Geräte Treiber unter Unix/Linux

Martin Schäf

7. Oktober 2004

1 Einleitung

Die Programmierung von Geräte Treibern unter Linux ist ein sehr populäres Thema, da für viele Geräte keine, oder nur sehr primitive Treiber von den Herstellern bereitgestellt werden, der Kernel jedoch einige Möglichkeiten bietet, selbst Treiber zu schreiben. Der Kernel stellt einen eigenen Befehlssatz bereit, der für die Treiberprogrammierung verwendet werden kann. Welche Schritte nötig sind um einen Treiber zu schreiben, und welche Arten von Treibern unterschieden werden müssen wird in den folgenden Kapiteln näher beschrieben.

2 Geräte Treiber als Module

Die gängigste Methode einen Treiber zu entwickeln ist, ihn als Modul zu kompilieren, und zum Kernel zu linken. Ein Modul kann mit jedem Linux C-Compiler erstellt werden. Module unterscheiden sich von sonstigen Anwendungen dadurch, dass sie keine main Routine haben, sondern einen Satz von Funktionen bereitstellen, die von anderen Programmen genutzt werden können. Module können unter Linux zum laufenden Betriebssystem hinzu gelinkt, oder wieder entfernt werden. Der Vorteil dieser Technik besteht darin, dass Module sehr leicht getestet werden können, ohne einen Neustart zu fordern. Dieser Vorteil ist gerade in der Treiberprogrammierung entscheidend, da ständige Neustarts die Entwicklungszeit erheblich erhöhen würden. Treiber können auch in den Kernel einkompiliert werden, wie beispielsweise Festplatten- oder Monitortreiber beim Systemstart. Da die Entwicklung von Treibern als Modul wesentlich weiter verbreitet ist, wird im Folgenden nur diese Methode genauer erklärt.

2.1 Module im Kernel

Ein Modul hat, sobald es zum Kernel gelinkt wurde zugriff auf den Kernel Speicher. Aus diesem Grund können in Modulen keine STL Funktionen

verwendet werden. Der Entwickler muss auf den Befehlssatz des Kernels zurückgreifen, wie er in kernel.h definiert ist.

2.2 Module zum Kernel linken und entfernen

Wird ein Treibermodul zum Kernel gelinkt, registriert es die von ihm bereitgestellten Methoden beim Kernel, damit dieser sie in Zukunft nutzen kann, falls Anfragen an das entsprechende Gerät kommen. Wird das Treibermodul wieder aus dem Kernel entfernt, werden auch die Methoden deinitialisiert. Dies geschieht praktischerweise automatisch, also muss sich der Treiber Programmierer nicht darum kümmern, und kann auf diesem Weg auch keine Speicherfehler verursachen.

3 Zuordnung von Treiber zu Gerät

Ein Treiber ist selbst dafür zuständig sein Gerät zu finden. Es gibt verschiedene mehr oder weniger effiziente Methoden die Hardware Ports nach bestimmten Geräten abzusuchen, auf die hier aber nicht näher eingegangen wird. Hat der Treiber sein Gerät gefunden und seine Funktionalität beim Kernel angemeldet, legt er, sofern es kein Netzwerk Treiber ist, eine Datei im /dev/ Verzeichnis an, über die mit dem Gerät kommuniziert werden kann, wie mit einer gewöhnlichen Datei. Netzwerk Treiber haben eine gesonderte Stellung, da es wenig sinnvoll ist eine Netzwerkverbindung zum lesen zu öffnen, oder einen seek Befehl darauf anzuwenden. Netzwerk Treiber werden im Detail später besprochen.

3.1 Major und Minor Nummern

Damit eine eindeutige Zuordnung von Treiber zu Gerät möglich ist werden vom Kernel Major und Minor Nummern vergeben. Generell gilt, dass ein Treiber mehrere Geräte verwalten kann, aber ein Gerät höchstens einen Treiber haben kann.

3.1.1 Die Major Nummer

Die Major Nummer ordnet einem Gerät seinen Treiber zu. Jeder Treiber hat seine eigene eindeutige Major Nummer.

