufräumarbeiten im Fehlerfall. Ebenso wie bei der Fehlerbehandlung werden die benutzten Ressourcen in der richtigen Reihenfolge vom Kernel abgemeldet und wieder freigegeben. In der `exit`-Funktion werden wenige andere beziehungsweise zusätzliche Funktionen benutzt, um beispielsweise wie mit `cdev_del` nicht nur das Treiberobjekt beim Kernel abzumelden, sondern auch den zugehörigen Speicher wieder freizugeben.

```

1 static void __exit waveshare_exit (void) {
2
3     gpio_set_value(resetPin, false);
4     gpio_set_value(wakeupPin, false);
5     gpio_free(resetPin);
6     gpio_free(wakeupPin);
7
8     uart_unregister_driver(&waveshare_uart_driver);
9     platform_driver_unregister(&waveshare_serial_driver);
10    device_destroy (waveshare_class, waveshare_dev_number);
11    class_destroy (waveshare_class);
12    cdev_del (waveshare_obj);
13    unregister_chrdev_region (waveshare_dev_number, 1);
14
15    PRINT ("module exited");
16 }
```

Listing 6: waveshare_exit

4.2 Die applikationsgegetriggerten Treiberfunktionen

Um von einer Anwendung aus durch Systemcalls mit einem Hardwaregerät zu kommunizieren, muss das Betriebssystem eine Verknüpfung zwischen dem symbolischen Gerätenamen und der Gerätenummer hergestellt haben. Dies geschieht normalerweise über den udev²-Mechanismus beim Laden des Treibers in der schon vorgestellten Funktion `alloc_chrdev_region()`. Zur Kommunikation selbst werden die Aufrufe der Applikation vom Kernel an die korrespondierenden Treiberfunktionen, die sogenannten applikationsgetriggerten Treiberfunktionen weitergeleitet. Diese sind in der ebenfalls schon erwähnten Struktur `struct file_operations` definiert, die (vergleichbar mit einem Interface in Java) alle möglichen Funktionen die von außen an einen Treiber gerichtet sein können sowie Verweise auf die tatsächlichen Implementierungen enthält.

Allerdings machen nicht alle dieser Systemaufrufe für jeden Treibertyp Sinn und so muss auch nicht jeder implementiert werden. Die gebräuchlichsten und auch für diesen Treiber verwendeten Treiberfunktionen sind `open()`, um eine Zugriffskontrolle auf die Treiberinstanz zu realisieren, dazu korrespondierend `close()`, um eine Treiberinstanz vor allem nach exklusiver Benutzung wieder frei zu geben, sowie die Funktionen für `read()` und `write`, die eine Möglichkeit für Datenaustausch zwischen Userspace und Kernelspace darstellen.

Im Folgenden sollen alle verwendeten applikationsgetriggerten Treiberfunktionen vorgestellt werden.

²udev ist ein Hintergrunddienst, der die Gerätedateien beim Booten bzw. Laden und Entladen eines Gerätetreibers, ausgelöst durch Hotplugging eines Geräts dynamisch verwaltet und für deren Rechteverwaltung zuständig ist

4.2.1 waveshare_driver_open()

Der Systemaufruf der `open()`-Funktion kann über den übergebenen Dateinamen ermitteln ob eine Datei oder wie im hier betrachteten Fall, ein Gerät über einen bestimmten Treiber geöffnet werden soll. Es ist möglich Hardwaregeräte mehreren Treibern zuzuordnen. Das Dateisystem ordnet über diesen symbolischen Gerätenamen das richtige Gerät anhand dessen Gerätenummer zu. Ein Prozess der so auf den Treiber zugreifen möchte, legt eine Struktur `struct file` an, die Parameter für den Zugriff auf den Treiber angibt, also ob lesender oder schreibender Zugriff erfolgen soll, ob es sich dabei um einen blockierenden Zugriff handelt, welches physische Gerät geöffnet werden soll. Diese ruft die im Folgenden näher erläuterte `open`-Funktion des Treibers auf, die hier `waveshare_driver_open()` genannt wird.

Die `waveshare_driver_open()` Funktion ist dafür verantwortlich zu überprüfen ob eine zugreifende Instanz den Treiber öffnen darf.

Der Kernel prüft schon zuvor, ob der Aufruf Zugriffsrechte auf die Datei verletzt, da gerade speziellere Hardware in Linux nur oft mit Root-Rechten benutzt werden darf. Ist die Prüfung des Kernels erfolgreich, wird die `open`-Funktion aufgerufen, die nun ihrerseits prüft, ob der Zugriff mit den in der Struktur `file` angegebenen Anforderungen erfolgen darf. Als Richtlinie für den Waveshare-Display-Treiber wurde festgelegt, dass nur ein aufrufender Prozess der Schreibrechte möchte auf einmal zugreifen darf. Aufrufende Prozesse, die nur lesend Zugreifen wollen, dürfen in beliebiger Anzahl zugelassen werden.

Im Quelltext wird die vom aufrufenden Prozess übergebene Struktur `struct file` als `driverinstance` bezeichnet. Mit `driverinstance->f_flags&0_ACCMODE` (Listing 7, Zeile 5/6) wird abgefragt, ob der Treiber in einem schreibenden Modus geöffnet werden soll. Ist dies der Fall wird ein Zugriffszähler, im Ausgangszustand mit dem Wert -1 versehen, in einem atomaren Vorgang, um eins erhöht und geprüft (Listing 7, Zeile 8). Ist das Ergebnis 0, darf die Instanz nun endlich zugreifen, die `open`-Methode ist zu Ende. Andernfalls wird der Zähler wieder zurückgesetzt und der Instanz mitgeteilt, dass das Gerät momentan beschäftigt ist (Listing 7, Zeile 12/14). Durch den atomaren Zugriff auf den Zugriffszähler wird ausgeschlossen, dass es durch Race Conditions zu inkonsistenten Zuständen kommt. Instanzen, die keinen schreibenden Zugriff benötigen, bekommen jeder Zeit Zugriff gewährt.

```

1 static atomic_t access_counter = ATOMIC_INIT(-1);
2
3 static int waveshare_driver_open (struct inode *devicefile, struct file
4     *driverinstance) {
5
6     if ( ((driverinstance->f_flags&O_ACCMODE) == O_RDWR) ||
7         ((driverinstance->f_flags&O_ACCMODE) == O_WRONLY) ) {
8
9         if (atomic_inc_and_test(&access_counter)) {
10             return 0;
11         }
12
13         atomic_dec(&access_counter);
14         PRINT ("sorry - just one writing instance");
15         return -EBUSY;
16     }
17
18     return 0;
19 }
```

Listing 7: waveshare_driver_open

4.2.2 waveshare_driver_close()

Mit dem Systemaufruf `close()` gibt eine Anwendung über die korrespondierende Treiberfunktion mögliche bestehende Zugriffssperren auf einen Treiber, also eine Ressource, wieder frei. In der Struktur `file_operations` wird die Referenz auf die implementierte `waveshare_driver_close()`-Funktion (Listing 8) unter dem Element `release` gespeichert.

Besitzt die zugreifende Instanz keine Schreibsperren, ist für den hier vorgestellten Treiber nichts weiter zu tun. Andernfalls, wenn das Zugriffsflag einer Instanz Schreibzugriff signalisiert, gibt diese die Ressource frei indem der Zugriffszähler in einem atomaren Vorgang verringert wird.

```

1 static int waveshare_driver_close (struct inode *devicefile, struct file
2     *driverinstance) {
3
4     if ( ((driverinstance->f_flags&O_ACCMODE) == O_RDWR) ||
5         ((driverinstance->f_flags&O_ACCMODE) == O_WRONLY) ) {
6         atomic_dec(&access_counter);
7     }
8
9     return 0;
10 }
```

Listing 8: waveshare_driver_close

4.2.3 waveshare_driver_read - Kommunikation zwischen Userspace und Kernelspace

Im Folgenden soll der Datenaustausch über den Systemcall `read` vorgestellt werden. Die Applikation fordert also über diesen Daten beim Treiber an, die in einen Puffer im Speicherbereich der Anwendung kopiert werden sollen. Dies übernimmt auf der Seite des Treibers die Funktion `waveshare_driver_read()`. Dabei ist zu beachten, dass das Memory-Management die Speicherbereiche von Kernel und Userspace streng voneinander trennt. Zwar übergibt die Anwendung dem Treiber die Adresse des Puffers, doch der darf den Puffer nicht direkt verwenden. Dies übernimmt für eine lesende Anfrage an den Treiber die Kernelfunktion `copy_to_user()`.

