Entwicklung eines Netzwerktreibers für FLink unter Linux

Diplomarbeit Sven Hummel
Matrikelnummer: 952108

Prof. Dr. rer. nat. Arno Fischer
Peter Wegner
Fachbereich Technik
Studienrichtung Angewandte Informatik
Fachhochschule Brandenburg

Brandenburg an der Havel 12. Juni 2000

Zusammenfassung

Die vorliegende Diplomarbeit befaßt sich mit dem Treiber für die FLinkkarte. Die FLinkkarte ist eine Gigabit–Netzwerkkarte für Ringnetzwerktopologien und wurde vom Forschungsinstitut DESY[1] Zeuthen im Rahmen des APEmille Projekts entwickelt. Ein eigens für die FLinkkarte entwickeltes Hardwareprotokoll (FLink) soll eine sehr hardwarenahe Netzwerkverbindung ermöglichen. Die Zielplattform für den Treiber ist Linux. Ziel der Diplomarbeit war es ein Treiber zu entwickeln mit folgenden Eigenschaften:

- hohe Stabilität
- hohe Transferleistung
- lauffähig unter Linuxkernel 2.2.x
- Unterstützung des Hardwarprotokolls FLink

[1] Deutsches Elektronen–Synchrotron

Danksagungen

Ich möchte mich recht herzlich für die geleistete Hilfen und Unterstützungen bei meiner Diplomarbeit bei folgenden Personen bedanken:

- Prof. Dr. rer. nat. Arno Fischer, Fachhochschule Brandenburg
- Ihno Krumreich, Fachhochschule Brandenburg
- Karl–Heinz Sulanke, DESY Zeuthen
- Peter Wegner, DESY Zeuthen
- Norbert Paschedag, DESY Zeuthen

Mein Dank gilt auch allen nichtgenannten Personen, die mir bei der Diplomarbeit zur Seite standen.

Eigenständigkeitserklärung

Hiermit erkläre ich, Sven Hummel, daß ich die vorliegende Diplomarbeit selbständig angefertigt und die benutze Literatur vollständig im Literaturverzeichnis aufgeführt habe.

Brandenburg an der Havel, den 12. Juni 2000

Sven Hummel

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Für die vom DESY Zeuthen entwickelte FLinknetzwerk–Karte soll ein Treiber geschrieben werden, um diese Karte anstelle einer Ethernet–Karte im Linux–System zu verwenden. Es ist dabei darauf zu achten, daß die Übertragungsmöglichkeiten der Karte, soweit dies nicht durch die Hardware eingeschränkt wird, genutzt werden können.

1.2 Vorgehensweise

Diese Diplomarbeit gliedert sich in 3 logische Teile. Im ersten Teil wird die FLinkkarte vorgestellt.

Der zweite Teil erklärt die Besonderheiten eines Treibers unter Linux. Schwerpunkt ist dabei der Netzwerktreiber. Hier werden wichtige Strukturen, Variablen und Funktionen betrachtet.

Der dritte Teil widmet sich dem vorhandenen Treiber. Im funktionalen Abschnitt wird erklärt, wie der Treiber prinzipell arbeitet. Im anschliessenden Abschnitt werden wichtige Funktionen detaillierter untersucht. Der letzte Abschnitt zeigt die Leistungsfähigkeit und Zuverlässigkeit des Treibers, aber auch die Probleme und Schwächen der aktuellen Version.

Kapitel 2

FLinkkarte

2.1 Die FLinkkarte

Die FLinkkarte wurde im Rahmen des APEmille Projekts vom Forschungsinstitut DESY Zeuthen entwickelt.Die FLinkkarte gibt es in verschiedenen Ausführungen. Abbildungen 2.1 zeigt die FLinkkarte mit der Standard–PCI Ausführung und Abbildung 2.2 zeigt die Compact–PCI Ausführung.

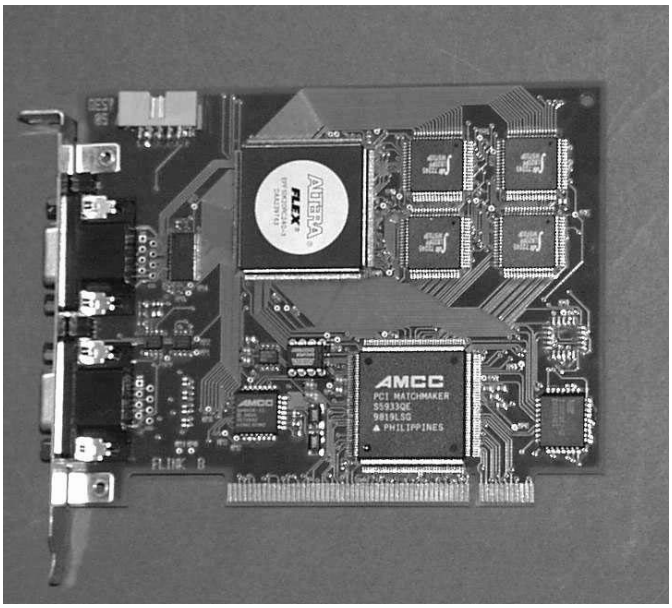


Abbildung 2.1: FLinkkarte für Standard–PCI

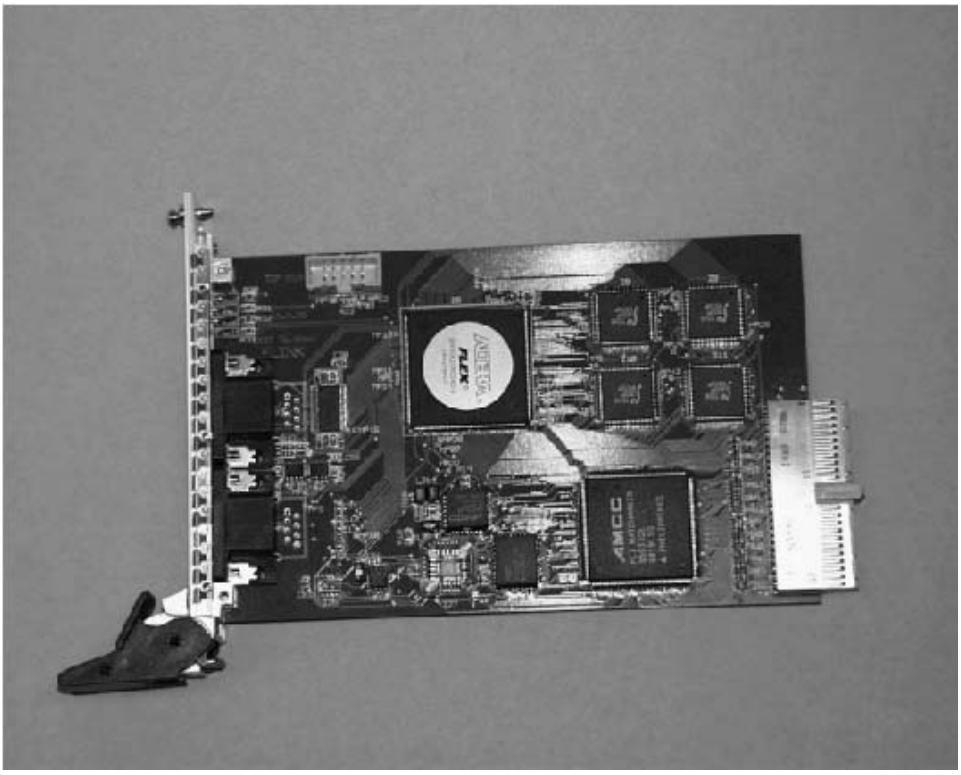


Abbildung 2.2: FLinkkarte für Compakt–PCI

2.1.1 APEmille Projekt

Das APEmille Projekt beschäftigt sich mit massiv parallelen Rechnern und beruhen auf der Entwicklung von italienischen Elementarteilchenphysikern des INFN (Istituto Nazionale di Fisica Nucleare). Die Rechner arbeiten nach dem SIMD (Single Instruction, Multiple Data) Prinzip, bei der alle Prozessoren völlig synchron arbeiten. Sie stellen eine hocheffektive und kostengünstige Lösung für Rechnungen auf dem Gebiet der Gitterfeldtheorie dar, die einen numerischen Zugang zu den grundlegenden Eigenschaften der Elementarteilchen und ihrer Wechselwirkung erlaubt. Damit können Projekte in der Theoretischen Elementarteilchenphysik bearbeitet werden, die auf Grund ihres Leistungsbedarf z.Z. nur mit Spezialrechnern in Japan und in den USA gelöst werden können.

Das Vorgängermodell APE100 gibt es in verschiedener Leistungsstufen. APEmille soll bis zu 1 Tflops Rechnenleistung erreichen und u.a. durch die 64 Bit Gleitkomma–Arithmetik, eine lokale Adressierbarkeit und schnellere Kommunikationsschnittstellen einige Beschränkungen der APE100 Familie aufheben, wodurch sich breitere Anwendungsmöglichkeiten ergeben.

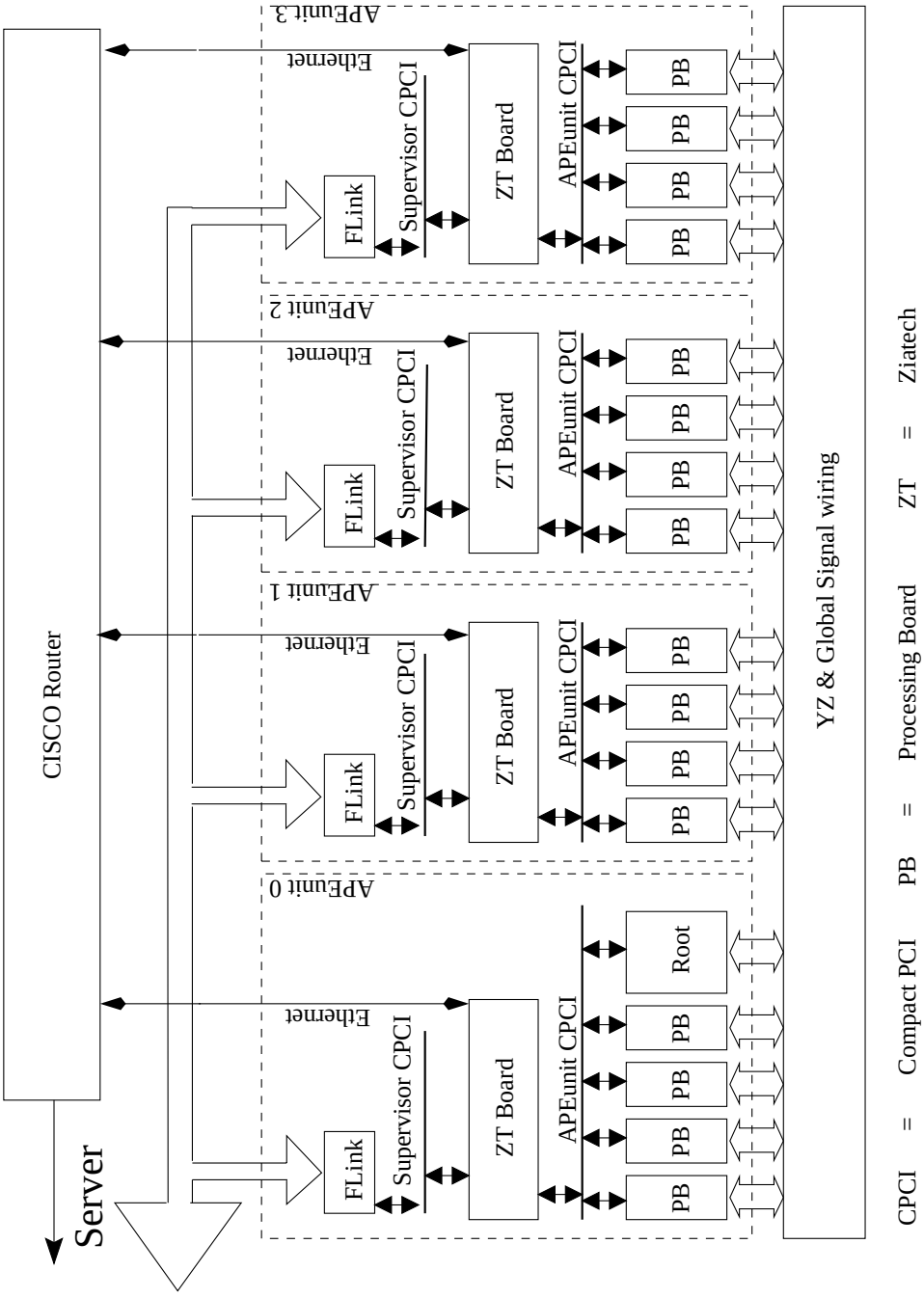


Abbildung 2.3: APEmille Rechner

Der APEmille Rechner (Abbildung 2.3) ist in mehere Einheiten (APEunits) aufgeteilt. Jede Einheit (APEunit, siehe Abbildung 2.4) besteht zumin-

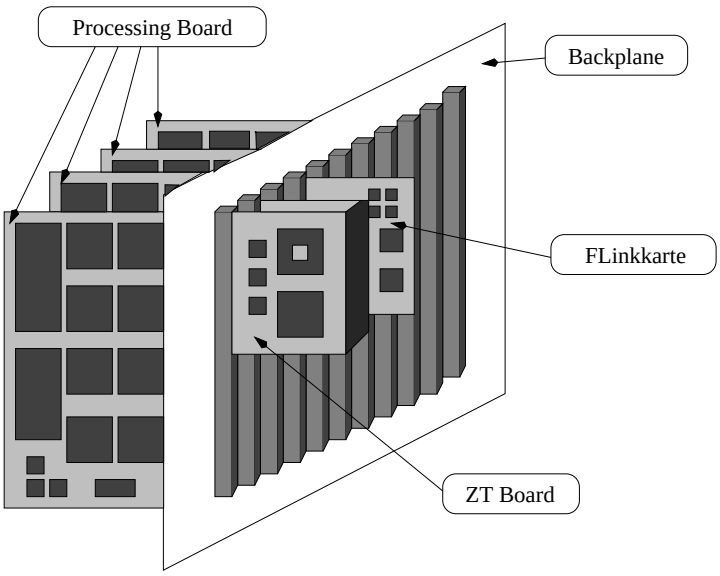


Abbildung 2.4: Darstellung einer APEunit

dest aus einer FLinkkarte, einem Ziatech Board und bis zu vier Processing Boards. Auf einem Processing Board befinden sich 8 Prozessoren und Steuer- und Kommunikationselemente. Das Ziatech Board ist wie ein PC auf einer Compact–PCI Karte mit x86–Prozessor, Arbeitspeicher, Ethernetkarte und Anschlüsse für Maus, Tastatur etc.. Im folgenden Text wird das Ziatech Board auch als Host–PC betitelt. Der Host–PC versorgt die Processing Boards mit Programmen und Daten. Die FLinkkarte ist wie die Ethernetkarte mit einem Server verbunden, der die Rechenwerte und Rechenergebnisse aufbewahrt. Der Vorteil der FLinkkkarte ist die hohe Transferleistung gegenüber der Ethernetkarte. Bis zu einem Gigabyte Rechenergebnisse können pro Host–PC anfallen, die in möglichst kürzester Zeit zum Server transferiert werden müssen. Die Ethernetkarten sind über einem Router (Cisco) mit dem Server und untereinander verbunden. Das Root Board dient zur Synchronisation und liefert den Takt für alle Prozessoren auf allen Processing Boards. Alle Processing Boards sind mit einer Leitung (YZ & Global Signal wiring) verbunden, die für die Gitterfeldsimulation benötigt wird. Abbildung 2.5 zeigt die datentechnische Anordnung der Prozessoren bei einem Processing Board, wenn ein Gitter des Typs 2 x 2 x 2 simuliert wird.

2.1.2 Hardwaredaten

Die FLinkkarte ist eine durch den Mitarbeiter des Forschungsinstitut DESY Zeuthen Karl–Heinz Sulanke entworfene Netzwerkkarte. Die Netzwerkkarte

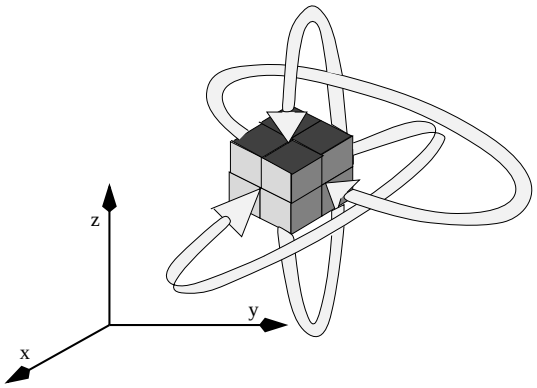
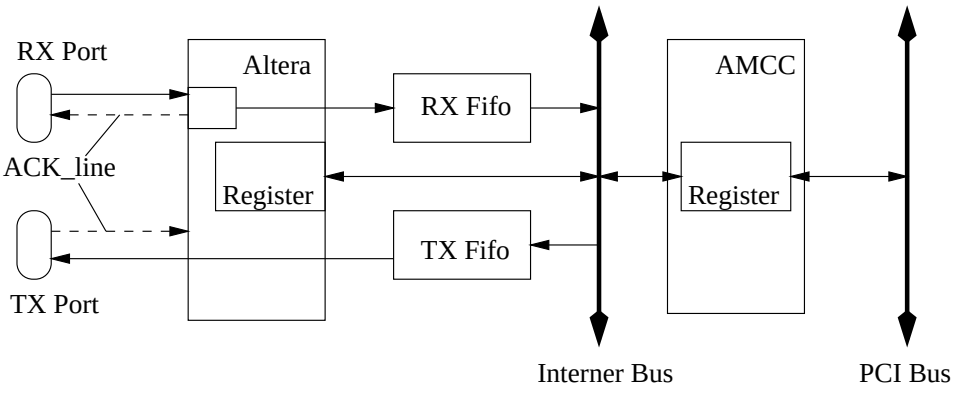


Abbildung 2.5: Gitterfeldanordnung

wird in Projekten wie dem APEmille–Projekt eingesetzt. Zusätzlich entwarf er dafür ein Hardware–Protokoll. Die Karte besitzt eine programmierbare Hardware und kann mit dem Peer–to–Peer Protokoll (P2P) oder dem Ringprotokoll betrieben werden. Die Netzwerkkarte wird in speziellen Ringnetzwerken eingesetzt. Die Kabellänge zwischen zwei nacheinanderfolgenden Karten kann bis zu 10 m betragen. Die aktuelle Version der Karte, mit



(stark vereinfacht)

Abbildung 2.6: Funktionelles Schaltbild der FLink Karte

33 MHz Taktfrequenz, schafft 66 MB Daten pro Sekunde zu transferieren. Zukünftige Versionen sollen bis zu 400 MB pro Sekunde Transferleistung erbringen.