3.1.2 Die Minor Nummer

Die Minor Nummer hilft dem Treiber zwischen den ihm zugeordneten Geräten zu unterscheiden. Jedes Gerät muss eine eindeutige Minor Nummer bzgl. seiner Major Nummer haben. Verwaltet der Treiber nur ein Gerät, wird die Minor Nummer ignoriert.

4 Arten von Treibern

Man unterscheidet verschiedene Arten von Geräte Treibern.

- Zeichen Treiber
- Block Treiber
- USB und FireWire Treiber
- Netzwerk Treiber

Block-, Zeichen- und Netzwerk Treiber werden im Folgenden näher beschrieben. Block Geräte sind z.B. Festplatten, Zeichen Geräte sind beispielsweise Maus oder Keyboard. Da sowohl Block, als auch Zeichen Geräte an den USB- oder FireWire Port angeschlossen werden können muss die Klasse der USB-, bzw FireWire Geräte Treiber separat behandelt werden, worauf hier jedoch nicht näher eingegangen wird. Grundsätzlich ist es für diese universellen Schnittstellen wichtig, zuerst den Typ des Gerätes zu erkennen, und dann entsprechend einen Zeichen-, Block- oder Netzwerk Treiber zu verwenden.

4.1 Zeichen Geräte Treiber

Aus Zeichen Geräte Treibern wird in den meisten Fällen zeichenweise gelesen. Random Access ist zwar möglich, wird aber nur selten benutzt. Zeichen Geräte sind vornehmlich Geräte, bei denen die ein- und ausgehenden Daten immer in fester Reihenfolge abgearbeitet werden, wie beispielsweise ein Keyboard.

4.1.1 Zeichen Geräte Treiber erstellen und laden

Ein Modul kann einen Zeichentreiber mit folgender Funktion anmelden:

```
int register_chrdev(unsigned int major, const char *name,
                    struct file_operations *fops);
```

mit Rückgabewert 0 oder einem negativen Fehlercode, falls das Anmelden fehlschlägt und den Parametern:

unsigned int major	Die Major Nummer des Treibers
const char *name	Der Gerätename
struct file_operations *fops	In der Struktur file_operantions werden Pointer auf alle Funktionen hinterlegt, auf die der Treiber reagieren soll.

4.1.2 Die Struktur file_operations

Die genaue Definition der Struktur file_operations ist in *linux/fs.h* definiert. Im Folgenden wird auf die wichtigsten Elemente der Struktur genauer eingegangen.

int (*open) (struct inode *, struct file *);	Funktion zum öffnen des Geräts.
ssize_t (*read) (struct file *, char *, size_t, loff_t *);	Muss vom Treiber implementiert werden, damit aus dem Gerät gelesen werden kann.
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);	Ermöglicht Programmen gerätespezifischen Zugriff
ssize_t (*write) (struct file *, const char *, size_t, loff_t * int);	Sendet Daten an das Gerät
loff_t (*llseek) (struct file *, loff_t, int);	Verändert die Leseposition

4.2 Block Geräte Treiber

Block Geräte unterscheiden sich im wesentlichen von Zeichen Geräten dadurch, dass der Zugriff immer Random Access ist. Die Daten werden blockweise gelesen. Desweiteren können nur Block Geräte ein Dateisystem aufnehmen und gemountet werden.

4.2.1 Block Treiber erstellen und laden

Block Geräte werden ähnlich wie Zeichen Geräte mit einer Funktion initialisiert.

```
int register_blkdev(unsigned int major, const char *name,
struct block_device_operations *bdops);
```

unsigned int major	Die Major Nummer des Treibers
const char *name	Der Gerätename
struct block_device_operations *bdops	Ein Pointer auf eine Struktur, die Funktionspointer auf die Treiberfunktionen enthält

4.2.2 Die Struktur block_device_operations

Die Struktur block_device_operations unterscheidet sich kaum von der für Zeichen Treiber verwendeten Struktur file_operations. Die meisten Methoden, wie open, close oder ioctl sind genauso getypt. Nur der Zugriff auf die Daten des Geräts erfolgt über eigene Methoden.

