Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 17

Kern-Technik

Der unüberschaubar große Zoo von Netzwerkgeräten spiegelt sich im Kernel in der Anzahl zugehöriger Treiber wider. Wie sie intern funktionieren, zeigt diese Folge der Kern-Technik. Ein virtuelles Netzwerk-Device dient als Einstieg für eigene Experimente. Eva-Katharina Kunst, Jürgen Quade

Wer den Umgang mit »ifconfig« und »route« gewohnt ist, findet sich in der Welt der Netzwerktreiber schnell zurecht, denn ihre Grundstruktur ist übersichtlich. Erst die Komplexität der Hardwarezugriffe lässt die Treiber zu umfangreichen Modulen anwachsen.

Real und virtuell

Die meisten Netzwerktreiber implementieren Zugriffe auf reale Kommunikationshardware. Doch nicht selten sind auch virtuelle Kommunikationsgeräte im Einsatz, zum Beispiel das Loopback- oder das Dummy-Device.

Egal ob virtuell oder real, ein Netzwerktreiber besteht im Wesentlichen aus acht Funktionen: der Treiber-Initialisierung und -Deinitialisierung, der Geräte-Initialisierung und -Deinitialisierung, dem Öffnen und Stoppen des Netzwerkinterface, dem Senden und schließlich dem Empfang eines Pakets. Die ersten vier Funktionen integrieren den Treiber in die Kernelinfrastruktur und initialisieren die Hardware, siehe [1]. Die vier übrigen Funktionen (Open, Stopp, Senden und Empfangen) stellen den Kern des Treibers dar. Sie implementieren das so genannte Netzwerk-Interface, also den Teil, über den Anwendungen respektive das Netzwerk-Subsystem auf die Hardware zugreifen.

Aus Sicht des Programmierers ist das Netzwerk-Interface ein zu erzeugendes Objekt. Meist erledigt er das bei der Treiber-Initialisierung. Die Funktion »alloc_netdev()« erzeugt und initialisiert ein solches Objekt vom Typ »struct net_device«. Dabei gilt es, eine ganze Reihe von Feldern sinnvoll zu belegen. Da dies schnell unübersichtlich wird, schlagen die Kernelentwickler vor, die Initialisierung in eine eigene Setup-Funktion auszulagern. Ihre Adresse übergibt man der Funktion »alloc_netdev()« als Parameter. Das Netzwerk-Subsystem ruft die Setup-Funktion auf, nachdem es Speicher für das Netzwerk-Interface reserviert hat.

Zur Initialisierung des Interface gibt es aber noch mehr Hilfe. So existieren fertige Funktionen, die zu einem spezifischen Übertragungsmedium gehörige Felder sinnvoll belegen. Für Ethernet beispielsweise heißt diese Funktion »ether_setup()«, für FDDI »fddi_setup()«. Der Programmierer ruft in seiner Setup-Funktion zuerst »ether_setup()« auf und belegt dann die übrigen Felder der Struktur »struct net_dev«. Insbesondere trägt er hier die Adressen der grundlegenden Treiberroutinen (Open, Stopp und Senden) ein.

Netzwerk-Interfaces

Die Funktion »alloc_netdev()« erwartet neben der Adresse der Setup-Funktion einen Formatstring, der den Namen des Netzwerk-Interface festlegt. Da mehrere gleiche oder auch unterschiedliche Treiber gleichnamige Interfaces anlegen können, unterscheiden Anwender und Kernel sie anhand einer laufende Nummer, zum Beispiel »eth0« und »eth1«. Der Format-String repräsentiert diese Nummer durch ein »%d«, für ein Ethernet-Interface also durch »eth%d«.

Die Funktion »alloc_netdev()« hat noch einen letzten Parameter: die Anzahl Bytes, die das Netzwerk-Subsystem zusätzlich am Ende der »struct net_device« reserviert, siehe Abbildung 1. Dieser Bereich steht für treiberspezifische und private Daten zur Verfügung.

