# Diplomarbeit

zum Thema

## Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur

an der

Technischen Universität Dresden
Fakultät Informatik
Institut für Betriebssysteme, Datenbanken und Rechnernetze
Lehrstuhl Betriebssysteme

Eingereicht von: René Stange
Geboren am: 18. April 1969
Geboren in: Bad Saarow-Pieskow
Matrikel-Nr.: 1279178
Eingereicht am: 02. Mai 1996

Betreuender Hochschullehrer:
Prof. Dr. H. Härtig

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

Die Anbindung von Ein-/Ausgabegeräten an das Bussystem eines Personalcomputers erfolgt über spezielle Gerätesteuerungen (Controller), auf die programmtechnisch (im allgemeinen) mittels Ein-/Ausgabebefehlen des Mikroprozessors zugegriffen wird. Die Menge der vorhandenen Geräte für Personalcomputer ist vielfältig. Dies trifft besonders für Systeme zu, die dem von der Firma IBM Anfang der achtziger Jahre begründeten Industriestandard folgen, der insbesondere durch Verwendung der Mikroprozessoren aus der Reihe Intel 80x86 gekennzeichnet ist.

Um die Komplexität der Hardware zu verbergen, werden in Betriebssystemen als Abstraktionsmittel gerätespezifische Programme - Gerätetreiber (engl. device driver) - verwendet. Die Hersteller von Gerätesteuerungen liefern gewöhnlich Gerätetreiber für einige wenige weit verbreitete Betriebssysteme mit kommerzieller Bedeutung. Treiber für Systeme mit geringer Verbreitung müssen bei Bedarf selbst entwickelt werden. Die Entwicklung eines Gerätetreibers ist aufwendig. Es soll jedoch möglichst eine Vielzahl von Geräten unterstützt werden. Dieses Problem bestand auch bei dem bei der Gesellschaft für Mathematik und Datenverarbeitung (GMD) entwickelten und am Lehrstuhl Betriebssysteme der Fakultät Informatik der Technischen Universität Dresden für Forschungszwecke verwendeten µ-Kern-basierten Betriebssystem L3 [Liedtke 91a].

Für die vergleichbare Problematik des Fehlens von Applikationen für neue Betriebssysteme ist die Emulation einer etablierten Betriebssystemumgebung ein bekanntes Verfahren, daß auch für L3 in Form einer UNIX-Emulation zur Anwendung kommt [Wolter 95]. Ein ähnliches Vorgehen erscheint auch für Gerätetreiber sinnvoll. Wenn es gelingt, die Umgebung, die ein Treiber in einem verbreiteten Betriebssystem vorfindet, im Zielsystem nachzubilden, so wird mit vergleichsweise geringem Aufwand eine Vielzahl von Gerätetreibern und damit die von ihnen unterstützte Hardware verfügbar. Bisher sind derartige Lösungen allerdings selten. Eine Ursache könnten die hohen Leistungsanforderungen sein, die an Treibersoftware gestellt werden. In [Goel 96] wird eine Emulationsumgebung für Gerätetreiber aus dem frei verfügbaren UNIX-ähnlichen Betriebssystem Linux für das Betriebssystem Mach [Mach 95] beschrieben.

Die vorliegende Arbeit hat zum Ziel, die systematische Übertragbarkeit von Linux-Gerätetreibern auf den µ-Kern L3 zu untersuchen. Linux bietet sich für diesen Zweck als Ausgangsbasis an, da es inklusive des Quellkodes (Systemkern und Gerätetreiber) gemäß der GNU General Public License [FSF 91] frei verfügbar ist und weil für Linux eine große Zahl von Treibern existiert. Der Versuch der Übertragung auf Quellkode-Ebene verspricht wesentlich mehr Erfolg als die Realisierung von Binärkompatibilität, die ohne Verfügbarkeit des Quellkodes notwendig wäre.

Wie sich im Laufe der Arbeit herausgestellt hat, ist eine einheitliche Behandlung von Linux-Gerätetreibern der verschiedenen interessierenden Geräteklassen (Netzwerk- und Plattentreiber) nicht möglich, da sie sich in ihren Schnittstellen zum Systemkern unterscheiden (siehe Abschnitt 2.2). Wegen der begrenzten verfügbaren Zeit mußte deshalb eine Beschränkung auf eine Geräteklasse erfolgen. Die Entscheidung fiel aus Bedarfsgründen und aus Gründen, die im nächsten Kapitel zum Ausdruck kommen, zugunsten von Netzwerktreibern.

Diese Arbeit ist in sechs Kapitel gegliedert. Das nachfolgende Kapitel zwei beschreibt den Aufbau und die Funktionsweise von Gerätetreibern in den relevanten Betriebssystemen L3 und Linux, wobei besonders auf Treiber für Netzwerkgeräte eingegangen wird. Ein weiterer Betrachtungsgegenstand ist die bereits erwähnte Emulation für Linux-Treiber in Mach. Aufbauend auf diesen Erkenntnissen wird in Kapitel drei ein Rahmen für einen L3-Netzwerktreiber entworfen, in den sich ein Linux-Netzwerktreiber einbinden läßt. Beim Entwurf auftretende Problemstellungen werden ausführlich diskutiert. Das vierte Kapitel beschäftigt sich mit ausgewählten Aspekten der programmtechnischen Umsetzung des Entwurfs. Im fünften Kapitel wird eine Bewertung der entstandenen Lösung vorgenommen. Das sechste Kapitel faßt die Ergebnisse der Arbeit zusammen und gibt einen Ausblick.

An dieser Stelle möchte ich mich herzlich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben, besonders bei Prof. Härtig für die fachliche Unterstützung und das Verständnis für meine Wünsche, bei Jean Wolter, der auf viele Fragen eine Antwort wußte, mehrfach die richtigen gestellt hat und der den PCI-Treiber implementierte, bei Robert Baumgartl für die große Unterstützung in der Endphase, bei den Mitarbeiterinnen und Mitarbeitern des Lehrstuhls Betriebssysteme, die dafür gesorgt haben, daß es eine schöne Zeit war und nicht zuletzt bei Jemandem, ohne den ich diese Arbeit wohl nie begonnen hätte.

# 2 Gerätetreiber

Ein Gerätetreiber bietet Dienste über eine systemspezifische Schnittstelle an und nutzt Dienste des Systemkerns. Er greift auf Hardware-Ressourcen (I/O-Ports, Interrupts) zu. Er ist in eine bestimmte Systemumgebung eingebettet. Dieses Kapitel untersucht Gerätetreiber der Betriebssysteme L3 und Linux. Dabei interessieren insbesondere folgende Aspekte:

- Welche Klassen von Geräten existieren? Wie werden eine Geräteklasse und verschiedene Geräte einer Klasse im System repräsentiert?
- In welcher Umgebung läuft der Treiber? Welche Nebenläufigkeiten existieren? Wie erfolgt die Synchronisation mit der Hardware (Behandlung von Interrupts) und mit Nutzern der angebotenen Dienste?
- Welche Dienste werden durch den Treiber angeboten? Über welche Schnittstelle sind diese zu erreichen?
- Welche Dienste und Ressourcen werden durch den Treiber selbst verwendet, und wie erfolgt der Zugriff zu diesen?
- Wie ist der Treiber ins System eingebunden? Wie wird er installiert und konfiguriert?

## 2.1 Gerätetreiber in L3

In diesem Abschnitt soll die Beschaffenheit von L3-Gerätetreibern untersucht werden. Die hierzu verfügbaren Informationen sind relativ spärlich und verstreut. L3 ist zwar mit Blick auf kommerzielle Anwendungen entwickelt worden [Liedtke 91a], war und ist jedoch vor allem auch ein Gegenstand der Betriebssystemforschung [Liedtke 93], weshalb dieser Umstand verständlich ist. Es ist hier nicht möglich, einen allgemeinen Überblick über L3 zu geben. Zu diesem Zweck wird z.B. auf [Wolter 95] verwiesen. Die Kenntnis von L3 wird im weiteren vorausgesetzt.

Das Betriebssystem L3 basiert auf einem µ-Kern. Das Konzept der Minimalisierung des Systemkerns und der Verlagerung von Funktionalität auf die Ebene von Nutzerprozessen wurde in L3 im Gegensatz zu anderen Systemen (z.B. Mach) konsequent auch für Gerätetreiber durchgesetzt. Geräte sind aktive Objekte, die in L3 durch Prozesse repräsentiert werden [Liedtke 91a]. Wenn Gerätetreiber Prozesse (im weiteren auch als Task bezeichnet) sind, so sind diese genauso unabhängig vom Systemkern und damit flexibel und austauschbar wie alle anderen Nutzerprozesse.

Als Ausnahme sind Gerätetreiber, die für den Systemstart benötigt werden, direkt in den L3-Kern integriert. Dies gilt insbesondere für den Festplattentreiber, auf den die Speicherverwaltung zugreift. Dieser Treiber ist nur durch Änderung des Kerns zu ersetzen. Das ist ein Grund, weshalb Plattentreiber in dieser Arbeit nicht speziell behandelt wurden.

Die Ein- und Ausgabe von bzw. zu Geräten und damit Gerätetreibern erfolgt über die *Interprozeßkommunikation (IPC)*, die in L3 besonders optimiert wurde [Liedtke 93]. Deren weniger effiziente Realisierung war in anderen Systemen, wie in dem bereits erwähnten Mach, der Grund, warum Gerätetreiber dort in den Systemkern integriert werden mußten. Die Interprozeßkommunikation ermöglicht die Übermittlung von strukturierten Nachrichten zwischen Prozessen. In [Hohmuth 96] werden die flexiblen Möglichkeiten zum Nachrichtenaufbau im einzelnen dargelegt. Als Bestandteile einer Nachricht sind 32-Bit-Wörter (DWord), Bytefolgen einer bestimmten Länge (String), Datenräume und Flexpages möglich. Für Treiberprozesse sind die letzten beiden Bestandteile weniger bedeutend, so daß sie in dieser Arbeit keine Rolle spielen.

Die Interprozeßkommunikation verläuft synchron. Es kommt erst dann zu einem Nachrichtenaustausch, wenn Sender- und Empfängerprozeß explizit eine Sende- bzw. Empfangsoperation durchführen. Ein Prozeß kann vom Systemkern den Abbruch der Kommunikation fordern, wenn sie nach einer bestimmten Zeit nicht zustande gekommen ist (Timeout). Der L3-Kern speichert keine Nachrichten, die nicht zugestellt werden können, weil der Empfänger nicht bereit ist. Über die Interprozeßkommunikation erfolgt eine Synchronisation zwischen Nutzerprozeß und Gerätetreiber, da beide in ihrer Ausführung solange blockiert werden, bis die Kommunikation durchgeführt wurde oder eine Zeitüberschreitung auftritt.

Der L3-Kern unterstützt Gerätetreiber als Nutzerprozesse durch folgende spezielle Konzepte [Liedtke 91a]:

- Ein Prozeß kann bei seiner Erzeugung als *resident* gekennzeichnet werden. Eine derartige sogenannte *residente Task* hat einen eigenen Adreßraum mit einer endlichen Anzahl von Seiten des physischen Hauptspeichers, die niemals von anderen Prozessen verdrängt werden. Auf diese Weise wird gesichert, daß sich die Antwortzeit von Gerätetreibern nicht durch eventuell notwendige Plattenzugriffe der virtuellen Speicherverwaltung drastisch erhöht.

  Die Belegung einer Seite des Hauptspeichers durch die residente Task geschieht, wenn eine Schreiboperation auf eine beliebige Adresse im Taskadreßraum durchgeführt wird. Es wird dann an dieser Adresse eine Seite bereitgestellt, die nach der Seitengröße von 4096 Byte ausgerichtet wird. Ist zu diesem Zweck die Verdrängung einer Seite einer anderen (nicht residenten) Task notwendig, so würde die Ausführung der residenten Task für den Zeitraum, den ein Plattenzugriff benötigt, blockiert. Es müssen deshalb während der Initialisierungsphase ggf. alle benötigten Seiten beschrieben werden.

  L3 verwaltet den virtuellen Speicher in Form von sogenannten *Datenräumen* (data spaces). Residente Tasks besitzen im Gegensatz zu anderen Prozessen immer nur einen einzigen Datenraum, dem Standarddatenraum, der den ausführbaren Programmkode, Daten und den

Stapelspeicher (Stack) enthält. Systemrufe, die mit Datenräumen operieren, sind nicht verwendbar.

Residente Tasks sind im Gegensatz zu anderen L3-Tasks nicht *persistent*, d.h. sie existieren nicht über die Laufzeit des Systems hinaus, wie das für normale Nutzerprozesse der Fall ist, sondern müssen bei jedem Systemstart neu erzeugt werden.

- Der Adreßraum einer Task läßt sich um *I/O-Adressen* erweitern. Konzeptionell ist es vorgesehen, dies selektiv für bestimmte Untermengen des I/O-Adreßraums zu realisieren, was bisher jedoch nicht umgesetzt wurde. Ein Gerätetreiber hat deshalb Zugriff auf den gesamten I/O-Adreßraum des Prozessors.
- Einem Thread kann bei seiner Erzeugung eine Hardware-*Interrupt*-Quelle (IRQ) zugewiesen werden. Beim Auftreten dieses Interrupts erzeugt der Kern eine spezielle IPC-Nachricht an den Thread. Interrupt-Nachrichten werden vom Kern gepuffert und können anhand des beim Empfang der Nachricht gelieferten Thread-ID des Absenders identifiziert werden (Abb. 1).

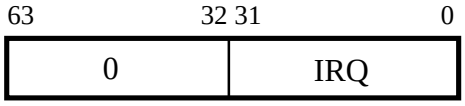


Abb. 1: Absender-Thread-ID einer Interrupt-Nachricht

Gerätetreiber bzw. deren Threads unterliegen genauso dem Einfluß der preemtiven L3-Scheduling-Strategie, wie alle anderen Threads. Ihnen kann unter Umständen ohne eigene Einwirkung der Prozessor entzogen werden. Damit dadurch kein drastischer Leistungsnachteil entsteht, sollte die Priorität eines Treiber-Threads auf den Wert 0 (maximale Priorität) gesetzt werden. Dies ist jedoch auch keine Garantie dafür, daß die Arbeit eines Gerätetreibers nicht etwa von einem Nutzerprozeß unterbrochen wird. Durch die L3-Thread-Priorität läßt sich nur der Anteil an Prozessorzeit steuern, die ein Thread im Mittel erhält. Die tatsächliche Priorität eines Threads wird vom Scheduler dynamisch verändert. Aus Treibersicht wäre die Einführung von statischen Prioritäten und die Möglichkeit der nicht-preemtiven Ausführung von Threads sinnvoll.

L3-Treiberprozesse können Dienste des L3-Kerns über Systemrufe nutzen. Dabei stehen ihnen als privilegierte Tasks einige zusätzliche Dienste (z.B. Einblendung von bestimmten Bereichen des physischen Hauptspeichers in den Taskadreßraum) zur Verfügung, die für normale Nutzerprozesse nicht zugänglich sind. Ein Treiber kann auch Dienste anderer Gerätetreiber über die Interprozeßkommunikation in Anspruch nehmen. Das gilt speziell für "logische" Treiber, die nicht direkt auf die Hardware zugreifen. Dies ist z.B. für SCSI-Treiber für die verschiedenen SCSI-Geräte (Festplatte, CD-ROM, Bandlaufwerk) denkbar, die für die

eigentliche Kommunikation mit der Hardware einen allgemeinen SCSI-Treiber verwenden, der abhängig vom verwendeten SCSI-Controller ist [Liedtke 91a].

Privilegierte Tasks stammen in der L3-Taskhierarchie von der Task "SYSUR" ab. Treiberprozesse werden beim Systemstart von der Task "SYSHW" erzeugt, die ihrerseits im SYSUR-Zweig zu finden ist.

### 2.1.1 General Driver Protocol (GDP)

Mit der Interprozeßkommunikation ist die Schnittstelle zwischen Gerätetreibern und Nutzerprozessen bereits benannt worden. Diese wird vom L3-Systemkern realisiert. Abgesehen davon macht der Kern jedoch keine Vorschriften für die Kommunikation mit Gerätetreibern, wenn man von dem im Kern enthaltenen Plattentreiber absieht. Darin unterscheidet sich L3 von anderen Betriebssystemen. Prinzipiell kann der Programmierer eines Gerätetreibers auf der Basis der Interprozeßkommunikation eine beliebige Kommunikationsvorschrift (Protokoll) für seinen Treiber verwenden, solange die Nutzerprozesse dieser folgen.

Aus systematischen Gründen ist es jedoch sinnvoll, eine allgemeine Vorschrift zu definieren, die von allen Gerätetreibern verwendet werden sollte. Diesen Zweck erfüllt das General Driver Protocol (GDP), das in [Liedtke 91b] spezifiziert ist. Diese Spezifikation ist allein jedoch nicht hinreichend für Treiber einer bestimmten Geräteklasse. [Heinrichs 88] führt eine Konkretisierung für Netzwerkgeräte durch (Abschnitt 2.1.3). Auf andere Geräteklassen geht diese Arbeit nicht speziell ein.

Die GDP-Kommunikation verläuft asymmetrisch und synchron. Der Nutzerprozeß sendet einen Auftrag (order) an den Gerätetreiber, der den Auftrag bearbeitet und eine Antwort (reply) zurücksendet. Dieser Ablauf wird durch die IPC-Funktionalität des L3-Kerns optimal unterstützt, da sich die Aussendung eines Auftrags und der Empfang der Antwort über die *Call*-Funktion mit einem einzigen Systemruf ausführen lassen. Der Gerätetreiber kann nach Empfang des Auftrags davon ausgehen, daß der Auftraggeber bereits auf die Antwort-Nachricht wartet und kann diese mit einem Timeout von 0 senden, was die Struktur des Treibers sehr vereinfacht [Liedtke 93] (siehe Abschnitt 3.2.2). Diese als *Client/Server-Modell* bekannte Struktur wird jedoch praktisch nicht konsequent verfolgt. Einige Gerätetreiber signalisieren das Eintreten eines bestimmten Ereignisses unaufgefordert durch das Senden einer Nachricht. Dies wird auch von Netzwerktreibern verlangt [Heinrichs 88].

Die übertragenen Nachrichten haben eine einheitliche Struktur. Jede GDP-Nachricht beginnt mit einem *Nachrichtenkode* (message code), der in einem DWord kodiert ist (Abb. 2). Darauf kann ein Datenobjekt (z.B. Datenblock, Zeichen, Zeichenkette, Blocknummer) folgen.

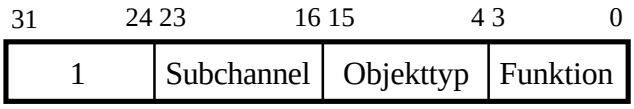


Abb. 2: Aufbau des GDP-Nachrichtenkodes

Im höchstwertigsten Byte des Nachrichtenkodes ist die *Protokollnummer* des GDP (1) enthalten. Da ein Prozeß Nachrichten verschiedener Protokolle empfangen kann, ist dies zur Identifizierung des Protokolls notwendig. Alle L3-IPC-Protokolle halten sich an diese Konvention. Die definierten Protokollnummern sind in [Accomodat 93] aufgeführt.