Die Karte besteht im wesentlichen aus ein PCI–BUS Kontroller (AMCC), 2 Speicherchips (Fifos) und einem Logikchip (Altera). Abbildung 2.6 enthält ein funktionales Schaltbild der FLinkkarte.

Der im PCI–BUS Kontroller (AMCC) enthaltene DMA–Kontroller, welcher eine Transferleistung von bis zu 132 MB/sek erreicht, eignet sich für das schnelle kopieren von Daten zwischen Hauptspeicher und Fifo.

Die Hardware faßt die beiden Fifos (RX und TX) zu einem logischem Fifo zusammen, wodurch Einer (der RX Fifo) nur lesbar und der Andere (der TX Fifo) nur beschreibar ist. Damit keine Daten verloren gehen, existiert eine Signalleitung (ACK_line). Die Signalleitung verhindert, das an eine Karte gesandt werden kann, dessen RX Fifo gefüllt ist. Abbildung 2.7 zeigt ein Netz-

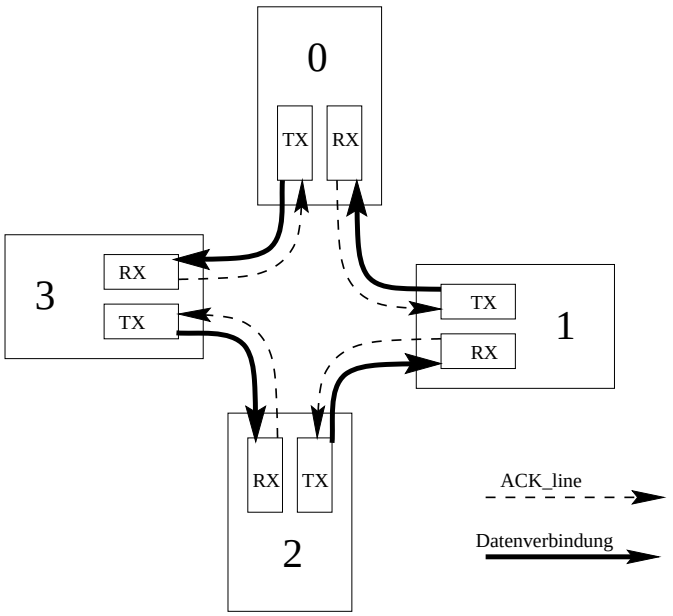


Abbildung 2.7: Darstellung eines Netzwerkrings

werk aus FLinkkarten. Gut erkennbar sind die Signalleitungen (ACK_line), die entgegen der Senderichtung gerichtet ist.

Der Logikchip (Altera) bestimmt das eigentliche Verhalten der Karte. Durch seine Programmierbarkeit lassen sich Fehler relativ schnell beheben, aber auch funktionell lassen sich Eigenschaften hinzufügen oder entfernen.

Die Karte arbeitet mit 32 Bit Datenblöcke. Deshalb können die Netzwerkpakte nur Vielfache von 32 Bit groß sein. Die Datenblöcke werden in 16 bit Portionen versandt, wobei die höheren Bits (Bit 31 – 16) vor den unteren Bits (Bit 15 – 0) versendet werden.

2.1.3 Interruptmöglichkeiten

Die Karte kann einen Interrupt auslösen. Nur der AMCC kann seinen Interruptwunsch direkt dem PCI–BUS übergeben. Die Altera benötigt für einen Interrupt den AMCC. Jener besitzt spezielle Register (Mailboxen), die einen Interrupt auslösen können, wenn sie gefüllt werden. Damit die Altera einen Interrupt auslösen kann, muß sie also die Mailbox beschreiben.

Die Altera ermöglicht bei folgenden Bedingungen einen Interrupt auszulösen:

- der RX Fifo ist nicht leer
- der RX Fifo hat eine bestimmte Menge an Daten
- der TX Fifo ist leer
- der TX Fifo hat eine bestimmte Menge freien Speicherplatz

Aber auch protokollspezifische Sachen, d.h. Ereignisse des Ringprotokolls, können durch den Altera einen Interrupt auslösen:

- Das Ende eines Ringpaketes (LWR oder GWR) ist empfangen worden
- ein normales Paket (LWR) wurde empfangen
- ein Broadcast Paket (GWR) wurde empfangen

Für den AMCC können folgende Interruptmöglichkeiten genutzt werden:

- eine Maibox wird gefüllt (z.B. vom der Altera)
- der DMA–Kontroller hat alles „versandt“
- der DMA–Kontroller hat alles „empfangen“
- Probleme mit dem PCI–Bus (Fehlermeldung)

2.2 Ringprotokoll

Bis zu 256 FLinkkarten können zu einem ringförmigen Netzwerk verbunden werden. Dabei wird der Sendeport mit dem Empfangsport der nachfolgenden Karte verbunden. Jede Karte kann nach dem Register–Insertion Prinzip [1] senden. Jede Karte besitzt eine 8 Bit Adresse. Zur Kommunikation wurden spezielle Worte (32 Bit) definiert. Abbildung 2.8 zeigt den Aufbau eines Wortes. Die Hardware wertet diese Worte mit aus. Daraus ist ersichtlich, warum

[1] Reißverschlußprinzip

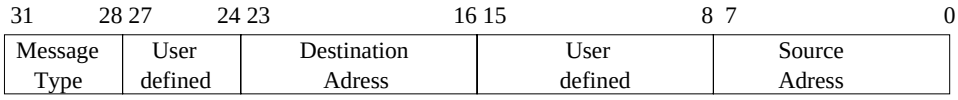


Abbildung 2.8: allgemeiner Aufbau eines Wortes

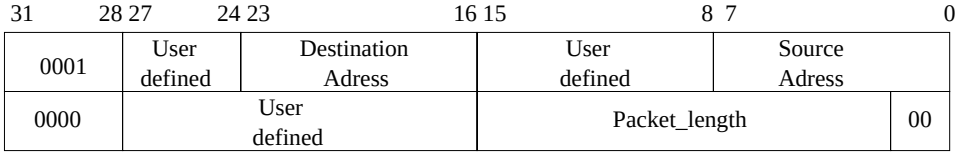


Abbildung 2.9: Hardwareheader für normale Pakete (LWR)

ein einheitlicher Aufbau, wie in Abbildung 2.8 gezeigt, erforderlich ist. Die Auswertung des Datenverkehrs durch die Hardware bietet Möglichkeiten wie Packeterkennung, Fehlererkennung etc.. Ein weiterer Vorteil ist, das innerhalb von 240 ns ein Packet weitergeleitet wird, wenn der Empfänger eine andere Karte ist. Abbildung 2.9 zeigt den Aufbau eines Hardwareheaders. Wie in Abschnitt 2.1.2 beschrieben, sendet die Hardware in 16 Bit Portionen. Die Worte sind so entworfen worden, das die Hardware immer die Zieladresse als Erstes empfängt. Weil die Hardware in 32 Bit Datenblöcken arbeitet, werden den letzten beiden Bits bei der Längenangabe standartmäßig der Wert 0 zugewiesen. Im folgendem eine Auflistung der definierten Worte:

LWR Teil eines Hardwareheaders, das Paket wird normal versendet

GWR Teil eines Hardwareheaders, das Paket ist ein Broadcast

PLE Teil eines Hardwareheaders, hier steht die Größe des Paket in Bytes

TOK Kurznachricht: Transfer OK

OPN Befehl: Meldet diese Karte im Netz an

CLS Befehl: Meldet diese Karte im Netz ab

RRS Befehl: Rücksetzen einer entfernten Karte

2.3 Peer–to–Peer Protokoll

Beim Peer–to–Peer Protokoll (P2P) bilden 2 Karten ein Netzwerk. Der Sendeport wird mit dem Empfangsport der anderen Karte verbunden. Anders als beim Ring Protokoll ist keine hardwaremäßige Unterstützung vorhanden.

Deshalb gibt es auch keine hardwaremäßige Paketerkennung. Die Daten werden aus dem Sendefifo in den Empfangsfifo der anderen Karte verschoben.

Kapitel 3

Linuxtreiber

3.1 Was ist Linux?

Linux ist ein freies UNIX Betriebssystem. 1991 startete Linus Thorvald die Entwicklung von Linux. Heute entwickeln tausende freiwilliger Programmierer aus dem Internet Linux weiter. Neben PC's ist es auch auf vielen anderen Plattformen verbreitet. In der heutigen Zeit ist das aktive Interresse und die Unterstützung vieler weltbekannter Firmen für dieses Betriebssystem geweckt worden.

Anders als bei anderen Betriebssystemen hat Linux kein festes Erscheinungsbild. Linux besteht aus dem Kernel und hunderten von Programmen größtenteils aus dem GNU–Software Projekt. Eine Zusammenstellung von Kernel und Programmen wird in Linuxkreisen als Distribution bezeichnet. Inzwischen gibt es unzählige Distributionen, die sich mehr oder weniger in ihrer Philosophie und ihren Zielen unterscheiden.

3.2 Was ist der Kernel?

Der Kernel bildet den fundamentalen Kern von Linux. Er stellt die grundlegensten Funktionen für Prozeßsteuerung, Speichermanagment, Dateisystem, Gerätesteuerung und Netzwerk. Viele Funktionalitäten des Kernels sind in Treibern realisiert. Dabei teilen sich häufig viele Treiber eine Gesamtaufgabe. Durch definierte Schnittstellen arbeiten die Treiber zusammen. Der Kernel enthält dafür unterstützende Funktionen, Variablen und Strukturen, z.B. *jiffies*. Die Variable *jiffies*. enthält die Anzahl der Timerinterrupte seit Systemstart. Sie wird von Treibern zur Zeitbestimmung genutzt.

Treiber die zur Laufzeit des Kernels hinzugefügt und entfernt werden können, werden als Module oder modulare Treiber bezeichnet.

3.3 Allgemeines

Kernelroutinen arbeiten mit einer abstrakten Vorstellung über die Hardware. Der Treiber für die Hardware hingegen kennt die Details der Hardware und stellt entsprechende Routinen zur Verfügung. Der Treiber ist die Schnittstelle zwischen Hardware und dem Kernel.

Aus Treibersicht gibt es nur 3 Gruppen in die Hardware einsortiert wird. Das sind zeichenorientierte Geräte, blockorientierte Geräte und Netzwerkinterface.

3.3.1 zeichenorientierte und blockorientierte Geräte

Auf ein zeichenorientiertes oder blockorientiertes Gerät kann zugegriffen werden wie auf eine Datei. Üblicherweise sind Systemaufrufe wie `open, close, read` und `write` implementiert. Ein blockorientiertes Gerät ist meist ein Medium, auf dem ein Dateisystem existiert. Auf blockorientierten Geräten werden nur Datenblöcke gelesen und geschrieben, wobei das Gerät festlegt wie groß ein Block ist. Auserdem kann auf blockorientierten Geräten wahlfrei zugegriffen werden. Im Gegensatz dazu werden zeichenorientierte Geräte normalerweise sequentiell gelesen und beschrieben, und die Ein/Ausgabe kann byteweise erfolgen. Ein typischer zeichenorientierter Gerätetreiber ist der Treiber für die serielle Schnittstelle und ein typischer blockorientierter Gerätetreiber ist der IDE–Festplattentreiber.

Linux verwaltet einen Großteil seiner Geräte, hauptsächlich zeichenorientierten und blockorientierten Geräte, mit Einträgen im Dateisystem unter */dev*. Jede dieser Einträge enthält eine Minor– und eine Majornummer. Die Majornummer identifiziert den Treiber für die Hardware. Die Minornummer wird frei vom Treiber benutzt. Es gibt Treiber die mehrere Hardwarekomponenten ansteuern. Durch die Minornummer kann der Treiber in einen solchen Fall die Hardware unterscheiden.

3.3.2 Netzwerkinterface

Ein Netzwerkinterface verbindet üblicherweise Rechner miteinander um Daten auszutauschen. Es werden Systemaufrufe wie `open, close` und `send` realisiert. Ein Äquivalent für `read`, wie bei zeichenorientierten und blockorientierten Geräten, gibt es in dieser Art nicht. Es gibt zwar eine `receive` Funktion, aber diese ruft Kernelroutinen auf, wenn die Daten verfügbar sind. Ein typisches Netzwerkinterface ist die Ethernet–Karte. Ein spezieller Teil des Kernels kümmert sich um die Datenpakete.

Im Gegensatz zu den zeichenorientierten und blockorientierten Geräten hat ein Netzwerkinterface normalerweise keinen Eintrag im Dateisystem. Netz-

werkinterface unterscheiden sich durch eindeutige Namen wie *eth0*. Spezielle Programme, wie *ifconfig* und *route*, ermöglichen die Nutzung des Netzwerkinterfaces durch Applikationsprogramme.

3.3.3 *User space* und *Kernel space*

Linux ist ein Multiprocessing Betriebssystem, d.h. mehrere Prozesse laufen gleichzeitig auf einer CPU. Das Betriebssystem verhindert, das sich die Prozesse gegenseitig negativ beeinflussen. Kein Prozess darf in den Adressraum des Anderen zugreifen oder eine Ressource ewig belegen. Dieser Schutzmechanismus wird über Levels realisiert. Bei Linux gibt es 2 Levels. Der Kernel wird im höchsten Level (supervisor mode), wo alles erlaubt ist, und die Programme werden im niedrigsten Level (user mode), wo nur eingeschränkte Rechte gelten, ausgeführt. Bei der Softwareprogrammierung wird vom *kernel space* und vom *user space* gesprochen. Wenn ein Programm im *user space* eine Speicherschutzverletzung verursacht, wird das Programm beendet und mit einem Debugger kann Zeile für Zeile der Fehler im Sourcecode gesucht werden. Aber wenn nun ein solch' fehlerhaftes Programm im *kernel space* läuft, ist mit schlimmeren Folgen, wie „Kernel Panik“[1], Softreset[2] oder Hardwareschäden, zu rechnen.

3.4 PCI–Geräte

PCI (Peripheral Component Interconnect) ist eine Sammlung an Spezifikationen, die das Zusammenarbeiten verschiedener Computerteile beschreibt. Laut PCI–Spezifikation wird jedes Gerät durch Busnummer, Gerätenummer und Funktionsnummer identifiziert. In einem PC existiert meistens nur ein PCI–Bus, aber es können bis zu 256 Busse vorhanden sein. Jeder Bus kann bis zu 32 Geräte ansprechen und jedes Gerät (Karte) kann aus mehreren funktionellen Einheiten bestehen.

Jeder PCI–Slot und damit jede PCI–Karte besitzt ein PCI–Konfigurationsregister, das irgendwo im PCI–Konfigurationsadressraum liegt. Zum Bootzeitpunkt handelt das BIOS mit jeder PCI–Karte die benötigten Resourcen, wie IRQ, E/A–Adressen usw., aus und vergibt sie möglichst Konfliktfrei.

Herstellernummer Die Herstellernummer ist weltweit eindeutig und identifiziert den Hersteller.

Gerätenummer Die Gerätenummer wird vom Hersteller vergeben.

Status Dieses Feld gibt Statusinformationen über das Gerät.

[1] Ein nicht behebbarer Fehler, der vom Kernel selbst gemeldet wird
[2] Ein von Software ausgelöster Neustart des Rechners

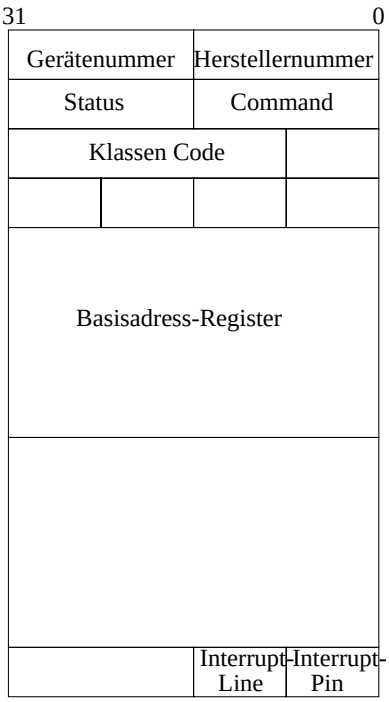


Abbildung 3.1: Darstellung des PCI–Konfigurationsregisters

Command Das Beschreiben dieses Feldes erlaubt dem System das Gerät zu kontrollieren.

Klassencode Identifiziert den Typ des Gerätes. Es gibt Standardklassen für jede Art von Geräten.

Basisadressregister Mit diesen Feldern können die PCI–E/A- und PCI–Speicheradressen des Gerätes bestimmt werden.

Interrupt Pin Vier physikalische Pin leiten den Interrupt von der Karte zum PCI–Bus. Sie werden standardmässig mit A, B, C und D bezeichnet. Dieses Feld beschreibt welchen Pin das Gerät benutzt.

Interrupt Line In diesem Eintrag steht der der Karte zugewiesene Interrupt.

Der PCI–Konfigurationscode des BIOS's findet ein PCI–Gerätes durch den Versuch ein Feld, z.B. Herstellernummer, aus dem Register auszulesen. Bei einem leerem PCI–Slot wird eine Fehlermeldung, z.B. 0xFFFFFFFF, zurückgeliefert. Das PCI-Konfigurationsregister erlaubt dem System das Gerät zu identifizieren und zu konfigurieren. Nur der PCI–Konfigurationscode kann lesend und schreibend auf den PCI–Konfigurationsadressraum zugreifen.