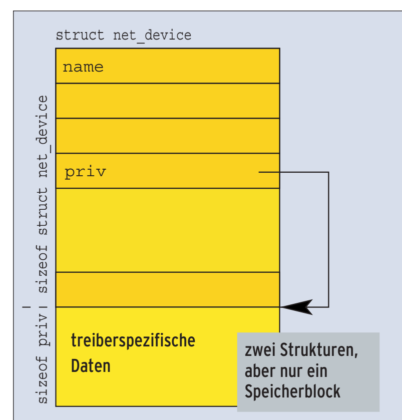

Abbildung 1: Platz für treiberspezifische Daten wird direkt im Anschluss an die Struktur »struct net_device« reserviert.

Für die beiden Speicherbereiche (»struct net_device« und für die privaten Daten) genügt es, einmal »kmalloc()« aufzurufen. Genauso gibt sie ein einziges »kfree()« wieder frei. Dabei muss sich niemand darum kümmern, ob der private Speicherbereich dynamisch oder statisch reserviert ist. Die so vorbereitete »struct net_device« übergibt man der Funktion »register_netdev()«. Damit meldet der Treiber während der Initialisierung das Netzwerk-Interface beim Subsystem an.

An- und Abmelden

Bei der Treiber-Deinitialisierung macht »unregister_netdev()« diese Registrierung wieder rückgängig. Die Funktion »free_netdev()« gibt das Interface-Objekt »struct net_device« frei.

Der Treiber eines echten Netzwerkadapters muss natürlich auch das Gerät selbst (de-)initialisieren. Wie das geht, hängt von der Hardware ab. Handelt es sich etwa um eine PCI-Karte, meldet sich der Treiber nicht nur beim Netzwerk-Subsystem, sondern auch beim PCI-Subsystem an, siehe [2]. Das wiederum ruft die zugehörige (De-)Initialisierungsfunktionen im Treiber auf.

Bei alter ISA-Hardware kann auch das Netzwerk-Subsystem die Geräte-Initialisierung auslösen. In diesem Fall ist im Element »init« der »struct net_device« die Funktionsadresse der Geräte-Initialisierung und im Element »uninit« jene der Deinitialisierungs-Routine abzulegen. Den zugehörigen Code führt das Netzwerk-Subsystem nach Aufruf der Setup-Funktion aus.

Da der Empfang von Paketen bei realer Hardware fast ausschließlich per Interrupt signalisiert wird, braucht jeder Treiber eine Interrupt-Service-Routine. Ist der Treiber beim Netzwerk-Subsystem angemeldet und die Hardware initialisiert, steht das neue Interface bereit. Es wird nicht über eine Major- und Minornummer identifiziert, sondern über den Namen, zum Beispiel »eth0«.

Anwendungen benutzen die Treiberfunktionen nicht direkt, sondern über Wrapper-Funktionen des Kernels. Die Methoden Open und Stopp ruft der Kernel auf, wenn der Anwender das Kommando »ifconfig up« respektive »ifconfig down« einsetzt, siehe Abbildung 2. Ist der Treiber dazu bereit, vom Netzwerk-Subsystem Pakete zu empfangen und zu verschicken, ruft er die Funktion »netif_start_queue()« auf. Diesen Aufruf findet man meist in den Open-Funktionen von Treibern. Bei Erfolg geben solche Funktionen »0« zurück, sonst einen negativen Fehlercode.

Die Stopp-Methode deaktiviert das Netzwerk-Interface. Der Aufruf von »netif_stop_queue()« sorgt dafür, dass der Kernel dem zugehörigen Interface nicht länger Aufträge zum Verschicken oder Empfangen erteilt.

Datentransfer

Die Hauptfunktionen des Treibers bestehen im Senden und Empfangen der Datenpakete. Die Schnittstelle zwischen Treiber und Netzwerk-Subsystem besteht im Wesentlichen aus den Socket-Puffern »struct sk_buff«, deklariert in »linux/skbuff.h«. Über diese Datenstruktur haben die Instanzen Zugriff auf Nutzdaten und Verwaltungsinformationen. Falls das Netzwerk-Subsystem ein Paket senden will, ruft es die in der »struct net_device« angegebene Sendefunktion auf und übergibt ihr einen Socket-Puffer.

Im Normalfall entnimmt der Treiber das dort abgelegte Paket und kopiert es ohne weitere Modifikation in die Hardware. Netzwerkadapter besitzen dafür eigenen Speicher (Sende- und Empfangspuffer). Die Sendefunktion quittiert die erfolgreiche Übergabe des Pakets an die Hardware durch den Rückgabewert »0«. Konnte das Paket nicht übergeben werden, gibt die Funktion »1« zurück.

