Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Johannes Reidl

Optimierung der Zeitpräzision eines Linux Ethernet Treibers auf Basis einer PTP Uhr

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Johannes Reidl

Optimierung der Zeitpräzision eines Linux Ethernet Treibers auf Basis einer PTP Uhr

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Franz Korf
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 14. November 2013

Johannes Reidl

Thema der Arbeit
Optimierung der Zeitpräzision eines Linux Ethernet Treibers auf Basis einer PTP Uhr

Stichworte
Echtzeit, Linux, Ethernet, Netzwerke, Treiber, TTEthernet, AS6802, IEEE 1588, PTP-Uhr, RTEthernet

Kurzzusammenfassung
Diese Arbeit befasst sich mit der Einbindung einer Netzwerkkarte, die in der Lage ist den Empfangszeitpunkt von Ethernetpaketen in Hardware zu bestimmen, in einen Linux TTEthernet Treiber. Zunächst wird der Stand der Entwicklung ermittelt und eine Anforderungsanalyse erstellt, danach die Auswahl besagter Netzwerkkarte diskutiert. Darauf folgend werden die benötigten Grundlagen erörtert und, darauf aufbauend, die Kopplung des TTEthernet- und des Netzwerkkartentreibers erklärt. Abschließend wird auf die Funktionalität und die Messergebnisse eingegangen.

Title of the paper
Optimization of the temporal precision of a Linux Ethernet driver on the basis of a PTP Clock

Keywords
Realtime, Linux, Ethernet, Network, Device Driver, TTEthernet, AS6802, IEEE 1588, PTP Clock, RTEthernet

Abstract
This thesis describes the integration of a network interface card, which is able to make hardware timestamps of incoming ethernet frames, in a Linux TTEthernet driver. At first the status of the development is looked at and a requirements analaysis is done. Secondly the choice of mentioned NIC is beeing discussed. Then the needed basics are introduced and on that basis the integration of the TTEthernet driver into the NICs driver is explained. At last the measuring results are beeing looked at and the functionality is proven.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

Am 29. Januar 1886 meldet der deutsche Erfinder Carl Benz seinen Entwurf für den „Motorwagen Nummer 1“zum Patent an. Dieses Fahrzeug besitzt neben einer Lenkstange, einem Motor und drei Vollgummi Drahtspeichenrädern keine nennenswerten Extras. Alles ist mechanisch und funktional (vgl. Benz und CO., 1886). Auch ist es in den Anfängen nur Wenigen vorbehalten einen solchen Wagen zu besitzen.

Im Laufe der Zeit hat sich viel an dem ursprünglichen Konzept eines Automobils verändert. Längst gehören Anschnallgurte, Airbags, eine Heizung und unzählige andere Neuerungen zur Standardausstattung. Mit der zunehmenden Technisierung der Gesellschaft haben auch viele elektronische Erweiterungen Einzug in das Auto gehalten. Eingebaute Navigation, sich automatisch auf den Fahrer einstellende Sitze und Radio sowie Bildschirme für die hinteren Sitzplätze sind nur ein paar Beispiele. Es ist auch nichts Ungewöhnliches mehr von dem Fahrzeug beim Einparken oder dem Zurücksetzen mit einer Rückfahrkamera unterstützt zu werden.