Der Linuxtreiber kann nur auf die PCI–E/A- und PCI–Speicheradressen der PCI–Karte zugreifen. Durch BIOS–Routinen können die PCI–Busse z.B. nach einer PCI–Karte mit dem einer bestimmten Herstellernummer und Kartennummer durchsucht werden. Im Erfolgsfall werden Gerätenummer, Busnummer und Funktionsnummer zurückgeliefert. Mit Hilfe dieser drei Nummern können durch andere BIOS–Routinen die anderen Felder ausgelesen werden. Mit dem 2.2.x Kernel wird dieser Aufwand für den Treiberschreiber etwas vereinfacht.

Für weiterführende Informationen zum Thema PCI werden „PCI Hardware and Software“ ([6]) und „PCI System Architekture“ ([9]) empfohlen.

3.5 Interrupt

Wird ein Interrupt benutzt, so benötigt der Treiber einen Interrupthandler. Der Treiber teilt dem Kernel den benötigten Interrupt sowie den dazugehörigen Interrupthandler mit. Erfolgt der gewünschte Interrupt, so ruft der Kernel den Interrupthandler auf. Jener hat nun die Möglichkeit den Interrupt zu bearbeiten.

Aufgrund der Limitierung der Anzahl der Interrupte können sich mehrere PCI–Geräte hardwaremäßig einen Interrupt teilen. Dieses Funktionalität muß dann aber auch der Treiber unterstützen, speziell der Interrupthandler. Ein Interrupthandler, der dies unterstützt wird als shared Interrupt–Handler bezeichnet, andernfalls als non-shared. Bei einem Interrupt werden alle zu diesem Interrupt gehörigen Interrupthandler aufgerufen. Daher muß ein shared Interrupt-Handler noch feststellen, ob sein Gerät der Verursacher des Interrupts ist.

3.6 Netzwerktreiber

Die Hauptaufgabe eines Netzwerktreibers ist es Pakete zu versenden und zu empfangen. Im Detail muß der Treiber die Hardware initialisieren, sie überwachen und Ausnahmezustände bearbeiten. Weiterhin bietet der Treiber Funktionen an, die Auskunft geben oder die Hardware konfigurieren. Auch Hardwareheader, wie zum Beispiel Ethernet, werden vom Treiber bearbeitet.

Als Netzwerktreiber müssen wenigsten die Funktionen `open()`, `close()`, `send()` und `receive()` bereitgestellt werden. Für Module müssen noch die Funktionen `init_module()` und `cleanup_module()` existieren. Ein Blick in die vollständige Device–Struktur läßt die Anzahl der möglichen Funktionen nur erahnen.

Eines der verbreitesten Netzwerkprotokolle ist TCP/IP.

3.6.1 TCP/IP, UDP/IP

Das Internet Protokoll (IP) wurde 1981 standardisiert. Jeder Rechner erhält eine 32 bit lange und einzigartige Adresse. Gewöhnlich wird diese Adresse als vier durch Punkte getrennte Zahlen dargestellt, z.B. 192.168.1.1. Die Adresse ist in Netzwerkadresse und Hostadresse unterteilt. Je nach IP–Adressklasse variieren die jeweiligen Anteile der Adressen. Ein IP–Paket enthält die Quell- und Ziel–IP–Adresse, eine Prüfsumme und andere nützlich Informationen. Die Prüfsumme wird aus dem Header des IP–Paketes gewonnen und erlaubt einen Aufschluß über die Unversehrtheit des Headers. Die Größe eines IP–Paketes variiert abhängig vom Transportmedium.[3] Andere Protokolle bauen auf das IP–Protokoll auf, z.B. TCP.

TCP (Transmission Control Protocol) ist verbindungsorientiertes Protokoll. Durch TCP wird zwischen den beiden Netzwerkprogrammen eine virtuelle Verbindung aufgebaut. TCP beinhaltet Maßnahmen die eine sichere Übertragung der Pakete erlauben. Dadurch werden Paketverluste, Paketduplikate und falsche Paketreihenfolgen unterbunden.

UDP (User Datagram Protocol) ist ein verbindungsloses Protokoll. Im Gegensatz zum verbindungsorientierten Protokoll wird keine virtuelle Verbindung aufgebaut und es existieren keine Maßnahmen für eine sichere Übertragung der Pakete.

Das IP–Paket selbst kann wiederum in ein anderes Protokoll eingebunden sein, z.B. Ethernet, abhängig vom Übertragungsmedium (Hardware). Die Hardwareprotokolle können eine andere Rechneradressierung als IP haben. Linux nutzt ARP (Address Resolution Protocol) um die verschiedenen Adressierungen ineinander zu konvertieren.

Zur Kommunikation zweier Netzwerkprogramme werden Sockets genutzt. Ganz einfach könnten Sockets als spezielle Pipes umschrieben werden.

3.6.2 Device–Struktur

Die Device–Struktur ist für einen Netzwerktreiber unentbehrlich. In dieser Struktur legt der Netzwerktreiber seine wichtigen Information ab. Aber auch der Kernel benötigt diese Struktur, um Informationen über den Netzwerktreiber zu bekommen. Den meisten Funktionen vom Treiber wird deshalb der Zeiger auf diese Struktur direkt übergeben. Eine gekürzte Fassung der Device–Struktur folgt:

char *name Der Name des Interfaces. In unserem Fall *fl0*. Die 0 weist darauf hin, das es die erste Karte dieser Art ist.

[3] Der Treiber teilt die maximale Größe eines IP–Packetes in der Variablen *dev->mtu* mit

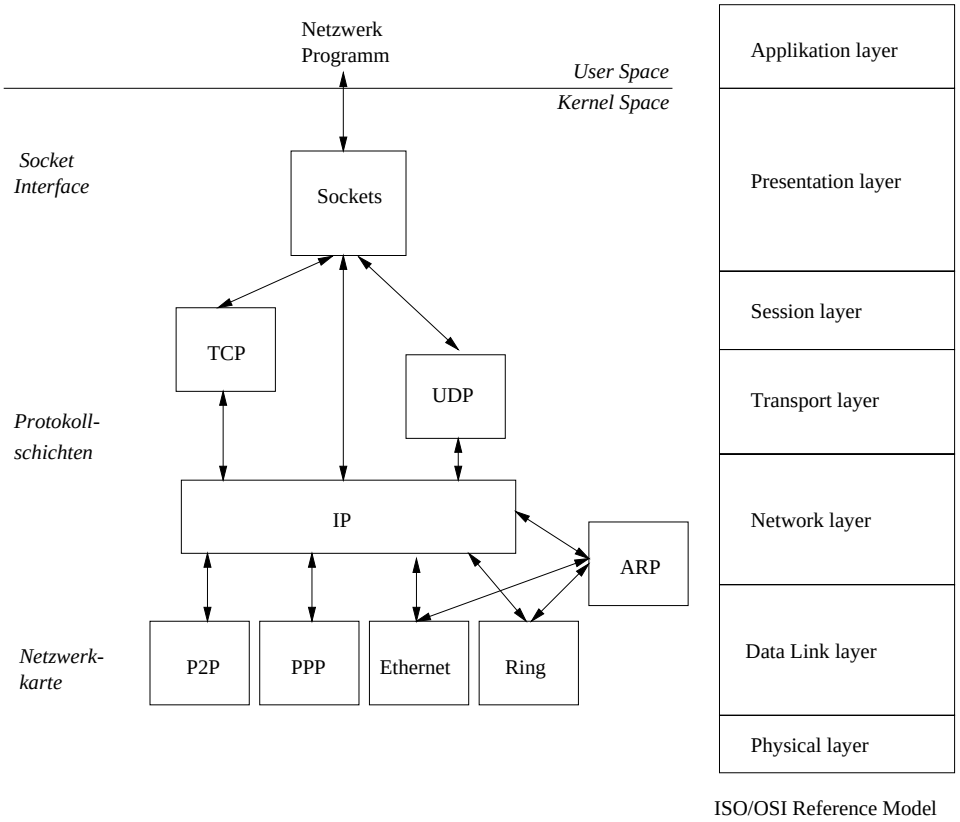


Abbildung 3.2: Darstellung der Netzwerkschichten

unsigned long base_addr Die E/A Basisadresse der Karte.

unsigned char irq Der benutzte Interrupt der Karte.

unsigned char interrupt Der Kernel erkennt an diesem Flag, ob der Treiber gerade einen Interrupt abarbeitet.

unsigned long tbusy Dieses Flag zeigt dem Kernel, ob der Treiber zur Zeit ein weiteres Paket versenden kann oder nicht.

unsigned short mtu Diese Variable definiert, wieviele Bytes (Octets) ein Paket inklusive Header maximal enthalten darf.

int (*open)(struct device *dev) Hier wird die Funktion eingetragen, die das Interface für den Kernel aktiviert. Typischerweise wird diese Funktion durch *ifconfig* aufgerufen.

int (*stop)(struct device *dev) Hier wird die Funktion eingetragen, die das Interface für den Kernel stillegt. Typischerweise wird diese Funktion durch *ifconfig* aufgerufen.

int (*hard_start_xmit)(struct sk_buff *skb, struct device *dev)
Die hier eingetragene Funktion wird vom Kernel aufgerufen, wenn ein Paket versendet werden soll. Das Paket befindet sich in einem Socket Buffer (sk_buff).

unsigned long trans_start In dieser Variable hält der Treiber den Startzeitpunkt des letzten versendeten Paketes fest. Die Zeiteinheit ist *jiffies*.

void *priv Dieser Zeiger steht zur freien Verfügung und weist meistens auf die treibereigene Struktur.

3.6.3 Socket Buffer

Über diese Struktur werden Netzwerkpakete zwischen Kernel und Treiber ausgetauscht. Ein Netzwerkpaket kann verschiedene Netzwerkprotokolle[4] beinhalten. Jedes dieser Protokolle werden durch unterschiedliche Funktionen realisiert, die ihrerseits ihre Header und Tails hinzufügen oder entfernen. Damit ein ständiges umkopieren der Daten vermieden wird, werden Socket Buffers benutzt. Hier eine gekürtzte Fassung der Struktur sk_buff:

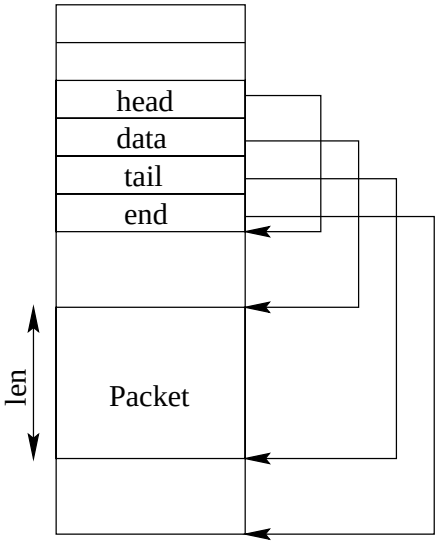


Abbildung 3.3: Darstellung des Socket Buffers

unsigned char *head Dieser Zeiger weist auf den Anfang des belegten Speicherbereichs,

[4] z.B. Ethernet, TCP, UDP, IP

unsigned char *end und hier auf das Ende.

unsigned char *data Hier wird der Anfang des Paketes festgehalten,

unsigned char *tail und hier das Ende.

unsigned long len Die Größe des Paket wird hier gespeichert.

unsigned char ip_summed In diesem Feld trägt der Treiber für jedes empfangene Paket das benötigte Prüfsummenverhalten ein.

unsigned char pkt_type Der Treiber setzt hier PACKET_HOST *(das Paket ist für uns)* PACKET_BROADCAST, PACKET_MULTICAST oder PACKET_OTHERHOST ein.

union { unsigned char *raw; [...]} mac Dieser Zeiger zeigt auf das Ende des Hardwareheaders.

Kapitel 4

Der Treiber

4.1 Herangehensweise

Der Treiber soll die Protokolle TCP/IP und UDP/IP unterstützen. Am einfachsten geschieht das durch einen Netzwerkinterface–Treiber. Deshalb wurde kein zeichenorientierter Treiber gewählt.

Die größte Unterstützung im Kernel haben mitunter die Ethernettreiber. Deshalb wurde mit einem Ethernettreiber für die FLinkkarte begonnen. Danach wurde der Ethernetheader durch einen eigenen Header ersetzt.

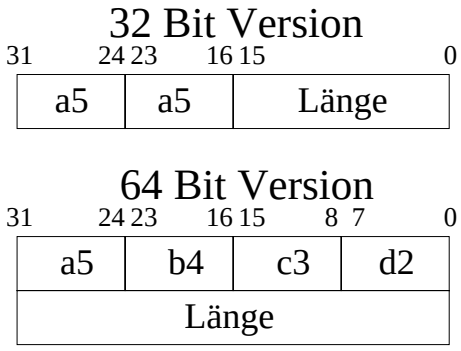


Abbildung 4.1: eigene Hardwareheader

Zuerst war der Header nur 32 Bit groß, danach aber, um ihn sicherer von den Daten unterscheiden zu können, wurde er auf 64 Bit vergrößert (siehe Abbildung 4.1). Auch die Ethernetroutinen wurden mit der Zeit ersetzt.

Nachdem der Treiber sich als stabil erwies, wurde wegen einer besseren Performance experimentiert. Leider verschlechterten diese Experimente zum Teil auch die Stabilität des Treibers. Als Ergebniss dieser Experimente exi-

stieren eine ganze Reihe von Sicherheitsroutinen, die die Stabilität des Treibers erhöhen sollen.

Der Treiber wurde auf den noch weit verbreiteten Kernel 2.0.36 entwickelt. Es gab zum Zeitpunkt des Schreibens eine recht ausführliche Dokumentation über die Treiberschnittstellen des 2.0.x Kernels. Zum 2.2.x Kernel gab es aber nur spärliche Informationen über die erfolgten Änderungen. In der letzten Phase wurden Änderungen vorgenommen, die den Betrieb auch mit dem Kernel 2.2.13 erlauben.

In der nächsten Phase sollte der Header durch die FLinkheader ersetzt, sowie eine Reihe von FLinkfunktionen realisiert werden. Dazu zählt auch die Nutzung des FLinkprotokolls. Weiterhin sollte an Möglichkeiten zur Performancesteigerung gearbeitet werden, da wahrscheinlich noch nicht das Maximum gefunden worden ist.

4.2 Grobe Funktionalität

Der aktuelle Treiber ist eine Peer–to–Peer Implementierung, d.h. er nutzt das P2P Protokoll. Im folgenden wird der funktionelle Teil behandelt, wie in der Grafik 4.2 auf der Seite 22 dargestellt ist. Wenn der Kernel einen Interrupt von der (FLink-)Karte erhält, ruft er den Interrupthandler auf. Der Interrupthandler wertet das INTCSR Register des AMCC aus. Stellt er dabei fest, daß der DMA Kontroller die gewünschte Datenmenge in den Socket Buffer kopiert hat, wird der Socket Buffer dem Kernel übergeben.

Befinden sich jedoch neue Daten im RX Fifo, so wird die Empfangsfunktion aufgerufen. Die Empfangsfunktion kontrolliert den RX Fifo auf Datenbestand. Wie bereits erwähnt, ist das Auslesen des leeren RX Fifos zu vermeiden. Im Normalfall befinden sich Daten im RX Fifo.

Am Anfang des Paketes wird der Hardware Header erwartet. Fehlt nun dieser Hardware Header, verwirft die Fehlerroutine den Rest des Paketes. Findet sich im Anschluß ein neuer Hardware Header, so wird er ausgewertet.

Der Hardware Header dient einmal als Erkennungszeichen für ein neues Paket. Aber auch die Längeninformation des Paketes im Hardware Header wird benötigt. Mit der Längeninformation wird ein Socket Buffer besorgt und der DMA Kontroller programmiert. Ab Treiberversion 0.3.1 können kleinere Pakete auch direkt ausgelesen werden. Damit muß nicht auch für kleine Pakete der DMA Kontroller programmiert werden. Dadurch wird Zeit eingespart und somit Performance gewonnen.

