NE2000: Hardware und Design eines Treibers

Ausarbeitung zum Praktikum AKBPII
vom 23.2. – 5.3.2004

von Manuel Götz

Übersicht:

1. Allgemeines zum NE2000-Standard

Im Rahmen des Praktikums AKBPII war die Aufgabe gestellt, einen Treiber für eine NE2000-kompatible Netzwerkkarte zu schreiben.
Zum NE2000-Standard sollte erst einmal eine kurze Entstehungsgeschichte betrachtet werden, da sie die Intention hinter dem Design erklärt.
Er wurde das erste Mal im NatSemi (National Semiconductor) Databook veröffentlicht. Er war allerdings nicht zur tatsächlichen Realisierung gedacht, auch sollte auch kein späterer Hardware-Standard geschaffen werden.
Es sollte „nur“ ein Vorschlag dafür gemacht werden, was eine Ethernet-Netzwerkkarte mindestens an Hardware zu bieten haben müsste.
Die Firma Novell jedoch, die auch damals schon eher für ihre Software bekannt war, benötigte für ihre Software eine entsprechende Hardware. Daher übernahm Novell einfach diesen Vorschlag.
Der Standard hat sich jedoch, nicht zuletzt aufgrund der Software von Novell,

durchgesetzt.
Auch heute werden noch NE2000-kompatible Karten erstellt und folglich Treiber dafür geschrieben, obwohl der Standard inzwischen veraltet ist und mit neueren Entwicklungen nur schwer konkurrieren kann. Dennoch werden sogar NE2000-kompatible PCI-Karten gefertigt.
Allerdings kann aufgrund der Hardware-Architektur dadurch fast kein Vorteil erzielt werden, v.a. im Bezug auf Geschwindigkeit.

Man sieht also: der NE2000-Standard ist ein weiteres Beispiel in der Geschichte der Standards, die sich einfach aufgrund von aktuellen Gegebenheiten durchgesetzt und behauptet haben, obwohl es vielleicht vom rein technischen Standpunkt aus bessere Lösungen gegeben hätte. Aber er existiert, ist weit verbreitet und erfreut sich auch heute noch großer Beliebtheit.
Daher ist es aus praktischer Sicht absolut nachvollziehbar, einen Treiber für NE2000-kompatible Karten zu schreiben und somit die Unterstützung durch das Betriebssystem zu gewährleisten.

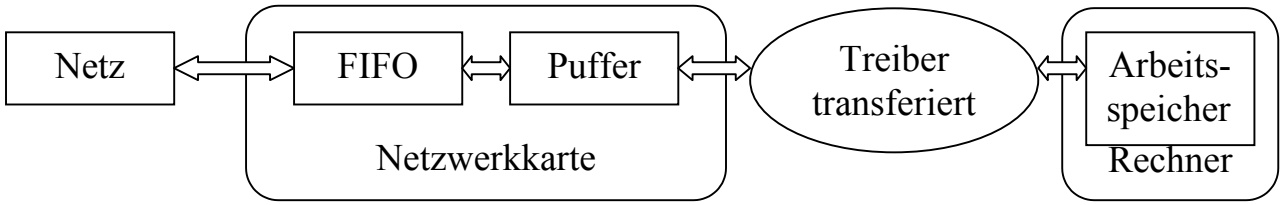
2. Hardware einer NE2000-kompatiblen Karte

Die im Folgenden genannten Hardware-Spezifika bilden nur eine minimale Ausgangsbasis, d.h. eine kompatible Karte muss z.B. über die genannten Register verfügen, sie kann aber auch noch viele weitere besitzen und anbieten, die im Standard nicht genannt werden. Es gibt dann im Prinzip einen kompatiblen Modus, in dem nur die Hardware-Ressourcen, die im Standard aufgeführt sind, genutzt werden und einen erweiterten Modus, in dem die Karte mit all ihren Fähigkeiten angesteuert wird. Die 2. Möglichkeit ist meist die Bessere und Schnellere, aber selbstverständlich ist ein Treiber für eine solche Ansteuerung nicht so universell einsetzbar und kompatibel, sondern eben kartentypspezifisch. In der Realität trifft man auch bei fast jeder NE2000-kompatiblen Karte auf beide Möglichkeiten. Ein gutes Beispiel sind die Realtek-Karten mit dem 8029-Chipsatz (z.B. RTL8029AS) (siehe [5]).