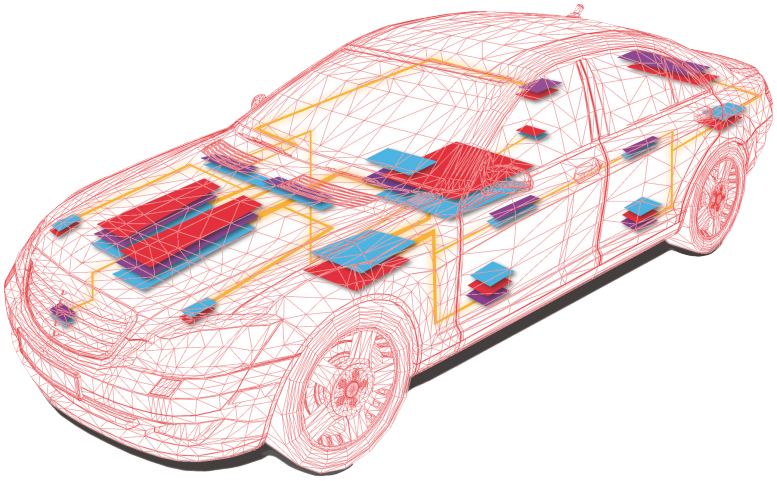

Abbildung 1.1: Bussysteme im Automobil

Viele dieser Neuerungen werden über Mikrocontroller gesteuert und geregelt und sind über verschiedenartige Bussysteme miteinander verbunden. Ein Phaeton der Volkswagen AG hat z.B. mehr als 60 Steuergeräte die über drei Bussysteme kommunizieren (vgl. Marscholik und Subke, 2011, S.3). Schematisch ist dies in Abbildung 1.1 dargestellt. Hierbei kommen häufig CAN (bis zu 1000Kbit/s) und FlexRay (2x 10Mbit/s) für zeitkritischen sowie MOST (50Mbit/s) bzw. LIN (20Kbit/s) für unkritischen Datenverkehr zum Einsatz (vgl. Zimmermann und Schmidgall, 2010, S.4). Mit der zunehmenden Anzahl an Neuerung besonders in puncto Fahrsicherheit stoßen die Bandbreiten der Busse an ihre Grenzen. Darüber hinaus werden durch die Heterogenität der Infrastruktur heutiger Fahrzeuge verschiedene Diagnose- und Reparaturwerkzeuge benötigt. Unter Anderem werden diese Probleme durch das TTEthernet-Protokoll(Time-triggered Ethernet), welches von der Firma TTTech entwickelt wird, adressiert. Es definiert die Erweiterung von Standard-Ethernet um zeitkritischen Datenverkehr und ist dabei unabhängig von der physikalischen Netzwerkschicht. Dadurch und wegen der viel höheren Bandbreite ist es möglich das gesamte Datenaufkommen über ein Bussystem zu leiten. Mit Hilfe beispielsweise eines Linux-Echtzeitsystems kann dieses TTEthernet-Protokoll realisiert werden und dieses System dann als ein Diagnosewerkzeug für den Bus, oder auch selbst als Teilnehmer im Netzwerk, fungieren.

Ziel dieser Arbeit

Diese Arbeit setzt auf die Bachelorarbeit „Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux“von Frieder Rick auf, in der ein TTEthernet Stack als Treiber für Linux entwickelt wurde, der mit minimalen Änderungen am Treiber der Netzwerkkarte auskommt und mit allen gängigen Netzwerkkarten zusammenarbeitet (vgl. Rick, 2012, S.10). Im Fazit weißt der Autor darauf hin, dass viele und komplexe Anpassungen im zugrunde liegenden Netzwerkkarten Treiber nötig sind um möglichst genaue Paket-Zeitstempel zu erhalten. Diese sind wichtig um eine ausreichend genaue Synchronisation mit dem Rest des TTEthernet-Netzwerks nach AS6802 (SAE Aerospace AS6802, 2011) zu erreichen und um festzustellen ob sie im erlaubten Zeitfenster angekommen sind. Das würde aber den ursprünglichen Designgedanken, nur minimale Änderungen vorzunehmen, verletzen und selbst dann, da es immer noch ein Software-basierter Ansatz ist, zu keinen exakten Zeitstempeln führen.

Eine alternative Lösung für dieses Problem ist die Verwendung einer Netzwerkkarte (im Folgenden auch als NIC bezeichnet, vom englischen „Network Interface Card“), welche in der Lage ist alle eingehenden Pakete in Hardware mit exakten Zeitstempeln zu versehen. Daher

ist das Hauptziel dieser Arbeit, unter Berücksichtigung des ursprünglichen Designgedankens, eine geeignete Netzwerkkarte zu finden und den TTEthernet Treiber mit dem Treiber der NIC zu verbinden. Ebenso wird der Synchronization Client nach AS6802 implementiert um das Linux-System mit einem TTEthernet-Netzwerk zu synchronisieren.