Will der Kernel ein Netzwerkpaket versenden, ruft er die Sendefunktion *flink_tx* auf und übergibt den Socket Buffer des zu versendenen Paketes. Die Funktion kontrolliert, ob nicht noch gesendet wird. Wird gerade noch gesendet, so wird aktiv auf die Sendebereitschaft gewartet. Tritt innerhalb einer

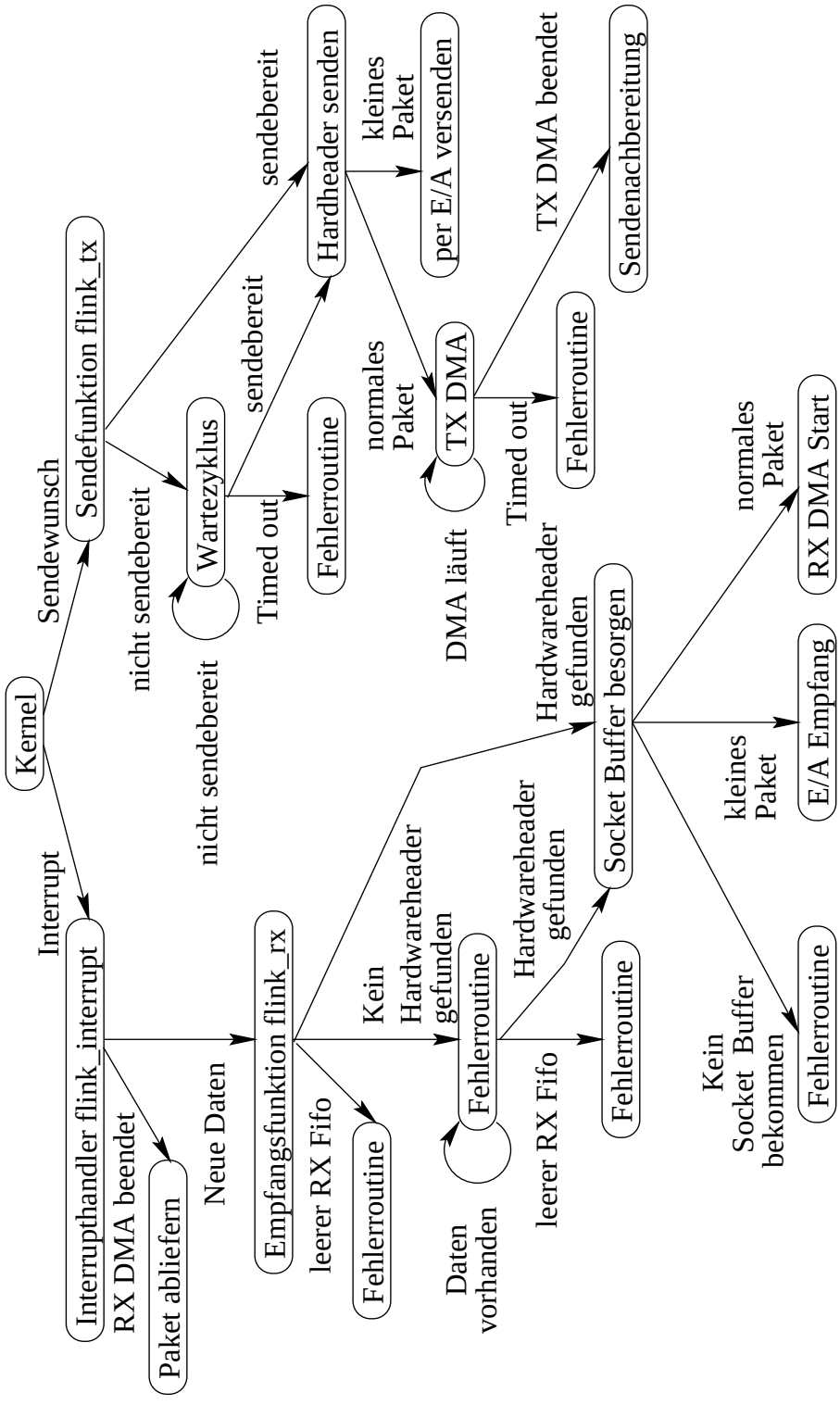


Abbildung 4.2: Funktionsweise Treiber 0.3.1

festgelegten Zeitspanne diese nicht ein, wird eine Fehlerroutine aufgerufen. Sind Treiber und Karte sendebereit, werden noch kleinere Anpassungen am Netzwerkpaket erledigt. Dazu zählt auch der Hardware Header, der vor dem Paket versendet wird. Dann wird der DMA Kontroller programmiert und damit der TX DMA gestartet. Danach wird aktiv (polling) auf das Ende des DMA Prozesses gewartet. Treiberversion 0.3.1 erlaubt das direkte Versenden kleinerer Paket ohne die DMA zu benutzen.

4.3 Ausgewählte Funktionen

4.3.1 Interrupthandler *flink_interrupt*

Der Interrupthandler versucht anhand des Interrupt Control and Status Register (INTCSR) aus dem AMCC die Ursache des Interrupts herausfinden und die passende Prozedur aufrufen.

Folgende Interruptquellen werden ausgewertet:

- Mailbox (Interruptwunsch von der Altera),
- Write Complete (RX DMA beendet),
- Master und Target Abort (Fehler vom AMCC gemeldet).

Als shared Interrupt–Handler muß kontrolliert werden, ob der Interrupt von der FLinkkarte stammt. In der Variablen *int_status* wird der Inhalt des INTCSR Register gelesen. Jede Ursache besitzt ein Statusflag. Daraus ergibt sich folgender Sourcecode:

```
if (int_status & INTR_ASSERTED ||
    int_status & INTR_MASTER_ABORT ||
    int_status & INTR_TARGET_ABORT)
{
    ...
} else {
    /* this interrupt is not for us */
    return;
}
```

Die Ursache *Write Transfer Count Register ist Null* wird durch das Flag INTR_WRITE_TR_COMPL angezeigt. Hierbei hat der DMA–Kontroller die programmierte Anzahl an Datenbytes aus den FIFO in den vorbereiteten Socket Buffer verschoben. In Abbildung 4.2 wird dies als „RX DMA beendet“ bezeichnet.

```
if (int_status &  INTR_WRITE_TR_COMPL)
{
    /* just complete with copy into memory */
    ...
    return;
}
```

Der Altera Chip kann eine Incoming Mailbox beschreiben. Je nach Konfiguration des Interrupt Enable Register (IER) des Altera Chips geschieht das, z.B. wenn der RX FIFO nicht mehr leer ist. Im Treiber wird die Mailbox ausschließlich dazu benutzt, zu signalisieren, das der RX Fifo nicht mehr leer ist. Damit dient das Incoming Mailbox Register (INTR_IN_MAILBOX), als Zeichen das ein neues Paket bzw. neue Daten eingetroffen ist/sind. Entsprechend wurde in der Abbildung 4.2 dieses Ereignis als „Neue Daten“ bezeichnet.

```
if (int_status  & INTR_IN_MAILBOX) {
    /* exclusive use of mailbox for incoming data */
    /* okay new packets arrived */
    /* mark that  an interrupt  will be served */
    if (test_and_set_bit(0,(void *) &dev->interrupt) !=0 ) {
        printk( KERN_ERR "Reentering Interruptroutine \n");
        return;
    }

    /* calling the receiving function */
    ret_value = flink_rx(dev);
    ...
    if (ret_value == 1) {
    /* No DMA started, maybe a failure occured? */
        ...
    }
    else { /* DMA started, normal case */
        ...
    }

    clear_bit(0,(void*) &dev->interrupt);
    return;
}
```

Einige Flags im INTCSR müssen wieder zurückgesetzt werden. Als Beispiel wird hier das Flag INTR_IN_MAILBOX (Neue Daten) genommen.

```
if(int_status & INTR_IN_MAILBOX) {
```

```
    /* exclusive use of mailbox for incoming data */
    /* okay new packets arrived */
     ...
    /* No DMA started, maybe a failure occured */
    /* now restore the old interrupt register
       all interrupts are allowed */
    int_status = (int_status & 0xffff);
    outl(INTR_IN_MAILBOX|int_status,
        dev->base_addr + AMCC_OP_REG_INTCSR );
    ...
}
```

Nach dem Beschreiben der Mailbox wird automatisch eine Sperre aktiviert, die das Überschreiben der Mailbox verhindert. Die folgende Zeile

```
outl(ICR_INTR_ENABLE,privptr->pld_addr + PLD_OP_REG_ICR);
```

entfernt diese Sperre. Damit kann die Mailbox wieder beschrieben werden.

Für den Fall das ein Master Abort oder Target Abort auftritt, existiert folgender Sourcecode.

```
/* this should clear Master-abort or Target-abort */
outl(int_status,dev->base_addr + AMCC_OP_REG_INTCSR );
...
return;
```

Dieser Sourcecode beseitigt nur den Interrupt, aber nicht die Ursache. Die Ursachen für solche Interrupte können nicht vorherbestimmt werden. Vielmehr zeugen Master Abort und Target Abort von Fehlern, die im Treiber oder in der Hardware zu suchen sind. Als Beispiel wäre da ein fehlerhaft programmierter DMA–Kontroller zu nennen.

4.3.2 Empfangsfunktion *flink_rx*

Im Wesentlichen verschiebt die Empfangsroutine Datenpakete aus dem Fifo in den Hauptspeicher. Der Kernel benötigt für jedes Paket noch zusätzliche Informationen.

Wie bereits erwähnt dreht sich beim Datenaustausch zwischen Kernel und Treiber alles um den Socketbuffer[1]. Der Pointer auf diese Struktur wird letztendlich an eine Kernelroutine weitergereicht.

Das Auslesen eines leeren Fifos führt zum Stillstand des Systems. Deshalb wird sicherheitshalber erst geprüft, ob wirklich Daten im RX Fifo vorhanden sind. In Abbildung 4.2 wird dies als „leerer RX Fifo“ dargestellt.

[1] siehe Kapitel 3.6.3

```
/* The try to read an empty RX-FIFO blocks the CPU,
so be extra carefull.
*/
if ( inl(dev->base_addr + AMCC_OP_REG_MCSR) &
                        MCSR32_A2P_FIFO_EMPTY)
{
    printk(KERN_WARNING "%s NO Data  !!! \n",dev->name);
    return 1; /*failure */
}
```

Damit die Empfangsroutine den Anfang eines neuen Paketen erkennt, existiert ein noch frei erfundener Hardware Header. Er beinhaltet ein 32-bit grosses Kontrollzeichen (siehe dazu auch Abbildung 4.1). Dieses Kontrollzeichen darf auch in den Nutzdaten vorkommen.

Der folgende Sourcecode zeigt in Abbildung 4.2 die Möglichkeit „Kein Hardwareheader gefunden“, wobei die eigentliche Fehlerroutine nur angedeutet wird.

```
/* the beginning of a new packet */
/* first 32bit, the Control-sign*/
infopkt=inl(dev->base_addr + AMCC_OP_REG_FIFO);
if((infopkt - CTRL_SIGN3 ) != 0)
{   /* this test for jam */
    /*failure*/
    ...
}
```

Im Hardware Header gibt es ein Feld für die Längenangabe. Darin ist die Anzahl der Bytes des IP Paketes enthalten, d.h. die 8 Bytes des Hardware Headers[2] werden nicht berücksichtigt. Mit dieser Information wird ein Socket Buffer angefordert, was in Abbildung 4.2 als Routine „Socket Buffer besorgen“ dargestellt ist.

```
/* alloc the a socketbuffer */
skb = dev_alloc_skb(len);
if (!skb) { /* havent got a socketbuffer */
    printk(KERN_WARNING "flink rx: low on Memory\n");
    ...
    return 1; /* failure */
}
```

Mit der Längenangabe wird auch der DMA Kontroller programmiert, der das IP Paket aus dem Fifo in den angeforderten Socket Buffer verschiebt.

[2] ab Treiberversion 0.2.3 ist der Header 64 Bit groß

Abbildung 4.2 fasst das unter der Routine „RX DMA Start“ zusammen. Fehlen nun Daten, wartet der DMA Kontroller auf die restlichen Bytes. Kommt nun ein weiteres Paket, so werden die fehlenden Bytes von diesem Paket genommen, da der Kontroller das Paket als solches nicht erkennt. Dieses weitere Paket ist nun zerstört und wird über die Fehlerroutine[3] verworfen. Ab Version 0.3.1 werden kleinere Pakete direkt eingelesen ohne DMA.

Regelmässig schaut die Alive-Routine nach, ob der DMA Kontroller nicht schon zu lange auf Daten wartet. Stellt die Alive-Routine dies fest, so stellt sie über eine Fehlerroutine einen definierten Zustand wieder her.

Fehlt dem System Speicher,[4] um das neue Paket aufzunehmen, wird es verworfen. Die Funktion flink_rx kann nicht auf Speicher warten oder es zu einen späteren Zeitpunkt erneut versuchen. Abbildung 4.2 zeigt diesen Fehlerfall der Routine „Socket Buffer besorgen“ als „Kein Socket Buffer bekommen“.

4.3.3 Sendefunktion *flink_tx*

Die Funktion kopiert das Datenpaket aus dem Socket Buffer in den TX Fifo. Dabei wird überprüft, ob physikalisch gesendet werden kann.

```
/* test if we are still sending */
if (test_and_set_bit(0,(void*) &privp->tx_busy) !=0 )
{/*we are still sending, try to handle  */
    ...
} /* end of test_and_set_bit */
```

Neben der Kontrolle, ob die Funktion nicht noch sendet, wird auch überprüft, ob das Paket noch in den TX Fifo paßt. Normalerweise leert sich der TX Fifo sofort durch das Versenden dessen Inhalts. Die Tests, ob gesendet werden kann oder nicht, werden in Abbildung 4.2 als die Fälle „sendebereit“ und „nicht sendebereit“ dargestellt. Der Einfachheit halber darf das Paket nicht größer als der TX Fifo sein.

Die Funktion *flink_tx* nutzt die Funktion *flink_hw_tx*, die den hardwarenahen Teil des Sendens erledigt.

Da dem IP Paket aus dem Socket Buffer unser Hardware Header fehlt, wird er vor dem IP Paket versendet. Die nun folgenden Sourcecode–Beispiele gehören zu der in Abbildung 4.2 gennanten Routine „Hardheader senden“.

```
/* send out the header */
outl(CTRL_SIGN3,dev->base_addr + AMCC_OP_REG_FIFO);
outl((long) dma_bytes,dev->base_addr + AMCC_OP_REG_FIFO);
```

[3] siehe obigen Sourcecode zum Kontrollzeichen
[4] siehe Anforderung des Socket Buffers

Noch zu erwähnen ist, daß die Paketlänge bestimmte Eigenschaften aufweisen muß. Die Eigenschaften sind:

- die Paketlänge ist ein Vielfaches von 4 Bytes
- die Paketlänge ist größer oder gleich der Mindestlänge von 48 Bytes

Erfüllt ein Paket nicht diese Eigenschaften, so erfolgt eine Korrekturmaßnahme.

```
/* we dont want to small packets  */
if (len < MIN_PKT_SIZE ) { len = MIN_PKT_SIZE ; }
dma_bytes=((len+3) & ~3);  /* need multiple of 4 byte*/
```

Die Sende- und Empfangseinheiten der Karte, d.h. auch der TX Fifo, arbeiten mit 32 Bit. Daher sollte die zu sendene Datenmenge ein Vielfaches von 4 Byte sein.

In einigen Treiberversionen[5] ist es notwendig, die Datenmenge im RX Fifo zu ermitteln. Dazu zählen alle Treiberversionen, die Packete ohne DMA empfangen können. Bei diesen Versionen wird in diesem Fall gewartet, bis das komplette Packet sich im RX Fifo befindet. Da aber erst ab 36 Bytes eine genaue Aussage möglich ist, wurde eine Mindestpaketgröße eingeführt.

4.4 Aktueller Status

Die letzte Treiberversion ist 0.3.1 Er erfüllt folgende Bedingungen:

- läuft unter einem 2.0.x Kernel
- läuft unter einem 2.2.x Kernel
- ist stabil
- hat eine vorzeigbare Performance
- unterstützt Interruptsharing

Es fehlen noch folgende Eigenschaften:

- FLink–Protokoll–support

[5] z.B. Treiber 0.3.1

4.4.1 Performance

Aus dem Internet wurde das Benchmark Tool „netperf[6]“ heruntergezogen. Auf der Homepage „*www.netperf.org*“ lassen sich Meßergebnisse von den verschiedensten Netzwerkkarten finden. Hauptziele von „netperf“ sind die rohen Datenübertragungsleistung und die „Request/Response Performance“ mit entweder TCP oder UDP und den Berkeley Sockets.

Die folgenden Diagramme 4.5 und 4.4 sind mit dem Kernel 2.2.13 und netperf der Version 2.1pl3 gemacht worden. Die Ergebnisse mit dem Kernel 2.0.36 liegen nur leicht darunter.

Abbildung 4.3 zeigt die aktuelle FLinkkarte im Vergleich mit einer Fast–Ethernetkarte. Natürlich sinkt der prozentuale TCP Transferleistung–Anteil mit steigender Gesamttransferleistung, was schon jetzt deutlich zu sehen ist.

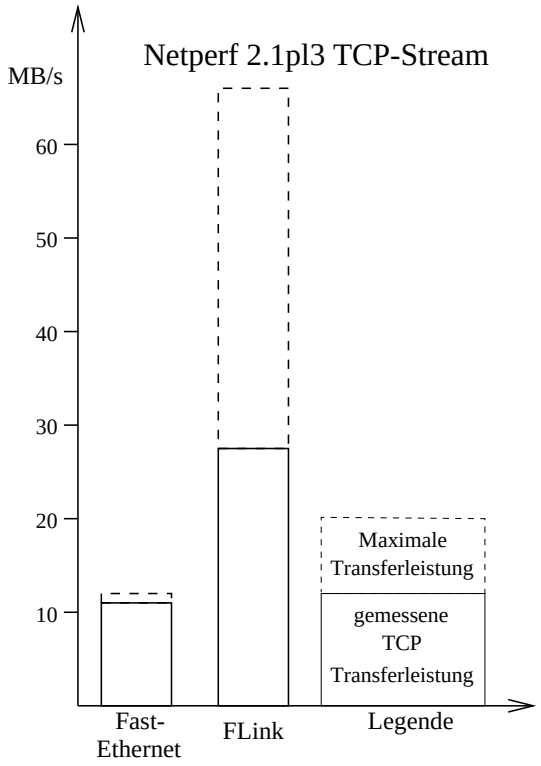


Abbildung 4.3: Leistungsvergleich

[6] Version 2.1pl3

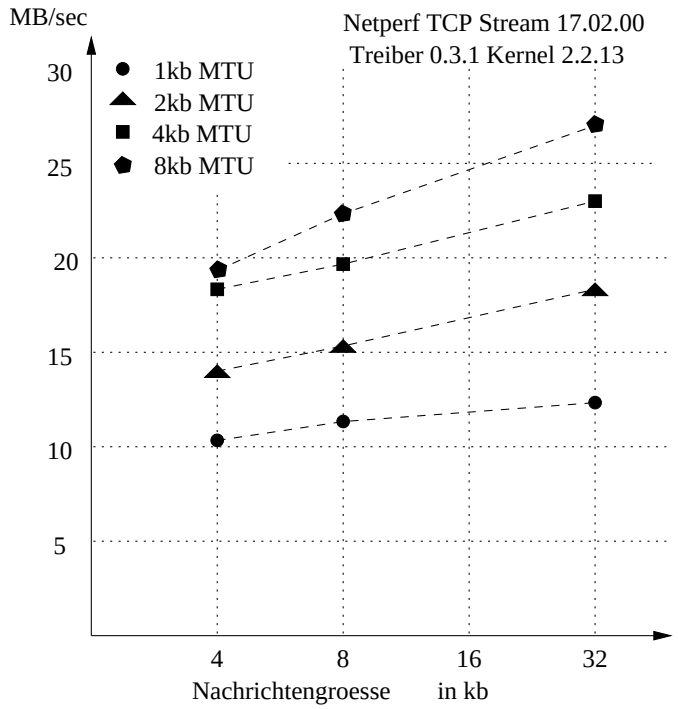


Abbildung 4.4: TCP Leistung

4.4.2 Stabilität

Ein selbstgeschriebenes Socketprogramm, welches einen bestimmten Speicherbereich ständig versandte, wurde nach 5 Tagen beendet. Der Treiber hatte die Version 0.2.7. Während dieser Zeit traten keinerlei Fehler auf. Diese Tatsache unterstreicht die Stabilität des Treibers trotz der Abstriche an Sicherheitskontrollen. Unter anderem wird nicht die Längeninformation auf gültige Angaben überprüft. Auch selten auftretenen Fehler, wie z.B. Paritätsfehler, können nur in bestimmten Fehlerroutinen erkannt werden.