Ein Gerätetreiber kann mehrere (maximal 256) physische (z.B. verschiedene Laufwerke) oder logische Geräte unterstützen, die nach GDP-Terminologie als *Subchannel* bezeichnet werden. Der Nachrichtenkode enthält ein Feld, das den adressierten Subchannel spezifiziert. Wird nur ein Gerät unterstützt, so ist dieses standardmäßig über den Subchannel 0 erreichbar. Die Adressierung eines bestimmten Gerätes ist in L3 damit eindeutig über den Namen des Gerätetreibers (genauer des Threads, der die GDP-Nachrichten empfängt) in Verbindung mit dem zum Gerät gehörenden Subchannel möglich. Ein Gerätetreiber, der in der Initialisierungsphase keine durch ihn zu unterstützenden Geräte findet, hat sich umzubenennen, indem er eine Fehlermeldung an den Thread-Namen anhängt [Liedtke 90]. Auf diese Weise ist er von Nutzerprozessen nicht erreichbar.

Der *Objekttyp* (object type) gibt die Art des behandelten Datenobjekts an. Das kann auch ein leeres Objekt (nil object) sein.

Das Feld *Funktion* (function) spezifiziert den Nachrichtentyp als Auftrag oder Antwort und bei Aufträgen die mit dem Datenobjekt durchzuführende Operation. Die Bedeutung der einzelnen Funktionsbits ist in Abbildung 3 angegeben. Abbildung 4 zeigt die definierten Bitkombinationen und erklärt die zugehörigen GDP-Funktionen.

| **Bit** | **Bezeichnung** | **Bedeutung** |
|---|---|---|
| 3 | order/reply | bei Auftrag an einen Gerätetreiber 1, sonst 0 |
| 2 | interrogate/ alternate | Auftrag: Ausführbarkeit der Operation prüfen (nicht wirklich ausführen), Antwort: Operation nicht ausführbar (Vorschlag einer Alternative) |
| 1 | early | asynchrone Operation (sofortige Antwort des Treibers gefordert), nur bei Auftrag relevant |
| 0 | in/out | Übertragungsrichtung (0 für Eingabe) |

Abb. 3: Bedeutung der GDP-Funktionsbits

| Binär | Dez. | Bezeichnung | Erklärung |
|---|---|---|---|
| 0000 | 0 | reply | Auftrag wurde ausgeführt bzw. ist ausführbar (Antwort auf interrogate). Zusätzlich zu dem im Nachrichtenkode spezifizierten Datenobjekt enthält die Nachricht ein Fehlerstatusobjekt. |
| 0100<br>0101 | 4<br>5 | alternate in requested<br>alternate out requested | Auftrag ist mit den übermittelten Parametern nicht ausführbar. Die Nachricht enthält korrigierte Werte, die den originalen Parametern so nah wie möglich kommen. |
| 1000<br>1001 | 8<br>9 | exec in<br>exec out | Synchrone Ein- bzw. Ausgabe von Daten oder Steuerinformationen (abhängig vom angegebenen Objekttyp). Die Antwort des Treibers wird nach Ausführung der Operation gesendet. |
| 1010<br>1011 | 10<br>11 | exec early in<br>exec early out | Asynchrone Ein- bzw. Ausgabe von Daten. Sofortige Antwort des Treibers. Falls bei einer Eingabe-Operation keine Daten vorliegen, ist Nil-Reply zu senden. |
| 1100<br>1101 | 12<br>13 | interrogate in<br>interrogate out | Ausführbarkeit der Operation prüfen. |

Abb. 4: GDP-Funktionen

Die Definition für die Funktionen exec early in und exec early out ist hier aus Platzgründen unvollständig wiedergegeben und auf Netzwerktreiber ausgerichtet. Die vollständige Definition ist in [Liedtke 91b] und [Hohmuth 96] zu finden.

| Geräteklasse | Beispiel-Gerät | Beispiel-Objekttyp |
|---|---|---|
| general | alle | nil object |
| stream IO | LAN-Controller | datagram |
| block IO | Festplatte | block sequence |
| input | Tastatur | scan ascii |
| display | Bildschirmadapter | window size |

Abb. 5: GDP-Geräteklassen

Das GDP definiert verschiedene Geräteklassen und zugehörige Objekttypen. Ein Datenobjekt kann ein einzelnes DWord sein, eine Datenstruktur, die aus DWords und Zeichenketten (indirect Strings, siehe [Hohmuth 96]) zusammengesetzt ist oder ein Nil-Objekt, das für ein nicht vorhandenes Objekt steht. Abbildung 5 zeigt die wichtigsten Klassen mit je einem

Beispiel für ein zugehöriges Gerät und einen für dieses Gerät definierten Objekttyp. Das Nil-Objekt hat in Verbindung mit einigen GDP-Funktionen eine besondere Bedeutung für das Öffnen und Schließen eines Gerätetreibers durch einen Nutzerprozeß (Abb. 6).

| GDP-Funktion | Bedeutung | Erklärung |
|---|---|---|
| nil exec in | open order | Öffnen des Treibers, Herstellen einer Verbindung zwischen Nutzerprozeß und Treiber |
| nil exec out | close order | Schließen der Verbindung zwischen Nutzerprozeß und Treiber |
| nil alternate in requested | open requested | Reaktion eines geschlossenen Treibers auf alle GDP-Aufträge außer Open |
| nil alternate out requested | close requested | Reaktion eines Treibers auf einen nicht ausführbaren Auftrag, wenn keine Alternative existiert |

Abb. 6: Bedeutung des GDP-Nil-Objekts (Open und Close)

Die allgemeine Diskussion des General Driver Protocols soll mit einem Beispiel abgeschlossen werden. Es wird die Eingabe eines von einem Netzwerktreiber empfangenen Datagramms an einen Nutzerprozeß betrachtet:

1. Der Nutzerprozeß sendet den GDP-Auftrag *datagram exec early in* an den Treiber. Die IPC-Nachricht besteht nur aus dem GDP-Nachrichtenkode mit dem Wert 0x0100101A. Darin enthalten ist der Subchannel 0, der Objekttyp datagram (0x101) und die GDP-Funktion exec early in (0xA).

2. Der Netzwerktreiber empfängt den Auftrag. Er prüft, ob bereits ein Datagramm empfangen wurde, das an den Nutzerprozeß übermittelt werden kann. Ist das der Fall, so sendet er die GDP-Antwort *datagram reply* an den Nutzerprozeß. Der Nachrichtenkode hat den Wert 0x01001010. Im Vergleich zu 1. hat sich lediglich der Funktionskode geändert (reply = 0). Die IPC-Nachricht enthält neben dem Nachrichtenkode das Objekt datagram als indirect String und das Fehlerstatusobjekt mit dem Wert 0 (kein Fehler).

   Ist kein Datagramm empfangen worden, so sendet der Treiber die GDP-Antwort *nil reply* mit dem Nachrichtenkode 0x01000000, da eine sofortige Antwort (early) gefordert ist. Hier hat sich lediglich der Objekttyp geändert. Das Nil-Objekt hat den Typ 0x000. Die Nachricht enthält außerdem das Fehlerstatusobjekt (wie oben), jedoch kein datagram-Objekt.

### 2.1.2 Hardware Configurator

Die Installation und Konfiguration von L3-Gerätetreibern erfolgt interaktiv in der Systemtask *hardware configurator* [Liedtke 90]. Die installierten Gerätetreiber sind in dieser Task im Verzeichnis "\driver\NAME" abgelegt, wobei für NAME der jeweilige Treibername verwendet wird. Die Speicherung von Gerätetreibern auf einer Treiberdiskette erfolgt ebenfalls in einem Verzeichnis mit diesem Pfadnamen. Treiber können über den hardware configurator von dort geladen werden.

Ein L3-Gerätetreiber liegt als Standarddatenraum mit dem Dateinamen "stdds" im Treiberverzeichnis vor. Der Datenraum enthält den ausführbaren Kode, Daten und den Stapelspeicher. Die Konfiguration des Treibers wird über ein ELAN-Programm mit dem Namen "install" vorgenommen, das sich im gleichen Verzeichnis befindet. Das Install-Programm erfüllt folgende Aufgaben:

- Veränderung (patch) von Konfigurationsdaten innerhalb des Treiber-Datenraums zur Einstellung von treiberspezifischen Parametern
- Zuweisen von Hardware-Ressourcen (I/O-Ports, Interrupt-Quelle, Seite des physischen Speichers) zum Treiber
- Setzen der Startadresse des ausführbaren Treiberkodes und der Adresse des Stapelspeichers im Standarddatenraum

Dem Install-Programm stehen neben den allgemeinen ELAN-Möglichkeiten einige spezielle Prozeduren für die Konfigurierung von Gerätetreibern zur Verfügung, die in [Liedtke 90] im einzelnen erläutert werden. Abbildung 7 demonstriert die wichtigsten Hilfsmittel anhand eines kurzen Kodebeispiels. Die verschiedenen Konfigurationsinformationen werden innerhalb des Standarddatenraumes am Ende des (32 MB großen) eingeblendeten Adreßraumes gespeichert.

```
STDDS VAR stdds := dataspace("stdds");      (* Standarddatenraum        *)

reset requests(stdds);                      (* alle Zuweisungen löschen *)

set registers(stdds, entry, stack);         (* Eintrittspunkt und
                                               Stackpointer setzen      *)
request(stdds, interrupt line(10));         (* IRQ 10 anfordern         *)
request(stdds, io port(0x340));             (* I/O-Port 0x340 anfordern *)
```

Abb. 7: ELAN-Prozeduren für die Konfigurierung von L3-Treibern

Die Verwaltung von I/O-Adressen erfolgt gegenwärtig nur, um Kollisionen zwischen verschiedenen Treibern feststellen zu können. Ein Treiber kann auch auf I/O-Ports zugreifen, die ihm nicht zugewiesen wurden.

### 2.1.3 Netzwerktreiber

Wenn in dieser Arbeit von (Computer-) Netzwerken die Rede ist, so sind Lokale Netze (LAN) und speziell Netze auf Basis des Ethernet-Standards gemeint, die heute die größte Verbreitung haben. Die für L3 vorhandene TCP/IP-Implementierung, die neben dem Ethernet-Treiber selbst die Basis für die Netzanbindung eines L3-Systems ist, unterstützt ebenfalls nur das Format von Ethernet-Datagrammen (-Frames/-Paketen, siehe Glossar). Ein Netzwerktreiber kann zwar die Komplexität der Hardware hinter einer allgemeinen Schnittstelle verbergen. Die netzspezifischen Adressen- und Datagrammformate sind jedoch auch Gegenstand von höheren Schichten der Netzsoftware. Es ist hier nicht möglich, diese Problematik erschöpfend zu behandeln. Es wird deshalb auf Literatur verwiesen, die sich ausführlich mit Rechnernetzen befaßt (z.B. [Tanenbaum 91], [Stevens 94]).

Die Aufgabe eines L3-Ethernet-Treibers ist der Empfang von Ethernet-Datagrammen und deren Weiterleitung an einen Nutzerprozeß (ggf. an mehrere Nutzerprozesse) und die Übernahme von Datagrammen von einem Nutzerprozeß für deren Aussendung. Ein Datagramm ist aus Treibersicht eine Folge von Bytes mit dem in Abbildung 8 gezeigtem Aufbau [Stevens 94]. Der Treiber empfängt Datagramme in diesem Format vom Nutzerprozeß und vom Netzwerk und nimmt daran bestimmte Manipulationen vor. Dazu gehört das Eintragen der eigenen Ethernet-Adresse als Quelladresse und das Auffüllen des Datagramms auf die minimale Größe von 60 Byte beim Senden bzw. die Analyse der Zieladresse beim Empfang.

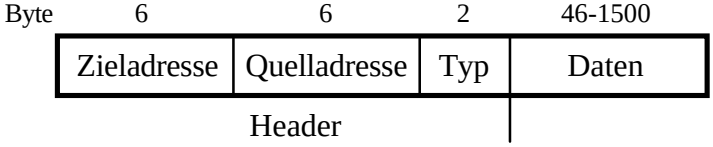


Abb. 8: Aufbau eines Ethernet-Datagramms

Im folgenden wird nun auf die Funktionsweise eines Netzwerktreibers im Betriebssystem L3 eingegangen. Zu diesem Zweck wurde der Assembler-Quellkode eines vorhanden Treibers [Schönbeck 94] analysiert.

Die Treiberfunktion wird mit einem Thread realisiert und ist in Abbildung 9 im Pseudokode sehr vereinfacht dargestellt. Nach der Initialisierung des Ethernet-Controllers wird eine Hauptschleife durchlaufen, in der über eine IPC-Wait-Operation (Empfang einer Nachricht eines beliebigen Absenders) auf eine Interrupt-Nachricht vom Systemkern oder eine GDP-Nachricht von einem Nutzerprozeß gewartet wird und nachfolgend eine Behandlung des Interrupts oder GDP-Auftrages erfolgt.

Vor dem Warten auf eine IPC-Nachricht wird dem Partnerprozeß der Empfang eines Datagramms vom Netz über eine *datagram alternate in request*-Nachricht signalisiert. Das ist

notwendig, um aufwendiges Polling zu vermeiden. Der Partnerprozeß blockiert sich ggf. selbst durch eine IPC-Empfangsoperation mit großem Timeout-Wert und wird durch die Signalisierung eines neuen Datagramms durch den Netztreiber aufgeweckt.

```
Hardware initialisieren

LOOP
      IF neues Datagramm vom Netz empfangen
            GDP_datagram_alternate_in_request an Partnerprozeß senden
      END

      auf IPC-Nachricht warten

      IF Interrupt-Nachricht empfangen
            Interrupt behandeln
      ELSE IF GDP-Auftrag empfangen
            IF GDP_datagram_exec_early_out
                  Datagramm an Netz senden
                  GDP_nil_reply an Partnerprozeß senden
            ELSE IF GDP_datagram_exec_early_in
                  IF Datagramm vom Netz empfangen
                        GDP_datagram_reply an Partnerprozeß senden
                  ELSE
                        GDP_nil_reply an Partnerprozeß senden
                  END
            ELSE andere GDP-Aufträge behandeln
            END
      END
END
```

Abb. 9: Grundstruktur eines L3-Ethernet-Treibers

Von Bedeutung sind die bei den IPC-Operationen verwendeten Timeout-Werte. Da der Treiber, wie bereits in Abschnitt 2.1.1 erläutert, bei Sendeoperationen von einem empfangsbereiten Partnerprozeß ausgehen kann, erfolgt das Senden von GDP-Antworten mit einem Timeout von 0 (sofortiger Abbruch bei Nichtbereitschaft des Empfängers). Dies ist auch nicht anders möglich, da der Treiber nur aus einem Thread besteht, der durch eine Sendeoperation mit Timeout blockiert werden könnte und in Folge dessen möglicherweise selbst nicht rechtzeitig auf Interrupts reagieren würde.

Der normale Timeout-Wert für die IPC-Wait-Operation ist *never* (kein Abbruch durch Zeitüberschreitung). Falls jedoch Datagramme vom Netz empfangen wurden, die noch nicht erfolgreich an den Partnerprozeß signalisiert werden konnten, so ist dies möglichst schnell zu wiederholen, und es wird deshalb in diesem Fall ein Timeout-Wert von 50 ms verwendet. Das passiert immer dann, wenn der Partnerprozeß zum Zeitpunkt des Aussendens der *datagram alternate in request*-Nachricht nicht empfangsbereit war. [Heinrichs 88] fordert zwar einen

Timeout-Wert von 20 ms für das Senden dieser Nachricht, was jedoch aus dem oben erläuterten Problem der Blockierung des Treiber-Threads nicht möglich ist.

Zusätzlich zum Ethernet-Treiber wurde der Quellkode des *Ethernet Dispatchers* der L3-TCP/IP-Implementierung analysiert, des Prozesses, der direkt mit dem Ethernet-Treiber kommuniziert und der also im Normalfall der Absender von GDP-Aufträgen und der Empfänger von Antworten des Treibers ist. Seine Aufgabe besteht in der Verteilung von durch den Netztreiber empfangenen Datagrammen an verschiedene Prozesse in Abhängigkeit vom Inhalt des Typfeldes (Protokoll-Identifikator) im Ethernet-Header. Datagramme die im Typfeld beispielsweise den Identifikator des IP-Protokolls (0x800) enthalten, werden an den Prozeß weitergeleitet, der für das IP-Protokoll zuständig ist. Der Ethernet Dispatcher verwendet ebenfalls das GDP für die Kommunikation mit den Protokollmanager-Prozessen, die über verschiedene GDP-Subchannel auf den Dispatcher zugreifen. Der besprochene Ethernet-Treiber macht den Ethernet Dispatcher eigentlich überflüssig, da er dessen Funktionalität integriert hat. Diese Möglichkeit wird von der TCP/IP-Software bisher jedoch nicht genutzt.

| **Objekttyp** | **Order** | **Reply** |
|---|---|---|
| nil | open (nil exec in)<br>0x01000008 | nil reply<br>0x01000000 |
| | close (nil exec out)<br>0x01000009 | nil reply<br>0x01000000 |
| datagram | datagram exec early in<br>0x0100101A | datagram reply<br>0x01001010, string(≤1514)<br>kein Datagramm vorhanden:<br>nil reply<br>0x01000000 |
| | datagram exec early out<br>0x0100101B, string(≤1514) | nil reply<br>0x01000000 |
| lan address | lan address exec in<br>0x01001208 | lan address reply<br>0x01001200, string(6) |
| | lan address exec out<br>0x01001209, string(6) | nil reply<br>0x01000000 |
| lan mode | lan mode exec in<br>0x01001218 | lan mode reply<br>0x01001210, mode, trx |
| | lan mode exec out<br>0x01001219, mode, trx | nil reply<br>0x01000000 |

Abb. 10: Für Ethernet-Treiber relevante GDP-Nachrichten

Neben den in der Abbildung 9 vorkommenden GDP-Nachrichten sind weitere möglich. [Heinrichs 88] nimmt eine Präzisierung der allgemeinen GDP-Beschreibung [Liedtke 91b] für Ethernet-Treiber vor. Abbildung 10 zeigt für jeden relevanten Objekttyp jeweils den Aufbau der Nachricht an den Treiber (Order) und dessen zugehörige Antwort-Nachricht (Reply), wobei der GDP-Nachrichtenkode hexadezimal angegeben ist. Zusätzlich in der Nachricht enthaltene Parameter sind angegeben, wobei string(n) ein indirekter IPC-Stringparameter der Länge n ist. Alle anderen Parameter sind DWords. Bei Reply-Nachrichten wird als letztes zusätzlich noch ein *error status object* übertragen (zu dessen Aufbau siehe [Liedtke 91b]), das hier weggelassen wurde. Es hat im Normalfall den Wert 0 (kein Fehler). Die Reihenfolge der Parameter ist bei den Nachrichten, die Strings enthalten, von besonderer Wichtigkeit, da der Ethernet Dispatcher als Partnerprozeß die V2-IPC-Systemrufe verwendet (siehe dazu [Hohmuth 96]).

Über den *mode*-Parameter innerhalb des *lan mode*-Objektes wird eingestellt, welche Datagramme der Treiber empfangen soll. Mögliche Werte sind 0 (nur Datagramme mit Zieladresse = eigene Adresse), 1 (zusätzlich Broadcasts), 2 (zusätzlich Broadcasts und Multicasts) und 3 (alle Datagramme). Ist das Bit $2^{16}$ gesetzt, so entfällt das sonst beim Senden geforderte Eintragen der eigenen Ethernet-Adresse als Quelladresse. Der Parameter *trx* (transceiver select) wurde für eine frühere Implementierung des Ethernet-Treibers verwendet und kann ignoriert werden.