Zeitstempel für Timeouts

Falls nach dem Kopieren kein Sendepuffer mehr frei ist, stoppt man den Versand mit »netif_stop_queue()«. Wird ein Sendepuffer frei, aktiviert »netif_start_queue()« den Versand wieder. Nach dem Kopieren der Daten speichert das Subsystem die aktuelle Zeit (im Kernelmaß Jiffies; Listing 1, Zeile 38). Diesen Zeitstempel nutzt es zum Beispiel, um Time-

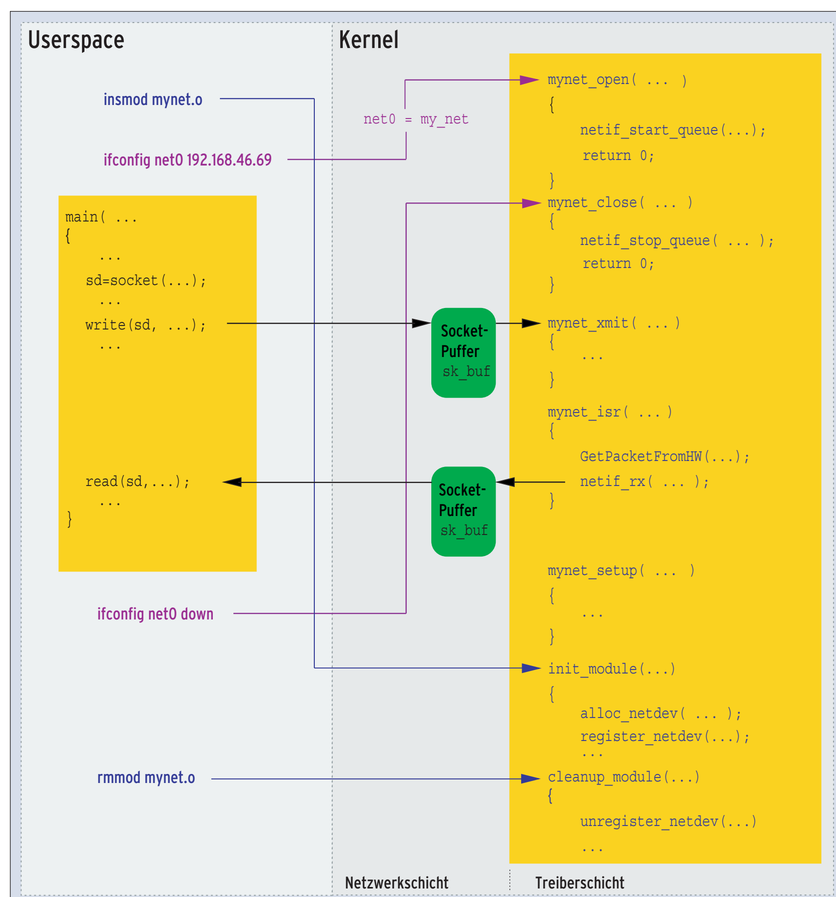

Abbildung 2: Funktionen und Befehle im Userspace rufen ihre Gegenstücke im Kernel auf.

outs zu überwachen, siehe Kasten „Timeout“. Sobald die Hardware signalisiert, dass sie das Paket verschickt hat, aktualisiert der Treiber die Sendestatistik (Listing 1, Zeilen 39 und 40) und gibt den Socketpuffer frei (Listing 1, Zeile 37). Bei älteren Karten kann man leider nur spekulieren, dass mit der Übergabe des Pakets an die Hardware der Versand erfolgreich war. Dabei laufen in der Sendefunktion die Aktualisierung der Statistik und die Socket-Freigabe ab.

Rückmeldung per Interrupt

Moderne Karten melden das Ergebnis per Interrupt. In diesem Fall protokolliert die Interrupt-Service-Routine (ISR) die statistischen Daten, wobei sie bei fehlgeschlagenem Versand statt der Sendestatistik die Fehlerstatistik aktualisiert.