Aufbau der Arbeit

Diese Arbeit gliedert sich in mehrere Teilgebiete. Zunächst werden die für das Verständnis nötigen Grundlagen in Kapitel 2 erläutert. Dabei werden die Begriffe der weichen bzw. harten Echtzeit voneinander abgegrenzt und ein Überblick über das TTEthernet-Protokoll gegeben. Hier wird auf die verschiedenen Nachrichtenklassen, deren Erkennung, und vor allem auf den Synchronisationsmechanismus eingegangen. Anschließend wird das Betriebssystem Linux unter Realtime-Aspekten betrachtet, sowie die zur Verfügung stehenden Zeit- und Interruptquellen aufgezeigt. Weiter wird allgemein das Netzwerktreiber-Design unter Linux und einige relevante Kernel-Strukturen erklärt.

Kapitel 3 befasst sich mit der Analyse der Ausgangssituation. Dabei wird der Implementierungsstand der zugrunde liegenden Arbeit betrachtet und die daraus resultierende Aufgabenstellung dargelegt.

Als Nächstes wird das Konzept in Kapitel 4 für diese Arbeit vorgestellt. Dazu werden zuerst mögliche Architekturoptionen aufgezeigt und danach die Auswahl einer geeigneten Netzwerkkarte beschrieben. Darauf wird die Anbindung des Treibers dieser Netzwerkkarte an den TTEthernet-Treiber thematisiert. Am Ende des Kapitels wird die Zeitsynchronisation besprochen. Zum Einen wird die interne Synchronisation der Netzwerkkarten-Uhr mit der Linux-Systemzeit beschrieben und zum Anderen die Synchronisation mit anderen Geräten und somit dem Rest des TTEthernet-Netzwerks.

Daraufhin wird in Kapitel 5 auf die Umsetzung des zuvor vorgestellten Konzepts eingegangen.

In Kapitel 6 wird der Versuchsaufbau, mit dem alle Funktionalitäten geprüft und die Messungen vorgenommen wurden, erläutert. Darüber hinaus werden die Messergebisse für die interne Uhrensynchronisation, wie auch für den Synchronization Client analysiert.

Zum Schluss werden in Kapitel 7 noch einmal alle Ergebnisse zusammengefasst und ein Ausblick auf mögliche weiterführende Projekte gegeben.

2 Grundlagen

In diesem Kapitel werden die Grundlagen, die für das Verständnis dieser Arbeit nötig sind, erläutert. Bevor auf den TTEthernet-Standard eingegangen wird, ist es zunächst nötig, die Begriffe „harte“ bzw. „weiche Echtzeit“ abzugrenzen. Danach wird das Betriebssystem Linux aus dem Kontext dieser Echtzeitanforderungen betrachtet.

2.1 Echtzeit

Ein System wird dann als „Echtzeitsystem“ bezeichnet, wenn es auf ein Ereignis innerhalb einer genau spezifizierten Zeit zu reagieren hat. Eine verspätete Nachricht ist dabei äquivalent zu einer die nie empfangen wurde und kann daher nicht verwendet werden. Aber nicht nur das Einhalten von diesen sogenannten „Deadlines“, also dem Zeitpunkt, zu dem eine Aktion des Systems spätestens eintreten muss, spielt eine zentrale Rolle. Ein weiterer wichtiger Aspekt ist, dass auf vorhersehbare Art und Weise reagiert werden muss (vgl. Sally, 2010, S.257).

2.1.1 Weiche und harte Echtzeit

Das Ausmaß der Konsequenzen solcher Echtzeitsysteme kann stark variieren. Werden die Deadlines nicht eingehalten, reichen die möglichen Folgen, wie auch in Tabelle 2.1 dargestellt, von einer verzerrten Videokonferenz bis hin zu einer Gefahr für viele Menschenleben. Daher und aus der Art, wie auf das Nicht-Einhalten der Deadlines reagiert wird, wird hierbei zwischen weicher und harter Echtzeit unterschieden.

