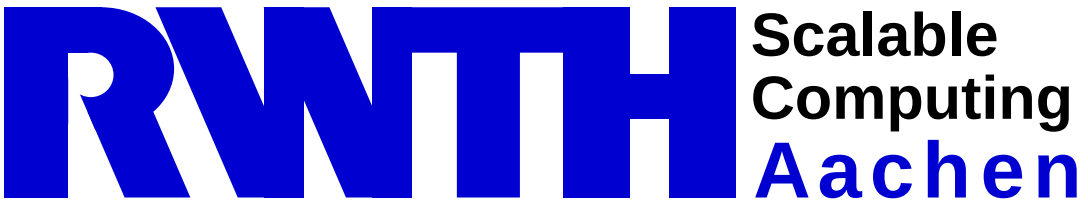



LEHRSTUHL FÜR BETRIEBSSYSTEME
Univ.-Prof. Dr. habil. Thomas Bemmerl

# ENTWICKLUNG UND IMPLEMENTIERUNG EINES NETZWERKKARTEN-TREIBERS SOWOHL UNTER LINUX ALS AUCH RT LINUX ZUR REALISIERUNG EINES ECHTZEITFÄHIGEN NETZWERKPROTOKOLLS

cand. ing. Michael Reke
Matrikelnummer 206302

**Diplomarbeit**

an der

Rheinisch-Westfälischen Technischen Hochschule Aachen
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Betriebssyteme

**Betreuer**

Dipl.-Inform. Stefan Lankes

**Gutachter**

Univ.-Prof. Dr. habil. Thomas Bemmerl

# Erklärung

Name: Michael Reke

Matrikelnr.: 206302

Hiermit erkäre ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Aachen, den 23.02.2001

# Inhaltsverzeichnis

# 1 Einleitung

Für ein Netzwerk ergeben sich aus dem Einsatzzweck unterschiedliche Anforderungen. Während in einer Anwendung z.B. die Bandbreite im Vordergrund steht, wird auf einem anderen Gebiet eventuell mehr Wert auf einen prioritätsbasierten Zugriff gelegt. Das führte in der Vergangenheit zu unterschiedlichen Hardware-Lösungen, die heute nebeneinander existieren. Soll ein Netzwerk in einem sicherheitskritischen Bereich eingesetzt werden, so ist die entscheidende Anforderung, das Verhalten des Systems in jeder Situation beschreiben zu können. Insbesondere die Reaktion auf einen Ausfall und das zeitliche Verhalten müssen bekannt sein. Wenn eine Nachricht auf einem Netzwerkkanal übertragen wird, so muss garantiert sein, dass das Paket den Empfänger vor Ablauf einer gesetzten Frist in jedem Fall erreicht. Diese Systeme fasst man unter dem Begriff der Echtzeit-Netzwerke zusammen.

Während diese in der Vergangenheit hauptsächlich in speziellen Bereichen wie der Raumfahrt oder Kernkraftwerken ihre Anwendung fanden, werden sie in Zukunft auch auf Gebieten mit großen Produktionsstückzahlen, wie der Automobiltechnik zum Einsatz kommen. Für zukünftige Systeme stellt sich die Herausforderung, komplexe sicherheitsrelevante Fahrzeugfunktionen mit Hilfe der Elektronik ohne mechanische Rückfallebenen zu realisieren. Diese sogenannten X-By-Wire Systeme für Lenkung, Bremse, Antriebsstrang und Fahrwerk lassen, neben Produktions- und Konstruktionsvorteilen, ein großes Potenzial für die Erhöhung aktiver und passiver Sicherheit erwarten [4]. Die Basis dieser Systeme ist die Kommunikation zwischen den verteilten Komponenten. Hierbei ist, neben den Anforderungen Fehlertoleranz und garantiertes Echtzeitverhalten, die Entkopplung des Zeitverhaltens (temporal firewall) von besonderer Bedeutung [4]. Das eingesetzte Netzwerkprotokoll muss dem gerecht werden.

Es hat sich in der Vergangenheit gezeigt, dass Technologien aus der Automobiltechnik, aufgrund der hohen Stückzahlen und der damit verbundenen niedrigen Produktionskosten, auch auf anderen Gebieten ihre Anwendung finden. Ein Beispiel hierfür ist der CAN-Bus. Ursprünglich zur Kommunikation im Fahrzeug entwickelt, wird er heute auch in der Automatisierungstechnik verstärkt eingesetzt. Es ist also zu erwarten, dass ein Echtzeit-Netzwerk ebenfalls in anderen Bereichen benutzt werden wird. Die derzeitigen Entwicklungen sehen jedoch für die Realisierung aufwändige Spezialhardware vor [4][11][26], was eine Adaption des Netzwerks auf andere Bereiche erschwert. Die Übertragung der Ansätze auf eine Software-Lösung, die Standard-Hardware nutzt, könnte dies jedoch kompensieren. Das Echtzeit-Netzwerk wäre dann unmittelbar in anderen technischen Sparten wie der Medizintechnik, der Prozesssteuerung, dem Flugzeugbau, usw. einsetzbar. Gegenstand dieser Diplomarbeit ist die Realisierung eines solchen Netzwerks.

Kapitel 2 beschreibt die Anforderungen und erläutert den gewählten Lösungsansatz. Mit Kapitel 3 wird eine Einführung in das Thema der Echtzeitnetzwerke gegeben und es erfolgt eine Diskussion der eingesetzten Komponenten mit Hinblick auf die gestellte Aufgabe. Die konkrete Umsetzung wird in Kapitel 4 vorgestellt. In Kapitel 5 erfolgt dann eine Bewertung der erzielten Ergebnisse in Bezug auf die genannten Anforderungen. Kapitel 6 bietet eine Zusammenfassung der Arbeit und einen Ausblick auf Erweiterungsmöglichkeiten.

# 2 Aufgabenstellung

Eine Grundvoraussetzung für den Aufbau echtzeitfähiger verteilter Systeme ist eine Netzwerkverbindung, die die speziellen Forderungen an die Übertragungsdauer erfüllt. Dies bedeutet, dass versandte Datenpakete innerhalb einer vorgegebenen Zeitspanne den Empfänger erreichen müssen und von ihm verarbeitet werden. In der Literatur wird gezeigt, dass diese Vorgabe am besten durch ein zeitgesteuertes System zu lösen ist, da hier im Gegensatz zu ereignisgesteuerten Systemen eine Unabhängigkeit von der momentanen Auslastung gegeben ist [11]. Hierbei wird die verfügbare Zeit in sogenannte Zeitscheiben aufgeteilt, die nach einem Zuteilungalgorithmus den Netzteilnehmern zugewiesen werden. Dies bedeutet im Zuteilungsfall für den Teilnehmer exklusiven Schreibzugriff auf das Netzwerk. Durch dieses Verfahren werden Datenkollisionen ausgeschlossen. Zurzeit werden jedoch in den meisten Anwendungsfällen ereignisgesteuerte Systeme verwendet, bei denen es abhängig von der Auslastung zu Kollisionen kommt.

Im Rahmen dieser Arbeit soll ein zeitgesteuertes Verfahren in einer PC-Umgebung implementiert werden, wie es im oberen Abschnitt beschrieben wurde. Dabei kommen herkömmliche Ethernet-Netzwerkkarten zum Einsatz. Als Betriebssystem wird RT Linux benutzt, das eine Echtzeiterweiterung zu Linux darstellt. Die Netzwerkschnittstelle von Linux soll parallel zur Echtzeit-Schnittstelle weiter betrieben werden. Dabei sollen die Phasen, in denen keine Echtzeit-Pakete übertragen werden, für Linux-Pakete zur Verfügung stehen. Im Gegensatz zu früheren Veröffentlichungen, die für ein zeitgesteuertes Netzwerk aufwändige Spezialhardware vorsehen [26][11], soll hier eine reine Software-Lösung in Form eines Netzwerkkarten-Treibers realisiert werden.

# 3 Grundlagen

## 3.1 Echtzeitfähigkeit im Netzwerk

Um Echtzeitfähigkeit im Netzwerk zu garantieren, sind spezielle Grundvoraussetzungen zu erfüllen. Diese Anforderungen unterscheiden sich von denen der herkömmlichen Netzwerke; so spielt zum Beispiel die Bandbreite einer Verbindung bei einem Echtzeitnetzwerk eine geringere Rolle als die Erfüllung der zeitlichen Vorgaben. Da aber auch herkömmliche Netzwerke, wie z.B. das Internet, ebenfalls immer größere Anforderungen hinsichtlich Echtzeitfähigkeit stellen (Videoübertragung, Telefongespräche, usw.), werden die hier vorgestellten Aspekte auch in diesem Bereich immer wichtiger. Sie bieten eine Möglichkeit zum Vergleich verschiedener Lösungen. Die wesentlichen Kriterien sollen zunächst kurz vorgestellt werden. Anschließend werden gängige Netzwerk-Zugriffsstrategien in Bezug auf diese Aspekte diskutiert.

### 3.1.1 Unterscheidungskriterien

Die im folgenden eingeführten Aspekte sind allgemeine Grundbegriffe, die teilweise im Widerspruch zueinander stehen. Die Gewichtung der einzelnen Eigenschaften hängt stark von der Anwendung ab. Eine Diskussion der Gegensätze erfolgt im Anschluss an die Auflistung.

- **Protokolllatenzzeit**

  Als Protokolllatenzzeit (protocol latency) wird die Zeit zwischen dem Start der Übertragung einer Nachricht an die Netzwerkschnittstelle des Sendeknotens bis zur Übergabe der Nachricht durch die Netzwerkschnittstelle des Empfängerknotens bezeichnet [11].

  Für ein Echtzeitnetzwerk ist eine möglichst kurze Protokolllatenzzeit wünschenswert. Wichtiger ist jedoch, dass sie einem minimalen Jitter unterliegt. Hiermit wird die maximale Streuung der Werte innerhalb eines betrachteten Zeitraums mit einer Anzahl von N Werten für die Latenzzeit $t_{Li}, i = 0 \ldots (N-1)$ bezeichnet:

  $$\text{Jitter} = \max(t_{Li}) - \min(t_{Li})$$

  Anwendungsprogramme verlassen sich auf die genaue Kenntnis der Latenzzeit. Der Jitter des Netzwerks überträgt sich damit auch auf die Applikation und ist somit die wichtigste Grösse zur Beurteilung der Leistungsfähigkeit einer echtzeitfähigen Netzwerklösung.

- **Zusammensetzbarkeit**

  Mit Zusammensetzbarkeit ist die Möglichkeit bezeichnet, mehrere Knoten zu einem System zu verbinden. Sie wird im wesentlichen durch folgende zwei Aspekte bestimmt [11].

1. Temporale Kapselung der Netzknoten: Das Kommunikationssystem der Netzknoten muss unabhängig von der Applikation arbeiten. Nur so kann ein störungsfreier Betrieb des Netzwerkes garantiert werden, da der zeitliche Abgleich durch die Kommunikationsschnittstelle selbst durchgeführt wird.

2. Erfüllung der Verpflichtungen der Clients: Der Master im Netzwerk muss sich auf die zeitlichen Verpflichtungen der Clients verlassen können. Kein Client darf den Master durch zu viele Anfragen überlasten und so die Funktion des gesamten Netzwerkes stören. Flusskontrolle kann helfen, diese Verpflichtung einzuhalten.

- **Flexibilität**

  Mit Flexibilität wird die Möglichkeit beschrieben, verschiedene Netzknoten miteinander zu verbinden. Diese können sich auch in ihrem zeitlichen Verhalten unterscheiden. Es muss aber auch eine flexible Netzkonfiguration möglich sein. Im Betrieb dürfen sich neue Knoten dem Netzwerk hinzuschalten.

- **Fehlererkennung**

  Es ist wichtig, dass ein Echtzeitnetzwerk in geeigneter, d.h. toleranter, Weise auf Fehler verschiedenster Art reagiert. Tritt ein Fehler auf, so muss der betroffene Knoten in einen sicheren Zustand fahren. Alle anderen Netzteilnehmer dürfen nicht beeinträchtigt werden. Das heisst, ein Fehler darf auftreten, darf aber nicht zum Ausfall des gesamten Systems führen (Fault Tolerance). Dabei wird zwischen verschiedenen Arten von Fehlern unterschieden:

  Kommunikationsfehler: Während einer Übertragung kann es z.B. durch elektromagnetische Einstreuung zu Fehlern kommen. Die Netzwerkschnittstelle muss diese Fehler erkennen und die betroffenen Netzknoten über das Auftreten informieren. Dabei darf es zu keiner Erhöhung der Protokolllatenzzeit oder des Jitters kommen. In einem Echtzeitnetzwerk ist es von besonderem Interesse, dass auch der Empfänger über den Verlust der Meldung informiert wird. Dies ist möglich, wenn die Meldungen periodisch verschickt werden. Der Empfänger weiss daher, wann er ein Paket zu erwarten hat. Bleibt das Paket aus, so ist ein Fehler aufgetreten. Dies steht im Gegensatz zu Nicht-Echtzeit-Netzwerken, wo Meldungen eher sporadisch verschickt werden und der Empfänger daher nicht wissen kann, wann er ein Paket erhält.

  Knotenfehler: In einem Echtzeitnetzwerk sollten alle Teilnehmer wissen, welche Knoten sich im Netz befinden. Fällt ein Knoten aus, so muss dies sicher erkannt werden und alle Teilnehmer müssen über den Ausfall informiert werden.

  End-to-End Acknowledgement: Jede Übertragung sollte in einem Echtzeit-System bestätigt werden. Dies ist aber nicht Aufgabe des Netzwerks sondern vielmehr Teil der Applikation. Wird beispielsweise ein Signal zum Schließen eines Sicherheitsventils ausgelöst, so sollte ein unabhängiger Sensor die neue Position des Stellers bestätigen. Das ist ein Beispiel für das Design-Prinzip „Never trust an actuator“ [11].

- **Netzstruktur**

  Im Prinzip kommt als Struktur nur eine sternförmige Busstruktur in Frage. Dafür sprechen die einfache Verkabelung im Gegensatz zu einer vollausgebauten Punkt-zu-Punkt-

Verbindung und die gleichförmige Übertragungszeit. Gegen eine Ringstruktur im Sinne von Token Ring[1] spricht die Fehlerbehandlung bei Knotenausfällen (Wer erzeugt das neue Token?). Bei der Strukturierung sollte auch darauf geachtet werden, dass die Netzknoten in sinnvolle kleine Einheiten im Sinne der Fehlertoleranz unterteilt werden. Dies garantiert, dass ein Ausfall einer Komponente nicht einen Gesamtausfall zur Folge hat. So müssen die sicherheitsrelevanten Komponenten in einem Automobil, bei dem die Steuerung ausschließlich auf verteilten elektronischen Systemen basiert, auch bei einem Unfall und dem daraus resultierenden Verlust eines Knotens erhalten bleiben.

Als Kommunikationsmodell sollte ein Multicast-Protokoll gewählt werden. Hierbei kann eine Meldung von mehreren Empfängern gleichzeitig empfangen und verarbeitet werden. Dies kann in einem Sicherheits-System mit mehreren Zustandsmonitoren wichtig sein. Tritt ein Fehler auf, so muss dieser möglichst gleichzeitig auf allen Monitoren gemeldet werden.

Wie zu Beginn des Abschnitts erläutert, erfolgt im folgenden eine Diskussion der Gegensätzlichkeit der eingeführten Kriterien.

Die Zusammensetzbarkeit von Knoten bezieht sich im wesentlichen auf die zeitliche Basis. Nur bei einer zeitlich aufeinander abgestimmten Implementation der Netzwerkknoten ist eine Zusammensetzbarkeit auf unterster Ebene möglich. In einem ereignisgesteuerten Netzwerk kann diese gemeinsame Zeitbasis jedoch nicht erreicht werden, da ein Client nur auf ein externes Ereignis reagiert und so alle Anfragen asynchron an den Host richtet.

Ein Netzwerk kann entweder für sporadische oder für periodische Daten effektiv arbeiten. Bei periodischen Daten ist ein möglichst kleiner Jitter erforderlich, da die Übertragungszeitpunkte a priori bekannt sind. Handelt es sich um sporadische Daten, so ist eine möglichst kleine Verzugszeit zwischen der Anforderung einer Datenübertragung und der tatsächlichen Übertragung notwendig. Diese Übertragungszeitpunkte sind vorher nicht bekannt.

Flexibilität kann in vollem Maße nur erreicht werden, wenn das Verhalten eines Knotens nicht im voraus bekannt sein muss. Sie steht damit im Gegesatz zur Fehlererkennung. Diese beruht in einem zeitgesteuerten System jedoch auf diesem a priori bekannten Verhalten der Knoten. Wird zu einem Zeitpunkt, an dem ein Paket erwartet wird, nichts empfangen, so muss davon ausgegangen werden, dass der Sender fehlerhaft arbeitet und das Netzwerk vor diesem Netzknoten geschützt werden muss.

Die örtliche Konzentration der Netzwerkkontrolle steht im Gegensatz zur Fehlertoleranz. Diese Tatsache betrifft vor allem Token-basierte Netzwerke. Hier muss ein zusätzlicher Token-failure Algorithmus dafür sorgen, dass bei einem Ausfall des Knotens, dem das Token zugeteilt war, das Token neu erzeugt wird. Dies widerspricht aber der Echtzeitanforderung, da ein verlorenes Token erst nach einer bestimmten Timeout-Periode erkannt werden kann. In einem zeitgesteuerten Netzwerk erlangt der nächste Netzknoten völlig unabhängig vom Ausfall eines Knotens die Schreibberechtigung.

Da Echtzeitsysteme häufig in sicherheitskritischen Bereichen eingesetzt werden, kann es not-

1. Bei dieser Netzstruktur wird ein Netzpaket, das sogenannte „Token“ erzeugt, das im Ring von Knoten zu Knoten weitergegeben wird. Nur der Knoten, der das Token gerade hat, darf senden; alle anderen Knoten empfangen zu dieser Zeit. Nachdem der Knoten den Sendevorgang beendet hat, gibt er das Token an den nächsten Knoten weiter.

wendig sein, eine aktive Redundanz zu implementieren. Dies bedeutet eine mehrfache Ausführung des gleichen Systems, so dass Ergebnisse mehrfach berechnet werden [1]. Es ist also eine Kopie des Systems aufzubauen, die zu jeder Zeit nach außen den gleichen Zustand repräsentiert wie das System selbst (Replica Determinismus). Wird ein Netzwerk mit einem auf Wahrscheinlichkeiten beruhenden Zugriff benutzt, so ist es nicht möglich, zu jeder Zeit einen konsistenten Zustand von System und Kopie zu erzeugen. Wird jedoch ein Netzwerk mit determiniertem zeitlichen Ablauf eingesetzt, so kann *Replica Determinismus* garantiert werden.

Diese Gegensätze führten zu unterschiedlichen Netzwerklösungen. Dabei sind die beiden Extrema in einer rein zeitgesteuerten oder einer rein ereignisgesteuerten Realisierung zu sehen. Zum Vergleich sind die in der Literatur vorgestellten Protokolle CAN, ARINC 629, PROFIBUS und TTP in Abbildung 3.1 dargestellt [10],[11],[26].

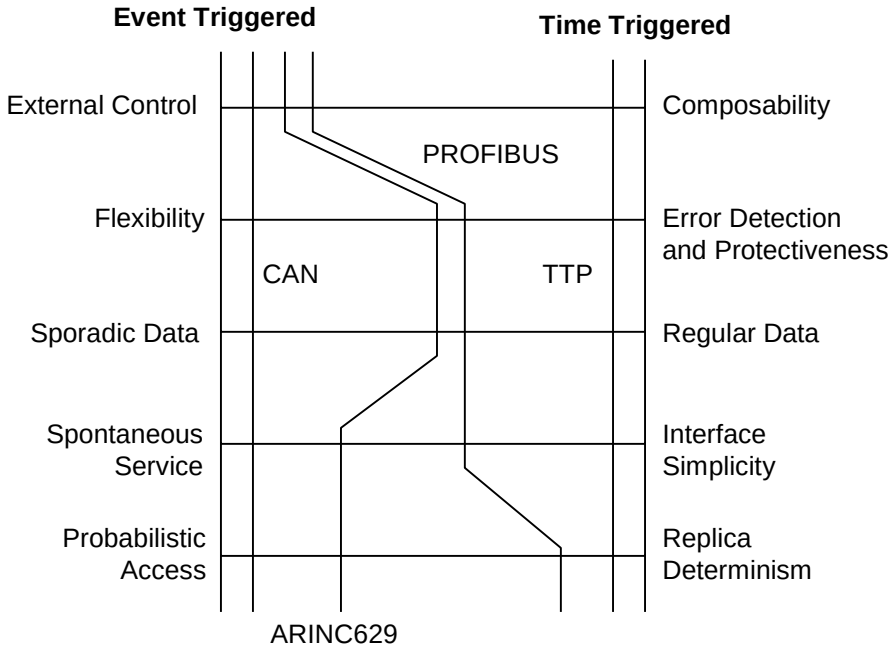



**Abbildung 3.1:** Vergleich der Protokoll Designs von CAN, ARINC629, PROFIBUS und TTP

### 3.1.2 Flusskontrolle

Um den korrekten Betrieb des Netzwerkes und des Datenverkehrs sicherzustellen, sollte eine Flusskontrolle (flow control) eingeführt werden. Es wird hier zwischen der expliziten und der impliziten Flusskontrolle unterschieden.

Bei der expliziten Flusskontrolle bestätigt der Empfänger eine Nachricht, indem er eine Acknowledgement-Meldung an den Sender zurückschickt. Das wichtigste Protokoll der expliziten Flusskontrolle ist das PAR-Protokoll (Positive Acknowledgement-or-Retransmission) [11]. Es handelt sich um ein ereignisgesteuertes System. Immer wenn ein Sender eine Nachricht schickt, startet er zunächst einen Zähler (retry counter). Empfängt der Sender innerhalb eines festgelegten Timeout-Intervalls die Bestätigung des Empfängers, so ist die Übertragung geglückt. Hat der Sender jedoch keine Bestätigung erhalten, so inkrementiert er den Zähler

und sendet die Nachricht erneut. Erreicht der Zähler seinen Höchststand, die maximal zulässige Anzahl an Wiederholungen, so bricht der Sender die Übertragung ab und meldet einen Misserfolg. Es sind viele Varianten dieses Grundkonzeptes bekannt, zusammenfassend können aber alle Varianten auf folgende vier Grundprinzipien zurückgeführt werden [11]:

(i). Der Sender beginnt die Übertragung

(ii). Der Empfänger kann den Sender über den bidirektionalen Kommunikationskanal anhalten

(iii). Ein Übertragungsfehler wird vom Sender und nicht vom Empfänger erkannt. Der Empfänger wird über einen Übertragungsfehler nicht informiert.

(iv). Zeitliche Redundanz wird zur Fehlerkorrektur benutzt (wiederholtes Senden), was eine Erhöhung der Latenzzeit im Fehlerfall zur Folge hat.

Bei der impliziten Flusskontrolle verständigen sich Sender und Empfänger a priori über die Übertragungszeitpunkte. Dies benötigt eine globale, einheitliche Zeitbasis. Der Sender schickt nur zu vereinbarten Zeitpunkten, während der Empfänger seinerseits zu den vereinbarten Zeitpunkten empfangsbereit ist und die Nachrichten entgegen nimmt. Das ermöglicht eine Fehlererkennung auf der Empfangsseite, wenn die Meldung nicht zum festgelegten Zeitpunkt eintrifft. Zur Flusskontrolle wird keinerlei Bestätigungsmeldung benötigt, weshalb eine Datenverbindung auch unidirektional ausgeführt werden kann. Zusätzliche Fehlertoleranz kann durch aktive Redundanz erreicht werden, ohne die Latenzzeit zu erhöhen. Der Sender kann mehrere Kopien einer Meldung im besten Fall auf unterschiedlichen Kanälen gleichzeitig versenden. Zum korrekten Empfang einer Meldung reicht es, wenn eine Kopie den Empfänger erreicht.

### 3.1.3 Zugriffsstrategien für Netzwerke

Die Art des Medienzugriffes (MAC -- media access control) entscheidet, ob es sich um ein ereignisgesteuertes (Event Triggered) oder ein zeitgesteuertes (Time Triggered) Netzwerk handelt. Der Unterschied zwischen beiden Verfahren soll zur Einführung an einem Beispiel erläutert werden.

Gegeben sei ein Alarmsystem, das aus 10 Knoten besteht, die ein Objekt überwachen (Abbildung 3.2). Jeder Knoten kann dabei 40 verschiedene binäre Alarmmeldungen erzeugen. Spätestens 100 ms nach dem Auftreten eines Alarms soll der Operator über den Alarmmonitor informiert werden. Die Bandbreite des Netzwerkes sei 100 kBit/s. Beide vorgestellten Lösungen benutzen als Basis-Protokoll CAN[1]. Jede CAN-Nachricht hat dabei einen Overhead von 44 Bit (vgl. Abbildung 3.3)

**Ereignisgesteuerte Lösung:**

Hier sendet ein Knoten eine Alarmmeldung sofort nach dessen Auftreten aus. Um die Nummer des Alarms anzuzeigen, reicht ein Datenfeld von 1 Byte. Zusammen mit einer Lücke von 4 Bit zwischen zwei Meldungen und dem Overhead von 44 Bit ergibt sich eine gesamte Meldungslänge von 56 Bit pro Alarm. Bezogen auf eine Bandbreite von

1. Die Arbeitsweise von CAN wird weiter unten in diesem Kapitel dargestellt. Sie ist aber für das Verständnis des Beispiels nicht wichtig.

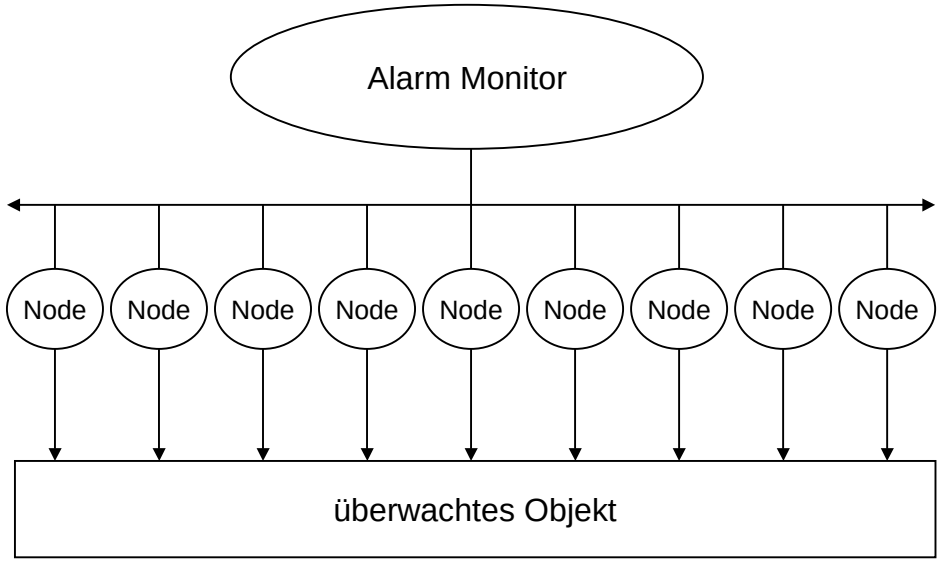



**Abbildung 3.2:** Beispiel eines Alarm Monitor Systems

| Feld | Arbitration | Control | Data Field | CRC | A | EOF |
|---|---|---|---|---|---|---|
| Bits | 11 | 6 | 0-64 | 16 | 2 | 7 |

**Abbildung 3.3:** Datenformat einer CAN Nachricht

100 kBit/s und die maximal erlaubte Verzögerung von 100 ms, können bis zu 180 Alarmmeldungen übertragen werden. Im Worst-Case-Szenario, bei dem jeder Knoten die maximale Anzahl Alarme verschickt, müssten jedoch 400 Meldungen übertragen werden.

**Zeitgesteuerte Lösung:**

Bei der zeitgesteuerten Lösung sendet ein Knoten periodisch alle 100 ms seinen Status. Dafür wird ein Datenfeld von 40 Bit (5 Byte) benötigt. Zusammen mit dem Overhead und einer Lücke von 4 Bit ergibt sich eine Gesamtlänge von 88 Bit je Nachricht. Bezogen auf die Bandbreite von 100 kBit/s und der Periode von 100 ms könnten 110 dieser Nachrichten versandt werden. Benötigt werden für 10 Knoten jedoch nur 10 Meldungen, was einer Belastung von weniger als 10% der gesamten Bandbreite entspricht. Zusätzlich ermöglicht die zeitgesteuerte Lösung eine Fehlererkennung aufgrund der periodischen Statusmeldungen; dies leistet die ereignisgesteuerte Lösung nicht.

Der Vergleich der erzeugten Netzbelastung durch das Alarmsystem wird in Abbildung 3.4 gezeigt. Der break-even-point wird bereits bei 16 Alarmen pro 100ms erreicht. Das entspricht 4% der Worst-Case-Alarmen.

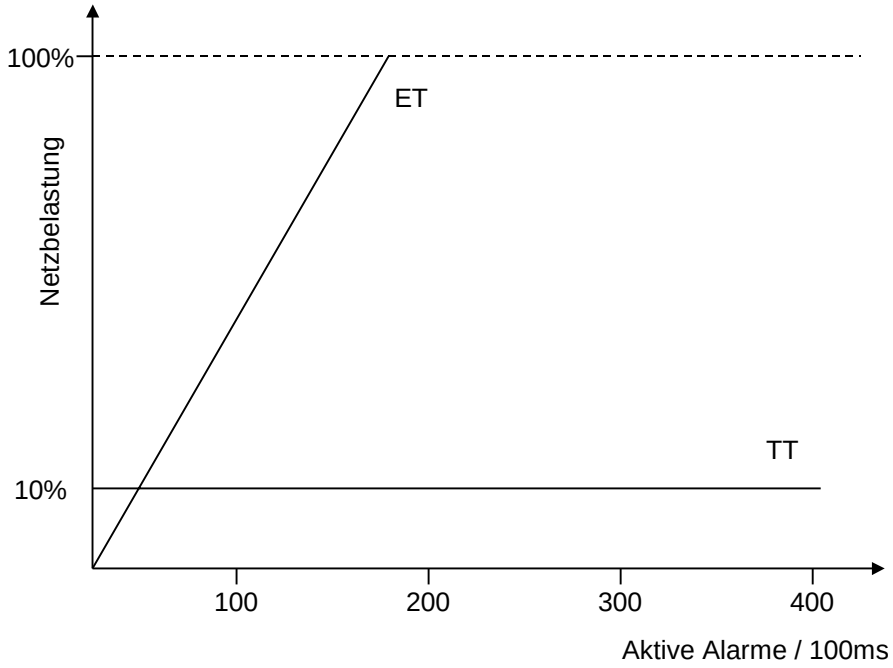



**Abbildung 3.4:** Vergleich der durch das Alarmsystem erzeugten Netzbelastung

Es handelt sich bei diesem Beispiel um ein typisches Problem, dass mit einem echtzeitfähigen Netzwerk gelöst wird. Hier erscheint die zeitgesteuerte Lösung als die weitaus sinnvollere. Kann jedoch auf die Echtzeitfähigkeit verzichtet werden so wird man sicher zu einer ereignisgesteuerten Lösung greifen, da so eine höhere Bandbreite zu erreichen ist. In dieser Arbeit wird mit Ethernet ein klassisches ereignisgesteuertes Netzwerk benutzt. Trotzdem soll ein zeitgesteuertes Protokoll implementiert werden. Um dem Leser einen Einblick in die Unterschiede bereits bestehender Systeme zu geben, die als Grundlage dienen, sollen im folgenden die wichtigsten Zugriffsstrategien vorgestellt werden. Als ereignisgesteuerte Verfahren werden CSMA/CD und CSMA/CA genannt, während Minislotting und TDMA als Vertreter mit einen zeitgesteuerten Zugriff vorgestellt werden.

Ein klassisches Beispiel für CSMA/CD (Collision Sense Multiple Access / Collision Detection) ist das Ethernet Protokoll. Ein Netzteilnehmer schreibt auf das Netzwerk. Erkennt er dabei das Auftreten einer Kollision, so beendet er den Schreibzugriff und versucht nach einer zufälligen Zeitspanne erneut zu senden. Es ist klar, dass eine solche Strategie unter harten Echtzeitbedingungen nicht zulässig ist, da nicht garantiert werden kann, dass die Datenpakete den Empfänger rechtzeitig erreichen.

Prominentester Vertreter von CSMA/CA (Collision Sense Multiple Access / Collision Avoidence) ist das CAN (Control Area Network) Protokoll. Hierbei ist jede Nachricht mit einer Priorität belegt, die durch ein Arbitration-Feld (11 Bit) zu Beginn jeder Meldung festgelegt wird. Die Bits dieses Feldes können entweder dominant (0) oder rezessiv (1) sein. Die Bits des Arbitration-Feldes werden nacheinander auf den Bus gelegt. Wird ein rezessives Bit durch ein dominantes Bit eines anderen Netzteilnehmers überschrieben, so wird die Übertragung gestoppt. Durch dieses prioritätenbasierte Zugriffsverfahren ergibt sich die Möglichkeit periodische Daten im Sinne des Rate Monotone Scheduling zu übertragen [16]. Bei diesem

Planungsverfahren ist die Frist für einen Prozess implizit durch seine Periode gegeben, d.h. die Ausführung des Prozesses muss vor der nächsten periodischen Startzeit beendet sein. Unter diesen Voraussetzungen lässt sich immer ein brauchbarer Plan finden, wenn für die Auslastung $\rho$ als hinreichende Bedingung gilt[1]:

$$\rho = \frac{\Delta e_1}{\Delta p_1} + \ldots + \frac{\Delta e_n}{\Delta p_n} \leq \rho_{max}$$

$$\rho_{max} = n(\sqrt[n]{2} - 1)$$

Für die Gleichung der maximalen Auslastung gilt weiter[2]:

$$\lim_{n \to \infty} \rho_{max} = \lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln(2) \approx 0,69$$

Das bedeutet aber, dass beliebig viele Netzteilnehmer periodische Daten senden dürfen, solange die Auslastung unter 69% liegt. Für diesen Fall ist CAN also unter allen Umständen als Netzwerk unter harten Echtzeitbedingungen einsetzbar [28]. Für ausgesuchte Beispiele lässt sich auch bei höherer Netzauslastung ein Plan finden, jedoch nicht in jedem Fall.