Leider kann diese Funktion, sowie weitere Funktionen zur Kommunikation zwischen Userspace und Kernelspace, nur an einem Beispiel statt an einer konkreten Kommunikation erläutert werden. Schwierigkeiten mit dem Aufbau der Verbindung zum Display über die UART-Treiber des Kernels verhindern dies. Der geplante Verbindungsauflauf und die erwähnten Schwierigkeiten werden in dieser Arbeit noch näher erläutert.

Trotzdem soll durch die beispielhafte Implementierung ein Verständnis für die Vorgänge auf Seiten des Treibers bei der Abarbeitung dieses Systemcalls geschaffen werden und erläutert, was für eine konkrete Implementierung für den Anwendungsfall des Displays verändert werden müsste. Die beispielhafte Implementierung ist bis auf die eigentlich vorgesehene Anwendung voll funktionstüchtig.

Das Gegenstück des Systemaufrufes `read` auf Treiberseite ist die Funktion `waveshare_driver_read(struct file *driverinstance, char __user *buffer, size_t max_bytes_to_read, loff_t *offset)`. Über die schon bei `open` erwähnte `struct file *driverinstance` wird hier geprüft ob der Lesezugriff im blockierenden oder nichtblockierenden Modus erfolgen soll. Der zweite Parameter, `char __user *buffer` enthält einen Zeiger auf einen Speicherbereich im Userspace (`__user`), der die zu lesenden Informationen aufnehmen soll. Mit dem dritten Parameter gibt die aufrufende Funktion an, wie viele Byte maximal in den Puffer kopiert werden sollen. Der letzte ist ein Zeiger auf ein Offset, der ein wahlfreien Zugriff innerhalb des Gerätes ermöglicht.

Die beiden Variablen `to_copy` und `not_copied` (Listing 9, Zeile 11/12) geben an, wie viele Bytes noch in den Puffer im Userspace geschrieben werden müssen und wie viele nicht kopiert wurden. Im funktionsfähigen Treiber würde der hier für Beispieldaten eingesetzte Puffer im Kernelspace, `kern_buffer` (Listing 9, Zeile 16), durch einen globalen Displaypuffer ersetzt werden, der zu jeder Zeit den aktuellen Inhalt des Displays speichert, da dieses beispielsweise durch noch nicht durchgeführte Aktualisierungen der Anzeige dem Puffer hinterher hinken kann.

Mit der ersten if-Abfrage (Listing 9, Zeile 24), wird der Fall abgedeckt, dass im nicht-blockierenden Modus gelesen wird, aber keine Daten zum Lesen vorhanden sind. In diesem Fall endet die Funktion mit dem Code `-EAGAIN`.

Ob gelesen werden kann, wird über das Macro `READ_POSSIBLE` (Listing 9, Zeile 5) abgefragt. Es überprüft eine atomare Variable, die im funktionalen Treiber je nach dem aktuellen Füllgrad des Displaypuffers, der identisch zur Anzeige des Displays gehalten

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

werden sollte, gesetzt würde. Im Beispiel verändert sich der Pufferinhalt nicht und die Variable bleibt konstant.

Mit der zweiten if-Abfrage (Listing 9, Zeile 28), wird gesteuert was passiert falls ein schlafender Prozess in dieser Funktion durch ein Signal unterbrochen wird, nämlich der Abbruch mit dem Code **-ERESTARTSYS**.

Nun findet das tatsächliche kopieren der zum Lesen zu Verfügung stehenden Daten in den Userspace statt. Zuerst wird ermittelt wie viele Byte kopiert werden sollen (Listing 9, Zeile 32). Begrenzende Faktoren sind hier wie viel Byte zum Lesen zur Verfügung stehen und wie viele maximal vom aufrufenden Prozess angefordert wurden, der kleinere dieser beiden Werte wird dafür gesetzt. Danach werden die Daten mit der Kernelfunktion `copy_to_user()` unter Angabe von Zielpuffer, Ausgangspuffer und der Menge der zu kopierenden Daten, in den Userspace verschoben. Als Rückgabewert wird angegeben wie viele Byte nicht kopiert werden konnten. Darauf wird der Rückgabewert der Funktion `waveshare_driver_read()` berechnet, der die Differenz zwischen zu kopierenden und kopierten Daten beschreibt.

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

```
1 // Gibt an, wie viele Byte gelesen werden können, sollte im funktionalen
2 // Treiber dynamisch anhand der Daten in einem Displaypuffer angepasst werden.
3 static atomic_t bytes_available = ATOMIC_INIT(5);
4
5 #define READ_POSSIBLE (atomic_read(&bytes_available) != 0)
6
7
8 ssize_t waveshare_driver_read (struct file *driverinstance, char __user *buffer,
9     → size_t max_bytes_to_read, loff_t *offset {
10
11         size_t to_copy = 0;
12         size_t not_copied = 0;
13
14         // Hier sollte der Inhalt eines Displaypuffers ausgelesen werden
15
16         char kern_buffer[128];
17         kern_buffer[0] = 'H';
18         kern_buffer[1] = 'A';
19         kern_buffer[2] = 'L';
20         kern_buffer[3] = 'L';
21         kern_buffer[4] = 'O';
22         kern_buffer[5] = '\n';
23
24         if ((!READ_POSSIBLE) && (driverinstance->f_flags&O_NONBLOCK)) {
25             return -EAGAIN;
26         }
27
28         if (wait_event_interruptible (wq_read, READ_POSSIBLE)) {
29             return -ERESTARTSYS;
30         }
31
32         to_copy = min ((size_t) atomic_read (&bytes_available),
33         → max_bytes_to_read+1);
34         not_copied = copy_to_user (buffer, kern_buffer, to_copy);
35
36         // Werden die Daten im funktionsfähigen Treiber dynamisch angepasst,
37         // muss so berechnet werden, wie viel noch zu lesen ist.
38         //atomic_sub ((to_copy - not_copied), &bytes_available);
39         // *offset += to_copy - not_copied;
40
41         return (to_copy - not_copied);
42 }
```

Listing 9: waveshare_driver_read

4.2.4 `waveshare_driver_write` - Kommunikation zwischen Userspace und Kernelspace

Die Funktion `waveshare_driver_write()`, ausgelöst durch den Systemaufruf `write`, funktioniert in den meisten Aspekten analog zur `waveshare_driver_read()` Funktion. Bei den Übergabeparametern handelt es sich um die selben mit der gleichen Bedeutung, daher wird davon abgesehen diese noch einmal zu erläutern.

In der Funktion wird mit der ersten if-Abfrage der Fall abgefangen, dass der Treiber im nichtblockierenden Modus geöffnet wurde und Hardware oder Treiber nicht bereit sind Daten zu schreiben. Dies wird analog zum Lesen über das Macro `WRITE_POSSIBLE` (Listing 10, Zeile 5) ermittelt, das prüft wie viele Byte geschrieben werden dürfen. Ist dies der Fall, endet die Funktion mit dem Rückgabewert `-EAGAIN`.

Die zweite if-Abfrage behandelt den Fall, dass ein Prozess während des Schlafens durch ein Signal unterbrochen wird und beendet die Funktion in diesem Fall mit `-ERESTARTSYS`.