Harte Echtzeit liegt dann vor, wenn das Nicht-Einhalten einer Deadline zu ernsthaften Konsequenzen führen kann, wie etwa der Zerstörung des Systems und dessen Umwelt oder gar der Gefährdung von Personen. Deutlich wird dies am Beispiel eines Airbags. Im Falle eines Unfalls ist das Auslösen zu einer genau definierten, von der Aufprallgeschwindig-

keit abhängigen Zeit, sehr wichtig. Löst der Airbag zu einem falschen Zeitpunkt oder gar nicht aus, kann dies zu ernsthaften Verletzungen führen.

Weiche Echtzeit hingegen kann das gelegentliche Verpassen einer Deadline tolerieren. Hier besteht keine Gefahr für das System oder Personen. Im schlimmsten Fall kann dies lediglich zu einer starken Beeinträchtigung der Funktion führen. Typische Systeme mit weichen Echtzeitanforderung sind z. B. Geräte für die Audio- bzw. Videowiedergabe. Werden hier die Deadlines nicht eingehalten führt dies im schlimmsten Fall zu einem stockenden Musikstück oder dem Fehlen von einzelnen Bildern.

	harte Echtzeit	**weiche Echtzeit**
„Worst Case Performance“	gleich der durchschnittlichen Performance	einige Vielfache der durchschnittlichen Performance
verpasste Deadline	Ergebnis ist wertlos	verminderter Wert des Ergebnisses
Konsequenzen	ernsthafte Konsequenzen: mechanischer Schaden, der Verlust des Lebens oder von Gliedmaßen	tolerierbare Reduktion der Systemqualität, wie etwa ein verzerrtes Telefongespräch
typische Anwendungsgebiete	Flugbetrieb, medizinische Geräte, sicherheitskritische Industriesteuerungen	Computer-Netzwerke, Telekommunikation, Unterhaltungsanwendungen

Tabelle 2.1: Harte Echtzeit im Vergleich zu weicher Echtzeit (vgl. Sally, 2010, S.258)

2.2 TTEthernet

Standard Ethernet erfüllt die Anforderungen an ein Echtzeit-System nicht und garantiert keine genau definierten Sende- bzw. Empfangszeitpunkte für kritischen Datenverkehr. Damit kann nicht sicher gestellt werden, dass wichtige Pakete rechtzeitig ankommen. Daher wurde das zu IEEE 802.3 kompatible Time Triggered Ethernet (SAE AS6802) von der Firma TTTech entwickelt. Time Triggered bedeutet hierbei, dass zeitkritische Nachrichten nach einem exakten Zeitplan transportiert werden.

2.2.1 Nachrichtenklassen und Priorisierung

Um sowohl den weichen, als auch den harten Echtzeitanforderungen begegnen zu können, implementiert Time Triggered Ethernet drei Nachrichtenklassen mit unterschiedlichen Prioritäten. Dabei ist die gesamte Paketübertragung in fest definierten Zeitzyklen organisiert. (vgl. TTTech Computertechnik AG, S.6f)

Time Triggered Nachrichten (TT) besitzen die höchste Priorität und können somit nicht verdrängt werden. Sie stellen kurze Latenzzeiten und sehr kleine zeitliche Abweichungen (Jitter) sicher und werden daher für zeitkritischen Datenverkehr verwendet. Sie werden statisch konfiguriert und jedes Gerät zwischen Sender und Empfänger kennt die Zeitpunkte, zu denen das Paket ankommen bzw. weiter versendet werden muss.

Rate Constrained Nachrichten (RC) haben die zweit höchste Priorität und verdrängen Best Effort Nachrichten. Sie werden unter anderem für weiche Echtzeitanforderungen, wie etwa das Streamen von Audio- oder Videoanwendungen, verwendet. Ist zu einem Zeitpunkt kein TT-Datenverkehr vorgesehen, so kann dem RC-Verkehr eine fest vordefinierte Bandbreite zugesichert und die Pakete übermittelt werden.

