

Entwicklung eines Linux Gerätetreibers am Beispiel eines e-Paper Displays

Anna-Lena Marx

17. 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	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	20
4.2.5 Testen der read und write Funktionen	22
4.2.6 waveshare_driver_poll	24
4.3 Das Sys-Filesystem	25
4.4 UART/Serial in Kernel	27
Bibliografie	28

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 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.

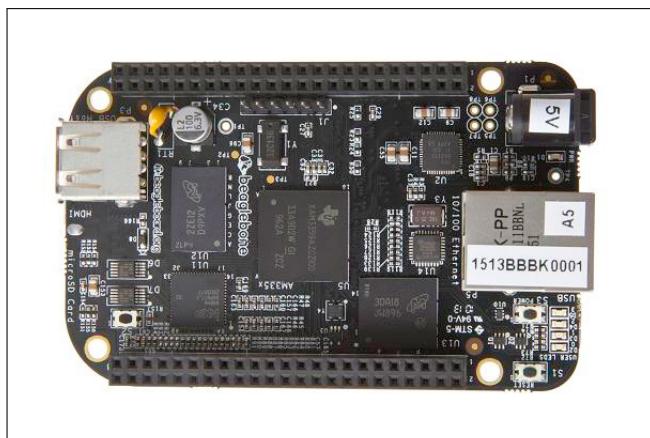
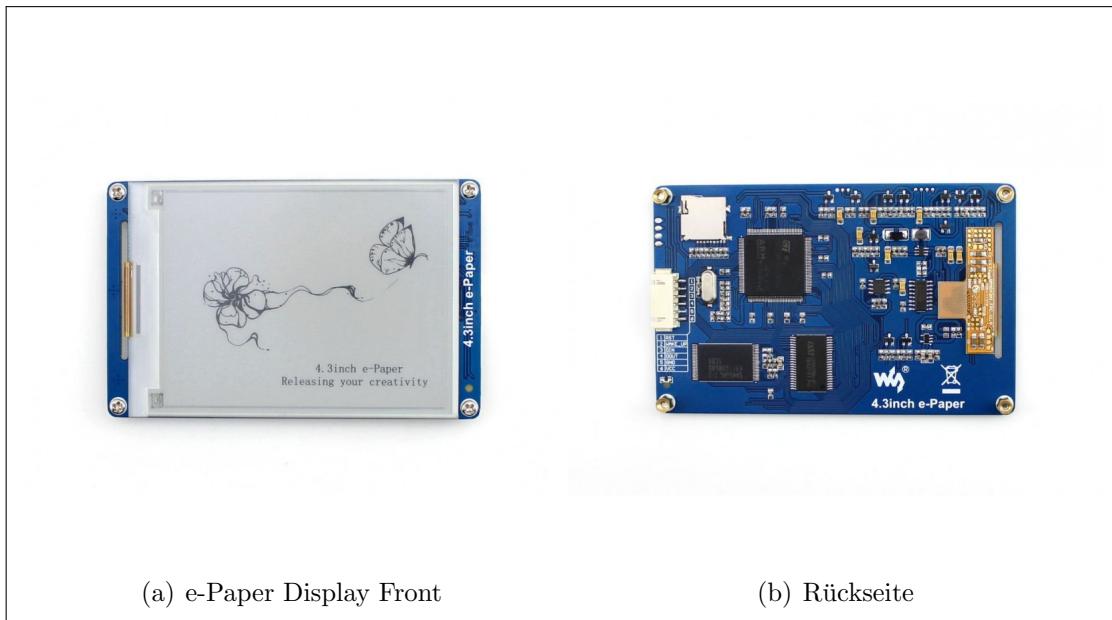


Abbildung 2.1: Beaglebone Black

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.

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.