Darauf folgt der Schreibvorgang der Daten vom Userspace in den Kernelspace. Wie auch zuvor beim Lesen wird wieder der minimale Wert an Bytes die kopiert werden dürfen ermittelt (Listing 10, Zeile 23) und nun mit der Funktion `copy_from_user()` in den Kernelspace geschrieben. Die Bytes, die nicht kopiert werden konnten werden wieder von den zu kopierenden abgezogen und bilden so den Rückgabewert der `waveshare_driver_write()` Funktion. Bei den übertragenen Daten handelt es sich um Strings. Eventuell muss vor der Weiterverarbeitung dieser noch eine Konvertierung erfolgen.

```

1 // Gibt an, wie viele Byte geschrieben werden können, sollte im funktionalen
2 // Treiber dynamisch anhand der Daten in einem Displaypuffer angepasst werden.
3 static atomic_t bytes_to_write = ATOMIC_INIT(10);
4
5 #define WRITE_POSSIBLE (atomic_read(&bytes_to_write) != 0)
6
7
8 ssize_t waveshare_driver_write (struct file *driverinstance, const char __user
9     *buffer, size_t max_bytes_to_write, loff_t *offset) {
10    size_t to_copy;
11    size_t not_copied;
12
13    char kern_buffer [56];
14
15    if ((!WRITE_POSSIBLE) && (driverinstance->f_flags&O_NONBLOCK)) {
16        return -EAGAIN;
17    }
18
19    if (wait_event_interruptible (wq_write, WRITE_POSSIBLE)) {
20        return -ERESTARTSYS;
21    }
22
23    to_copy = min ((size_t) atomic_read(&bytes_to_write),
24    ↳ max_bytes_to_write);
25    not_copied = copy_from_user (kern_buffer, buffer, to_copy);
26
27    // Hier sollte eigentlich der Displaypuffer mit den Eingaben aus dem
28    ↳ Userspace
29    // verändert werden (und evtl. ein Refresh des Displays stattfinden)
30
31    // Werden die Daten im funktionsfähigen Treiber dynamisch angepasst,
32    // muss so berechnet werden, wie viel noch schreiben ist.
33    //atomic_sub ((to_copy - not_copied), &bytes_to_write);
34    // *offset += (to_copy - not_copied);
35
36    return (to_copy - not_copied);
37}

```

Listing 10: waveshare_driver_write

4.2.5 Testen der read und write Funktionen

Ob die Systemaufrufe und vor allem die Implementierungen im Treiber erwartungsgemäß funktionieren, kann unter Linux sehr leicht getestet werden. Das kompilierte Modul wird an die richtige Stelle kopiert und geladen.

Mit dem Befehl `cat` kann die Funktion `waveshare_driver_read()` getestet werden. Allerdings liest `cat` bis EOF, also 0 Byte zurückgegeben wird, was bei der Funktion normalerweise nicht vorkommt, siehe Abbildung 4.1. Dadurch ließt `cat` immer wieder

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

den Inhalt des Treibers, bis der Benutzer manuell via STR+C abbricht.

```
cat /dev/waveshare
```

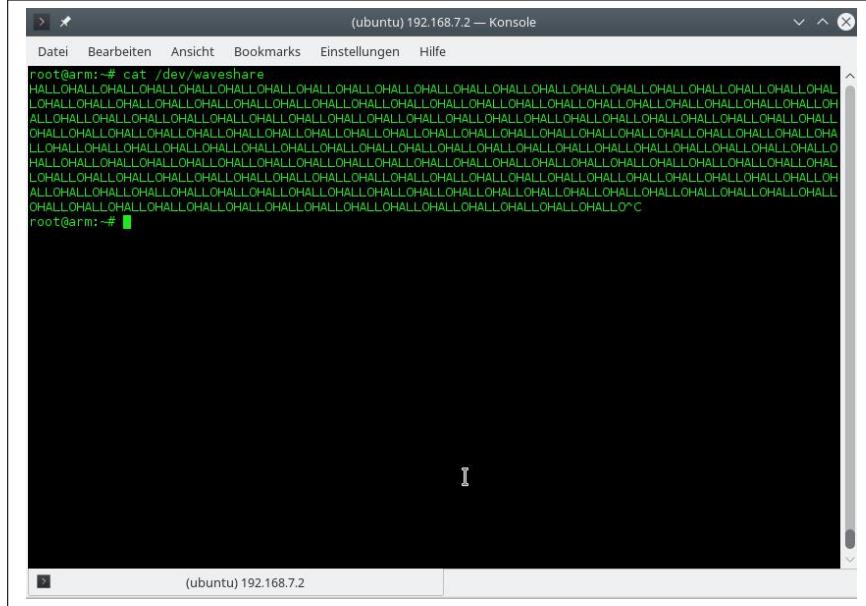


Abbildung 4.1: Lesen vom Treiber aus dem Userspace

Der Befehl `echo` ermöglicht es die Funktion `waveshare_driver_write()` wie in Abbildung 4.2 zu triggern.

```
echo hallo > /dev/waveshare
```

```

root@arm:~# echo hallo > /dev/waveshare
root@arm:~# dmesg | tail
[ 351.063075] waveshare - in the read function
[ 351.063273] waveshare - in the read function
[ 351.063409] waveshare - in the read function
[ 351.063957] waveshare - in the read function
[ 351.064259] waveshare - in the read function
[ 351.064412] waveshare - in the read function
[ 351.065198] waveshare - in the read function
[ 441.289566] waveshare - write - to copy = 6
[ 441.289620] waveshare - write - not_copied = 0 <>[ 441.289666] waveshare - you wrote hallo
[ 441.289666] to driver
root@arm:~#

```

Abbildung 4.2: Schreiben des Treibers aus dem Userspace

4.2.6 waveshare_driver_poll

Mit dem Systemaufruf `poll`, hat eine Applikation die Möglichkeit mehrere Quellen, wie zum Beispiel Treiber oder Dateien auf Veränderungen zu überwachen. Die Implementierung auf Seiten des Treibers hat die Aufgabe der Anwendung mitzuteilen, ob mit dem nächsten Aufruf von `read` Daten gelesen beziehungsweise mit dem nächsten Aufruf von `write` Daten geschrieben werden können. Es ist möglich, dass die aufrufende Anwendung nach dem Aufruf von `poll` schlafen gelegt wird, z.B. wenn keine Daten zum Lesen oder Schreiben vorhanden sind. Um dem Betriebssystem mitzuteilen, dass diese Daten für den Treiber nun wieder vorhanden sind, müssen diesem alle Warteschlangen bekannt gegeben werden, die eine solche wartende Applikation enthalten könnten.

In Listing 11, Zeile 6 und 7 wird gezeigt wie dem Betriebssystem über die Funktion `poll_wait()` die Warteschlangen mitgeteilt werden, über die Prozesse aufgeweckt werden, die auf Daten zum Lesen beziehungsweise zum Schreiben warten.

Die erste if-Abfrage (Zeile 9) prüft das schon bekannte Macro `READ_POSSIBLE` das angibt, ob Daten zum Lesen vorhanden sind. Ist dies der Fall, werden Statusbits, in einer Bitmaske verordert, gesetzt. Dabei bedeutet das Flag `POLLIN`, dass Daten von der Treiberinstanz die nicht blockiert gelesen werden können. Das Flag `POLLRDNORM` gibt an, dass Daten zum Lesen verfügbar sind.

Die zweite if-Abfrage (Zeile 12) prüft über das Macro `WRITE_POSSIBLE` ob Daten geschrieben werden dürfen. In diesem Fall werden die Flags in der Bitmaske auf `POLLOUT` und `POLLWRNORM` gesetzt, womit angegeben wird, dass Daten geschrieben werden dürfen, ohne dass der Treiber blockiert und dass überhaupt Daten geschrieben werden dürfen.

Die Bitmaske, die nun besagte Informationen enthält, wird an die aufrufende Anwendung zurückgegeben.

```

1  unsigned int waveshare_driver_poll (struct file *driverinstance, struct
2  	→  poll_table_struct *event_list) {
3
4
5  	unsigned int mask = 0;
6
7  	poll_wait (driverinstance, &wq_read, event_list);
8  	poll_wait (driverinstance, &wq_write, event_list);
9
10 	if (READ_POSSIBLE) {
11 		mask |= POLLIN | POLLRDNORM;
12 	}
13 	if (WRITE_POSSIBLE) {
14 		mask |= POLLOUT | POLLWRNORM;
15 	}
16
17 	return mask;
18 }
```

Listing 11: waveshare_driver_poll

4.3 Das Sys-Filesystem

Das Sys-Filesystem sysfs, auch Driver-Filesystem genannt, ist ein virtuelles Filesystem, dass im sogenannten Gerätemodell abbildet, wie Prozessoren und Controller mit Peripheriegeräten zusammenhängen. Darin ist nicht nur Hardware verzeichnet, sondern auch zugehörige Software wie zum Beispiel Treiber. Das sysfs löst langfristig das alte Modell des procfs (process filesystem), ein Dateisystem um System- und Prozessinformationen bereitzustellen und zu manipulieren, ab. Für diese Arbeit sollen nur Erweiterungen für das sysfs geschrieben werden.