Best Effort Nachrichten (BE) haben die niedrigste Priorität und entsprechen Standard Ethernet IEEE 802.3 Paketen. Sie werden zwischen den Sendeintervallen von TT- und RC-Nachrichten gesendet. Muss TT- oder RC-Datenverkehr übermittelt werden, so werden BE-Nachrichten verdrängt. Es kann nicht sicher gestellt werden, dass Pakete dieser Nachrichtenklasse erfolgreich übermittelt werden.

2.2.2 Erkennung der Nachrichtentypen

Der Aufbau eines TTEthernet-Frames entspricht exakt dem eines Standard Ethernet Pakets und ist in Abbildung 2.1 dargestellt. Um dennoch eine Unterscheidung vornehmen zu können, wird das „Destination Feld“, also die Zieladresse, anders interpretiert. Das sechs Byte große Feld ist in ein Vier Byte TT-Marker Feld und ein Zwei Byte großes TT-ID Feld unterteilt. Der TT-Marker zeigt an, dass es sich um zeitkritischen Datenverkehr handelt und die TT-ID identifiziert die Nachricht. Darüber hinaus legt sie fest, ob es sich um eine TT- oder eine RC-Nachricht handelt. Die TT-Adressen befinden sich hierbei im Multicast-Adressbereich. Ein TTEthernet-Switch liest diese Felder aus und behandelt die Nachrichten dann entsprechend ihrer Priorität. Bei der Konfiguration des Netzwerks ist darauf zu achten, dass kein Fall auftritt, in dem zwei TT-Nachrichten zur gleichen Zeit eintreffen.

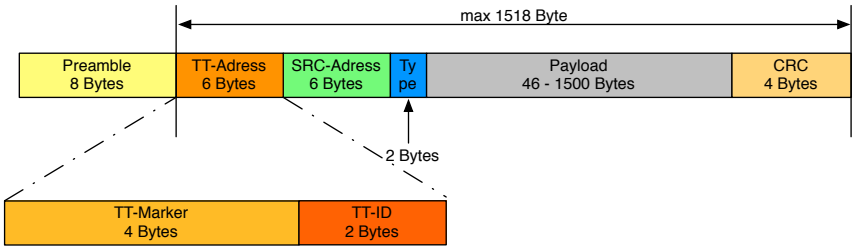

Abbildung 2.1: Aufbau eines TTFrames (vgl. Bartols, 2010, S.32)

2.2.3 Zeitsynchronisation nach SAE AS6802

Die Zeitsynchronisation von Endgeräten ist in einem TTEthernet-Netzwerk von zentraler Bedeutung, da es, um feststellen zu können ob die definierten Empfangs- und Sendezeiten der zeitkritischen Nachrichten eingehalten wurden, nötig ist, eine globale Uhrzeit zu haben.

Das Synchronisationsprotokoll definiert für alle am Netzwerk teilnehmenden Geräte drei verschiedene Rollen:

- Synchronization Client (SC)
- Synchronization Master (SM)
- Compression Master (CM)

Jedes der Geräte kann eine dieser Rollen einnehmen. Dabei ist es auch möglich dem Netzwerk zur Laufzeit neue Komponenten hinzu zu fügen. Die Synchronisation findet in zwei Schritten statt. Zunächst senden die SMs im ersten Schritt sogenannte Protocol Control Frames (PCF) an den CM. Dieser bildet aus den relativen Empfangszeiten einen Mittelwert und sendet im zweiten Schritt einen neuen PCF an alle SMs und SCs, welche wiederum ihre Uhrzeit anhand dieses Frames korrigieren. Dieser Vorgang wird in jedem TT-Zyklus wiederholt. Für eine genaue Beschreibung des SMs und des CMs sei hier auf die Spezifikation SAE Aerospace AS6802 (2011) verwiesen. Auf den Synchronization Client wird in Kapitel 4 genauer eingegangen.