## 2.2 Gerätetreiber in Linux

Linux ist ein Betriebssystem mit monolithischem Kern. Nach [Tanenbaum 90] ist ein derartiges System eine Ansammlung von Prozeduren, wobei jede Prozedur jede andere aufrufen kann, wenn dazu Bedarf besteht. Ein solcher Ansatz führt selten zu einer übersichtlichen Struktur. Dieses Problem ist auch beim Linux festzustellen, obwohl große Anstrengungen zur Strukturierung unternommen werden. Für die in dieser Arbeit vorgesehene systematische Übernahme von Gerätetreibern auf das L3-System durch Emulation einer andereren Systemumgebung sind wohldefinierte Schnittstellen eine Voraussetzung. Linux ist für diesen Zweck sicher nicht ideal. Wegen seiner freien Verfügbarkeit, seiner großen Verbreitung und der daraus resultierenden Vielzahl von vorhandenen Gerätetreibern wird es trotzdem als Basis verwendet. Die Informationen in diesem Abschnitt beziehen sich auf die Linux-Version 1.2.13, die zum Zeitpunkt des Beginns dieser Arbeit als letzte stabile Version galt. Viele Informationen wurden aus dem Quellkode des Linux-Kerns [Linux 95] selbst entnommen, weitere sind in [Beck 94] und [Johnson 95] enthalten.

Das Linux-System stellt sich für den Entwickler als eine große Ansammlung von Quelldateien in der Programmiersprache C sowie einiger Assembler-Quellen dar. Der Linux-Kern wird durch die Übersetzung und Bindung aller Systembestandteile zu einem Objektprogramm

erzeugt. Ein Linux-Gerätetreiber besteht in diesem Kontext ebenfalls aus einem oder mehreren C-Quelldateien. Er bietet seine Dienste über C-Funktionsschnittstellen an und greift selbst auf andere Kerndienste über Funktionsaufrufe zu. Da er im Adreßraum des Systemkerns läuft, sind für ihn auch beliebige Kernvariablen verfügbar. Die Schnittstellen sind als Funktions-Prototypen und Datenstruktur-Definitionen in C-Header-Dateien festgelegt, die über den C-Präprozessor in die Quelldateien eingefügt werden.

Die Installation eines Gerätetreibers erfolgt durch Hinzufügen seiner Quelldateien zum Quellkode des Systemkerns und dessen nachfolgende Neuübersetzung. Für die Änderung von Konfigurationsparametern kann ebenfalls die Erzeugung eines neuen Kerns erforderlich sein. Um diesen Aufwand zu umgehen, bieten viele Gerätetreiber die Möglichkeit, bestimmte Hardwareparameter (z.B. IRQ, I/O-Adressen) selbstständig zu ermitteln (*autoprobing*). Außerdem können beim Systemstart einige Parameter in Textform als Optionen an den Kern übergeben werden (Boot-Parameter) oder über Systemrufe (ioctl) eingestellt werden.

Neuere Linux-Versionen unterstützen auch das dynamische Hinzufügen und Entfernen von Gerätetreibern während der Laufzeit des Systems als sogenanntes *Module*. Der Treiber muß in diesem Fall im relokativen Objektkode-Format vorliegen und wird in den Kern-Adreßraum geladen. Dabei erfolgt ein dynamisches Binden. Das dynamische Laden wird jedoch nicht von allen Treibern unterstützt, weil u.a. hardwareabhängig die Gefahr eines Systemabsturzes während des autoprobing besteht. Diese Möglichkeit wurde deshalb nicht weiter verfolgt.

Für die Behandlung von Interrupts besteht im Linux die Wahl zwischen sogenannten schnellen und langsamen Behandlungsroutinen [Beck 94]. Während der Ausführung von schnellen Interrupt-Routinen sind weitere Interrupts gesperrt. Langsame Interrupt-Routinen können dagegen durch andere Interrupts unterbrochen werden. Dies ist der normale Fall, der auch für Ethernet-Treiber zur Anwendung kommt [Gortmaker 95]. Die Interrupt-Behandlung erfolgt in zwei Schritten. Zunächst wird eine Funktion des für den Interrupt zuständigen Gerätetreibers aufgerufen, die dieser über den Kerndienst *request_irq()* registriert hat. Diese Funktion erledigt nur die sofort notwendigen Arbeiten und aktiviert bei Bedarf durch Aufruf von *mark_bh(nr)* eine sogenannte *Bottom-Half-Routine* (*nr* selektiert eine von maximal 32 verschiedenen Routinen), die für die weitere Behandlung zuständig ist und die erst dann zur Ausführung kommt, wenn keine weiteren Interrupts anstehen. Durch diese Zweiteilung werden zeitaufwendige Operationen aus der eigentlichen Interrupt-Routine hinaus verlagert, da diese nicht durch sich selbst unterbrechbar ist und somit möglicherweise Interrupts verloren gehen könnten.

Linux unterscheidet folgende Klassen von Geräten:

- Character (z.B. Konsole)
- Block (z.B. IDE-Festplatte)
- SCSI (z.B. SCSI-Festplatte)
- Net (Netzwerkgeräte)
- Sound (Soundkarten)

Für diese Arbeit interessieren vor allem Netzwerk- und SCSI-Geräte. Abbildung 11 zeigt die Einbettung von Netzwerk- und SCSI-Treibern in den Linux-Kern. Der Zugriff eines Nutzerprozesses auf eine SCSI-Festplatte erfolgt über das virtuelle Dateisystem (VFS), den logischen Dateisystem-Treiber (Beispiel ext2), den Puffer-Cache und den SCSI-Treiber oder vom VFS direkt zum Treiber (physischer Zugriff). Der VFS (virtual file system switch) realisiert dabei lediglich eine einheitliche Schnittstelle für verschiedene Dateisysteme [Beck 94]. Ein Netzwerkgerät ist über den VFS, die Socket-Schicht, die TCP/IP-Protokoll-schicht und den Netztreiber erreichbar, wobei der VFS und die TCP/IP-Schicht umgangen werden können. Das Öffnen eines Netzwerkgerätes ist nur über den *socket*-Systemruf möglich, der über den VFS nicht erreichbar ist. Es existiert keine einheitliche Schnittstelle mit voller Funktionsfähigkeit für beide interessierende Klassen von Gerätetreibern, weshalb eine getrennte Behandlung von Netzwerk- und SCSI-Treibern notwendig ist. Aus Zeitgründen beschränkt sich diese Arbeit auf Netzwerktreiber.

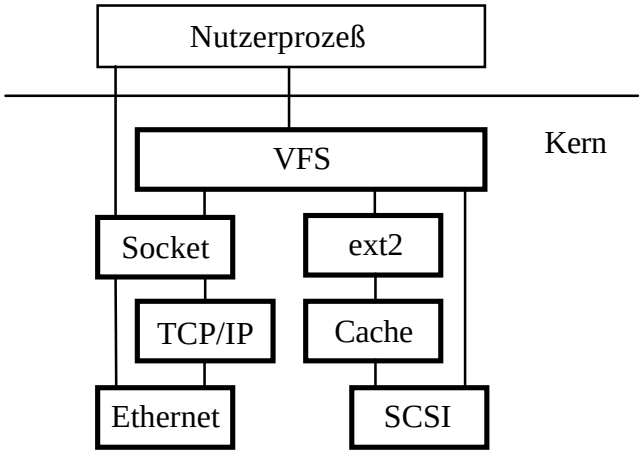


Abb. 11: Einbettung von Netzwerk- und SCSI-Treibern in den Linux-Kern

### 2.2.1 Struktur der Netzsoftware

Aus Abbildung 11 geht bereits die stark vereinfachte Struktur der Linux-Netzsoftware mit Socket-Schicht, Protokoll-Schicht (neben TCP/IP sind weitere Protokolle implementiert) und Netzwerktreiber hervor. Ein *Socket* ist ein vom Linux-Kern bereitgestelltes Abstraktionsmittel für den Endpunkt einer Netzverbindung, dessen sich Nutzerprozesse bedienen können, um mit anderen Prozessen über ein Netzwerk hinweg zu kommunizieren (siehe dazu [Stevens 95]). Diese Schnittstelle ist in Hinblick auf die vorgesehene systematische Übertragung von Linux-Netzwerktreibern auf das L3-System jedoch völlig ungeeignet, weshalb hier nicht weiter darauf eingegangen werden soll.

Ein L3-Netztreiber sendet und empfängt Ethernet-Datagramme (siehe Abschnitt 2.1.3). Er kennt keine Netzverbindungen und befindet sich unterhalb der Protokoll-Schicht, die das TCP/IP-Protokoll (oder andere) implementiert. In der Linux-Netzarchitektur existiert eine vergleichbare Schnittstelle zwischen Netztreiber und Protokoll-Schicht. Oberhalb des Netztreibers befindet sich eine relativ dünne Netzwerkgeräte-Schicht, die im Bild nicht dargestellt ist. Ihre Aufgabe besteht in der Verwaltung der ggf. mehreren Netzwerkgeräte, in der Übernahme, Pufferung und Übergabe von zu sendenden Datagrammen an den betreffenden Netztreiber bzw. in der Verteilung von empfangenen Datagrammen an die zugehörige Protokollinstanz nach Auswertung des Typfeldes im Ethernet-Header.

```
struct device
{
  char           *name;                     /* interface name        */

  /* I/O specific fields */
  unsigned long  mem_end;                   /* shared mem end        */
  unsigned long  mem_start;                 /* shared mem start      */
  unsigned long  base_addr;                 /* device I/O address    */
  unsigned char  irq;                       /* device IRQ number     */

  unsigned char  tbusy;                     /* transmitter busy flag */
  struct device  *next;                     /* link to next device   */
  unsigned char  dev_addr[MAX_ADDR_LEN];    /* interface hw address  */
  struct sk_buff_head  buffs[DEV_NUMBUFFS]; /* transmit queues       */

  /* pointers to device specific service routines */
  int (*init)(struct device *dev);
  int (*open)(struct device *dev);
  int (*stop)(struct device *dev);
  int (*hard_start_xmit)(struct sk_buff *skb, struct device *dev);
  void (*set_multicast_list)(struct device *dev, int num_addrs, void *addrs);
  struct enet_statistics* (*get_stats)(struct device *dev);
};
```

Abb. 12: device-Struktur

Für die Verwaltung von Netzwerkgeräten wird im Linux-Kern die *device*-Struktur verwendet, die in der Quelldatei netdevice.h [Linux 95] deklariert ist und die in Abbildung 12 dargestellt wird. Die Linux-Netzimplementation ist kein Muster für gute Datenabstraktion, weshalb in dieser und in anderen Strukturen Daten verschiedener Schichten gemischt sind, was das Verständnis erschwert und ein Beispiel für die Strukturierungsprobleme von monolithischen Systemen ist. Diese und alle folgenden Darstellungen sind stark vereinfacht, um den Blick auf die Gesichtspunkte zu konzentrieren, die für diese Arbeit von Bedeutung sind. Für eine vollständige Darstellung wird auf [Beck 94] verwiesen.

Für jedes Netzwerkgerät wird während der Initialisierung eine device-Struktur erzeugt und mit Werten belegt. Neben dem Namen (z.B. "eth0") und den Hardware-Parametern werden die Ethernet-Adresse (dev_addr[]) und eine Reihe von Zeigern gesetzt, über die der Zugriff auf verschiedene Funktionen des für das Netzgerät zuständigen Treibers erfolgt. Diese Funktionen bilden die eigentliche Dienstschnittstelle des Netzwerktreibers. Auf sie wird später genauer eingegangen. Im Feld *buffs[]* wird der Anker für die doppelt verkettete Liste der auf die Aussendung wartenden Datagramme verwaltet. Genaugenommen existieren drei getrennte Warteschlagen für verschiedene Prioritäten, von denen bisher jedoch nur eine genutzt wird.

Die einzelnen Datagramme sind in der *sk_buff*-Struktur (socket buffer) gespeichert, die in der Datei skbuff.h deklariert ist. Abbildung 13 zeigt zusätzlich einige Hilfsfunktionen des Kerns zum Belegen und Freigeben eines Puffers und für das Einfügen (am Listenanfang bzw. -ende) und Entfernen aus einer Liste von Puffern.

```
struct sk_buff
{
  struct sk_buff  *next;    /* linked list */
  struct sk_buff  *prev;

  struct device   *dev;     /* associated network device */

  unsigned long   len;      /* number of bytes in data[] */
  unsigned char   data[0];
};

struct sk_buff *alloc_skb(unsigned int size, int priority);
void dev_kfree_skb(struct sk_buff *skb, int mode);

void skb_queue_head(struct sk_buff_head *list, struct sk_buff *buf);
void skb_queue_tail(struct sk_buff_head *list, struct sk_buff *buf);
struct sk_buff *skb_dequeue(struct sk_buff_head *list);
```

Abb. 13: sk_buff-Struktur mit Hilfsfunktionen

Es soll nun der Ablauf beim Senden und Empfangen eines Datagramms betrachtet werden. Die Abbildungen 14 und 15 zeigen die beteiligten Funktionen der Netzwerkgeräte-Schicht (definiert in der Datei dev.c) bzw. des Netzwerktreibers. Angenommen, die IP-Protokoll-schicht will ein Datagramm senden:

1. Das Datagramm wird als sk_buff-Struktur an die Funktion *dev_queue_xmit()* übergeben. Der Puffer wird an des Ende der Sendewarteschlange *dev->buffs* angehängt und nachfolgend das erste Datagramm aus der Warteschlange entnommen (FIFO-Bedienstrategie) und zur Aussendung an die vom Treiber abhängige Funktion *dev->hard_start_xmit()* übergeben. An dieser Stelle erfolgt der Zugriff auf einen der bei der Besprechung der device-Stuktur erwähnten Funktionszeiger, die die Treiberschnittstelle bilden. In dev->hard_start_xmit sei ein Verweis auf die Funktion nic_start_xmit (Abb. 15) eingetragen, die nun die Kontrolle erhält. Der konkrete Name dieser Funktion ist treiberabhängig. NIC (network interface controller) steht hier für einen beliebigen Netzwerk-Controller.

```
void dev_queue_xmit(struct sk_buff *skb, struct device *dev)
{
    skb_queue_tail(dev->buffs, skb);
    skb = skb_dequeue(dev->buffs);

    if(dev->hard_start_xmit(skb, dev) != 0)
        skb_queue_head(dev->buffs, skb);
}

void netif_rx(struct sk_buff *skb)
{
    skb_queue_tail(&backlog, skb);
    mark_bh(NET_BH);
}

void net_bh()
{
    while(   !dev->tbusy
          && (skb = skb_dequeue(dev->buffs)) != NULL)
    {
        if(dev->hard_start_xmit(skb, dev) != 0)
            skb_queue_head(dev->buffs, skb);
    }

    while((skb = skb_dequeue(&backlog)) != NULL)
        ip_rcv(skb, skb->dev);
}
```

Abb. 14: Die Funktionen dev_queue_xmit, netif_rx und net_bh (dev.c)

```
int nic_start_xmit(struct sk_buff *skb, struct device *dev)
{
    if(dev->tbusy)
        return 1;

    dev->tbusy = 1;

    output_to_nic(skb->data, skb->len);

    if(!nic_buffer_full)
        dev->tbusy = 0;

    dev_kfree_skb(skb, FREE_WRITE);

    return 0;
}

void nic_interrupt(int irq, struct pt_regs *regs)
{
    struct device *dev = irq2dev_map[irq];

    while(interrupt_pending)
    {
        if(is_transmit_interrupt)
        {
            dev->tbusy = 0;
            mark_bh(NET_BH);
        }

        if(is_receive_interrupt)
        {
            get_len_from_nic(packet_len);

            if((skb = alloc_skb(packet_len, GFP_ATOMIC)) != NULL)
            {
                input_from_nic(skb->data, packet_len);

                skb->len = packet_len;
                skb->dev = dev;

                netif_rx(skb);
            }
        }
    }
}
```

Abb. 15: Die Funktionen nic_start_xmit und nic_interrupt des Netzwerktreibers

2. *nic_start_xmit()* testet zunächst, ob der Netzadapter noch ausreichend freien Pufferspeicher zur Aufnahme des Datagramms hat. Dieser Zustand wird in der device-Struktur im Feld *dev->tbusy* (transmitter busy) vermerkt. Ist der Adapter bereits belegt, so erfolgt eine sofortige Rückkehr mit dem Resultat 1. Daraufhin hängt dev_queue_xmit() den Puffer wieder am Anfang der Sendewarteschlange ein, wo er auf den nächsten Sendeversuch wartet. Im anderen Fall wird das Netzgerät nun temporär als belegt markiert und das

Datagramm an die Hardware ausgegeben. Sollte der Netzadapter danach immernoch genug Pufferspeicher zur Aufnahme eines weiteren Datagramms haben (der verwendete 3Com-3C595-Controller kann beispielsweise bis zu 11 Datagramme der Maximalgröße von 1514 Byte aufnehmen), so wird dies in dev->tbusy vermerkt. Nachfolgend erfolgt die Freigabe der sk_buff-Strukur und die Rückkehr zu dev_queue_xmit(), die die Kontrolle an den Aufrufer zurückgibt.

3. Wenn der Netzadapter die Aussendung beendet hat, erzeugt er einen Interrupt, der zum Aufruf der Funktion *nic_interrupt()* führt, die zunächst anhand des übergebenen IRQ (Interrupt-Quelle) über das Kern-globale Feld *irq2dev_map[]* die zugehörige device-Struktur ermittelt. Das ist notwendig, weil ein Treiber ggf. mehrere Netzadapter bedienen kann, die verschiedene Interrupt-Quellen darstellen. Danach werden alle anstehenden Interrupt-Bedingungen dieses Gerätes behandelt. Im Falle eines Sendebestätigungs-Interrupts (is_transmit_interrupt) wird der Sender als frei (tbusy = 0) markiert und mit *mark_bh(NET_BH)* die Bottom-Half-Routine der Netzgeräte-Schicht (die Funktion *net_bh()*) zur Ausführung markiert. NET_BH ist eine Konstante mit dem Wert 4.

4. *net_bh()* (Abb. 14) erhält im Anschluß an die Interrupt-Routine nic_interrupt() die Kontrolle und versucht, solange der Sender nicht belegt ist und Datagramme in der Warteschlange sind, diese für einen erneuten Sendeversuch an den Treiber zu übergeben. Scheitert die Aussendung wird der Puffer wieder in die Liste eingekettet. Die Möglichkeit des Vorhandenseins mehrerer aktiver Netzgeräte ist hier nicht berücksichtigt. In Wirklichkeit wird die while-Schleife nacheinander für jedes aktive Gerät durchlaufen.

Betrachten wir nun den Empfang eines Datagramms:

1. Der Netzadapter hat ein Datagramm vom Netz empfangen und löst einen Interrupt aus, was zum Aufruf von *nic_interrupt()* führt. Diesmal wird der Empfangs-Interrupt (is_receive_interrupt) behandelt. Der Treiber ermittelt die Länge des empfangenen Datagramms (*get_len_from_nic*), belegt eine sk_buff-Struktur von entsprechender Größe und übernimmt das empfangene Datagramm von der Hardware in den Puffer (*input_from_nic*). Das Datagramm wird durch Aufruf der Funktion *netif_rx()* an die Netzgeräte-Schicht übergeben.

2. *netif_rx()* fügt den Puffer an das Ende der Empfangswarteschlange (*backlog*) an und aktiviert (wie oben) die Bottom-Half-Routine.

3. *net_bh()* wird im Anschluß an die Interrupt-Behandlung aufgerufen. In der zweiten while-Schleife von net_bh() werden die empfangenen Datagramme aus der Warteschlange entnommen und an die zuständigen Protokollinstanzen übergeben. Die Abbildung zeigt beispielhaft den Aufruf der Funktion *ip_rcv()*, die IP-Pakete entgegennimmt. In der Realität ist dieser Vorgang komplexer. Die Netzgeräte-Schicht verwaltet eine Liste von Pakettypen

und zugehörigen Empfangsfunktionen. Diese Funktionalität wird für den L3-Netztreiber jedoch nicht benötigt.