Minislotting ist ein Verfahren, dass bei ARINC629, einem Protokoll aus der Flugzeugindustrie [11], und Byteflight eingesetzt wird. Hierbei wird die Zeit in kleine Zeitscheiben (Minislots) unterteilt, die länger sein müssen als die Laufzeit einer Nachricht auf dem Netzwerk-Kanal. Die Zuteilung des Netzwerkzugriffes erfolgt ähnlich dem *Bakery Algorithm* [13] von Lamport. Jeder Netzknoten bekommt eine persönliche, eindeutige Anzahl von Minislots zugewiesen, die zusammen seine persönliche Zeitperiode $TG_i$ bilden. Möchten zwei Prozesse gleichzeitig senden, so warten beide zunächst eine Synchronisationsperiode der Länge $SG$ ab, in der niemand auf dem Netzwerk sendet. Ist dieser Zeitabschnitt vergangen, so treten die Prozesse in einen gedachten Warteraum ein. Hier müssen sie warten bis ihre persönlichen Zeitperiode $TG_i$ abgelaufen ist. Erst dann hat ein Prozess die Erlaubnis zu senden. Falls nicht schon ein anderer Knoten vor ihm eine Nachricht auf den Kanal gelegt hat, beginnt er die Übertragung. Der Prozess mit der kleinsten Zeitspanne fängt demnach zuerst an zu senden. Nachdem er seine Nachricht übertragen hat, setzen alle anderen Prozesse im Warteraum

1. $\Delta e$ ist dabei die Ausführungszeit eines Prozesses, $\Delta p$ seine Periode
2. Herleitung:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}(2^{\varepsilon} - 1) \qquad ,\varepsilon = \frac{1}{n}$$

$$= \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}(e^{\varepsilon \ln(2)} - 1) = \lim_{\varepsilon \to 0} \frac{1}{\varepsilon}(1 + \varepsilon \ln(2) + O(\varepsilon^2) - 1) \qquad ,e^{\varepsilon \ln(2)} = 1 + \varepsilon \ln(2) + O(\varepsilon^2)(\text{Taylor})$$

$$= \lim_{\varepsilon \to 0} \Big( \ln(2) + \underbrace{\frac{1}{\varepsilon} O(\varepsilon^2)}_{\to 0} \Big) = \ln(2)$$

ihren Zähler wieder auf $TG_i$ und fangen nach dessen Ablauf ihrerseits an zu senden, falls ihnen nicht wieder ein anderer Knoten zuvorkommt. Mit dem Beginn eines Sendevorgangs startet jeder Knoten ein weiteren Zähler, vor dessen Ablauf er nicht wieder in den Warteraum eintreten darf, um die nächste Nachricht zu versenden. Mit diesem Verfahren wird also ein zeitgesteuerter Zugriff nach Prioritäten (bestimmt durch $TG_i$) realisiert. Eine globale Zeit jedoch nicht notwendig. Indem jedoch der Wiedereintritt in den Warteraum verzögert ist, wird verhindert, dass ein Knoten trotz höchster Priorität das Netzwerk dominiert.

Beim TDMA Verfahren (Time Division Multiple Access) wird die gesamte Zeit in sogenante Zeitscheiben unterteilt. Jeder Netzknoten erhält bestimmte Zeitscheiben zugewiesen, in denen er senden darf. Dabei ist es zwingend notwendig, dass alle Netzknoten mit derselben Zeitbasis (global time) arbeiten, da der Netzzugriff nur durch das Voranschreiten der Zeit geregelt wird. Je nach Protokoll wiederholt sich der Netzzugriff für jeden einzelnen Knoten periodisch. Ein Beispiel für ein Echtzeitnetzwerk mit diesem Zuteilungsverfahren ist das Time Triggered Protocol (TTP) [11], [26].

## 3.2 Echtzeitbetriebssysteme

In diesem Kapitel wird zunächst ein Überblick über die existierenden Echtzeitbetriebssysteme gegeben. Das ermöglicht eine Einordnung des in dieser Arbeit verwendeten Betriebssystem RT Linux. Anschließend wird dessen Aufbau und Programmiermodell beschrieben. Dieses Wissen soll dem Leser das Verständnis der in Kapitel 4 vorgestellten Implementation des Treibers erleichtern.

### 3.2.1 POSIX.4

Auf dem Sektor der Echtzeitbetriebssysteme existiert eine große Anzahl unterschiedlicher Systeme. Je nach Anwendung und eingesetzter Plattform werden unterschiedliche Formen von Betriebssystemen verwendet. Um diesem Trend entgegen zu wirken, hat das IEEE (Institute of Electronical and Electronics Engineers) den Standard POSIX.4 (offziell POSIX-Standard 1003.1b) veröffentlicht. Es handelt sich bei POSIX (Portable System Interface for Computer environments) um einen Standard für Quellcode. POSIX konforme Programme können somit für beliebige POSIX konforme Betriebssysteme kompiliert werden. Der gesamte Standard ist in viele Themengebiete unterteilt, von denen POSIX.4 die Echtzeitfähigkeit von Anwendungen umfasst. Folgende Klassen von Funktionen zur Verwaltung von Echtzeitpunkten sind enthalten [1]:

- Synchronisierung mit Semaphoren
- Vergabe von Prioritäten an Prozesse
- Zuordnung gemeinsamer Speicherbereiche an mehrere Prozesse
- Nachrichtenübertragung zwischen Prozessen
- Signalisierung asynchroner Ereignisse
- Zuordnung von nicht verdrängbaren Hauptspeicherbereichen an Prozesse
- Synchrone und asynchrone Ein-/Ausgabe

In der Vergangenheit beschränkte sich der Einsatz von Echtzeitanwendungen auf das Gebiet der eingebetteten Systeme (Embedded Systems). Hiermit wurden bisher Mikrokontrollerlösungen bezeichnet, die speziell für eine Anwendung entwickelt wurden. In letzter Zeit geht jedoch der Trend aufgrund immer günstiger werdender Preise durch den Massenmarktes dahin, auch im diesem Bereich Standard-Hardware einzusetzen. Zu nennen sind hier vorallem PC-basierte Systeme, die teilweise bereits auf die Größe eines DIMM-Speichermoduls geschrumpft sind, wie sie z.B. von der Firma Jumptec angeboten werden. Die meisten dieser Systeme besitzen einen Prozessor mit beschränkter Leistungsfähigkeit und eine einfach strukturierte Peripherie. Aus diesem Grund können sie sehr klein und kostengünstig ausgeführt werden. Auf diesen Systemen werden rudimentäre Echtzeitsysteme eingesetzt, die als Laufzeitumgebung in das Echtzeitprogramm einkompiliert werden [1]. Die Programmentwicklung erfolgt auf einer Cross-Entwicklungsumgebung auf einer Standard-Plattform. Programme können hier mit entsprechenden Tools geschrieben, kompiliert und simuliert werden. Erst nach erfolgreichem Test in dieser Umgebung wird die Anwendung auf das Zielsystem geladen und eingesetzt. Ein Vertreter dieser Gruppe ist das rudimentäre Betriebssystem VRTX, es finden sich aber speziell in diesem Bereich noch sehr viele proprietäre Lösungen.

Die zweite Gruppe, nach den rudimentären Echtzeitsystemen, bilden die speziellen Echtzeitbetriebssysteme, die gezielt für Echtzeitanwendungen entwickelt wurden. Diese nutzen jedoch Standard-Plattformen, wie z.B. PCs, und bieten eine Hardware-Abstraktions-Schicht (HAL). Der Kern dieser Betriebssyteme integriert häufig Eigenschaften, die bei Standard-Betriebssystemen weniger zu finden sind. Ein Beispiel ist eine Netzwerkschnittstelle, die eine Interprozesskommunikation auch zwischen mehreren Netzknoten ermöglicht. Gleichzeitig beschränkt sich der Kern aber auf die wesentlichen Elemente und ist daher oft sehr klein (Mikrokernel-Prinzip [1]). Auf diese Weise können Kontextwechsel sehr effizient ausgeführt werden. Ein Vertreter dieser Gruppe ist das kommerzielle Betriebssystem QNX.

Als dritte Gruppe sind die Echtzeiterweiterungen zu Standard-Betriebssystemen zu nennen. Hierbei werden Systeme wie Linux oder WindowsNT mit Echtzeiteigenschaften ausgestattet oder so umstrukturiert, dass Echtzeitanforderungen erfüllt werden können. Gleichzeitig können aber immer noch die Vorteile der zugrundeliegenden Betriebssysteme, wie die grafische Benutzeroberfläche oder die ausgereiften Bibliotheken, genutzt werden. In diese Gruppe fällt auch das in dieser Arbeit verwendete RT Linux, neben anderen wie z.B. LynxOS und Windows CE. Diese Systeme werden in zunehmenden Maße auch im Bereich der eingebetteten Systeme eingesetzt, wenn dort PC-Hardware benutzt wird. Entwicklungen können so auch im Software-Bereich kostengünstiger ausfallen, da Programmierer auf Standard-Betriebssystemen oftmals besser geschult sind. Desweiteren ist es für ein Neueinstellung einfacher bei der derzeitig angespannten Personalsituation solche Software-Entwickler zu finden.

### 3.2.2 Aufbau und Leistungsfähigkeit von RT Linux

RT Linux ist eine Echtzeiterweiterung zu Linux, dessen Quelldateien mit der Installation verändert werden. Ein neuer Kern muss daher kompiliert und installiert werden. Nach dem Bootvorgang ist das Echtzeitsystem noch nicht aktiv. Das geschieht durch Hinzuladen der speziellen Module

- rtl_time.o
- rtl_sched.o

- rtl_posixio.o
- rtl_fifo.o

zum Betriebssystemkern. Der Start erfolgt also wie das nachträgliche Einbinden eines Treibers. Mit diesen Modulen wird ein eigenständiger, echtzeitfähiger Betriebssystemkern installiert, dessen Aufgabe ist die Verwaltung der Echtzeit-Threads der RT Applikationen ist. Der eigentliche Linux-Kern ist aus Sicht des RT Kerns ebenfalls nur ein Thread, allerdings mit der niedrigsten Priorität im System (Idle-Thread) [30]. Nach dem Start des Echtzeitsystems hat der RT Kern durch das Abfangen der Interrupts die volle Kontrolle über die Hardware, insbesondere die CPU, übernommen. Jeder Interrupt wird durch eine Service-Routine des RT Kerns bedient. Falls der Interrupt nicht von einer Echtzeitapplikation genutzt wird, erfolgt eine Weitergabe an den Linux-Kern. Dies geschieht aber erst, wenn keine Rechenzeit für das RT System benötigt wird. Für die Zuteilung der CPU durch den Scheduler stehen verschiedene Verfahren, wie z.B. Priortitäten, Earliest Deadline First, usw., zur Verfügung. Zur Kommunikation mit dem Anwendungsbereich (User-Space) und dem Linux-Kern, können spezielle RT-FIFOs genutzt werden. Abbildung 3.5 fasst den Aufbau von RT Linux zusammen.

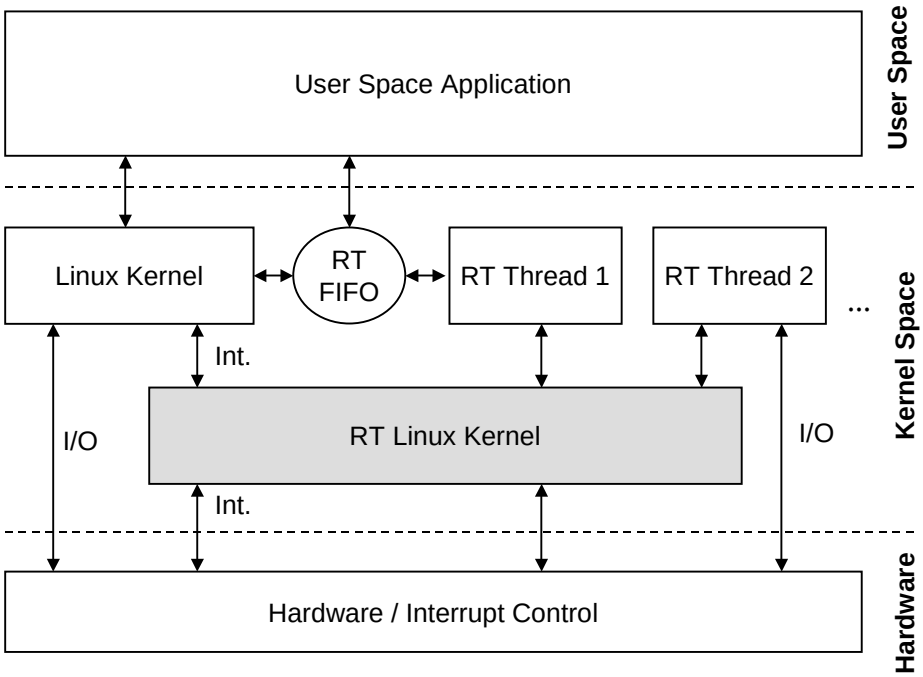



**Abbildung 3.5:** Der Aufbau von RT Linux mit Kommunikationswegen

Durch diese Struktur des Betriebssystems arbeitet jedes Echtzeitprogramm im Kern-Speicherbereich. Daraus ergibt sich der Vorteil, dass ein Datenaustausch über gemeinsame Speicherbereiche (shared memory) sehr einfach möglich ist. Ebenso sind Symbole, die als *extern* deklariert sind, jedem Kernmodul bekannt und können so von jedem Treiber aufgerufen werden. Von dieser Möglichkeit wird bei der Kommunikation von Treibern untereinander in der vorliegenden Arbeit intensiv Gebrauch gemacht. Ein weiterer Vorteil liegt im vollen Zugriff auf die Hardware, um effiziente Echtzeitanwendung entwickeln zu können. Als ein Nachteil ist jedoch die beschränkte Fähigkeit Systemaufrufe auszuführen, anzusehen. Es ist

beispielsweise nicht möglich eine Datei direkt zu öffnen. Desweiteren kann keine dynamische Speicherzuweisung im Echtzeitbereich benutzt werden, was sich im Laufe der Arbeit als eine Schwierigkeit herausgestellt hat.

Insgesamt gesehen stellt RT Linux aber eine ideale Entwicklungsplattform für die hier realisierte Anwendung dar, da Treiber und Echtzeitsystem sehr einfach durch den gemeinsamen Speicherbereich kombinierbar sind. Dies wird in der Beschreibung der Realisierung in Kapitel 4 deutlich.

Bevor mit einer Entwicklung begonnen werden kann, ist es erforderlich, eine Abschätzung der Leistungsfähigkeit der eingesetzten Plattform durchzuführen. Als ein Maß hierfür kann die Interruptlatenzzeit herangezogen werden. Das ist die Zeit, die zwischen dem Auslösen eines Interrupts und der Abarbeitung des ersten Befehls des zuständigen Interrupthandlers vergeht.

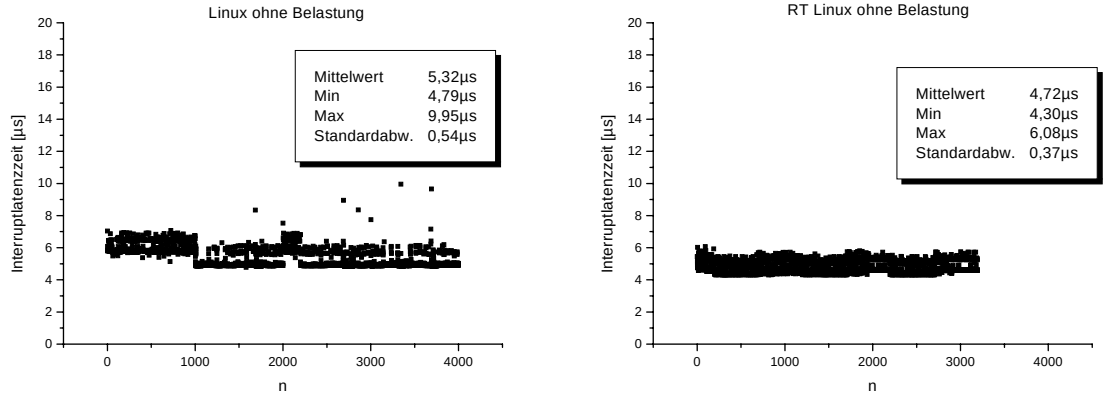



**Abbildung 3.6:** Vergleich der Interruptlatenzzeit bei Linux und RT Linux ohne Belastung

Während in Abbildung 3.6 die Interruptlatenzzeit von Linux und RT Linux ohne Belastung verglichen wird, sollen in Abbildung 3.7 die Unterschiede in der Latenzzeit mit Belastung aufgezeigt werden. Als Last diente bei den Messungen das Schreiben und Lesen einer 20 MByte großen Datei auf die lokale Festplatte des Rechners. Sehr deutlich sind die Unterschiede besonders in den Min/Max-Werten zu erkennen.

RT Linux wurde in der Version 2.2 eingesetzt. Gegenüber der Version 1 wurde hier die POSIX-Konformität bezüglich der Thread-Schnittstelle eingeführt. Zusätzlich enthält diese Version die Fähigkeit, auch Multiprozessorplattformen (SMP) bedienen zu können. Gleichzeitig zum Verlauf der Arbeit haben die Entwickler von RT Linux die Version 3 veröffentlicht. Hier konnten einige Fehler der früheren Versionen beseitigt werden und es wurde eine Debug-Möglichkeit geschaffen. Dies ist eine sehr nützliche Eigenschaft, da sich die Fehlersuche auf der Treiberebene sehr schwierig gestaltet. Bildschirmausgaben sind nur auf das Syslog-Fenster möglich, was unter Linux ein Umschalten auf eine spezielle Konsole (ALT-F10) für Kernel-Ausgaben erfordert. Ein Anhalten von Programmen zur Kontrolle eines Variableninhalts war im Kern bisher überhaupt nicht möglich. Aufbauende Entwicklungen sollten neuere Versionen von RT Linux benutzen.

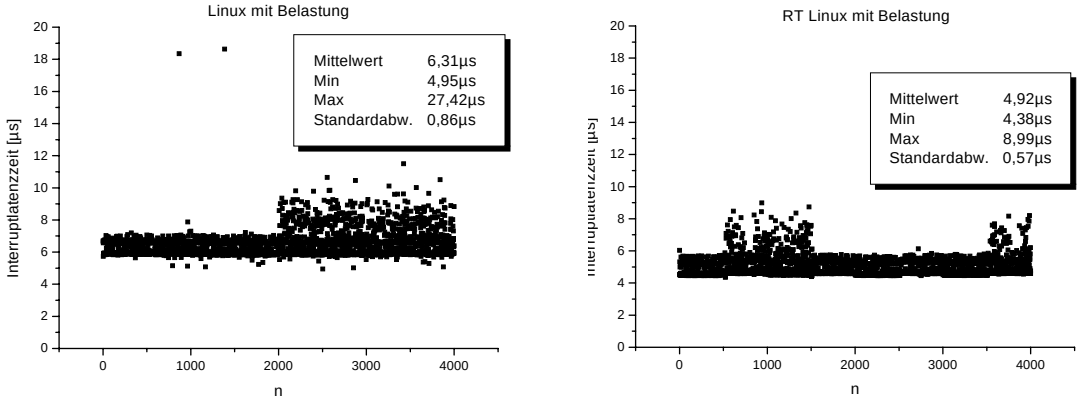



**Abbildung 3.7:** Vergleich der Interruptlatenzzeit bei Linux und RT Linux mit Belastung

### 3.2.3 Das RT Linux Programmiermodell

Leider hält sich RT Linux nur bedingt an die POSIX.4 Schnittstelle. Es ist jedoch Ziel in der Entwicklung des Betriebssystems diese Konformität weitestgehend zu erfüllen [29]. Für das Thread-Handling wird die PThread-Schnittstelle (Posix-Thread) genutzt. Somit lautet die Funktion zur Erzeugung eines Echtzeit-Threads:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void* (*fn)(void*), void* arg);
```

Die Funktion startet den Thread, indem die Funktion `fn` ausführt wird. Durch `attr` können verschiedene Parameter, wie z.B. die Priorität, für den Thread gesetzt werden. Viele dieser Threads müssen periodisch ausgeführt werden. Deshalb stellt RT Linux spezielle Funktionen zur Verfügung, die periodische Threads behandeln. Diese periodischen Threads sind im POSIX.4 Standard nicht vorgesehen. Deshalb haben die entsprechenden Funktionsbezeichnungen ein `_np` für *non portable* als Endung, so wie alle Funktionen die nicht Posix-konform sind. Durch

```
int pthread_make_periodic_np(pthread_t thread,
                   hrtime_t start_time,hrtime_t period);
```

wird ein Thread auf *periodisch* geschaltet. Der Thread startet nach Ablauf der Startzeit `start_time` die Ausführung mit der Periodenlänge `period`. Beide Argumente werden in Nanosekunden angegeben. Der aktuelle Zeitpunkt in Nanosekunden kann durch Aufruf der Funktion `gethrttime()` bestimmt werden. Die wirkliche Granularität dieses Timers ist dabei hardwareabhängig. Die Periodenlänge kann nachträglich noch verändert werden, indem der Wert des Feldes `period` in der Struktur `p_thread` des entsprechenden Threads verändert wird. Dabei wird aber vorausgesetzt, dass der Thread bereits durch `pthread_make_periodic_np` als periodisch definiert wurde.

Nach dem Ende der Ausführungszeit eines Threads, muss er die CPU wieder freigeben. Er blockiert dann bis die nächste Periode beginnt. Dies wird durch Aufruf folgender Funktion

erreicht:

```
int pthread_wait_np(void);
```

Ein periodischer Thread wird also praktisch in einer Endlosschleife implementiert, an deren Ende jeweils der Aufruf von `pthread_wait_np` steht. Wird der Thread nicht mehr benötigt, kann er durch Aufruf der Funktion

```
int pthread_delete_np(pthread_t thread);
```

gelöscht werden. Soll innerhalb eines Threads eine Gleitkommaberechnung ausgeführt werden, so muss dies dem Scheduler vorher durch Aufruf der Funktion

```
int pthread_setfp_np(pthread_t thread, int flag);
```

mitgeteilt werden. Das hat zur Folge, dass bei einem Kontextwechsel, der einen solchen Thread betrifft, die entsprechenden Fliesskomma-Register mitgesichert werden. Standardmässig geschieht das aus Effizienzgründen nicht.

Sehr wichtig für den Austausch von Daten zwischen dem Anwendungsbereich und einem RT Thread sind die Echtzeit-FIFOs (`/dev/rtf0..`/`dev/rtf`*n*). Von der Applikation aus gesehen werden sie als zeichenorientierte Geräte im Verzeichnisbaum behandelt. Von der RT Linux Seite aus werden sie durch spezielle Funktionen angesprochen. Mit

```
int rtf_create(unsigned int fifo, int size);
```

wird ein FIFO-Kanal mit der Nummer `fifo` und der Größe `size` initialisiert. Aus dem Anwendungsbereich können nur die Kanäle geöffnet werden, die zuvor mit `rtf_create` initialisiert wurden. Die Freigabe des Kanals erfolgt durch Aufruf folgender Funktion:

```
int rtf_destroy(unsigned int fifo);
```

Die FIFOs können von der RT Linux Seite mit Hilfe der Funktionen

```
int rtf_put(unsigned int fifo, char* buf, int count);
int rtf_get(unsigned int fifo, char* buf, int count);
```

beschrieben und ausgelesen werden. Im Anwendungsprogramm werden die regulären Operationen für die Dateibehandlung *open, write, read* und *close* benutzt.

Eine typische RT Linux Anwendung, die einen periodischen Thread erzeugt, ist in Abbildung 3.8 dargestellt. Sehr deutlich ist im Beispiel der Modulaufbau des Programms durch die Funktionen `init_module()` und `cleanup_module()` zu erkennen. Das Programm erzeugt einen Thread mit der Periode 500 µs der nach einer Zeit von 3 ms startet. Wenn das Modul aus dem Kern entfernt wird, so wird auch der Thread gelöscht.

## 3.3 Netzwerkkartentreiber unter Linux

Im Linux-Kernel bilden Netzwerktreiber neben blockorientierten und zeichenorientierten Treibern eine eigenständige Gruppe. Sie besitzen keine Repräsentation im Dateibaum (/dev-

```
#include <linux/module.h>
#include <linux/kernel.h>

#include <rtl_sched.h>

pthread_t mythread;

void* thread_func(void* arg)
{
    while(1)
    {
        // do something
        pthread_wait_np();
    }

    return 0;
}
int init_module(void)
{
    hrtime_t now = gethrtime(); // akt. Zeit in Nanosekunden

    // Erzeugen eines Threads
    pthread_create(&mythread, NULL, thread_func, (void*)0);
    // Umaschalten auf Periode der Länge 500µs
    pthread_make_periodic_np(mythread, now + 3000000, 500000);

    return 0;
}

void cleanup_module(void)
{
    pthread_delete_np(mythread);
}
```

**Abbildung 3.8:** Beispiel einer RT Linux Anwendung

Verzeichnis) und werden paketorientiert angesprochen. Der wichtigste Unterschied zwischen einem Netzwerktreiber und z.B. einem Festplattentreiber ist aber, dass eine Festplatte aufgefordert wird, einen Buffer in den Kernel zu schicken, während das Netzwerkgerät selbst darum bittet, ein eingehendes Paket an den Kernel übergeben zu dürfen [23].

Das Netzwerk-Subsystem des Linux-Kernels ist vollständig protokollunabhängig entworfen worden. Das gilt sowohl für Netzwerkprotokolle, wie IP oder IPX, als auch für Hardware-Protokolle, wie Ethernet oder Token Ring. Der Austausch zwischen dem Netzwerktreiber und dem eigentlichen Kernel geschieht paketweise. Damit können Protokollinterna vor dem

Treiber und die physikalische Übertragung vor dem Protokoll verborgen bleiben [23]. Dieser Aufbau entspricht dem ISO/OSI-Referenzmodell.

Wird ein Netzwerkkartentreiber zum Kernel hinzugeladen, so fordert er Ressourcen an. Gleichzeitig bietet er Fähigkeiten an, die der Kernel von nun an nutzen kann. Die Referenzierung der Zugriffsfunktionen des Netzwerkkartentreibers erfolgt über die `device`-Struktur. Sie wird beim Start des Moduls mit gültigen Werten gefüllt. Dabei wird der Netzwerkkarte ein eindeutiger Name `eth0...n` vom Betriebssystem zugewiesen. In der `device`-Struktur sind die Ressourcen, wie IRQ und Basisadresse, der Netzwerkkarte abgelegt. Sie enthält aber auch Zeiger auf die Zugriffsfunktionen des Treibers, sowie einen Zeiger auf eine lokale Struktur des Treibers mit privaten Feldern. Anhang C zeigt die Definition dieser Struktur.

Soll ein Paket versandt werden, so ruft der Treiber die Funktion

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

auf, die im Treiber implementiert ist. Das zu versendende Paket ist dabei in dem Socket-Buffer `skb` enthalten. Diese Struktur wird weiter unten in diesem Kapitel eingeführt. Innerhalb der Funktion wird das Paket an die Netzwerkkarte übertragen. Das geschieht in der Regel durch DMA-Transfer. Bevor das Paket jedoch an `hard_start_xmit` übertragen wird, muss zunächst noch ein Header dem Paket hinzugefügt werden. Das geschieht durch die Funktion

```
int (*hard_header) (struct sk_buff *skb,
             struct device *dev,
             unsigned short type,
             void *daddr,
             void *saddr,
             unsigned len);
```

aus der `device`-Struktur. Sie wird jedoch nicht im Treiber implementiert, sondern ist Bestandteil von Linux. Sie bezieht sich auf die Art des eingesetzten Mediums, in unserem Fall also Ethernet. Mit dem Aufruf dieser Funktion werden Sender und Empfänger (`daddr, saddr`), sowie das Datenformat (`type`) festgelegt. Nachdem ein Paket versandt wurde, wird der Speicher des entsprechenden Socket-Buffers wieder freigegeben.

Wird ein Paket von der Netzwerkkarte empfangen, so speichert der Treiber die Nachricht ebenfalls in einem Socket-Buffer. Anschließend muss das Betriebssystem über den Empfang des Paketes informiert werden. Das geschieht im Treiber durch Aufruf der Funktion

```
void netif_rx(struct sk_buff *skb);
```

Das Paket wird in dieser Funktion an höhere Protokollschichten (z.B. IP) weitergegeben.

Im folgenden soll auf die Verwendung der Socket-Buffer eingegangen werden. Die vollständige Definition von `struct sk_buff` ist in Anhang D wiedergegeben. Um Speicher für einen neuen Socket-Buffer zu reservieren, wird die Funktion

```
struct sk_buff *dev_alloc_skb(unsigned int len);
```

benutzt. Sie alloziert Speicher für einen Socket Buffer dessen `skb->data`-Feld die Länge `skb->len` besitzt. Gleichzeitig werden die Felder, `skb->head`, `skb->tail` und `skb->data` initialisiert. Die Allozierung geschieht mit der Priorität GFP_ATOMIC[1] und es werden zusätzlich 16 Bytes zwischen `skb->head` und `skb->data` reserviert. Dieser Speicherbereich wird für den Hardware-Header benutzt, falls dieser benötigt wird. Bevor ein Socket-Buffer für eine Ethernet-Schnittstelle mit Daten gefüllt werden kann, muss er noch 2 Byte zusätzlich reservieren. Dabei werden `data` und `tail` inkrementiert. Dadurch wird erreicht, dass der IP-Header nach dem 14 Byte großen Ethernet-Header an einer 16 Byte Grenze steht [23]. Der Prototyp der entsprechenden Funktion, hat folgendes Format:

```
void skb_reserve(struct sk_buff *skb, int len);
```

In unserem Fall muss `len` also mit 2 belegt werden. Um Daten in den Socket Buffer zu kopieren, wird eine Kombination der Funktionen `memcpy` und

```
unsigned char *skb_put(struct sk_buff *skb, int len);
```

benutzt. `skb_put` aktualisiert den Zeiger auf das Datenende und gibt einen Zeiger auf den neu erzeugten Datenbereich zurück [23]. Somit sieht ein typischer Aufruf für das Kopieren in einen Socket Buffer wie folgt aus:

```
memcpy(skb_put(skb,len),&buff,len);
```

Wird ein Socket Buffer nicht mehr benötigt, so kann der allozierte Speicher mit Hilfe der Funktion