Protocol Control Frames

Die Synchronisation findet über Protocol Control Frames statt. Sie haben innerhalb der RC-Nachrichtenklasse die höchste Priorität. Diese PCFs sind standard Ethernet-Frames mit der

minimalen Payload von 46 Bytes. Zu erkennen sind sie daran, dass deren Ethernet-Typ-Feld den Wert 0x891d hex hat(vgl. SAE Aerospace AS6802, 2011, S.30f). Abbildung 2.2 zeigt die Payload eines solchen Frames, wobei nur 28 Bytes belegt und 18 Bytes für zukünftige Erweiterungen reserviert sind.

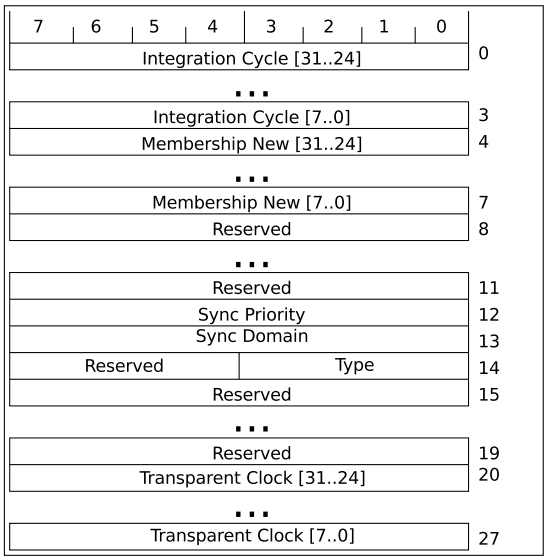

Abbildung 2.2: Aufbau eines PC-Frames (vgl. SAE Aerospace AS6802, 2011, S.30)

Tabelle 2.2 zeigt die Beschreibung der einzelnen Felder. Es gibt drei Arten von PCFrames, die anhand des *pcf_type* zu erkennen sind: Coldstart Frames CS (0x04 hex), Coldstart Acknowledge Frames CA (0x08 hex) und Integration Frames IN (0x02 hex). CS- und CA-Frames kommen bei der Initialisierung des Netzwerks zum Einsatz und werden von den SMs erzeugt. Die Synchronization Clients senden keine PCFs, sie reagieren nur auf IN-Frames und passen ihre Zeit anhand des Transparent-Clock Feldes an.

Mit jedem Knotenpunkt zwischen Sender und Empfänger eines PCF wird das Transparent-Clock Feld aktualisiert. Hierbei werden die jeweiligen Übertragungsverzögerungen auf dieses Feld addiert, bevor der Frame weiter versendet wird. Die maximal mögliche Verzögerung wird im Voraus berechnet und statisch in das Feld eingetragen.

Name	Länge	Beschreibung
pcf_integegration_cycle	32 Bit	Integrationszyklus in dem der PCF gesendet wurde
pcf_membership_new	32 Bit	Bitvector mit einer statisch konfigurierten 1 zu 1 Beziehung eines Bits zu einem Synchronization Master im System
pcf_sync_priotity	8 Bit	Statisch konfigurierter Wert in jedem Synchronization Master/Client und Compression Master
pcf_sync_domain	8 Bit	Statisch konfigurierter Wert in jedem Synchronization Master/Client und Compression Master
pcf_type	4 Bit	Frame Typ des PCF
pcf_transparent_clock	64 Bit	Hält die akkumulierten Verzögerungen eines PCF begonnen bei seinem Ursprung bis zu seinem Ziel. Die Zeit wird in Vielfachen von Picosekunden dargestellt.

Tabelle 2.2: PCF PAYLOAD (vgl. SAE Aerospace AS6802, 2011, S.31)

2.3 Das Betriebssystem Linux

Linux implementiert die Unix-API und ist daher ein Unix ähnliches Betriebssystem. Die erste Version wurde 1991 von Linus Torvalds an der Universität Helsinki für den damals hochentwickelten Intel 80386 Mikroprozessor entwickelt.