Da das sysfs dynamisch aus der erkannten Hardware sowie Treibern generiert wird, ist es als Entwickler nur nötig selbst tätig zu werden falls ein nicht standardisiertes Bussystem genutzt oder ein neues integriert wird, falls das Energiemanagement der Geräte durch den Treiber gesteuert werden soll, und wenn über Attribute des sysfs Treiber oder Geräte verändert oder ausgelesen werden sollen. Ebenfalls können neue Geräteklassen definiert werden.

Für den Waveshare-Treiber soll sowohl eine eigene Gerätekasse „waveshare“ definiert werden, sowie Informationsabfrage oder Kontrollzugriff über Attribute des sysfs auf die Hardware. Über solche Kontrollzugriffe, die aus dem Userspace erfolgen, können beispielsweise für Sensoren verschiedene Mess- bzw. Auswertungsintervalle initiiert und kontrolliert werden, ohne die `waveshare_driver_read()` bzw. `waveshare_driver_write()` Funktionen zu ändern oder zu parametrisieren. Für das Display wäre es denkbar über das sysfs verschiedene Funktionen um das Display zu beschreiben anzubieten, die sich beispielsweise im Zeichensatz (das Display unterstützt auch chinesische Schriftzeichen) oder in der Schriftgröße unterscheiden. Natürlich könnten auch schlicht Informationen über Display und Treiber in Attributen hinterlegt werden.

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

Besonders interessant ist hierbei, dass das sysfs selbst im Userspace angesiedelt ist, die Informationen aber vom Kernelmodul erhalten kann, also beispielsweise Sensorergebnisse direkt von einer Messfunktion im Treiber abfrägt. Dadurch entfallen einige Punkte die beim Datenaustausch zwischen User- und Kernelspace in den applikationsgetriggerten Funktionen `read` und `write` beachtet werden mussten.

Auf Grund der Probleme bei der Kommunikation mit der Hardware finden sich auch für die Zugriffsfunktionen auf die sysfs-Attribute nur beispielhafte Implementierungen um Anwendung und Möglichkeiten dieser zusätzlichen Schnittstelle zwischen Treiber bzw. Hardware und Userspace.

Zuerst soll eine Funktion für lesenden Zugriff auf ein sysfs-Attribut erstellt werden (Listing 12). Dabei wird die Information über die `sprintf`-Funktion in den entsprechenden Puffer des Lesenden kopiert. Zurückgegeben wird die Anzahl der kopierten Bytes. In dieser Funktion könnte aber auch beispielsweise `strncpy` oder ähnliches verwendet werden, während in der applikationsgetriggerten Lesefunktion nur der Datenaustausch mittels `copy_to_user()` möglich ist. Hier können z.B. Ergebnisse des Treibers auf sehr einfache Weise im Userspace verfügbar gemacht werden.

```
1 static ssize_t waveshare_sysfs_read (struct device *dev, struct device_attribute
2   __ATTR(*attr, char *buf) {
3
4     char example [] = {"TEST"};
5
6     return sprintf (buf, "%s \n", example);
7 }
```

Listing 12: waveshare_sysfs_read

Nun soll noch eine Funktion für schreibenden Zugriff (Listing 10) auf ein solches Attribut erstellt werden. Es ist nicht zwingend notwendig für ein Attribut immer sowohl lesenden als auch schreibenden Zugriff zu realisieren. Attribute, die nur Informationen über Hardware und/oder Treiber zur Verfügung stellen, werden nur mit einer lesenden Zugriffsfunktion verknüpft. Auch diese Funktion ist nicht sehr spannend, da der Datenaustausch ausschließlich im Userspace stattfindet. Allerdings können so erhaltene Daten im Treiber ausgewertet und weiterverarbeitet werden.

```

1 static ssize_t waveshare_sysfs_write (struct device *dev, struct device_attribute
2     *attr, const char *buf, size_t size) {
3     int val = 0;
4     PRINT ("write a number");
5
6     val = simple_strtoul (buf, NULL, 10);
7     printk (KERN_INFO "You wrote %d \n", val);
8
9     return size;
}

```

Listing 13: waveshare_sysfs_write

Nachdem die Zugriffsfunktionen für das Attribut definiert wurden, werden sie über das Macro DEVICE_ATTR (Listing 14) mit einem Attributnamen verknüpft und entsprechende Zugriffsrechte gesetzt. Der erste Parameter des Macros gibt den Namen des Attributs an, für welches die Zugriffsfunktionen definiert wurden, der Zweite beschreibt die Zugriffsrechte (entweder als Zahl oder als Konstantennamen). Die letzten beiden Parameter geben die Adressen der Funktionen für lesenden bzw. schreibenden Zugriff an. Wird nur einer der beiden Angeboten, bleibt der jeweilig andere Parameter NULL. Dies muss sich auch in den Zugriffsrechten widerspiegeln, ist keine Schreibfunktion definiert, darf auch kein schreibender Zugriff zugelassen werden.

```

1 static DEVICE_ATTR (wav_sysfs, 0644, waveshare_sysfs_read,
2     waveshare_sysfs_write);

```

Listing 14: Das DEVICE_ATTR Macro

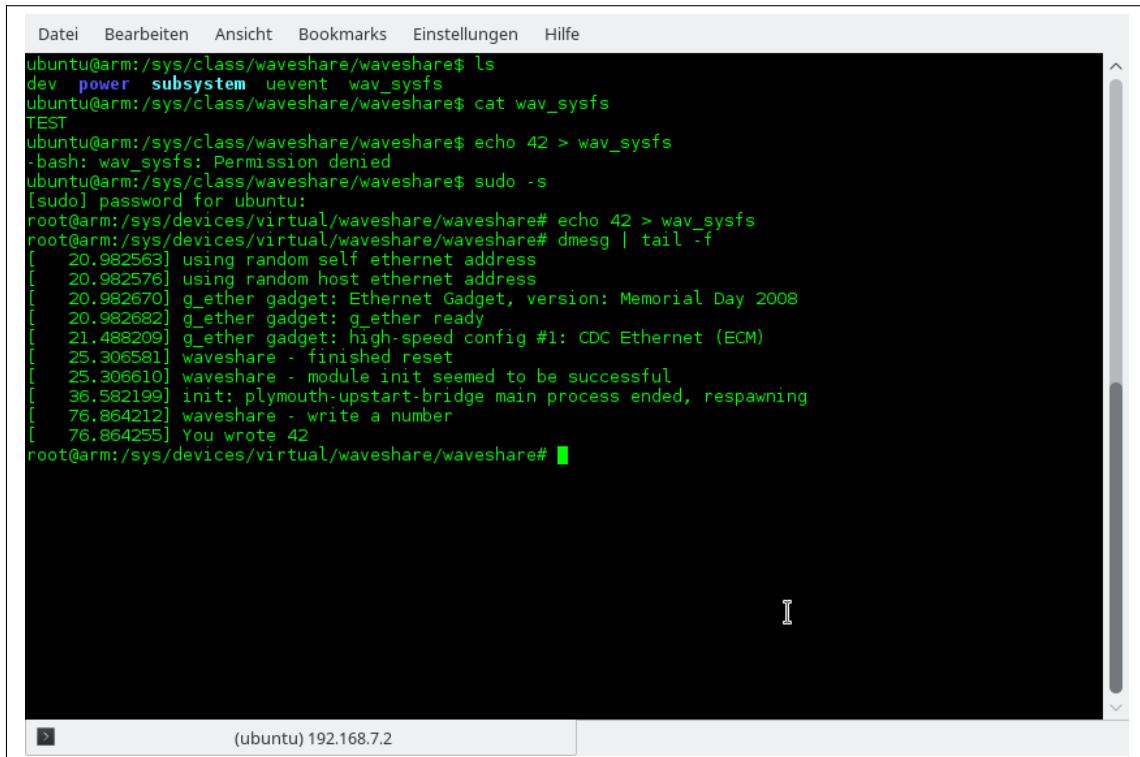
Zuletzt erfolgt noch das Erstellen der eigenen sysfs-Gerätekasse „waveshare“ sowie die Beauftragung des Gerätemodells die Attributdatei anzulegen. Dies geschieht in der waveshare_init Funktion (Listing 15) über class_create und device_create_file.