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 Aufbau und Grundlagen

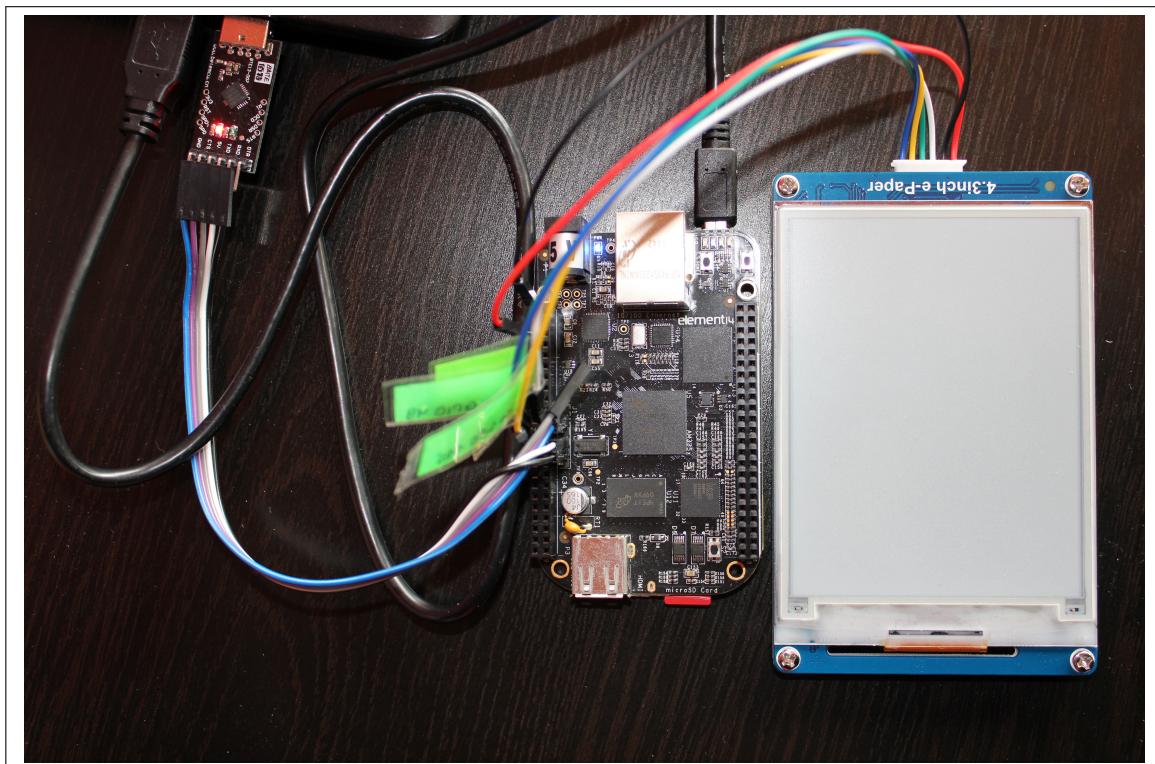


Abbildung 3.1: Aufbau der Hardware

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.whup_m3 ipc: could not get rproc handle
[ 3.664572] omap-sham 53100000:sham: Initialization failed
[ 3.664572] omap-sham 53100000:sham: Initialization failed
[ 3.818148] cpus cpuid: cpuid clock notification entry, entry
[ 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
[12.017965] 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.

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 folgende 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** 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

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

3 Aufbau und Grundlagen

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`.

4 Das Kernelmodul

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.

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 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-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. 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 definiert:

```

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

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 sieht eine solche Fehlerbehandlung wie folgt aus:

```

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

Listing 5: `waveshare_init` Fehlerbehandlung

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

Die `exit`-Funktion `waveshare_exit()` 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 blockierend 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 *driverinstance) {
4
5     if ( ((driverinstance->f_flags&O_ACCMODE) == O_RDWR) ||
6         ((driverinstance->f_flags&O_ACCMODE) == O_WRONLY) ) {
7
8         if (atomic_inc_and_test(&access_counter)) {
9             return 0;
10        }
11
12        atomic_dec(&access_counter);
13        PRINT ("sorry - just one writing instance");
14        return -EBUSY;
15    }
16
17    return 0;
18 }
```

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 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 *driverinstance) {
2
3     if ( ((driverinstance->f_flags&O_ACCMODE) == O_RDWR) ||
4         ((driverinstance->f_flags&O_ACCMODE) == O_WRONLY) ) {
5         atomic_dec(&access_counter);
6     }
7
8     return 0;
9 }
```

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

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.

```

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), max_bytes_to_read+1);
33     not_copied = copy_to_user (buffer, kern_buffer, to_copy);
34
35     // Werden die Daten im funktionsfähigen Treiber dynamisch angepasst,
36     // muss so berechnet werden, wie viel noch zu lesen ist.
37     //atomic_sub ((to_copy - not_copied), &bytes_available);
38     // *offset += to_copy - not_copied;
39
40     return (to_copy - not_copied);
41 }
```

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.

4 Das Kernelmodul

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 *buffer,
9 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), max_bytes_to_write);
24     not_copied = copy_from_user (kern_buffer, buffer, to_copy);
25
26     // Hier sollte eigentlich der Displaypuffer mit den Eingaben aus dem Userspace
27     // verändert werden (und evtl. ein Refresh des Displays stattfinden)
28
29     // Werden die Daten im funktionsfähigen Treiber dynamisch angepasst,
30     // muss so berechnet werden, wie viel noch schreiben ist.
31     //atomic_sub ((to_copy - not_copied), &bytes_to_write);
32     //**offset += (to_copy - not_copied);
33
34     return (to_copy - not_copied);
35 }
```

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. Dadurch lässt `cat` immer wieder den Inhalt des Treibers, bis der Benutzer manuell via STR+C abbricht.

4 Das Kernelmodul