Empfängt der Netzwerkadapter ein Paket, löst er normalerweise auch einen Interrupt aus. In diesem Fall muss der Treiber einen Socket-Puffer erzeugen, also »dev_alloc_skb()« oder »alloc_skb()« mit dem Parameter »gfp_mask« als »GFP_ATOMIC«.

Die Hardware kopiert die Daten in den Socket-Puffer und passt dessen Zeiger auf die Daten an. Im Socket-Buffer muss außerdem stehen, um welchen Protokolltyp es sich handelt und über welches Gerät die Daten entgegengenommen wurden. Gängige Ethernet-Protokolltypen sind beispielsweise ARP »ETH_P_ARP« oder IP »ETH_P_IP«. Den Protokolltyp bestimmt bei Ethernet-Adaptern die Funktion »eth_type_trans()«, deren Rückgabewert der Treiber ins Protokollfeld des Socket-Puffers einträgt, Listing 1, Zeile 23.

Das empfangene und im Socket-Puffer aufbereitete Paket übergibt man dem Netzwerk-Subsystem mit der Funktion »netif_rx()«, die das Paket im Prozess-Kontext weiter verarbeitet. Auch die Freigabe des Socket-Puffers erfolgt durch das Subsystem, nicht durch den Treiber.

Timeout

Der Treiber kann das Verschicken von Daten über das Netzwerk-Interface zeitlich überwachen lassen. Dazu muss er in die »struct net_device« die Adresse einer Watchdog-Funktion und die Zeit eintragen, nach der die Watchdog-Funktion aufgerufen werden soll:

```
dev->tx_timeout = my_net_watchdog;
dev->watchdog_timeo = 5 * HZ;
```

Die Funktion »void my_net_watchdog(struct net_device *dev)« führt der Kernel aus, falls der Treiber nach dem letzten Senden die Funktion »netif_stop_queue()« aufgerufen hat und seither eine Zeit von »watchdog_timeo« vergangen ist.

Bleibt nur noch die Statistik zu aktualisieren (Listing 1, Zeilen 24 und 25) und den Empfangszeitstempel abzulegen (Zeile 26). Für die Statistik besitzt der Kernel eine eigene Datenstruktur. Der Anwender ruft die hier gesammelten Informationen etwa durch den Aufruf von »ifconfig« ab. Die Ausgabe der Informationen ist standardisiert, sodass der Entwickler in der Methode »get_stats()« nur die Adresse der Datenstruktur zurückgeben muss (Zeile 58).

Listing 1 zeigt den Code für ein Netzwerk-Nulldevice. Es wirft mangels realer Hardware alle zu verschickenden Pakete einfach weg. Es besitzt weder eine Geräte-Initialisierung, noch -Deinitialisierung. Die vollständige Version auf dem Linux-Magazin-Server [3] enthält eine Empfangsfunktion als Interrupt-Service-Routine, die über die Präprozessor-Direktive »#if 0« auskommentiert ist, denn ohne Hardware gibt es auch keine Interrupts. Der Treiber lässt sich mit dem Makefile von [3] übersetzen und mit »insmod net.ko« laden.

Das implementierte Netzwerk-Interface trägt den Namen »net0«. Mit »ifconfig net0 10.10.10.10« weist man ihm eine Beispiel-IP-Adresse zu. Es bringt allerdings nichts, vom eigenen Rechner aus auf die Adresse »10.10.10.10« zuzugreifen, da der Kernel lokale Zugriffe direkt