```
void dev_kfree_skb(struct sk_buff *skb);
```

wieder freigegeben werden.

Jedes Ethernet-Paket entspricht einem normierten Datenformat. Es handelt sich hierbei um das IEEE 802.3 MAC Frame Format, wie in Abbildung 3.9 dargestellt. Danach besitzt jedes Ethernet-Paket einen 14 Byte langen Header gefolgt von dem eigentlichen Datenpaket. Dieses wird auf eine 16 Bit Grenze normiert. Anschließend wird die Checksumme berechnet und als 4 Byte Feld angehängt.

## 3.4 Zeitsteuerung (Time Triggered Technology)

Wie in Abschnitt 3.1 bereits gezeigt werden konnte, wird zur Realisierung eines zeitgesteuerten, verteilten Systems eine globale Zeitbasis benötigt. Dazu müssen die lokalen Uhren der einzelnen Systemknoten auf diese globale Zeitbasis synchronisiert werden. Diese Synchronisierung ist die wesentliche Aufgabe, die eine Netzwerkschnittstelle, wie sie in dieser Arbeit vorgestellt wird, leisten muss.

1. Wird ein Stück Speicher mit der Priorität GFP_ATOMIC angefordert, so wird der aufrufende Prozess unter keinen Umständen blockiert. Bei einer Allocierung mit der Priorität GFP_KERNEL ist das nicht garantiert.

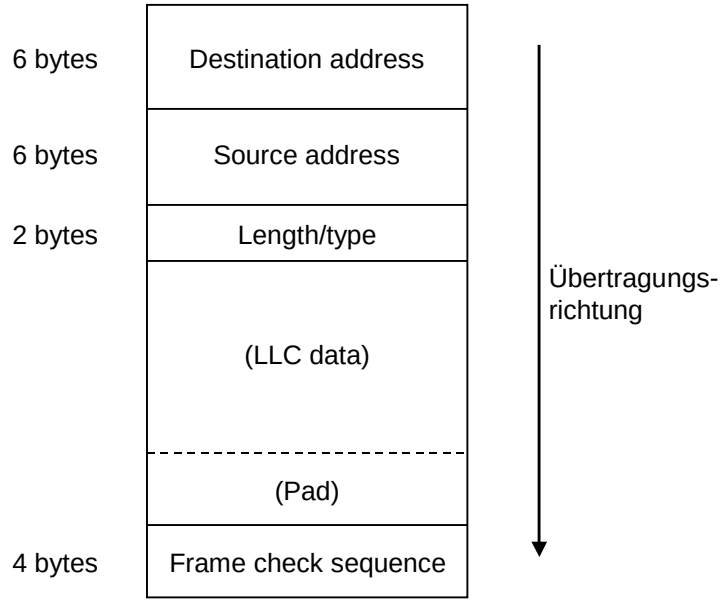



**Abbildung 3.9:** IEEE 802.3 MAC Frame Format [31]

### 3.4.1 Grundbegriffe der Zeitmessung

Die temporale Reihenfolge von Ereignissen beruht auf der Definition der Zeitlinie. Das Kontinuum der wirklichen Zeit (Echtzeit, real time) kann durch eine gerichtete Zeitschiene modelliert werden, die aus einer unendlich großen Menge {T} von Zeitpunkten besteht, die die folgenden Eigenschaften erfüllen:

(i). {T} ist eine geordnete Menge in dem Sinne, dass wenn zwei Elemente p und q dieser Menge gegeben sind, sich die Relationen *p und q gleichzeitig, p liegt vor q* und *q liegt vor p* gegenseitig ausschliessen. Diese Ordnung wird die temporale Reihenfolge genannt.

(ii). {T} ist eine dichte Menge. Das heisst, es lässt sich immer mindestens ein q finden, dass zwischen p und r liegt wenn mit p und r nicht derselbe Zeitpunkt bezeichnet ist. p, q und r sind Zeitpunkte und Elemente der Menge.

Ein Abschnitt dieser Zeitschiene wird Dauer genannt. Ein Ereignis tritt genau zu einem Zeitpunkt auf, ihm kann keine Dauer zugeordnet werden. Treten zwei Ereignisse zum selben Zeitpunkt auf, so geschehen diese Ereignisse simultan. Während Zeitpunkte einer einzigen, absoluten Ordnung auf der Zeitschiene unterliegen, so ist die temporale Ordnung von Ereignissen nicht die einzige. Simultane Ereignisse können also nach einem anderen Kriterium sehr wohl einer Ordnung unterliegen.

Eine physikalische Uhr ist ein Gerät zur Zeitmessung. Sie besteht aus einem Oszillator, der auf einem physikalischen Prinzip beruht und periodisch Ereignisse erzeugt. Ein Zähler wird mit jedem Ereignis inkrementiert und liefert so ein Maß für die aktuelle Zeit. Das periodische

Ereignis wird *Mikrotick (microtick)* genannt. Die Dauer zwischen zwei aufeinander folgenden Mikroticks ist *Granularität (granularity)* einer Uhr. Die Granularität führt bei jeder digitalen Uhr zu einem Digitalisierungsfehler. Im folgenden sollen Mikroticks mit einem Index zur Kennzeichnung verschiedener Uhren und mit einem zweiten Index zur fortlaufenden Nummerierung gekennzeichnet werden. Somit wird Mikrotick i von Uhr k mit $\mathrm{microtick}_i^k$ bezeichnet.

Es sei weiterhin eine einheitliche Referenzuhr z für alle Uhren im System gegeben. Diese Referenzuhr stimmt mit dem internationalen Zeitstandard absolut überein. Die Frequenz des Oszillators dieser Uhr ist mit $f^z \approx 10^{15}\mathrm{Hz}$ so hoch, dass die daraus resultierende Granularität dieser Uhr von ca. 1 Femtosekunde vernachlässigt werden kann. Damit kann der absolute Zeitpunkt des Auftretens eines Ereignisses e mit $z(e)$ gekennzeichnet werden. Eine Dauer kann durch ein Anzahl von Mikroticks der Referenzuhr z angegeben werden. Die Granularität einer Uhr k kann durch die Anzahl der Referenzticks zwischen zwei Mikroticks mit $n^k$ angegeben werden.

Der Drift einer pysikalischen Uhr k zwischen Mikrotick i und i+1 ist gegeben durch das Frequenzverhältnis zwischen dieser Uhr und der Referenzuhr z beim Auftreten des Mikroticks i. Der Drift kann durch Messung der Zeitspanne zwischen zwei Ticks der Uhr k mit der Referenzuhr z und anschließendem Dividieren durch die nominale Granularität $n^k$ bestimmt werden:

$$\mathrm{drift}_i^k = \frac{z(\mathrm{microtick}_{i+1}^k) - z(\mathrm{microtick}_i^k)}{n^k} \tag{3.1}$$

Dies ist ein Wert sehr nahe 1. Daher wird der Begriff der Driftrate $\rho_i^k$ eingeführt, die wie folgt definiert ist:

$$\rho_i^k = \left| \frac{z(\mathrm{microtick}_{i+1}^k) - z(\mathrm{microtick}_i^k)}{n^k} - 1 \right| \tag{3.2}$$

Eine perfekte Uhr hat demnach eine Driftrate von 0. Reale Uhren haben Driftraten, die sich mit der Zeit durch Umwelteinflüsse (Temperatur, Spannung, Alterung) verändern. Die Veränderung der Driftrate sollte aber durch die im Datenblatt angegebene maximale Driftrate $\rho_{max}^k$ begrenzt sein. Typische Driftraten liegen im Bereich von $10^{-2}$ bis $10^{-7}$. Das heisst aber, dass Uhren niemals völlig synchron laufen können und somit ständig nachsynchronisiert werden müssen, auch im normalen Betrieb, wenn kein Fehler vorliegt.

Bezogen auf den hier diskutierten Treiber, mit dem ein zeitgesteuertes Netzwerk realisiert wird, bedeutet das aber, dass die Zeit, die für die Sende- und Empfangszeitpunkte verantwortlich ist, ständig auf den Netzwerkverkehr synchronisiert werden muss. Die Zeitbasis des verwendeten Betriebssystems wird aus dem Prozessortakt gebildet. Damit ist sie der gleichen Driftrate unterlegen.

Angenommen ein Oszillator, der den Prozessortakt zur Verfügung stellt, habe eine Driftrate von $10^{-6}$, was ein realistischer Wert ist. Bezogen auf eine Taktfrequenz von 400MHz bedeu-

tet dies einen Frequenzunterschied von 400Hz. Das heisst der Oszillator erzeugt in jeder Sekunde 400 Schwingungen mehr oder weniger. Das bedeutet aber einen maximalen Fehler von 1µs in jeder Sekunde. Dieser Fehler muss von einer Synchronisationseinrichtung kompensiert werden. Soll der Gangunterschied zweier Uhren einen Wert von 10µs nicht überschreiten, so müssen die Uhren nach mindestens 10 Sekunden erneut synchronisiert werden.

Zwei verschiedene Fehlerfälle können bei physikalischen Uhren auftreten. Die Driftrate kann sich so verändern, dass sie jenseits der maximalen Driftrate liegt, oder es tritt eine Fehlfunktion im Zähler auf, so dass ein statischer Fehler in der Bestimmung des richtigen Zeitpunktes vorliegt. Abbildung 3.10 stellt diese Fehler grafisch dar.

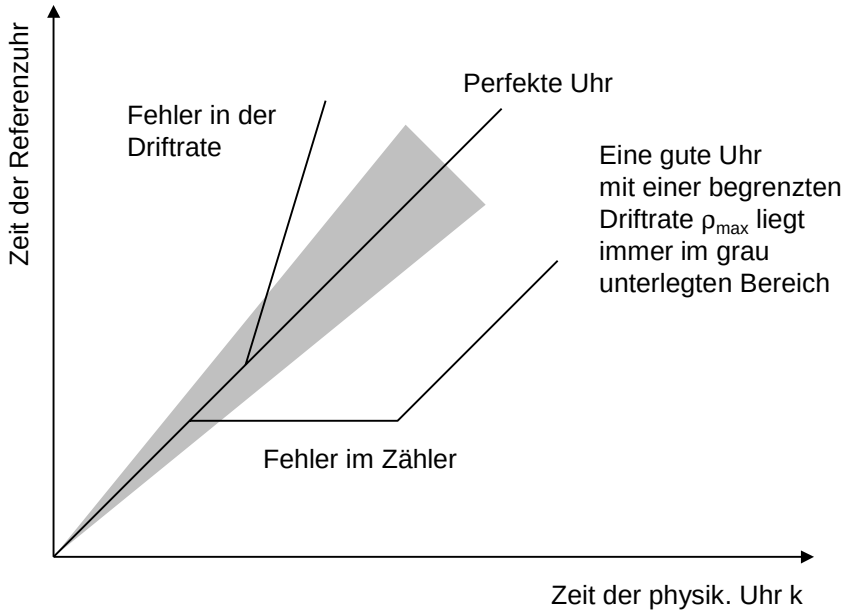



**Abbildung 3.10:** Fehlerfälle bei physikalischen Uhren

### 3.4.2 Synchronisation

Der Offset zum Zeitpunkt des Mikroticks i zwischen zwei Uhren j und k mit der gleichen Granularität ist bestimmt durch

$$\mathrm{offset}_i^{jk} = \left| z(\mathrm{microtick}_i^j) - z(\mathrm{microtick}_i^k) \right| \qquad (3.3)$$

Damit ist die Zeitdifferenz zwischen den jeweiligen Mikroticks der beiden Uhren bezogen auf die Mikroticks der Referenzuhr bezeichnet.

Ist eine Menge von Uhren $\{1, 2, \ldots, n\}$ gegeben, so wird der maximale Offset zwischen zwei Uhren dieser Menge

$$\Pi_i = \max\{offset_i^{jk}\}, \forall 1 \leq j,k \leq n \tag{3.4}$$

definiert als Präzision $\Pi_i$ der Menge zum Mikrotick i. Das Maximum von $\Pi_i$ im betrachteten Zeitintervall wird die Präzision $\Pi$ der Menge genannt. Der Prozess der gegenseitigen Synchronisation der Uhren der Menge innerhalb einer bestimmte Präzision wird *interne Synchronisation* genannt.

Der Offset zwischen einer Uhr k und der Referenzuhr z zum Mikrotick i ist die Genauigkeit $(accuracy_i^k)$. Das Maximum im betrachteten Zeitraum wird dann als $accuracy^k$, als Genauigkeit der Uhr k bezeichnet. Um die Genauigkeit einer Uhr in gegebenen Grenzen zu halten, ist eine ständige Synchronisierung mit der externen Referenzuhr notwendig. Dies wird als die *externe Synchronisation* bezeichnet.

Als externe Zeitbasis bieten sich international gültige Zeiten an. Dabei sind vor allem die International Atomic Time (TAI) und die Universal Time Coordinated (UTC) zu nennen.

Bei der TAI wird die Sekunde als eine Dauer von 9 192 631 770 Perioden der Strahlung eines spezifischen Cäsium-Isotops definiert. Hieraus ergibt sich eine kontinuierliche Zeiteinteilung ohne Unregelmäßigkeiten. Die Dauer einer Sekunde wurde abgeleitet von astronomischen Größen, wie sie die zweite internationale Zeitbasis (UTC) benutzt.

Bei der Universal Time Coordinated handelt es sich um eine Zeitbasis, die aus astronomischen Beobachtung der Erdrotation in Bezug zur Sonne abgeleitet wurde. Sie löste 1972 den bis dahin anerkannten Standard, die Greenwich Mean Time (GMT), ab. Die Dauer einer Sekunde entspricht der TAI. Die Erdrotation ist jedoch nicht gleichförmig, so dass von Zeit zu Zeit eine sogenannte leap second in die Zeitbasis eingefügt wird. Per Definition zeigten die UTC und die TAI am 1. Januar 1958 0.00 Uhr absoluten Gleichstand. Bis heute haben sie einen Gangunterschied von mehr als 30s erreicht. Der Zeitpunkt, wann die leap seconds eingefügt werden, wird vom Bureau International de l'Heure bestimmt und publik gemacht. Somit ist der aktuelle Offset zwischen TAI und UTC jederzeit bekannt.

Eine einfache Möglichkeit zur externen Synchronisationen der lokalen Uhr eines Rechners bietet das Network Time Protocol (NTP). Über das Internet werden Zeitinformationen ausgetauscht und die lokale Uhr danach gestellt. Wenn ein lokaler Timeserver eingesetzt wird ist eine Synchronisation bis auf 1 ms möglich.

Für den zeitlichen Abgleich in einem Echtzeitsystem spielt die interne Synchronisation jedoch eine größere Rolle. Jeder Netzknoten besitzt eine eigene Uhr, die auf die übrigen Netzteilnehmer sehr präzise (im Bereich einiger Mikrosekunden) abgestimmt werden muss. Eine externe Synchronisation aller Knoten mit gleicher Präzision ist mit realistischen Aufwand nicht zu erreichen. Es ist also die Aufgabe eines Synchronisationsalgorithmus, die Uhren aller Netzknoten innerhalb einer vorgegebenen Präzision $\Pi$ zu halten. Dieser Algorithmus sollte fehlertolerant arbeiten, da eine präzise, globale Zeit entscheidend für das Netzwerk ist.

Die Synchronisation wird periodisch wiederholt. Der Abstand zwischen zwei Synchronisationsvorgängen ist dabei das Resynchronisationintervall $R_{int}$. Nach dem Ablauf eines Intervalls wird eine Uhr neu justiert, indem eine Korrekturzeit berechnet wird, die der aktuellen Zeit aufaddiert oder abgezogen wird. Diese Funktion zur Berechnung der individuellen Korrekturzeit wird Konvergenzfunktion $\Phi$ genannt. Nach dem Abgleich driften die Uhren wieder auseinander. Dabei beschreibt der Drift Offset $\Gamma$ die maximale Divergenz zwischen zwei Uhren im betrachteten Netzwerk. $\Gamma$ hängt dabei von der maximalen Driftrate $\rho$ und der Länge des Resynchronisationsintervalls ab.

$$\Gamma = 2\rho R_{int} \tag{3.5}$$

Eine Menge von Uhren kann nur synchronisiert werden, wenn folgende Bedingung erfüllt ist:

$$\Phi + \Gamma \leq \Pi \tag{3.6}$$

Angenommen die Uhren am Ende eines Synchronisationsintervalls sind an den Rand des Präzisionsintervalls gelangt, so muss die Konvergenzfunktion $\Phi$ die Uhren soweit synchronisieren, dass sie während des nächsten Synchronisationsintervalls die vorgegebene Präzision $\Pi$ nicht überschreiten (Abbildung 3.11) [11],[9].

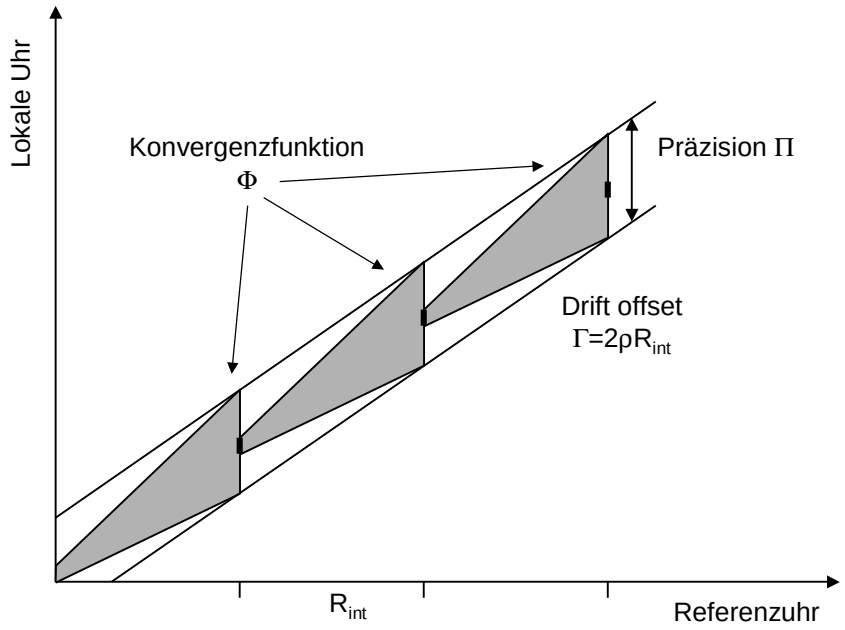



**Abbildung 3.11:** Synchronisationsbedingung

Im Gegensatz zu diesem Ansatz eine Software-Synchronisation einzusetzen, steht die Hardware-Synchronisation, wie sie in Multiprozessorsytemen oder auch in verteilten Rechnersytemen (z.B. PC-Cluster) verwendet wird. Die meisten dieser Systeme arbeiten mit einer Regelung im Sinne der Nachlaufsynchronisation oder phase-locked-loop (PLL) [18],[22]. Dabei werden die lokalen Uhren der einzelnen Knoten auf ein Referenzsignal eingeregelt. Der Vorteil dieser Möglichkeit liegt in der großen Genauigkeit des Verfahrens, ein Nachteil ist jedoch die aufwändige Hardware und die damit verbundenen Kosten. Untersuchungen haben ergeben, dass sich die besten Ergebnisse mit adaptiv-parametrierten Reglern erzielen lassen [18].

### 3.4.3 Fehlertoleranz und FTA-Algorithmus

Ein fehlertolerantes System sollte sich tolerant gegenüber den sogenannten Byzantinischen Fehlern verhalten[1]. Eine Uhr mit einem derartigen Fehlerverhalten liefert verschiedene Werte für die Uhrzeit, wenn sie von unterschiedlicher Seite abgefragt wird. Ein Beispiel dazu ist in

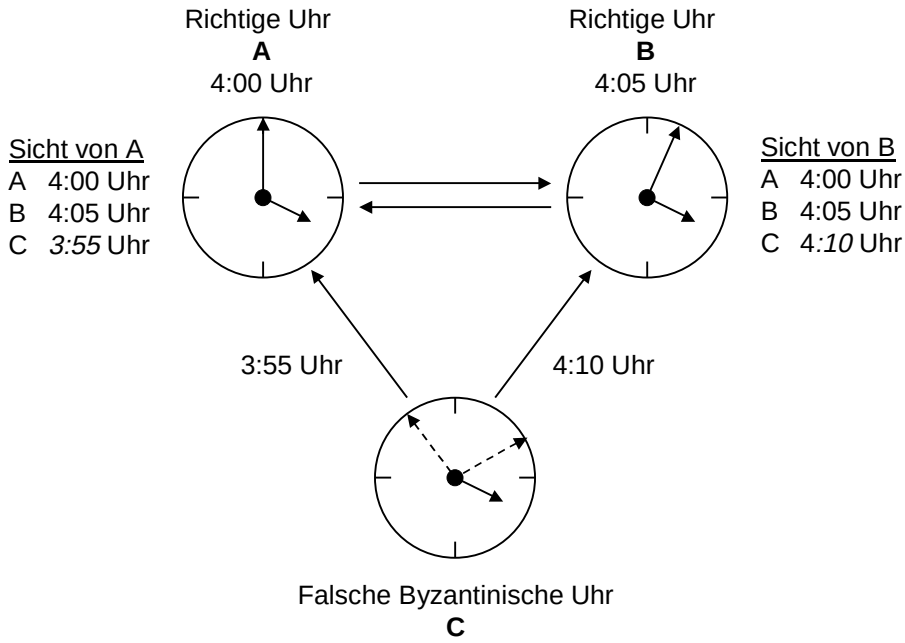



**Abbildung 3.12:** Byzantinischer Fehler

Abbildung 3.12 dargestellt. Uhr A und Uhr B sind richtig gehende Uhren, während Uhr C die falsche Byzantinische Uhr („two-faced“ malicious clock) ist. Fragt A die Uhrzeiten der anderen Uhren ab, so liefern A und B den richtigen Wert, C liefert jedoch mit 3:55 Uhr einen

1. Die Bezeichnung „Byzantinischer Fehler“ geht auf das Problem der byzantinischen Generäle zurück, wie es von Lamport et al. in [15] beschrieben wurde. Als die byzantinische Armee einmal eine feindliche Stadt belagerte, war sie in mehrere Lager ausserhalb der Stadt aufgeteilt, die jeweils von einem General befehligt wurden. Einige dieser Generäle waren jedoch Verräter. Eine Kommunikation war nur über Boten möglich. Das Problem besteht nun darin einen Algorithmus zu finden, nach dem sich die Generäle auf einen einheitlichen Plan (Angriffszeitpunkt) einigen können, obwohl einige Generäle diesen Plan sabotieren.

falschen Wert. Wenn B die Uhrzeiten abfragt, liefern A und B jeweils ihren richtigen Wert, während Uhr C nun 4:10 Uhr zurückgibt. Somit ist der gesamte Datensatz nicht mehr konsistent. Es konnte nach [14] gezeigt werden, dass k Byzantinische Fehler toleriert werden können, wenn für die Anzahl der Uhren im Netzwerk N gilt:

$$N \geq (3k + 1) \tag{3.7}$$

Damit ist klar, dass ein System, das sich fehlertolerant gegenüber zumindest einer falschen Uhr verhalten soll, mindestens aus vier Knoten bestehen muss.

Aufbauend auf diesen Überlegungen zur Fehlertoleranz wurde der FTA-Algorithmus (Fault Tolerant Averaging) als eine Lösung für die Konvergenzfunktion $\Phi$ entwickelt [9]. Zur Bestimmung des eigenen Offset sammelt ein Knoten die Zeiten aller anderen Netzteilnehmer in einem Vektor $y_i$. Nachdem dieser Vektor komplett ist, werden die Elemente der Größe nach geordnet. Anschließend berechnet sich der Korrekturterm $C_j$ zu jedem Knoten j nach Gleichung (3.8). k bezeichnet die Anzahl der falschen Uhren [9].

$$C_j = -\frac{1}{N - 2k} \sum_{i = k + 1}^{N - k} y_{i,j} \tag{3.8}$$

Ausgehend von einer Präzision $\Pi$ für das Netzwerk, ist der Fehler, den eine Byzantinische Uhr bei zwei verschiedenen Knoten erzeugt:

$$E_{byz} = \frac{\Pi}{N - 2k} \tag{3.9}$$

Im schlimmsten Fall für k Byzantinische Uhren ergibt sich hieraus ein Fehlerterm von:

$$E_{k,\,byz} = \frac{k\Pi}{N - 2k} \tag{3.10}$$

Wird außerdem ein Jitter $\varepsilon$ der Nachrichten auf dem Netzwerk angenommen, so ergibt sich für die Konvergenzfunktion folgender Term:

$$\Phi(N, k, \varepsilon) = \frac{k\Pi}{N - 2k} + \varepsilon \tag{3.11}$$

Setzt man in diese Gleichung nach Beziehung (3.6) die obere Grenze für die Präzision

$\Pi = \Gamma + \Phi$ ein, dann ergibt sich nach einfacher algebraischer Umformung:

$$\Pi(N, k, \varepsilon, \Gamma) = (\varepsilon + \Gamma)\frac{N - 2k}{N - 3k} = (\varepsilon + \Gamma)\mu(N, k) \tag{3.12}$$

Durch Gleichung (3.12) wird die Präzision des FTA-Algorithmus ausgedrückt. $\mu(N, k)$ wird dabei der Byzantinische Fehlerterm genannt. Sein Einfluss auf die Präzision kann auch tabellarisch dargestellt werden (Tabelle 3.1). Zu beachten ist, dass sich dieser Fehlerterm auf einen

| | N | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Fehler | 4 | 5 | 6 | 7 | 10 | 15 | 20 | 30 |
| 1 | 2 | 1,5 | 1,33 | 1,25 | 1,14 | 1,08 | 1,06 | 1,03 |
| 2 | | | | 3 | 1,5 | 1,22 | 1,14 | 1,08 |
| 3 | | | | | 4 | 1,5 | 1,27 | 1,22 |

**Tabelle 3.1:** Byzantinischer Fehlerterm $\mu(N, k)$

Zeitenvektor bezieht, der nach jedem Rundlauf aktualisiert wird. Sollte also eine dauerhafte Verschlechterung der Präzision eintreten, so müssten über alle Zyklen mehrere Byzantinische Fehler auftreten, was ein äußerst unwahrscheinlicher Fall ist.

Ein praktisches Beispiel zum Einsatz des FTA-Algorithmus ist das TTP/C-Protokoll der Firma TTTech Wien [26]. Es handelt sich hierbei um ein zeitgesteuertes Echtzeitprotokoll, das in einem ASIC implementiert wurde. Ein Netzwerkknoten besteht immer aus einem TTP/C Controller, einem Host und einem Interface (CNI), auf das Cotroller und Host zugreifen können (vgl. Abbildung 3.13). Der Netzwerkzugriff wird autonom durch den Controller bestimmt. Hier ist auch die zeitliche Synchronisation realisiert. Jeder Knoten besitzt einen eigenen Parametersatz, indem die Scheduling-Informationen für die zu versendenden und zu erwartenden Nachrichten gespeichert sind. Dieser Parametersatz wird message descriptor list (MEDL) genannt. Die Netzwerkzugriffsstrategie ist TDMA (time division multiple access), deren Rundlauflänge durch die Länge der MEDL bestimmt wird. Nachdem ein Rundlauf beendet ist, wird der nächste mit dem gleichen temporalen Zugriffsmuster, möglicherweise jedoch mit anderem Nachrichteninhalt, gestartet. Der Aufbau der MEDL ist in Abbildung 3.14 dargestellt. Die Zeitsynchronisation nutzt den FTA-Algorithmus.

Zusätzlich bietet das Protokoll Möglichkeiten zur weiteren Erhöhung der Fehlertoleranz an. Dabei wird die redundante Ausführung von Kommunikationskanälen in Kombination mit einem Voter benutzt (Replica Determinismus). Die zugehörige Logik ist ebenfalls im Controller untergebracht. Aufgrund der Zeitsteuerung und des a priori festgelegten Schedulings ist der Nachrichten-Overhead sehr gering, da keinerlei Statusbits, Synchronisationsnachrichten oder ähnliches gesendet werden müssen.

Die praktische Einrichtung eines solchen Netzwerkes erfolgt durch ein Konfigurations-Software-Tool. Hier wird festgelegt, wie die Netzwerkstruktur aufgebaut ist und welche Nachrichten ausgetauscht werden sollen. Es handelt sich also um ein explizites Planungsverfahren.

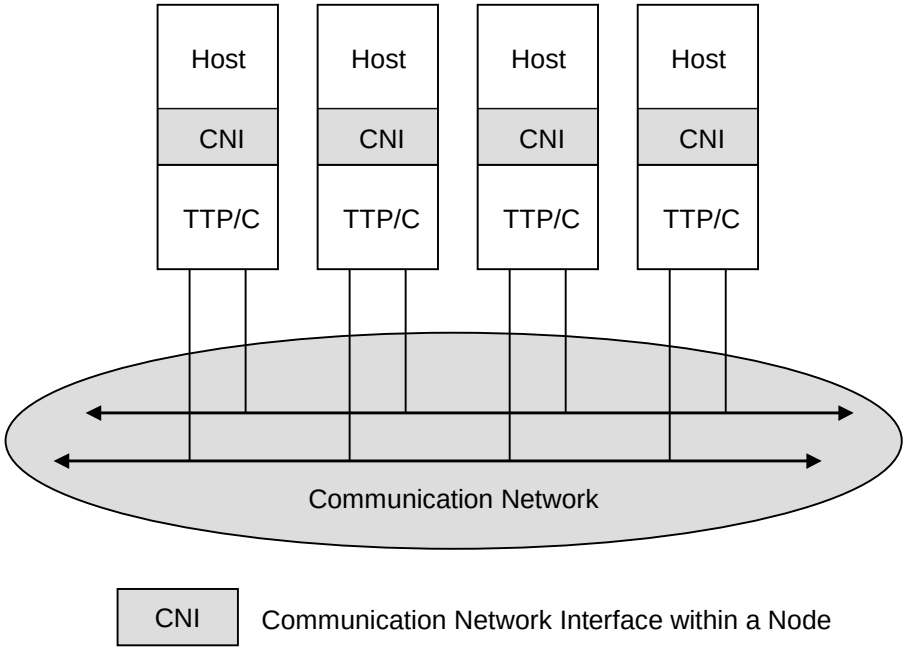



**Abbildung 3.13:** Beispiel eines TTP/C Netzwerkes

| SRU-Time | Address | Attributes | | | |
|---|---|---|---|---|---|
| | | D | L | I | A |
| | | | | | |
| | | | | | |
| | | | | | |

**Attributes:**

| | | |
|---|---|---|
| D | Direction | Richtung der Kommunikation |
| L | Length | Länge der Nachricht |
| I | Init | Initialisierungsbit, zeigt an ob die Nachricht eine Init-Nachricht oder eine normale Nachricht ist |
| A | Additional | zusätzliche Information über Mode-Wechsel |

**Abbildung 3.14:** Aufbau der MEDL

# 4 Realisierung

Im Rahmen der Arbeit wurde ein Treiber implementiert, mit dessen Hilfe ein echtzeitfähiges Ethernet-Netzwerk gebildet werden konnte. Er bietet den Ausgangspunkt um die in Kapitel 3 erarbeiteten Aspekte in einer reinen Software-Umgebung zu untersuchen. Dazu wird ein einfaches Protokoll vorgestellt, das eine Bewertung der Leistungsfähigkeit des Treibers ermöglicht und eine Grundlage für aufbauende Entwicklungen bietet.

Als Testaufbau wurden drei PCs benutzt. Es handelte sich dabei um PII 400MHz-Maschinen mit jeweils 128 MByte Hauptspeicher. Die Computer waren in ihrer Grundkonfiguration (Mainboard, Prozessor, Netzwerkkarten, ...) identisch. Es waren jeweils zwei Netzwerkkarten vorhanden um zum einen eine Datenverbindung zum Hausnetz herzustellen und zum anderen eine unabhängig programmierbare Karte für die Entwicklung nutzen zu können. Die Rechner waren über einen Ethernet-Hub verbunden. Das ist wichtig, da ein Hub im Gegensatz zu einem Switch alle Pakete an jeden Teilnehmer unmittelbar weitergibt. Ein Switch hätte möglicherweise zeitliche Verschiebungen verursacht und die Multicast-Kommunikation gestört.

## 4.1 Aufbau und Schnittstellen

Die Netzwerkkartentreiber bilden im Linux-Kernel nach Abschnitt 3.3 eine eigenständige Gruppe von Treibern. Um das Netzwerk, das mit Echtzeitfähigkeiten auszustatten war, auch weiterhin von Linux aus nutzen zu können, musste die Grundstruktur des Treibers beibehalten werden. Um jedoch den Echtzeitanforderungen gerecht zu werden, war es außerdem notwendig, direktere Schnittstellen zum Kartentreiber zu schaffen. So konnte die Nutzung der dynamischen Datenstrukturen für Netzwerkpakete im Linux-Kernel umgangen werden, da hier keine verlässlichen zeitlichen Aussagen zu treffen sind. Insgesamt sind drei verschiedene Zugänge zum Treiber realisiert: der „normale“ Zugang von Linux zur IP-Kommunikation, eine Repräsentation des Treibers im /dev-Verzeichnis als separates Modul und der direkte Zugang für RTLinux in Form von globalen Zugriffsfunktionen im Kernel.

Der Treiber besteht im wesentlichen aus dem modifizierten Linux-Orginaltreiber *3c59x_rt.o* und der Eigenentwicklung *3c_rt_drv.o*, in dem alle spezifischen Echtzeitalgorithmen implementiert sind. Das Modul wird aus den drei Programmdateien *3c_interface.c*, *3c_protocol.c* und *3c_mac.c* zu einem nachladbaren RTLinux-Modul gelinkt.

Nach dem Booten des Systems ist der modifizierte Orginaltreiber *3c59x_rt.o* im Kernel aktiv. Er wird automatisch beim Hochfahren gestartet. In diesem Modul befindet sich kein RT Linux-Aufruf, da das Echtzeitsystem zur Bootzeit noch nicht geladen ist und die entsprechenden Symbole dem Kernel noch nicht bekannt sind. Die Funktion des Treibers ist in diesem Zustand unverändert gegenüber der regulären Linux Installation. Nach dem Start des RT Linux Echtzeitsystems, kann das Modul *3c_rt_drv.o* in den Kernel geladen werden. Bei dessen Start wird der modifizierte Orginaltreiber auf Echtzeitbetrieb umgeschaltet. In diesem Zustand hat dieser, außer zur Fehlerbehandlung, keinen direkten Zugriff mehr auf die Netzwerkkarte. Stattdessen werden zwei Prioritätswarteschlangen (Senden und Empfangen) initialisiert, in denen alle Pakete zwischengespeichert werden (vgl. Abbildung 4.1). Dabei werden Standard TCP/IP-Pakete mit der niedrigsten Priorität (0) eingeordnet, Pakete von der Device-Schnittstelle haben mit 1 die zweitniedrigste Priorität, während die Priorität von RT Linux Paketen von 2-255 frei gewählt werden kann. Somit ist gewährleistet, dass Echtzeitpakete bevorzugt abgearbeitet werden. RT Linux-Pakete werden durch Aufruf einer globalen

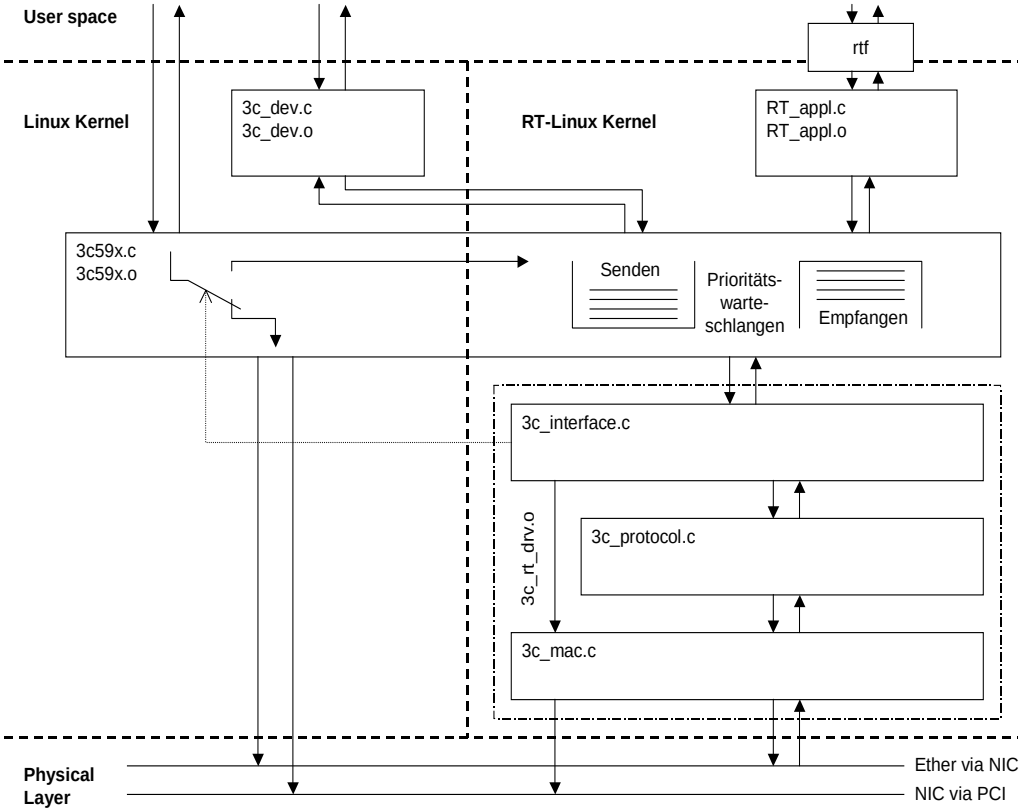



**Abbildung 4.1:** Struktur des Treibers

Kernel-Funktion eingeordnet. Die Prioritätswarteschlangen werden auf der anderen Seite von dem Modul *3c_rt_drv.o* bearbeitet. Somit wird die gesamte Netzwerkkommunikation über dieses Modul geführt, weshalb hier alle echtzeitrelevanten Algorithmen vereinigt werden konnten.

Die Dreiteilung des Echtzeitmoduls ergibt sich aus seinen unterschiedlichen Aufgaben. Während in *3c_interface.c* alle Software-Schnittstellen zum System, einschließlich der Funktionen `init_module(void)` und `cleanup_module(void)` realisiert sind, vereinigt *3c_mac.c* alle Funktionen, die für den Hardware-Zugriff auf die Netzwerkkarte zuständig sind. Die Zeitsteuerung und die Protokollfunktionen, also der entscheidende Teil des Treibers, befindet sich in *3c_protocol.c*.

Es ist für die Realisierung der Prioritätswarteschlangen notwendig, dynamisch Kernelspeicher zu allokieren. Ebenso muss Speicher zur Verarbeitung der Socket-Buffer für die Netzwerkkommunikation reserviert werden. Es ist jedoch nicht möglich, diese Speicherfunktionen sicher im RT Modus von RT Linux aufzurufen, da sie durch das Sperren und Wiederfreigeben von Interrupts synchronisiert werden. Dies bezieht sich jedoch auf die Linux-Interrupts, die von der Echtzeiterweiterung abgefangen werden. Wird der Kernel-Speicherbereich im Echtzeitbetrieb von RT Linux verändert, so ist kann kein konsistenter Zustand des Speichers für die Zugriffsfunktionen im Linux Bereich garantiert werden. Um diese Aufrufe im Echtzeitbetrieb zu umgehen, wurden Software-Interrupts eingerichtet. Die Service-Routinen für diese Interrupts werden im Linux-Modus ausgeführt. Es wurden des-

halb statische Ringpuffer eingerichtet, in denen Zeiger auf Speicherblöcke, die im voraus allokiert wurden, abgelegt werden. Ein RT Linux Thread kann sich aus diesen Ringpuffern bedienen, was eine determinierte und damit echtzeitfähige Zugriffszeit bedeutet. Der Thread löst anschließend einen entsprechenden Software-Interrupt aus, in dessen Service-Routine der Ringpuffer wieder mit vorher reservierten Blöcken aufgefüllt wird.

## 4.2 Programmierung der Netzwerkkarte

Als Entwicklungsobjekt wurde die Netzwerkkarte 3c905B der Firma 3Com gewählt. Es handelt sich hierbei um eine aktuelle, handelsübliche 100 Mbit Karte. Der Interruptbetrieb dieser Karte ist frei programmierbar, d.h. der Treiber bestimmt, bei welchem Ereignis ein Interrupt ausgelöst werden soll. Die Konfiguration und die Steuerung der Karte erfolgt über einige Command- und Statusregister, der Datentransfer der Ethernet-Pakete erfolgt über DMA. Dabei nutzt die Karte PCI-Busmastering.

### 4.2.1 Aufbau und Funktionsweise der Karte

Alle wichtigen Funktionen der Karte sind in einem zentralen ASIC untergebracht. Ergänzt wird dieser Chip nur noch durch ein serielles EEProm sowie einem BIOS ROM. Das gesamte PCI Management, sowie der Zugriff auf das Netzwerkmedium, inklusive Ethernet Zugriffprotokoll, sind ebenfalls im ASIC untergebracht.

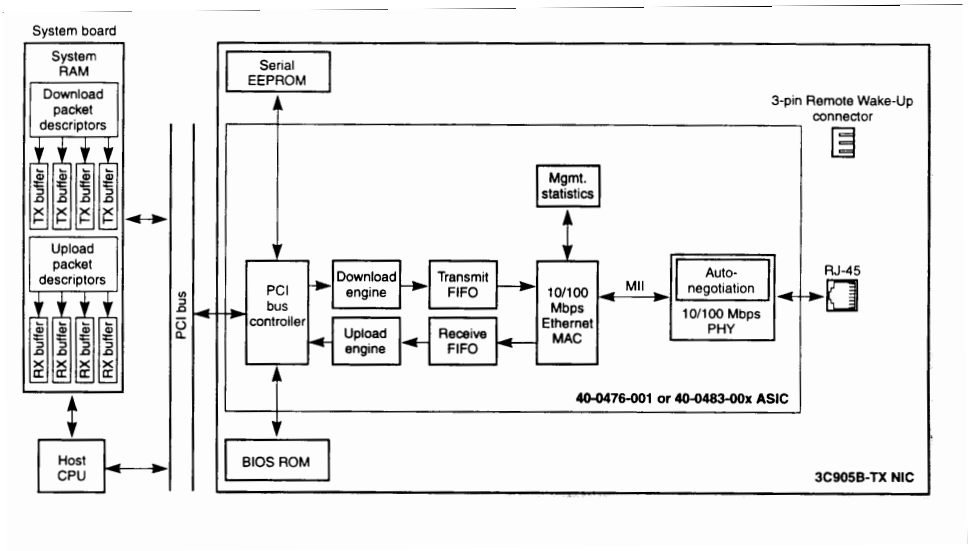



**Abbildung 4.2:** 3C905B System Architektur [31]

Der Aufbau der Karte ist in Abbildung 4.2 dargestellt. Zum Versenden müssen die Daten zunächst aus dem Hauptspeicher des PC auf die Karte übertragen werden. Dafür ist die sogenannte Download engine zuständig. Zeitlich koordiniert sie den Datentransfer so, dass keine inkonsistenten Zustände in der Netzwerkkarte entstehen. Die Daten werden in einen FIFO

Speicher (Transmit FIFO) auf der Karte zwischengespeichert. Empfangene Pakete werden in einem eigenen FIFO (Receive FIFO) zwischengespeichert und von der Upload engine in den Hauptspeicher übertragen. Die Grösse der FIFO Speicher beträgt jeweils 2 kByte. Der PCI bus controller übernimmt bei jedem Datentranfer die Rolle des PCI-Busmasters und öffnet so einen DMA-Kanal über den die Daten direkt übertragen werden könnnen.

### 4.2.2 Datenaustausch

Um Daten über die Netzwerkkarte zu versenden, legt der Treiber eine verkettete Liste im Hauptspeicher an. Die Elemente dieser Liste haben ein ganz bestimmtes Datenformat, so wie es die Karte erwartet. Abbildung 4.3 zeigt den Aufbau dieser Struktur.

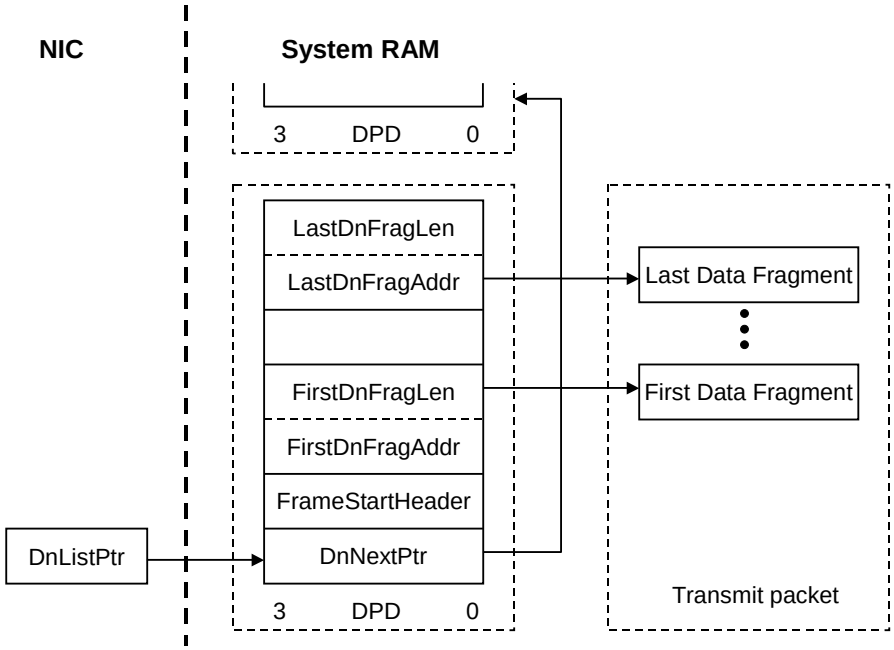



**Abbildung 4.3:** Downlist

Liegen Daten zum Versenden bereit, so schreibt der Treiber die Adresse des ersten Listenelementes in das DnListPtr-Register der Netzwerkkarte. Nachdem alle Fragmente des Paketes in den FIFO-Speicher übertragen wurden[1], wird die Adresse des nächsten Paketes (DnNextPtr) automatisch in das DnListPtr-Register übernommen und verarbeitet. Dieser Vorgang ist beendet, sobald ein DnNextPtr-Feld den Wert 0 enthält.

Der Einsatz von dynamischen Listen ist jedoch unter Echtzeitbedingungen nicht sinnvoll, da sich die Verarbeitungszeit nicht deterministisch verhält. So dauert ein Datentransfer von 5 verketteten Paketen natürlich länger, als die Übertragung nur eines Paketes. Wird aber bei jedem Sende- und Empfangsvorgang nur ein Paket verarbeitet, so hat nur noch dessen Länge einen Einfluss auf die Übertragungsdauer.

Um das Versenden nur eines Paketes zu realisieren, wird in unserem Fall die Adresse einer

1. Die Erkennung des letzten Fragmentes erfolgt durch Setzen des letzten Bit des entsprechenden DnFragLen-Feldes.

Datenstruktur in das DnListPtr-Register geladen, deren DnNextPtr den Wert 0 enthält. Die Liste ist damit beendet und enthält nur ein Element. Ebenso enthält die Datenstruktur nur ein Fragment, dessen Adresse das `s_kbuff.data`-Feld des aktuell zu verarbeitenden Socket-Buffers ist (vgl. Abschnitt 3.3 zu Socket Buffern). Dadurch reduziert sich die Download-Struktur wie in Abbildung 4.4 gezeigt.

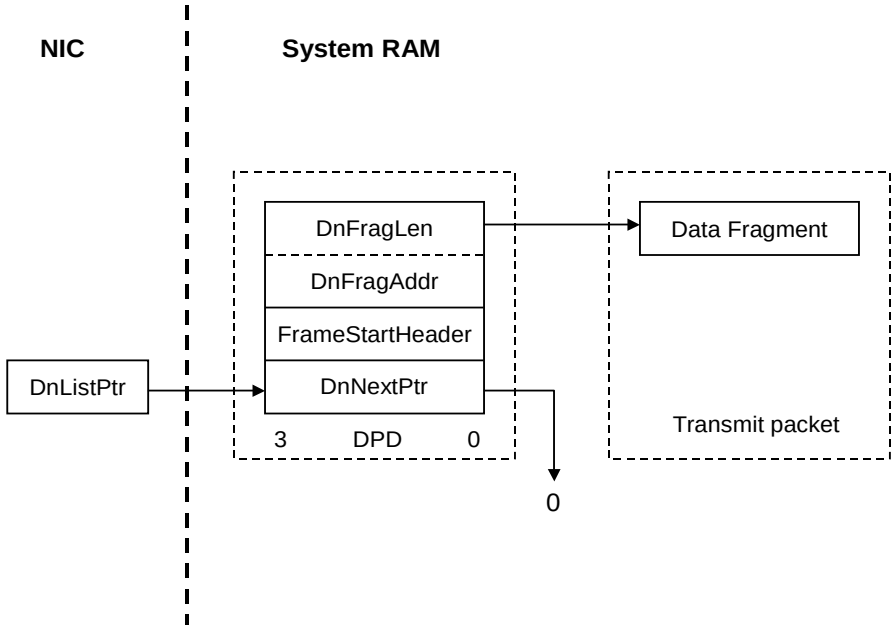



**Abbildung 4.4:** Downlist speziell für Echtzeitverhalten

Der Typdefinition der Listenfelder hat folgendes Format.