2.1. Allgemeiner Aufbau

Zu Beginn der Hardwarebeschreibung sollen erst einmal die grundlegenden Elemente der Hardware genannt werden, mit denen man sich als Programmierer eines Treibers auseinandersetzen muss.
Dies sind die Register, der Speicher und, nur am Rande, der FIFO.
Die Register benötigt man zur Bedienung und Konfiguration der Karte.

Im Speicher werden die eingehenden und ausgehenden Pakete zwischengespeichert, bzw. vom und zum Arbeitsspeicher des Rechners kopiert. Sowohl auf Register als auch auf Speicher hat der Programmierer direkten Zugriff. Er muss den Speicher aber beim Initialisieren der Karte erst einmal richtig konfigurieren, um ihn nutzen zu können.
Der FIFO dient der Karte, um Schwankungen in der Übertragungs-geschwindigkeit des Netzes auszugleichen ([6], S.9). Auf diesen ist kein softwaremäßiger Zugriff vorgesehen. Er wird daher hier nur der Vollständigkeit halber genannt und spielt in der weiteren Betrachtung keine Rolle.
Zum besseren Verständnis zeigt folgende Grafik den Weg jedes ein- und ausgehenden Pakets durch die Hardware
(Eingehend: von links nach rechts; Ausgehend: von rechts nach links):



2.2. Register

No (Hex)	Page0		Page1	Page2
	[R]	[W]	[R/W]	[R]
00	CR	CR	CR	CR
01	CLDA0	PSTART	PAR0	PSTART
02	CLDA1	PSTOP	PAR1	PSTOP
03	BNRY	BNRY	PAR2	-
04	TSR	TPSR	PAR3	TPSR
05	NCR	TBCR0	PAR4	-
06	FIFO	TBCR1	PAR5	-
07	ISR	ISR	CURR	-
08	CRDA0	RSAR0	MAR0	-
09	CRDA1	RSAR1	MAR1	-
0A	***8029ID0***	RBCR0	MAR2	-
0B	***8029ID1***	RBCR1	MAR3	-
0C	RSR	RCR	MAR4	RCR
0D	CNTR0	TCR	MAR5	TCR
0E	CNTR1	DCR	MAR6	DCR
0F	CNTR2	IMR	MAR7	IMR
10-17	Remote DMA Port			
18-1F	Reset Port			

(Tabelle siehe [5], S. 11)

In oben stehender Tabelle erhält man einen kompletten Überblick über die Register der Karte. Die Register sind in 3 Seiten unterteilt. Dies ist für das Ansprechen von Bedeutung, denn bevor man ein Register lesen oder schreiben kann, muss man zuerst einmal die Seite im Command-Register (CR) festlegen, d.h. man braucht z.B. für jeden Register-Lesezugriff vorher noch einen Schreibzugriff im Command-Register.
Dies erklärt, warum das Command-Register auf jeder Seite verfügbar sein muss: man kann für dieses vorher keine Seite angeben.