4.4.3 Aussichten

Der Treiber (0.3.1) kann schon voll eingesetzt werden. Aufgrund der Beschränkungen des P2P Protokoll ist das FLinkprotokolls sehr wichtig für einen sinnvollen Einsatz.

Es gibt eine Menge Routinen, die der Stabilität des Treibers dienen sollen. Die Stabilität des Treibers ist entscheident für seine Brauchbarkeit. Anderseits bewirken zusätzliche Code–Zeile eine Verschlechterung der Performance. Deshalb ist kritisch zu überprüfen, ob die eine oder ander Routine entfernt oder hinzugefügt werden sollte.

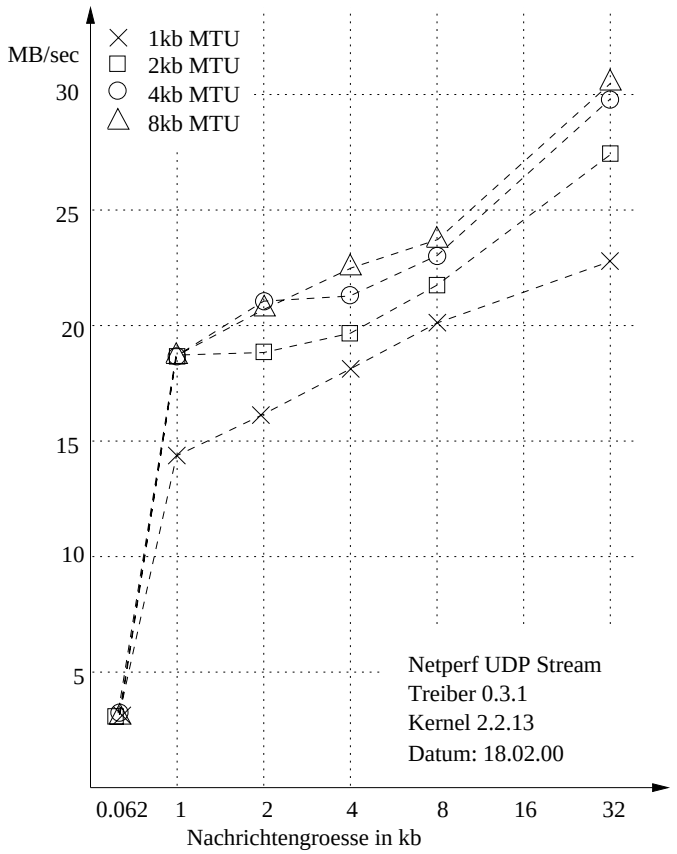


Abbildung 4.5: UDP Leistung

Möglicherweise ist die maximale Leistung für diese Karte noch nicht erreicht. Der Austausch von bestimmten Codefragmenten durch schnellere oder das Hinzufügen von neuen Ideen versprechen immer noch eine Performancesteigerung. Allerdings sind keine spektakulären Performancesteigerungen zu erwarten.

4.4.4 Schwächen und Verbesserungsvorschläge

Gerade für sehr kleine Pakete sinkt die Performance drastisch. Obwohl kleinere Pakete per E/A Zugriffe eingelesen werden, sind die Werte noch niedrig. Möglicherweise läßt sich durch eine neue Kombination der aktuellen Festlegungen bessere Werte erziehlen.

Da bei der Verwendung von Interrupts eine gewisse Latenzzeit entsteht, reagiert der Treiber etwas später, als durch reines Polling möglich wäre. Es wäre deshalb vorteilhafter, die Ereignisse durch Polling abzufragen. Eini-

ge Ereignisse, wie zum Beispiel das Mailboxflag, nutzen immer noch den Interrrupt.

Da der FLinktreiber das FLinkprotokoll nicht unterstützt, können hardwaremäßige unterstützende Funktionalitäten nicht genutzt[7] werden. Durch die Nutzung des FLinkprotokoll könnte deshalb mehr Performance gewonnen werden.

[7] durch PPP deaktiviert

Anhang A

Abkürzungen

ARP	Address Resolution Protocol
DESY	Deutsches Elektronen–Synchrotron
FLink	Fast Link
ggf.	gegebenenfalls
INFN	Istituto Nazionale di Fisica Nucleare
IP	Internet Protokoll
kb	kilobyte
MB	Megabyte
MB/s	Megabyte je Sekunde
Mbits	Millionen Bits
MTU	Maximum Transfer Unit
P2P	Peer–to–Peer Protokoll
PPP	Point–to–Point Protokoll
PC	Personal Computer
sk_buff	Socket Buffer
TCP	Transmission Control Protocol
u.a.	unter anderen
UDP	User Datagram Protocol
usw.	und so weiter
z.B.	zum Beispiel
z.Z.	zur Zeit

Literaturverzeichnis

[1] Alessandro Rubini: *Linux Device Drivers* O'Reilly & Associates, 1998, ISBN: 1-56592-292-1

[2] Applied Micro Circuits Corporation (AMCC): *PCI PRODUCTS DATA BOOK S5920 /S5933*, 1998

[3] David A. Rusling *The Linux Kernel*, Version 0.8-3, 25.1.99, GNU Documentation Project

[4] DEUTSCHES ELEKTRONEN–SYNCHROTRON: *FLINK, Hochgeschwindigkeits Interface für die Vernetzung von PC's*, DESY INFORMATION, DESY Zeuthen

[5] DEUTSCHES ELEKTRONEN–SYNCHROTRON: *Auf dem Wege ins Teraflop–Zeitalter mit APEmille, Massiv–Paralleles Rechnen bei DESY Zeuthen*, DESY INFORMATION, DESY Zeuthen

[6] Edward Solari, George Willse: *PCI Hardware and Software*, Zweite Auflage, 1994 Annabooks, ISBN: 0-929392-19-1

[7] K.–H. Sulanke: *FLINK – Point to Point Protocol, short description*, Rev. 1.1, 25. October 1999, DESY Zeuthen

[8] Karl–Heinz Sulanke *FLINK – short description*, Rev. 1.5, 20. September 1999, DESY Zeuthen

[9] Tom Shanky, Don Anderson *PCI System Architekture*, Vierte Auflage, 1999 Addison Wesley, ISBN: 0-201-30974-2

[10] W. Richard Stevens *UNIX Network Programming*, 1990 Prentice Hall, ISBN: 0-13-949876-1

Anhang B

Kernel–Funktionen

Treiber können nicht die Bibliotheken der Anwenderprogramme nutzen. Es werden Funktionen benötigt, die im Kernelspace bestimmte Aufgaben erledigen. Im folgenden werden einige dieser Funktionen kurz beschrieben.

B.1 Print–Funktionen

B.1.1 printk()

Mit `printk()` können Nachrichten auf der Console ausgegeben werden, ähnlich dem `printf()`. Ein Beispiel:

```
printk(KERN_WARNING "Eine Warnung \n");
```

Zusätzlich kann eine Priorität mitangegeben werden. Die Console kann so konfiguriert werden, das nur Nachrichten ab einem bestimmten Prioritätslevel ausgegeben werden.

B.2 Bitoperationsfunktionen

B.2.1 test_and_set_bit(), clear_bit()

Mit diesen Funktionen können Bits atomar verändert werden. `test_and_set_bit()` liefert als Ergebnis den alten Wert des veränderten Bits zurück. Ein Beispiel:

```
unsigned long flag = 0; /* 32 Bit */
unsigned int oldvalue;
```

```
/* das 6. Bit soll den Wert 1 enthalten */
oldvalue = test_and_set_bit(5,flag);
/* das 6. Bit soll den Wert 0 enthalten */
clear_bit(5,flag);
```

B.3 E/A–Funktionen

B.3.1 inl(), outl(), insl(), outsl()

Mit diesen Funktionen kann auf I/O–Portadressen[1] zugegriffen werden und 32 Bit lange Wörter gelesen bzw. geschrieben werden. Bei diesen Funktionen ist Vorsicht zu genießen. Da diese Funktionen direkt vom Prozessor ausgeführt werden, führt das Lesen auf eine Portadresse, die kein 32 Bit langes Wort liefert, zum „Ewigen Warten“ des Prozessor.

Sind mehrere hintereinanderliegende[2] 32 Bit lange Wörter zu empfangen oder zu senden, ist mit `inl()` und `outl()` eine Schleife notwendig. Die Befehle `insl()` und `outsl()` realisieren diesen Schleife. Ein Beipiel zur Verwendung von `inl()` und `outl()`:

```
unsigned long status;

/* Angenommen unsere Portadresse ist 0x330h */
status=inl(0x330); /* lesen */
outl(status,0x330); /* schreiben */
```

B.4 PCI–Funktionen

Zwischen den Kernelversionen 2.0.x und 2.2.x wurden die PCI–Funktionen überarbeitet. Die PCI–Funktionen im 2.0.x Kernel identifizieren ein PCI–Gerät durch Busnummer, Gerätenummer und Funktionsnummer. Diese Nummern werden in den Sourcecodebeispielen in den Variablen *pci_dev* und *pci_device_fn* gespeichert. Im 2.2.x Kernel nutzen die PCI–Funktionen die Struktur *pci_dev*.

B.4.1 pcibios_present(), pci_present()

Der Treiber kann mit diesen Funktionen die Anwesentheit eines PCI–System feststellen. In Abhängigkeit der verwendeten Kernelversion wird die eine oder die andere Funktion genutzt. Beide Funktionen liefern 0 im Fehlerfalle zurück.

[1] E/A Adressen
[2] gemeint ist die Organisation im Speicher

```
/* this code is for kernelversion 2.0.x ! */
if( pcibios_present() != 0 ) {
    /* PCI system exist */
} else {
    /* No PCI system found */
}
```

B.4.2 pcibios_find_device(), pci_find_device()

Die Funktionen *pcibios_find_device()* und *pci_find_device()* suchen nach einem PCI–Gerät mit dem die Herstellernummer und Gerätenummer übereinstimmen. Bei *pcibios_find_device()* werden im Erfolgsfalle die Busnummer, Gerätenummer und Funktionsnummer in den Variablen geschrieben.

```
/* this code is for kernelversion 2.0.x ! */
unsigned short vendor_id,device_id,dev_found;
unsigned char pci_bus,pci_device_fn;
unsigned int ret_value;
ret_value = pcibios_find_device(vendor_id,device_id,dev_found,
                                &pci_bus,&pci_device_fn);
/* On success ret_value = PCIBIOS_SUCCESSFUL  */
```

Die Funktion *pci_find_device()* nutzt eine Variable der Struktur *pci_dev*, um die Informationen aus dem PCI–Register zu speichern.

```
/* this code is for kernelversion 2.2.x ! */
struct pci_dev *dev = NULL;
dev = pci_find_device(VENDOR_ID,DEVICE_ID,dev);
```

B.4.3 PCI–Konfigurationsadressraum benutzen

Unter Kernel 2.0.x sind die Funktionen

- pcibios_read_config_byte()
- pcibios_read_config_word()
- pcibios_read_config_dword()
- pcibios_write_config_byte()
- pcibios_write_config_word()
- pcibios_write_config_dword()

definiert. Mit Hilfe dieser Funktionen können die Felder des PCI–Konfigurationsregisters ausgelesen und ggf. auch beschrieben werden. Entsprechend ähnliche Funktionen existieren auch unter Kernelversion 2.2.x, aber die Informationen lassen sich leichter aus der Strukur *pci_dev* auslesen.

B.5 Ressourcenmanagment–Funktionen

B.5.1 kmalloc(), kfree()

Treiber benötigen Speicher mit den verschiedensten Eigenschaften. Zum Beispiel dürfen bestimmte Speicherbereiche nicht auslagerungsfähig sein oder sich in einem physikalischen linearen Adreßraum befinden. Für diese speziellen Anforderungen werden andere Speicherbereitstellungsfunktionen benötigt, als für Anwendungsprogramme.

B.5.2 E/A Bereiche und Interrupte

E/A-Adressen und IRQ's sind begrenzte Ressourcen. Mit Hilfe dieser Funktionen, `request_region()`, `request_irq()`, `release_region()`, `free_irq()`, realisiert der Kernel Listen über die verbrauchten Ressourcen. Diese Listen können z.B. mit `cat /proc/ioports` betrachtet werden.

B.5.3 Timer–Funktionen

Einige Treiber benötigen einen Timer, um eine Funktion nach einer gewissen Zeit zu starten. Der Kernel realisiert zwei nutzbare Timermechanismen. Der neue Timermechanismus des Kernel (Abbildung B.1) benutzt eine verkettete Liste der *timer_list* Datenstruktur. Die Liste ist in aufsteigender Reihenfolge der Ablaufzeiten (expire time) sortiert. Als Zeiteinheit wird jiffies verwendet.

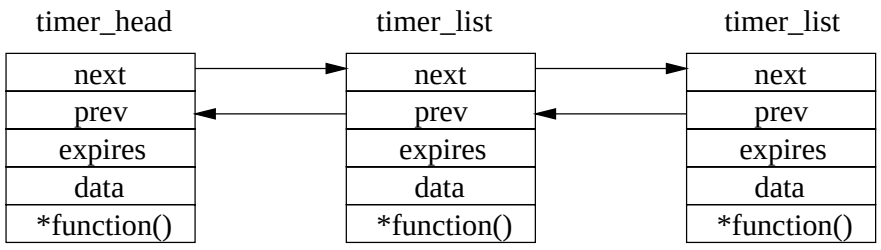


Abbildung B.1: Timerliste

init_timer(), add_timer(), del_timer()

Damit der Treiber den Timermeachnismus des Kernels nutzen kann, benötigt er ein Listenelement des Typ's *timer_list*. Mit *init_timer()* wird das Element initialisiert. Die Funktion *add_timer()* fügt das Element in die Timer–Liste ein und *del_timer()* löscht den Eintrag aus der Timer–Liste. Beim Eintreten der Ablaufzeit wird das Element aus der Liste gelöscht und die eingetragende Funktion aufgerufen.

B.6 Treiberverwaltungsfunktionen

B.6.1 MOD_DEC_USE_COUNT, MOD_INC_USE_COUNT

MOD_DEC_USE_COUNT, MOD_INC_USE_COUNT sind Macros die den *Usage–Counter* herauf- oder herabzählen. Üblicherweise werden diese Macros in den *open()* oder *close()* Funktionen modularer Treiber benutzt. Es soll damit verhindert werden das ein Modul entladen werden kann, während es noch benutzt wird.

B.6.2 register_netdev(), unregister_netdev()

Diese Funktionen werden vom Treiber benutzt, um ein Netzwerkinterface beim Kernel an- oder abzumelden.