```
struct boom_tx_desc {
  u32 next;    /* Last entry points to 0.   */
  s32 status;  /* bits 0:12 length, others see below.  */
  u32 addr;
  s32 length;
};
```

Die Belegung der Felder in der Senderoutine wurde wie folgt implementiert[1].

```
mac_buf->tx_buf.next   = 0;
mac_buf->tx_buf.addr   = cpu_to_le32(virt_to_bus(skb->data));
mac_buf->tx_buf.length = cpu_to_le32(skb->len | LAST_FRAG);
mac_buf->tx_buf.status = cpu_to_le32(skb->len);
```

Um das Ende des Download-Vorgangs zu erkennen pollt der Treiber auf das DnListPtr-Register. Wird der Wert 0 erkannt, so ist der Datentranfer beendet und das Datenpaket wurde

1. Durch die Funktion `cpu_to_le32` wird das Datenformat des übergebenen Integer-Wertes in das Little-Endian-Format konvertiert, wie es die Karte erwartet.

in den FIFO der Karte übertragen. Der Speicher des entsprechenden Socket-Buffers wird freigegeben. Die Karte sorgt selbstständig für das physikalische Versenden des Paketes auf dem Netzwerkmedium. Im Normalfall kommt hier das Ethernet-Zugriffsprotokoll CSMA/CD zum Einsatz, das in der Hardware der Netzwerkkarte fest implementiert ist. Bei diesem Protokoll kommt es zu Kollisionen von Paketen, wenn zwei Netzteilnehmer gleichzeitig senden. Nach einer zufälligen, kurzen Wartezeit versucht die Netzwerkkarte das Paket erneut zu senden [27]. Diese Funktion kommt jedoch bei der vorliegenden Implementierung des Treibers nicht zum Tragen, da die einzelnen Netzteilnehmer nur in zugewiesenen Zeitscheiben senden dürfen. Es kann so niemals zur Kollision kommen, das CSMA/CD Protokoll ist praktisch transparent geschaltet.

Die Übertragung empfangener Pakete von der Netzwerkkarte in den Hauptspeicher des PC wird als Upload bezeichnet. Prinzipiell erfolgt diese Übertragung analog zum Datentransfer vom Hauptspeicher zur Karte. Der Treiber reserviert einen geeigneten Speicherbereich (UPD), und lädt dessen Adresse in das UpListPtr-Register der Netzwerkkarte (vgl. Abbildung 4.5).

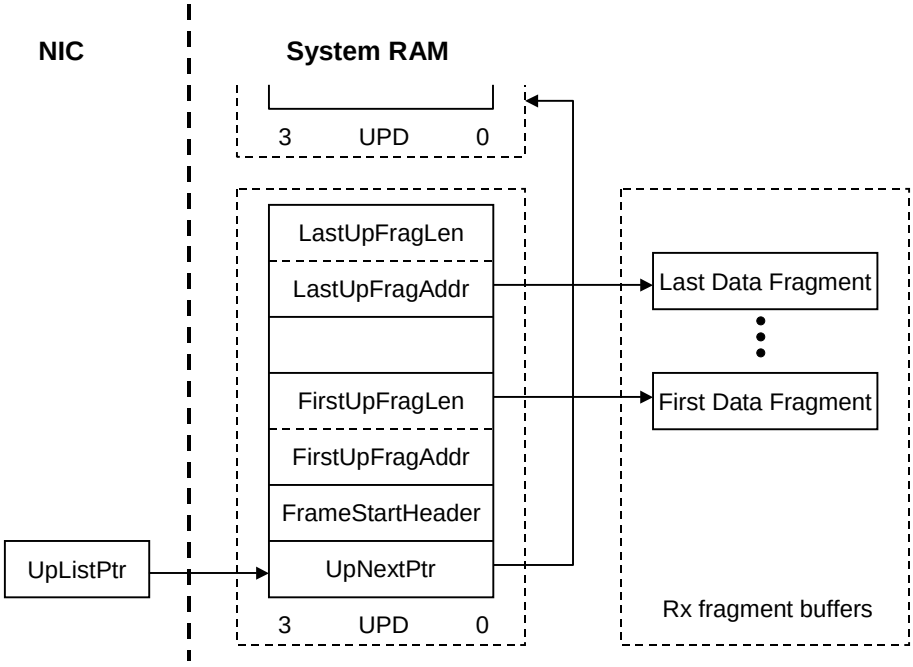



**Abbildung 4.5:** Uplist

Da aber auf der Senderseite jeweils nur ein Paket verschickt wird, muss im Treiber auch hier nur ein Feld für die Empfangsliste vorgesehen werden. Nach dem Empfang wird für das Paket ein Socket-Buffer angelegt, der vom Treiber und dem Linux-Kernel verarbeitet werden kann. Die Datenstruktur, in die ein empfangenes Paket geladen wird, wurde analog der Datenstruktur für zu versendende Pakte eingesetzt. Sie besteht auch hier aus einer Liste mit nur einem Element und einem Fragment wie es in Abbildung 4.6 gezeigt wird.

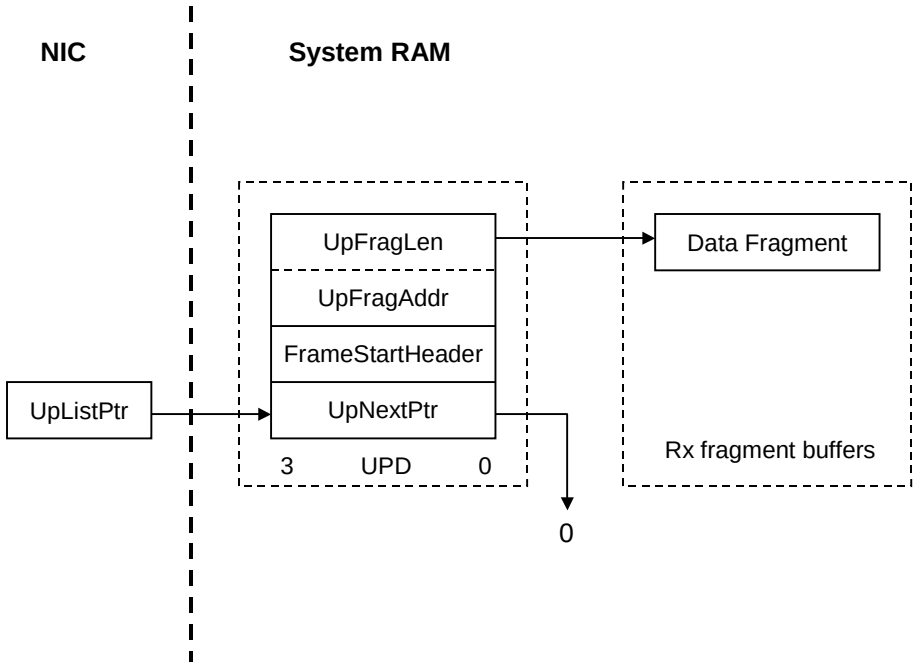



**Abbildung 4.6:** Uplist speziell für Echtzeitverhalten

Die Typdefinition eines Listenfeldes der Empfangsliste hat folgendes Format.

```
struct boom_rx_desc {
  u32 next;    /* Last entry points to 0.   */
  s32 status;
  u32 addr;    /* Up to 63 addr/len pairs possible. */
  s32 length;  /* Set LAST_FRAG to indicate last pair. */
};
```

Nachdem die Netzwerkkarte ein Paket empfangen hat, füllt sie die Felder der Struktur mit gültigen Werten. In das UpListPtr-Register wird der Inhalt des Feldes `next` übernommen. In unserem Fall befindet sich jedoch in diesem Feld der Wert 0, so dass die Karte die Datenübertragung in den Hauptspeicher des PC stoppt. Nachdem das Paket vom Treiber ausgewertet wurde, d.h. der Inhalt in einen Socket Buffer übertragen wurde, wird die Adresse des UPD erneut in das UpListPtr-Register geladen. Zuvor hat der Treiber die Felder `status` und `length` auf den Wert 0 gesetzt, so dass die Karte wieder einen gültigen Speicherbereich vorfindet, um das nächste Paket zu empfangen. Die Karte wurde gewissermaßen wieder „scharf“ gemacht. Im folgenden wird der Programmcode zu diesem Vorgang gezeigt[1].

1. Beim Wiedereinschalten der Empfangsbereitschaft wird zunächst das UpStall Kommando ausgeführt. Dies bewirkt ein Warten auf die Bereitschaft der Karte, eine Änderung im UpListPtr-Register zuzulassen. Dies ist der Fall, wenn die Karte nicht mehr mit derAbarbeitung einer Aufgabe belegt ist.

```
// NIC hat die empfangenen Daten in den rx-buffer geladen
pkt_len = le32_to_cpu(mac_buf->rx_buf.status) & 0x1fff;

memcpy(skb_put(mac_skb, pkt_len), bus_to_virt(
       le32_to_cpu(mac_buf->rx_buf.addr)), pkt_len);

mac_skb->protocol = eth_type_trans(mac_skb, rt_dev);

// Empfangsbuffer zurücksetzen
mac_buf->rx_buf.status = 0;
mac_buf->rx_buf.next  = 0;

// NIC wieder empfangsbereit schalten
outw(UpStall, rt_ioaddr + EL3_CMD);
// Wait for stall to complete
for (i=600; i>=0; i--)
  if ((inw(rt_ioaddr + EL3_CMD) & CmdInProgress)==0) break;
outl(virt_to_bus(&mac_buf->rx_buf), rt_ioaddr + UpListPtr);

outw(UpUnstall, rt_ioaddr + EL3_CMD);
```

Um die Netzwerkkarte an die besonderen Bedingungen der Echtzeitfähigkeit anzupassen, mussten noch einige weitere Modifikationen gegenüber der Standard-Implementation vorgenommen werden. So werden alle Pakete auf dem Netzwerk im promiscuous-Mode ausgetauscht, d.h. das Pakete, die verschickt werden, von allen Netzteilnehmern empfangen werden. So können mehr Werte für die Zeitsteuerung gewonnen werden, da der Eintreffzeitpunkt von mehr Paketen bestimmt werden kann (vgl. 4.3). Desweiteren wurde eine Benutzung von Interrupts beim Senden und Empfangen vermieden. Statt dessen wird unter Ausnutzung der Kenntnis der Sende- und Empfangszeitpunkte (Zeitsteuerung) im Polling-Betrieb die Ausführung der Aktion ausgelöst bzw. überwacht. Auf diese Weise hat der Jitter der Interruptlatenzzeit keinen Einfluss mehr auf das Echtzeitverhalten. Zur Fehlerbehandlung können die Interrupts jedoch weiterhin genutzt werden, da hier eine unmittelbare, determinierte Abarbeitung nicht notwendig ist. Deshalb wurde dieser Teil unverändert gegenüber dem Originaltreiber im Modul *3c59x_rt.o* belassen.

## 4.3 Implementierung der Zeitsteuerung

Zur Einhaltung der Echtzeitbedingungen wurde ein zeitgesteuerter Ansatz gewählt. Während in Abschnitt 3.4 die theoretischen Grundlagen zu dieser Lösung diskutiert wurden, soll hier die praktische Umsetzung vorgestellt werden. Grundsätzlich bedeutet Zeitsteuerung, dass Pakete von einem Netzteilnehmer zu einem ganz bestimmten Zeitpunkt gesendet werden, während alle anderen Teilnehmer empfangsbereit sind. Die Grundvoraussetzung ist jedoch das Vorhandensein einer globalen Zeit. Dies kann nur erreicht werden, indem jeder Netzteilnehmer seine lokale Uhr ständig auf die Netzzeit synchronisiert. Bei dem vorliegenden System aus PC und Echtzeitbetriebssytem ist als lokale Zeit diejenige anzusehen, die der Scheduler benutzt. In unserem Fall wird diese Zeit im wesentlichen durch den Prozessortakt und den Timer-Interrupt bestimmt.

### 4.3.1 Ein Regler für die Synchronisationsaufgabe

Zwei Größen bestimmen die Synchronisation verschiedener Uhren

- Offset
- Drift

Ein Synchronisationsalgorithmus muss also diese zwei Größen kompensieren. Der einfachste und unmittelbarste Ansatz hierfür wäre ein zyklisches Messen des Offset und jeweiliges Verstellen der lokalen Uhr. In RT Linux ist jedoch ein direkter Eingriff auf die Zeitbasis des Echtzeitsystems nicht möglich, ohne den Quellcode des Betriebssystems zu verändern. Es ist aber sehr einfach die Periode eines RT-Threads zu verändern. Eine Offset-Korrektur wäre also auch zu realisieren, indem zyklisch die Periode des Scheduling-Threads um den Betrag des gemessenen Offset verändert wird. Der Thread selbst setzt dann bei der nächsten Ausführung seine Periode auf den Originalwert zurück. Dieses Verfahren kann jedoch dazu führen, dass sich die Periode sehr stark von einer Ausführung zur nächsten ändert. Da ein solches Verhalten rechnerisch nur schwer erfasst werden kann, wurde dem eine Regelung der Periode des RT-Threads nach einem bekannten Verfahren vorgezogen.

Da die Regelung auf die Taktrate des RT-Threads wirkt, ist es sinnvoller von einer globalen Frequenz an Stelle einer globalen Zeit zu sprechen. Die ursprünglichen Größen Offset und Drift werden dadurch in die Größen

- Frequenz und
- Phase

transformiert. Wird das Sendeintervall im Netzwerk als globale Frequenz gesetzt, so ist es Aufgabe der Regelung, die Taktrate des RT-Threads auf diese Frequenz frequenz- und phasenstabil einzuregeln. Dieses System einer automatischen phasenstarren Kopplung eines Oszillators (hier der RT-Thread) an einen Bezugsoszillator (globale Frequenz) wird in der Nachrichtentechnik Nachlaufsynchronisation oder PLL (phase locked loop) genannt [25]. Die prinzipielle Anordnung zeigt Abbildung 4.7.

Das Steuergesetz für den Nachlaufoszillator, der hier durch den Echtzeit-Thread repräsentiert wird, lautet

$$f_{rt} = f_0 + k\Delta f \tag{4.1}$$

Hierbei ist $T_{rt} = 1/f_{rt}$ die aktuelle Periode des Echtzeit-Threads, während $T_0 = 1/f_0$ die Basisperiode beim Start des Treibers darstellt. Die Regelung bewirkt damit eine differenzielle Änderung der Periode von ihrem Ausgangswert. Der Phasendetektor liefert einen zeitlichen Versatz zwischen dem erwarteten Einreffen eines Paketes und dem tatsächlichen Eintreffen. Dies kann als Phasenverschiebung der lokalen Frequenz ($f_{rt}$) und der globalen Frequenz des Netzwerkes ($f_g$) interpretiert werden. Diese Phasenverschiebung $\varphi$ wird am Summationspunkt mit der Führungsgröße $\varphi_0$ verglichen und liefert so eine Abweichung $\Delta\varphi = \varphi_0 - \varphi$ von ihrem Sollwert. Dieser Wert ist die Eingangsgröße für den Regler mit der Übertragungsfunktion $A_R$. Der Regler wurde zum Vergleich als PI-Regler und PID-Regler ausgeführt. Da es sich hier um ein digitales System handelt, musste zunächst die äquivalente zeitdiskrete Berechnungsvorschrift des Reglers aus der Übertragungsfunktion[1] berechnet werden. Die

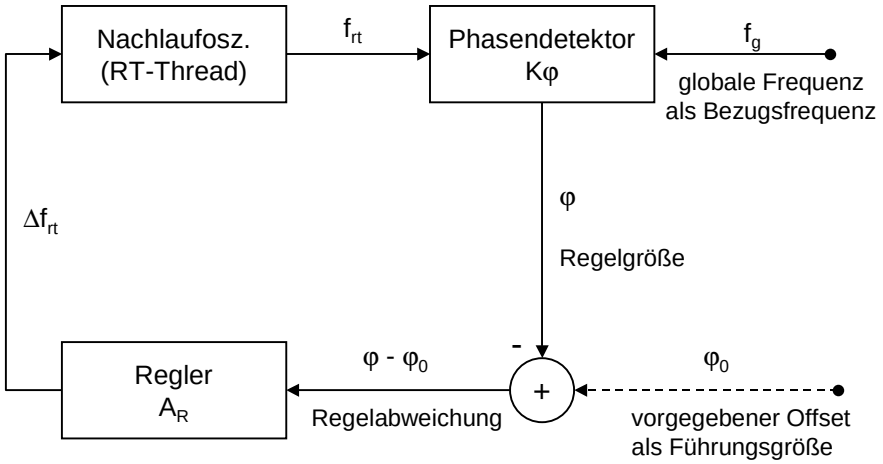



**Abbildung 4.7:** Prinzip des Phasenregelkreises (PLL)

zeitkontinuierliche Gleichung lautet für den PI-Regler [17]

$$u_{PI}(t) = K_R\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau\right] \tag{4.2}$$

Dabei steht $u(t)$ für die Ausgangsgröße des Reglers also $\Delta f_{rt}$ und $e(t)$ für die Eingangsgröße also $\Delta\varphi$. Die entsprechende Gleichung für den PID-Regler lautet [17]

$$u_{PID}(t) = K_R\left[e(t) + \frac{1}{T_I}\int_0^t e(\tau)d\tau + T_D\frac{d}{dt}e(t)\right] \tag{4.3}$$

Werden die Integrale durch eine Summe approximiert und die Differenziale durch eine Differenz, so erhält man für (4.2)

$$u_{PI}(k) = K_R\left[e(k) + \frac{T}{T_I}\sum_{\nu=0}^{k-1} e(\nu)\right] \tag{4.4}$$

1. Die Übertragungsfunktion eines PI-Reglers ist im Laplace-Raum gegeben durch.

$$A_R = \frac{U(s)}{E(s)} = \frac{K_R}{T_I}\frac{1 + T_I s}{s}$$

Transformiert in den Zeitbereich ergibt sich Gleichung (4.2).

und für (4.3).

$$u_{PID}(k) = K_R\left[e(k) + \frac{T}{T_I}\sum_{\nu=0}^{k-1} e(\nu) + \frac{T_D}{T}[e(\nu) - e(\nu-1)]\right] \tag{4.5}$$

Dabei ist T die Periode der äquidistanten Abtastung des Signals. Für die vorliegende Arbeit bedeutet das, dass in jeder Zeitscheibe von jeder Netzwerkkarte ein Paket empfangen werden muss, um eine äquidistante Folge von Eingangswerten für die Regelung zu bekommen.

Um den Regler im Rechner nachzubilden, müssen für die Gleichungen (4.4) und (4.5) rekursive Algorithmen gefunden werden. Dazu wird zunächst $u(k) - u(k-1)$ berechnet und es gilt nach (4.4)

$$\begin{aligned} u_{PI}(k) - u_{PI}(k-1) &= K_R e(k) - K_R(k-1) + \frac{T}{T_I}\sum_{\nu=0}^{k-1} e(\nu) - \frac{T}{T_I}\sum_{\nu=0}^{k-2} e(\nu) \\ &= K_R[e(k) - e(k-1)] + K_R\frac{T}{T_i}e(k-1) \end{aligned}$$

Wird diese Gleichung nach $u_{PI}(k)$ aufgelöst, so erhält man die einfache Beziehung.

$$\begin{aligned} u_{PI}(k) = u_{PI}(k-1) + q_0 e(k) + q_1 e(k-1) \qquad & ,q_0 = K_R \\ & ,q_1 = K_R\left(\frac{T}{T_I} - 1\right) \end{aligned} \tag{4.6}$$

Ebenso kann eine rekursiver Algorithmus für den PID-Regler gefunden werden.

$$\begin{aligned} u_{PID}(k) = u_{PID}(k-1) + q_0 e(k) + q_1 e(k-1) + q_2 e(k-2) \quad & ,q_0 = K_R\left(1 + \frac{T_D}{T}\right) \\ & ,q_1 = -K_R\left(1 + 2\frac{T_D}{T} - \frac{T}{T_I}\right) \\ & ,q_2 = K_R\frac{T_D}{T} \end{aligned} \tag{4.7}$$

Dem Phasendetektor kommt bei dieser Anordnung eine zentrale Bedeutung zu, da die Phase als einzige Messgröße in den Regelkreis eingeht und somit ausschlaggebend ist für die Genauigkeit des Reglers.

### 4.3.2 Praktische Umsetzung des Zeitversatzdetektors

An die Stelle des Phasendetektors tritt in der vorliegenden Arbeit ein Zeitversatzdetektor. Er gibt die Zeit an, um die ein Paket versetzt zum Aufwecken des RT-Threads eintrifft. Im Verhältnis zur Basisperiode des RT-Threads entspricht dieser Zeitversatz einer Phase. Es ist

jedoch nicht notwendig, den Zeitversatz in die Phase umzurechnen, da er ebenfalls als Eingangswert für die Regelung dienen kann. Der Zeitversatzdetektor wird durch die Empfangsprozedur gebildet. Daraus ergibt sich, dass der Zeitversatz nur positive Werte annehmen kann, da die Empfangsprozedur keine Zeitpunkte erkennen kann, die vor ihrem Aufruf lagen. Somit muss die Funktion ausreichend lange vor dem erwarteten Eintreffen eines Paketes aufgerufen werden. Die Implementation der Empfangsprozedur `RT_Protocol_Receive_Buffer` wird in Abbildung 4.8 gezeigt. Der Zeitversatz wird mit Hilfe dieser Funktion bestimmt und im Parameter `*delta_t` an den Aufrufer zurückgegeben. Der Funktion wird im Parameter `timeout` eine Timeout-Zeit übergeben. Es wird solange versucht, ein Paket von der Karte zu lesen (Polling auf das UpListPtr-Register, vgl. 4.2), bis diese Zeit abgelaufen ist oder ein Paket empfangen wurde. Die Funktion `read_timestamp()` liefert dabei den Inhalt des Pentium-Timer-Registers zurück. In diesem Register werden die Takte der CPU-Clock gezählt und stellen so die feingranularste Zeitbasis des PC dar. Der Aufrufer rechnet dann über die CPU-Taktrate den Wert auf Nanosekunden um.