Weitere wichtige Register sind z.B. die PARs (Physical Address Register). Hier steht die Ethernet-Adresse, mit der jedes eingehende Paket verglichen wird. Danach wird über dessen Annahme entschieden (wenn sie mit der Zieladresse des Pakets übereinstimmt, ist das Paket für diesen Rechner gedacht und wird angenommen).
Die Funktionen von BNRY (Boundary Register), CURR (Current Register), PSTART und PSTOP sind für den Empfangspuffer wichtig und werden dort näher erklärt.
Das IMR (Interrupt Mask Register) wird beim Hochfahren gesetzt (kann danach aber noch verändert werden). In ihm bestimmt man, welche Ereignisse einen Interrupt auslösen sollen und welche nicht.
Durch Auslesen des ISR (Interrupt Status Register) beim Eintreffen eines Interrupts kann man erkennen, welches Ereignis den Interrupt ausgelöst hat.
Dies war ein Ausschnitt der wichtigsten Register. Einige werden später bei der Anwendung noch genauer betrachtet. Das Wichtigste hier ist die Struktur, also die Unterteilung in Seiten, und die Tatsache, dass einige Register auf unterschiedlichen Seiten gelesen und geschrieben werden. Genauere und vollständige Informationen zu den einzelnen Registern sind der Spezifikation oder einigen Dokumenten aus der am Schluss angefügten Literaturliste zu entnehmen. (v.a.: [5], S.11-16)

2.3. Speicher

2.3.1 Speicherlayout

Das zweite große Hardware-Element auf der Karte ist der Speicher.
Er ist untergliedert in verschiedene Bereiche, wobei die wichtigsten sind:

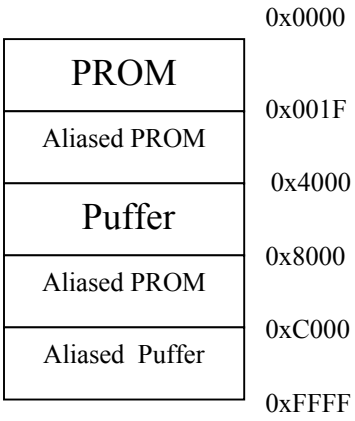
- PROM

  Dieser Teil ist als persistenter ReadOnly-Speicher ausgelegt und enthält wichtige Daten, die zum Beispiel zum Initialisieren benötigt werden und beim Herunterfahren der Karte nicht verloren gehen dürfen([3], S.9).

- Puffer

  Im Puffer werden die im laufenden Betrieb ein- und ausgehenden Pakete zwischengespeichert, bis sie vom Treiber in den Arbeitsspeicher geschrieben oder versandt werden.

Zur Veranschaulichung folgt der Aufbau des Speichers in grafischer Form. Oben nicht näher beschriebene Teile sind dabei im weiteren Verlauf bedeutungslos und werden nur der Korrektheit wegen angezeigt:



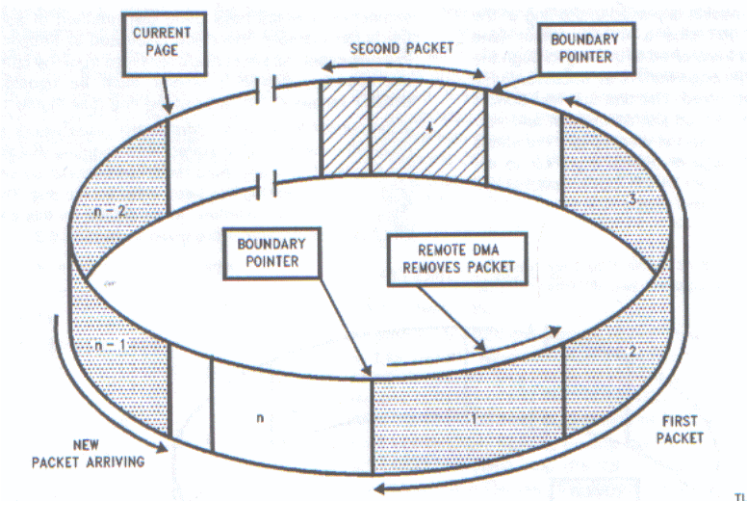
(Grafik siehe [1], S.2)

Da der Puffer das Stück im Speicher ist, auf den der Treiber sicherlich die meisten Zugriffe durchführt, soll er nun näher betrachtet werden.