4 Das Kernelmodul - Geräteunabhängige Grundfunktionen

```
1 static int __init waveshare_init (void) {
2 [...]
3
4         waveshare_class = class_create (THIS_MODULE, WAVESHARE);
5
6         if (IS_ERR (waveshare_class)) {
7             PRINT ("sysfs class creation failed, no udev support");
8             goto free_cdev;
9         }
10
11     waveshare_dev = device_create (waveshare_class,
12                                     NULL, waveshare_dev_number, NULL, "%s", WAVESHARE);
13
14     if(device_create_file(waveshare_dev, &dev_attr_wav_sysfs)) {
15         PRINT ("failed to register attribute file under /dev/waveshare");
16     }
17
18 [...]
19 }
```

Listing 15: sysfs Geräteklaasse und Attribut erstellen

Der Zugriff über die Linux-Kommandozeile auf das sysfs ist in der nachfolgenden Abbildung 4.3 dargestellt.



The screenshot shows a terminal window with a black background and white text. At the top, there's a menu bar with German labels: Datei, Bearbeiten, Ansicht, Bookmarks, Einstellungen, Hilfe. Below the menu, the terminal prompt is "ubuntu@arm:/sys/class/waveshare/waveshare\$". The user runs several commands:

- "ls" shows directory contents: dev, power, subsystem, uevent, wav_sysfs.
- "cat wav_sysfs" shows the current value of the wav_sysfs attribute.
- "echo 42 > wav_sysfs" attempts to write to the attribute, failing with "Permission denied".
- "sudo -s" starts a root shell.
- "echo 42 > wav_sysfs" succeeds in the root shell, changing the attribute value.
- "dmesg | tail -f" shows kernel messages, including logs for the g_ether gadget being initialized and a message about the waveshare module.
- "plymouth-upstart-bridge main process ended, respawning" indicates the system is booting.
- "You wrote 42" is a confirmation message from the echo command.

At the bottom of the terminal window, it says "(ubuntu) 192.168.7.2".

Abbildung 4.3: Zugriff auf das sysfs Attribut

5 Das Kernelmodul - Der hardwarespezifische Treiber

Nachdem nun die Grundfunktionen eines Linux Kernelmoduls vorgestellt wurden, sollen im Folgenden die spezifischen Funktionen für die Kommunikation mit dem Waveshare e-Paper Display über die UART-Schnittstelle erläutert werden.

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

5.1 Zuordnung zwischen Display und Treiber

Eine der größeren Änderungen im Linux Kernel war die Einführung des Device Trees für Computer, die auf ARM Prozessoren basieren. Damit werden fest in den Kernel einkompilierte Board Beschreibungsdateien langsam ersetzt. Nun werden Informationen über verfügbare Schnittstellen, wie beispielsweise UART- oder I2C-Interfaces, dynamisch zum Kernel dazu geladen, der so auch auf der ARM-Plattform generischer verwendbar sein soll.

Dies führt dazu, dass auch im Kernelmodul eine flexiblere Art der Zuordnung zwischen Hardware und Treiber genutzt werden muss und über sogenannte „Compatible-Strings“ (Listing 16, Zeile 4 bzw. 16/17) realisiert wird. Die Compatible-Strings sind in Device Tree Description Files (Ausschnitt siehe Listing 16) für beispielsweise einen Bus oder einen Port definiert.

Im Treiber wird eine Zuordnung zu dem hier benötigten UART-Interface statt. Es wird definiert, dass der Waveshare Treiber geladen werden soll, wenn ein Gerät an einer Schnittstelle des Prozessors, die die Compatible-Strings `ti,am3352-uart` oder `ti,omap3-uart` trägt und die UART-Interfaces des Beaglebone Blacks bezeichnet, erkannt wird. Die Definition auf welche Compatible-Strings geachtet werden soll, geschieht in der `struct of_device_id waveshare_uart_of_ids[]` (Listing 16, Zeile 15).

In der Struktur `platform_driver`, die die Adressen der hardwarespezifischen Treiberfunktionen enthält, wird die `struct of_device_id waveshare_uart_of_ids[]` für den Waveshare-Treiber verwaltet (Listing 16, Zeile 29).

```

1 // In den Kernelquellen
2 // linux/arch/arm/boot/dts/am33xx.dtsi
3 uart0: serial@44e09000 {
4     compatible = "ti,am3352-uart", "ti,omap3-uart";
5     ti,hwmods = "uart1";
6     clock-frequency = <48000000>;
7     reg = <0x44e09000 0x2000>;
8     interrupts = <72>;
9     status = "disabled";
10    dmas = <&edma 26>, <&edma 27>;
11    dma-names = "tx", "rx";
12 };
13
14 // Im Waveshare Treiber waveshare.c
15 static const struct of_device_id waveshare_uart_of_ids[] = {
16     { .compatible = "ti,omap3-uart" ,},
17     { .compatible = "ti,am3352-uartti" ,},
18     { },
19 };
20 MODULE_DEVICE_TABLE(of, waveshare_uart_of_ids);
21
22 static struct platform_driver waveshare_serial_driver = {
23     .probe = waveshare_uart_probe,
24     .remove = waveshare_uart_remove,
25     .id_table = waveshare_uart_of_ids,
26     .driver = {
27         .name = "waveshare_uart",
28         .owner = THIS_MODULE,
29         .of_match_table = waveshare_uart_of_ids,
30     },
31 };

```

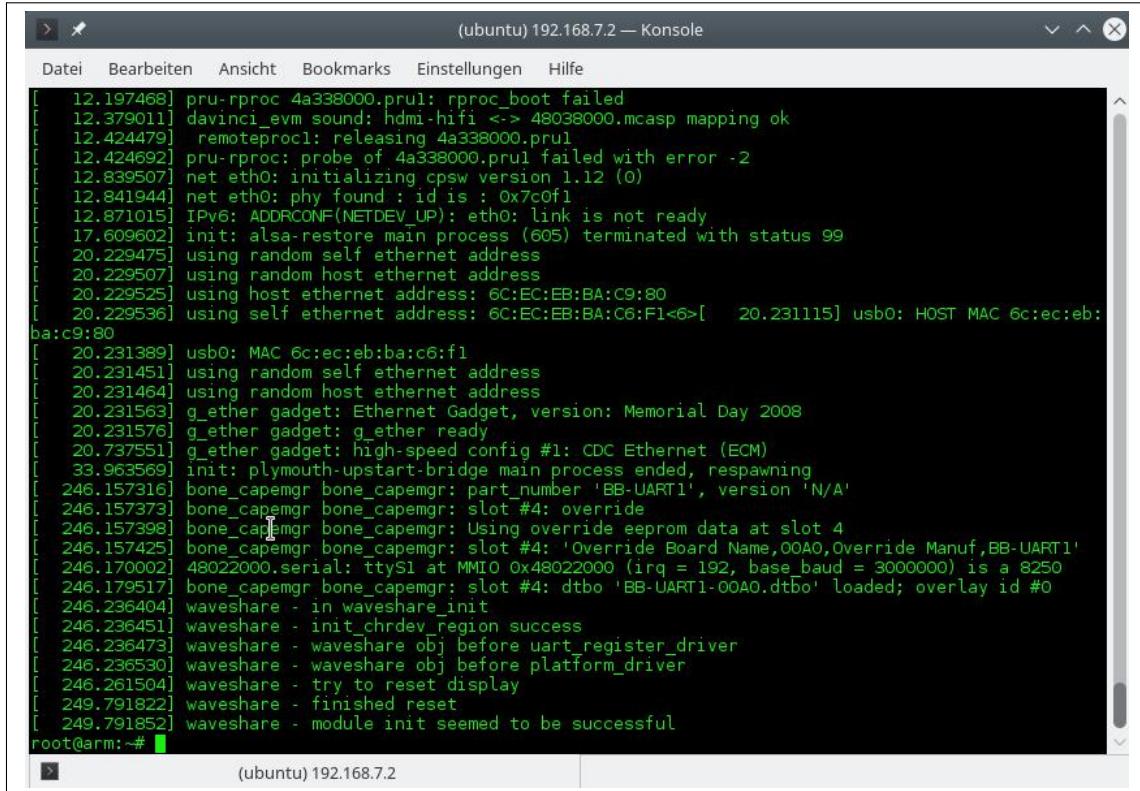
Listing 16: Verknüpfung von Hardware und Treiber über Compatible-Strings