Listing 1: Null-Device »net.c«

```
01 #include <linux/fs.h>
02 #include <linux/module.h>
03 #include <linux/netdevice.h>
04 #include <linux/init.h>
05
06 static struct net_device *my_net;
07 ...
23    skb->protocol = eth_type_trans(skb, dev);
24    stats->rx_packets++;
25    stats->rx_bytes += packet_length;
26    dev->last_rx = jiffies;
27 ...
33 static int my_net_send(struct sk_buff *skb,
   struct net_device *dev)
34 {
35    struct net_device_stats *stats = dev->priv;
   //netdev_priv(dev);
36
37    kfree_skb( skb );
38    dev->trans_start = jiffies;
39    stats->tx_bytes += skb->len;
40    stats->tx_packets++;
41    return 0;
42 }
43
44 static int my_net_open(struct net_device *dev)
45 {
46    netif_start_queue(dev);
47    return 0;
48 }
49
50 static int my_net_close(struct net_device *dev)
51 {
51    netif_stop_queue(dev);
53    return 0;
54 }
55
56 static struct net_device_stats
   *my_net_get_stats( struct net_device *dev )
57 {
58          return dev->priv;
59 }
60
61 static void __init my_net_setup( struct
   net_device *dev )
62 {
63    ether_setup(dev);
64    dev->open = my_net_open;
65    dev->stop = my_net_close;
66    dev->hard_start_xmit= my_net_send;
67    dev->get_stats = my_net_get_stats;
68    dev->flags |= IFF_NOARP;
69 }
70
71 static int __init net_init(void)
72 {
73    if( (my_net=alloc_netdev(sizeof(struct
   net_device_stats),
74             "net%d",my_net_setup))==NULL )
75       return -ENOMEM;
76    return register_netdev(my_net);
77 }
78
79 static void __exit net_exit(void)
80 {
81    unregister_netdev(my_net);
82    free_netdev(my_net);
83 }
84
85 module_init( net_init );
86 module_exit( net_exit );
87 MODULE_LICENSE("GPL");
```

abfängt und gar nicht an den Treiber weiterreicht. Ein Ping auf die Adresse »10.10.10.1« ruft wie gewünscht den Treiber auf. Dass er aktiv ist, lässt sich nun durch Aufruf der Netzstatistik überprüfen: »ifconfig net0«.

Vorschau

Netzwerktreiber können recht komplex werden. Weil sich in Kernel 2.6 aber wenig geändert hat, sind ältere Dokumentationen noch aktuell. Wer tiefer einsteigen will, sollte sich zum Beispiel die passenden Abschnitte in den Büchern **[4]**, **[5]** und **[6]** anschauen und den Skeleton-Treiber »isa-skeleton.c« im Kernel-Quellcode (siehe **[7]**) studieren.

In anderen Kernelbereichen hat sich wesentlich mehr geändert. So bringt Version 2.6 Mechanismen für asynchrone Ein- und Ausgabe mit, die den Datentransport beschleunigt. Was sie können und wie man sie benutzt, zeigt die nächste Folge der Kern-Technik. *(ofr)*

Infos

[1] Quade, Kunst: „Linux-Treiber entwickeln“, Dpunkt Verlag, Heidelberg, Juni 2004

[2] Eva-Katharina Kunst, Jürgen Quade, „Kern-Technik“, Folge 3: Linux-Magazin 10/03, S. 81

[3] Listings und Makefile: [http://www.linux-magazin.de/Service/Listings/2004/12/Kern-Technik/]

[4] Wehrle et al., „Linux Netzwerkarchitektur“, (Kernel 2.4): Addison-Wesley 2002

[5] Beck et al., „Linux Kernelprogrammierung“, (Kernel 2.4): Addison-Wesley, 6. Auflage, 2001

[6] Rubini, Corbet, „Linux Device Drivers“, (Kernel 2.4): O'Reilly, 2. Auflage, 2001

[7] Template-Netzwerktreiber: [http://lxr.linux.no/source/drivers/net/isa-skeleton.c?v=2.6.5]

Die Autoren

Eva-Katharina Kunst, Journalistin, und Jürgen Quade, Professor an der Hochschule Niederrhein, sind seit den Anfängen von Linux Fans von Open Source. Unter dem Titel »Linux Treiber entwickeln« haben sie zusammen ein Buch zum Kernel 2.6 veröffentlicht.