Schreiben und Lesen übernimmt eine strategy, oder request Methode. Da Daten nur blockweise gelesen werden, fallen die read und write Methoden wie sie für Zeichen Geräte verwendet werden weg. Die genaue Funktionalität der strategy Methode kann in [1] nachgelesen werden.

4.3 Netzwerk Geräte

Netzwerk Geräte werden, im Gegensatz zu anderen Geräten nicht im /dev/ Verzeichnis durch eine Datei repräsentiert, sondern in einer globalen Liste von Netzwerk Geräten hinterlegt. Ein Grund dafür ist, dass der Benutzer nicht direkt auf das Gerät wie beispielsweise auf eine Festplatte zugreifen soll.

Netzwerk Geräte unterscheiden sich im Wesentlichen von allen anderen Geräten dadurch, dass der Zugriff asynchron ist. Eine Netzwerk Karte muss immer bereit sein Pakete zu empfangen. Eine read operation würde in einem Netzwerk Treiber folglich nicht viel Sinn machen.

5 Arbeitsweise eines Netzwerk Geräte Treibers

Einen Netzwerk Treiber unter Linux zu schreiben ist weitaus einfacher als man denken würde, da Linux ein sehr weit entwickeltes Netzwerk Subsystem besitzt, dass sämtliche Protokollfragen und das Verteilen der Pakete an die Prozesse übernimmt. Der Netzwerk Treiber selbst muss sich nur um die physikalische Übermittlung und mögliche Timeouts, sowie das Verhindern von Datenverlust kümmern. Ein Netzwerk Treiber besitzt Warteschlangen, die Pakete vom Netzwerk Subsytem annehmen, oder aus dem Netzwerk entgegen nehmen, und diese entweder weiter senden, oder auf deren Abholung warten. Wichtig ist nun, dass der Treiber feststellt, wann eine Warteschlange voll ist, und die entsprechenden Maßnahmen einleitet.

5.1 Anmelden beim Kernel

Wie auch bei den bereits erwähnten Gerätetypen gibt es für Netzwerk Geräte eine Struktur. Die Struktur net_device. Ein Netzwerk Treiber kann über diese Struktur mit der Funktion

```
int register_netdev(struct net_device* netd);
```

beim Kernel angemeldet werden.

5.1.1 Die net_device Struktur

Die net_device Struktur ist sehr umfangreich und kann in der Datei linux/netdevice.h nachgelesen werden. Für die Programmierung eines einfachen Treibers sind nur wenige Felder und Funktionen wichtig, die im Folgenden beschrieben werden.

Felder der net_device Struktur	
char name [IFNAMSIZ]	Name des Geräts
unsigned long base_addr	Die I/O-Basisadresse der Netzwerk-Schnittstelle
unsigned char irq	Die IRQ des Geräts
unsigned char dma	Der vom Gerät allozierte DMA-Kanal
unsigned long state	Der Gerätezustand
struct net_device *next	Ein Zeiger auf das nächste Gerät in der globalen verketteten Liste
Methoden der net_device Struktur	
int (*open)(struct net_device *dev)	Öffnen des Geräts
int (*stop)(struct net_device *dev)	Anhalten des Geräts
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev)	Fordert die Übertragung eines Paketes an
void (*tx_timeout)(struct net_device *dev)	Methode zum behandeln von Timeouts

5.1.2 Die sk_buff Struktur

Die Kommunikation mit dem Netzwerk Subsystem findet über Soket Buffer statt. Die Daten, die der Netzwerk Treiber erhält, und die der Treiber dem Netzwerk Subsystem bereitstellt müssen als sk_buff Struktur überreicht werden. Die Details dieser Struktur sind in linux/skbuff.h nachzulesen. Im Folgenden werden die wichtigsten Felder dieser Struktur erklärt.