5.1.1 Hotplugging

Eine besondere Bedeutung bei der Zuordnung zwischen Hardware und Treiber hat das Macro `MODULE_DEVICE_TABLE()` (Listing 16, Zeile 20). Es hilft bei der Realisierung des Hotplugging Mechanismus im Kernel, also für ein neu erkanntes Gerät automatisch den passenden Treiber zu laden. Das Macro wird dafür zu Variablen aufgelöst, die Informationen über die angegebene Struktur mit den Compatible-Strings enthalten. Wird das kompilierte Modul zum Kernel geladen und mit `depmod` dessen Abhängigkeiten aufgelöst, sucht `depmod` auch nach den von dem Macro erzeugten Variablen und generiert aus diesen für jeden Bustyp Map-Files, z.B. `modules.usbmap` für Geräte am USB-Bus. Wird ein neues Gerät erkannt, werden die Map-Dateien auf die erzeugten Variablen durchsucht und falls vorhanden, der zugehörige Treiber geladen.

Das folgende Bild 5.1 zeigt die Logs des Kernels nachdem das Kernelmodul in das rich-

tige Verzeichnis kopiert, der Befehl `depmod -a` ausgeführt wurde und das UART1-Interface des Beaglebone Black mit `echo BB-UART1 > /sys/devices/platform/bone_capemgr/slots` aktiviert wurde. Es zeigt ab [246.157316] die Aktivierung von UART1 und anschließend ab [246.236404] die Log- und Debugmeldungen der Initialisierung des Waveshare-Treivers, ohne dass dieser explizit von Hand geladen worden wäre.



```
(ubuntu) 192.168.7.2 — Konssole
Datei Bearbeiten Ansicht Bookmarks Einstellungen Hilfe
[ 12.197468] pru_rproc 4a338000.pru1: rproc_boot failed
[ 12.379011] davinci_evm sound: hdmi-hifi <-> 48038000.mcasp mapping ok
[ 12.424479] remoteproc: releasing 4a338000.pru1
[ 12.424692] pru_rproc: probe of 4a338000.pru1 failed with error -2
[ 12.839507] net eth0: initializing cpsw version 1.12 (0)
[ 12.841944] net eth0: phy found : id is : 0x7c0f1
[ 12.871015] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
[ 17.609602] init: alsu-restore main process (605) terminated with status 99
[ 20.229475] using random self ethernet address
[ 20.229507] using random host ethernet address
[ 20.229525] using host ethernet address: 6C:EC:EB:BA:C9:80
[ 20.229536] using self ethernet address: 6C:EC:EB:BA:C6:F1<>[ 20.231115] usb0: HOST MAC 6c:ec:eb:ba:c9:80
[ 20.231389] usb0: MAC 6c:ec:eb:ba:c6:f1
[ 20.231451] using random self ethernet address
[ 20.231464] using random host ethernet address
[ 20.231563] g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
[ 20.231576] g_ether gadget: g_ether ready
[ 20.737551] g_ether_gadget: high-speed config #1: CDC Ethernet (ECM)
[ 33.963569] init: plymouth-upstart-bridge main process ended, respawning
[ 246.157316] bone_capemgr bone_capemgr: part_number 'BB-UART1', version 'N/A'
[ 246.157373] bone_capemgr bone_capemgr: slot #4: override
[ 246.157398] bone_capemgr bone_capemgr: Using override eeprom data at slot 4
[ 246.157425] bone_capemgr bone_capemgr: slot #4: 'Override Board Name,00A0,Override Manuf,BB-UART1'
[ 246.170002] 48022000.serial: ttyS1 at MMIO 0x48022000 (irq = 192, base baud = 3000000) is a 8250
[ 246.179517] bone_capemgr bone_capemgr: slot #4: dtbo 'BB-UART1-00A0.dtbo' loaded; overlay id #0
[ 246.236404] waveshare - in waveshare_init
[ 246.236451] waveshare - init_chrdev_region success
[ 246.236473] waveshare - waveshare obj before uart_register_driver
[ 246.236530] waveshare - waveshare obj before platform_driver
[ 246.261504] waveshare - try to reset display
[ 249.791822] waveshare - finished reset
[ 249.791852] waveshare - module init seemed to be successful
root@arm:~#
```

Abbildung 5.1: Kernellog Hotplugging

5.2 Die serielle Kommunikation über das UART-Interface im Kernel

Im folgenden wird die Hardwareinitialisierung im Kernel über das UART-Interface vorgestellt. Über dieses Hardware-Interface soll eine serielle Kommunikation realisiert werden. Dazu wird der Serial Core Treiber des Kernels genutzt, auf dem der Waveshare-Treiber als „Stacked Driver“ aufgebaut wird. Serielle Geräte, wie auch das Waveshare-Display, fallen in Linux unter die Kategorie der `tty`-Geräte. Benannt nach der ältesten seriellen Schnittstelle blieb dieser Name bis heute erhalten.

5.2.1 Registrierung beim Serial Core Treiber

Die Initialisierung des UART-Treibers beginnt schon in der Modul-Initialisierung. Hier wird mit der Funktion `uart_register_driver()` (Listing 17, Zeile 21) der Waveshare Treiber beim Serial Core Treiber des Kernels registriert. Dazu wird der Funktion die Struktur `struct uart_driver waveshare_uart_driver` (Zeile 1) übergeben. Darin sind Besitzer, Namen sowie die maximale Anzahl an unterstützten UART Ports und eine eventuell angebotene Konsole angegeben.

Darauf wird der Treiber mit der Funktion `platform_driver_register()` beim Kernel noch als Plattform Treiber angemeldet. In der hier übergebenen Struktur `struct platform_driver waveshare_serial_driver` (Listing 17, Zeile 9) sind die Funktionen hinterlegt, die für die weitergehende Initialisierung der seriellen Kommunikation zur Hardware genutzt werden. Unabhängig vom verwendeten Hardwaretyp werden diese als `probe` und `release` bezeichnet. Der Nutzen der Struktur `waveshare_uart_of_ids` wurde schon zuvor im Kapitel über Hotplugging erläutert.

Nach einer erfolgreichen Initialisierungen sind neue Einträge im proc- und im sys-Dateisystem sichtbar. Dabei handelt es sich z.B. um `/proc/tty/driver/waveshare` sowie um `/sys/bus/platform/drivers/waveshare_uart`, die durch die vorgestellten Funktionen angelegt werden.

```

1 static struct uart_driver waveshare_uart_driver = {
2     .owner = THIS_MODULE,
3     .driver_name = "waveshare",
4     .dev_name = DEVICENAME,
5     .nr = 1,
6     .cons = NULL,
7 };
8
9 static struct platform_driver waveshare_serial_driver = {
10    .probe = waveshare_uart_probe,
11    .remove = waveshare_uart_remove,
12    .id_table = waveshare_uart_of_ids,
13    .driver = {
14        .name = "waveshare_uart",
15        .owner = THIS_MODULE,
16        .of_match_table = waveshare_uart_of_ids,
17    },
18 };
19
20
21 static int __init waveshare_init (void) {
22 [...]
23     if (uart_register_driver(&waveshare_uart_driver)) {
24         goto free_uart;
25     }
26
27     if (platform_driver_register(&waveshare_serial_driver)) {
28         goto free_platform;
29     }
30 [...]
31 }
```

Listing 17: Hardwareabhängige Initialisierung in waveshare_init

5.2.2 Probleme

Nach dieser Modulinitialisierung sollte normalerweise automatisch die in der Struktur `struct platform_driver waveshare_serial_driver` angegebene `probe()`-Funktion aufgerufen werden. Leider passiert dies nicht und die weitere hardwarespezifische Initialisierung in dieser Funktion bleibt unberührt. Woran genau dies scheitert ist sehr schwer herauszufinden, da Debugging im Kernel wie schon erwähnt, nur in einem begrenzten Rahmen möglich ist.

