

Entwicklung eines Linux Gerätetreibers am Beispiel eines e-Paper Displays

Anna-Lena Marx

2. 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	Vorbereitungen zum Coding	5
3.2.1	Kernelquellen	5
3.2.2	Buildsystem	6
4	Das Kernelmodul	8
4.0.1	Kernelspace - eine Vorwarnung	8
4.0.2	Debugging	8
4.1	Das Grund-Modul - <code>__init</code> und <code>__exit</code>	9
4.1.1	Init des Waveshare e-Paper-Treibers	9
4.1.2	Fehlerbehandlung bei Initialisierungen	11
4.1.3	Alles hat ein Ende - Die <code>exit</code> -Funktion	12
4.2	Die Treibereinsprungspunkte	13
	Bibliografie	14

1 Einleitung

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

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

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

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

2 Hardware

2.1 Beaglebone Black

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

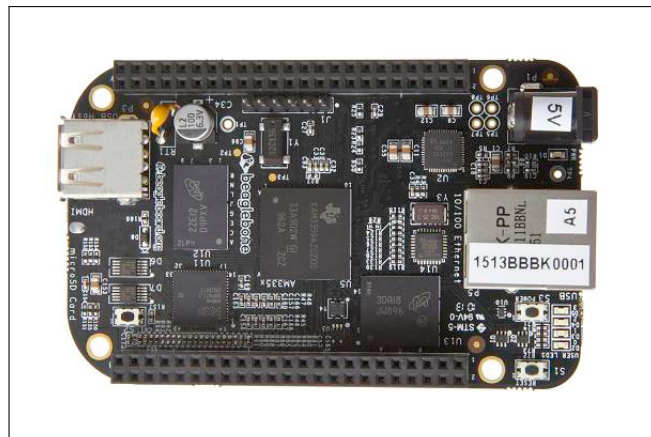


Abbildung 2.1: Beaglebone Black

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.

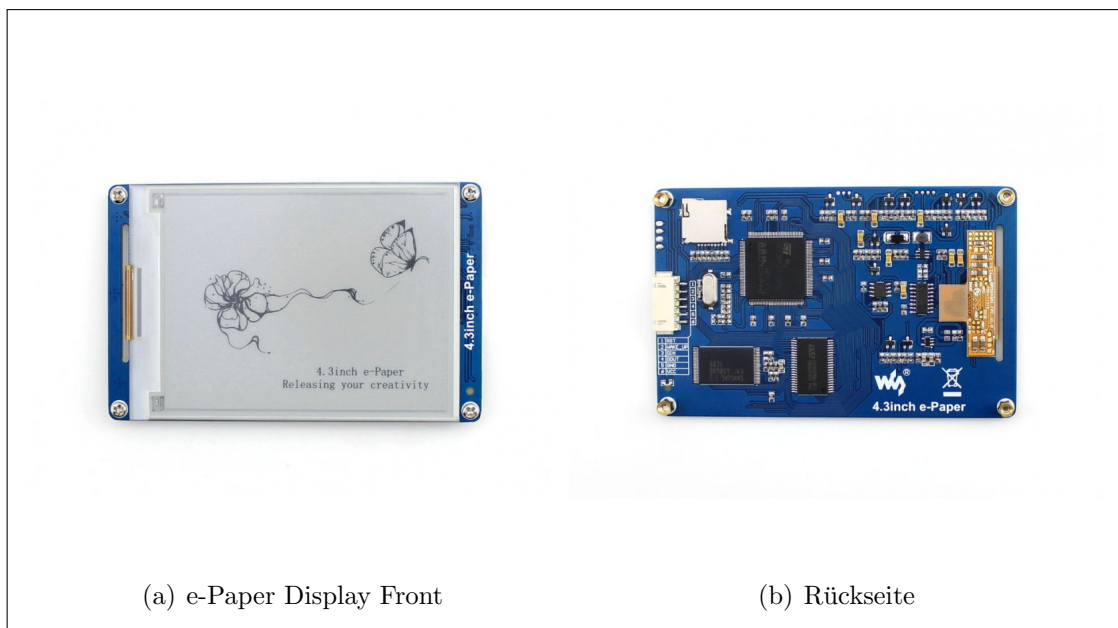


Abbildung 2.2: Waveshare e-Paper-Display

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 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

```

Willkommen zu minicom 2.7
Optionen: I18n
Übersetzt am Sep  6 2015, 19:49:19.
Port /dev/ttyUSB0, 21:36:40

Drücken Sie CTRL-A Z für Hilfe zu speziellen Tasten
8250120 bytes read in 723 ms (10.9 MiB/s)
debug: [/boot/initrd.img-4.1.12-ti-r29] ...
4075682 bytes read in 385 ms (10.6 MiB/s)
debug: [/boot/dtbs/4.1.12-ti-r29/am335x-boneblack.dtb] ...
59295 bytes read in 130 ms (445.3 KiB/s)
debug: [console=tty0 console=ttyO0,115200n8 root=/dev/mmcblk0p1 rootfstype=ext4 rootwait coherent_pool=1M quiet cape_universal=enable] ...
debug: [bootz 0x82000000 0x88000000:3e31ce 0x88000000] ...
kernel image @ 0x82000000 [ 0x00000000 - 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 ...

3.282985] wkup_m3_ipc 44e11324.wkup_m3_ipc: could not get rproc handle
3.664579] omap-sham 53100000.sham: Initialization failed.
3.686916] cpu cpu0: cpu0 clock notifier not ready, retry
3.818148] bone_capemgr bone_capemgr: slot #0: No cape found
3.878137] 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
12.017865] pru-rproc 4a334000.pru0: rproc boot failed
12.265144] remoteproc1: failed to load am335x-pru1-fw
12.296259] remoteproc1: request_firmware failed: -2
12.396816] 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 or finalize resolvconf[ OK ]

```

Abbildung 3.1: Minicom Ausgabe während des Boot-Vorgangs

3.2 Vorbereitungen zum Coding

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

3.2.1 Kernelquellen

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

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

```

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

```

Listing 1: Laden und Kompilieren der Kernelquellen

3.2.2 Buildsystem

Kernelmodule können entweder als fester Bestandteil des Kernels (In-Tree) oder als dynamisch ladbares Modul außerhalb des Kernels kompiliert werden. Im ersten Fall muss 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 Objektdatetei (`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 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

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-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 Symbollisten auflöst und noch nicht alle Hardwareplattformen werden unterstützt. Im Rahmen dieser Arbeit wird daher auf die im Kernel alt hergebrachte Art der Fehlersuche mit `printk` gesetzt, das mit `printf` in der C-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 einfügen 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 hardware-spezifische 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 Rückgabotyp `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 erste Parameter eine Referenz auf eine Struktur vom Typ `dev_t`, eine Geräteummer, 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 Geräteummer assoziiert wird. 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

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` (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
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()` wird beim Entladen des Treibers vom Kernel 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 Treibereinsprungpunkte

List of Listings

1	Laden und Kompilieren der Kernelquellen	6
2	Makefile	7
3	Debug-Macro	9
4	waveshare_init	11
5	waveshare_init Fehlerbehandlung	12
6	waveshare_exit	13

Abbildungsverzeichnis

2.1	Beaglebone Black	2
2.2	Waveshare e-Paper-Display	3
3.1	Minicom Ausgabe während des Boot-Vorgangs	5