2.3.2 Puffer

Der Puffer soll sowohl die ein-, als auch die ausgehenden Pakete speichern. Dazu muss er bei der Initialisierung erst einmal in 2 Teile aufgespalten werden. Dies geschieht, indem nach einem Reset der Karte die Register TPSR, PSTART und PSTOP gesetzt werden.
Das TPSR (Transmit Page Start Register) enthält die Adresse, an die das nächste (falls sie nie geändert wird auch alle) zu versendende Paket geschrieben wird. Da ein Ethernet-Paket bis zu 1514 Byte groß sein kann, sollte man nach dieser Adresse auch mindestens so viel Platz bereitstellen und frei lassen.
Ist ein Paket an diese Stelle geschrieben worden, muss nur noch der Versendebefehl im Command-Register gegeben werden.
Das Paket wird dann automatisch an den FIFO auf der Karte übertragen und von dort auf das Netz ([2], S.1+2).
Ist dies geschehen, wird einer von 2 möglichen Interrupts ausgelöst. Entweder wurde das Paket erfolgreich versandt oder es gab einen Fehler. Der Treiber kann dann entsprechend reagieren ([5], S.13).
Der Puffer für die empfangenen Pakete wird durch PSTART und PSTOP begrenzt. Der Grund dafür, dass beim Empfangspuffer 2 Register zum begrenzen nötig sind, liegt darin, dass der Empfangspuffer in Form eines Ringpuffers aufgebaut ist.
Dieser Ringpuffer besteht aus 256 Byte großen Seiten.
Die Anzahl dieser Seiten hängt natürlich von seiner Größe ab.
PSTART beschreibt den Beginn der ersten Seite des Ringpuffers und PSTOP den Beginn der ersten Seite, die nicht mehr gültig ist.
Innerhalb dieser Grenzen müssen alle vom FIFO erhaltenen Nachrichten gespeichert werden.

Das folgende Bild zeigt den Aufbau des Ringpuffers, die darin dargestellten Pointer werden im Anschluss erklärt.



(Grafik und Beschreibung nach [2])

Jedes eingehende Paket wird von der Karte mit einem 4 Byte großen Header versehen, in dem der Status des Pakets (z.B. fehlerfrei), der Beginn des nächsten Pakets (die Seite mit dem nächsten Header) und die Länge des Pakets enthalten sind.
Um den Ringpuffer auszulesen muss man mit folgenden Registern arbeiten:

- BNRY(Boundary Pointer)
  Der Boundary Pointer zeigt immer auf den Vorgänger der nächsten zu lesenden Seite, die in den Arbeitsspeicher transferiert werden soll. Ist das Paket aus dem Kartenspeicher ausgelesen und in den Arbeitsspeicher eingelesen worden, muss der Boundary Pointer vom Treiber weitergesetzt werden. Dies geschieht mit Hilfe des Headers, in dem die Seite, die den Beginn des nächsten Pakets enthält, steht. Beachtet werden müssen hierbei die durch PSTART und PSTOP definierten Grenzen des Ringpuffers, d.h. der Boundary Pointer muss nach Überschreiten des PSTOP-Wertes wieder beim PSTART-Wert beginnen.

- CURR(Current Page Register)
  Der Current Pointer zeigt auf die nächste freie Seite, in die Pakete geschrieben werden können. Er wird nur beim Initialisieren einmal gesetzt und danach selbstständig von der Karte verwaltet. Er beachtet automatisch die Grenzen des Ringpuffers. Der Current Pointer sollte nach dem Hochfahren nicht mehr vom Treiber verändert werden.