Anhang C

Sourcecode

C.1 flink.c

```
/*
 * flink.c --  a Simple Network Driver for the Flink card
 * Ethernet-less
 *
 */

#ifndef __KERNEL__
#  define __KERNEL__
#endif
#ifndef MODULE
#  define MODULE
#endif

#define __NO_VERSION__ /* don't define
                          kernel_version in module.h */
#include <linux/module.h>
#include <linux/version.h>

char kernel_version [] = UTS_RELEASE;

#include <linux/sched.h>
#include <linux/kernel.h> /* printk() */
#include <linux/malloc.h> /* kmalloc() */
#include <linux/errno.h>  /* error codes */
#include <linux/types.h>  /* size_t */
#include <linux/interrupt.h> /* mark_bh */
```

```
#include <linux/netdevice.h>  /* struct device,
                                 and other headers*/
#include <linux/etherdevice.h>/* eth_type_trans */
#include <linux/ip.h>         /* struct iphdr */
#include <linux/tcp.h>        /* struct tcphdr */
#include <linux/skbuff.h>     /* struct sk_buff */

#include <linux/config.h>     /* PCI Funktions */
#include <linux/pci.h>
#if (LINUX_VERSION_CODE < 0x20155)
#include <linux/bios32.h>
#endif
#include <asm/dma.h>
#include <asm/io.h>   /* for inb() outb() etc. */
#include <asm/irq.h>  /* for enable_irq() disable_irq() */
#include <asm/bitops.h>
#include <asm/system.h> /* interrupts,like cli()*/
#if (LINUX_VERSION_CODE < 0x20155)
#include <asm/checksum.h>
#endif
#include <linux/ioport.h> /* request_region()
                             release_region()*/
#include <linux/delay.h>      /* for udelay() */
#include <linux/if_arp.h>     /* for ARPHRD_ ? */

#ifdef ALTERA_INIT
#include "flink.ttf" /* initialisationcode
                        for the altera chip */
#endif               /* attention! it's not
                        really Kalle's file */
#include "flink.h"
#ifndef MAX_MTU
# define MAX_MTU 8192 /*8k */ /* bytes */
#endif
#ifndef CTRL_SIGN3    /* !!! 32 bit */
# define CTRL_SIGN3 0xa5b4c3d2 /* */
#endif
#ifndef FLINK_ALIVE_TIMEDOUT
# define FLINK_ALIVE_TIMEDOUT 1000 /* jiffies */
#endif
#ifndef FLINK_TIMEDOUT
# define FLINK_TIMEDOUT 500 /* Loops with 1 usec Periode */
#endif
```

```
/* define the minimum length for a packet*/
#ifndef MIN_PKT_SIZE
# define MIN_PKT_SIZE 48  /* 48 byte == 12 words(32) */
#endif
#ifndef DMA_SIZE   /* handle with care,
                      use 0 to deaktivate */
# define DMA_SIZE 100  /* byte */
#endif
#if (LINUX_VERSION_CODE < 0x20123)
#define test_and_set_bit(val,addr) set_bit(val,addr)
#endif
#if (LINUX_VERSION_CODE > 0x20139)
#define enet_statistics net_device_stats
#endif
#if (LINUX_VERSION_CODE < 0x20159)
#define DEV_FREE_SKB(skb) dev_kfree_skb(skb,FREE_WRITE)
#else
#define DEV_FREE_SKB(skb) dev_kfree_skb(skb)
#endif
#if (LINUX_VERSION_CODE < 0x20155)
#define LINUX_20X
#endif

#ifdef DEBUG
static int debug = 15;
/* bit value debugmode
 * 0    1     DMA
 * 1    2     Sending
 * 2    4     Receiving
 * 3    8     interrupt
 * 4   16     rx-packetdata
 * 5   32     tx-packetdata
 * 6   64     time-measure
 * 7  128     not used
 */
#endif

/* prototyps */
/*
 * this function deals with the receiving of data
 * it will be invoken from the interrupt-handle routine
 *
 */
```

```
int flink_rx(struct device *dev);

/*
 * this function handle the interrupts
 * called by kernel
 */
void flink_interrupt(int irq, void *dev_id,
                     struct pt_regs *regs);

/*
 * this function calls the kernel
 * when someone wants to use the
 * device , typically 'ifconfig ethX up'
 *
 */
int flink_open(struct device *dev);

/*
 * this funtion is the opposite of flink_open()
 *
 */
int flink_release(struct device *dev);

/*
 * this function is if someone
 * want to configure the
 * the card, but there isn't really
 * something to configure
 *
 */
int flink_config(struct device *dev, struct ifmap *map);

/*
 * this function deals with the hardwaredetails,
 * if someone wants to sent something
 *
 */
int flink_hw_tx(long *buf, int len, struct device *dev);

/*
 * this function is called by
 * the kernel if someone want to
 * send something
```

```
 *
 */
int flink_tx(struct sk_buff *skb, struct device *dev);

/*
 * this function is called by the kernel if someone
 * wants to start a io-command,
 * this function does nothing
 * as printing the command and return
 *
 */
int flink_ioctl(struct device *dev, struct ifreq *rq,
                int cmd);

/*
 * this function allows the kernel
 * to update the drivers statics
 *
 */
struct enet_statistics *flink_stats(struct device *dev);

/*
 * this function allows somebody to change the mtu,
 *
 */
int flink_change_mtu(struct device *dev, int new_mtu);

/*
 * this function is called by init_module(),
 * its search for the flink cards, until the maximum
 * number of cards are reached or no more cards are
 * found. typicall are one or two cards.
 *
 */
int flink_probe(struct device *dev);

/*
 * this function is called by flink_probe().
 * it does the mainpart of configuration.
 * in this function the altera chip is initialised.
 *
 */
static struct device *flink_found(struct device *dev,
```

```
                                    int baseaddr0,
                                    int baseaddr1,
                                    int baseaddr3,
                                    int number);

/*
 * this function is called by the kernel
 * if the driver is loaded
 *
 */
int init_module(void);

/*
 * this function is called by the kernel
 * if the driver is unloaded
 *
 */
void cleanup_module(void);

/* this function is a fake */
int init_flink(struct device *dev);

/* the alive routine */
void alive(unsigned long data);


/*
 * In this structure the missing items
 * in the device-structure
 * are stored.
 */
struct flink_priv {
  struct enet_statistics stats; /* for the statistics */
  struct sk_buff *rx_skb; /* to remember to socketbuffer */
  struct sk_buff *tx_skb; /* to remember to socketbuffer */
  int txerror;        /* to know if we reentering a error
                         situation (exclude deadlooks)*/
  u32 pld_addr;       /* to know the offset of the
                                  Altera-chip */
  struct flink_priv *next;      /* nextpointer */
  struct device  *dev;          /* devicepointer */
  unsigned int tx_busy;  /* to know if we are sending */
  struct timer_list timer;      /* rx-timer */
```

```
  unsigned long rx_dma_start; /* to know the
                                 last start of the
                                   rx-dma */
};

/* A list of all installed devices,
for removing the driver module. */
static struct flink_priv *flink_card_list = NULL;

/* this function is needed for register_netdev */
int init_flink(struct device *dev)
{
  return 0;
}

/* the alive routine, should the driver keep alive */
void alive(unsigned long data)
{
  struct device *dev = (struct device *) data;
  struct flink_priv *privp = (struct flink_priv *) dev->priv;
  unsigned long status_register;

  /* if we start a DMA, we rescue the
   * socketbufferpointer (skb)
   * and clear privp->rx_skb after succesful DMA.
   * privp->rx_dma_start is
   * updated by every DMA start.
   * So privp->rx_dma_start is for
   * this  routine only valid with a rescued skb !
  */
  if((jiffies > (privp->rx_dma_start + FLINK_RX_TIMEDOUT))
      && (privp->rx_skb !=NULL))
  {
  /* What's happend ?
   wrong programmed DMA-Controller?
   transmit abort? or bad hardware?
  */
  /* making clean status, but if the rest comes
  --> JAM ON LINE     */
    printk(KERN_WARNING "RX-Timedout, whats wrong?\n");
    DEBUGMSG("RX-TC=%d bytes \n",
             inl(dev->base_addr + AMCC_OP_REG_MWTC));
    DEBUGMSG("RX-address stand by %08lx \n",
```

```
            inl(dev->base_addr + AMCC_OP_REG_MWAR));
    /* we want to stop the DMA-machine,
     * but if the counter is
     * zero we got a false interrupt,
     * so extract the
     * permission for INTR and stop the controller!! */
    status_register=inl(dev->base_addr + AMCC_OP_REG_INTCSR);
    /*exclude INTR_WRITE_TR_COMPL */
    status_register=(status_register & 0xbfff);
    outl(status_register|ENABLE_INTR_IN_MAILBOX_1_BYTE_0,
         dev->base_addr + AMCC_OP_REG_INTCSR);
    outl(0,dev->base_addr + AMCC_OP_REG_MWTC); /* Stop it*/
    privp->stats.rx_errors++; /* update errors */
    DEV_FREE_SKB(privp->rx_skb); /* free it */
    privp->rx_skb=NULL;
    /* enable interrupt line from Altera-chip to AMCC-chip */
    /* Int_enable */
    outl(ICR_INTR_ENABLE,privp->pld_addr + PLD_OP_REG_ICR);
  }
  init_timer(&privp->timer); /* generate a timerfeld */
  /* fill out the necessary for the timer */
  privp->timer.function=&alive; /* timedout function */
  /*  data=32bit-value;dev=32bit-pointer */
  privp->timer.data = (unsigned long) dev;
  /* when ? */
  privp->timer.expires = jiffies + FLINK_ALIVE_TIMEDOUT;
  add_timer(&privp->timer); /* start the timer */
  return;
}

/*
 * Receive a packet: retrieve, encapsulate and
 * pass over to upper levels
 */
int flink_rx(struct device *dev)
{
  struct sk_buff *skb;
  struct flink_priv *privptr = (struct flink_priv *)dev->priv;
  u_char *data ;
  int len = 0;                /* length in bytes */
  int again = 1;
  int received, dma_words;
  /* a header  is sended before the
```

```
   * real packet, it's contain the length of the packet.
   * in this version of header the first dword
   * contain the Control sign and
   * the second contains the length (in bytes)
   *
   * 31                   0 31                 0 bit
   *  |--------------------||------------------|
   *   Control Sign                Length
   *
  */
  int header=0; /* header */

  /*
   * The packet has been retrieved from the transmission
   * medium. Build an skb around it, so upper layers can
   * handle it
  */
#ifdef DEBUG
  if (debug & 4) {
    DEBUGMSG("rx: Entry ............\n");
  }
#endif
  /* The try to read an empty RX-FIFO blocks the CPU,
     so be extra carefull.
  */
  if ( inl(dev->base_addr + AMCC_OP_REG_MCSR)
      & MCSR32_A2P_FIFO_EMPTY)
  {
    printk(KERN_WARNING "%s NO Data  !!! \n",dev->name);
    return 1; /* failure */
  }
  /* the beginning of a new packet */
  /* now we get our header,
   */
  /* first 32bit, the Control-sign*/
  header=inl(dev->base_addr + AMCC_OP_REG_FIFO);
  if((header - CTRL_SIGN3 ) != 0)
  { /* this test for jam */
    unsigned long status_register;
    int jam = 1;         /* Numbers of Jam-words(32) found */
    printk(KERN_WARNING "%s: Jam on the line \n",dev->name);
    again = 1; /* nessesary to set it again? */
    while (again) {
```

```
printk(KERN_WARNING "%s: %08lx \n",dev->name,header);
/* test if there is a bigger (hardware) problem */
status_register=inl(privptr->pld_addr + PLD_OP_REG_ISR);
if ( (status_register & ALIGNMENT_ERROR)
    || (status_register & PARITY_ERROR))
{
  printk(KERN_WARNING
         "%s: Hardware Problem (ISR) %08lx\n",
         dev->name,status_register);
  /* clear the big RX-FIFO */
  /* should clear it  */
  outl(CLEAR_RX_FIFO,privptr->pld_addr + PLD_OP_REG_ICR);
  * switch off (rx-reset) */
  outl(CLEAR_RX_BLOCKING,
       privptr->pld_addr + PLD_OP_REG_ICR);
  /* this should reset the internal (AMCC) FIFO,
   * we must do it
   * because we can find a Headersign
   * and no correct length-word(32)*/
  /* clear the little RX-FIFO */
  outl(MCSR32_A2P_FIFO_RESET|MCSR32_READ_TR_ENABLE,
  dev->base_addr + AMCC_OP_REG_MCSR);
}
/* the jam serie is (hopefully) over */
if( inl(dev->base_addr + AMCC_OP_REG_MCSR)
    & MCSR32_A2P_FIFO_EMPTY)
{
  printk(KERN_WARNING
         "%s: NO MORE DATA after %d Jam-Words(32)\n",
         dev->name,jam);
      return 1;
}
/* read the next word(32) */
header=inl(dev->base_addr + AMCC_OP_REG_FIFO);
jam++; /* count it */
/* is it our sign for a new packet?? */
if((header - CTRL_SIGN3 ) == 0 )
{
  printk(KERN_WARNING
         "%s: Have eaten %d Jam-words(32) \n",
         dev->name,jam-1);
  again = 0; /* stop the Jam-routine */
}
```

```
    }
    /* If this condition was true
     * we have jam on the line,
       so ignore it and return */
  }
  again=1;
  /* Normally after the headersign
   * follow the length-word(32), but
   * what would be if the length-word(32) is missing? */
  if( inl(dev->base_addr + AMCC_OP_REG_MCSR)
      & MCSR32_A2P_FIFO_EMPTY)
  {
    while(again > 0) {
      if(again > FLINK_TIMEDOUT)
      {   /* death line */
        printk("NO length word(32)\n");
        return 1;    /* failure */
      }
      if( inl(dev->base_addr + AMCC_OP_REG_MCSR)
          & MCSR32_A2P_FIFO_EMPTY)
      {   /* Nothing avaiable */
        again++;
        udelay(1);
      } else
      {   /* just arived */
        again=0;
      }
    }
  }
  len=inl(dev->base_addr + AMCC_OP_REG_FIFO);
  /* maybe we should controll, if we get a realistic
  length-information, but we didnt do it yet*/
#ifdef DEBUG
  if (debug & 4) {
    DEBUGMSG("lenght is %i bytes\n",len);
  }
#endif
  /* alloc the a socketbuffer */
  skb = dev_alloc_skb(len);
  if (!skb) { /* havent got a socketbuffer */
    printk(KERN_WARNING "flink rx: low on Memory\n");
    /* What to do, we have read the header and coulnd
    store the packet. we coulnd reenter the routine so
```

```
  throw the packet away. and we coulnd use the dma
  controller, because we woulnd save it.
  */
  while (len > 0)
  {
    /* dont forgot to check if data is available */
    if ( inl(dev->base_addr + AMCC_OP_REG_MCSR)
         & MCSR32_A2P_FIFO_EMPTY)
    {
      printk(KERN_WARNING
             "%s Data misssing !!! \n",dev->name);
      len=0;
    } else {
      inl(dev->base_addr + AMCC_OP_REG_FIFO);
      len=len - 4;
    }
  }
  return 1; /* failure */
}
/* fill the socketbuffer with information */
skb->dev = dev;
/* this updates the socketbuffer, we get a pointer
   were to write the packet */
data= (u_char *) skb_put(skb,len);
if (len < DMA_SIZE)
{ /* this should be better received by CPU-IO */
  again=1;
  dma_words = len >>2; /* we needs the amount of words*/
  DEBUGMSG(" DMA not needed, reading by CPU \n");
  while(again > 0)
  {
    received = ((int) (inl(privptr->pld_addr + PLD_OP_REG_GRX)
                & 0x00000FFF));
    received = received + 8; /* this is wrong
                                if received is 0,
                                but thats why
                                a MIN_PKT_SIZE exist */
    if (again > FLINK_TIMEDOUT)
    { /* Giving up */
      printk(KERN_WARNING
             "flink: RX: NO DMA: What's wrong? Time over! \n");
      return 1; /* failure */
    }
```

```
      else {
        udelay(1); /* 1 microsecond, too much? */
        again++;
      }
      if(dma_words < received || dma_words == received )
      {
        DEBUGMSG("RX: NO DMA: %i Loops occured with 1 usec \n",
                 again - 1);
        again = 0;
      }
    }
    insl(dev->base_addr + AMCC_OP_REG_FIFO,
         (long *) data,dma_words);
    privptr->stats.rx_packets++;
    /* some necessary things */
    skb->protocol = htons(ETH_P_IP);
    skb->mac.raw = skb->data;
    /* maybe this can left, because it's the defaultvalue */
    skb->pkt_type = PACKET_HOST;
    /* we believe in our hardware*/
    skb->ip_summed = CHECKSUM_UNNECESSARY;
    netif_rx(skb); /* inform the upper layers */
    return 1; /* No DMA Controller programmed */
  }
  /* program the DMA controller */
  /* address */
  outl(virt_to_bus(data),dev->base_addr + AMCC_OP_REG_MWAR);
#ifdef DEBUG
  if (debug & 1) {
    DEBUGMSG("DMA rx\n");
    DEBUGMSG("RX: skb=%08lx \n",skb);
    DEBUGMSG("RX: TARGET=%08lx \n",data);
    DEBUGMSG("RX: Read from AMCC= %08lx \n",
             inl(dev->base_addr + AMCC_OP_REG_MWAR));
  }
#endif
  outl(len,dev->base_addr + AMCC_OP_REG_MWTC); /* amount */
  privptr->rx_skb=skb; /* rescue skb pointer */
  privptr->rx_dma_start=jiffies; /*notify when DMA started*/
  return 0; /* it's all for now, */
}   /* end of flink_rx() */

/*
```

```
 * The typical interrupt entry point (1.3.70 and later)
 */
void flink_interrupt(int irq, void *dev_id,
                     struct pt_regs *regs)
{
#ifdef DEBUG
  unsigned long statusword;
  unsigned long mailbox;
  unsigned long time = jiffies;
#endif
  unsigned long int_status;
  struct flink_priv *privptr;
  int ret_value;

  /* extract our device pointer */
  struct device *dev = (struct device *) dev_id;

#ifdef DEBUG
  if (debug & 8) {
    DEBUGMSG("interrupt entry...\n");
  }
#endif
  if (!dev ) { /* Are we paranoid? */
    printk(KERN_WARNING
           ": (flink_interrupt) No Devicepointer found\n");
    return;
  }

  /* our privat pointer */
  privptr    =   (struct flink_priv *)(dev->priv);
#ifdef DEBUG
  statusword =   inl(dev->base_addr|AMCC_OP_REG_MCSR);
  mailbox    =   inl(dev->base_addr | AMCC_OP_REG_MBEF);
#endif
  int_status =   inl(dev->base_addr | AMCC_OP_REG_INTCSR);

  /* Normally we must only controll for INTR_ASSERTED, but
   * the docu (of AMCC) means something other.
   * So we are carefull and do what the docu means.
  */
  if (  int_status & INTR_ASSERTED ||
        int_status & INTR_MASTER_ABORT ||
        int_status & INTR_TARGET_ABORT )
```

```
  {
#ifdef DEBUG
    if (debug & 8) {
      DEBUGMSG("interrupt handler, get status: %08lx \n"
               ,statusword);
      DEBUGMSG("interrupt handler, interruptstatus: %08lx \n"
               ,int_status);
      DEBUGMSG("interrupt handler, Mailboxstatus: %08lx \n"
               ,mailbox);
    }
#endif
    /* for measure, else dont need it */
    /* so uncommend the next line, if you need it */
    /* inl(privptr->pld_addr + PLD_OP_REG_ICR); */
    if (int_status &  INTR_WRITE_TR_COMPL)
    {
      /* just complete with copy into memory */
      /* now restore old interrupt register
         (this means )
         the lower 16bits -
         Interrupt on write transfer compl. */
      int_status = (int_status & 0xbfff); /*filter selection*/
      /* clear the INTR_WRITE_TR_COMPL-flag
       + restore old value + enable Mailbox */
      outl(INTR_WRITE_TR_COMPL|int_status
           |ENABLE_INTR_IN_MAILBOX_1_BYTE_0,
           dev->base_addr + AMCC_OP_REG_INTCSR);
      privptr->stats.rx_packets++;
      /* some necessary things */
      privptr->rx_skb->protocol = htons(ETH_P_IP);
      privptr->rx_skb->mac.raw = privptr->rx_skb->data;
      /*maybe this can left, beacaus it's the defaultvalue*/
      privptr->rx_skb->pkt_type = PACKET_HOST;
      /* we believe in our hardware*/
      privptr->rx_skb->ip_summed = CHECKSUM_UNNECESSARY;
      netif_rx(privptr->rx_skb); /*inform the upper layers*/
      privptr->rx_skb=NULL; /* unset it */
      /* interruptline from PLD-chip over AMCC-chip */
      /* set "lost" Intr */
      outl(ICR_INTR_ENABLE,
           privptr->pld_addr + PLD_OP_REG_ICR);
      return;
    }
```

```
    if (int_status  & INTR_IN_MAILBOX) {
      /* exclusive use of mailbox for incoming data */
      /* okay new packets arrived */
#ifdef DEBUG
      if (debug & 8) {
        DEBUGMSG(" interrupt: new data arrived \n");
      }
#endif
      /* mark that  an interrupt  will be served */
      if (test_and_set_bit(0,(void *) &dev->interrupt) !=0) {
        printk( KERN_ERR "Reentering Interruptroutine \n");
        return;
      }
      ret_value = flink_rx(dev);
      inl(dev->base_addr + AMCC_OP_REG_IMB1); /*dummy read*/
      if (ret_value == 1) {
        /* No DMA started, maybe a failure occured? */
        /* now restore old interrupt register
           all interrupts are allowed */
        int_status = (int_status & 0xffff);/*filter selection*/
        outl(INTR_IN_MAILBOX|int_status,
             dev->base_addr + AMCC_OP_REG_INTCSR);
             /* Int_enable for PLD-chip*/
        outl(ICR_INTR_ENABLE,
             privptr->pld_addr + PLD_OP_REG_ICR);
      }
      else { /* DMA started, normall case */
        /* now restore old interrupt register
           please no new packet to receive, so disable it*/
        int_status = (int_status & 0xefff); /*filter selection*/
        outl(INTR_IN_MAILBOX|int_status
             |ENABLE_INTR_WRITE
             ,dev->base_addr + AMCC_OP_REG_INTCSR);
        /* don't enable INTR for PLD-chip before this packet is
           saved! */
      }
#ifdef DEBUG
      if (debug & 8) {
        DEBUGMSG("leaving interrupt new data\n");
      }
#endif
      clear_bit(0,(void*) &dev->interrupt);
#ifdef DEBUG
```

```
      if (debug & 64)
      {
        DEBUGMSG("INTR: mailbox timestart %i timediff %i \n"
        ,time, jiffies - time);
      }
#endif
      return;
    }
    /* this should clear Master-abort or Target-abort */
    outl(int_status,dev->base_addr + AMCC_OP_REG_INTCSR );
    /* did we forget anything ?*/
    printk(KERN_WARNING
           " interrupt: Unknown int +++ INTCSR %08lx \n"
           ,int_status);
    return;
  } else {
    /* this interrupt is not for us */
    return;
  }
}

/*
 * Open and close
 */

int flink_open(struct device *dev)
{
#ifdef DEBUG
  unsigned long statusword;
  unsigned long int_status;
#endif
  struct flink_priv *privptr= dev->priv;

  /* request_region(), request_irq(), ....   */
  /* just do */
  request_region(dev->base_addr, BASE_ADDR0_RANGE, dev->name);
  /* Did anybody can take our IO-addr?
   * I think no, so I don't reserve the other IO-addr!
   * We will see.
  */
  DEBUGMSG("request region succesful\n");
  if (request_irq(dev->irq, flink_interrupt,
      SA_SHIRQ, dev->name, dev)) {
```

```
    printk(KERN_WARNING
           "flink: Can't install handler for irq %d\n"
           ,dev->irq);
    /* Region free */
    release_region(dev->base_addr, BASE_ADDR0_RANGE);
    return -EAGAIN;
  };
  DEBUGMSG("request irq succesful, IRQ %d\n",dev->irq);
  dev->start = 1;
  dev->tbusy = 0;
  MOD_INC_USE_COUNT;
  /* this should look periodical if anything is good */
  init_timer(&privptr->timer); /* generate a timerfeld */