```
int RT_Protocol_Receive_Buffer(struct sk_buff** skb, long int
*delta_t, long int timeout)
{
  long int t1;
  int err;

  // Auslesen des Pentium Timer Registers
  t1 = read_timestamp();

  do{
    err = RT_MAC_Receive_Buffer(skb);
    *delta_t = read_timestamp()-t1;
  }while((err < 0) && ((*delta_t) < timeout));

  if ((*delta_t) >= timeout)
    return PROTOCOL_TIMEOUT;
  else
    return err;
}
```

**Abbildung 4.8:** Implementierung der Empfangsprozedur in *3c_protocol.c*

Aus dieser Implementierung des Zeitversatzdetektors ergibt sich seine Kennlinie, die in Abbildung 4.9 gezeigt wird. Der Wert $\alpha T$ wird bestimmt durch die Timeout-Zeit. Dies ist der maximale Zeitversatz, der erkannt werden kann. Sinnvollerweise wird diese Zeit in Abhängigkeit von der Basisperiode T des RT-Threads angegeben. Im vorliegenden Fall liegt $\alpha$ bei 0,6 in der Synchronisationsphase und bei 0,4 während der Run-Phase des Protokolls. Jedes Paket, das nach $\alpha T$ eintrifft, wird erst beim nächsten Aufruf von `RT_Protocol_Receive_Buffer` erkannt. Dabei ist zu beachten, dass der Zeitversatzdetektor eine Totzeit $\Delta t_t$ hat, die keinen korrekten Wert für den Zeitversatz darstellt. Diese Zeit wird bestimmt durch die Laufzeit der Funktion, d.h. das Auslesen des Timer-Registers und den Aufruf der MAC-Empfangsroutine. Sie lag im verwendeten Versuchsaufbau bei ca. 5µs.

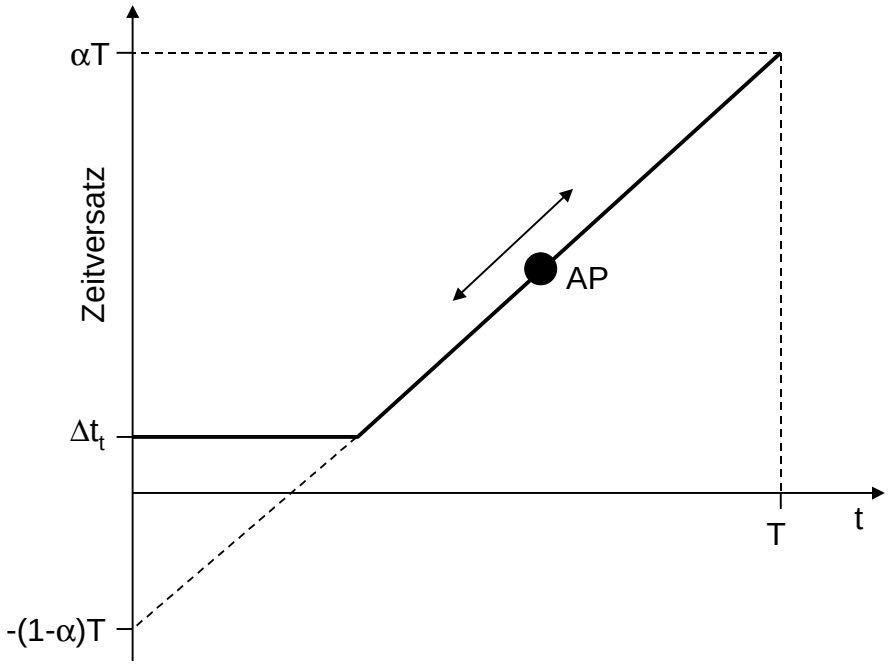



**Abbildung 4.9:** Kennlinie des Zeitversatzdetektors

Der Arbeitspunkt des Zeitversatzdetektors (AP), also der Soll-Zeitversatz als Führungsgröße, muß so gewählt werden, dass der Zeitversatzdetektor mit maximaler Aussteuerung im linearen Bereich arbeitet. Zu Beginn der Synchronisationsphase wird der Sollwert auf 20µs festgesetzt, ein Wert mit dem jeder Rechner richtig arbeiten sollte. Während der Synchronisation wird dann der korrekte Wert bestimmt, da dieser hardwareabhängig ist.

Die Bestimmung des Arbeitspunktes ist im hohen Maße vom Timing im Netzwerk abhängig. Nach dem Start des Treibers steht die Sollvorgabe für den Zeitversatz auf 20µs. Die Periode des RT-Threads wird so eingeregelt, dass alle Pakete 20µs nach dem jeweiligen Ausführungszeitpunkt eintreffen. Zu dieser Zeit weiss der Treiber allerdings noch nicht, wie lang die Laufzeit eines Paketes von einem Netzknoten zu einem anderen ist. Die gewünschte Sollvorgabe für den Zeitversatz muss aber genau diesen Wert haben, damit das Paket den Empfänger rechtzeitig erreicht. Abbildung 4.10 zeigt den Zustand des Treibers in dieser Phase. Nachdem der Client, also ein Netzknoten, der sich neu zum Netz hinzuschaltet, seinen Zeitversatz auf konstant 20µs eingeregelt hat, sendet er als Antwort auf ein Synchronisationspaket *sync* des Master eine Antwort *resync* an den Master. Dieser misst seinerseits den Zeitverzug, mit dem das *resync*-Paket einrifft. Ausgehend davon, dass der Client seinen Versatz auf 20µs eingeregelt hat, kann er die Paketlaufzeit auf dem Netz bestimmen. Der Zeitversatz zwischen Master und Client beträgt x µs (vgl. Abbildung 4.10). Sendet der Client sein *resync*-Paket und geht man von einer Laufzeit von ebenfalls $20\mu s + x$ aus, so ist der Zeitversatz (Offset), den der Master misst, folgender[1].

$$ofs = 20\mu s + 2x \tag{4.8}$$

1. Es ist zu beachten, dass bei allen Berechnungen der Zeitversatz x auch negativ sein kann.

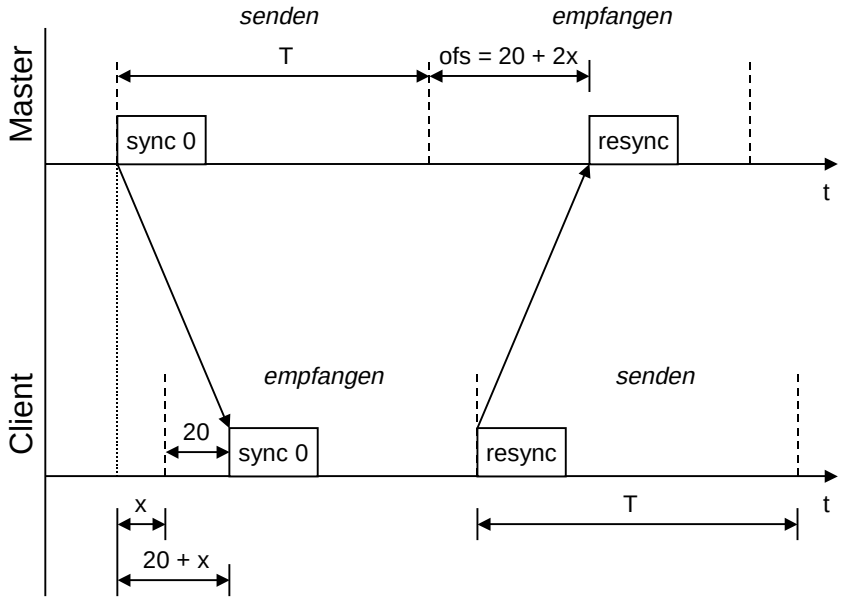



**Abbildung 4.10:** Bestimmung des Arbeitspunktes des Zeitversatzdetektors

Der Arbeitspunkt des Zeitversatzdetektors, also die Sollvorgabe für den Regler ist demnach.

$$\Delta t_{AP} = 20\mu s + \frac{ofs - 20\mu s}{2} \tag{4.9}$$

Bei dieser Berechnung wird immer davon ausgegangen, dass sich die Zeitspanne für das Senden vom Master zum Client und zurück, nicht verändert.

Mit dem nächsten *sync*-Paket sendet der Master den berechneten Arbeitspunkt für den Zeitversatzdetektor an den Client. Dieser ändert daraufhin die Sollvorgabe für seinen Regler und regelt die Periode des Threads darauf ein. Nachdem dies geschehen ist, schaltet der Treiber in den Modus *run* und nimmt am Netzwerk teil. Die Erkennung, ob der Regler die Sollvorgabe eingeregelt hat, erfolgt über ein Toleranzband von $\pm 3\mu s$. Liegt die Stellgröße 1000 mal hintereinander in diesem Toleranzband, so geht der Treiber von einer erfolgten Synchronisation aus.

Um auf unvorhergesehene Ereignisse in fehlertoleranter Art und Weise reagieren zu können, wurde noch eine Fensterung und Mittelung der Zeitversatzwerte vorgenommen. Dabei kam der in der Literatur als FTA-Algorithmus (**F**ault **T**olerant **A**verage) bezeichnete Algorithmus zum Einsatz [9][21] (vgl. Abschnitt 3.4). Hierbei wird jeweils der arithmetische Mittelwert der letzten Werte (hier 10) bestimmt. Dabei werden der größte und der kleinste Wert nicht beachtet.

$$e_{FTA} = \frac{1}{N-2} \sum_{k=0}^{N-2} e_k \qquad e_k \in M,\ M = \{e_i|_{min,max}, 1 \le i \le N\} \tag{4.10}$$

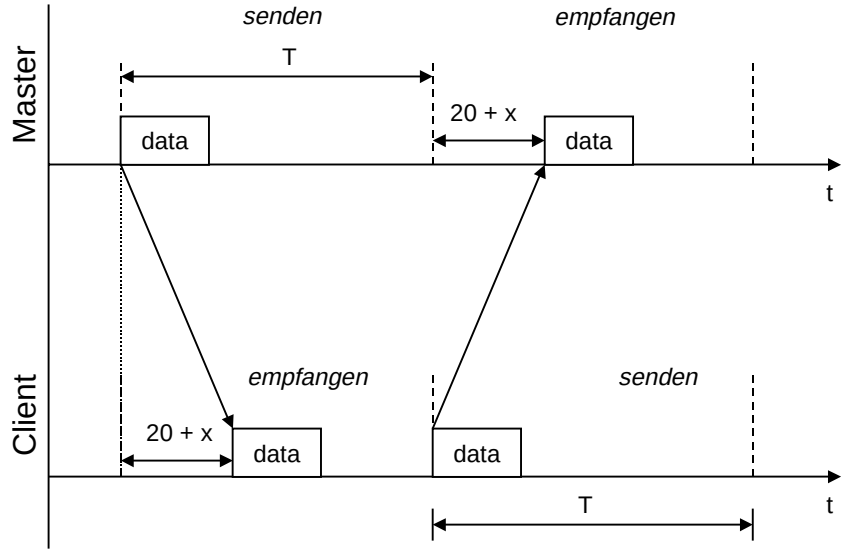



**Abbildung 4.11:** Timing im Netzwerk nach erfolgter Synchronisierung des Client

In der Literatur wird die Mittelwertbildung jedoch über die Netzteilnehmerzahl gebildet. Befinden sich also N Knoten im Netz, so wird der Zeitversatz von jedem Knoten zur lokalen Uhr bestimmt. Anschließend wird auch hier der größte und der kleinste Wert gelöscht, um danach den arithmetischen Mittelwert zu bilden. In unserem Fall ist es jedoch günstiger, die Mittelwertbildung über die letzten Werte durchzuführen. So erhält man nach jedem empfangenen Paket einen neuen Wert für den Zeitversatz. Außerdem können so auch temporär vorhandene Fehler bei wenigen Netzteilnehmern toleriert werden. Da bei dem benutzten Protokoll, bei dem die Netzknoten nacheinander senden, ebenfalls alle Teilnehmer in die Mittelwertbildung eingehen, gelten für dieses Verfahren auch alle Aussagen und Beweise bezüglich der Fehlertoleranz.

## 4.4 Echtzeit-Protokoll

Aufgabe des Protokolls ist es, die Zeitscheiben unter den Netzteilnehmern aufzuteilen. Dabei bekommt jeder die gleiche Bandbreite zugeteilt. Die Aufteilung der Zeitscheiben ist in Abbildung 4.12 dargestellt.

### 4.4.1 Protokollaufbau

Jedem Knoten im Netz wird eine eindeutige Knotennummer zugeteilt. Sie liegt zwischen 1 und 255. Der Master hat immer die Knotennummer 1. Zu Beginn eines jeden Zyklus schickt der Master ein SYNC Signal. Ein Byte dieses Signals enthält die aktuelle Anzahl der Knoten im Netz, ein weiteres Byte bestimmt den Knoten, der als nächstes eine Antwort (RESYNC) auf das SYNC Paket senden muss. Es folgt dann eine Zeitscheibe, in der der Master ein Nutzpaket sendet. Anschließend erhält jeder Client eine Zeitscheibe zum Absetzen seines Nutzpa-

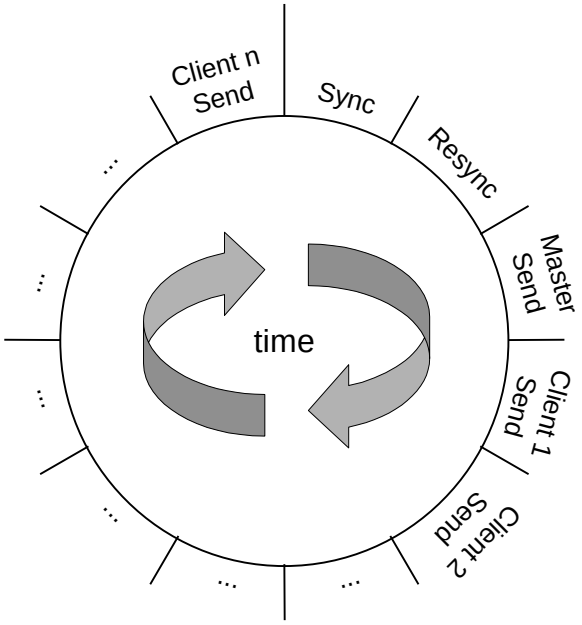



**Abbildung 4.12:** Aufteilung der Zeitscheiben

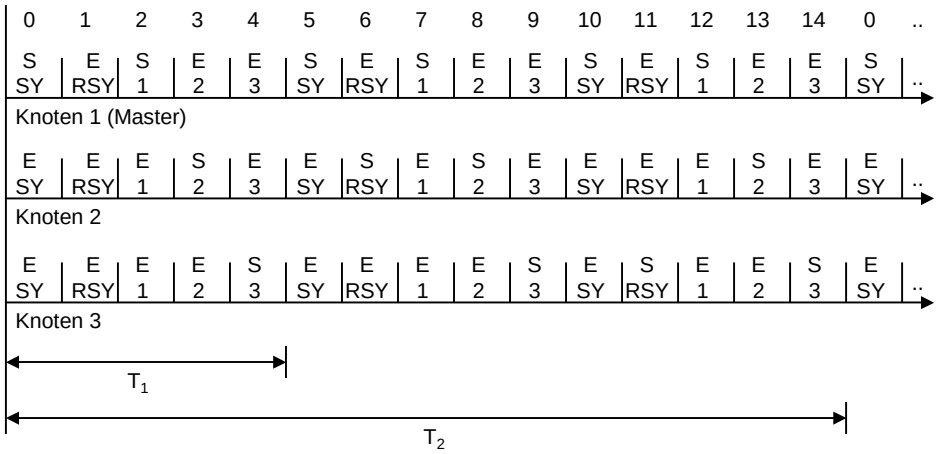
ketes. Während ein Netzknoten sendet, müssen alle anderen Netzknoten empfangsbereit sein. Die Zeit, nach der sich dieser Zyklus wiederholt, ist also von der Anzahl der Knoten im Netz abhängig und beträgt

$$T_1 = (n + 2)\text{timeslots}$$

Da aber in jedem Zyklus nur ein RESYNC Signal vorgesehen ist, kann jeweils nur ein Netzknoten abgefragt werden, ob er noch bereit ist. Der Master fragt mit Hilfe des SYNC Signals nacheinander jeden Netzknoten auf seine Bereitschaft ab. Daraus ergibt sich eine zweite Periodenlänge $T_2$, die die Zeit zwischen zwei RESYNC-Signalen eines Knotens angibt. Die Zeitscheibe für das RESYNC Signal von Knoten 1 (dem Master) wird für neue Netzknoten reserviert. Im Normalfall sendet in dieser Zeitscheibe kein Knoten ein Signal, da es keinen Sinn macht, dass der Master auf sein eigenes SYNC Signal antwortet. Will sich aber ein neuer Knoten mit dem Netz verbinden, so schickt er in dieser Zeitscheibe ein RESYNC Signal. Dieses Signal wird vom Master erkannt und er teilt allen anderen den neuen Netzteilnehmer mit, indem er die Knotenanzahl erhöht. Diese zweite Periodenlänge ergibt sich ebenfalls abhängig von der Anzahl der Knoten im Netz zu

$$T_2 = (n^2 + 2n)\text{timeslots}$$

Die Aufteilung der Zeitscheiben für mehrere Knoten ist in Abbildung 4.13 im Beispiel für 3 Knoten dargestellt.




**Legende:**

| | | |
|---|---|---|
| S SY | Senden | SYNC |
| E SY | Empfangen | SYNC |
| S RSY | Senden | RESYNC |
| E RSY | Empfangen | RESYNC |
| S n | Senden | Knoten n |
| E n | Empfangen | Knoten n |

$n = 3$

$T_1 = n + 2 = 5$
$T_2 = n^2 + 2n = 15$

**Abbildung 4.13:** Aufteilung der Zeitscheiben im Beispiel für 3 Knoten

Die Zuteilung, was in jeder Zeitscheibe getan werden soll, wird zu Beginn jeder Zeitscheibe in Abhängigkeit von der Knotennummer und der im Netz vorhandenen Knoten berechnet. Diese Berechnung wird nur durch das Fortschreiten der Zeit bestimmt. Eine Zuteilung z. B. nach dem Prinzip „Wenn SYNC erhalten, sende RESYNC“ erfolgt nicht, da dies einer Ereignissteuerung entspräche, die im vorliegenden Fall aber vermieden werden sollte. Die praktische Umsetzung dieser Berechnung wird weiter unten dargestellt.

### 4.4.2 Implementation durch State-Machines

Das gesamte Protokoll konnte in einer State-Machine zusammen gefasst werden. Diese besteht aus den drei großen Zuständen

- PROTOCOL_INIT
- PROTOCOL_SYNCHRONIZE
- PROTOCOL_RUN

Abbildung 4.14 zeigt das Zustands-Übergangs-Diagramm.

In jedem Zustand wird nach *Master-Mode* und *Not-Master-Mode* unterschieden. Beim Hinzuladen des Treibers zum Kernel wird durch einen Parameter festgelegt, ob der Treiber als Master arbeiten soll oder nicht[1]. Im *Master-Mode* bestimmt der Knoten den Takt im Netzwerk während im *Not-Master-Mode* die Regelung auf den Netztakt greift. Im Synchronisations-Zustand hat der Master also nichts zu tun.

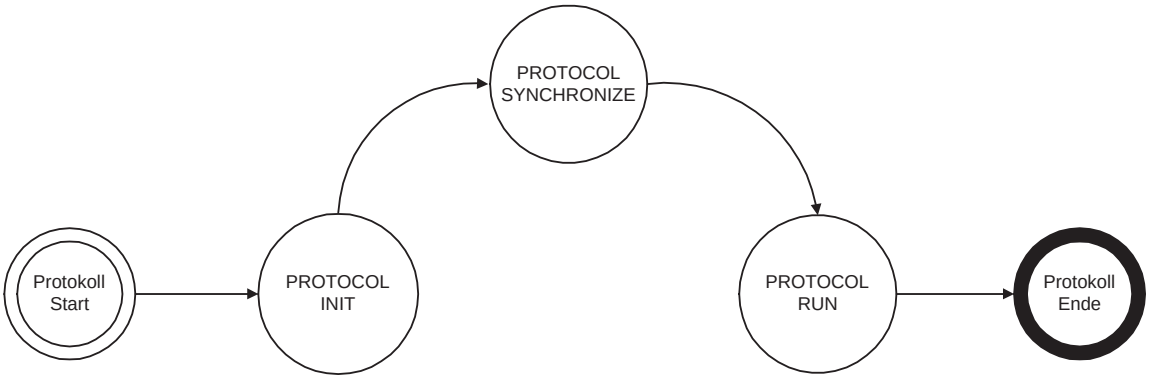



**Abbildung 4.14:** State-Machine des Echtzeit-Protokolls

### Synchronisationsphase

Befindet sich der Master im Zustand PROTOCOL_RUN, sendet er in zyklischen Abständen Synchronisationspakete. Diese werden von den Clients im Zustand PROTOCOL_SYNCHRONIZE ausgewertet. Dieser Synchronisationsvorgang wurde eben-

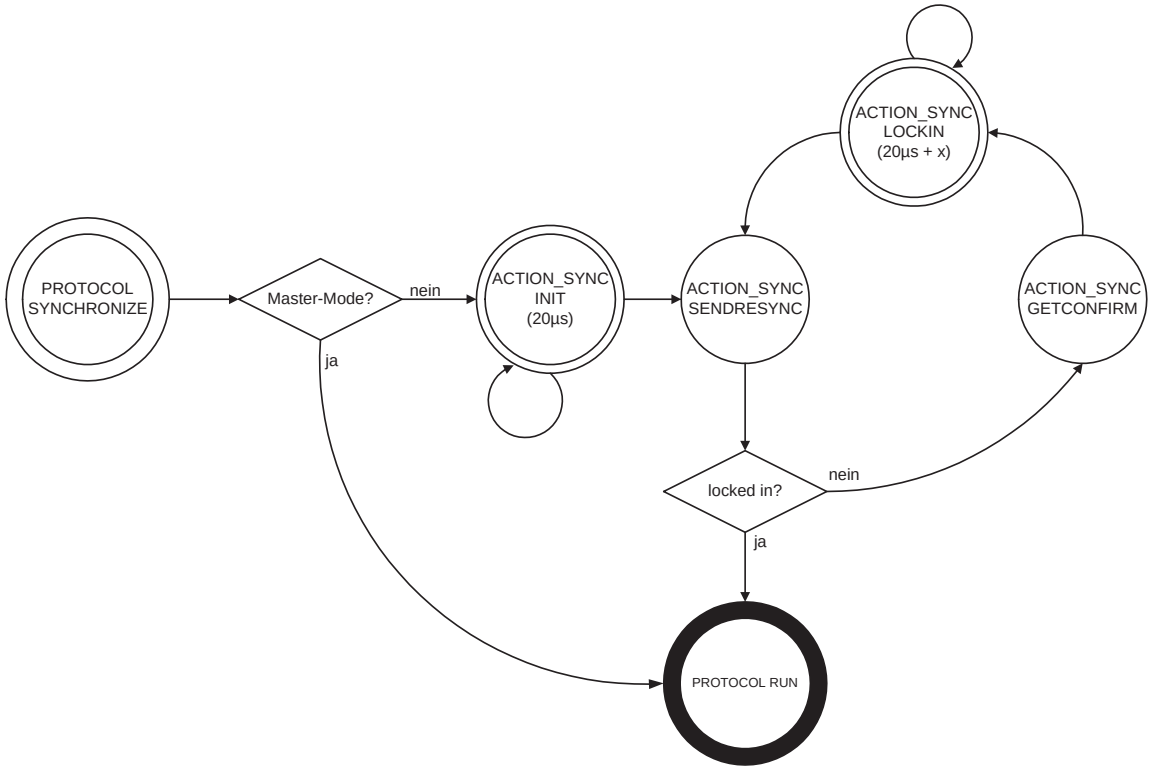



**Abbildung 4.15:** State-Machine in der Synchronisationsphase

falls in einer State-Machine abgebildet (Abbildung 4.15). Zunächst regelt der Client einen Zeitversatz von 20 µs fix ein. Wenn dieser Zeitversatz eine bestimmte Zeit lang im Toleranz-

1. In zukünftigen Implementierungen sollte hier eine dynamische Master-Zuweisung eingeführt werden.

band gehalten wurde, schaltet die State-Machine in den Zustand ACTION_SYNC_SENDRESYNC in der dafür vorgesehenen Zeitscheibe und sendet ein RESYNC Signal. Als Knotennummer wird hier der Wert 0 angegeben. Dieser Wert zeigt dem Master an, dass nur der Offset bestimmt werden soll, der Knoten aber noch nicht bereit ist, sich dem Netz aufzuschalten. Der Master empfängt dieses RESYNC-Paket und misst den zeitlichen Versatz des Paketes zur globalen Zeit. Daraufhin schickt er in seiner nächsten Synchronisationszeitscheibe den gemessenen Versatz, den dieser im State ACTION_SYNC_GETCONFIRM erwartet. Nach dem Empfang dieses Paketes schaltet der Client in den Zustand ACTION_SYNC_LOCKIN, in dem er den Zeitversatz einregelt, den der Master ihm mitgeteilt hat. Wenn auch dieser in dem Toleranzband gehalten werden kann, schaltet die Statemachine erneut in den Zustand ACTION_SYNC_SENDRESYNC und sendet in der für neue Netzknoten vorgesehenen Zeitscheibe ein RESYNC-Paket. Diesmal wird jedoch der Wert *Anzahl der Netzknoten + 1* als Knotennummer angegeben. Das interpretiert der Master als Zeichen für die Bereitschaft des Knotens, sich dem Netz aufzuschalten. Daraufhin inkrementiert der Master die Anzahl der Knoten im Netz bei seinem nächsten SYNC-Paket. Ab diesem Zeitpunkt wird für den neuen Knoten dann eine Zeitscheibe reserviert, in der er senden darf. Die Nummer des neuen Knotens entspricht jetzt der neuen Anzahl der Knoten im Netz. Der neue Knoten wird also immer hinten an die Zuteilungsliste angehängt.

**Run-Phase**

In der Run-Phase ist die Ablaufsteuerung für den „normalen“ Betrieb, in dem Nutzdaten versandt werden, ebenfalls als State-Machine implementiert. Das zugehörige Zustands-Übergangs-Diagramm ist in Abbildung 4.16 dargestellt. Jeder Zustand ist genau eine Zeitscheibe

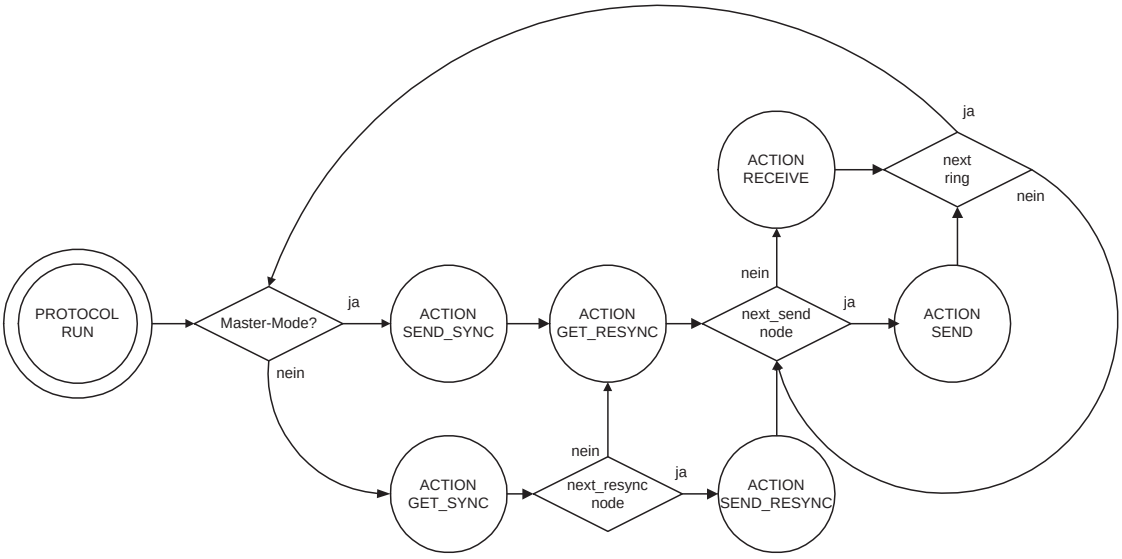



**Abbildung 4.16:** State Machine der Run-Phase des Protokolls

lang aktiv. Er wird zu Beginn jeder Zeitscheibe durch einen Zuteilungsalgorithmus in Abhängigkeit vom Zeitscheibenzähler `ctr`, der Knotennummer `Nr` und der Anzahl der Knoten `N` im Netz bestimmt. Der Zeitscheibenzähler kann dabei Werte zwischen 0 und $(T_2 - 1)$ annehmen; er wird unter Benutzung der Modulo-Operation inkrementiert:

```
ring_count = (++ring_count) % ring_size;
```

In der Funktion

```
static int Calc_next_action(int ctr, int Nr, int N);
```

ist der Zuteilungsalgorithmus implementiert. Sie liefert als Rückgabewert den Status der State-Machine für die nächste Zeitscheibe (vgl. Abbildung 4.17).

```
static int Calc_next_action(int ctr, int Nr, int N)
{
  int c = ctr % (N+2); // N+2 ist die innere Periode T1

  switch(c){
    case 0:
      // SYNC Timeslot
      if (Nr == 1)
        return ACTION_SEND_SYNC;
      else
        return ACTION_GET_SYNC;
      break;

    case 1:
      // RESYNC Timeslot
      if (Nr == 1)
        return ACTION_GET_RESYNC;
      if (ctr == (Nr-1)*(N+2)+1)
        return ACTION_SEND_RESYNC;
      else
        return ACTION_GET_RESYNC;
      break;

    default:
      // Senden oder Empfangen
      if (c == (Nr+1))
        return ACTION_SEND;
      else
        return ACTION_RECEIVE;
      break;
  }
}
```

**Abbildung 4.17:** Implementierung des Zuteilungsalgorithmus für die Run-Phase

Zu Beginn jeder inneren Periode $T_1$ sendet der Master im Status ACTION_SEND_SYNC ein Synchronistaionspaket (SYNC). Es enthält die Anzahl der Knoten im Netz und ein Byte,

das angibt, welcher Knoten als nächstes eine Antwort auf das Synchronisationspaket (RESYNC) senden muss. Dieser Knoten (`next_resync_node`) wird wie folgt berechnet:

```
next_resync_node = (ring_count / (node_counter+2)) + 1;
```

Jeder Client muss während dieser Phase bereit sein, ein SYNC-Paket zu empfangen. Er befindet sich dazu im Status ACTION_GET_SYNC. Nach dem Empfang des Pakets vergleicht er seine Knotenanzahl mit dem Wert im SYNC-Paket und verändert gegebenenfalls seinen internen Wert. Die Knotenanzahl kann sich somit nur zu diesem Zeitpunkt im Protokoll ändern. Empfängt der Client kein SYNC-Paket, so wertet er dies als Fehler im Master und schaltet seine äußere State-Machine zurück in den Zustand PROTOCOL_INIT.

Im nächsten Zustand ACTION_SEND_RESYNC bzw. ACTION_GET_RESYNC sendet ein Client ein Antwortpaket auf das Synchronisationspaket. Diese Antwort enthält die Nummer des sendenden Knotens. Der Master und die restlichen Clients vergleichen diese Nummer mit der Zahl, die das `next_resync_node`-Feld des SYNC-Paket enthielt. Wird kein Paket empfangen oder stimmen die Knotennummer nicht überein, so wird ein Fehler erkannt und die Nummer des entsprechenden Knotens zwischengespeichert. Im Falle eines Fehlers dekrementiert der Master in der nächsten SYNC-Phase seine Anzahl der Knoten im Netz und teilt diese neue Anzahl den restlichen Clients mit. Diese haben ebenfalls die fehlerhafte Knotennummer gespeichert und ändern ihre Anzahl der Knoten entsprechend dem SYNC-Paket. Falls die fehlerhafte Knotennummer kleiner ist als die eigene Nummer, so wird die eigene Nummer dekrementiert. Das stellt sicher, dass keine Löcher in der Knotenliste entstehen. Es sind also immer alle Knotennummern von 1 bis Knotenanzahl besetzt. Die erste Zeitscheibe in der äußeren Periode $T_2$ ist, wie bereits bei der Beschreibung der Synchronisationsphase erwähnt, für neue Knoten reserviert. Deshalb erfolgt die Messung des zeitlichen Versatzes für einen neuen Knoten durch den Master ebenfalls in diesem Zustand ACTION_GET_RESYNC, wenn `next_resync_node` den Wert 1 enthält. Der gemessene Offset wird zwischengespeichert und dem nächsten SYNC-Paket angehängt. Dass nur der Offset gemessen werden soll, zeigt der neue Knoten dadurch an, dass er das Byte für die Knotennummer im RESYNC-Paket auf den Wert 0 setzt. Ist der neue Knoten bereit sich dem Netz aufzuschalten, so setzt er dieses Byte auf den Wert Knotenanzahl + 1 (s.o.). Der Master inkrementiert daraufhin im nächsten Synchronisationszustand seine Anzahl der Knoten im Netz und teilt dieses den restlichen Clients mit. Dieser Zustand ACTION_SEND_RESYNC bzw. ACTION_GET_RESYNC ist somit sehr wichtig für die Fehlererkennung und die Veränderung der Knotenanzahl im Netz.

Der Zustand ACTION_SEND ist für das Versenden von Nutzdaten vorgesehen. Es wird hier das Paket mit der höchsten Priorität der Prioritätswarteschlange entnommen und versandt. Falls kein Paket bereitlag wird ein Dummy-Paket verschickt. Alle anderen Netzteilnehmer sind in dieser Zeitscheibe im Zustand ACTION_RECEIVE und empfangen ein Paket. Da in jeder Zeitscheibe ein Paket verschickt wird, kann der gemessene Offset als Eingangsgröße für die Regelung benutzt werden. Nach dem Empfang eines Paketes wird anhand des Protokollbytes und der Priorität bestimmt, für welche der drei möglichen Schnittstellen das Paket bestimmt ist. Dabei gilt folgende Hierarchie:

- Echtzeitanwendung: Protokollfeld hat die Echtzeitkennung; die Priorität ist größer 1
- Device Schnittstelle: Protokollfeld hat die Echtzeitkennung; die Priorität ist gleich 1
- Linux Bereich (IP-Paket): Protokollfeld hat keine Echtzeitkennung

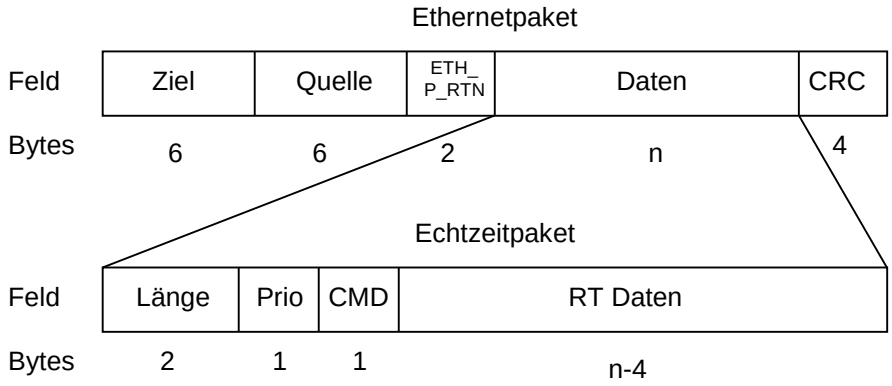
Die Pakete werden dann entsprechend in die Empfangs-Prioritätswarteschlange eingeordnet.

### 4.4.3 Datenformate

Im folgenden soll das Datenformat der unterschiedlichen Pakete sowie der Schnittstellen für den Treiber dargestellt werden. Da es sich um ein Ethernet-Netzwerk handelt, haben alle Pakete den gleichen äußeren Aufbau nach IEEE 802.3 mit einem 14 Byte Protokollkopf und einem 4 Byte Checksummen-Feld (vgl. Abbildung 3.9 auf Seite 24). Werden über den Treiber Echtzeit-Datenpakete versandt, so werden diese durch einen bestimmten Wert im Protokollfeld des Ethernetkopfes eindeutig gekennzeichnet. Dazu wurde die Konstante ETH_RTN wie folgt definiert:

```
#define ETH_P_RTN 0x60FF // Protokoll-Nr für RT-Pakete
```

Der Treiber unterscheidet prinzipiell nach RT Paketen und nicht RT Paketen. Dabei werden nicht RT Pakete werden ohne Vorverarbeitung an den Linux-Kern weitergegeben. Bei den Echtzeitpaketen erwartet der Treiber ein bestimmtes Datenformat der Nutzbytes. Hier wird die Länge des Paketes, seine Priorität, sowie ein Kommando-Byte angegeben. Anhand des Kommando-Bytes (CMD) erkennt der Treiber, ob es sich um ein Paket für die Protokollsteuerung (z.B. SYNC) handelt oder ob das Paket Echtzeitdaten enthält. Der Paketaufbau ist in Abbildung 4.18 dargestellt.




**Werte für CMD:**

| | |
|---|---|
| CMD_SYNC | 0x01 |
| CMD_RESYNC | 0x02 |
| CMD_DUMMY | 0x03 |
| CMD_DATA | 0x10 |

**Abbildung 4.18:** Aufbau der Echtzeitpakete

Aufbauend auf dieser Grundstruktur haben die Pakete zur Protokollsteuerung ebenfalls ein festgelegtes Format. Die inhaltliche Funktion der Felder wurde bereits in Abschnitt 4.4.2 beschrieben. Abbildung 4.19 zeigt das Datenformat der Protokollpakete.

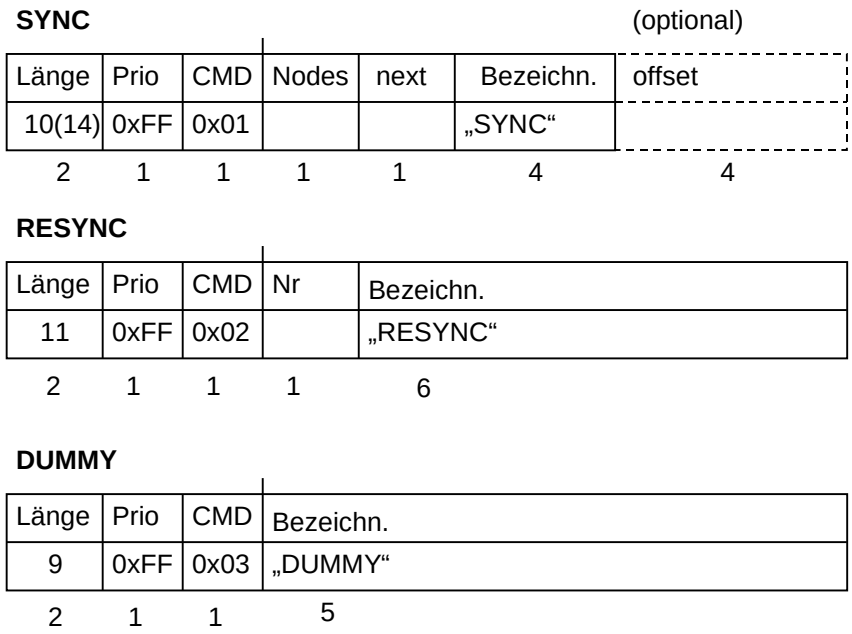



**Abbildung 4.19:** Datenformat der Protokollpakete

Zum Versenden und Empfangen von Datenpaketen stellt der Treiber die Zugriffsfunktionen

```
int RT_Send_Packet(char* packet);