struct net_device *rx_dev;, struct net_device *dev	Die Geräte, die diesen Puffer senden oder empfangen.
unsigned char *head;, unsigned char *data;, unsigned char *tail;, unsigned char *end;	Mit diesen Zeigern werden die Daten im Paket adressiert
unsigned long len	Länge der Daten mit head und tail
unsigned char ip_summed;	Prüfsumme für eingehende Pakete
unsigned char pkt_type	Flag zum Kennzeichnen ausgehender Pakete

5.1.3 Senden

Die sk_buff Struktur enthält die Daten bereits in der Form wie sie über das Netzwerk gesendet werden sollen. Der Programmierer muss sich folglich nicht um Protokoll Fragen kümmern.

Wenn der Kernel ein Paket senden will ruft er die Funktion

```
int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
```

auf, die das Paket in die Warteschlange für ausgehende Pakete stellt. Die Verwaltung der Warteschlange gehört zu den wichtigsten Aufgaben des Netzwerk Treibers. Der Treiber muss garantieren, dass keine Daten verloren gehen. So ist die Funktion hard_start_xmit immer gegen nebenläufige Aufrufe geschützt. Ist die Warteschlange des Treibers voll, muss dem Kernel mitgeteilt werden, dass keine Pakete mehr angenommen werden können. Der Kernel stoppt den sendenden Prozess, bis der Treiber sich mittels Interrupt meldet und die Warteschlange wieder frei gibt.

5.1.4 Empfangen

Das Empfangen von Paketen ist etwas aufwendiger als das Senden, da ein Interrupt Handler registriert werden muss. Es gibt zwei generelle Methoden, wie ein Treiber Daten empfangen kann. Entweder er fragt in regelmässigen Abständen nach, ob Daten für ihn vorliegen, oder er meldet einen Interrupt Handler an. Der Interupt Handler hat den Vorteil, dass er höheren Datendurchsatz ermöglicht, und gleichzeitig weniger Prozessorlast verursacht.

Der Netzwerk Treiber lässt also die Funktion für das Empfangen von Paketen durch einen Interrupt aufrufen. Seine Aufgabe ist es dann ausreichend Speicher für das Paket zu allozieren, und es dem Netzwerk Subsystem bereitzustellen.

5.2 Interrupt Handler

Ein Treiber der eine Interrupt Leitung benutzen will muss dies beim Kernel mit der Funktion

```
int request_irq(unsigned int irq,
                void (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                const char *dev_name,
                void *dev_id);
```

beantragen. Die Funktion gibt 0 zurück, falls die Leitung benutzt werden kann, und einen negativen Fehlercode, falls die Leitung schon belegt ist.

unsigned int irq	Die Nummer der bewünschten IRQ Leitung
void (*handler)(int, void *, struct pt_regs *)	Die Funktion, die aufgerufen wird, sobald der Interrupt ausgelöst wird.
unsigned long flags	Der Interrupt kann mittels flags näher spezifiziert werden.
const char *dev_name	Der Name des Geräts, dass den Interrupt verursacht hat
void *dev_id	Wird nur bei gemeinsamer Interrupt benutzung verwendet.

5.3 Fehlerbehandlung durch Timeouts

Der Programmierer kann für seinen Treiber eine Timeout Funktion definieren, wie sie in der net_device Struktur beschrieben ist. Die Funktion wird von einem Kernel Timer aufgerufen, falls das Netzwerk Gerät für einen bestimmten Zeitraum nicht reagiert. Dieser Zeitraum wird vom Programmierer festgelegt, und sollte etwas grösser sein, als die Zeit, die zum Übertragen eines Paketes benötigt wird. Die Timeout Funktion muss den Fehler suchen, beheben und das Gerät wieder freigeben. Die häufigste Fehlerursache ist ein verlorener Interrupt, der von der Timeout Funktion dann manuell ausgelöst wird.

Literatur

[1] Alessandro Rubini und Jonathan Corbet: "Linux Device Drivers" 2. Auflage, 2001

[2] Alan Cox: "Network Buffers And Memory Management" In Linux Journal, issue 29, September 1996