### 2.2.2 Netzwerktreiber

Nachdem der letzte Abschnitt die Abläufe beim Senden und Empfangen von Datagrammen dargestellt hat, soll nun die Kern-Schnittstelle eines Netzwerktreibers genau beschrieben werden. Diese stellt sich als eine Menge von C-Funktionen dar. Diese Informationen wurden durch Analyse des Quellkodes eines konkreten Treibers [Becker 95] gewonnen. Die angegebenen Funktionsnamen sind lediglich Beispiele.

Für die Initialisierung eines Netztreibers existieren zwei verschiedene Möglichkeiten. Welche Variante zur Anwendung kommt, hängt davon ab, ob ein Treiber DMA-Operationen (direkter Speicherzugriff) durchführt.

**int nic_probe(struct device *dev)**

Diese Funktion wird während des Systemstarts (in der Quelldatei Space.c) für alle eingebundenen Treiber aufgerufen und überprüft, ob ein vom jeweiligen Treiber unterstützter Netzadapter im System vorhanden ist. Dabei werden ggf. verschiedene I/O-Adressen abgesucht. Ist die Suche erfolgreich, so erfolgt eine Grundinitialisierung. Dazu gehört das Eintragen von Zeigern auf die nachfolgend besprochenen Funktionen in die übergebene *device*-Struktur (Abb. 16), über die der Zugriff zu diesen erfolgt. Im Erfolgsfall gibt die Funktion den Wert 0 zurück.

```
dev->open               = &nic_open;
dev->stop               = &nic_close;
dev->hard_start_xmit    = &nic_start_xmit;
dev->get_stats          = &nic_get_stats;
dev->set_multicast_list = &set_multicast_list;
```

Abb. 16: Setzen der Funktionszeiger in nic_probe()

**unsigned long nic_init(unsigned long mem_start, unsigned long mem_end)**

Netzwerktreiber, die DMA-Operationen durchführen, benötigen Speicher, der innerhalb der ersten 16 MB liegt, da der DMA-Controller keinen Speicher über dieser Grenze adressieren kann. Aus historischen Gründen wird der Speicher über diese spezielle Initialisierungsfunktion bereitgestellt, die anstelle von *nic_probe()* (in der Quelldatei net_init.c) aufgerufen wird und als Parameter die Anfangs- und Endadresse des verfügbaren DMA-fähigen Speichers erhält. Die Funktion liefert als Resultat die erste freie Adresse nach dem von ihr belegten Speicherbereich. Ihre sonstigen Aktivitäten entsprechen denen von nic_probe(). Da sie jedoch keine device-

Strukur als Parameter übergeben erhält, muß diese anderweitig bereitgestellt werden. Zu diesem Zweck existiert die Hilfsfunktion *init_etherdev()*, die außerdem u.a. einige Felder innerhalb der Struktur auf Ethernet-typische Werte setzt.

**int nic_open(struct device *dev)**

Diese Funktion aktiviert den Netzwerk-Controller. Sie installiert eine Funktion zur Interrupt-Behandlung und gibt den Wert 0 zurück, wenn kein Fehler auftritt.

**int nic_close(struct device *dev)**

Diese Funktion deaktiviert den Netzwerk-Controller. Ihr Resultat ist (soweit bekannt) immer 0, da ein Fehler nicht auftreten kann.

**int nic_start_xmit(struct sk_buff *skb, struct device *dev)**

Diese Funktion wurde bereits im letzten Abschnitt besprochen (Abb. 15). Sie versucht, das übergebene Datagramm an den Netzadapter zur Aussendung auszugeben und liefert im Erfolgsfall 0 zurück.

**struct enet_statistics *nic_get_stats(struct device *dev)**

Die von dieser Funktion zurückgegebene Datenstruktur enthält verschiedene Statistikwerte für das Netzgerät, wie die Anzahl der gesendeten und empfangenen Datagramme und der aufgetretenen Fehler. Die Struktur enet_statistics ist in der Quelldatei if_ether.h deklariert.

**void set_multicast_list(struct device *dev, int num_addrs, void *addrs)**

Über diese Funktion wird der Multicast-Filter des Netzwerk-Controllers gesetzt. *addrs* zeigt auf ein Feld mit Multicast-Adressen (6 Byte je Adresse) und *num_addrs* gibt deren Anzahl an. Bei num_addrs = 0 (Standardeinstellung) werden nur Unicasts und Broadcasts empfangen. num_addrs = -1 aktiviert den *Promiscuous Mode*, in dem alle Datagramme unabhängig von der Zieladresse empfangen werden. (zu den Begriffen siehe Glossar)

## 2.3 Emulation für Linux-Gerätetreiber in Mach

[Goel 96] beschreibt das Design und die Leistungsfähigkeit einer Emulation für Linux-Gerätetreiber innerhalb des Mach-Mikrokerns (Mach 4.0, Version UK02p21), die auch als Quellkode verfügbar ist [Mach 95]. Durch Zusätze zum Mach-Kern (ca. 2000 Zeilen C-Kode), Änderungen an diesem und die Emulation von Linux-Kernfunktionen wurde die Möglichkeit geschaffen, alle Netzwerk- und SCSI-Gerätetreiber für den ISA- und PCI-I/O-Bus der Linux-Version 1.3.35 unmodifiziert in den Mach-Kern zu übernehmen. In [Mach 95] sind jedoch nur Netzwerktreiber und keine PCI-Unterstützung enthalten.

Ausgangspunkt der Entwicklung war die schlechte Verfügbarkeit von Gerätetreibern für den Mach-Mikrokern, der die Notwendigkeit der Nutzung von neu entwickelter I/O-Hardware gegenüberstand. [Goel 96] geht zunächst auf die Ursachen dieses Problems ein und schlägt dann als Ausweg die Emulation der Treiberschnittstellen des Linux-Kerns und die damit mögliche Übernahme der umfangreichen Linux-Treiberbasis auf das Mach-System vor. Als Voraussetzung wird das Vorhandensein von gut definierten und relativ stabilen Schnittstellen hervorgehoben.

Das Papier [Goel 96] ist hier vor allem aus folgenden Gründen interessant:

1. Sein Gegenstand entspricht im wesentlichen genau der Problematik der hier vorliegenden Arbeit. Es wird mit Linux die gleiche Ausgangsbasis verwendet. Unterschiedlich ist lediglich das Zielsystem. Damit läßt sich erwarten, daß bestimmte Informationen über das Linux-System und die bei der Entwicklung gemachten Überlegungen und Entwurfsentscheidungen auch hilfreich für diese Arbeit sind.
2. Es ist die bisher einzig bekannt gewordene Veröffentlichung zur Übernahme von unverändertem Quellkode von Gerätetreibern eines Betriebssystems in ein anderes, wie auch von den Autoren selbst festgestellt wird. Sie ist deshalb vor allem auch aus methodischen Gesichtpunkten von Interesse.

Der Inhalt des Papiers soll hier nicht im einzelnen wiedergegeben werden. Es wird hauptsächlich das Vorgehen bei der Entwicklung betrachtet. Wo dies von Bedeutung ist, erfolgt außerdem, wie schon in anderen Abschnitten, eine Beschränkung auf Netzwerktreiber.

[Goel 96] geht zunächst auf die Funktion von Gerätetreibern in Linux 1.3.35 und Mach 4.0 ein, speziell auf die:

- Möglichkeit der statischen oder dynamischen Einbindung von Treibern in den Linux-Kern (Mach unterstützt nur die statische Einbindung, die deshalb verwendet wird.)
- von Linux unterschiedenen Geräteklassen (Jede Klasse hat eine eigene Schnittstelle.)
- Schichtenstruktur der Linux-Treiber (generische und hardwareabhängige Schicht bei SCSI-Treibern, bei Netzwerktreibern keine Schichtung)

Bei der nachfolgenden Betrachtung der Treiberschnittstellen wird festgestellt, daß die Emulation auf zwei Aufgaben reduziert werden kann:

1. Nachbildung einer jeden Funktion und jeder Variablen, die ein Linux-Treiber benutzen könnte
2. Implementierung aller Funktionen der Mach-Treiberschnittstelle durch die Kombination eines Linux-Treibers mit dem Emulationskode

Die Erfüllbarkeit des zweiten Punktes kommt bei dem vorgenommenen Vergleich der Schnittstellen beider Systeme zum Ausdruck. Es erfolgt eine Abbildung von Funktionen der

Mach-Treiberschnittstelle auf die der Linux-Schnittstelle. Wo diese nicht direkt möglich ist, weil keine äquivalente Funktion existiert, wird eine andere Lösung gesucht. Beispielsweise ist die Mach-Funktion *device_set_filter*, die einen Datagramm-Filter mit einem Netzwerktreiber verbindet, durch einen Linux-Treiber nicht implementierbar. Da jedoch praktisch kein Mach-Treiber diese Funktion selbst realisiert, sondern alle auf eine generische Funktion des Mach-Kerns zurückgreifen, kann dies für Linux-Treiber auch generell erfolgen. Funktionen, die zwar definiert sind, bisher jedoch von niemand benutzt werden (z.B. *device_map*), müssen zunächst nicht implementiert werden.

Unterhalb der definierten Mach-Treiberschnittstelle und über den Gerätetreibern existiert eine Schicht generischen Kodes, der von allen Mach-Treibern verwendet wird, d.h. der einzelne Treiber hält sich gar nicht an die definierte Schnittstelle, sondern benutzt eine implementationsabhängige Schnittstelle, die für Linux-Treiber aber nicht brauchbar ist. Dieser generische Kode muß deshalb geändert werden, um den direkten Zugang zur eigentlichen Treiberschnittstelle herzustellen.

Bei der Nachbildung der Umgebung, in der ein Linux-Treiber läuft, ergeben sich einige Probleme, deren Lösung in [Goel 96] beschrieben wird. Von besonderem Interesse ist die Adressierung von Speicher im Linux-Kern. Linux blendet das Kernsegment in das obere Gigabyte des 32-Bit-Adreßraums (ab Adresse 0xC0000000) ein und lädt die Segmentregister des Prozessors mit der Basisadresse 0xC0000000, so daß virtuelle Adressen ab 0 beginnend generiert werden. Daraus folgt die Übereinstimmung von virtuellen und physischen Adressen innerhalb des Linux-Kerns. Linux-Treiber müssen deshalb keine Übersetzung von Adressen durchführen, wenn (wie für DMA-Operationen) physische Adressen benötigt werden. Mach verwendet jedoch eine Basisadresse von 0 und generiert virtuelle Adressen ab 0xC0000000, weshalb Mach-Treiber eine Adreßumrechnung vorzunehmen haben. Diese Differenz ließ sich nur durch eine Änderung am Mach-Kern beheben. Er verhält sich in dieser Hinsicht nun wie der Linux-Kern.

Der Linux-Kern bietet Dienste zur Verwaltung von Hardware-Ressourcen (z.B. IRQ, I/O-Adreßbereiche) an, die von Treibern genutzt werden. Mach bietet keine derartige Unterstützung. Der entsprechende Kode wurde deshalb von Linux portiert. Da sich die normalen Mach-Treiber jedoch nicht an diese Verwaltungsmechanismen halten, sollten sie nicht gleichzeitig mit Linux-Treibern verwendet werden.

Zur Bewertung der entstandenen Lösung wurden Vergleichsmessungen an einem durch die Emulation in Mach eingebundenen Linux-Treiber und einem herkömmlichen Mach-Treiber vorgenommen, wobei die Messung der Laufzeit eines Kodeabschnittes innerhalb des Treibers mit Hilfe der RDMSR-Instruktion des Pentium-Prozessors erfolgte. Die Autoren stellen fest, daß sich durch die Emulation nur eine sehr geringe Erhöhung der Laufzeit ergibt. Teilweise war der emulierte Treiber sogar schneller, wofür keine Erklärung gegeben werden konnte. Es wird jedoch nicht dargestellt, was den herkömmlichen Mach-Treiber als Vergleichskriterium

prädestiniert. Es besteht die theoretische Möglichkeit, daß dieser nicht optimal implementiert ist, wodurch die Bewertung der Emulationslösung zu gut ausfallen würde. Die Autoren schließen mit der Feststellung, daß sich Gerätetreiber gut für Emulationen eignen, solange der I/O-Bus langsam und Prozessoren schnell sind und somit die Ausführung einer überschaubaren Anzahl von Maschinenbefehlen eines Emulators ohne merklichen Einfluß auf die Leistungsfähigkeit bleibt.

In Auswertung von [Goel 96] läßt sich feststellen, daß die dort gewählte Vorgehensweise bei der Entwicklung und die aufgeworfenen Problemstellungen auch für die vorliegende Arbeit von Bedeutung sind und beim nachfolgenden Entwurf berücksichtigt werden können. Ebenso sind die nicht von vornherein zu erwartenden guten Leistungswerte der Mach-Treiberemulation eine weitere Motivation für das hier vorgesehene äquivalente Verfahren.

Im Unterschied zu der Emulation für Linux-Gerätetreiber in Mach erfolgt hier nicht die Übernahme des Treiber-Quellkodes aus dem Linux-Kern in einen anderen Systemkern. Die Kernumgebung eines Linux-Treibers soll vielmehr innerhalb eines Nutzerprozesses nachgebildet werden, in den dadurch ein Treiber eingebunden werden kann, um die Funktion eines L3-Gerätetreibers zu gewährleisten. Daraus ergeben sich einige zusätzliche Probleme, wie die Abbildung der Synchronisationsbeziehungen im Linux-Kern (Interrupt-Behandlung) auf L3-Threads.

# 3 Entwurf

Nachdem im vorangegangenen Kapitel der Aufbau und die Funktionsweise von Gerätetreibern und speziell von Netzwerktreibern in den Betriebssystemen L3 und Linux analysiert wurde, sollen nun im einzelnen Problemstellungen erörtert werden, die beim Entwurf des Rahmens eines L3-Netzwerktreibers von Bedeutung sind, in den der Quellkode eines Linux-Netztreibers in unveränderter Form übernommen werden kann. Für die verschiedenen Probleme werden Lösungsalternativen (falls vorhanden) mit ihren Vor- und Nachteilen dargestellt. Einige Entscheidungen kamen bereits in Kapitel 2 zur Sprache, da an verschiedenen Stellen eine Beschränkung erfolgen mußte, für die jeweils eine Begründung gegeben wurde.

## 3.1 Ausgangspunkte und Problembereiche

Die Entwicklung von Gerätetreibern in dieser Arbeit ist als Beitrag zu der in der Einleitung erwähnten (Multiserver-) UNIX-Emulation [Wolter 95] vorgesehen, weshalb die Treiberschnittstelle speziell auf deren Anforderungen ausgerichtet werden sollte. Da die Arbeiten an der UNIX-Emulation jedoch noch nicht ausreichend weit fortgeschritten sind, mußte darauf verzichtet werden. Es wird deshalb die normalerweise für L3-Treiber geltende Schnittstellenkonvention (GDP) verwendet, die Thema von Abschnitt 2.1.1 war.

Als Ausgangspunkte für den Entwurf lassen sich formulieren:

- Entwicklung eines auf dem L3-System in einem Nutzerprozeß lauffähigen Netzwerktreibers
- Verwendung des GDP als Treiberschnittstelle
- Realisierung des hardwareabhängigen Teils des Treibers durch Einbindung eines Linux-Netzwerktreibers in Quellkodeform, Emulation der Linux-Umgebung des Treibers
- Der Quellkode des Linux-Treibers wird möglichst unverändert übernommen.
- Es werden keine Änderungen am L3-Systemkern vorgenommen.

Die beim Entwurf zu lösenden Probleme können in folgende Bereiche unterteilt werden, auf die in den nächsten Abschnitten eingegangen wird:

1. Abbildung der Nebenläufigkeiten im Linux-Kern auf L3-Threads
2. Nachbildung der Speicheradressierung innerhalb des Linux-Kerns
3. Abbildung der L3-Treiberschnittstelle (GDP) auf die Funktionen der Linux-Schnittstelle
4. Emulation der Dienste des Linux-Kerns

## 3.2 Entwurf der Thread-Struktur

Innerhalb des Linux-Kerns bestehen verschiedene Nebenläufigkeiten, die Auswirkung auf die Ausführung eines Gerätetreibers haben. Soll ein Linux-Treiber in einem L3-Nutzerprozeß ablaufen, so sind die Synchronisationsbeziehungen innerhalb des Linux-Kerns auf vom L3-Kern bereitgestellte Mittel abzubilden. Für die Realisierung von Nebenläufigkeiten innerhalb einer L3-Task werden Threads zur Verfügung gestellt [Wolter 95]. Dieser Abschnitt beschäftigt sich mit dem Entwurf einer Thread-Struktur für den zu entwickelnden Netzwerktreiber. Dabei wird die Frage geklärt, wieviele Threads zur Ausführung eines Linux-Treibers benötigt werden und in welcher Beziehung diese zueinander stehen.

Der Zugriff auf einen Netztreiber erfolgt im Linux durch die Netzwerkgeräte-Schicht. Betrachtet man das Senden und den Empfang eines Datagramms, so ergeben sich folgende Ausführungspfade (siehe Abschnitt 2.2.1):

1. Die Funktion *dev_queue_xmit()* der Netzgeräte-Schicht ruft die Funktion *nic_start_xmit()* des Treibers. Dies kann gleichgesetzt werden mit dem Aufruf einer beliebigen anderen Funktion der Treiberschnittstelle, die in Abschnitt 2.2.2 erläutert wurde.
2. Beim Auftreten eines Interrupts wird die vom Netztreiber installierte Behandlungsroutine *nic_interrupt()* aufgerufen, die entweder selbst *mark_bh(NET_BH)* ausführt (Sendebestätigungs-Interrupt) oder dies über den Aufruf der Funktion *netif_rx()* tut (Empfangs-Interrupt).
3. Nach Ausführung von *nic_interrupt()* wird die Funktion *net_bh()* als Bottom-Half-Routine aufgerufen, sobald keine weiteren Interrupts behandelt werden müssen. *net_bh()* wird beim erneuten Auftreten eines Interrupts ggf. unterbrochen und nach Beendigung von *nic_interrupt()* fortgesetzt.

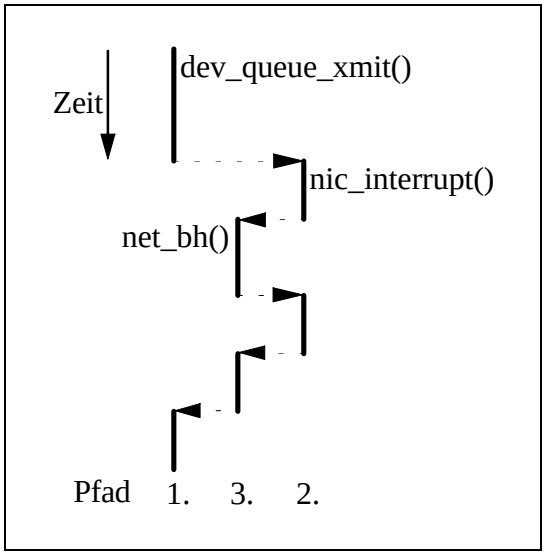


Abb. 17: Ausführungspfade in der Netzwerkgeräte-Schicht

### 3.2.1 Lösung mit drei Threads

Zur genauen Nachbildung des zuvor beschriebenen und in Abbildung 17 illustrierten Ablaufs innerhalb einer L3-Task werden offensichtlich drei Threads benötigt. Ein L3-Netztreiber muß zusätzlich noch die Interprozeßkommunikation mit seinem Partnerprozeß durchführen (siehe Abschnitt 2.1.3). Da das Senden eines Datagramms durch Aufruf von *dev_queue_xmit()* jeweils nach dessen Übermittlung mit einer GDP-Nachricht des Partnerprozesses erfolgt, wäre die Interprozeßkommunikation auf Basis des GDP Teil des ersten Ausführungspfades. Der erste Thread wartet damit ständig auf einen GDP-Auftrag und ruft je nach dessen Art ggf. eine Funktion der Linux-Treiberschnittstelle auf. Das Auftreten eines Interrupts wird einem L3-Thread über eine Nachricht vom Kern zugestellt (Abschnitt 2.1), auf die der zweite Thread wartet und bei Erhalt die Interrupt-Routine *nic_interrupt()* ausführt und nachfolgend den dritten Thread zur Abarbeitung der Funktion *net_bh()* anstößt.