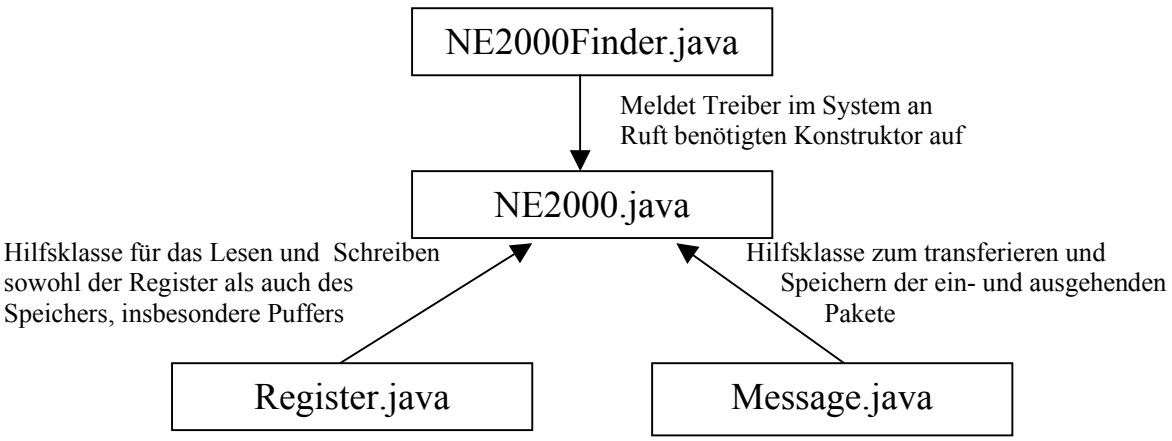
Um einen Überlauf des Ringpuffers zu vermeiden muss genügend Speicher aus dem Puffer für den Empfang von Paketen bereitgestellt werden, d.h. die Differenz von PSTOP und PSTART ist zu maximieren.
Die komplette Transferierung der Speicherinhalte ist nur über DMA-Zugriffe möglich. Im NE2000-Standard ist kein Memory Mapping vorhergesehen, daher muss für jeden Speicherzugriff auf Remote DMA zurückgegriffen werden.

3. Design eines NE2000-kompatiblen Treibers

Da im 2.Kapitel die Grundlagen betreffs der Hardware gelegt wurden, kann nun das Design eines Treibers für die NE2000-Karten erläutert werden. Als Vorlage dient hier der im Rahmen des AKBPII-Praktikums erstellte Treiber. Dieser kann und soll durchaus nicht als Alleinlösung gesehen werden. Es ist mehr als ein Vorschlag gedacht, der natürlich noch verbesserungsfähig ist.

3.1. Klassen

Zu Beginn wird das grundlegende Design dargestellt.
Der Treiber wurde in 4 Klassen unterteilt, um seine Aufgaben zu erfüllen.
Das folgende Diagramm soll einen Überblick vermitteln:



3.2. Starten des Treibers

Beim Hochfahren des Systems wird von der boot.rc aus über mehrere Klassen die NE2000Finder.java aufgerufen. Diese sucht nach einer NE2000-kompatiblen Karte im Rechner. Das Suchen geschieht mit Hilfe der Vendor ID und Device ID, die mit allen gefundenen Netzwerkkarten verglichen wird.

Gilt eine davon als NE2000-kompatibel, war die Suche erfolgreich. Nachdem durch die IDs der genaue Typ der Karte bekannt ist, bleibt noch zu prüfen, ob es eine PCI- oder eine ISA-Karte ist. Bei einer ISA-Karte müssen in der boot.rc noch zusätzliche Parameter wie die z.B. Interrupt-Nummer angegeben werden, damit die Karte gestartet werden kann. Sind alle Angaben plausibel, wird nun der entsprechende Konstruktor der NE2000.java gestartet. Die verschiedenen Konstruktoren sind leider nötig, da z.B. die Realtek initiale, nicht NE2000-kompatible Einstellungen braucht, um sie in den kompatiblen Modus zu versetzen. Wurde der Konstruktor erfolgreich gestartet, wird an dem zurückgelieferten Objekt noch die von der NE2000.java angebotene Methode open() aufgerufen, die die Initialisierung der Karte vornimmt.