Heute arbeiten hunderte Entwickler auf der ganzen Welt daran und Linus ist noch immer maßgeblich an der Verbesserung und Weiterentwicklung von Linux beteiligt. Im Laufe der Jahre wurde Linux dabei auf viele andere Prozessor-Architekturen portiert, unter anderem Hewlett-Packard's Alpha, Intel's Itanium, AMD's AMD64, PowerPC und ARM (vgl. Bovet und Cesati, 2008, S.1). Wegen dieser Eigenschaft, der freien Verfügbarkeit und dem offenen Quellcode hat Linux in den verschiedensten Bereichen Einzug gehalten und ist nicht mehr aus der Computerwelt wegzudenken. Es findet vor allem in Servern und eingebetteten Systemen Anwendung und ist dabei, dank seiner Anpassbarkeit, einer Vielzahl von Anforderungen gewachsen.

Linux wurde als Allzweck- und Mehrbenutzer-Betriebssystem konzipiert. Die Ziele eines Solchen stehen im Allgemeinen im Konflikt mit den Anforderungen an ein Realtime-Betriebssystem, denn hier steht das Minimieren der maximalen Verzögerungen und eine deterministische Abarbeitung von Routinen an erster Stelle und nicht der allgemeine Durchsatz (vgl. Abbot, 2006, S.167). Von sich aus ist Linux demnach nicht für Echtzeitanwendungen geeignet. Jedoch wird dieses Problem durch einen Patch adressiert. Dieser erweitert den Kernel um die Fähigkeiten eines echten Realtime-Betriebssystems (Ts'o u. a., 2013).

2.3.1 Scheduling unter Linux

Linux ist ein multitaskingfähiges Betriebssystem. Das bedeutet, dass mehrere Prozesse gleichzeitig ausgeführt werden können. Gibt es mehr Prozesse als physikalische CPUs, so müssen die Tasks von Zeit zu Zeit unterbrochen werden, damit ein anderer an die Reihe kommen kann, um den Eindruck der Gleichzeitigkeit zu erwecken.

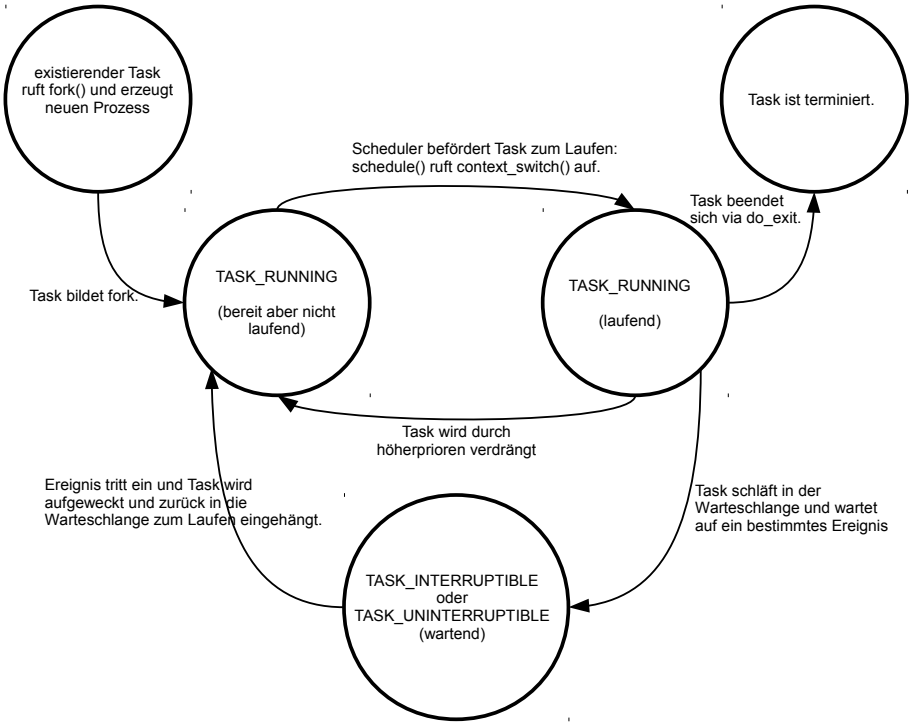

Abbildung 2.3: Flussdiagramm der Linux-Prozess-Zustände (vgl. Love, 2010, S.28)