```
cat /dev/waveshare
```

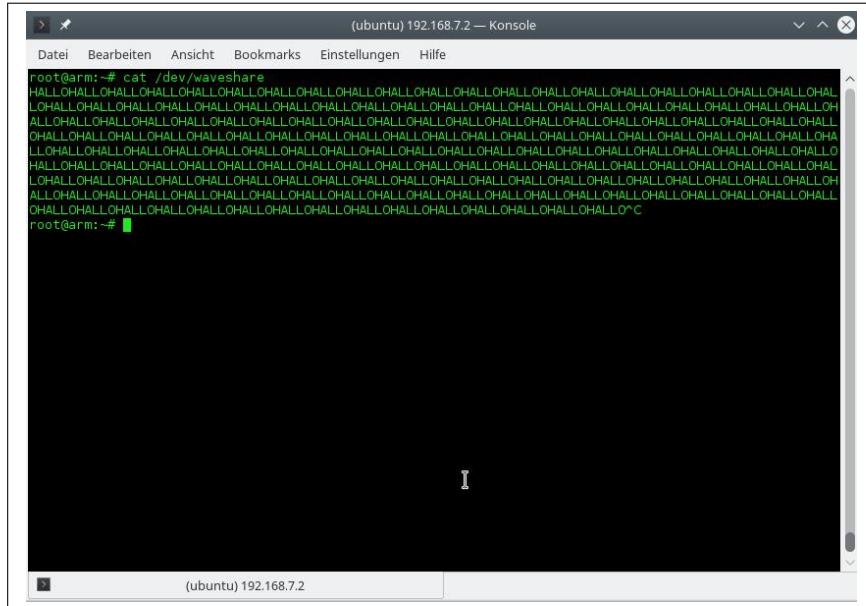


Abbildung 4.1: Lesen vom Treiber aus dem Userspace

Der Befehl `echo` ermöglicht es die Funktion `waveshare_driver_write()` zu triggern.

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

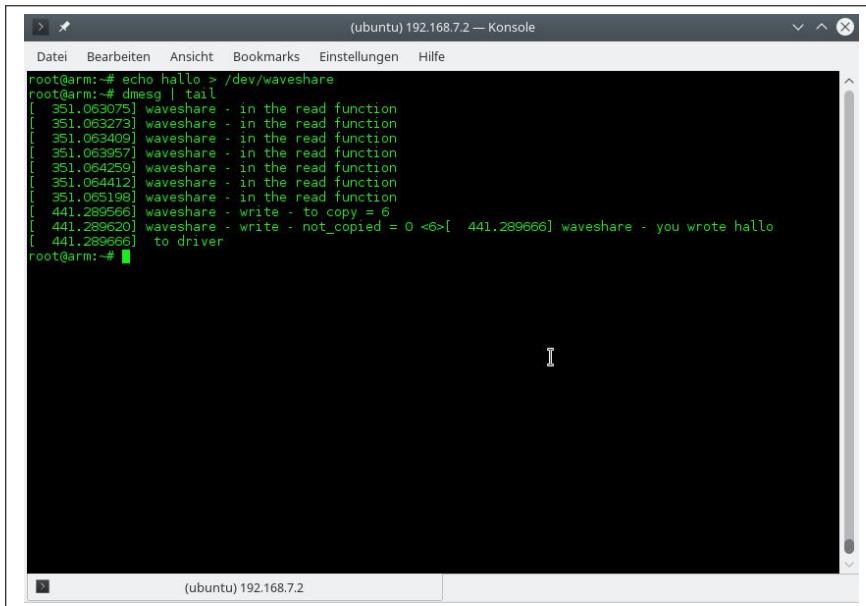


Abbildung 4.2: Schreiben des Treibers aus dem Userspace

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

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.

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 `strcpy` 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 *attr,
2         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 *attr,
2         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;
10 }
```

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 jeweils 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, waveshare_sysfs_write);
```

Listing 14: DEVICE_ATTR

4 Das Kernelmodul

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`.

```
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 Gerätekasse und Attribut erstellen

4.4 UART/Serial in Kernel

Literatur

- [1] *4.3inch e-Paper User Manual.* Techn. Ber. Apr. 2015.
- [2] *Archlinux ARM Beaglebone Black.*
- [3] Shraddha Barke, Hrsg. *Kernel Build.* Okt. 2015.
- [4] *Building Beaglebone Black Kernel.*
- [5] Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman. *Linux Device Drivers -*. Sebastopol: Ö'Reilly Media, Inc.", 2005. ISBN: 9780596555382.
- [6] *Kernel -Serial controller device driver programming.* Free Electrons.
- [7] Greg Kroah-Hartman. *How to Create a sysfs File Correctly.* Juni 2013.
- [8] *Linux Drivers Device Tree Guide.*
- [9] *Low Level Serial API.*
- [10] Alexander Patrakov. *Writing Systemd Files.* Jan. 2011.
- [11] *Serial Drivers.* Free Electrons. F, Dez. 2010.
- [12] 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.
- [13] *Serial over USB on Beaglebone Black.*
- [14] *serial_core.h.* Deep Blue Solutions Ltd., 2000.

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	24
12	waveshare_sysfs_read	26
13	waveshare_sysfs_write	26
14	DEVICE_ATTR	26
15	sysfs Gerätekasse und Attribut erstellen	27

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	23