  /* fill out the necessary for the timer */
  privptr->timer.function=&alive; /* timedout function */
  /*  data=32bit-value;dev=32bit-pointer */
  privptr->timer.data = (unsigned long) dev;
  /* when? */
  privptr->timer.expires = jiffies + FLINK_ALIVE_TIMEDOUT;
  add_timer(&privptr->timer); /* start the timer */
  /* now we enable the PLD-chip to interrupt us
   * if there is a interrupt, normally the
   * interrupt of the pld-chip will be blocked
  */
  outl(ENABLE_INTR_IN_MAILBOX_1_BYTE_0,
       dev->base_addr + AMCC_OP_REG_INTCSR);
#ifdef DEBUG
  /* get some information for debuging */
  statusword = inl(dev->base_addr + AMCC_OP_REG_MCSR);
  int_status = inl(dev->base_addr + AMCC_OP_REG_INTCSR);
  DEBUGMSG("open_method, MCSR status: %08lx \n",statusword);
  DEBUGMSG("open_method, INTCSR status: %08lx \n",int_status);
#endif
  return 0;
}

int flink_release(struct device *dev)
{
  struct flink_priv *privptr= dev->priv;

  /* release io's, irq and such things */
  del_timer(&privptr->timer); /* kill alive-timer */
  /* no intr any more */
```

```
  outl(0l,dev->base_addr + AMCC_OP_REG_INTCSR);
  dev->start = 0;
  dev->tbusy = 1; /* can't transmit any more */
  release_region(dev->base_addr, BASE_ADDR0_RANGE);
  DEBUGMSG(" release region \n");
  free_irq(dev->irq, dev);
  DEBUGMSG(" release IRQ %d \n",dev->irq);
  MOD_DEC_USE_COUNT;
  return 0;
}

/*
 * Configuration changes (passed on by ifconfig)
 */
int flink_config(struct device *dev, struct ifmap *map)
{
  /* can't act on a running interface */
  if (dev->flags & IFF_UP)
    return -EBUSY;
  /* Don't allow changing the I/O address */
  if (map->base_addr != dev->base_addr) {
    printk(KERN_WARNING "flink: Can't change I/O address\n");
    return -EOPNOTSUPP;
  }
  /* Don't allow changing the IRQ */
  if (map->irq != dev->irq) {
    printk(KERN_WARNING "flink: Can't change IRQ\n");
    return -EOPNOTSUPP;
  }
  /* ignore other fields */
  return 0;
}

/*
 * Transmit a packet (low level interface)
 */
int flink_hw_tx(long *buf, int len, struct device *dev)
{
  /*
   * This function deals with hardware details.
   * In other words, this function implements
   * the flink behaviour,
   * while the other procedure (flink_tx)
```

```
   * is rather device-independent
  */
#ifdef DEBUG
  int i;
  char *p;
#endif
  int dma_bytes;        /* transmission size in words */
  unsigned long flags; /* save the interrupt flags */

#ifdef DEBUG
  if (debug & 2) {
    DEBUGMSG("Entry: Hardware_tx ...\n");
  }
#endif
  /*
   * we dont want to small packets
  */
  if (len < MIN_PKT_SIZE ) { len = MIN_PKT_SIZE ; }
#ifdef DEBUG
  if (debug & 2) {
    DEBUGMSG("hw_tx: lenght is %i \n",len);
  }
#endif
  dma_bytes=((len+3) & ~3);  /* need multiple of 4 byte*/
#ifdef DEBUG
  if (debug & 32) {
    for(p = (char *)buf, i=0;i<(len);i++,p++)
    {
      printk(" %02x",*p & 0xff);
    }
    printk("\n");
  }
  if (debug & 1) {
    DEBUGMSG(" skb anfang %08lx ende %08lx \n",
             ((long)buf) & ~4095,(long)(buf + len) & ~4095);
  }
#endif
  /*the follow code
   * does'nt will be interrupted by any interrupt*/
  save_flags(flags);
  cli();
  /* our critical code is the 64 bit header,
   * dont split it! */
```

```
  /* send out the header */
  outl(CTRL_SIGN3,dev->base_addr + AMCC_OP_REG_FIFO);
  outl((long) dma_bytes,dev->base_addr + AMCC_OP_REG_FIFO);
  if (len < DMA_SIZE) /* we must do it now,
                         or sometime we will lose a packet*/
  {
    outsl(dev->base_addr + AMCC_OP_REG_FIFO,buf,dma_bytes>>2);
  }
  restore_flags(flags); /* end of critical code */
  if (len < DMA_SIZE)
  {
    return 0; /* No DMA */
  }
#ifdef DEBUG
  if (debug & 1) {
    DEBUGMSG("hw_tx: DMA beginn ...\n");
    DEBUGMSG(" adresse %08lx \n",buf);
  }
#endif
  outl(virt_to_bus(buf),
       (dev->base_addr) + AMCC_OP_REG_MRAR); /* adress */
#ifdef DEBUG
  if (debug & 1) {
    DEBUGMSG("read from AMCC adresse= %08lx \n",
             inl(dev->base_addr + AMCC_OP_REG_MRAR));
  }
#endif
  outl(dma_bytes,
       (dev->base_addr) + AMCC_OP_REG_MRTC); /* set amount */
#ifdef DEBUG
  if (debug & 1) {
    DEBUGMSG("hw_tx: should transfer it to the card now ....\n");
    DEBUGMSG(" MCSR %08lx \n",
             inl(dev->base_addr + AMCC_OP_REG_MCSR));
    DEBUGMSG(" INTCSR %08lx \n",
             inl(dev->base_addr + AMCC_OP_REG_INTCSR));
  }
#endif
  return 1; /* DMA is running */
}

/*
 * Transmit a packet (called by the kernel)
```

```
 */
int flink_tx(struct sk_buff *skb, struct device *dev)
{
  int len;
  int again =1;
  struct flink_priv *privptr = (struct flink_priv *)dev->priv;
#ifdef DEBUG
  if (debug & 2) {
    DEBUGMSG("Want to send something...\n");
  }
#endif
  /* test if we are still sending */
  if (test_and_set_bit(0,(void*) &privptr->tx_busy) !=0 )
  {   /*we are still sending, try to handle  */
#ifdef DEBUG
    if (debug & 2) {
      DEBUGMSG("I'm busy with sending...\n");
    }
#endif
    /* stop more packets */
    test_and_set_bit(0,(void*) &dev->tbusy);
    /* look if the miss the end of DMA */
    if (inl(dev->base_addr + AMCC_OP_REG_MCSR)
        & MCSR32_READ_TR_READY )
    { /* DMA Compl. */
      DEV_FREE_SKB(privptr->tx_skb);
      privptr->tx_skb=NULL;
      privptr->stats.tx_packets++;
      clear_bit(0,(void*) &dev->tbusy);
    }
    else {
      printk(KERN_WARNING "%s No sending possible\n",
             dev->name);
      DEV_FREE_SKB(privptr->tx_skb);
      privptr->tx_skb=NULL;
      return -ENETDOWN; /* Network is breaking down,
                           restart all drivers! */
    }
  } /* end of test_and_set_bit */
  if (skb == NULL) {
#ifdef DEBUG
    if (debug & 2) {
      DEBUGMSG("tint for %p\n",dev);
```

```
    }
#endif
#if (LINUX_VERSION_CODE < 0x20155)
    dev_tint(dev); /* we are ready to transmit */
#endif
    return 0;
  }
  len = skb->len; /* in bytes !! */
  /* Ask the card, if the packet fits in the fifo */
  /* Attention, there is a failure in the calculation,
     the FIFO is 8 words(32) (from AMCC) larger then
     FIFO_BUFFER_SIZE and we didn't recognize the size
     of our header, but so long our header isn't greater
     than 8 words(32) this is okay
  */
  outl(FIFO_BUFFER_SIZE + 1 - (len >> 2),
       privptr->pld_addr + PLD_OP_REG_STX);
  dev->trans_start = jiffies; /* save the timestamp */
  /* if used Fifo-mem < STX_value
     then in ISR the TxAEF is set */
  while((inl(privptr->pld_addr + PLD_OP_REG_ISR)
        & ISR_TX_ALMOST_EMPTY) ==0 )
  {
    if(jiffies > (dev->trans_start + FLINK_TX_TIMEDOUT))
    {
      printk(KERN_WARNING
             "Not able to Send (Network busy)\n");
      /* No Choice we couldn't send,
      */
      privptr->stats.tx_errors++;
      if(test_and_set_bit(0,(void*) &privptr->txerror) !=0 ) {
        return -ENETDOWN; /* Network is broken down */
      } else {
        return -EBUSY;/*Not enough space on tx-fifo
                        and timedout*/
        /* give the card a second chance */
      }
    }
  }
  /* the card is able to send,
   * all problem seems to be solved */
  clear_bit(0,(void*) &privptr->txerror);
  /* Maybe we should prepare a timeout-timer for DMA,
```

```
     but I see no reason for it. The packet will fit in the
     fifo, so why the DMA should stopped?
  */
#ifdef DEBUG
  if (debug & 2) {
    DEBUGMSG("Sending prepared, calling device-specific ...\n");
  }
#endif
  again = flink_hw_tx( (long *) skb->data, len, dev);
  if( again == 0)
  { /* No DMA Started */
    DEV_FREE_SKB(skb);
    privptr->stats.tx_packets++;
    clear_bit(0,(void*) &privptr->tx_busy);
  } else { /* DMA Started */
    while (again) {
    int tickssofar = jiffies - dev->trans_start;
    if (inl(dev->base_addr + AMCC_OP_REG_MCSR)
        & MCSR32_READ_TR_READY )
    { /* DMA Compl. */
      DEV_FREE_SKB(skb);
      privptr->stats.tx_packets++;
      again = 0;
      clear_bit(0,(void*) &privptr->tx_busy);
    }
    if (tickssofar > FLINK_TX_TIMEDOUT && again !=0 )
    {
      printk(KERN_WARNING
             "flink: transmit timeout, any problem?\n");
#ifdef DEBUG
        if (debug & 2) {
          DEBUGMSG("Counter of flink is %i \n",
                   inl(dev->base_addr + AMCC_OP_REG_MRTC));
        }
#endif
        /* give the card a second chance */
            privptr->stats.tx_errors++;
        privptr->tx_skb=skb;/* rescue the skb-pointer */
        again = 0; /* end of endless loop */
      }/* of timedout */
    }    /* of while */
  } /* end of else */
#ifdef DEBUG
```

```
  if (debug & 2) {
    DEBUGMSG("tx: leaving ... \n");
  }
#endif
  return 0; /*  */
}

/*
 * Ioctl commands
 */
int flink_ioctl(struct device *dev, struct ifreq *rq,
                int cmd)
{
  /* Yes this function does nothing, should we offer
     such a function?  */
  DEBUGMSG("ioctl cmd %#x \n",cmd);
  return 0;
}

/*
 * Return statistics to the caller
 */
struct enet_statistics *flink_stats(struct device *dev)
{
  struct flink_priv *privptr = (struct flink_priv *)dev->priv;
  return &privptr->stats;
}

/*
 * The "change_mtu" method is usually not needed.
 *
 */
int flink_change_mtu(struct device *dev, int new_mtu)
{
  /* check ranges */
  if ((new_mtu < 68) || (new_mtu > MAX_MTU))
    return -EINVAL;
  /*
   * Do anything you need, and the accept the value
  */
  dev->mtu = new_mtu;
  return 0; /* success */
}
```

```
int flink_probe(struct device *dev)
{ /* find out data of our cards */
  int dev_found, cards_found=0;
#ifdef LINUX_20X
  /* if this fails you couldnt have the card */
  if (pcibios_present ()) {
#else
  struct pci_dev *pci_dev=NULL;

  /* if this fails you couldnt have the card */
  if (pci_present ()) {
#endif
    for (dev_found=0;dev_found< MAX_DEVS_FLINK;dev_found++) {
#ifdef LINUX_20X
      unsigned char pci_bus, pci_device_fn;
      int ret;
#endif
      u8 pci_irq_line;
      u16 vendor, device;
      u32 pci_ioaddr,baseaddr1,baseaddr3;

      vendor = FLINK_VENDOR; device = FLINK_DEVICE;
      /* search for devices with this vendor and device id's */
#ifdef LINUX_20X
      if (pcibios_find_device ((u_short) vendor,
          (u_short) device,(u_short) dev_found,&pci_bus,
          &pci_device_fn) != PCIBIOS_SUCCESSFUL)
        break; /* failure */
#else
      pci_dev=pci_find_device((u_short) vendor,
                              (u_short) device,pci_dev);
      if( pci_dev ==0|| pci_dev ==NULL)
        break;   /* failure */
#endif
#ifdef LINUX_20X
      /* read out basis address0 */
      pcibios_read_config_dword(pci_bus, pci_device_fn,
                                PCI_BASE_ADDRESS_0,
                                &pci_ioaddr);
      /* read out assigned interrupt */
      pcibios_read_config_byte(pci_bus, pci_device_fn,
                               PCI_INTERRUPT_LINE,
```

```
                              &pci_irq_line);
      /* explicit set I/O Acess Enable + Bus Master Enable,
       * without Bus Master Enable
       * the DMA machine doesnt work! */
      pcibios_write_config_word(pci_bus, pci_device_fn,
                                PCI_COMMAND, 0x5l);
#else
      pci_ioaddr = pci_dev->base_address[0];
      pci_irq_line = pci_dev->irq;
      /* explicit set I/O Acess Enable + Bus Master Enable,
       * without Bus Master Enable
       * the DMA machine doesnt work! */
      pci_set_master(pci_dev);
#endif
      /* Check for I/O or Memory Mapped Operation Registers*/
      if (!(pci_ioaddr & PCI_BASE_ADDRESS_SPACE)) {
        /* memory mapped */
            /* for now we must end here */
        printk (KERN_WARNING
                 "flink: baseaddr[0]: Memory mapped not implemented\n");
        return -EIO;
      }
      else {
        /* I/O mapped  */
        /* Remove I/O space marker */
        pci_ioaddr &= PCI_BASE_ADDRESS_IO_MASK;
        /*dev->base_addr=data;*/ /* the OP_base_addr */
      }
      if (pci_irq_line <= 0 || pci_irq_line >= 16)
        printk(KERN_WARNING
               " WARNING: The PCI BIOS assigned this PCI "
               " card to IRQ %d, which is unlikely to work!.\n"
               KERN_WARNING
               " You should use the PCI BIOS setup to assign"
               " a valid IRQ line.\n", pci_irq_line);
      DEBUGMSG("Card found at I/O %#x, IRQ %d.\n",
               pci_ioaddr, pci_irq_line);
      /* read out base addr 1 */
#ifdef LINUX_20X
      ret = pcibios_read_config_dword (pci_bus, pci_device_fn,
                                       PCI_BASE_ADDRESS_1,
                                       (unsigned int *) &baseaddr1);
      if (ret != PCIBIOS_SUCCESSFUL) {
```

```
        printk (KERN_WARNING
                "flink:  BASE_ADDRESS[1] ReadingError %s\n",
                pcibios_strerror (ret));
        return -EIO; /* OK we have a problem with the card*/
      }
#else
      baseaddr1 = pci_dev->base_address[1];
      /* if we havent a baseaddr
         than we havent a IO-Space-marker */
#endif
      /* Check for I/O or Memory Mapped Operation Registers*/
      if (!(baseaddr1 & PCI_BASE_ADDRESS_SPACE)) {
        /* memory mapped */
        /* for now we must end here */
#ifdef LINUX_20X
        printk (KERN_WARNING
                "flink: baseaddr[1]: Memory mapped not implemented\n");
#else
        printk (KERN_WARNING
                "flink: baseaddr[1]: IO-Adress not found \n");
#endif
        return -EIO;
      }
      else {
        /* I/O mapped  */
        /* Remove I/O space marker */
        baseaddr1 &= PCI_BASE_ADDRESS_IO_MASK;
      }
      DEBUGMSG("Baseaddr[1] at I/O %#x.\n", baseaddr1);
#ifdef  ALTERA_INIT
#ifdef LINUX_20X
      /* read out base addr 3,
         used for programming the Altera-chip */
      ret = pcibios_read_config_dword (pci_bus, pci_device_fn,
                                       PCI_BASE_ADDRESS_3,
                                       (unsigned int *) &baseaddr3);
      if (ret != PCIBIOS_SUCCESSFUL) {
        printk (KERN_WARNING
                "flink:  BASE_ADDRESS[3] ReadingError %s\n",
                pcibios_strerror (baseaddr3));
        return -EIO; /* OK we have a problem with the card */
      }
#else
```

```
      baseaddr3 = pci_dev->base_address[3];
      /* if we havent a baseaddr
         than we havent a IO-Space-marker */
#endif
      if (!(baseaddr3 & PCI_BASE_ADDRESS_SPACE)) {
        /* memory mapped */
        /* for now we must end here */
#ifdef LINUX_20X
        printk (KERN_WARNING
                "flink: baseaddr[3]: Memory mapped not implemented\n");
#else
        printk (KERN_WARNING
                "flink: baseaddr[3]: IO-Adress not found \n");
#endif
        return -EIO;
      }
      else {
        /* I/O mapped  */
        /* Remove I/O space marker */
        baseaddr3 &= PCI_BASE_ADDRESS_IO_MASK;
      }
      DEBUGMSG("BASE_ADDRESS[3] at I/O %#x \n",baseaddr3);
#else
      baseaddr3 = 0;
#endif
      dev = flink_found(0,pci_ioaddr,baseaddr1,baseaddr3,
                        dev_found);
      if (dev == 0) {
        /* Should not happen. */
        printk(KERN_ERR
               "flink: Probe of PCI card at %#x failed.\n",
               pci_ioaddr);
        printk(KERN_WARNING
               "flink:got IRQ %i IO-0 %#x IO-1 %#x IO-3 %#x \n",
               pci_irq_line,pci_ioaddr,baseaddr1,baseaddr3);
        continue;
      } else {
        struct flink_priv *flink_card =
        kmalloc(sizeof(struct flink_priv), GFP_KERNEL);
        dev->priv = flink_card; /* point to the right site */
        /* zero the memory */
        memset(dev->priv,0,sizeof(struct flink_priv));
        flink_card->next = flink_card_list;
```

```
        flink_card_list = flink_card;
        flink_card->dev = dev;
        /* to guarantee that this pointer is null */
        flink_card->tx_skb = NULL;
        /*
         * Then, allocate the priv field. This encloses
         * the statistics and a few private fields.
         */
        flink_card->pld_addr=baseaddr1;
        dev->irq = pci_irq_line;
        dev->base_addr = pci_ioaddr;
        dev->init = init_flink;
        register_netdev(dev);
        DEBUGMSG(" devicename is %s \n",dev->name); /**/
        /* Now we clear the Rx-Reset-flag,
           because its blocking the receive */
        outl(CLEAR_RX_BLOCKING,baseaddr1 + PLD_OP_REG_ICR);
        /* Interupt enable Register */
        outl(INTR_ON_RX_NOT_EMPTY,baseaddr1 + PLD_OP_REG_IER);
        /* enable interrupt for pld-chip */
        outl(ICR_INTR_ENABLE,baseaddr1 + PLD_OP_REG_ICR);
        /* this interrupt point to incoming mailbox 1,
         * so activat this, I will do it in the open method
         */
        /* Set the read and write transfer enable bits,
           now the card is allowed to make a DMA Transfer */
        outl(MCSR32_WRITE_TR_ENABLE|MCSR32_READ_TR_ENABLE,
             dev->base_addr + AMCC_OP_REG_MCSR);
      }
      dev = 0;
      cards_found++;
    }
    return cards_found ? 0 : -ENODEV;
  } else return -ENODEV; /* no device */
}

static struct device *flink_found(struct device *dev,
                                  int baseaddr0,
                                  int baseaddr1,
                                  int baseaddr3,
                                  int number)
{
#ifdef ALTERA_INIT
```

```
  extern char         altera[];
  int                 i;
  int                 j;
  char                byte;
  int                 count=0;
  int                 anzahl=sizeof(altera);
#endif

  if (dev == 0 || dev == NULL) {
    /* No dev-pointer exist, so create one */
    int alloc_size = sizeof(struct device) + sizeof("fl%d    ") +3;
    alloc_size &= ~3;        /* get a word(32) boundary */
    dev= (struct device *) kmalloc(alloc_size, GFP_KERNEL);
    memset(dev,0,alloc_size);
    dev->name= (char *) (dev + 1);/* point the pointer
                                     to his area */
    sprintf(dev->name, "fl%d", number);
    DEBUGMSG(" found %s \n",dev->name);
#ifdef LINUX_20X
    for (i=0;i<DEV_NUMBUFFS;i++)
      skb_queue_head_init(&dev->buffs[i]);
#else
    dev_init_buffers(dev);
#endif
  }
  /* Reset card. Who knows what state it was left in. */
  outl(MCSR32_GLOBAL_RESET,baseaddr0 + AMCC_OP_REG_MCSR);
  outl(0l,baseaddr0 + AMCC_OP_REG_MCSR); /* must clear */
#ifdef ALTERA_INIT
  DEBUGMSG("card_found: Starting initialisierungscode for Altrea\n");
  outl(0l,baseaddr3);
  udelay(1000l);/* wait 1000 microsec =
                   1millisec = max for udelay*/
  outl(4l,baseaddr3);
  udelay(1000l);
  outl(0l,baseaddr3);
  udelay(1000l);
  DEBUGMSG(" looking for what happen ...\n");
  if((inl(baseaddr3) & 1l) == 1l) {
    printk(KERN_WARNING
           "flink: signal CONF_DONE didn't go low at start\n");
    printk(KERN_WARNING "flink: Is IO %#x correct ?\n",
           baseaddr3);
```

```
    return 0; /* failure, no dev-pointer back*/
  }
  DEBUGMSG("card_found: ...starting ALTERA configuration\n");
  for (i=0;i<anzahl ;i++) {
    byte=altera[i];
    count++;
    for (j=0;j<8;j++) {
      /* DATA, DCLK=low  */
      outl((long) (byte & 1),baseaddr3);
      /* DATA, DCLK=high */
      outl((long) (byte & 1) | 2,baseaddr3);
      byte>>=1;
    }
  }
  DEBUGMSG("card_found: %i Bytes downloaded\n",count);
  udelay(10001); /* is this value realy nessesary ?*/
  if((inl(baseaddr3) & 1l)!=1l) {
    printk(KERN_WARNING
           "flink: Warning, signal CONF_DONE doesn't go high at the
           end\n");
    return 0; /* failure, no dev-pointer back*/
  }
  for (j=0;j<10;j++) { /* 10 cycles for initialisation */
    outl(2l,baseaddr3);        /* DCLK=high */
    outl(0l,baseaddr3);        /* DCLK=low  */
  }
  DEBUGMSG("init: Initialisation und Configuration of ALTERA done\n");
#endif
  dev->open               = flink_open;
  dev->stop               = flink_release;
  dev->set_config         = flink_config;
  dev->hard_start_xmit    = flink_tx;
  dev->do_ioctl           = flink_ioctl;
  dev->get_stats          = flink_stats;
  dev->change_mtu         = flink_change_mtu;
  dev->mtu                = MAX_MTU;
  dev->flags   = IFF_NOARP|IFF_POINTOPOINT|IFF_MULTICAST;
  dev->hard_header        = NULL;
  dev->rebuild_header     = NULL;
  dev->set_mac_address    = NULL;
#ifdef LINUX_20X
  dev->header_cache_bind  = NULL;
#else
```

```
  dev->hard_header_cache   = NULL;
#endif
  dev->header_cache_update = NULL;
  dev->hard_header_len     = 0;
  dev->addr_len            = 0;
  dev->type                = ARPHRD_PPP; /* ARPHRD_SLIP
                                            works too */
#ifdef LINUX_20X
  dev->family              = AF_INET; /*Adress family of the
                                 interface usually need'nt
                                 to be assign needed for
                                 pa_addr, pa_brdaddr,
                                 pa_mask */
#endif
  return dev; /* succes */
}


/*
 * The devices
 */
char flink_names[8]= " "; /*  eight-byte buffers */
struct device flink_devs = {
  flink_names, /* name -- set at load time */
  0, 0, 0, 0,  /* shmem addresses */
  0x000,       /* ioport */
  0,           /* irq line */
  0, 0, 0,     /* various flags: init to 0 */
  NULL,        /* next ptr */
  NULL,  /* init function, fill other fields with NULL's */
};

/*
 * Finally, the module stuff
 */
int init_module(void)
{
    /* this procudure is called by the kernel
       then the module is loaded */
    int ret;

    DEBUGMSG ( " >>> START <<<\n");
    ret=flink_probe(0);
```

```
    return  ret ;
}

void cleanup_module(void)
{
  struct device *dev;
  struct flink_priv *this_card;

  /* No need to check MOD_IN_USE,
     as sys_delete_module() checks. */
  while (flink_card_list) {
    dev = flink_card_list->dev;
    unregister_netdev(dev);
    release_region(dev->base_addr, BASE_ADDR0_RANGE);
    kfree(dev);
    this_card = flink_card_list;
    flink_card_list = flink_card_list->next;
    kfree(this_card);
  }
  return;
}
```