3.3. Initialisieren der Karte

Durch das Initialisieren wird die Karte beim Hochfahren in einen betriebsfähigen Zustand versetzt. Dazu müssen folgende Aufgaben erledigt werden, die open() in der NE2000.java ausführt:

1. Ein Hardware-Reset der Karte, durch Schreiben in den ResetPort.
2. Initialisieren des DCR (Data Configuration Register)
   In diesem Register werden Einstellungen vorgenommen, die v.a. die Art der Übertragung der Daten beim DMA-Zugriff zwischen Rechner und Karte betreffen, z.B. Byte- oder Wortweise Übertragung, die Byte Order etc.
3. Löschen von RBCR0, RBCR1 (Remote Byte Count Register)
   Diese Register geben an, wie lang das zu übertragende Paket im Puffer ist, bevor die Übertragung an den FIFO erfolgt.
   Die Größenangabe erfolgt in Bytes.
   Die Register werden initial auf 0 gesetzt.
4. Initialisieren des RCR (Receive Configuration Register)
   Die Einstellungen betreffen die ankommenden Pakete.
   Man kann entscheiden, ob man beschädigte Pakete speichern möchte oder gleich verwirft, ob man Broad- und Multicast-Pakete empfangen möchte u.v.m.
5. Initialisieren des TCR (Transmit Configuration Register)
   In diesem Schritt wird vor allem über die Art des Loopback-Modus entschieden, der während der restlichen Initialisierungen aktiv ist.
   Es ist Modus 1 oder 2 zu wählen.
6. Initialisieren des Ringpuffers
   BNRY (Boundary Pointer), PSTART und PSTOP werden auf die gewünschten Adressen im Puffer gesetzt. Es ist zu beachten, dass diese 8-bit Register von einer 16-bit Adresse nur die höchstwertigen 8 bit enthalten. Aufgrund der 256 Byte großen Seiten im Ringpuffer

ist eine feinere Auflösung unnötig.

7. Löschen des ISR (Interrupt Status Register)
   Das ISR wird mit 0xFF überschrieben. Dadurch werden die im bisher undefinierten Zustand evtl. anliegenden (hier noch unsinnigen) Interrupts gelöscht.
8. Initialisieren des IMR (Interrupt Mask Register)
   In diesem Register entscheidet man, welche Interrupts die Karte senden soll und welche maskiert werden sollen.
9. Initialisieren der PAR0-5 (Physical Address Register)
   Aus dem PROM im Speicher der Karte (an Stelle 0) wird die Ethernet-Adresse ausgelesen und anschließend in die PAR geschrieben.
   Mit diesen Registern wird die Zieladresse jedes ankommenden Pakets verglichen und dann über dessen Annahme entschieden.
9. Initialisieren der MAR0-7 (Multicast Address Register)
   Siehe 9., nur für Multicast-Pakete
11. Initialisieren des CURR (Current Pointer)
    Der Current Pointer wird am besten auf den Wert (Boundary Pointer + 1) gesetzt, also eine Seite weiter im Ringpuffer.
12. Setzen des Startmodus im Command-Register
    Der Stopwert im Command-Register wird gelöscht und das Start-Bit gesetzt. Zusätzlich wird der DMA-Zugriff aktiviert.
13. Setzen des TCR (Transmit Configuration Register)
    Das TCR wird nun auf den Normal Mode zurückgesetzt.
    Erst jetzt ist die Karte betriebsbereit.