Dabei ergibt sich das Problem der Thread-Synchronisation. Da im Linux zu jedem Zeitpunkt immer nur einer der drei Ausführungspfade aktiv ist (nur ein Prozessor), darf auch immer nur ein ganz bestimmter Thread ausgeführt werden. Ein L3-Thread hat einen Zustand und eine Priorität. Der Zustand gibt darüber Auskunft, ob der Thread momentan ausgeführt wird bzw. ausführbar (*bereit*) ist oder ob er auf das Eintreten eines bestimmten Ereignisses wartet (*wartend*). Der L3-Scheduler wählt aus der Menge der ausführbaren Threads jeweils einen aus, der den Prozessor erhält. Bei dieser Entscheidung berücksichtigt er die Thread-Priorität. Threads mit einer hohen Priorität erhalten mehr Prozessorzeit als niedriger priorisierte. Es werden jedoch auch dann Threads mit geringer Priorität ausgeführt, wenn Threads mit höherer Priorität ausführbar sind. Damit kann der Prioritätswert nicht zur Synchronisierung von Threads einer Task verwendet werden.

Ein Thread, der gerade ausgeführt wird, kann durch Aufruf einer IPC-*Receive*-Operation von sich aus in den Zustand *wartend* wechseln. Beim Empfang einer IPC-Nachricht wird er vom Systemkern wieder für *bereit* erklärt. Privilegierte Prozesse, zu denen Gerätetreiber gehören, können auch einen bestimmten Thread über einen Systemruf von der weiteren Ausführung ausschließen (*blockieren*) und später wieder freigeben. Wie sich die Nebenläufigkeiten in der Linux-Netzwerkgeräte-Schicht und dem Netztreiber mit diesen Mitteln zur Synchronisation auf drei Threads abbilden lassen, zeigt Abbildung 18.

Der erste Thread (*gdp_thread*) empfängt GDP-Aufträge von beliebigen Threads und behandelt diese. Er bedient sich dazu der Funktionen des Netzwerktreibers. Der zweite Thread wartet auf Interrupt-Nachrichten. Beim Auftreten eines Interrupts blockiert er die anderen Threads in der weiteren Ausführung, ruft die Behandlungsroutine des Netztreibers (*nic_interrupt()*) und gibt den dritten Thread (*bh_thread*) wieder frei. Dieser übernimmt die Abarbeitung der Bottom-Half-Routine *net_bh()* und muß dazu ggf. vom Interrupt-Thread durch Senden einer Nachricht angestoßen werden, falls er sich beim Auftretens des Interrupts im Wartezustand befindet. Sein Zustand wird im Flag *bh_running* vermerkt.

```
gdp_thread()
{
    while(1)
    {
        ipc_receive(any_thread);
        ...
        dev_queue_xmit();
        ...
    }
}

interrupt_thread()
{
    while(1)
    {
        ipc_receive_interrupt();

        block(gdp_thread);
        block(bh_thread);

        nic_interrupt();    /* may set bh_active */

        unblock(bh_thread);

        if(!bh_running)
            if(bh_active)
            {
                bh_running = 1;
                ipc_send(bh_thread, never_timeout);
            }
            else
                unblock(gdp_thread);
    }
}

bh_thread()
{
    while(1)
    {
        ipc_receive(interrupt_thread);

        while(bh_active)
        {
            bh_active = 0;
            net_bh();
        }

        bh_running = 0;
        unblock(gdp_thread);
    }
}
```

Abb. 18: Abbildung der Nebenläufigkeiten auf drei Threads

Die Notwendigkeit der Ausführung der Bottom-Half-Routine wird in der Funktion *nic_interrupt()* durch *mark_bh()* angezeigt, was lediglich zum Setzen des Flags *bh_active* führt. *bh_thread* setzt das Flag zurück und ruft die Funktion *net_bh()*. Wenn nach deren Ausführung *bh_active* nicht schon wieder gesetzt ist, so ist die Ausführung der Bottom-Half-Routine beendet und der GDP-Thread wird wieder zur Ausführung zugelassen (*unblock*). Falls *bh_thread* im Wartezustand (*!bh_running*) war und auch nicht aktiviert werden mußte, so wurde der GDP-Thread schon vom Interrupt-Thread selbst wieder freigegeben.

Es stellt sich die Frage, ob die dargestellten Nebenläufigkeiten überhaupt exakt emuliert werden müssen und ob nicht eine Serialisierung der verschiedenen Operationen erfolgen kann. Der Aufwand zur Thread-Synchronisation (einige Systemrufe je Interrupt) ist nicht unerheblich. Dazu kommt, daß beim Empfang eines Datagramms auch eine IPC-Nachricht vom Bottom-Half-Thread an den GDP-Thread gesendet werden muß, da dieser vermutlich auf eine Nachricht vom Partnerprozeß wartet, welcher nun durch eine *datagram alternate in request*-Nachricht über den Empfang des Datagramms informiert werden muß (vergleiche Abb. 9). Diese Nachricht kann aber nur vom GDP-Thread selbst gesendet werden, weil er als Kommunikationspartner über seinen Thread-ID identifierbar ist. Der Bottom-Half-Thread ist beim Partnerprozeß nicht bekannt.

Verzichtet man auf den Bottom-Half-Thread, falls dies möglich ist, reduziert sich der Synchronisationsaufwand im Interrupt-Thread auf die *block/unblock*-Operationen für den GDP-Thread. Die Bottom-Half-Routine könnte dann alternativ durch den Interrupt-Thread oder den GDP-Thread aufgerufen werden, wobei der zweite Fall offensichtlich sinnvoller ist, um die Verlagerung von zeitaufwendigen Operationen aus der eigentlichen Interrupt-Routine in die Bottom-Half-Routine nicht aufzuheben. Über den Empfang eines Datagramms muß der GDP-Thread jedoch ebenfalls durch eine IPC-Nachricht vom Interrupt-Thread informiert werden, damit er die *datagram alternate in request*-Nachricht an den Partnerprozeß senden kann (siehe oben).

### 3.2.2 Lösung mit einem Thread

Der analysierte L3-Netzwerktreiber [Schönbeck 94] verwendet nur einen einzelnen Thread. Ist es eventuell auch hier möglich, alle Operationen sequentiell auszuführen (Abb. 19)? Wann hat die Verwendung von mehreren Threads überhaupt Sinn? In [Tanenbaum 95] heißt es: "Threads wurden eingeführt, um Parallelität kombiniert mit sequentieller Ausführung und blockierenden Systemrufen zu ermöglichen". Durch Überlegung kann man auch feststellen, daß die Ausführung von eigentlich quasi-parallelen Operationen, deren einzelne Dauer sehr gering ist, ohne merklichen Einfluß auch in einer festen Reihenfolge möglich ist. Da durch den Scheduler ohnehin eine Serialisierung vorgenommen wird, besteht der Nachteil einer Lösung ohne speziellen Interrupt-Thread einzig in der verzögerten Reaktion auf Interrupts. Betrachtet man

die durch den Netztreiber durchzuführenden Operationen, so fällt lediglich die Übertragung von Datagrammen über die Interprozeßkommunikation und deren Ausgabe an die Hardware (bei Nutzung von I/O-Befehlen) zeitlich ins Gewicht. Die Eingabe eines Datagramms vom Netzadapter wird in der Interrupt-Routine selbst vorgenommen und ist ohnehin nicht unterbrechbar.

```
single_thread()
{
    while(1)
    {
        ipc_receive(any_thread_or_interrupt);

        if(is_interrupt_message)
        {
            nic_interrupt();

            if(bh_active)
            {
                bh_active = 0;
                net_bh();
            }
        }
        else
            handle_gdp_order();
    }
}
```

Abb. 19: Sequentielle Ausführung mit einem Thread

Eine IPC-Operation mit einer Nachrichtengröße, die der maximalen Datagrammgröße des Ethernet (1514 Byte) entspricht, dauert bei einem 486-Prozessor mit 50 MHz Taktfrequenz ca. 40 µs [Liedtke 93]. Die Ausgabe eines Datagramms dieser Größe an die Hardware benötigt mit dem *OUTSD*-Befehl (je 4 Byte parallel) auf dem Pentium-Prozessor (13 Takte je Wort [Podschun 95]) bei gleicher Taktrate (1514/4 * 13) / 50 ≈ 100 µs. Die größte anzunehmende Verzögerung auf einen Interrupt ergibt sich bei der aufeinanderfolgenden Ausgabe von mehreren Datagrammen durch Aufrufe von *dev->hard_start_xmit()* in der Funktion *net_bh()* (Abb. 14). Bei dem für diese Arbeit verwendeten 3C595-Controller würde dies im ungünstigsten (vermutlich seltenen) Fall (Ausgabe von 11 Datagrammen in Folge) etwa 1.1 ms in Anspruch nehmen, um die die Übernahme eines empfangenen Datagramms nach dem Empfangs-Interrupt verzögert würde. Die zusätzliche Verzögerung durch eine IPC-Operation kann dabei vernachlässigt werden. In dieser Zeit können auf dem Ethernet (10 MBit/s) 1375 Byte übertragen werden, deren Pufferung für keinen Netzadapter ein Problem darstellen dürfte. Bei dem neueren Fast Ethernet (100 MBit/s) wären 13750 Byte zu puffern. Der 3C595-Fast-Ethernet-Controller hat für den Empfang einen Pufferspeicher von 48 KByte.

Der Verzicht auf die Nutzung mehrerer Threads scheint also möglich. Welche weiteren Voraussetzungen müssen dafür erfüllt sein?

1. Der Linux-Netzwerktreiber darf an keiner Stelle aktiv auf einen Interrupt warten (*busy waiting*). Durch die sequentielle Ausführung würde der Interrupt nie behandelt werden.
2. Der Treiber darf keine aufwendigen Operationen durchführen oder solche, die zu seinem Blockieren führen können.

Von der Erfüllung der ersten Voraussetzung kann nach Analyse einiger Treiber ausgegangen werden. Aus 2. folgt für die Implementierung des Treibers:

- IPC-Operationen müssen einen Timeout-Wert von 0 verwenden. Diese Forderung wird von [Schönbeck 94] erfüllt. Hier kommt der in Abschnitt 2.1.1 erwähnte Nutzen der IPC-Call-Operation für die Vereinfachung der Treiberstruktur zur Geltung.
- Es ist darauf zu achten, daß nicht bestimmte Funktionen eine "verdeckte" Interprozeß-kommunikation mit Timeout durchführen (z.B. mit dem Supervisor). Solche Operationen sind nur dann erlaubt, wenn der Treiber nicht aktiviert ist (z.B. zur Initialisierung).
- Es darf keine Blockierung des Threads für die Bereitstellung von Seiten des Hauptspeichers durch die virtuelle Speicherverwaltung erfolgen. Alle Seiten müssen deshalb zu Beginn beschrieben werden (siehe Abschnitt 2.1).

### 3.2.3 Schlußfolgerung

Die sequentielle Ausführung der dargestellten Operationen und damit die vereinfachte Nachbildung der Abläufe in der Linux-Netzwerkgeräte-Schicht und im Netzwerktreiber durch einen einzelnen L3-Thread erscheint möglich. Dabei wird der Aufwand für die Synchronisation mehrerer Threads hinfällig, was auch für die Verwaltung von Datenstrukturen (z.B. Warteschlangen) von Bedeutung ist, da keine Wettkampfbedingungen (*race conditions*) auftreten können. Insgesamt verspricht dies eine wesentlich einfachere und weniger fehleranfällige Lösung und damit Zeitersparnis bei Implementierung und Test. Dem steht eine zu erwartende höhere Interrupt-Latenz (speziell im Empfangsfall) entgegen, die jedoch nicht zum Verlust von Datagrammen führen dürfte. Die Richtigkeit dieser Aussage ist später zu prüfen. Die Implementierung des L3-Netzwerktreibers soll deshalb mit nur einem Thread erfolgen, dessen Grobstruktur aus Abbildung 19 hervorgeht.

## 3.3 Emulation der Speicheradressierung im Linux-Kern

In [Goel 96] wird auf die besondere Art der Speicheradressierung innerhalb des Linux-Kerns hingewiesen (siehe Abschnitt 2.3), durch die virtuelle Adressen auf äquivalente physische Adressen abgebildet werden. Für einen Linux-Gerätetreiber bedeutet dies, daß er für Operationen, die physische Adressen benötigen (z.B. DMA), keine explizite Adreßübersetzung vornehmen muß. Für die Übernahme eines Linux-Treibers in eine L3-Task muß deshalb eine gleichartige Adreßabbildung realisiert werden. Innerhalb einer residenten L3-Task ist jedoch keine Einflußnahme auf die Beziehung virtueller und physischer Adressen möglich. Beim Beschreiben einer bestimmten virtuellen Adresse wird an dieser Stelle eine beliebige freie Seite des physischen Speichers bereitgestellt, wobei eine Adreßübereinstimmung absolut zufällig wäre.

Der direkte Speicherzugriff (DMA) ermöglicht die Übertragung des Inhaltes von Speicherbereichen an Gerätesteuerungen (und umgekehrt) ohne Belastung des Prozessors, was beispielsweise zur Ein- und Ausgabe von Datenblöcken einer Festplatte oder von Datagrammen über einen Netzwerkadapter verwendet wird. Der beim direkten Speicherzugriff adressierte Speicher dient aus Sicht eines Gerätetreibers immer zum Puffern von Daten. Dadurch reduziert sich das Problem der Äquivalenzabbildung von Adressen auf Speicherbereiche, die als Datenpuffer verwendet werden.

Diese Bereiche sind für Netzwerktreiber genau lokalisierbar. Wegen einer weiteren Beschränkung des für DMA-Operationen verwendbaren Speichers, der unterhalb der 16 MB Grenze liegen muß, erfolgt dessen Bereitstellung über die spezielle Initialisierungsfunktion *nic_init(mem_start, mem_end)* des Treibers (siehe Abschnitt 2.2.2). Die Größe des benötigten fortlaufenden Pufferspeichers hängt vom betreffenden Netztreiber ab. Nach der Analyse einiger Treiber kann relativ sicher eine Maximalgröße von 128 KByte angenommen werden (Linux-Version 1.2.13). Normalerweise werden weniger als 64 KByte benötigt. Es wird nur ein einziger derartiger Speicherbereich je Netztreiber gebraucht.

Die Bereitstellung eines Speicherbereiches mit äquivalenten virtuellen und physischen Adressen an beliebiger Stelle im Adreßraum des Treiberprozesses ist mit den vorhandenen Mechanismen des L3-Kerns zwar nicht möglich, kann jedoch durch eine relativ kleine Kernänderung realisiert werden. Der neue Systemruf *MapIdemPotent (MIDP)* durchsucht die Liste der freien Seiten des Hauptspeicher nach fortlaufendem Speicher der gewünschten Größe ([Goel 96] beschreibt eine ähnliche Lösung für Mach) und blendet diesen so in den virtuellen Adreßraums der Task ein, daß sich die benötigte Adreßabbildung ergibt. Durch diese Änderung wurde einer Zielvorgabe für den Entwurf (L3-Kern bleibt unverändert) widersprochen. Es handelt sich jedoch nur um eine kleine abgeschlossene Funktion, die zudem nur dann benötigt wird, wenn der Netztreiber DMA-Operationen durchführt.

## 3.4 Abbildung der Treiberschnittstelle

Der L3-Netzwerktreiber muß die Funktionen der L3-Treiberschnittstelle (GDP) unterstützen. Die Ansteuerung der Netzwerk-Hardware erfolgt durch den eingebundenen Linux-Treiber, der die Funktionen der Linux-Treiberschnittstelle bereitstellt. Es muß eine Abbildung der für den L3-Treiber geforderten Schnittstellenfunktionalität auf die durch den Linux-Treiber gebotenen Möglichkeiten erfolgen. In diesem Abschnitt werden die Erkenntnisse aus Kapitel 2 genutzt.

### 3.4.1 Treiberschnittstellen in L3 und Linux

Die L3-Treiberschnittstelle ist eine IPC-Schnittstelle. Syntax und Semantik der zwischen Treiber und Nutzerprozeß ausgetauschten Nachrichten wird durch das GDP definiert. Der Treiber empfängt GDP-Aufträge, bearbeitet diese und sendet eine Antwort.

Ein Linux-Treiber bietet dagegen eine C-Funktionsschnittstelle an. Beim Aufruf einer Schnittstellenfunktion werden Parameter übergeben. Der Treiber liefert bei der Rückkehr aus der Funktion Ergebniswerte. Zur Linux-Treiberschnittstelle sind auch die Funktionen *netif_rx()* und *mark_bh()* (verzögerter Aufruf von *net_bh()*) der Netzwerkgeräte-Schicht zu zählen, die der Treiber von sich aus aufruft (*upcall*) und die *device*-Struktur, auf die sowohl der Treiber als auch die Netzwerkgeräte-Schicht zugreifen. In Abbildung 20 werden die beiden Treiberschnittstellen gegenübergestellt. Es ist zu beachten, daß der Zugriff auf die Funktionen der Linux-Schnittstelle über Zeiger in der *device*-Struktur erfolgt. Hier werden jedoch auch die in Abschnitt 2.2.2 verwendeten beispielhaften Funktionsnamen benutzt, deren zugehöriger Zeiger aus Abbildung 16 hervorgeht.

| L3 (GDP) | Linux |
|---|---|
| open | nic_open |
| close | nic_close |
| datagram exec early in | nic_start_xmit |
| datagram exec early out | nic_get_stats |
| lan address exec in | set_multicast_list |
| lan address exec out | |
| lan mode exec in | netif_rx |
| lan mode exec out | mark_bh (net_bh) |
| | |
| | struct device |

Abb. 20: Treiberschnittstellen in L3 und Linux

Es wird zunächst eine allgemeine Abbildungsvorschrift der L3-IPC-Schnittstelle auf die Linux-Funktionsschnittstelle benötigt. Es handelt sich hier offensichtlich um eine Art des entfernten Prozeduraufrufs (*Remote Procedure Call*), für den folgender grundsätzlicher Ablauf (hier angewendet für die Treiberschnittstelle) typisch ist:

1. Der Treiber empfängt einen GDP-Auftrag als IPC-Nachricht.
2. Die Parameter des GDP-Auftrages werden aus der Nachricht entnommen und in die Linux-Form konvertiert.
3. Die zugehörige Funktion der Linux-Treiberschnittstelle wird aufgerufen.
4. Die zurückgelieferten Werte werden in die GDP-Form übersetzt, in eine Nachricht eingetragen und an den Auftraggeber gesendet.

Diese Vorgehensweise ist jedoch nur dann möglich, wenn für einen GDP-Auftrag eine Linux-Funktion mit gleicher Semantik existiert. Anderenfalls muß die Funktionalität durch den Emulationskode bereitgestellt werden, wobei er auf Funktionen des Linux-Treibers zurückgreifen kann. Im folgenden wird auf die einzelnen GDP-Aufträge eingegangen.