int RT_Receive_Packet(char* Prio,char* packet);
```

zur Verfügung. Bei der Benutzung der Sendefunktion ist zu beachten, dass das grundsätzliche Datenformat für Echtzeitpakete einzuhalten ist, d.h. im übergebenen Paket-String muss der Protokollkopf aus Länge, Priorität und CMD eingehalten werden. Alle Datenpakete sollte CMD_DATA als Kommando-Byte benutzen. Die Prototypen können durch Einbinden der Header-Datei `3c_rt_drv.h` einem Echtzeitmodul bekannt gemacht werden. Auf diese Weise kann das Netzwerk für eine Echtzeitkommunikation genutzt werden.

# 5 Bewertung

Dieses Kapitel dient der Bewertung des vorgestellten Treibers hinsichtlich des Echtzeitverhaltens in Bezug auf die in Abschnitt 3.1 eingeführten Kriterien. Anschließend erfolgt eine Diskussion der Leistungsfähigkeit, um im letzten Teil eine Klassifizierung hinsichtlich der Anwendbarkeit des Netzwerkes im Kontext bestehender Lösungen zu ermöglichen.

## 5.1 Echtzeitverhalten

Zur Bewertung des Echtzeitverhaltens ist an erster Stelle gemäß Abschnitt 3.1 die Protokolllatenzzeit zu nennen. Hierbei muss zwischen zwei Zeiten unterschieden werden:

5. Die Zeit, die von dem Aufruf der Sende-Funktion des Treibers auf der einen Seite bis zum Empfang des Pakets durch eine Applikation auf der anderen Seite vergeht.

6. Die Zeit, die der Treiber benötigt um eine Nachricht an einen anderen Knoten zu übergeben.

Im ersten Fall wird die Dauer im wesentlichen durch das Protokoll bestimmt. Wird ein Netzwerk mit zwei Knoten betrachtet, so ist für jeden Knoten alle 4 ms eine Zeitscheibe zum Versenden eines Pakets vorgesehen. Das Gleiche gilt für den Empfang. Daher liegt im schlimmsten Fall die Übertragungsdauer bei 8 ms. Dies ist die garantierte Frist, die der Treiber einhält. Im besten Fall wird das Paket innerhalb einer Zeitscheibe versendet[1]. Somit liegt der Jitter bei ebenfalls 8 ms. Die hier vorgestellte Software-Lösung bietet die Möglichkeit, dieses Verhalten durch eine erweiterte Schnittstelle (API) wesentlich zu verbessern. Es ist denkbar, eine dynamische Vergabe der Zeitscheiben oder Synchronisation der Anwendung mit dem Treiber vorzusehen.

Um die Leistungfähigkeit des Treibers in Bezug auf die Latenzzeit zu bewerten, ist es jedoch wesentlich interessanter die zweite Zeit zu untersuchen, da diese Verhalten auch durch eine optimierte API nicht beeinflusst werden kann. Betrachtet wird die Zeit, die benötigt wird um das Paket der Prioritätswarteschlange des Senders zu entnehmen, über das Netzwerk zu verschicken und dem Puffer des Empfängers zu übergeben. Sie liegt immer innerhalb einer Zeitscheibe. Aus den Ausführungen zur Realisierung der Zeitsteuerung (vgl. Abschnitt 4.3) wird deutlich, dass diese Zeit dem Soll-Zeitversatz für den Regler entspricht (vgl. Abbildung 4.11, „20 + x“). Im vorliegenden Versuchsaufbau lag dieser Wert bei ca. 27 µs. Der Jitter wird im wesentlichen durch das Synchronisationsverhalten der Knoten aufeinander bestimmt. Er lag im Versuchsaufbau bei ca. 3 µs. Eine ausführliche quantitative Untersuchung des Regelverhaltens erfolgt in Abschnitt 5.2.

Um die Zusammensetzbarkeit von mehreren Knoten zu gewährleisten, sind nach Abschnitt 3.1 zwei Grundvoraussetzungen notwendig. Zum einen muss eine temporale Kapselung der Netzknoten unabhängig von der Applikation gegeben sein, zum anderen muss die Verpflichtung der Clients, den Master nicht zu überlasten, erfüllt sein. Beide Voraussetzungen werden von dem Netzwerk erfüllt. Die Zeitsynchronisation erfolgt automatisch innerhalb des Treibers, so dass sie unabhängig von der Applikation arbeitet. Die Forderung, den Master nicht

1. Die Laufzeit auf dem Netzwerk kann in diesem Fall vernachlässigt werden, da sie im Mikrosekundenbereich liegt.

zu überlasten, wird aufgrund der zeitgesteuerten Implementierung erfüllt. Da jeder Client nur in den ihm zugewiesenen Zeitscheiben senden darf, kann es zu keiner Überlastung des Netzwerkes kommen.

Ein weiteres Kriterium ist die Flexibilität. Hiermit wird die Möglichkeit bezeichnet, die Knotenanzahl im Netz, während des laufenden Betriebes zu ändern. Durch das vorgestellte Protokoll ist dies gewährleistet. Nach einem Synchronisationsvorgang kann sich ein neuer Knoten dem Netzwerk aufschalten und am Datenverkehr teilnehmen. Unter Flexibilität ist aber auch eine variable Nutzung des Netzwerkes zu verstehen. Der hier behandelte Treiber bietet, neben einer Nutzung als Echtzeit-Kommunikations-Einheit, zusätzlich die Möglichkeit, Daten aus dem Linux Bereich auszutauschen. Damit wird die Idee von RT Linux fortgeführt, ein Standard-Betriebssytem als Plattform für ein aufgesetztes Echtzeitmodul zu benutzen. Der Standard-Netzwerkzugang wird um einen echtzeitfähigen Zugang erweitert.

Weiterhin ist die Fehlererkennung als Kriterium zu nennen. Der Ansatz dazu ist die Kenntnis der Netzteilnehmerzahl und der eigenen Knotennummer bei jedem Teinehmer. Da der Treiber in jeder Zeitscheibe, in der er die Sendeerlaubnis hat, ein Paket sendet, ist hierdurch die implizite Fehlererkennung eines Knotenausfalls gewährleistet. Empfangen die anderen Netzknoten in dieser Zeitscheibe kein Paket, so gehen sie von einem Fehler in dem entsprechenden Knoten aus und streichen ihn aus der Liste der Netzteilnehmer. Auf zeitliche Fehler wird durch den FTA Algorithmus in toleranter Weise reagiert. Durch die Nichtberücksichtigung des größten und kleinsten Wertes innerhalb des Berechnungsfensters werden einige zeitliche Fehler durch verspätetes Senden eines Knotens toleriert. Die Zeitsynchronisation der übrigen Netzteilnehmer ist nicht gefährdet.

Als ein Schwachpunkt des vorgeschlagenen Protokolls kann das Vorhandensein eines Masters angesehen werden. Fällt der Master aus, so fährt das gesamte Netzwerk herunter. Die Clients reagieren zwar in sicherer Art und Weise und schalten zurück in die INIT-Phase des Protokolls, ein Datenaustausch ist jedoch nicht mehr möglich. Die Struktur des Treibers bietet aber die Möglichkeiten, solche Unzulänglichkeiten in ein zukünftiges Protokoll einfließen zu lassen.

Hinsichtlich der Netzstruktur erfüllt der Treiber die Voraussetzungen. Mit der hier verwendeten Ethernet-Hardware kann eine sternförmige Netzstruktur gut realisiert werden. Im Testaufbau wurden die drei Netzwerkleitungen an einem Hub als Sternpunkt zusammengeführt. Da durch den Promiscuous-Modus, in dem die Netzwerkkarten betrieben werden, jede Nachricht von jedem Knoten empfangen werden, wird ein Fehler in allen Netzknoten gleichzeitig erkannt.

## 5.2 Leistungsfähigkeit

Das zeitliche Verhalten stellt ein wichtiges Maß für die Leistungsfähigkeit des hier vorgestellten, zeitgesteuerten Netzwerks dar. Daher ist eine Bewertung des verwendeten Regelungsansatzes (vgl. Abschnitt 4.3) von großer Bedeutung.

### 5.2.1 Der Zeitversatzdetektor

Der Zeitversatzdetektor liefert den Zeitversatz als Eingangsgröße für den Regler. Seine Kennlinie kann gemessen werden, indem der Regelalgorithmus deaktiviert wird und mit einem Client der Zeitversatz der eintreffenden Pakete, die der Master sendet, gemessen wird. Die

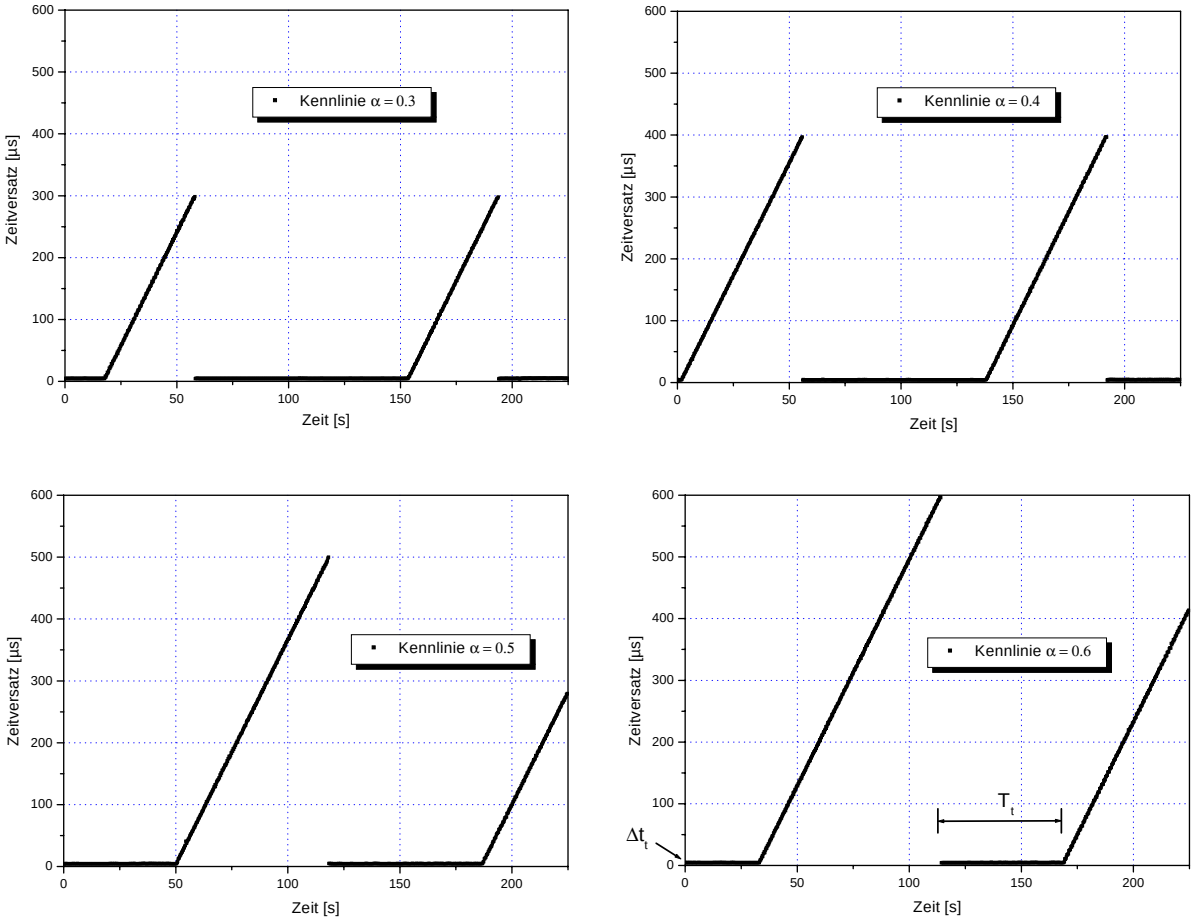



**Abbildung 5.1:** Gemessene Kennlinie des Zeitversatzdetektors in Abhängigkeit von $\alpha$. Die Basisperiode T des RT Threads beträgt 1 ms.

Kennlinie ist in ihrer Abhängigkeit von $\alpha T$ in Abbildung 5.1 dargestellt. Die Totzeit $\Delta t_t$ lag im vorliegenden Fall bei ca. 4,8 µs. Diese Zeit wird vom Zeitversatzdetektor zurückgeliefert, falls das Paket, das empfangen werden sollte, von der Netzwerkkarte bereits in den Arbeitsspeicher geladen wurde. Dieser Bereich, der in der Abbildung mit $T_t$ bezeichnet wurde, ist bei $\alpha = 0,6$ am kleinsten. Deshalb wird der Zeitversatzdetektor in der Synchronisationsphase mit diesem Wert betrieben. In der Zeitscheibe für das RESYNC-Paket wird ebenfalls $\alpha = 0,6$ benutzt, da ein neuer Knoten seinen eigenen Zeitversatz noch nicht kennt. Die bisherigen Knoten im Netzwerk müssen daher mit einer größeren Verzögerung im Eintreffen des Paketes rechnen. Im normalen Betrieb wird der Zeitversatzdetektor mit $\alpha = 0,4$ betrieben. Diese Umschaltung erfolgt, um die Auslastung des Systems im Fehlerfall möglichst gering zu halten. Da im normalen Betrieb von einem eingeschwungenen Regelsystem ausgegangen werden kann und sich der Zeitversatzdetektor damit in einem stabilen Arbeitspunkt befindet, reicht hier der beschränkte lineare Bereich aus.

Anhand der Kennlinie des Zeitversatzdetektors lässt sich aber auch die Driftrate der Rechner zueinander bestimmen. Sie entspricht der Steigung der Kennlinie im linearen Bereich. In der hier gezeigten Messung konnte sie zu 7,31 µs/s bestimmt werden. Die Driftrate lag also bei

$$\rho = 7,31 \cdot 10^{-6}$$

Bezogen auf die CPU-Taktgeschwindigkeit der beiden Rechner von 400 MHz ergibt sich ein Frequenzunterschied der Quarzoszillatoren von 2,9 kHz, das ist weniger als 1 Promille. Es ist die Aufgabe des Reglers, diesen Fehler zu kompensieren.

Zusätzlich lässt sich an der Kennlinie der Einfluss des in Abschnitt 4.3 vorgestellten FTA-Algorithmus aufzeigen. Er hat die Aufgabe, zu verhindern, dass fehlerhafte Knoten das gesamte Netzwerk zum Erliegen bringen. Konkret heißt das, dass z.B. ein großer Zeitversatz durch einen ausgefallenen Knotenn nicht mit in die Regelung eingeht. Gleichzeitig bewirkt der Algorithmus durch die Fensterung und Mittelung der Daten eine Tiefpassfilterung, was zu einer kleineren Varianz der Eingangswerte für die Regelung führt. Anhand von Abbildung 5.2 und Abbildung 5.3 wird der Einfluss der FTA-Filterung deutlich.

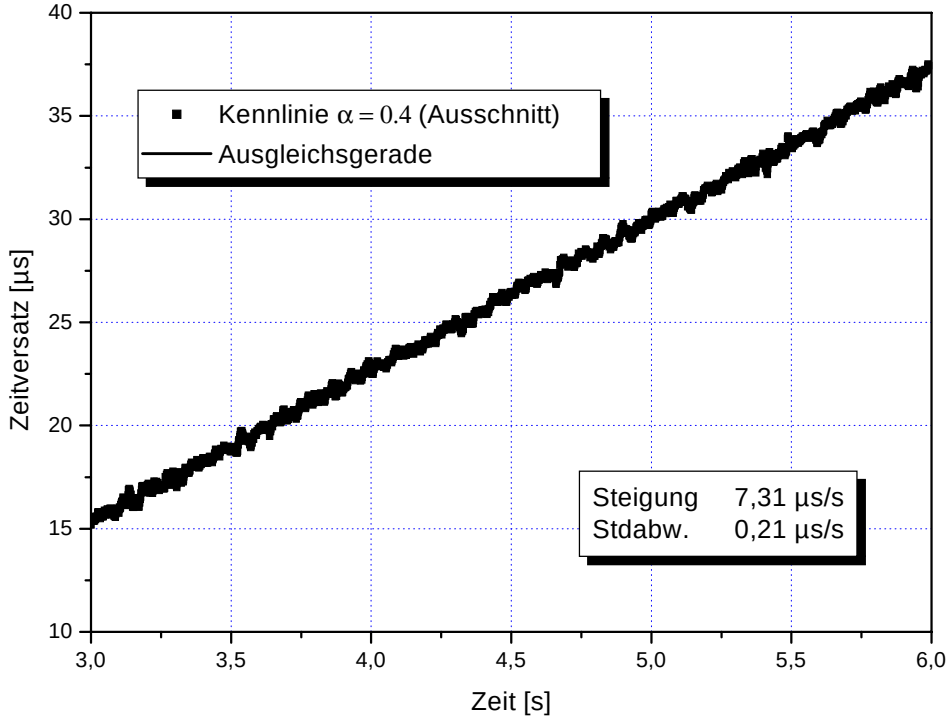



**Abbildung 5.2:** Ausschnitt aus der Kennlinie des Zeitversatzdetektors mit FTA-Filterung

### 5.2.2 Die Regelung

Die Regelparameter mussten für den vorliegenden Regelfall optimiert werden und es war eine Auswahl des eingesetzten Algorithmus (PI oder PID) zu treffen. Die Bewertung eines Reglers erfolgt nach den zwei Kriterien

- Führungsverhalten und
- Störverhalten.

Dabei wird mit dem Führungsverhalten die Reaktion des Reglers auf einen Sollwertsprung bezeichnet. Das Einschaltverhalten kann ebenfalls als Sollwertsprung, nämlich von 0 auf den ersten Sollwert, betrachtet werden. Mit dem Störverhalten wird die Reaktion des Reglers auf störende äußere Einflüsse bezeichnet. Bezogen auf die vorliegende Anwendung ist das Führungsverhalten entscheidend für den Einschwingvorgang eines neuen Knotens auf das Netz-

werk sowie das Einregeln des individuellen Offset. Das Störverhalten hingegen bestimmt die Fähigkeit des Reglers, die Driftrate des Quarzoszillators auszugleichen. Eine Optimierung der Regelparameter kann bei genauer Kenntnis der Regelstrecke rechnerisch und theoretisch bestimmt werden. Dazu ist es erforderlich, ein Modell für die Strecke und die Störeinflüsse zu erstellen. In Kombination mit der Übertragungsfunktion des Reglers kann dann die optimale Lage der Pol- und Nullstellen des aufgeschnittenen Regelkreises, die das Regelverhalten eindeutig charakterisieren, bestimmt werden. Da ein solches Modell in den meisten Fällen jedoch nicht vorliegt, kann eine Bestimmung der Parameter nur empirisch erfolgen. Mit Kenntnis des Einflusses der einzelnen Größen kann aber auch mit diesem Verfahren, was auch im vorliegenden Fall zum Einsatz kam, ein gutes Ergebnis erzielt werden.

Der erste Ansatz zur Realisierung ist ein PI-Regler. Der Integralanteil in der Übertragungsfunktion ist notwendig, da nur so der stationäre Endwert, d.h. die exakte Übereinstimmung von Regelgröße und Führungsgröße erreicht werden kann. Ein reiner Proportionalregler würde immer eine bleibende Regelabweichung ergeben. In der Arbeit wurde jedoch ein PID-Regler verwendet, da mit dieser Reglerstruktur flexibler auf die Eigenschaften der Regelaufgabe eingegangen werden konnte. Der Differenzialanteil bewirkt ein schnelleres Einschwingverhalten sowie ein Stabilisierung des Reglers im Störverhalten. Es sei jedoch erwähnt, dass auch mit einem PI-Regler ein zufriedenstellendes Ergebnis erreicht werden konnte. Da jedoch der minimal höhere Berechnungs- und Speicheraufwand in Anbetracht der hier verwendeten leistungsfähigen Rechner vernachlässigbar war, war der PID-Regler die erste Wahl. Die Diskussion der unterschiedlichen Einflüsse der Regelparameter soll daher auch am

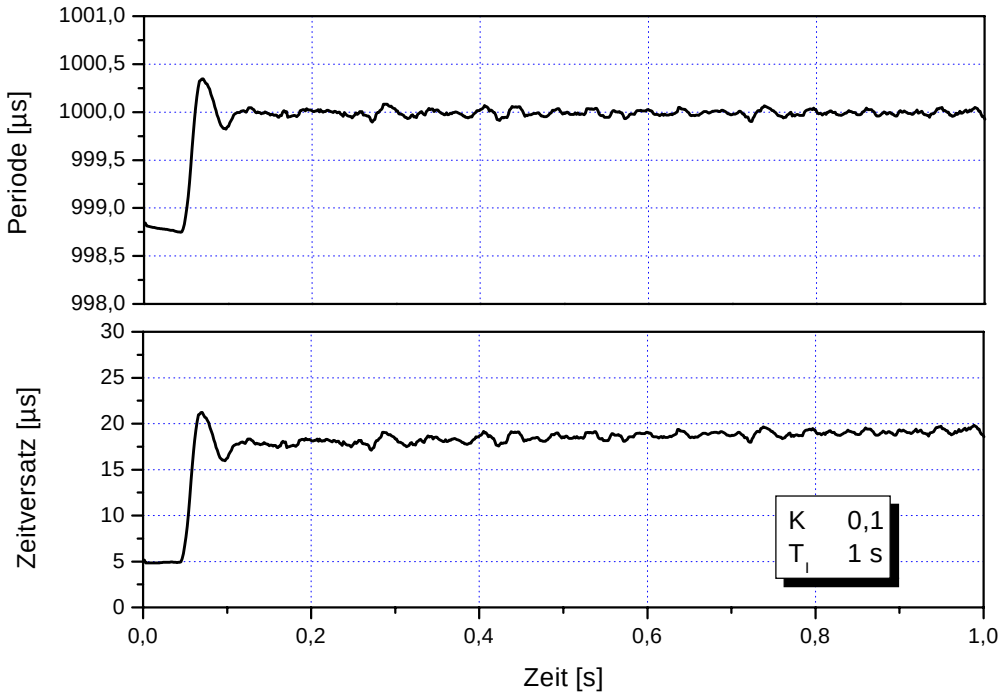



**Abbildung 5.3:** Einschwingverhalten des PI-Reglers mit optimierten Regelparametern

PID-Regler erläutert werden. In Abbildung 5.4 ist das Einschwingverhalten des PID-Reglers mit den am Ende des Optimierungsprozesses gefundenen Regelparametern gezeigt, während in Abbildung 5.3 der gleiche Sachverhalt für den PI-Regler dargestellt ist. Es ist jeweils der

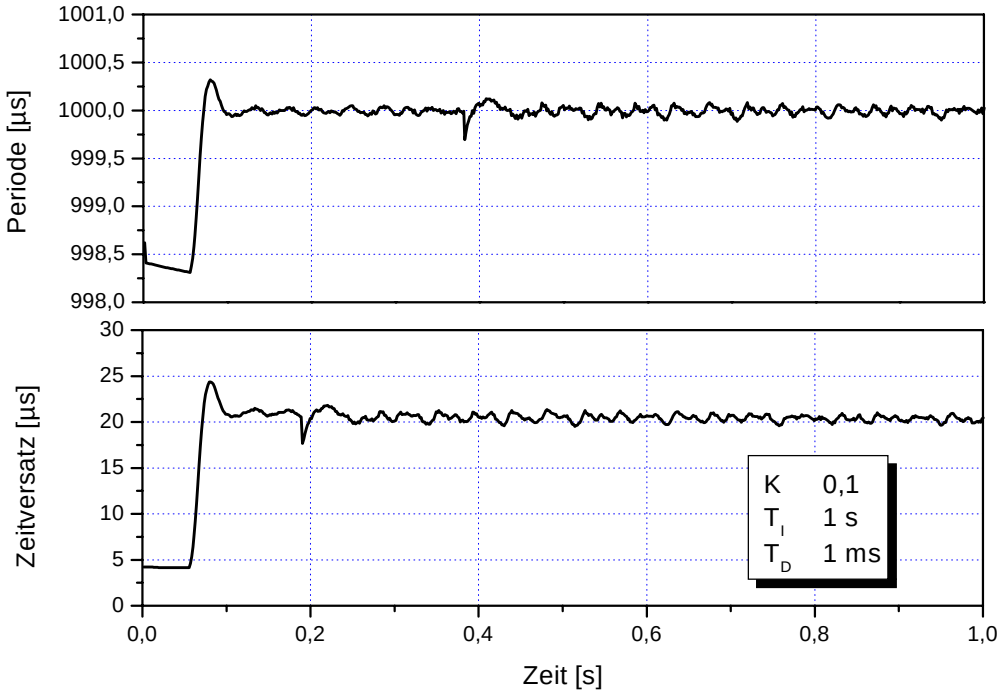



**Abbildung 5.4:** Einschwingverhalten des PID-Regler mit optimierten Regelparametern

Einschwingvorgang des Reglers auf einen Sollwert von 20 µs zu Beginn der Synchronisationsphase eines neuen Knotens dargestellt. Es sei darauf hingewiesen, dass nicht in jedem der im folgenden gezeigten Bilder die gleiche Skalierung benutzt wurde, da nur so eine sinnvolle Darstelllung des vollständigen Einschwingvorganges möglich wurde. Auffallend an allen Darstellungen ist ein kurzer Bereich konstanten Zeitversatzes zu Beginn jedes Einschwingvorganges. Dieser Bereich entspricht der unvermeidlichen Totzeit des Zeitversatzdetektors und ist daher auch von unterschiedlicher Länge, je nachdem in welchem Punkt der Kennlinie des Detektors die Messung begonnen wurde. Der Bereich ist für die Bewertung des Regelverhaltens jedoch unerheblich. Die für die Regelparameter benutzten Symbole K, $T_I$ und $T_D$ entsprechen den in Abschnitt 4.3 eingeführten Bezeichnungen.

Den größten Einfluss auf die Geschwindigkeit eines Reglers hat der Proportionalanteil. Wird ein zu kleiner Wert gewählt, so geht die Regelabweichung nur unzureichend gewichtet in die Berechnung ein und der Regler reagiert sehr langsam auf einen Sollwertsprung. Wird jedoch ein zu großer Wert gewählt, so reagiert der Regler mit heftigen Schwingungen. Dies kann sogar zu einem unstabilen Verhalten führen. Mit einem Wert von K = 0,1 wurden die besten Ergebnisse erzielt. In Abbildung 5.5 wird der Einfluss des Proportionalanteils auf das Einschwingverhalten gezeigt. Bei einem Wert von K = 0,05 wird der stationäre Endwert erst nach fast 3 s erreicht, während dieser Wert bei K = 0,1 in gut einem Zehntel der Zeit erreicht wird (Abbildung 5.4). Eine Verdopplung des Propotionalanteils auf K = 0,2 führt jedoch zu heftigen Überschwingen im Zeitversatz und der Periode.

Als nächstes soll der Einfluss des Integralanteils $T_I$ untersucht werden. Aufgabe dieses Anteils ist es, die Regelabweichung aufzuaddieren, um dadurch am Ende den Sollwert exakt einzuregeln. Wird für $T_I$ eine zu lange Zeit gewählt, so wird dieses Ziel nicht erreicht und es

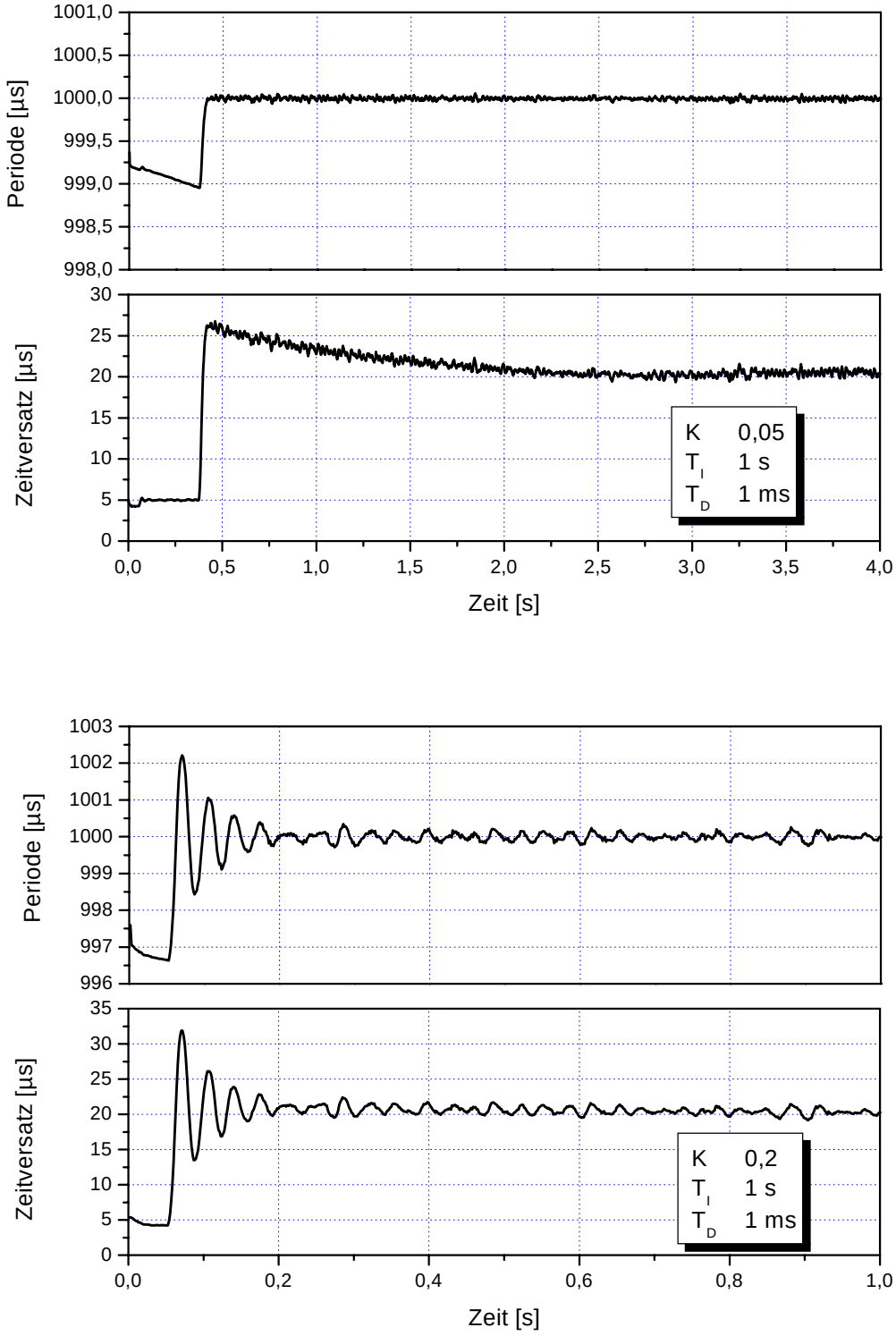



**Abbildung 5.5:** Einfluss des Proportionalanteils K auf das Führungsverhalten des PID-Reglers. Ein zu kleiner Wert verlangsamt den Regler, während ein zu grosser Wert die Schwingneigung erhöht.

bleibt eine Regelabweichung. Wird der Anteil durch einen kleinen Wert für $T_I$ jedoch zu stark gewichtet, so reagiert der Regler zu schnell und es kommt zu großem Überschwingen. In Abbildung 5.6 ist der Einfluss des Integralanteils auf den hier eingesetzten Regler gezeigt. Ein Wert von 1 s wurde nach Bewertung des Störverhaltens (s.u.) als optimal gefunden.

Als letztes soll der positive Einfluss des Differenzialanteils $T_D$ anhand der Abbildung 5.3 und Abbildung 5.4 deutlich gemacht werden. Gegenüber dem PI-Regler kann der PID-Regler den Sollwert schneller erreichen. Der negative Einfluss des Proportional- und Integralanteils auf die Geschwindigkeit kann durch den Differenzialanteil teilweise kompensiert werden. Neigt das Regelsystem zu Schwingungen, so hat der Differenzialanteil eine stabilisierende Wirkung, da er gerade diese meist hochfrequenten Änderungen der Regelgröße ausgleicht. Wird der Einfluss von $T_D$ jedoch zu groß gewichtet, so kann es zu geradezu „hektischen“ Veränderungen der Stellgröße kommen, da der Regler hier auf ein meist unvermeidliches Rauschen im Signal der Messeinrichtung (hier der Zeitversatzdetektor) reagiert. Abbildung 5.7 verdeutlicht diesen Zusammenhang. Um diesen Effekt zu vermeiden, wird oft eine zusätzliche Polstelle in der Übertragungsfunktion eingeführt, die den Einfluss des Differenzialanteils (Nullstelle in der Übertragungsfunktion) bei hohen Frequenzen beschneidet. Da der Differenzialanteil in unserem Fall durch einen Wert von $T_D = 1\text{ms}$ ohnehin nur gering gewichtet wurde, ist diese Kompensation nicht notwendig.

Um das Führungsverhalten auch an einem Sollwertsprung zu verdeutlichen, wird in Abbildung 5.8 der Sprung von 20 µs auf den zugewiesenen Offset während der Synchronisationsphase gezeigt. In diesem Bild ist die Arbeitsweise des PLL-Prinzips sehr gut erkennbar. Eine

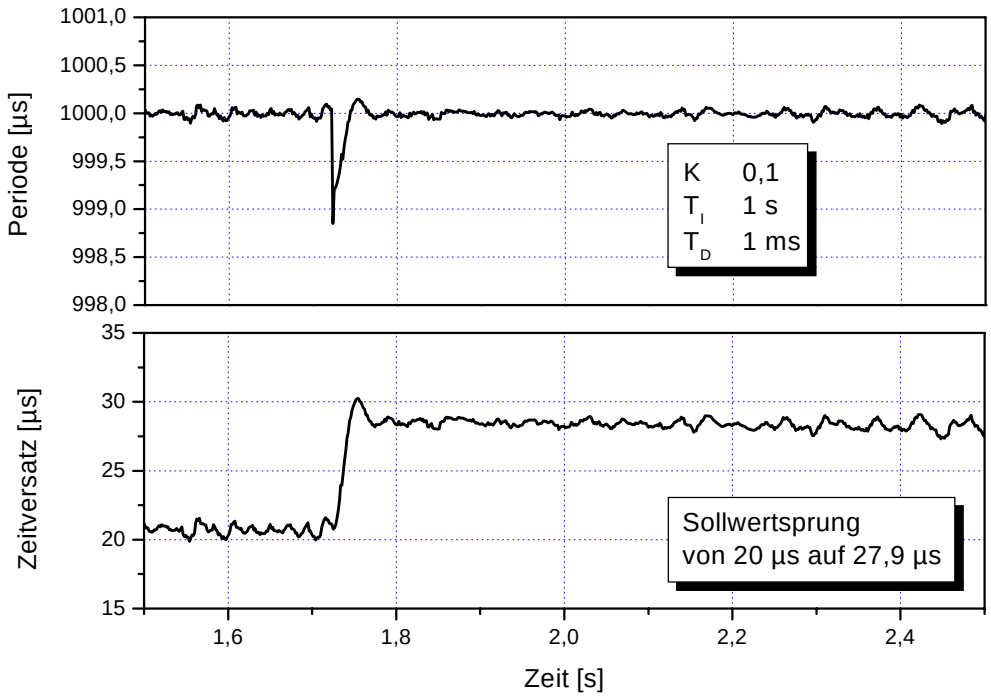
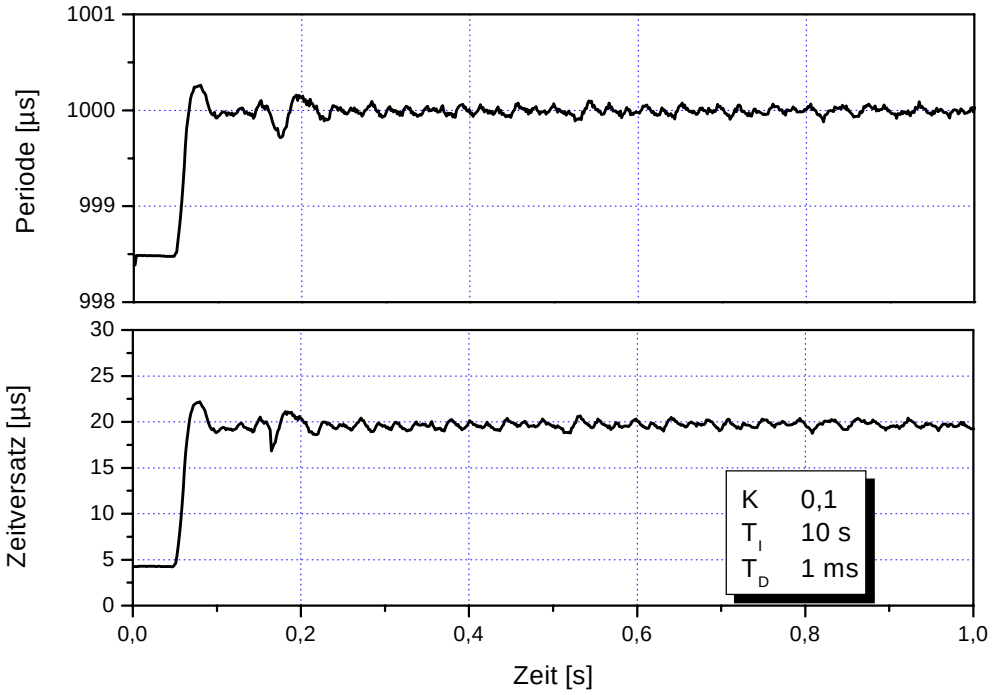



**Abbildung 5.8:** Sollwertsprung während der Synchronisationsphase

Änderung des Sollwertes (Zeitversatz) hat eine kurzfristige Änderung der Regelgröße (Periode) zur Folge. Nach dem Erreichen des neuen Sollwertes wird die Regelgröße jedoch auf den ursprünglichen Wert zurückgeführt.



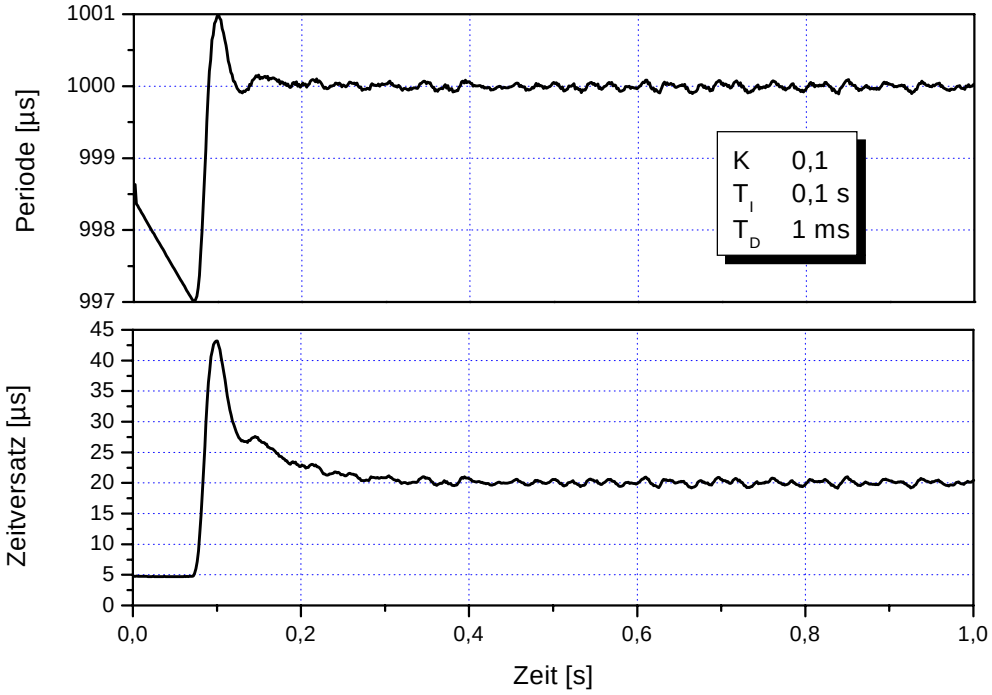
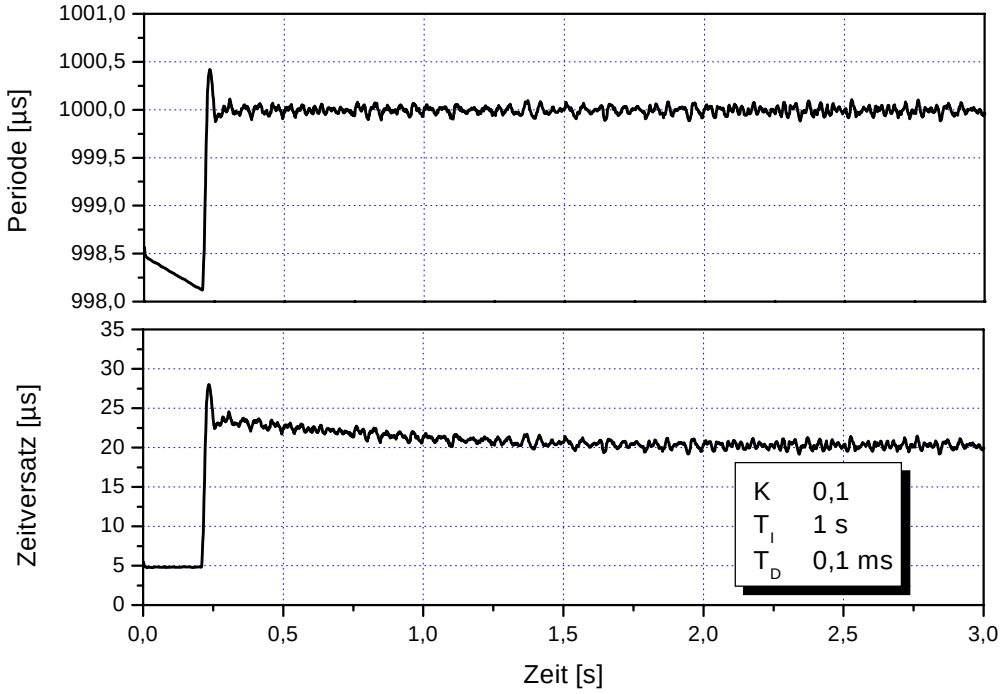




**Abbildung 5.6:** Einfluss des Integralanteils $T_I$ auf das Regelverhalten eines PID-Reglers. Ein zu großer Wert (oben) führt zu einem gut gedämpften System, jedoch mit einer relativ großen Regelabweichung. Ein zu kleiner Wert führt zu großem Überschwingen (unten).



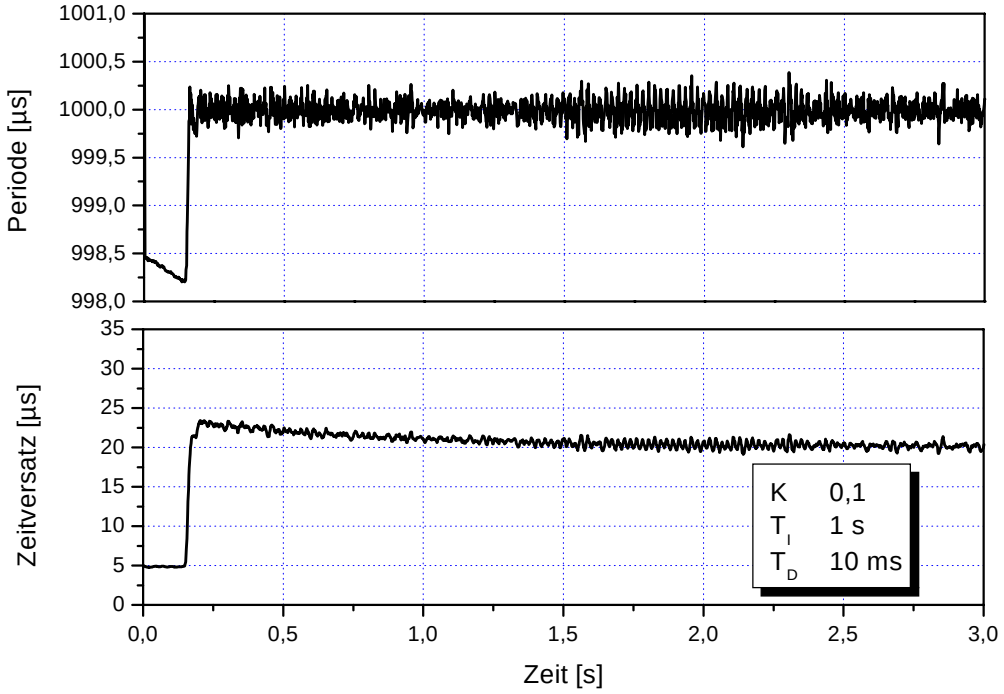




**Abbildung 5.7:** Einfluss des Differenzialanteils auf das Führungsverhalten des PID-Reglers. Während ein zu kleiner Anteil zu keiner Beschleunigung des Einregelvorgangs gegenüber dem PI-Regler führt, kann ein zu großer Anteil den Regler zu hektisch reagieren lassen.

Wie Eingangs erwähnt ist die zweite Größe in der Bewertung eines Regler das Störverhalten. Da sich hierbei die Sollvorgabe nicht ändert, kann dieser Zustand auch als das stationäre Regelverhalten bezeichnet werden. Ziel einer Optimierung des Reglers ist hierbei die möglichst exakte Einregelung der Sollvorgabe, bei einer minimalen Varianz der Regelgröße. Dieses Ziel steht zum Teil im Gegensatz zu dem Ziel einer Abstimmung auf das Führungsverhalten. Beispielsweise führt ein stark gewichteter Integralanteil zu einer sehr exakten Einregelung der Sollvorgabe, erzeugt jedoch großes Überschwingen im Führungsverhalten. Eine gute Reglerabstimmung stellt somit immer einen guten Kompromiss zwischen diesen beiden Zielen dar. Im Verlauf der Messungen hat sich das Störverhalten im Vergleich zum Führungsverhalten als wesentlich unkritischer erwiesen. In Abbildung 5.9 ist das Störverhalten des PID-Reglers im Vergleich zum PI-Regler aufgezeigt. Es ist gut zu erkennen, dass in beiden Fällen ein Toleranzband ± 3 µs in den Werten für den Zeitversatz eingehalten werden kann. Dies ist im Verhältnis zur Periode des RT-Threads von 1 ms ein sehr guter Wert. Um eine quantitative Bewertung des Störverhaltens zu ermöglichen, wird der Mittelwert, als Maß für die Einregelung der Sollvorgabe, und die Standardabweichung, als Maß für die Veränderungen in der Regelgröße, herangezogen. Dabei wurde jeweils ein Zeitraum von 5 s während der Synchronisationsphase untersucht. Während der Messungen bestand das Netzwerk aus zwei Rechnern. Die Sollvorgabe für den Zeitversatz betrug 20 µs und mit der oben bestimmten Driftrate von $\rho = 7,31 \cdot 10^{-6}$ ergab sich für die Periode ein Wert von $T = 999,99269\mu s$, der eingeregelt werden musste. Die Ergebnisse dieser Messung sind in Tabelle 5.1 dargestellt.

| alle Werte in [µs] | **Zeitversatz** Mittelwert | Standard-abw. | **Periode** Mittelwert | Standard-abw. |
|---|---|---|---|---|
| PID-Regler $K = 0,1, T_I = 1s, T_D = 1ms$ | 20,1288 | 0,3178 | 999,9926 | 0,0324 |
| PI-Regler $K = 0,1, T_I = 1s$ | 20,0103 | 0,4056 | 999,9925 | 0,0393 |
| PID-Regler $K = 0,05, T_I = 1s, T_D = 1ms$ | 20,1918 | 0,3516 | 999,9924 | 0,0170 |
| PID-Regler $K = 0,2, T_I = 1s, T_D = 1ms$ | 20,0461 | 0,4464 | 999,9923 | 0,0936 |
| PID-Regler $K = 0,1, T_I = 10s, T_D = 1ms$ | 20,4964 | 0,3751 | 999,9925 | 0,0362 |
| PID-Regler $K = 0,1, T_I = 0,1s, T_D = 1ms$ | 20,0094 | 0,3580 | 999,9925 | 0,00373 |
| PID-Regler $K = 0,1, T_I = 1s, T_D = 0,1ms$ | 20,0717 | 0,3808 | 999,9927 | 0,0360 |
| PID-Regler $K = 0,1, T_I = 1s, T_D = 10ms$ | 20,0052 | 0,3681 | 999,9927 | 0,1074 |

**Tabelle 5.1:** Stationäres Regelverhalten in Abhängigkeit der Regelparameter

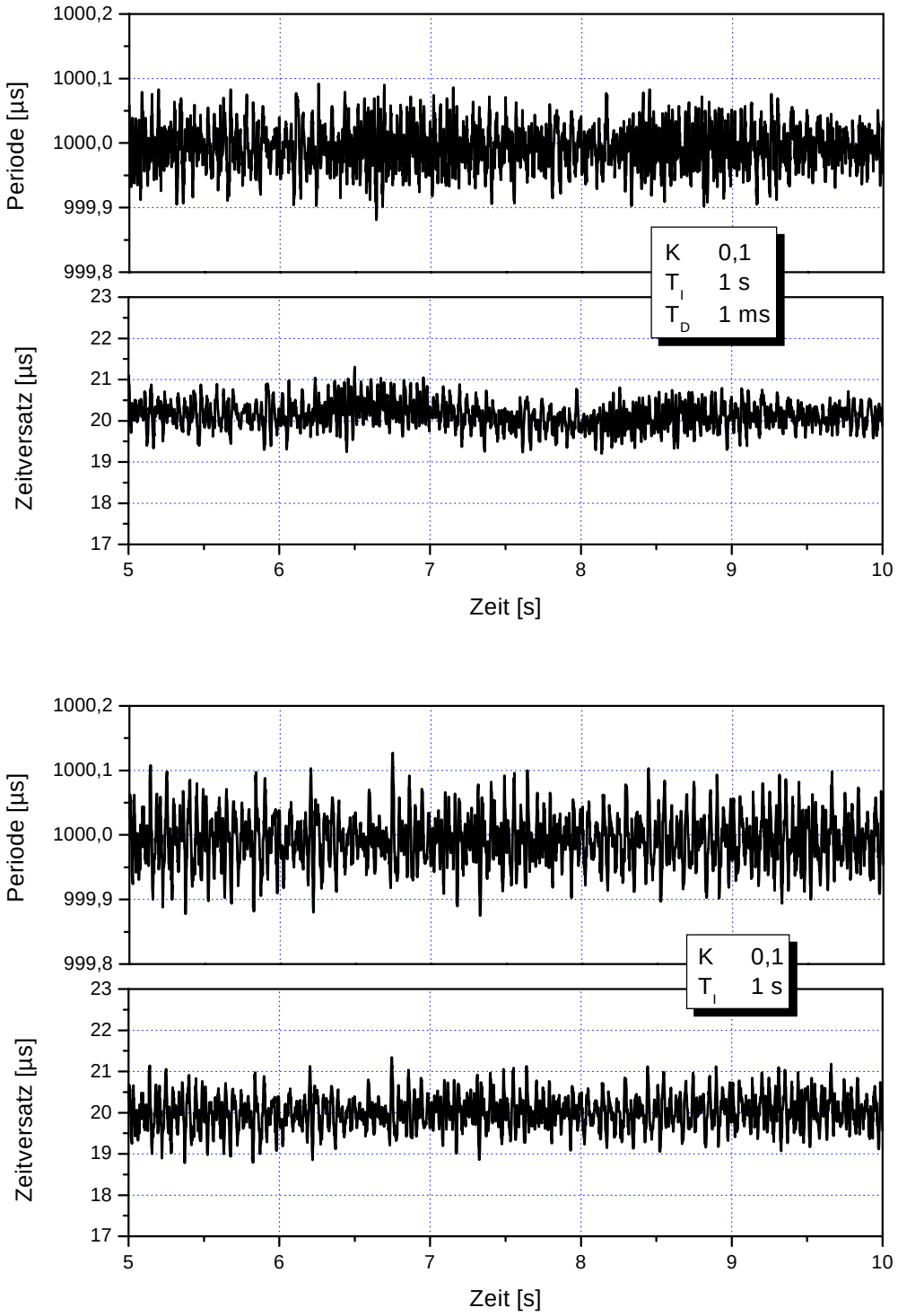



**Abbildung 5.9:** Störverhalten des PID-Reglers (oben) im Vergleich zum PI-Regler (unten). Es sind nur unwesentliche Unterschiede zu erkennen.

Anhand der Tabelle wird deutlich, weshalb für die Zeitkonstante des Integralanteils der Wert 1 s als Kompromiss gewählt wurde. Bei $T_I = 0,1s$ wird die Sollvorgabe ausgezeichnet eingeregelt, jedoch kam es zu starkem Überschwingen im Führungsverhalten (Abbildung 5.6, unten). Der scheinbar günstigste Wert von $T_I = 10s$ im Führungsverhalten mit guter Dämpfung (Abbildung 5.6, oben) führte jedoch zur größten Regelabweichung in der Messung des Störverhaltens. Ein Wert von $T_I = 1s$ stellt somit einen guten Kompromiss dar, da beide Anforderungen ausreichend erfüllt werden. Die Standardabweichung ist hier mit 0,3178 µs ebenfalls am geringsten. Auffallend ist außerdem der Wert für die Standardabweichung der Periode bei einem Differenzialanteil von $T_D = 1s$. Wie bereits in Abbildung 5.7, unten, zu erkennen war kam es hier zu einem starken Ansprechen des Reglers auf das Rauschen im Zeitversatzsignal. Diese Beobachtung wird durch den großen Wert für die Standardabweichung bestätigt.

Der benutzte FTA-Algorithmus hat auf das Störverhalten ebenfalls einen positiven Einfluss. Die Fenstergröße für die Mittelwertbildung wurde auf 10 Werte festgelegt. Wie bereits in Abschnitt 5.2.1 erläutert, bedeutet dies eine Tiefpassfilterung des Zeitversatzsignals. Durch die Fenstergröße wird die Mittelung über einen Zeitbereich von 10 ms durchgeführt. Dies ist aber im Verhältnis zur gewählten Zeitkonstante des Integralanteils von 1 s ein relativ kleiner Wert. Deshalb kann dieser Anteil bei der Bewertung des Regelverhaltens vernachlässigt werden. Ohne Filterung wird sich lediglich eine Vergrößerung des Rauschens einstellen. Wird jedoch das Störverhalten untersucht, so ergeben sich gravierende Unterschiede in der Varianz der Regelgröße (Abbildung 5.10). Wird z.B. die Knotenanzahl im Netzwerk erhöht, so ergibt sich ohne FTA-Algorithmus eine wesentlich größere Streuung des gemessenen Zeitversatzes. Wird der FTA-Algorithmus jedoch eingesetzt, so ist die vorhandene Streuung nahezu konstant.

### 5.2.3 Bandbreite

Eine weitere wichtige Größe für ein Netzwerk ist die Bandbreite. Sie gibt an, wieviele Daten sich in einer Zeiteinheit über das Netzwerk übertragen lassen. Üblicherweise wird mit der Bandbreite der maximalen Wert der Bandbreite bezeichnet. Besonders in ereignisgesteuerten Netzwerken, wie dem Internet oder einem lokalen Netz, ist die Bandbreite das entscheidende Kriterium. In der hier untersuchten Netzwerklösung ist sie jedoch erst an zweiter Stelle nach der Einhaltung der Zeitanforderung zu nennen. Ein entscheidender Punkt für die Bandbreite ist die Paketgröße. In der Regel haben Netzwerke mit kleiner Paketgröße, wie z.B. CAN, eine geringere Bandbreite (1 Mbit) als Netzwerke mit großen Paketgrößen, wie z.B. Ethernet (10/100 Mbit). Die hier benutzten Netzwerkkarten haben in ihrem standardmäßigen Einsatz eine maximale Bandbreite von 100 Mbit. Dabei wird eine Paketgröße von ca. 1500 Byte benutzt, die als maximale Größe durch die Karte festgelegt ist. In dem vorgestellten Treiber wurde dieser Wert übernommen. Da die gesamte Zeit in Zeitscheiben aufgeteilt ist und diese den einzelnen Knoten jeweils exklusiv zur Verfügung stehen, führt dies gleichzeitig zu einer Aufteilung der Bandbreite. Ein weiterer Teil wird für die Synchronisation eingesetzt und steht somit ebenfalls nicht zur Verfügung.

Abhängig von der Knotenzahl n und der Periode des RT Threads T in Millisekunden und der Paketgröße P in Byte ergibt sich die theoretische Bandbreite in Mbit/s wie folgt

$$\frac{B}{[MBit/s]} = \frac{1000}{(n+2)} \frac{T^{-1}}{[ms^{-1}]} \frac{P}{[Byte]} \frac{8}{1024^2} \tag{5.1}$$

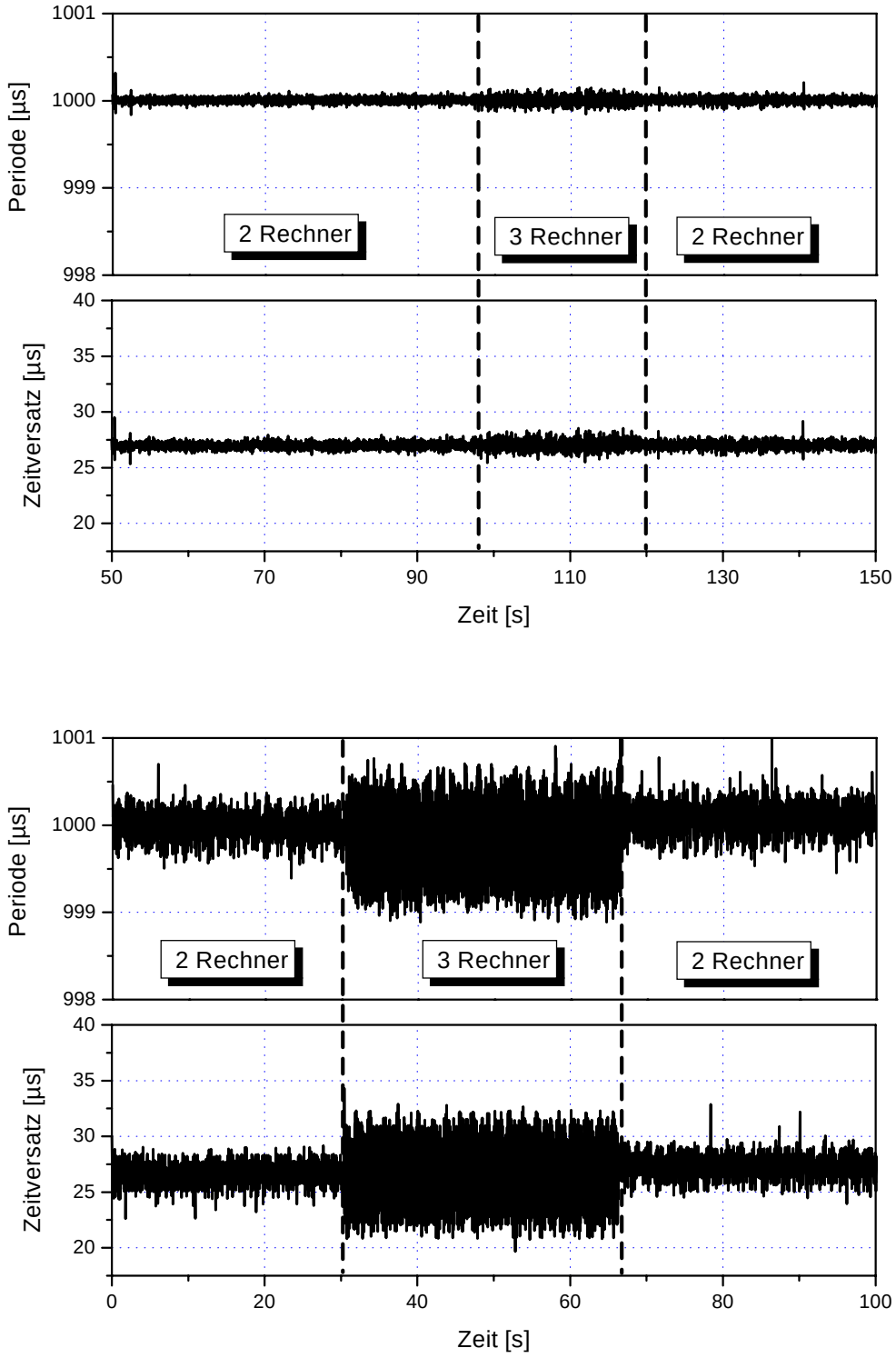



**Abbildung 5.10:** Einfluss des FTA-Algorithmus auf das stationäre Regelverhalten. Oben sind der gemessene Zeitversatz und die Periode des RT Threads bei einem Wechsel der Knotenzahl mit FTA-Algorithmus gezeigt. Unten ist ein gleicher Vorgang ohne FTA-Filter dargestellt.

Diese Formel ergibt sich aus der Überlegung, dass die Gesamtanzahl der Pakete in einer Sekunde durch $n + 2$ (Anzahl der Knoten und zwei Protokollzeitscheiben) geteilt werden muss. In jedem Paket werden P Bytes übertragen und auf Mbit umgerechnet. Dabei wird die eigentliche Laufzeit auf dem Netzwerk vernachlässigt. Gleichung (5.1) lässt sich auch tabellarisch darstellen:

| Paketgröße P [Byte] | 1500 | 1500 | 100 | 100 |
|---|---|---|---|---|
| Periode T [ms] | 1 | 0,5 | 1 | 0,5 |
| n = 2 | 2,861 | 5,722 | 0,191 | 0,381 |
| n = 3 | 2,289 | 4,578 | 0,153 | 0,305 |
| n = 4 | 1,907 | 3,815 | 0,127 | 0,254 |
| n = 5 | 1,635 | 3,270 | 0,109 | 0,218 |

**Tabelle 5.2:** Bandbreite in Mbit/s in Abhängigkeit von der Paketgröße, der Periode und der Knotenanzahl

Es ist deutlich zu erkennen, dass sich die Bandbreite mit der Verkürzung der Periodendauer erhöht. Ein Wert von $T = 1ms$ hat sich aber als gut im Hinblick auf die Regelung ergeben. Wenn eine kürzere Basisperiode gewählt wird, dann ist dazu die Auflösung des Pentium-Timers und der Scheduler-Periodendauer geringer. Das führt zu einem größeren Rauschen im Zeitversatz- und Periodensignal (Quantisierungsrauschen).

In vorliegender Implementation wurde eine Paketgröße von 1500 Byte bei einer Periode von 1 ms eingesetzt. Die theoretische Bandbreite konnte auch im Versuch bestätigt werden. Zu diesem Zweck wurde eine TCP/IP-Strecke zwischen zwei Rechnern über das Echtzeit Netzwerk unter Linux aufgebaut. Über diese Teststrecke wurde eine Datei mit einer Größe von 1,5 MByte an den Zielrechner geschickt. Da es sich bei TCP um ein Protokoll mit expliziter Flusskontrolle handelt, terminiert die `write`-Funktion erst, nachdem der Zielrechner den Empfang bestätigt hat. Um die Bandbreite zu messen, muss lediglich die Zeit zwischen dem Aufruf und dem Beenden von `write` gemessen werden. Es konnten so Werte von ca. 2,1 Mbit/s ermittelt werden. Die Differenz zur theoretischen Bandbreite von 2,86 Mbit/s ist mit dem Protokollaufwand für TCP/IP zu erklären. Da es sich aber um eine Messung unter Linux handelt und diese nicht im Echtzeit-Bereich durchgeführt wurde, kann dies als eine „worst case“ Abschätzung betrachtet werden.

### 5.2.4 Auslastung

Ein weiterer Punkt zur Bewertung des Treibers stellt die Auslastung des Echtzeit-Systems dar. Da das Netzwerk nur als eine Art Dienstleistung des Betriebssystems anzusehen ist, sollte es nicht zuviel Rechenzeit des Echtzeit-Systems beanspruchen, die dann für die eigentliche Applikation nicht mehr zur Verfügung steht. Die Auslastung kann, in Kenntnis der Periode, über die Ausführungszeit des RT-Threads bestimmt werden. Zur Messung wurde auch hier der Pentium-Timer (vgl. Abschnitt 4.3.2) herangezogen. Sie erfolgte in der Run-Phase des Protokolls, da sich der Treiber während des laufenden Betriebs immer in diesem Zustand befindet. Es wurden Messungen im Master-Modus und im Client-Modus vorgenommen. Die Messergebnisse sind in Abbildung 5.11 grafisch dargestellt.

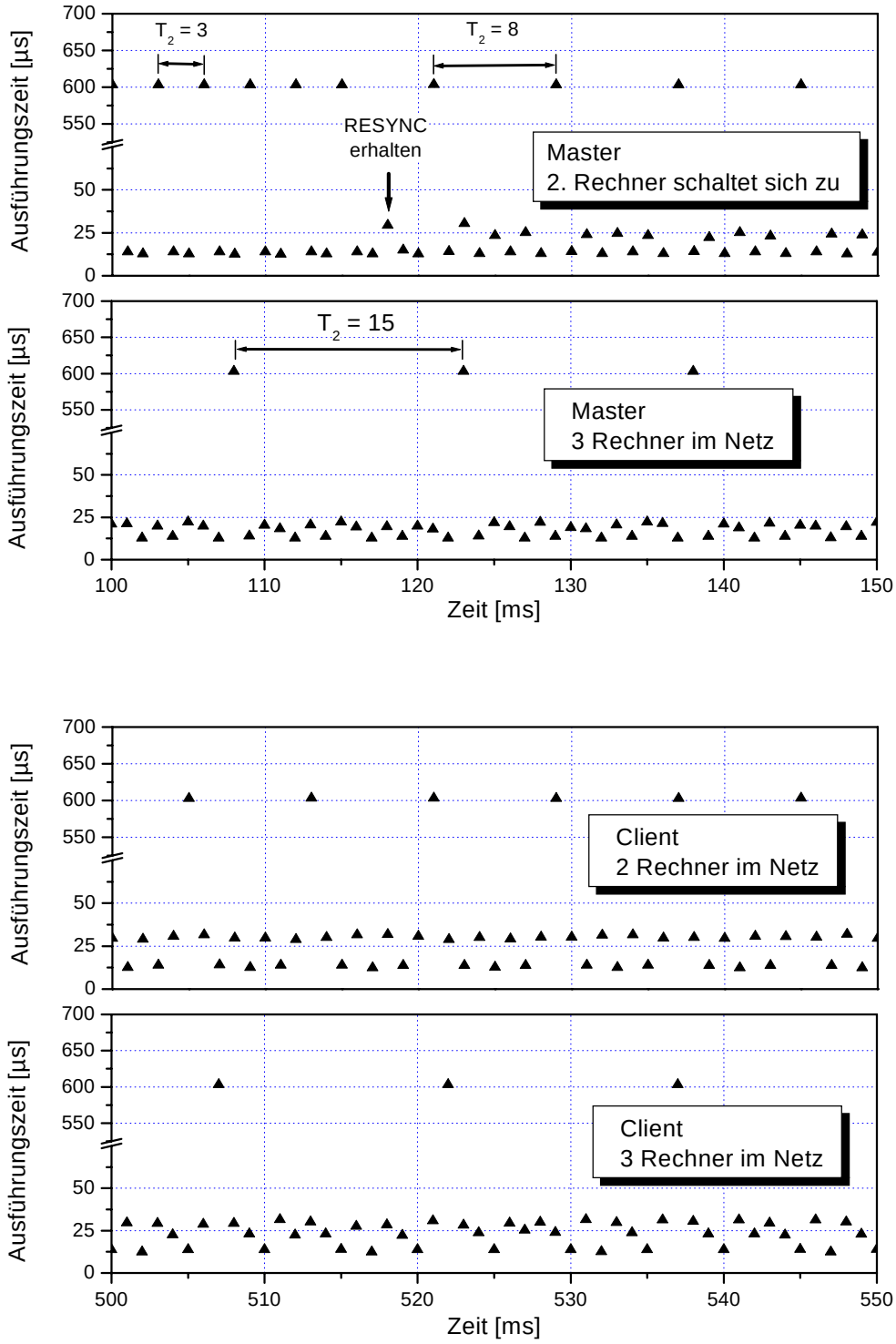



**Abbildung 5.11:** Ausführungszeit des RT Threads in der Run-Phase. Jeder Punkt repräsentiert eine Zeitscheibe. Sehr deutlich sind an der langen Ausführungszeit die Zeitscheiben zu erkennen, in denen auf ein RESYNC-Paket eines neuen Knotens gewartet wird.

Der arithmetische Mittelwert der Ausführungzeit $\Delta e_m$ ergibt im Verhältnis zur Periode $\Delta p$ des RT Threads von 1 ms, die Auslastung des Systems $\rho$.

$$\rho = \frac{\Delta e_m}{\Delta p}$$

Die so berechneten Werte sind in folgender Tabelle dargestellt:

| | Knotenanzahl | n = 1 | n = 2 | n = 3 |
|---|---|---|---|---|
| **Master** | mittl. Ausführungszeit $\Delta e_m$ | 209,41 µs | 91,67 µs | 56,39 µs |
| | Auslastung $\rho$ | 20,94 % | 9,17 % | 5,64 % |
| **Client** | mittl. Ausführungszeit $\Delta e_m$ | -- | 95,65 µs | 61,84 µs |
| | Auslastung $\rho$ | -- | 9,57 % | 6,18 % |

**Tabelle 5.3:** Auslastung des Echtzeit Systems durch den Treiber

Die Ausführungszeit wird im wesentlichen durch die Häufigkeit, mit der der Treiber auf ein RESYNC Paket eines eventuellen neuen Knotens wartet, bestimmt. Aufgrund der Einstellung des Zeitversatzdetektors ergibt sich hier eine Ausführungszeit von ca. 600 µs, was eine Auslastung von 60 % in dieser Zeitscheibe bedeutet. Bei der Entwicklung einer Echtzeit-Applikation, die das Netzwerk benutzt, muss dieses Verhalten berücksichtigt werden. Wird für eine Applikation mit zwei Rechnern eine Planung nach monotonen Raten vorgesehen, so stehen noch ca. 60 % der gesamten Rechenzeit für das Programm zur Verfügung, wenn unter allen Umständen ein brauchbarer Plan gefunden werden soll.

## 5.3 Verwendungsmöglichkeiten

Durch die Echtzeitfähigkeiten zeichnet sich diese Netzwerklösung für eine Verwendung als Prozessbus aus. Vielfach ist in einer solchen Anwendung, wie z.B. einer Maschinensteuerung, eine temporale Verlässlichkeit sehr wichtig. Oftmals wird in aktuellen Systemen auf eine verteilte Struktur verzichtet, da kein geeignetes Netzwerk zur Verfügung steht. Mit dem hier vorgestellten Netzwerk kann ein echtzeitfähiges Netzwerk ohne zusätzlichen Hardware-Aufwand realisiert werden, wodurch die Kosten gering gehalten werden können. Die zeitgesteuerte Architektur garantiert eine Unabhängigkeit der Auslastung von der Anzahl der übertragenen Daten. Fehlerhafte Knoten werden tolerant behandelt. Dadurch eignet sich das Netzwerk auch für einen Einsatz in sicherheitskritischen Bereichen.

Mit einer Bandbreite von gut 2 Mbit/s ist das Netzwerk als als Alternative zum CAN-Bus (max. 1Mbit/s) in der Automatisierungstechnik zu sehen, wenn ein echtzeitfähiges Netzwerk benötigt wird. Der Einsatz der jetzigen Lösung im Bereich der eingebetteten Systeme hängt stark von der zur Verfügung stehenden Prozessorleistung ab, da der Treiber relativ viel Rechenzeit beansprucht. Ein anderes Protokoll könnte helfen, diesen Nachteil zu beseitigen. Als Konkurrenz auf dem Gebiet der eingebetteten Systeme sind an erster Stelle Produkte zu sehen, die ebenfalls eine zeitgesteuerten Ansatz verfolgen, wie TTP/C und TTP/A der Firma TTTech [8][26] oder TTCAN der Firma Bosch [5]. Gegenüber diesen Netzwerken zeichnet

sich die in dieser Arbeit diskutierte Lösung durch ihre Flexibilität aus. Durch ein anderes Protokoll, was lediglich eine Anpassung der Treiber-Software erfordern würde, wäre eine dynamische Bandbreitenzuweisung denkbar, ohne die Hardware zu ändern. Die Zeitscheiben könnten dann statt der jetzigen Gleichverteilung an alle Netzteilnehmer, entsprechend ihrer Anforderung der Übertragungsbandbreite dynamisch verteilt werden.

# 6 Zusammenfassung und Ausblick

Mit dieser Arbeit wird ein Netzwerkkartentreiber vorgestellt, mit dessen Hilfe sich ein echtzeitfähiges Netzwerk von PCs unter Linux und RT Linux realisieren lässt. Das Ziel der Diplomarbeit wurde damit im Sinne der Aufgabenstellung erreicht. Es wird gezeigt, dass ein zeitgesteuertes Zugriffsverfahren mit Standard-Hardware gut geeignet ist, die Anforderungen zu erfüllen. Die Implementierung erfolgte für die handelsübliche 100 MBit Ethernetkarte 3Com 905B.

Ziel eines Echtzeitnetzwerkes ist es, neben dem korrekten Versenden von Daten auch zeitliche Vorgaben zu erfüllen, d.h. das Einhalten einer Frist für die Übertragung einer Nachricht muss garantiert werden. In der hier beschriebenen Lösung kann die Frist mit der hohen Genauigkeit von ± 3 µs eingehalten werden. Dieses Ziel wurde mittels des zeitgesteuerten Zugriffsverfahrens erreicht, im Gegensatz zu ereignisgesteuerten Methoden, wie sie in den meisten bestehenden Netzen zu finden sind. Statt eine Nachricht jederzeit, ohne Beachtung des Datenverkehrs auf der Leitung, zu senden, werden den einzelnen Knoten Zeitscheiben zugeteilt, in denen sie den exklusiven Zugriff auf das Netz besitzen. Dadurch ist sichergestellt, dass es zu keinen Kollisionen von Datenpaketen kommt, was ein unvorhersehbares zeitliches Verhalten zur Folge hätte. Das bereits in der Hardware von Ethernetadaptern implementierte Kollisions-Detektions-Protokoll (CSMA/CD) konnte so ausgeschaltet werden und ist damit für den gesamten Übertragungskanal als transparent anzusehen.

Eine globale Zeit hat für ein verteiltes, zeitgesteuertes System, wie es das beschriebene Netzwerk darstellt, eine fundamentale Bedeutung. Da die Zuteilung der Sendefreigabe nur durch das Fortschreiten der Zeit synchronisiert wird, muss die interne Uhr jedes Knotens unbedingt dem Netzwerk angepasst werden. Es wird gezeigt, dass die Driftrate von Quarzoszillatoren, wie sie als Taktgeber in PCs verwendet werden, in der Größenordnung $10^{-6}$ liegt. Die Zeit, die der Scheduler von RT Linux nutzt, um dem Sende- und Empfangsthread Rechenzeit zuzuteilen, basiert ebenfalls auf diesem Oszillator und ist somit der gleichen Driftrate unterlegen. Mit der Messung der Empfangszeitpunkte der Ethernetpakete wird eine Möglichkeit vorgestellt, den Drift der lokalen Uhr gegenüber dem Netzwerk mit hoher Genauigkeit zu bestimmen. Es wird gezeigt, dass eine Regelung der Periode des Sende- und Empfangsthread einen sehr präzisen, kontinuierlichen Ausgleich des gemessenen Zeitversatzes erlaubt. Dafür wurde mit dem PID-Regler ein, aus der klassischen Regelungstechnik bekannter Algorithmus, der vorliegenden Aufgabe angepasst.

Da jedes Echtzeitprogramm für RT Linux im Kernel-Bereich läuft, konnte der gesamte Treiber in einem einzigen Modul abstrahiert werden, das dynamisch zum Betriebssystemkern gebunden werden kann. Als Besonderheit kann das Netzwerk gleichzeitig unter RT Linux als auch über die Linux Netzwerkschnittstelle genutzt werden. Das ermöglicht vielfältige Verwendungsmöglichkeiten in der Anbindung von Netzknoten. Zur Unterscheidung nach Echtzeit- und Nicht-Echtzeitpaketen werden Prioritäten und Protokollkennungen verwendet. Ein aufwändiges Protokoll, basierend auf dem oben beschriebenen Zeitscheibenverfahren, steuert den Zugriff auf das Medium und ermöglicht eine dynamische Netzkonfiguration. Ein neuer Knoten kann sich ohne Einschränkung der Netzwerkkommunikation, im laufenden Betrieb aufschalten. Nach einer passiven Beobachtungsphase eines neuen Knotens erfolgt der exakte, aktive Abgleich des zeitlichen Rahmens.

Das Protokoll ist als Basisprotokoll für zukünftige Entwicklungen zu sehen. Es vereinigt die dynamische Netzkonfiguration und die Vorgabe des Taktes der Paketübertragung in einem als Master definierten Netzknoten. In aufbauenden Arbeiten kann hier eine Optimierung hinsichtlich des jeweiligen Einsatzzweckes erfolgen. Wenn das Netzwerk für einen Einsatz in einem sicherheitskritischen Bereich vorgesehen ist, so könnte die dynamischen Netzkonfiguration einem statischen Planungsverfahren weichen, bei dem auf die Masterfunktion verzichtet werden kann. Im Gegensatz dazu könnte ein Ausbau der dynamisch Planung in einer anderen Umgebung gefordert sein. Hier ist beispielsweise eine Verwendung als Prozessbus in einer Fertigungseinrichtung denkbar, in der einzelne Maschinen zu Wartungszwecken ausgeschaltet werden müssen, ohne den Produktionsprozess zu beeinträchtigen. Ebenfalls ist ein Ausbau der Netzwerklösung hinsichtlich der Bandbreite (z.B. für Bildverarbeitung zur Qualitätssicherung) durch eine Verringerung des Protokollanteils am Datenverkehr denkbar. Für eine solche Anwendung könnte das Synchronisationsintervall vergrößert oder die Taktrate im Netz erhöht werden.

Obwohl mit dem eingesetzten Regelalgorithmus ein gutes Ergebnis erzielt werden konnte, ist in einem adaptiv parametrierbarem Regler ein weiteres Optimierungspotenzial zu sehen. In Kapitel 5 ist mit der ausführlichen Untersuchung des Stör- und Führungsverhalten eine gute Basis gegeben, um in zukünftigen Entwicklungen eine optimale Anpassung des Parametersatzes an unterschiedliche Arbeitspunkte zu gewährleisten.

In folgenden Arbeiten wird der Treiber zu einem vollständigen Kommunikationspaket ausgebaut werden. Hierzu bietet sich eine Software-Schnittstelle an, die dem POSIX.4 Standard entspricht [6]. In den darin spezifizierten Funktionen zur synchronen und asynchronen Kommunikation ist eine sinnvolle Erweiterung zu der hier vorgestellten Treiberschnittstelle zu sehen. Vor allem mit synchroner Ein-/Ausgabe kann eine Echtzeit-Applikation an den zeitlichen Rahmen des Netzwerkes angepasst werden, da dieser durch die Regelung der RT-Thread-Periode nicht eindeutig an den der Applikation gekoppelt ist. Mit einer POSIX.4 Schnittstelle wird das Netzwerk in eine verteilte, objektorientierte Programmierumgebung eingebunden. Diese entsteht zurzeit am Lehrstuhl für Betriebssysteme der RWTH Aachen unter dem Projektnamen ROFES (Real-time CORBA for Embedded Systems) [12].

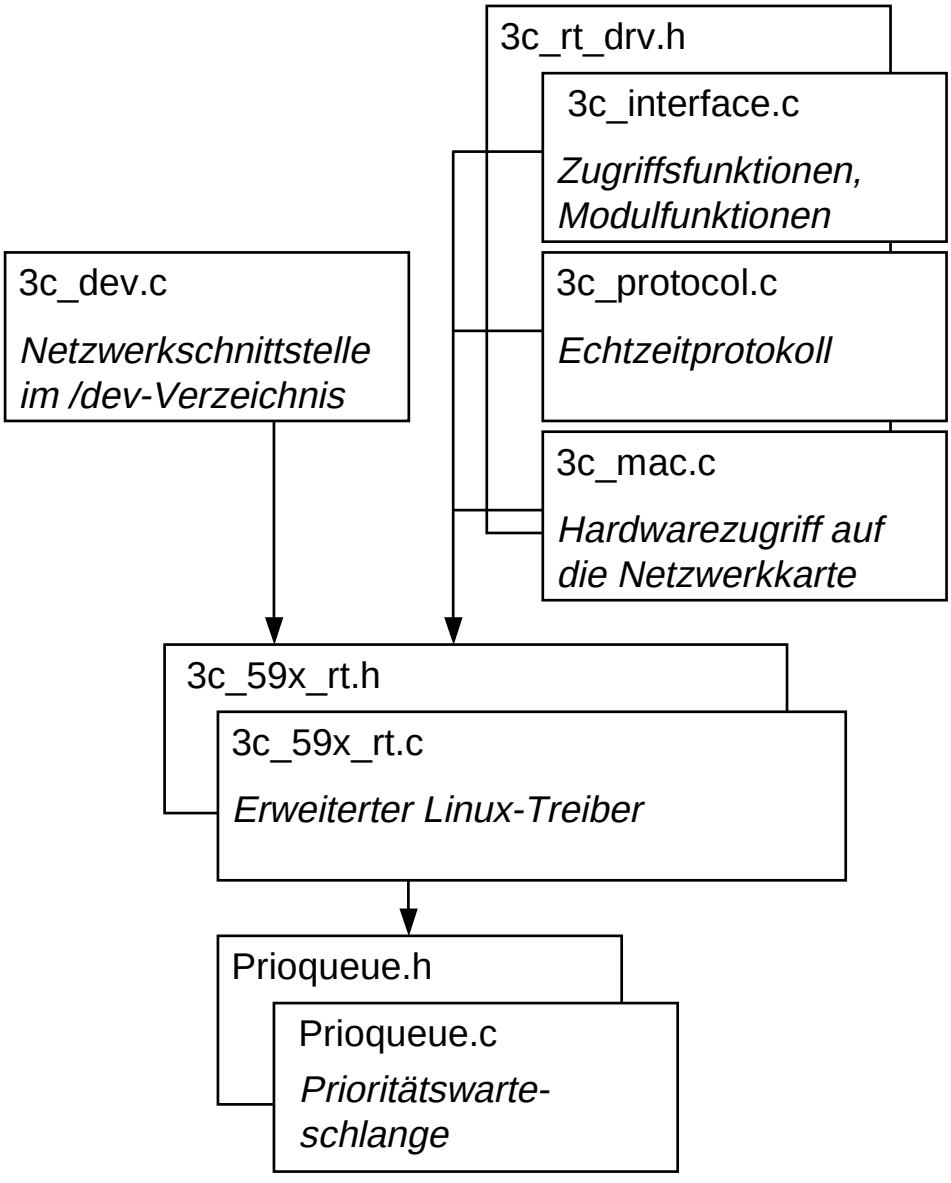
# A Anhang: Symbole und Abkürzungen

| | |
|---|---|
| AP | Arbeitspunkt |
| B | Bandbreite |
| CAN | Control Area Network |
| CMD | Kommando Byte der Echtzeitpakete |
| CNI | Communication Network Interface |
| CSMA/CA | Collision Sense Multiple Access / Collision Avoidence |
| CSMA/CD | Collision Sense Multiple Access / Collision Detection |
| CRC | Cyclic Redundancy Check (Checksummenfeld) |
| DIMM | Dual Inline Memory Module |
| DMA | Direct Memory Access |
| $\Delta e$ | Ausführungszeit eines Prozesses |
| $\Delta p$ | Periode eines Prozesses |
| $\Delta t$ | Zeitversatz |
| $\Delta t_{AP}$ | Arbeitspunkt des Zeitversatzdetektors |
| $\Delta t_t$ | Totzeit des Zeitversatzdetektors |
| e(k) | Eingangsfolge des Reglers |
| ET | Event Triggered |
| $\varepsilon$ | Jitter |
| f | Frequenz |
| FIFO | First In First Out |
| FTA | Fault Tolerant Averaging |
| $\varphi$ | Phase |
| $\Phi$ | Konvergenzfunktion |
| $\Gamma$ | Drift Offset |
| HAL | Hardware Abstraction Layer |
| IEEE | Institute of Electronical and Electronics Engineers |
| IP | Internet Protocol |
| IRQ | Interrupt Request |

| | |
|---|---|
| ISO/OSI | International Standards Organisation / Open Systems Interconnection |
| $K, K_R$ | Proportionalitätskonstante des Reglers |
| MAC | Media Access Control |
| MEDL | Message Description List |
| $microtick_i^k$ | Mikrotick i der Uhr k |
| $\mu(N, k)$ | Byzantinischer Fehlerterm |
| NIC | Network Interface Card |
| $n^k$ | nominale Granularität der Uhr k |
| P | Paketgröße |
| PAR | Positive Acknowledgement-or-Retransmission |
| PI-Regler | Proportional-Integral-Regler |
| PID-Regler | Proportional-Integral-Differenzial-Regler |
| PLL | Phase Locked Loop |
| POSIX | Portable System Interface for Computer Environments |
| PThread | POSIX Thread |
| $\Pi$ | Präzision |
| $R_{int}$ | Resynchronisationsintervall |
| ROFES | Real-Time CORBA for Embedded Systems |
| RT | Real Time |
| rtf | Real Time FIFO |
| $\rho$ | Auslastung, Driftrate |
| $t_L$ | Latenzzeit |
| SG | Synchronisation Gap bei Minislotting |
| T | Periode, Abtastperiode |
| $T_D$ | Zeitkonstante des Differenzialanteils |
| $T_I$ | Zeitkonstante des Integralanteils |
| TAI | International Atomic Time |
| TCP | Transmission Control Protocol |
| TDMA | Time Devision Multiple Access |
| TG | Time Gap bei Minislotting |

| | |
|---|---|
| TT | Time Triggered |
| TTP | Time Triggered Protocol |
| u(k) | Ausgangsfolge des Reglers |
| UTC | Universal Time Coordinated |
| z(e) | Zeitpunkt des Auftretens von e nach der Referenzuhr |

# B Anhang: Dateienübersicht



# C Anhang: Device Struktur

Quelle: `/usr/src/rtlinux-2.2/linux/include/linux/netdevice.h`