Gebräuchliche Funktionen von Netzwerktreibern

Funktion	Beschreibung
»struct net_device *alloc_netdev(int sizeof_priv, const char *name, void (*setup)(struct net_device *))« (in »linux/netdevice.h«)	Diese Funktion legt Speicher für ein Objekt »struct net_device« an. Zum Initialisieren des Speichers dient »setup()«. Das erzeugte Netzwerk-Interface spricht man später über den Namen »name« an. Zusätzlich zum Objekt reserviert die Funktion »sizeof_priv« Bytes für treiberspezifische (private) Daten. Der Pointer »priv« enthält die Startadresse dieser Daten. Konnte sie keinen Speicher allozieren, gibt die Funktion »0« zurück.
»struct net_device *alloc_etherdev(int sizeof_priv)« (in »linux/etherdevice.h«)	Die Funktion erzeugt ein Objekt vom Typ »struct net_device« und ruft dabei zur Initialisierung »ether_setup()« auf. Damit repräsentiert sie den folgenden Aufruf: »netstruct = alloc_netdev(sizeof_priv,„eth%d“,ether_setup);«
»void free_netdev(struct net_device *dev)« (in »linux/netdevice.h«)	Löst die Referenz auf das Netzwerk-Interface »dev«. Falls es sich um die letzte Referenz handelt, gibt sie das Objekt frei. Mit »alloc_netdev()« zusätzlich angelegter Speicher wird ebenfalls verfügbar.
»int register_netdev(struct net_device *dev)« (in »linux/netdevice.h«)	Die Funktion registriert den durch die Struktur »dev« gegebenen Netzwerktreiber beim Betriebssystem. Ohne Fehler gibt die Funktion »0«, sonst einen Fehlercode zurück.
»void unregister_netdev(struct net_device *dev)« (in »linux/netdevice.h«)	Meldet das Netzwerk-Interface »dev« wieder beim Kernel ab.
»void ether_setup(struct net_device *dev)« (in »linux/netdevice.h«)	Diese Funktion initialisiert das Netzwerk-Interface »dev« als ein Ethernet-Interface.
»void netif_start_queue(struct net_device *dev)« (in »linux/netdevice.h«)	Mit dieser Inline-Funktion signalisiert der Treiber dem Netzwerk-Subsystem, dass das Netzwerk-Interface »dev« bereit ist Pakete entgegenzunehmen und weiterzureichen.
»void netif_stop_queue(struct net_device *dev)« (in »linux/netdevice.h«)	Über diese Inline-Funktion signalisiert der Treiber dem Netzwerk-Subsystem, dass das Netzwerk-Interface »dev« nicht in der Lage ist Pakete entgegenzunehmen und zu versenden.
»struct sk_buff *alloc_skb (unsigned int size, int gfp_mask)« (in »linux/skbuff.h«)	Diese Funktion reserviert Speicher für einen Socket-Puffer mit einem Datenpuffer der Länge »size«. Da unter Umständen innerhalb von »alloc_skb()« per »kmalloc()« Speicher alloziert wird, gibt »gfp_mask« an, ob die aufrufende Instanz schlafen gelegt werden darf oder nicht. Im Interrrupt-Kontext muss daher für »gfp_mask« der Wert »GFP_ATOMIC« stehen. In diesem Fall kann man alternativ die Funktion »dev_alloc_skb(unsigned int siz)« verwenden (entspricht einem Aufruf von »alloc_skb(size,GFP_ATOMIC)«). Konnte sie keinen Puffer reservieren, gibt die Funktion »0« zurück, sonst die Adresse des Socket-Puffers.
»void kfree_skb(struct sk_buff *skb)« (in »linux/skbuff.h«)	Die Funktion dekrementiert den Referenzzähler des Puffers »skb«. Wird der Socket-Puffer von keiner Instanz mehr verwendet, gibt sie ihn frei.
»unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev)« (in »linux/etherdevice.h«)	Diese Funktion bestimmt den Protokolltyp des in »skb« enthaltenen Ethernet-Pakets. Außerdem stellt sie fest, ob es sich um ein Broadcast- oder ein Multicast-Paket handelt und ob das Paket überhaupt für den eigenen Rechner (»dev«) bestimmt ist. In »skb->pkt_type« wird das zugehörige Flag gesetzt.
»int netif_rx(struct sk_buff *skb)« (in »linux/netdevice.h>«)	Mit Hilfe dieser Funktion übergibt ein Treiber das empfangene Paket »skb« dem Netzwerk-Subsystem zur Weiterverarbeitung. Die oberen Protokollschichten können bei hoher Last das Paket verwerfen. Folgende Rückgabewerte der Funktion gibt es: »NET_RX_SUCCESS«, »NET_RX_CN_LOW«, »NET_RX_CN_MOD«, »NET_RX_CN_HIGH« und »NET_RX_DROP«.