### 3.4.2 Open und Close

Aus GDP-Sicht wird durch diese Aufträge eine logische Verbindung zwischen Netztreiber und einem Nutzerprozeß geöffnet bzw. geschlossen. Diese Funktionalität muß vom Emulationskode bereitgestellt werden. Es ist jedoch sinnvoll, die Netzwerk-Hardware nur dann zu aktivieren, wenn eine Verbindung zum Nutzerprozeß besteht, was durch entsprechende Verwendung der Linux-Funktionen *nic_open()* bzw. *nic_close()* möglich ist. Es besteht aber das Problem, daß für einige Linux-Treiber (z.B. den NE2000-Treiber) keine *nic_close()*-Funktion implementiert wurde. Der zugehörige Zeiger *dev->stop* hat dann den Wert NULL. Da sich in diesem Fall der weitere Empfang von Datagrammen nicht unterbinden läßt, müssen diese verworfen werden.

### 3.4.3 Senden und Empfangen von Datagrammen

Betrachtet man die Funktionalität der beiden Treiberschnittstellen genauer, so läßt sich feststellen, daß die Schnittstelle eines Linux-Treibers auf einer niedrigeren Ebene als die eines L3-Treibers liegt. Das gilt besonders für die Funktionen zum Senden und Empfangen von Datagrammen. Ein L3-Netztreiber nimmt ein zu sendendes Datagramm über den GDP-Auftrag *datagram exec early out* entgegen und übergibt es an den Netzadapter. Falls die Übergabe zur Zeit nicht möglich ist, weil der Netzadapter nicht mehr genug freien Pufferspeicher hat, so wird das Datagramm vom L3-Treiber gepuffert. Ein Linux-Treiber nimmt dagegen selbst keine Pufferung vor. Kann die Funktion *nic_start_xmit()* ein Datagramm nicht sofort übergeben, so

verwirft sie es und teilt dies dem Aufrufer mit, der es später erneut zur Aussendung übergeben kann. Dieser Ablauf wird in Abschnitt 2.2.1 dargestellt.

Die Pufferung von Datagrammen wird in Linux durch die Netzwerkgeräte-Schicht durchgeführt. Übernimmt man diese Funktionalität in den L3-Netztreiber, so ist eine Abbildung der GDP-Aufträge auf die Linux-Funktionen möglich, die bereits in Abbildung 14 gezeigt wurden (*dev_queue_xmit()*, *netif_rx()*, *net_bh()*). Dazu muß auch die Linux-Pufferverwaltung (*sk_buff*) übernommen werden, was aber ohnehin notwendig ist, weil auch die Linux-Treiber selbst mit der *sk_buff*-Struktur operieren.

Damit wird bei Erhalt eines *datagram exec early out*-Auftrags das zu sendende Datagramm in die *sk_buff*-Struktur eingetragen und an die Funktion *dev_queue_xmit()* übergeben. Der weitere Ablauf unterscheidet sich nicht von den Vorgängen in der Linux-Netzgeräte-Schicht, die bereits dargestellt wurden.

Empfangene Datagramme werden durch die Netzgeräte-Schicht in der Funktion *net_bh()* selbstständig an die verschiedenen Protokollinstanzen weitergeleitet, was durch Aufruf einer bestimmten Funktion geschieht (*upcall*). Bei dem L3-Netztreiber ist das nicht möglich, da durch die Client/Server-Struktur der GDP-Kommunikation alle Aufträge vom Nutzerprozeß ausgehen. Der entsprechende Kode wird deshalb aus *net_bh()* entfernt. Empfängt der Linux-Treiber jetzt ein Datagramm, so wird dies wie vorher an *netif_rx()* übergeben und an die Warteschlange *backlog* angehängt. Erhält der L3-Treiber einen *datagram exec early in*-Auftrag, so greift der Emulationskode selbst auf die Warteschlange zu. Da die Funktion *net_bh()* keinen Beitrag mehr zum Empfang von Datagrammen leistet, kann auch ihr (verzögerter) Aufruf durch *mark_bh(NET_BH)* aus *netif_rx()* gestrichen werden.

Hier soll noch auf eine Besonderheit des *Ethernet Dispatchers* (siehe Abschnitt 2.1.2) hingewiesen werden. Dieser verwendet zur Übergabe von zu sendenden Datagrammen nicht den GDP-Auftrag *datagram exec early out*, wie in [Heinrichs 88] gefordert, sondern unverständlicher Weise die synchrone Operation *datagram exec out*, die von [Schönbeck 94] jedoch genauso wie *datagram exec early out* behandelt wird. Aus Kompatibilitätsgründen muß sich der zu entwickelnde Treiber äquivalent verhalten, auch wenn dies nicht der allgemeinen GDP-Spezifikation entspricht. Da für Netzwerktreiber in [Heinrichs 88] aber gar keine synchrone Sendeoperation definiert ist, hat diese Besonderheit keine negativen Auswirkungen auf korrekt arbeitende Nutzerprozesse.

### 3.4.4 Abfrage und Setzen der Ethernet-Adresse

Über den GDP-Auftrag *lan address exec in* kann ein Nutzerprozeß die Ethernet-Adresse des Netzadapters abfragen. Der Linux-Treiber stellt keine äquivalente Funktion bereit. Die Adresse wird jedoch während der Initialisierung in das Feld *dev_addr[]* der *device*-Struktur eingetragen und kann vom L3-Treiber dort entnommen und an den Nutzerprozeß übermittelt werden.

Für das Setzen der sogenannten MAC-Adresse (*medium access control*), das ist beim Ethernet die Ethernet-Adresse, wurde im Linux die Funktion *set_mac_address()* definiert, die ebenfalls über einen Zeiger in der *device*-Struktur erreichbar ist (fehlt in Abbildung 12). Diese Funktion ist aber für Ethernet-Treiber nicht implementiert. Der GDP-Auftrag *lan address exec out* kann durch den L3-Treiber deshalb nicht auf einfache Weise bereitgestellt werden. Die Manipulation der Ethernet-Adresse im Feld *dev_addr[]* innerhalb der *device*-Struktur hat keinen Sinn, da sie an die Hardware ausgegeben werden muß.

Es wäre prinzipiell möglich, den Ethernet-Controller über die Funktion *set_multicast_list()* (siehe nächster Abschnitt) in den sogenannten *Promiscuous Mode* zu schalten, in dem er ausnahmslos alle Datagramme empfängt und die Adresse durch den Emulationskode selbst vergleichen zu lassen, was jedoch eine hohe Belastung des Prozessors zur Folge hätte. Da das Setzen der Ethernet-Adresse nach [Schönbeck 96] bisher von keiner Netzanwendung benutzt wird, muß der GDP-Auftrag *lan address exec out* zunächst nicht implementiert werden.

### 3.4.5 Einstellung der Betriebsart

Ethernet-Controller verfügen über einen Datagramm-Filter, der gewährleistet, daß nur die Datagramme empfangen werden, die direkt an die jeweilige Station (Unicast) oder an alle Stationen gerichtet sind (Broadcast). Der Filter vergleicht die Zieladresse des Datagramms (vgl. Abbildung 8) mit der eigenen und mit der Broadcast-Adresse auf Übereinstimmung. Zusätzlich ist auch die Adressierung einer Gruppe bestimmter Stationen möglich (Multicast), wobei jeder Multicast-Gruppe eine eigene Adresse zugeordnet ist.

Der Hauptzweck des *lan mode exec out*-Auftrages besteht in der Einstellung des Datagramm-Filters des Netzwerk-Controllers. Die in [Heinrichs 88] definierten Betriebsarten wurden bereits in Abschnitt 2.1.3 erklärt und sind in Abbildung 21 aufgeführt.

| Betriebsart | empfangene Datagramme | set_multicast_list |
|---|---|---|
| 0 | Unicast | fehlt |
| 1 | Unicast, Broadcast | 0 |
| 2 | Unicast, Broadcast, Multicast | (> 0) |
| 3 | alle (Promiscuous Mode) | -1 |

Abb. 21: Betriebsarten des Datagramm-Filters

Ein Linux-Netztreiber bietet zur Beeinflussung des Datagramm-Filters die Funktion *set_multicast_list()* an. In der Abbildung ist auch der Parameter von *set_multicast_list()* angegeben, der die jeweilige Betriebsart auswählt. Für die L3-Betriebsarten 1 und 3 ist ein Äquivalent vorhanden. Sollen diese über einen *lan mode exec out*-Auftrag eingestellt werden,

so muß also nur der Aufruf von *set_multicast_list()* mit dem entsprechenden Parameter erfolgen.

Die Betriebsart 0 wird jedoch nicht unterstützt. Der Empfang von Broadcast-Datagrammen läßt sich bei einem Linux-Treiber nicht unterbinden. Diese müssen deshalb durch den Emulationskode selbst herausgefiltert werden, was durch softwaremäßigen Vergleich mit der Broadcast-Adresse leicht möglich ist. Da Broadcast-Datagramme relativ selten auftreten, stellt dies keinen großen zusätzlichen Aufwand dar.

Schwieriger ist die Realisierung von Betriebsart 2. Während ein L3-Netztreiber in diesem Modus ausnahmslos alle Multicast-Datagramme empfängt, muß einem Linux-Treiber eine Liste von interessierenden Multicast-Gruppen-Adressen übergeben werden. Daher auch der Name der Funktion *set_multicast_list()*. Will ein Nutzerprozeß diese Betriebsart einstellen, so hat der L3-Treiber keine Information darüber, welche Multicast-Datagramme eigentlich interessieren. Welche Adressen sollen also an den Linux-Treiber übergeben werden?

Bei der Analyse verschiedener Linux-Treiber ergab sich, daß viele Treiber keinen Gebrauch von der übergebenen Adreßliste machen und ebenfalls alle Multicast-Datagramme empfangen. Das liegt zum Teil daran, daß einige Netzwerk-Controller keine Filterung von bestimmten Multicast-Adressen ermöglichen. Das gilt auch für den verwendeten 3C595-Controller. Andere Controller unterstützen dies zwar, der Treiber macht aber keinen Gebrauch davon [Gortmaker 95]. Bei diesen Treibern braucht lediglich eine beliebige Multicast-Adresse an *set_multicast_list()* übergeben werden, um das für einen L3-Treiber geforderte Verhalten zu realisieren.

Bei Linux-Treibern, die die Multicast-Adreßliste tatsächlich verwenden, besteht die Möglichkeit, alle Datagramme zu empfangen (*Promiscuous Mode*) und wiederum eine Filterung durch die Emulationssoftware vorzunehmen. Diese Lösung funktioniert mit allen Treibern. Sie ist jedoch ineffizient. Die meisten vom Netz empfangenen Datagramme werden vermutlich verworfen, müssen jedoch zunächst vom Controller in den Speicher übernommen werden, um den Adreßvergleich vornehmen zu können. Besser wäre es in diesem Fall, eine Änderung am Linux-Treiber vorzunehmen. Die Entfernung des entsprechenden Kodes aus der Funktion *set_multicast_list()* sollte nicht allzu schwierig sein.

Es besteht hier die Wahl zwischen einer allgemeingültigen ineffizienten Lösung und einer nicht mit allen Linux-Treibern funktionsfähigen effizienten Lösung. Da es keinen großen Aufwand erfordert, sollen beide implementiert werden, so daß sie je nach Treiber verwendet werden können.

Der GDP-Auftrag *lan mode exec in* liefert lediglich die aktuell eingestellte Betriebsart zurück und benötigt keine Unterstützung durch den Linux-Treiber.

## 3.5 Emulation der Dienste des Linux-Kerns

Linux-Gerätetreiber benutzen Dienste des Linux-Kerns, die im Normalfall als C-Funktionen aufgerufen werden. Wenn ein Linux-Treiber in einer L3-Task ausgeführt werden soll, müssen ihm diese Funktionen dort ebenfalls zur Verfügung gestellt werden. Dieser Abschnitt behandelt die Emulation folgender Kerndienste:

- dynamische Speicherverwaltung
- PCI-Unterstützung

### 3.5.1 Dynamische Speicherverwaltung

Der Linux-Kern bietet folgende Funktionen für die dynamische Speicherverwaltung an:

- void *kmalloc(size_t size, int priority)
- void kfree(void *ptr)
- void kfree_s(void *ptr, int size)

*kmalloc()* liefert einen Zeiger auf einen freien Speicherbereich im Kern-Segment mit der Größe *size* (in Byte), der nicht der virtuellen Speicherverwaltung unterliegt, also nie ausgelagert wird. Der Speicherbereich kann durch späteren Aufruf von *kfree()* oder *kfree_s()* wieder freigegeben werden, wobei der Zeiger auf den Speicherblock zu übergeben ist. *kfree_s()* existiert nur aus historischen Gründen. In früheren Linux-Versionen war es durch die Übergabe der allozierten Größe *size* schneller als *kfree()*, heute gibt es durch eine neue Implementation keine Unterschiede mehr [Beck 94].

Der Parameter *priority* von *kmalloc()* bestimmt, was geschehen soll, wenn kein freier Speicher verfügbar ist. Wird als Priorität die Konstante *GFP_KERNEL* übergeben (Normalfall), so lagert der Kern den Inhalt einer (oder mehrerer) Speicherseite(n) auf die Festplatte aus, um den geforderten Speicher bereitstellen zu können. Während des Plattenzugriffs wird der aktuelle Prozeß blockiert. *kmalloc()* kann auch aus einer Interrupt-Behandlungsroutine ausgerufen werden. In diesem Fall ist *GFP_ATOMIC* als Prioritätswert zu übergeben. Wenn kein Speicher verfügbar ist, liefert *kmalloc()* in diesem Fall sofort 0 zurück, da eine Interrupt-Routine kein Prozeß ist und nicht blockiert werden kann [Johnson 95]. Dieser Wert wird auch bei zeitkritischen Operationen und innerhalb der virtuellen Speicherverwaltung verwendet. Zusätzlich existiert die Konstante *GFP_DMA*, die zu den bereits erläuterten Prioritätswerten hinzuaddiert werden kann. *kmalloc()* liefert dann DMA-fähigen Speicher, der unterhalb der 16 MB Grenze liegt.

Das von *kmalloc()* verwendete Verwaltungsverfahren ähnelt dem Buddy-Verfahren [Tanenbaum 90] und wird in [Beck 94] erläutert. Es stützt sich auf die Kern-Funktion *__get_free_pages()*, die eine oder mehrere fortlaufende Seiten mit einer Größe von 4 KByte

bereitstellt und unterteilt diese Seiten in Speicherblöcke bestimmter Größe (z.B. 32, 64, 128, 252, 508, 1020, 2040, 4080 Byte). Bei einer Speicheranforderung wird immer ein Block der nächst passenden Größe zugeordnet, wodurch Speicher verschenkt wird, was zur Verhinderung der Fragmentierung des Speichers jedoch notwendig ist.

Zur Bereitstellung einer dynamischen Speicherverwaltung innerhalb einer L3-Task könnte im Prinzip ein beliebiges Verfahren angewendet werden, das die Fragmentierung verhindert und effizient ist. Die schnellste Lösung ist jedoch die Portierung des Linux-Kodes. Der einzige denkbare Nachteil dieser Lösung ist die Verschwendung von Speicher. Es ist jedoch kein Verfahren bekannt, das die gleiche Zeiteffizienz mit einer ökonomischeren Speichernutzung verbindet.

Für die Übernahme des Linux-Kodes muß die Funktionalität von *__get_free_pages()* und *free_pages()* bereitgestellt werden. Die einfachste Möglichkeit ist die statische Belegung einer bestimmten Anzahl von Seiten zum Initialisierungszeitpunkt (durch einfaches Beschreiben der Seiten) und deren Verwaltung in einer verketteten Liste. Es stellt sich die Frage, wieviel Seiten von einem Linux-Netztreiber benötigt werden. Es ist schwierig, darüber eine allgemeine Aussage zu machen. Die Belegung von Seiten, die eigentlich nicht benutzt werden, verschwendet Speicher, da es auch nicht möglich ist, eine Seite, die eine residente L3-Task einmal beschrieben hat, wieder freizugeben. Es wäre deshalb eventuell sinnvoll, zunächst nur wenige Seiten auszufassen, und erst wenn die Anzahl der freien Seiten einen bestimmten Schwellwert unterschreitet, die Freiseitenliste aufzustocken. Zu diesem Zweck müßte ein zweiter Thread verwendet werden, da beim Beschreiben einer neuen Seite eventuell ein Plattenzugriff und damit ein Blockieren des Threads notwendig sein könnte, was beim Entwurf der Thread-Struktur in Abschnitt 3.2.2 als unzulässig herausgestellt wurde.

Aus Aufwandsgründen soll zunächst auf die dynamische Anpassung der belegten Seitenanzahl verzichtet werden. Die endgültige Entscheidung wird bis zum Vorliegen eines lauffähigen Treibers verschoben, weil dann genauere Aussagen über dessen Verhalten möglich sind. Eine ggf. später notwendige Änderung ist lokal auf den Kode der Speicherverwaltung begrenzt und damit kein Problem.

Bei der vorgesehenen statischen Belegung von Seiten innerhalb der L3-Task ist der Prioritätsparameter ohne Bedeutung und kann ignoriert werden, da sich *kmalloc()* dann ohnehin immer so verhält, als wäre *GFP_ATOMIC* angegeben. Ein Blockieren des Threads zur Seitenauslagerung kann nicht auftreten. *GFP_DMA* wird von den Netztreibern der Linux-Version 1.2.13 nicht verwendet und wird zunächst nicht unterstützt. Die maximale Größe eines Speicherblocks ist auf eine Seite (genauer 4072 Byte) beschränkt, was für Ethernet-Treiber ausreicht, da diese Speicher höchstens in der Größe eines Datagramms (1514 Byte) zuzüglich der *sk_buff*-Struktur (16 Byte) belegen.

### 3.5.2 PCI-Unterstützung

PCI (*Peripherial Component Interconnect*) ist ein verbreiteter Standard, der neben der Definition eines Bussystems auch die Verwaltung von Hardware-Konfigurationsinformationen definiert. Ein PCI-Gerät kann maximal 256 verschiedene Konfigurationsregister besitzen, wobei die Bedeutung der ersten 64 Register für alle Geräte gleich ist [Stiller 96]. Treiber von PCI-Geräten müssen auf diese Register lesend und teilweise auch schreibend zugreifen können. Zu diesem Zweck existiert das PCI-BIOS, das auch eine Unterstützung für 32-Bit-Betriebssysteme bietet. Direkte I/O-Zugriffe auf die Konfigurationsregister sind nicht praktikabel, da es verschiedene Realisierungsmöglichkeiten gibt, die durch die einheitliche BIOS-Schnittstelle verdeckt werden.

Linux-Gerätetreiber können sich für den Zugriff auf das PCI-BIOS folgender Funktionen bedienen:

- *pcibios_present()* (prüft Existenz des PCI-BIOS)
- *pcibios_find_device()*, *pcibios_find_class()* (sucht ein bestimmtes Gerät bzw. ein Gerät einer Geräteklasse)
- *pcibios_read_config_byte()*, *pcibios_read_config_word()*, *pcibios_read_config_dword()* (Lesen eines Konfigurationsregisters mit verschiedener Wortbreite)
- *pcibios_write_config_byte()*, *pcibios_write_config_word()*, *pcibios_write_config_dword()* (Schreiben auf ein Konfigurationsregister mit verschiedener Wortbreite)

Das L3-System bietet bisher keine PCI-Unterstützung. Da neuere Gerätesteuerungen meist PCI-Geräte sind, was auch für den 3C595-Controller gilt, wird diese jedoch benötigt. Der Teil des Linux-Quellkodes, der die PCI-Funktionen implementiert (Datei bios32.c), läßt sich fast unverändert übernehmen. Dazu ist lediglich die Frage zu klären, wie das PCI-BIOS in den Adreßraum einer residenten L3-Task eingeblendet werden kann. Dies ist über einen Systemruf (*NOLT*) möglich. Die Funktion *bios32_init()* sucht den Adreßbereich von 0xE0000 bis 0xFFFFF nach dem *BIOS32 Service Directory* ab (beginnt mit der Zeichenkette "_32_"), über das in einem mehrstufigen Prozeß der Eintrittspunkt des PCI-BIOS gefunden wird. Dieser Bereich muß deshalb im Task-Adreßraum verfügbar sein.