Sowohl `uart_register_driver()` als auch `platform_driver_register()` laufen nicht in die Fehlerbehandlung. Es kann also davon ausgegangen werden, dass beide Funktionen korrekt ausgeführt wurden und der Treiber im Kernel bzw. beim Serial Core Treiber registriert wurde. Auch die Einträge im `proc`- und `sys`-Dateisystem, `/proc/tty/driver/waveshare` sowie `/sys/bus/platform/drivers/waveshare_uart`, die bei einem korrekten Ablauf erwartet werden, sind vorhanden. Ebenfalls für eine kor-

rekte Ausführung an dieser Stelle spricht, dass bei einem expliziten Aufruf der `probe()`-Funktion eine Fehlermeldung erscheint, die besagt, dass der Plattform Treiber schon registriert wurde. Die `probe()`-Funktion wird dabei nicht betreten.

Nach längerer, erfolgloser Fehlersuche wurde in Absprache mit dem betreuenden Professor an dieser Stelle abgebrochen. Dennoch existieren die prototypischen Funktionen `probe()` und `release()` zur hardwarespezifischen Initialisierung bzw. Abmeldung, sowie Funktionen, die darstellen wie ohne die vorhandenen Schwierigkeiten die tatsächliche Kommunikation zum Waveshare Display, anhand der zur Verfügung stehenden API sowie Referenzimplementierungen aus dem Linux Kernel, realisiert werden sollte. Durch besagte Schwierigkeiten konnten diese nicht auf ihre Korrektheit überprüft werden, dennoch sollen diese Funktionen im weiteren Verlauf dieser Arbeit vorgestellt werden, um zu zeigen was der Treiber im geplanten Funktionsumfang leisten sollte.

5.2.3 Die `probe()`-Funktion

Während in der `init`-Funktion vor allem das Kernel Modul selbst initialisiert wird, übernimmt die `probe`-Funktion die Initialisierung des Gerätes, dass am UART-Interface erkannt wurde. Diese Trennung ist nötig, um die Hotplugging Funktionalität zu realisieren. Die folgende Implementierung basiert auf den zur Verfügung stehenden APIs und Referenz-Implementierungen im Linux-Kernel. Die Funktionsfähigkeit konnte nicht überprüft werden.

Die `probe`-Funktion (Listing 18, Zeile 9) beginnt Speicher für die Struktur `waveshare_uart_port` zu mit `devm_kzalloc()` zu allozieren. Besonders an dieser Methode ist, dass der von der Struktur belegte Speicher an die Existenz des Treibers geknüpft ist und danach automatisch freigegeben wird. Der hier in dieser Struktur gespeicherte Port, an dem das Gerät angeschlossen ist, wird als Referenz in der Struktur `uart_port` des Serial Core Treibers gespeichert. Anschließend werden Ressourcen für das Plattform-Gerät reserviert und unter `port->mempool` wird über die Funktion `devm_ioremap_resource()` (Listing 18, Zeile 24) der I/O-Speicher dieses Geräts zugreifbar gemacht. Über diesen Speicher sollen Daten zwischen Treiber und Hardware übertragen werden. In `port->ops` (Zeile 32) wird eine Referenz auf Funktionen gespeichert, die das Verhalten des über das UART-Interface angeschlossenen Geräts definieren. Im weiteren wird vorgestellt, wie eine solche Funktion für die Initialisierung des e-Paper-Displays aussehen könnte. Im letzten interessanten Part der `probe`-Funktion wird mit der Funktion `uart_add_one_port()` (Zeile 36) der vorher definierte UART-Port mit der `uart_driver`-Struktur beim Serial Core Treiber registriert. Abschließend werden `platform_set_drvdata()` die treiberspezifischen Daten für das Plattformgerät gesetzt.

Das Gegenstück zur `probe()`-Funktion ist die `release()`-Funktion, hier `waveshare_uart_remove()`, in der die getätigten Registrierungen beim Serial Core Treiber bzw. beim Kernel aufgehoben werden müssen. Allozierter Speicher muss nicht explizit freigegeben werden, da dies durch die besondere Allokation automatisch geschieht.

5 Das Kernelmodul - Der hardwarespezifische Treiber

So bleibt nur noch mit der Funktion `uart_remove_one_port()` die Registrierung beim Serial Core Treiber zu lösen.

```

1  struct waveshare_uart_port {
2      struct uart_port port;
3  };
4
5  static struct uart_ops waveshare_uart_ops = {
6      .startup = waveshare_uart_startup,
7  };
8
9  static int waveshare_uart_probe (struct platform_device *pdev) {
10
11      struct waveshare_uart_port *wav_port;
12      struct uart_port *port;
13      struct resource *mem_res;
14      unsigned int baud;
15
16      wav_port = devm_kzalloc (&pdev->dev,
17          sizeof (struct waveshare_uart_port), GFP_KERNEL);
18      if (!wav_port) {
19          return -EINVAL;
20      }
21
22      port = &wav_port->port;
23
24      mem_res = platform_get_resource (pdev, IORESOURCE_MEM, 0);
25      port->mempbase = devm_ioremap_resource(&pdev->dev, mem_res);
26      if (IS_ERR(port->mempbase)) {
27          return PTR_ERR(port->mempbase);
28      }
29
30      port->mapbase = mem_res->start;
31      port->dev = &pdev->dev;
32
33      port->ops = &waveshare_uart_ops;
34
35      baud = 115200;
36
37      if (uart_add_one_port(&waveshare_uart_driver, &wav_port->port)) {
38          goto free_uart_add_one_port;
39      }
40
41      platform_set_drvdata(pdev, wav_port);
42
43      return 0;
44
45 free_uart_add_one_port:
46     return -1;
47 }
```

Listing 18: Hardwareabhängige Initialisierung in waveshare_uart_probe()

5.2.4 Prototypische Steuerung des e-Paper Displays

Wie schon angerissen, soll erläutert werden wie die Funktion zum Start des Displays, auf die in der Struktur `struct uart_ops waveshare_uart_ops` unter `.startup` verwiesen wird, realisiert werden könnte. Da das e-Paper Display in der Lage ist die Darstellung auch ohne Spannungszufuhr zu halten soll beim Start zuerst eine Leerung des Displayinhalts durchgeführt werden, um mit einem definierten Ausgangszustand, also einer leeren Anzeige, zu starten. Laut dem Manual des Herstellers (Literaturverzeichnis [1]), lautet der benötigte Befehlsrahmen A5 00 09 2E CC 33 C3 3C 82.

Um diesen übermitteln zu können wird eine Hilfsfunktion `waveshare_uart_write()` (Listing 19, Zeile 1) definiert, die ein einzelnes Byte über schon erwähntem I/O-Speicher überträgt.

Die Funktion `waveshare_uart_startup()` selbst, stellt daher nur noch die Übertragung des Befehlsrahmens über die Hilfsfunktion dar.

Auf diese Weise könnten weitere Befehle sowie das Setzen von Text und Grafiken entsprechend der vorgesehenen Befehlsrahmen realisiert werden, um die Funktionalität des Treibers zu vervollständigen. Text zur Darstellung auf dem Display könnte über die `write()`-Funktion oder Attributdateien im `sysfs` in den Kernelspace eingelesen werden, dort verarbeitet und in den entsprechenden Befehlsrahmen eingebettet an das Display gesendet werden.

```

1 static inline void waveshare_uart_write (struct waveshare_uart_port *up, int
2   ↵ offset, char value) {
3     return writeb (value, up->port.membase + offset);
4   }
5
5 static int waveshare_uart_startup (struct uart_port *port) {
6
7   struct waveshare_uart_port *wave_port = container_of (port, struct
8   ↵ waveshare_uart_port, port);
9
9   waveshare_uart_write (wave_port, 0x00, 0xA5);
10  waveshare_uart_write (wave_port, 0x00, 0x00);
11  waveshare_uart_write (wave_port, 0x00, 0x09);
12  waveshare_uart_write (wave_port, 0x00, 0x2E);
13  waveshare_uart_write (wave_port, 0x00, 0xCC);
14  waveshare_uart_write (wave_port, 0x00, 0x33);
15  waveshare_uart_write (wave_port, 0x00, 0xC3);
16  waveshare_uart_write (wave_port, 0x00, 0x3C);
17  waveshare_uart_write (wave_port, 0x00, 0x82);
18
19  return 0;
20 }
```

Listing 19: Reset des Displays

6 Fazit

Das Ziel dieser Projektarbeit war es einen im Linux Kernel angesiedelten Treiber für das Waveshare e-Paper Display zu schreiben. Dazu wurde ein allgemeines Kernelmodul geschrieben und um die Treiberfunktionalität für das Display erweitert. Hierbei sollte besondere Aufmerksamkeit in die Unterschiede zur klassischen Anwendungsentwicklung fließen.