Die Reihenfolge dieser Schritte ist weitgehend einzuhalten, einige wenige können vertauscht werden. Sind sie alle richtig ausgeführt worden, ist die Karte nun sende- und empfangsbereit. (siehe [1], S.5; [4], S.1)

3.4. Versenden von Paketen

Zum Versenden von Paketen bietet der Treiber in der NE2000.java zwei Methoden an, die transmit(...) und die transmit1(...).
Die transmit-Methode hat als Parameter nur ein Memory-Objekt, das verschickt werden soll. Die transmit1-Methode bietet durch die zusätzlichen Parameter offset und size noch die Möglichkeit, erst ab einem bestimmten offset im Memory-Objekt mit dem Versenden zu beginnen oder die Länge in Bytes anzugeben, die verschickt werden soll. Das hat u.a. den Vorteil, dass bei einem 40 Byte großen Paket auch nur 40 Byte versendet werden und nicht 1514, wie es bei der transmit-Methode der Fall ist. Da die transmit-Methode einfach die transmit1-Methode mit den zusätzlichen Parametern 0 als offset und 1514 als size aufruft, verfahren beide Methoden im weiteren Verlauf gleich.

Zuerst wird geprüft, ob die Pufferstelle im Versendespeicher, an die das Paket geschrieben werden soll, frei ist. Ist dies nicht der Fall, wird die Nachricht in eine Liste mit zu versendenden Paketen eingehängt, die jedesmal abgearbeitet wird, wenn ein Paket fertig versandt wurde (dies erkennt man am entsprechenden Interrupt). Ist die Pufferstelle frei, wird zuerst einmal in die TPSR0/1 (Transmit Page Start Register) die Adresse mit dem Beginn der Nachricht im Puffer geschrieben. Anschließend schreibt man in die TBCR0/1 (Transmit Byte Count Register) die Länge des Pakets. Nun schreibt man noch den Inhalt des Memory-Objekts an die in den TPSR0/1 angegebene Stelle.
Ist dies geschehen, gibt man mit dem Command-Register den Befehl zum Start des Versendevorgangs. (siehe [1], S.12+13; [4], S.6)
Zum Schluß muss nur noch ein leeres Memory-Objekt zurückgegeben werden.
Da die transmit-Methoden sowohl von außerhalb als auch von der Klasse selbst aufgerufen werden können (um die Liste der zu versendenden Pakete abzu-arbeiten), ist noch zu synchronisieren. Es darf immer nur eine transmit-Methode auf einmal in Ausführung sein.

3.5. Empfangen von Paketen

Das Empfangen von Paketen beginnt damit, dass man einen Interrupt wegen eines eingegangenen Pakets im Ringpuffer erhält. Dieses liest man mit Hilfe des Headers, der die Länge des Pakets enthält, aus und hängt es in eine Liste mit empfangenen Paketen ein. Der Header des ersten zu lesenden Pakets steht genau an der Stelle, auf die der Boundary Pointer zeigt. Dann wird der Boundary Pointer (BNRY) wieder richtig gesetzt, da dieser immer noch auf der Seite vor dem Beginn des gerade ausgelesenen Pakets steht. Er wird auf die Seite vor dem Beginn des nächsten Pakets gesetzt, dessen Adresse man ebenfalls aus dem Header erhält. Ist dies alles geschehen, wird überprüft, ob der Boundary Pointer genau den Wert (Current Pointer – 1) enthält, denn da das ganze Auslesen im Interrupthandler erfolgt ist, können inzwischen weitere Pakete und Interrupts eingetroffen sein, die ignoriert wurden, weil die Interrupts gesperrt waren.
Der Auslesevorgang wird also solange wiederholt, bis der Boundary Pointer genau eine Seite hinter dem Current Pointer steht. Dann wurde alles ausgelesen.
Zum Schluss werden die Interrupts wieder freigegeben.
Danach wird noch einmal überprüft, ob zwischen Prüfen der Pointer und dem Freigeben der Interrupts nicht noch ein Paket eingetroffen ist.
Sollte dies der Fall sein, beginnt man von vorne, die Interrupts werden gesperrt, das Paket ausgelesen etc ([4], S.9).