Da in einem System mehrere PCI-Geräte aktiv sein können, ist es keine gute Lösung, wenn jeder Gerätetreiber selbstständig auf das PCI-BIOS zugreift. Das ist auch deshalb problematisch, weil der Maschinenkode im BIOS bestimmte I/O-Ports benutzt, die dem Gerätetreiber zugewiesen werden müßten (siehe Abschnitt 2.1.2). Der *hardware configurator* verhindert aber, daß mehrere Treiber Verfügung über die selben I/O-Adressen haben. Bisher kann ein L3-Treiber jedoch auch auf die I/O-Adressen zugreifen, die ihm gar nicht zugewiesen wurden, weil dies vom Kern nicht überwacht wird, was aber nicht so bleiben muß. Das gilt

genaugenommen auch für den eingeblendeten BIOS-Speicherbereich selbst, der auch nur einem einzigen Treiber zur Verfügung stehen dürfte.

Dieses Problem läßt sich durch die Bereitstellung von PCI-Konfigurationsdiensten über einen speziellen L3-PCI-Treiber lösen, auf den auch das ELAN-Install-Programm eines PCI-Gerätetreibers über IPC-Nachrichten zugreifen kann. Es wäre sonst beispielsweise nicht möglich, automatisch den IRQ eines PCI-Gerätes zu ermitteln und dem Gerätetreiber zuzuweisen.

Für die Kommunikation mit L3-Treibern wird in [Liedtke 91b] das GDP empfohlen. Die dort definierten Geräteklassen und Objekttypen eignen sich jedoch nicht für den PCI-Treiber. Es wird deshalb eine Erweiterung für die Objekttyp-Tabelle vorgeschlagen (Abb. 22).

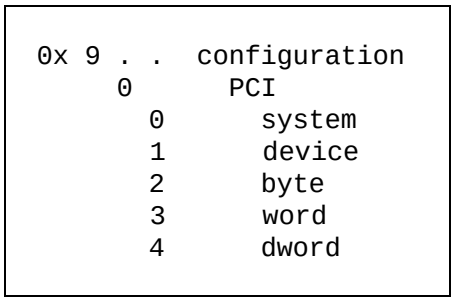


Abb. 22: GDP-Erweiterungen für den PCI-Treiber

Unter der neuen Geräteklasse *configuration* (vorgeschlagener Identifikator: 9) können beliebige Konfigurationsdienste eingeordnet werden. In der Objektgruppe *PCI* werden die Objekttypen *system* (z.B. Version des PCI-BIOS), *device* (PCI-Geräteadresse bestehend aus Bus/Device/Function) (vgl. [Stiller 96]), *byte*, *word* und *dword* (Inhalt von Konfigurationsregistern in verschiedener Wortbreite) definiert. Für die Objekttypen *system* und *device* ist nur die GDP-Funktion *exec in* (z.B. *pci device exec in* entspricht *pcibios_find_device()*) zulässig, für die Typen *byte*, *word* und *dword* ist auch *exec out* (Lesen bzw. Ändern eines Registers) erlaubt.

Ruft ein Linux-Netztreiber in der L3-Emulation eine der am Anfang des Abschnittes aufgeführten Funktionen des Linux-Kerns auf, so führt dies nun zu einer IPC-Kommunikation mit dem PCI-Treiber. Es ist darauf zu achten, daß diese Funktionen nur in der Initialisierungsphase des Treibers aufgerufen werden (was normalerweise der Fall ist), da die Interprozeßkommunikation in diesem Fall mit Timeout-Werten größer 0 erfolgen muß.

# 4 Implementierung

Im folgenden werden Aspekte der Implementierung eines L3-Netzwerktreibers auf Basis der Erkenntnisse des vorangegangenen Kapitels dargestellt. Neben der Behandlung von Teilen des Quellkodes spielt auch die Vorgehensweise bei der Programmentwicklung eine Rolle. Diese wurde durch den Umstand beeinflußt, daß fremder Programmkode in Form eines unbekannten Linux-Treibers eingebunden wird, über dessen Qualität und konkretes Verhalten von vornherein keine Aussagen möglich sind. Quelltextausschnitte, die hier gezeigt werden, sind im Interesse einer übersichtlichen Darstellung auf das Wesentliche reduziert.

## 4.1 Konfiguration und Initialisierung

Die Funktion des Treibers wird durch Parameter beeinflußt, die konfigurierbar sein müssen. Dazu gehören insbesondere Informationen über die Hardware (z.B. I/O-Basisadresse). Wegen der Einbindung von verschiedenen Linux-Treibern ist auch eine treiberabhängige Konfiguration notwendig. In diesem Zusammenhang sind auch die Vorgänge bei der Initialisierung von Bedeutung.

### 4.1.1 Treiberabhängige Konfiguration

```
#define CONFIG_NAME             "L3.BIOS.3C59X"
#define CONFIG_VERSION          30000
#define CONFIG_DATE             "96-04-10"

#define CONFIG_PCI
#undef  CONFIG_SOFT_MULTICAST

#define CONFIG_INIT_FUNCTION    tc59x_init
#define CONFIG_INIT_DMAMEM      92

#define CONFIG_RESERVED_PAGES   20

#define CONFIG_IRQ              5
#define CONFIG_BASE_ADDR        0xe800
#define CONFIG_MEM_START        0
#define CONFIG_MEM_END          0
```

Abb. 23: Treiberabhängige Konfigurationsdaten (driverconf.h)

Linux-Treiber unterscheiden sich u.a. in dem Namen ihrer Initialisierungsfunktion. In Abschnitt 3.4.5 wurden verschiedene Varianten der Multicast-Unterstützung diskutiert, die beide implementiert werden sollen, um sie treiberabhängig einsetzen zu können. Für die Genierung eines bestimmten Treibers sind deshalb einige Änderungen innerhalb des Quelltextes

erforderlich, die zentral in der Datei *driverconf.h* erfolgen. Abbildung 23 zeigt deren Inhalt beispielhaft für den Linux-Treiber des 3C595-Controllers [Becker 95]. Neben Versionsinformationen, der wahlweisen Einbindung der PCI-Unterstützung, Auswahl der Multicast-Unterstützung sowie Name und Speicherbedarf der Initialisierungsfunktion sind hier die Standardwerte der Parameter des Konfigurationsbereiches enthalten, auf den nachfolgend eingegangen wird.

### 4.1.2 Hardwareabhängige Konfiguration

Die Konfiguration von Gerätetreibern erfolgt im L3-System durch den *hardware configurator*, der sich eines treiberspezifischen ELAN-Install-Programms bedient, das die Daten des Treibers manipuliert. Für den zu entwickelnden L3-Netzwerktreiber wurde zu diesem Zweck eine Datenstruktur definiert (in *config.h*), die mit einer Signatur beginnt, über die sie innerhalb des Treiberkodes gefunden werden kann (Abb. 24). Sie enthält außerdem die Treiberversion zur Kompatibilitätsprüfung durch das Install-Programm, die Adressen des Eintrittspunktes und des Stapelspeichers, die Anzahl von zu reservierenden Seiten für die dynamische Speicherverwaltung (siehe Abschnitt 3.5.1) und die Hardwareparameter des Netzwerk-Controllers, die an den Linux-Treiber übergeben werden. Linux ermöglicht die Konfigurierung von Netztreibern über einen sogenannten Boot-Parameter (*ether*) [Gortmaker 95]. Die verschiedenen Werte werden in die *device*-Struktur (Abb. 12) eingetragen, die an die Initialisierungsfunktion des Treibers übergeben wird.

```
struct config
{
    char            signature[8];   /* "_CONFIG_" */

    char            name[24];
    unsigned int    version;

    unsigned long   entry;
    unsigned long   stack;

    unsigned int    reserved_pages;

    unsigned char   irq;
    unsigned long   base_addr;
    unsigned long   mem_start;
    unsigned long   mem_end;
}
config_area;
```

Abb. 24: Struktur des Konfigurationsbereichs (config_area)

Der Linux-Kern unterstützt die Verwaltung von I/O-Adreßbereichen über diese Funktionen:

- void request_region(unsigned int from, unsigned int num, const char *name)
- void release_region(unsigned int from, unsigned int num)
- int check_region(unsigned int from, unsigned int num)

Ein Linux-Treiber kann einen bestimmten Adreßbereich über die Funktion *request_region()* anfordern. Vorher prüft er jedoch mit *check_region()*, ob dieser Bereich bereits vergeben ist. Viele Linux-Treiber versuchen, die I/O-Basisadresse ihres Controllers selbst herauszufinden, indem sie bestimmte Adressen nach ihm absuchen. Ein L3-Treiber sollte jedoch nicht auf Adressen zugreifen, die ihm nicht durch den *hardware configurator* zugewiesen wurden. *check_region()* wird deshalb in L3 so emuliert, daß nur die konfigurierte Basisadresse (*config_area.base_addr*) an den Linux-Treiber als noch nicht vergeben gemeldet wird. Dadurch wird verhindert, daß er versucht, auf unerlaubte Adressen zuzugreifen.

Ähnlich wird verfahren, wenn ein Linux-Netztreiber den IRQ ermitteln will, über den Interrupts an ihn zugestellt werden. Der Linux-Kern bietet dafür folgende Funktionen:

- void autoirq_setup(int waittime)
- int autoirq_report(int waittime)

Der Treiber ruft zunächst die *autoirq_setup()*-Funktion auf, generiert anschließend einen Interrupt und erhält von *autoirq_report()* mitgeteilt, über welchen IRQ ein Interrupt vom Linux-Kern festgestellt wurde, der dann zu dem von ihm angesteuerten Netzwerk-Controller gehört. Auch diese Funktionalität kann in einer residenten L3-Task nicht originalgetreu nachgebildet werden, da der IRQ ihr schon bei der Erzeugung zugewiesen werden muß. Der Emulationskode ignoriert deshalb Aufrufe von *autoirq_setup()* und liefert beim Aufruf von *autoirq_report()* einfach den konfigurierten IRQ (*config_area.irq*) zurück.

### 4.1.3 Initialisierung des Linux-Netzwerktreibers

Der Linux-Kern unterstützt die gleichzeitige Verwendung mehrerer Netzwerk-Controller, die von einem einzigen oder von verschiedenen im Linux-Kern enthaltenen Netztreibern angesteuert werden können. Für den L3-Netztreiber stellt sich die Frage, ob dieser ebenfalls die Fähigkeit haben soll, mehrere Netzadapter parallel zu betreiben. Die GDP-Spezifikation für Ethernet-Treiber [Heinrichs 88] und der Treiber [Schönbeck 94] sehen keine gleichzeitige Ansteuerung mehrerer Netzadapter vor. Der hier beschriebene Treiber tut dies deshalb auch nicht. [Schönbeck 94] kann aber für verschiedene Adaptertypen konfiguriert werden. Es wäre prinzipiell möglich, mehrere Linux-Treiber in einen L3-Treiber einzubinden, was jedoch Speicher verschwendet (etwa 10 KByte je Treiber), weil immer nur ein Treiber aktiv ist. Deshalb wird nur ein Treiber integriert.

Wegen des Verzichtes auf die Einbindung mehrerer Linux-Treiber und der Unterstützung von nur einem aktiven Netzwerkgerät läßt sich vermuten, daß die Abläufe bei der Initialisierung im Emulationskode stark vereinfacht werden können, was sich jedoch nicht bestätigt hat. Wie sich herausstellte, gibt es keine Möglichkeit, um generell zu verhindern, daß ein Netztreiber mehrere im System vorhandene Netzadapter erkennt und initialisiert, was u.a. daran liegt, daß Treiber existieren, die die Funktion *check_region()* nicht verwenden. Der Emulationskode muß deshalb die Initialisierung mehrerer Netzwerkgeräte zulassen. Da jedoch nur ein Netzwerk-Controller verwendet werden kann, wird dieser ggf. durch Vergleich der in der *device*-Struktur eingetragenen Basisadresse mit der konfigurierten Adresse (*config_area.base_addr*) gesucht. Weil die Zuordnung von I/O-Adressen zu Geräten eindeutig ist, kann über die konfigurierte Adresse zweifelsfrei ein bestimmter Netzadapter selektiert werden.

Linux-Treiber, die DMA-Operationen durchführen, werden über eine spezielle Funktion initialisiert (*nic_init()*, siehe Abschnitt 2.2.2), der die Adresse eines DMA-fähigen Speicherbereiches übergeben wird. Die Größe des durch den Treiber belegten Bereiches ist erst nach der Rückkehr aus der Funktion bekannt. Der Speicher muß jedoch vor deren Aufruf mit dem Systemruf *MapIdemPotent* (siehe Abschnitt 3.3) bereitgestellt werden, wobei die gewünschte Größe anzugeben ist. Dieses Problem läßt sich (ohne Analyse des Linux-Treibers) nur so lösen, daß zunächst die maximal zu erwartende Größe (z.B. 128 KByte) angenommen und bei einem Test des Treibers die tatsächlich benutzte Größe festgestellt und nachfolgend verwendet wird. Der entsprechende Wert wird als Konstante *CONFIG_INIT_DMAMEM* in die Datei *driverconf.h* eingetragen (Abb. 23).

## 4.2 Behandlung von GDP-Aufträgen

Dieser Abschnitt stellt den Ablauf bei der Behandlung von GDP-Aufträgen am Beispiel des Sendens eines Datagramms über die *datagram exec early out*-Nachricht dar. Abbildung 25 zeigt die Hauptschleife des Netzwerktreibers in der Funktion *main()*.

Zunächst wird die Funktion *gdp_wait()* aufgerufen, die für den Empfang von GDP-Aufträgen und anderen IPC-Nachrichten verantwortlich ist (Abb. 26). Für die IPC-Empfangsoperation *Wait()* muß ein Puffer bereitgestellt werden, in dem das zu sendende Datagramm übermittelt wird. Um unnötige Kopiervorgänge zu vermeiden, ist es sinnvoll, hier mit *alloc_buff()* sofort eine *net_buff*-Struktur (entspricht der *sk_buff*-Struktur aus Abbildung 13) mit der maximal notwendigen Größe zuzuweisen, die später an den Linux-Netztreiber übergeben werden kann. Die Pufferverwaltung des Emulationskodes verwendet andere Namen als der Linux-Kode, bietet aber die selbe Funktionalität. Die Umbenennung ist zur Verhinderung von wechselseitigen Abhängigkeiten im Quellkode erforderlich. Die Zuweisung des Puffers erfolgt durch Manipulation der Nachrichtenstruktur *order*. Die Funktion *gdp_wait()* kehrt erst nach Empfang einer IPC-Nachricht zurück.

```
for(;;)
{
    gdp_wait(&thread, &order_msg);

    if(IsInterrupt(thread))
        do_irq(InterruptIRQ(thread));
    else
        switch(order_msg.dw1 & PROT_INDEX_MASK)
        {
        case GDP_INDEX:
            gdp_order(&thread, &order_msg);
            break;

        case DCP_INDEX:
            dcp_order(&thread, &order_msg);
        }
}
```

Abb. 25: Hauptschleife des Netztreibers (main.c)

```
void gdp_wait(ThreadT *thread, OrderMsgT *order)
{
    if(order->size == DW3DOPE && (buff = alloc_buff(ETH_FRAME_MAX+1)))
    {
        order->size = DW3STRDOPE;
        order->str.StrBufAddress = buff->data;
        order->str.StrBufSize = ETH_FRAME_MAX+1;
    }

    do
    {
        ...
        error = IPCError(Wait(thread, order, timeout));
    }
    while(error);
}
```

Abb. 26: Funktion gdp_wait() (gdp.c)

In *main()* wird anhand des von *gdp_wait()* gelieferten Thread-ID festgestellt, ob eine Interrupt-Nachricht vom Kern empfangen wurde, was ggf. zur Behandlung des Interrupts durch die Funktion *do_irq()* führt, die die vom Linux-Treiber bereitgestellte Interrupt-Routine und nachfolgend bei Bedarf die Bottom-Half-Routine aufruft (vgl. Abb. 19). Andernfalls wird der im ersten DWord der empfangenen Nachricht enthaltene Protokoll-Identifikator ausgewertet. Handelt es sich um eine GDP-Nachricht, wird diese zur Behandlung an die Funktion *gdp_order()* weitergeleitet. Für Testzwecke wurde ein spezielles Protokoll definiert, das

*Driver Control Protocol (DCP)* genannt wurde. Beim Empfang einer DCP-Nachricht erfolgt der Aufruf der Funktion *dcp_order()*.

```
void gdp_order(ThreadT *thread, OrderMsgT *order)
{
    ...
    switch(order_code)
    {
    case GDP_DATAGRAM_OUT:
        if(  (len = order->str.StrLength) < ETH_HEADER_LEN
           || len > ETH_FRAME_MAX)
        {
            send_alt(thread, order);
            return;
        }

        if(!interrogate)
        {
            if(len < COPY_BUFF_THRESHOLD && (buff = alloc_buff(len)))
                memcpy(buff->data, order->str.StrAddress, len);
            else
            {
                buff = ((struct net_buff *)order->str.StrAddress)-1;
                order->size = DW3DOPE;
            }

            buff->len = len;
            memcpy(&buff->data[ETH_ADDR_LEN],
                   net_get_address(), ETH_ADDR_LEN);

            net_tx(buff);
        }

        send_nil(thread, GDP_NIL_REPLY, GDP_OK);
        break;
    ...
    }
}
```

Abb. 27: Funktion gdp_order() (gdp.c)

Die Funktion *gdp_order()* (Abb. 27) verzweigt nach einigen allgemeinen Operationen, die hier nicht dargestellt sind, in Abhängigkeit vom GDP-Nachrichtenkode (*order_code*) zur Behandlung der verschiedenen GDP-Aufträge. Im betrachteten Fall des Sendens eines Datagramms wird geprüft, ob dessen Länge im erlaubten Bereich liegt. Ist dies nicht gegeben, so wird der Auftrag mit einer *datagram alternate out request*-Nachricht abgewiesen. Handelt es sich bei dem empfangenen GDP-Auftrag um eine *interrogate*-Nachricht, so will der Partnerprozeß lediglich prüfen, ob die Sendeoperation zulässig ist, sie jedoch nicht wirklich ausführen. Es kann dann sofort eine positive Antwort (*nil reply*) gesendet werden.

An dieser Stelle wird ein Problem behandelt, das durch die in *gdp_wait()* zur Einsparung von Kopieroperationen vorgenommene Zuweisung eines Puffers maximaler Größe entsteht. Handelt es sich nämlich nur um ein sehr kurzes Datagramm, so wird Speicher verschenkt. Falls oberhalb des Netztreibers mehrere offene Netzverbindungen bestehen und das Netzwerk stark belastet ist, so könnte die Warteschlange der auszusendenden Datagramme möglicherweise eine Länge erreichen, bei der diese Speicherverschwendung zum Problem wird. Es wurde deshalb ein Schwellwert für die Datagrammlänge vorgesehen (*COPY_BUFF_THRESHOLD*), bei dessen Unterschreiten ein Datagramm in einen Puffer adäquater Größe umkopiert wird. Für kurze Datagramme hält sich der Zeitaufwand beim Kopieren in Grenzen, es wird jedoch relativ viel Speicher gespart.