Während Schnittstellen in dieser normalerweise sehr gut für andere Entwickler dokumentiert und spezifiziert sind, muss der Entwickler im Kernelspace mit eher minimalistischer Dokumentation arbeiten. Oft gilt hier „der Quellcode ist genug Dokumentation“ und so ist es nötig sehr viel Code zu lesen um Funktionsweisen und Einsatz einiger Mechanismen zu verstehen. Ebenso kommt es durch die Entwicklung des Linux Kernels in relativ kurzen Intervallen oft zu Änderungen der Kernel-APIs. Dadurch sind die Dokumentationen im Kernel nicht immer aktuell und die wenigen vorhandenen Fachbücher zum Linux Kernel sehr schnell hoffnungslos veraltet. Es erfordert eine hohe Einsatzbereitschaft des Entwicklers mit dieser Situation umzugehen und sich immer wieder neu einzulesen. Andererseits bringt das Entwickeln im Kernelspace ein sehr umfassendes Verständnis von den Vorgängen im Betriebssystem, die auch der klassischen Anwendungsentwicklung zu Gute kommen. Im Kernel müssen viele Konzepte beachtet werden, die sich auch z.B. in der parallelen Anwendungsprogrammierung wiederfinden, da bei einem Treiber immer davon ausgegangen werden muss, dass dieser von mehreren Prozessen gleichzeitig verwendet wird.

Bedauerlicherweise wurden in dieser Arbeit nicht alle gesteckten Ziele erreicht. So ist die Treiberfunktionalität nicht gegeben, während das grundlegende Kernelmodul ordnungsgemäß arbeitet und dem Anwender die üblichen Schnittstellen zum Zugriff bereitstellt. Dass es nicht möglich war in der vorhandenen Zeit das Problem zu lösen liegt auch an einem weiteren Unterschied zur Anwendungsentwicklung. Einen Debugger der in der Lage ist auf Hochsprachenebene in Echtzeit Schritt für Schritt durch Kernelcode zu gehen gibt es nur in einem experimentellem Stadium und nicht für die verwendete ARM-Plattform. Ebenso erwies sich die Realisierung der seriellen Kommunikation über das UART-Interface im Kernel als deutlich aufwendiger und komplexer als von ähnlichen Schnittstellen im Kernel bekannt. Der vorhandene Low-Level-Treiber ist sehr schlecht in seiner genauen Funktionsweise dokumentiert und überlässt dem Entwickler viel Arbeit auf der Ebene der Byte-Übertragung. Dadurch ist es sehr schwierig auftretende Probleme zuzuordnen und zu beheben. Für das konkrete Problem in dieser Arbeit lässt sich kein Fehler feststellen, wieso der Treiber nicht in die erforderliche Funktion läuft, konnte nicht geklärt werden. An dieser Thematik könnte weiter angeknüpft werden.

Trotz aller Eigenheiten die die Entwicklung im Kernelspace mit sich bringt und den ambitionierten Zielen dieser Arbeit war die Entwicklung des Treibers eine sehr lohnens-

6 Fazit

werte Erfahrung mit sehr vielen Einblicken in die Funktionsweise eines Betriebssystems, die in jedem Bereich der Informatik von Nutzen sind.

Literatur

- [1] *4.3inch e-Paper User Manual.* "<http://www.waveshare.com/w/upload/7/71/4.3inch-e-Paper-UserManual.pdf>". waveshare, Apr. 2015.
- [2] *Archlinux ARM Beaglebone Black.* "<http://archlinuxarm.org/platforms/armv7/ti/beaglebone-black>".
- [3] Shraddha Barke, Hrsg. *Kernel Build.* "<http://kernelnewbies.org/KernelBuild>". Okt. 2015.
- [4] *Building Beaglebone Black Kernel.* "http://wiki.beyondlogic.org/index.php/BeagleBoneBlack_Building_Kernel".
- [5] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman. *Linux Device Drivers* -. Sebastopol: O'Reilly Media, Inc., 2005. ISBN: 9780596555382.
- [6] Jonathon Corbet. *Platform devices and device trees.* "<https://lwn.net/Articles/448502/>". Juni 2011.
- [7] Jonathon Corbet. *The platform device API.* "<https://lwn.net/Articles/448499/>". Juni 2011.
- [8] *Kernel -Serial controller device driver programming.* "<http://free-electrons.com/doc/serial-drivers-lab.pdf>". Free Electrons.
- [9] Greg Kroah-Hartman. *Hotpluggable devices and the Linux kernel.* "<https://www.kernel.org/doc/ols/2001/hotplug.pdf>". 2001.
- [10] Greg Kroah-Hartman. *How to Create a sysfs File Correctly.* "<http://kroah.com/log/blog/2013/06/26/how-to-create-a-sysfs-file-correctly/>". Juni 2013.
- [11] *Linux Drivers Device Tree Guide.* "http://elinux.org/Linux_Drivers_Device_Tree_Guide".
- [12] *Low Level Serial API.* "<https://www.kernel.org/doc/Documentation/serial/driver>".
- [13] Alexander Patrakov. *Writing Systemd Files.* "<http://patrakov.blogspot.de/2011/01/writing-systemd-service-files.html>". Jan. 2011.
- [14] Thomas Petazzoni. *Serial Drivers.* "<http://free-electrons.com/doc/serial-drivers.pdf>". Free Electrons, Dez. 2010.
- [15] Jürgen Quade und Eva-Katharina Kunst. *Linux-Treiber entwickeln - Eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung - jetzt auch für Raspberry Pi.* 4. akt. und erw. Aufl. Heidelberg: Dpunkt.Verlag GmbH, 2015. ISBN: 9783864902888.

Literatur

- [16] *Serial over USB on Beaglebone Black.* ”<http://hipstercircuits.com/serial-over-usb-on-beaglebone/>”.
- [17] *serial_core.h.* ”http://lxr.free-electrons.com/source/include/linux/serial_core.h”. Deep Blue Solutions Ltd., 2000.
- [18] *The Linux Kernel API.* ”<https://www.kernel.org/doc/htmldocs/kernel-api/>”.

Verzeichnis der Quelltexte

1	Laden und Kompilieren der Kernelquellen	7
2	Makefile	8
3	Debug-Macro	11
4	waveshare_init	13
5	waveshare_init Fehlerbehandlung	14
6	waveshare_exit	15
7	waveshare_driver_open	17
8	waveshare_driver_close	17
9	waveshare_driver_read	20
10	waveshare_driver_write	22
11	waveshare_driver_poll	25
12	waveshare_sysfs_read	26
13	waveshare_sysfs_write	27
14	Das DEVICE_ATTR Macro	27
15	sysfs Geräteklaasse und Attribut erstellen	28
16	Verknüpfung von Hardware und Treiber über Compatible-Strings	30
17	Hardwareabhängige Initialisierung in waveshare_init	33
18	Hardwareabhängige Initialisierung in waveshare_uart_probe()	36
19	Reset des Displays	37

Abbildungsverzeichnis

2.1	Beaglebone Black	2
2.2	Waveshare e-Paper-Display	3
3.1	Aufbau der Hardware	5
3.2	Minicom Ausgabe während des Boot-Vorgangs	5
4.1	Lesen vom Treiber aus dem Userspace	23
4.2	Schreiben des Treibers aus dem Userspace	24
4.3	Zugriff auf das sysfs Attribut	28
5.1	Kernellog Hotplugging	31