C.2 flink.h

```
/*
 * flink.h -- definitions for the network module
 *
 */

/*
 * Macros to help debugging
 */

#undef DEBUGMSG  /* undef it, in all cases */
#ifdef DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
```

```
#    define DEBUGMSG(fmt, args...) printk(KERN_DEBUG
                                          " flink: "fmt, ## args)
#  else
     /* This one for user space */
#    define DEBUGMSG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define DEBUGMSG(fmt, args...) /* not debugging: nothing */
#endif

/*
 * Define some values
 */
#define MAX_DEVS_FLINK 3
/* Vendor and device ID's */
#define FLINK_VENDOR   /* 0x10e8*/    0x1234
#define FLINK_DEVICE   /*  0x4750*/   0x5678

#define FIFO_BUFFER_SIZE 4096  /*(4k x 32bit) = 4k (D)Words*/
/* ok the range of first base address */
#define BASE_ADDR0_RANGE 0x40 /*I don't know if it's correct*/
/* offsets */
#define AMCC_OP_REG_IMB1 0x10 /* Offset of Incoming Mailbox 1*/
#define AMCC_OP_REG_MWAR 0x24 /* Offset of Master Write
                                 Adress Register */
#define AMCC_OP_REG_MWTC 0x28 /* Offset of Master Write
                                 Transfer Count */
#define AMCC_OP_REG_MRAR 0x2c /* Offset of Master Read
                                 Adress Register */
#define AMCC_OP_REG_MRTC 0x30 /* Offset of Master Read
                                 Transfer Count */
#define AMCC_OP_REG_INTCSR 0x38 /* Offset of Interrupt
                                 Control/Status Register */
#define AMCC_OP_REG_MBEF 0x34 /* offset of Mailbox Empty/Full
                                 Status Register (readonly) */
#define AMCC_OP_REG_FIFO 0x20 /*offset of the FIFO's (rx + tx)*/

/* offsets for PLD */
#define PLD_OP_REG_ICR 0x00 /* Interface Control Register */
#define PLD_OP_REG_ISR 0x04 /* Interface Status Register */
#define PLD_OP_REG_IER 0x08 /* Interrupt Enable Register */
#define PLD_OP_REG_GRX 0x0c /* GET RxFifoLevel Register */
#define PLD_OP_REG_SRX 0x10 /* Set RxFifoLevel Register */
```

```
#define PLD_OP_REG_STX 0x14 /* TxFifoLevelRegister */

/*  values  */
#define ENABLE_INTR_TR_COMPL 0x0000c000L /* Enable for
                        Interrupt on Transfer Complete
                                for Read and Write */
#define ENABLE_INTR_WRITE    0x00004000L /* Enable write
                                        int see below */
#define ENABLE_INTR_READ     0x00008000L /* Enable read int */
#define INTR_WRITE_TR_COMPL  0x00040000L /* the interruptflag
                                WRITE TRANSFER COMPLETE */
#define INTR_READ_TR_COMPL   0x00080000L /* the interruptflag
                                 READ TRANSFER COMPLETE */
#define INTR_MASTER_ABORT    0x00100000L /* occurs
                                when no target response,
                                  see AMCC S5933 S.3-61 */
#define INTR_TARGET_ABORT    0x00200000L /*AMCC S5933 S.3-61*/
#define INTR_ASSERTED        0x00800000L /* notify if
                                 one or more intr. of
                                       INTR_READ_TR_COMPL,
                                       INTR_WRITE_TR_COMPL,
                                       INTR_IN_MAILBOX
                                   occurs */
#define ADD_ON_RESET_PIN    0x01000000L /* occurs */
#define ENABLE_INTR_IN_MAILBOX_1_BYTE_0  0x00001000 /* see
                                         AMCC S5933 page 3-59
                                         INTCSR-register */
#define INTR_IN_MAILBOX 0x00020000L /* incoming mailbox
                                       interrupt */

/* some def. from amcc */
#define AMCC_OP_REG_MCSR        0x3c

/* status from pld  Interface Status Register */
#define ISR_RX_NOT_EMPTY        0x00000001L
#define ISR_TX_EMPTY            0x00000004L
#define ISR_TX_ALMOST_EMPTY     0x00000008L

/* 32-bit handling of MCSR */

#define  MCSR32_GLOBAL_RESET    0x0F000000L  /* reset */
#define  MCSR32_MB_RESET        0x08000000L
#define  MCSR32_A2P_FIFO_RESET  0x04000000L
```

```
#define  MCSR32_P2A_FIFO_RESET  0x02000000L
#define  MCSR32_ADD_ON_RESET    0x01000000L
#define MCSR32_READ_TR_ENABLE   0x00004000L /*read transfer
                                             control*/
#define MCSR32_READ_IF_4_EMPTY  0x00002000L
#define MCSR32_READ_PRIORITY    0x00001000L
#define MCSR32_WRITE_TR_ENABLE  0x00000400L /*write transfer
                                             control*/
#define MCSR32_WRITE_IF_4_FILLED 0x00000200L
#define MCSR32_WRITE_PRIORITY   0x00000100L
#define MCSR32_WRITE_TR_READY   0x00000080L /* status */
#define MCSR32_READ_TR_READY    0x00000040L
#define MCSR32_A2P_FIFO_EMPTY   0x00000020L
#define MCSR32_A2P_FIFO_HALF    0x00000010L
#define MCSR32_A2P_FIFO_FULL    0x00000008L
#define MCSR32_P2A_FIFO_EMPTY   0x00000004L
#define MCSR32_P2A_FIFO_HALF    0x00000002L
#define MCSR32_P2A_FIFO_FULL    0x00000001L
#define ALIGNMENT_ERROR         0x00000020L
#define PARITY_ERROR            0x00000010L

#define        FLINK_LWR        0x10000000L
#define        FLINK_GWR        0x30000000L
#define        FLINK_PLE        0x00000000L
#define        FLINK_TOK        0x50000000L
#define        FLINK_OPN        0x20000000L
#define        FLINK_CLS        0x40000000L
#define        FLINK_RRS        0x60000000L

#define FLINK_TX_TIMEDOUT  10 /* =100msec
                                 (1jiffie=10msec on PC) */
#define FLINK_RX_TIMEDOUT  20 /* =200msec */

/* choose the interruptsource for IER */
#define INTR_ON_RX_NOT_EMPTY     0x00000001L
#define INTR_ON_RX_LEVEL         0x00000002L

/* a value for Interrupt Enable for ICR */
#define ICR_INTR_ENABLE          0x00000001L
#define CLEAR_RX_FIFO            0x00000010L
#define CLEAR_RX_BLOCKING        0x00000000L

#define FLINK_GET_COMMAND(i)   ((i) & 0xF0000000L)
```

```
#define FLINK_GET_SOURCEADR(i) ((i) & 0x000000FFL)
#define FLINK_GET_DESTADR(i)   (((i) & 0x00FF0000L) >> 16)
#define FLINK_GET_PKTID(i)     (((i) & 0x0000FF00L) >> 8)
#define FLINK_COMMAND_VALID(i) (!(((i) & 0xF0000000L) > 0x60000000))
#define FLINK_PKT_LENGTH(i)    ((i) & 0x0000FFFCL)

#define FLINK_CREATE_PLE_PKT(i) (i)
#define FLINK_CREATE_OPN_PKT(i) (((i) << 16) + (i) + ( 1 << 29))
#define FLINK_CREATE_CLS_PKT(i) (((i) << 16) + (i) + ( 1 << 30))
#define FLINK_CREATE_GWR_PKT(i) (((i) << 16) + (i) + ( 3 << 28))
#define FLINK_CREATE_LWR_PKT(i,j) (((j) << 16) + (i) + ( 1 << 28))

#define FLINK_CHECK_OPN_PKT(i,j) (((i) & 0x40000000L)
                                  && (((i) & 0x000000FFL) & (j)))
#define FLINK_CHECK_CLS_PKT(i,j) (((i) & 0x80000000L)
                                  && (((i) & 0x000000FFL) & (j)))
```

Index