```
struct device
{

  /*
   * This is the first field of the "visible" part of this struc-
ture
   * (i.e. as seen by users in the "Space.c" file).  It is the
name
   * the interface.
   */
  char   *name;

  /*
   *I/O specific fields
   *FIXME: Merge these and struct ifmap into one
   */
  unsigned longrmem_end;/* shmem "recv" end*/
  unsigned longrmem_start;/* shmem "recv" start*/
  unsigned longmem_end;/* shared mem end*/
  unsigned longmem_start;/* shared mem start*/
  unsigned longbase_addr;/* device I/O address*/
  unsigned intirq;/* device IRQ number*/

  /* Low-level status flags. */
  volatile unsigned charstart;/* start an operation*/
  /*
   * These two are just single-bit flags, but due to atomicity
   * reasons they have to be inside a "unsigned long". However,
   * they should be inside the SAME unsigned long instead of
   * this wasteful use of memory..
   */
  unsigned longinterrupt;/* bitops.. */
  unsigned longtbusy;/* transmitter busy */

  struct device*next;

  /* The device initialization function. Called only once. */
  int    (*init)(struct device *dev);
  void   (*destructor)(struct device *dev);
```

```
  /* Interface index. Unique device identifier*/
  int   ifindex;
  int   iflink;

  /*
   *Some hardware also needs these fields, but they are not
   *part of the usual set specified in Space.c.
   */

  unsigned charif_port;/* Selectable AUI, TP,..*/
  unsigned chardma;/* DMA channel*/

  struct net_device_stats* (*get_stats)(struct device *dev);
  struct iw_statistics*(*get_wireless_stats)(struct device *dev);

  /*
   * This marks the end of the "visible" part of the structure.
All
   * fields hereafter are internal to the system, and may change
at
   * will (read: may be cleaned up at will).
   */

  /* These may be needed for future network-power-down code. */
  unsigned longtrans_start;/* Time (in jiffies) of last Tx*/
  unsigned longlast_rx;/* Time of last Rx*/

  unsigned shortflags;/* interface flags (a la BSD)*/
  unsigned shortgflags;
  unsignedmtu;/* interface MTU value*/
  unsigned shorttype;/* interface hardware type*/
  unsigned shorthard_header_len;/* hardware hdr length*/
  void  *priv;/* pointer to private data*/

  /* Interface address info. */
  unsigned charbroadcast[MAX_ADDR_LEN];/* hw bcast add*/
  unsigned charpad;/* make dev_addr aligned to 8 bytes */
  unsigned chardev_addr[MAX_ADDR_LEN];/* hw address*/
  unsigned charaddr_len;/* hardware address length*/

  struct dev_mc_list*mc_list;/* Multicast mac addresses*/
  int   mc_count;/* Number of installed mcasts*/
  int   promiscuity;
  int   allmulti;
```

```
  /* For load balancing driver pair support */

  unsigned longpkt_queue;/* Packets queued*/
  struct device*slave;/* Slave device*/

  /* Protocol specific pointers */

  void  *atalk_ptr;/* AppleTalk link */
  void  *ip_ptr;/* IPv4 specific data*/
  void                  *dn_ptr;       /* DECnet specific data */

  struct Qdisc*qdisc;
  struct Qdisc*qdisc_sleeping;
  struct Qdisc*qdisc_list;
  unsigned longtx_queue_len;/* Max frames per queue allowed */

  /* Bridge stuff */
  int   bridge_port_id;

  /* Pointers to interface service routines.*/
  int   (*open)(struct device *dev);
  int   (*stop)(struct device *dev);
  int   (*hard_start_xmit) (struct sk_buff *skb,
                 struct device *dev);
  int   (*hard_header) (struct sk_buff *skb,
              struct device *dev,
              unsigned short type,
              void *daddr,
              void *saddr,
              unsigned len);
  int   (*rebuild_header)(struct sk_buff *skb);
#define HAVE_MULTICAST
  void  (*set_multicast_list)(struct device *dev);
#define HAVE_SET_MAC_ADDR
  int   (*set_mac_address)(struct device *dev,
                void *addr);
#define HAVE_PRIVATE_IOCTL
  int   (*do_ioctl)(struct device *dev,
               struct ifreq *ifr, int cmd);
#define HAVE_SET_CONFIG
  int   (*set_config)(struct device *dev,
                 struct ifmap *map);
#define HAVE_HEADER_CACHE
  int   (*hard_header_cache)(struct neighbour *neigh,
                  struct hh_cache *hh);
```

```
  void  (*header_cache_update)(struct hh_cache *hh,
                  struct device *dev,
                  unsigned char *  haddr);
#define HAVE_CHANGE_MTU
  int   (*change_mtu)(struct device *dev, int new_mtu);

  int   (*hard_header_parse)(struct sk_buff *skb,
                unsigned char *haddr);
  int   (*neigh_setup)(struct device *dev, struct neigh_parms *);
  int   (*accept_fastpath)(struct device *, struct dst_entry*);

#ifdef CONFIG_NET_FASTROUTE
  /* Really, this semaphore may be necessary and for not fastroute
code;
     f.e. SMP??
   */
  int   tx_semaphore;
#define NETDEV_FASTROUTE_HMASK 0xF
  /* Semi-private data. Keep it at the end of device struct. */
  struct dst_entry*fastpath[NETDEV_FASTROUTE_HMASK+1];
#endif
};
```

# D Anhang: Socket Buffer

Quelle: `/usr/src/rtlinux-2.2/linux/include/linux/skbuff.h`

```
struct sk_buff {
  struct sk_buff* next;/* Next buffer in list */
  struct sk_buff* prev;/* Previous buffer in list */
  struct sk_buff_head * list;/* List we are on*/
  struct sock*sk;/* Socket we are owned by */
  struct timevalstamp;/* Time we arrived*/
  struct device*dev;/* Device we arrived on/are leaving by*/

  /* Transport layer header */
  union
  {
    struct tcphdr*th;
    struct udphdr*uh;
    struct icmphdr*icmph;
    struct igmphdr*igmph;
    struct iphdr*ipiph;
    struct spxhdr*spxh;
    unsigned char*raw;
  } h;

  /* Network layer header */
  union
  {
    struct iphdr*iph;
    struct ipv6hdr*ipv6h;
    struct arphdr*arph;
    struct ipxhdr*ipxh;
    unsigned char*raw;
  } nh;

  /* Link layer header */
  union
  {
    struct ethhdr*ethernet;
    unsigned char *raw;
  } mac;

  struct  dst_entry *dst;

  char cb[48];
```

```
  unsigned int len;/* Length of actual data*/
  unsigned intcsum;/* Checksum */
  volatile char used;/* Data moved to user and not MSG_PEEK*/
  unsigned charis_clone,/* We are a clone*/
      cloned, /* head may be cloned (check refcnt to be sure). */
      pkt_type,/* Packet class*/
      pkt_bridged,/* Tracker for bridging */
      ip_summed;/* Driver fed us an IP checksum*/
  __u32priority;/* Packet queueing priority*/
  atomic_tusers;/* User count - see datagram.c,tcp.c */
  unsigned shortprotocol;/* Packet protocol from driver. */
  unsigned shortsecurity;/* Security level of packet*/
  unsigned inttruesize;/* Buffer size */

  unsigned char*head;/* Head of buffer */
  unsigned char*data;/* Data head pointer*/
  unsigned char*tail;/* Tail pointer*/
  unsigned char *end;/* End pointer*/
  void (*destructor)(struct sk_buff *);/* Destruct function*/
#ifdef CONFIG_IP_FIREWALL
        __u32           fwmark;                 /* Label made by
fwchains, used by pktsched*/
#endif
#if defined(CONFIG_SHAPER) || defined(CONFIG_SHAPER_MODULE)
  __u32shapelatency;/* Latency on frame */
  __u32shapeclock;/* Time it should go out */
  __u32shapelen;/* Frame length in clocks */
  __u32shapestamp;/* Stamp for shaper    */
  __u16shapepend;/* Pending */
#endif

#if defined(CONFIG_HIPPI)
  union{
    __u32ifield;
  } private;
#endif
};
```

# Literaturverzeichnis


[1] T. Bemmerl, M. Pfeiffer: *Realzeitsysteme I*, Umdruck zur Vorlesung, Lehrstuhl für Betriebssysteme an der RWTH-Aachen, 1999

[2] T. Bemmerl, S. Lankes: *Betriebssystempraktikum: Realzeitverarbeitung*, Lehrstuhl für Betriebssysteme an der RWTH-Aachen, 2000

[3] I. N. Bronstein, K. A. Semendjajew: *Taschenbuch der Mathematik*, B.G. Teubner Verlagsgesellschaft, 1991

[4] E. Dilger, T. Führer, B. Müller, S. Poledna, T. Thurner: *X-By-Wire: Design von verteilten, fehlertoleranten und sicherheitskritischen Anwendungen in moderenen Kraftfahrzeugen*, Systemengineering in der KFZ-Entwicklung, VDI Berichte 1374, VDI Verlag, Düsseldorf 1997, pp. 427-442, 1997

[5] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther: *Time Triggered Communication on CAN (Time Triggered CAN-TTCAN)*, Robert Bosch GmbH

[6] Bill O. Gallmeister: *POSIX.4, Programming for the real world*, O'Reilly & Associates, Inc., 1. Auflage, 1995

[7] B. W. Kernighan, D. M. Richie: *Programmieren in C, 2. Ausg. ANSI-C*, Carl Hanser Verlag, 1990

[8] H. Kopetz, G. Grünsteidl: *TTP -- A Protocol for Fault-Tolerant Real-Time Systems*, IEEE Compter, Jan. 1994, pp. 14-23, 1994

[9] H. Kopetz, W. Ochsenreiter: *Clock Synchronization in Distributed Real-Time Systems*, IEEE Trans. on Computers, Vol. 36(8): pp. 933-940, August 1987

[10] H. Kopetz: *A Comparison of CAN and TTP*, Proc. of the IFAC Distributed Computer Systems Workshop, Como, Italy September 9-11, 1998

[11] H. Kopetz: *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997

[12] Lehrstuhl für Betriebssysteme: *The ROFES Project*, http://www.lfbs.rwth-aachen.de/users/stefan/rofes, 2000

[13] L. Lamport: *A New Solution of Dijkstra's Concurrent Programming Problem*, Comm. ACM. Vol. 8 (7), pp. 453-455, 1974

[14] L. Lamport, P.M. Melliar-Smith: *Synchronizing Clocks in the Presence of Faults*, Journal of the ACM, Vol. 21, pp. 52-78, 1985

[15] L. Lamport, R. Shostak, M. Pease: *The Byzantine Generals Problem*, ACM Trans. on Programming Lang. and Systems, Vol. 4, No. 3, pp. 382-401, 1982

[16] C.L. Liu, J.W. Layland: *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, Vol. 20, pp. 46-61, 1973

[17] H. Meyr: *Regelungstechnik und Systemtheorie*, Lehrstuhl für integrierte Systeme der Signalverarbeitung, RWTH-Aachen, 2. Aufl, 1996

[18] D.L. Mills: *Modelling and Analysis of Computer Network Clocks*, Electrical Engineering Department Report 92-5-2, University of Delaware, Mai 1992, 29pp., 1992

[19] M. Pease, R. Shostak, L. Lamport: *Reaching Agreement in the Presence of Faults*, Journal of the ACM, Vol. 27(2), pp. 228-234, 1980

[20] S. Poledna, A. Burns, A. Wellings, P. Barrett: *Replica Determinism and Flexible Scheduling in Hard Real-Time Dependable Systems*, IEEE Trans. on Computers, Vol. 49, No. 2, pp. 100-111, 2000

[21] H. Pfeifer, D. Schwier, F. W. von Henke: *Formal Verfication for Time-Triggered Clock Synchronization*, 7th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-7), San Jose, CA, 6-8 Jan. 1999

[22] P. Ramanathan, K. G. Shin, R. W. Butler: *Fault-Tolerant Clock Synchronization in Distributed Systems*, IEEE Computer 23, Okt. 1990, pp. 33-42, 1990

[23] A. Rubini: *Linux Gerätetreiber*, O'Reilly Verlag, 1998

[24] C. Tanzer, M. Glück: *TTPos-The Time-Triggered and Fault-Tolerant RTOS*, TTTech Computertechnik GmbH, Wien, www.tttech.com

[25] U. Tietze, Ch. Schenk: *Halbleiter-Schaltungstechnik*, 10. Auflage, Springer-Verlag, 1993

[26] TTTech: *Specification of the TTP/C Protocol*, Spec. Ver. 0.5, Doc. ed. 1.0, TTTech Computertechnik AG, 1999

[27] B. Walke: *Angewandte Informatik 2, Rechnerorganisation und Systemleistung*, Lehrstuhl Kommunikationsnetze an der RWTH-Aachen, 1995

[28] U. Wenkebach: *CAN in der Medizintechnik*, Elektronik Heft 16/97, 1997

[29] V. Yodaiken, M. Barabanov: *RTLinux Version Two*, VJY Associates LLC, 1999

[30] V. Yodaiken: *The RTLinux Manifesto*, Department of Computer Science, New Mexico Institute of Technology, http://www.rtlinux.org

[31] 3Com: *3C90x and 3C90xB NICs Technical Reference*, 3Com Corporation, 1998