Um Pakete zu empfangen muss der Empfänger einen Thread beim Treiber anmelden. Dieser Thread wird gestartet, wenn ein Paket in die Liste der empfangenen Pakete eingehängt wurde und wieder schlafen gelegt, wenn die Liste leer ist oder der Empfänger keine Pakete mehr aufnehmen kann.

Wird der Thread nicht von außerhalb gestoppt, läuft er so lange, bis kein Paket mehr in der Liste übrig ist. Während der Thread läuft, liest er jeweils ein Paket aus der Liste aus und gibt das Memory-Objekt mit dem Inhalt an den Empfänger weiter. Damit der NE2000.java nicht die Memory-Objekte ausgehen, bekommt der Thread im Gegenzug ein Memory-Objekt vom Empfänger zurück, in das er später wieder ein anderes angekommenes Paket aufnehmen kann.
Ist die Anzahl der Pakete, die im Treiber gespeichert werden können, erschöpft, weil zu wenige Pakete abgeholt und immer neue empfangen werden, wird automatisch immer das älteste Paket einfach mit einem neueren überschrieben und geht verloren. Um die verlorenen Pakete muss sich dann eine übergeordnete Schicht kümmern, d.h. sie muss eine Anfrage für nochmaliges Übermitteln an den Sender stellen.

4. Ausblick

Zum Abschluß erfolgt ein kleiner Ausblick, der zeigt, was bei der wirklichen Implementierung des Treibers erreicht wurde. Zudem nennt er noch einige wünschenswerte Ergänzungen bzw. Modifizierungen.

Der Treiber läuft getestet mit 2 Karten, einer Realtek und einer Winbond.
Die Realtek läuft allerdings nicht sofort im NE2000-Modus, sondern es müssen dafür noch gesonderte Einstellungen gemacht werden.
Daher benötigt sie auch einen eigenen Konstruktor.

Ein guter Ansatzpunkt zur Verbesserung wäre, den Interrupthandler wesentlich zu verschlanken und z.B. das Auslesen eines Pakets aus dem Ringpuffer in einem Thread durchzuführen, der vom Interrupthandler nur noch geweckt wird. Somit müssten nicht während des gesamten Auslesens die Interrupts gesperrt bleiben. Dies würde die Interruptlatenzzeit reduzieren.
Es war jedoch bei keinem einzigen Test mit einer realen Anwendung der Fall war, dass die Nachrichten schneller kamen, als der Interrupthandler dies bewältigen konnte. Die Liste für die eingehenden Nachrichten wuchs niemals. Einzig und allein ein FloodPing ermöglichte uns überhaupt ein Prüfen dieser Funktion. Aber auch damit lief der Treiber absolut stabil und konnte schließlich nur durch einen überforderten Garbage Collector gestoppt werden.

Daraus ergibt sich abschließend, dass ein Treiber vorliegt, der im Detail sicher noch Verbesserungen verträgt, aber auch jetzt schon seine Aufgabe mit angemessener Performance und hoher Stabilität erledigt.

5. Literaturliste

[1] AT/LANTIC Software Developer´s Guide
National Semiconductor
Application Note 887
David Milne
May 1993

[2] DP8390 Network Interface Controller: An Introductory Guide
National Semiconductor
Application Note 475
May 1993

[3] DP83905EB-AT AT/LANTIC Hardware User´s Guide
National Semiconductor
Application Note 897
Larry Wakeman
July 1993

[4] Writing Drivers for the DP8390 NIC Family of Ethernet Controllers
National Semiconductor
Application Note 874
July 1993

[5] RTL8029AS
Realtek PCI Full-Duplex Ethernet Controller with built-in SRAM
Advanced Information
Realtek Semiconductor Corp.

[6] W89C940
ELANC-PCI (TWISTED-PAIR ETHER-LAN CONTROLLER WITH PCI-INTERFACE)
Winbond
Electronics Corp.