Vor der Übergabe des Datagramms an die Sendefunktion *net_tx()* (entspricht der Funktion *dev_queue_xmit()* aus Abbildung 14) wird noch die eigene Netzadresse mit *memcpy()* als Absender eingetragen. Die logische Schnittstelle zum Netzwerktreiber, die der Emulationskode für die Ausführung von GDP-Aufträgen bereitstellt, ist in der Quelldatei *net.h* definiert (Abb. 28). Durch diese vorgenommene Abstraktion ist der L3-orientierte GDP-Kode klar vom Linux-abhängigen Emulationskode getrennt.

```
int net_open(void);
void net_close(void);

void net_tx(struct net_buff *buff);

int net_rx_stat(void);
struct net_buff *net_rx(void);

unsigned char *net_get_address(void);

void net_set_filter(unsigned int mode);
```

Abb. 28: Schnittstelle zum Netzwerktreiber (net.h)

## 4.3 Programmentwicklung

Für die Entwicklung des L3-Netztreibers wurde Linux als Entwicklungsumgebung benutzt. Der Zugriff auf L3-Systemdienste erfolgt über eine Funktionsbibliothek, die für die Entwicklung der UNIX-Emulation [Wolter 95] geschaffen wurde und in [Hohmuth 96] beschrieben wird. Es werden nur einfache Funktionen benutzt, die im wesentlichen direkt auf Systemrufe abgebildet werden. Der unter Linux generierte Treiberkode wird auf das L3-System übertragen und dort mit Hilfe eines ELAN-Programms (*runx*) in einen Datenraum übertragen.

Die von den Linux-Netzwerktreibern verwendeten C-Header-Dateien wurden direkt übernommen. Ihr Inhalt wurde auf die benötigten Definitionen von Konstanten, Datenstrukturen und Funktionen reduziert. Einige Header-Dateien sind leer, müssen jedoch existieren, weil sie in den Treibern referenziert werden. Die Frage, welche Definitionen von Wichtigkeit sind, ist nicht in allen Fällen klar zu beantworten, da einem Linux-Treiber prinzipiell alle Kerndienste zur Verfügung stehen. Der Emulationskode kann jedoch nur einen kleinen Teil davon nachbilden.

Die verdeckte Verwendung von nicht unterstützten Funktionen würde Fehler verursachen, die schwer zu identifizieren wären. Es ist z.B. denkbar, daß ein Treiber durch die Manipulation eines Feldes innerhalb einer Datenstruktur oder durch die Übergabe eines bestimmten Parameters an eine Funktion eine konkrete Aktion auslösen will, die durch dem Emulationskode jedoch nicht nachgebildet wird. Solche Probleme sollten durch den Compiler oder den Emulationskode selbst festgestellt werden. Deshalb wurden Definitionen entfernt, die bisher nicht benutzt werden und deren Auswirkung nicht überblickt werden kann, so daß sich bei Bezugnahme auf diese ein Übersetzungsfehler ergibt. Funktionen, die nur einen Teil der Linux-Funktionalität nachbilden, melden zur Laufzeit, wenn nicht unterstützte Parameter übergeben wurden. Bei der Einbindung von neuen Linux-Netztreibern kann dadurch eine Erweiterung des Emulationskodes erforderlich werden.

# 5 Bewertung

Nach dem Vorliegen einer funktionsfähigen Implementation des L3-Netzwerktreibers auf Basis der Überlegungen in den Kapiteln 3 und 4 wurden einige Untersuchungen durchgeführt, deren Ergebnisse hier dargestellt werden sollen. Das wichtigste Resultat ist dabei, daß es gelungen ist, einen funktionsfähigen L3-Treiber zu erstellen, in den der unveränderte Quellkode verschiedener Linux-Netztreiber eingebunden werden kann.

Die entstandene Lösung wurde mit folgenden Netzwerk-Controllern getestet:

- 3Com Etherlink III 3C595-TX (PCI, Linux-Treiber 3c59x.c)
- LCS-8634L (NE2000 kompatibel, Linux-Treiber ne.c)

Der 3C595-Controller unterstützt den Fast-Ethernet-Standard (100BaseTX), wurde bisher jedoch nur mit 10 MBit/s betrieben. Der verwendete Rechner hat einen Pentium-Prozessor mit einer Taktfrequenz von 100 MHz und 32 MByte Hauptspeicher. Die bisherigen Tests erfolgten im praktischen Betrieb. Meßergebnisse mit definierten Bedingungen liegen zum gegenwärtigen Zeitpunkt noch nicht vor. Es sind deshalb keine zuverlässigen Aussagen über die Leistung der getesteten Treiber verfügbar. Einige Erkenntnisse wurden dennoch gewonnen.

Unter Verwendung der L3-TCP/IP-Implementation erfolgten u.a. Dateiübertragungen mit dem FTP-Protokoll. Zeitweise wurde der Netzadapter in den *Promiscuous Mode* geschaltet, in dem er alle Datagramme empfängt, und das Netz künstlich durch andere Rechner belastet. Bei beiden Netzwerk-Controllern konnten keine Datagrammverluste festgestellt werden, was die Überlegungen beim Entwurf der Thread-Struktur in Abschnitt 3.2.2 bestätigt. Es wurden dazu die von den Linux-Treibern über die Funktion *dev->get_stats()* gelieferten Werte betrachtet.

Bei der Übertragung einer Datei mit dem NE2000-kompatiblen Controller wahlweise mit dem erstellten Treiber und dem Treiber [Schönbeck 94] ergab sich, daß der emulierte Treiber eine um ca. 20 % schnellere Übertragung ermöglichte und dabei weniger Prozessorzeit verbrauchte. Diese Feststellung hat nur einen bedingten Wert, weil keine eindeutigen Randbedingungen herrschten und keine Aussagen über den als Vergleichkriterium herangezogenen Treiber gemacht werden können. Sie zeigt jedoch, daß die Annahme, eine Emulationslösung für Gerätetreiber könne keine ähnlichen Leistungswerte wie eine systemspezifische Implementierung erbringen, nicht zutrifft. Die gleiche Dateiübertragung verlief unter Linux allerdings um ein Mehrfaches schneller als unter L3, was vermutlich zum großen Teil auf die L3-TCP/IP-Implementierung zurückzuführen ist.

In Abschnitt 3.5.1 ergab sich bei der Behandlung der dynamischen Speicherverwaltung die Frage, wieviel Speicherseiten (von 4 KByte Größe) der Netztreiber dafür benötigt. Bei den durchgeführten Tests hat sich gezeigt, daß im Normalfall nicht mehr als 5 Seiten zugleich verwendet werden. Bei der statischen Reservierung von (zur Sicherheit) 10 Seiten für die dynamische Speicherverwaltung würde der Treiber insgesamt etwa 80 KByte physischen

Speicher belegen, was kein Problem darstellen dürfte. Die im genannten Abschnitt erörterte dynamische Belegung von Seiten ist deshalb unnötig.

Über die entwickelte Emulation zur Einbindung eines Linux-Netztreiber in das L3-System läßt sich hier feststellen, daß diese Lösung gut funktioniert und zumindest Leistungswerte erbringt, die dem Treiber [Schönbeck 94] entsprechen. Da das Ziel dieser Arbeit in der Bereitstellung von Gerätetreibern durch das Mittel der Emulation einer anderen Systemumgebung lag, wobei gewisse Leistungseinbußen von vornherein erwartet wurden, ist dieses Ergebnis bereits zufriedenstellend. Es sollen jedoch noch Messungen durchgeführt werden, die genauere Aussagen über die Leistungsfähigkeit der erstellten Lösung zulassen.

# 6 Zusammenfassung und Ausblick

Diese Arbeit befaßte sich mit der systematischen Übertragung von Gerätetreibern des monolithischen Betriebssystems Linux auf das µ-Kern-basierte L3-System. Dazu wurde zunächst die Funktionalität von Gerätetreibern in L3 und in Linux betrachtet. Es kam zum Ausdruck, daß das Linux-System spezifische Treiberschnittstellen für die verschiedenen Geräteklassen verwendet, weshalb keine Allgemeinbehandlung für alle Gerätetreiber erfolgen konnte, was die Beschränkung auf Netztreiber zur Folge hatte. Bei der Suche nach einer günstigen Thread-Struktur für die Abbildung der Nebenläufigkeiten im Linux-Kern ergab sich, daß auch eine sequentielle Ausführung des Linux-Kodes durch einen einzigen Thread möglich ist und außerdem am schnellsten zu realisieren ist, weshalb diese Variante implementiert wurde. Wegen des derzeitigen Standes bei der Entwicklung der UNIX-Emulation [Wolter 95] war deren besondere Berücksichtigung bei der Gestaltung der Treiberschnittstelle nicht möglich.

Wie im vorangegangenen Kapitel zum Ausdruck kam, funktioniert der entwickelte Netzwerktreiber, in den der Quellkode eines Linux-Treibers in unveränderter Form eingebunden werden kann, zufriedenstellend. Es sind jedoch weitere Messungen zur Bestimmung der genauen Leistungsfähigkeit erforderlich. Linux-Treiber eignen sich gut für die Einbindung in andere Systeme, wie auch am Beispiel von Mach [Goel 96] deutlich wurde. Das L3-System bietet die notwendige Funktionalität für die Emulation einer Linux-Kernumgebung innerhalb eines Nutzerprozesses. Lediglich zur Nachbildung der Speicheradressierung war eine Änderung am L3-Kern notwendig, die jedoch nur für Treiber benötigt wird, die DMA-Operationen ausführen. Die automatische Konfiguration von Linux-Gerätetreibern ist in L3 nicht möglich, was jedoch vor den Treibern verborgen werden kann.

Der entwickelte Quellkode umfaßt derzeit etwa 2500 Zeilen, wobei nur die Teile berücksichtigt sind, die neu entwickelt oder stark verändert wurden. Der Netztreiber ist zwar bereits verwendbar, in der nächsten Zeit sind jedoch noch folgende Arbeiten erforderlich:

- Meldungen des Linux-Treibers über das Driver Control Protocol verfügbar machen
- Inbetriebnahme des PCI-Treibers und dessen Verwendung durch den Netztreiber
- Unterstützung von DMA-Operationen

Abschließend kann festgestellt werden, daß die Emulation einer fremden Systemumgebung zur Übernahme von Quellkode ein vielversprechendes Verfahren ist, das auch für andere Zwecke in Betracht kommt. Für die Multiserver-UNIX-Emulation wäre die Übernahme des vollständigen Netzwerkkodes aus dem Linux-System denkbar. Am Lehrstuhl Betriebssysteme der Fakultät Informatik der Technischen Universität Dresden wird bereits an der Portierung des Linux-Kerns auf den L3-Nachfolger L4 gearbeitet. Außerdem soll die in dieser Arbeit noch ausgesparte Emulation für SCSI-Gerätetreiber durch andere Arbeiten realisiert werden.

# Glossar

**Autoprobing**: Verfahren zur selbstständigen Ermittlung von Hardware-Parametern durch Linux-Gerätetreiber

**BIOS**: (Basic Input Output System) im Festwertspeicher eines Personalcomputers enthaltenes Steuerprogramm für die Hardware

**Broadcast**: Rundsendung an alle angeschlossenen Stationen

**Controller**: elektronische Baugruppe für den Anschluß von Peripheriegeräten an ein Computersystem

**Datagramm (Datagram, Frame, Packet, Paket)**: Diese Begriffe bezeichnen eine über ein Computernetzwerk übertragene Dateneinheit. Datagram(m) und Frame (im englischen Sprachraum auch Packet) wird für Daten niedriger Protokollebenen verwendet, Paket im allgemeinen für Daten der Vermittlungsschicht.

**DMA**: (Direct Memory Access) Übertragung von Daten zwischen einer Gerätesteuerung und dem Hauptspeicher ohne Verwendung des Prozessors, ausgeführt durch den DMA-Controller

**ELAN**: Programmiersprache aus der ALGOL-Familie, die im L3-System verwendet wird

**GDP**: (General Driver Protocol) Konvention für die Interprozeßkommunikation mit Gerätetreibern im L3-System

**I/O-Adresse (-Port)**: Die Ein- und Ausgabe von Daten von bzw. zu Gerätesteuerungen erfolgt mit speziellen Prozessorbefehlen über I/O-Ports. Die 80x86-Prozessoren können 65536 verschiedene Ports über I/O-Adressen unterscheiden.

**Interrupt**: asynchrone Unterbrechung der Befehlsabarbeitung durch eine Gerätesteuerung, die zum Aufruf eines speziellen Unterprogramms (Interrupt-Routine) führt

**IRQ**: (Interrupt Request Level) IBM-kompatible Personalcomputer haben 16 verschiedene Steuerleitungen zur Auslösung von Interrupts, die Gerätesteuerungen zugeordnet werden können. Der IRQ identifiziert eine bestimmte Interrupt-Quelle mit einer Nummer von 0 bis 15. Einige IRQ sind fest zugewiesen (z.B. IRQ 1 für die Tastatursteuerung). Jede Interrupt-Quelle hat eine bestimmte Priorität (daher der Name).

**LAN**: (Local Area Network) Computernetzwerk mit geringer Ausdehnung

**Multicast**: Sendung an eine Gruppe von Stationen

**NIC**: (Network Interface Controller) elektronische Baugruppe für den Anschluß eines Computersystems an ein Computernetzwerk

**PCI**: (Peripherial Component Interconnect) Standard für den Anschluß von Gerätesteuerungen an ein Computersystem, definiert die elektrischen und mechanischen Eigenschaften eines Bussystems und die Verwaltung von Konfigurationsinformationen der angeschlossenen Geräte

**Promiscuous Mode**: Betriebsart eines Netzwerk-Controllers, in der dieser alle Datagramme (unabhängig von deren Zieladresse) empfängt und weiterleitet

**Scheduler**: Teil eines Multitasking-Betriebssystems, der die Zuteilung des Prozessors steuert

**SCSI**: (Smaller Computer Systems Interface) Standard zum Anschluß von maximal 7 Peripheriegeräten (z.B. Festplatte, CD-ROM) an ein Computersystem

**Seite**: (des Speichers) Verwaltungseinheit bestimmter Größe (4 KByte bei 80x86-Prozessoren) für die virtuelle Speicherverwaltung

**Subchannel**: Kanalnummer im Nachrichtenkode einer GDP-Nachricht, die verschiedene physische oder logische Geräte unterscheidet

**Supervisor**: L3-Systemprozeß, der für die Verwaltung von Tasks und Threads zuständig ist

**Task**: Adreßraum mit mindestens einem Kontrollfluß (Thread)

**TCP/IP**: Familie von Netzprotokollen mit weiter Verbreitung, die ursprünglich für das ARPANET (Vorgänger des Internet) entwickelt wurde. TCP (Transmission Control Protocol) steuert die Ende-zu-Ende-Verbindung zweier Kommunikationspartner. IP (Internet Protocol) realisiert die Übertragung von einzelnen Paketen über ggf. mehrere Rechner hinweg.

**Thread**: Kontrollfluß innerhalb einer Task, identifizierbar über einen Thread-ID (64-Bit-Wort in L3)

**Timeout**: Angabe, nach welcher Zeit eine Operation abgebrochen werden soll, wenn sie bis dahin nicht vollendet ist

**Unicast**: Sendung an genau eine Station

# Literaturverzeichnis

[Accomodat 93] L3 Referenzhandbuch; Accomodat GmbH, 1993

[Beck 94] Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner, D.: Linux-Kernel-Programmierung, Algorithmen und Strukturen der Version 1.0; Addison-Wesley, Bonn, 1994

[Becker 95] Becker, D.: 3Com 3c590/3c595 "Vortex" Ethernet Driver for Linux; Quellkode, Version 0.08, 14.09.1995, http://cesdis.gsfc.nasa.gov/linux/drivers/vortex.html

[Goel 96] Goel, S., Duchamp, D.: Linux Device Driver Emulation in Mach; Computer Science Department, Columbia University, 1996 USENIX Annual Technical Conference

[Gortmaker 95] Gortmaker, P. (ed.): Linux Ethernet-Howto; Version 2.4, 27 May 1995, http://sunsite.unc.edu/mdw/HOWTO/Ethernet-HOWTO.html

[FSF 91] GNU General Public License; Free Software Foundation, Inc., Cambridge, MA, Version 2, June 1991

[Heinrichs 88] Heinrichs, D.: Ethernet Treiber unter L3; L3 Papier, 20.09.1988

[Hohmuth 96] Hohmuth, M., Schalm, M., Wolter, J.: L3 Documentation; TU Dresden, 1996

[Johnson 95] Johnson, M.K.: The Linux Kernel Hackers´ Guide; Version 0.6, 1995

[Liedtke 90] Liedtke, J., Ruland, R.: L3 - Driver Installation and Configuration; L3 Papier, 01.03.1990

[Liedtke 91a] Liedtke, J., Bartling, U., Beyer, U., Heinrichs, D., Ruland, R., Szalay, G.: Two Years of Experience with a µ-Kernel Based OS; Operating Systems Review, April 1991, S. 51 ff.

[Liedtke 91b] Liedtke, J., Ruland, R.: L3 - IO Conventions; L3 Papier, 04.09.1991

[Liedtke 93] Liedtke, J.: Improving IPC by Kernel Design; 14th ACM Symposium on Operation System Priciples (SOSP), Asheville, NC, 1993, S. 175 ff.

[Linux 95] Linux-Kernel; Quellkode, Version 1.2.13, August 1995, ftp://www.inf.tu-dresden.de/pub/linux/kernel-funet/v1.2/linux-1.2.13.tar.gz

[Mach 95] Mach4-Kernel; Quellkode, Version UK02p21, 03.11.1995, ftp://flux.cs.utah.edu/flux/mach4-i386-UK02p21.tar.gz

[Podschun 95] Podschun, T.E.: Das Assemblerbuch - 8086/88/87, 80286/287, 80386/387, 80486 und Pentium, Addison-Wesley, Bonn, 1995

[Schönbeck 94] Schönbeck, M.: L3 BIOS Driver for Novell NE1000, NE2000 etc.; Quellkode, Version 30031, 31.10.94

[Schönbeck 96] Persönliche Mitteilung von M. Schönbeck

[Stevens 94] Stevens, W.R.: TCP/IP Illustrated, Volume 1, The Protocols; Addison-Wesley, Reading, MA, 1994

[Stevens 95] Stevens, W.R., Wright, G.R.: TCP/IP Illustrated, Volume 2, The Implementation; Addison-Wesley, Reading, MA, 1995

[Stiller 96] Stiller, A.: Pascal goes PCI, Schnittstelle zum PCI-BIOS; c´t 2/1996, S. 266 ff.

[Tanenbaum 90] Tanenbaum, A.S.: Betriebssysteme - Entwurf und Realisierung, Teil 1 - Lehrbuch; Carl Hanser Verlag, München, 1990

[Tanenbaum 91] Tanenbaum, A.S.: Computer Networks; Prentice-Hall Inc., Englewood Cliffs, NJ, 1991

[Tanenbaum 95] Tanenbaum, A.S.: Verteilte Betriebssysteme; Prentice-Hall-Verlag, München, 1995

[Wolter 95] Wolter, J.: Emulation des UNIX-Prozeßkonzepts auf dem Mikrokern L3; Diplomarbeit, Technische Berichte der Fakultät Informatik, TU Dresden, Dezember 1995