Dabei gibt es zwei Arten des Scheduling: das Kooperative und das Präemtive. Beim kooperativen Scheduling entscheidet der Task selbst, in welchen Abständen er die CPU abgibt. Präemtives Scheduling ermöglicht es einen laufenden Prozess zu unterbrechen und einem Höherprioren die CPU zu gegeben. Dazu wird eine Instanz benötigt, die diesen Vorgang verwaltet und den Prozessen Rechenzeit zuteilt. Diese Instanz stellt der Scheduler dar. Abbildung 2.3 zeigt dabei die Zustände, in denen sich ein Prozess befinden kann. Wird ein neuer Task gestartet, so befindet er sich im Zustand „bereit“ und wartet darauf dass ihm vom Scheduler die CPU zugeteilt wird. Ist dies der Fall, so wechselt er in den Zustand „laufend“. Hier bleibt er

solange, bis er entweder beendet wird, von einem Höherprioren verdrängt und sich dadurch wieder in „bereit“ einreihen muss oder er die CPU ab gibt, da er in einer Warteschlange auf ein Ereignis wartet.

Linux implementiert präemtives Scheduling. Dabei wird noch einmal zwischen zwei verschiedenen Schedulerklassen unterschieden: (vgl. Love, 2010, S.66f)

SCHED_NORMAL (CFS) ist die Schedulerklasse für normale Prozesse. Hier wird jedem Prozess die gleiche CPU-Zeit zugeteilt und derjenige ausgewählt, der sich am längsten in der Warteschlange befindet.

SCHED_FIFO und SCHED_RR werden für Echtzeitprozesse verwendet. Sie haben eine höhere Priorität als das CFS und verdrängen somit normale Prozesse. SCHED_FIFO implementiert einen einfachen first in - first out Algorithmus ohne feste CPU-Zeitslots. Ein Task läuft solange bis er vollständig abgearbeitet wurde, blockiert, die CPU von selbst abgibt oder von einem anderen Realtime-Task mit höherer Priorität abgelöst wird. Gibt es zwei RT-Prozesse mit der selben Priorität, so wechseln sich diese nach dem Round Robin Verfahren ab.

SCHED_RR ist identisch mit SCHED_FIFO, nur dass hier den Prozessen Zeitslots zugeteilt werden. Verbraucht ein solcher Prozess seine Zeitslots, so kommt ein anderer mit der selben Priorität an die Reihe.

Die Prioritäten werden statisch konfiguriert und reichen von 0, der niedrigsten, bis 99, der höchsten Priorität. Diese Schedulingklassen zeigen weiches Echtzeitverhalten. Der Kernel versucht die Timing-Deadlines einzuhalten, kann jedoch nicht garantieren dass dieses Ziel immer erreicht wird. Durch das Realtime Patch Projekt (vgl. Ts’o u. a., 2013) wird der Kernel so modifiziert, dass er harte Echtzeitanforderungen erfüllt und voll-präemtives Scheduling unterstützt.

2.3.2 Vom Umgang mit Zeiten und Interrupts

Timer und Interrupts sind wichtige Konzepte in der Welt der Computer. Viele Geräte, die mit dem PC verbunden sind, operieren mit einer signifikant langsameren Geschwindigkeit als das System selbst. Dabei ist es nicht wünschenswert, dass der Prozessor auf eine Antwort bzw. ein Ereignis von einem externen Gerät wartet und somit keine anderen Operationen durchführen kann. Daher gibt es Interrupts, um der CPU ein Signal zu senden, dass ein Ereignis eingetreten ist. (vgl. Corbet u. a., 2005, S.258)

Linux Interrupts

Wird ein Interrupt ausgelöst, so unterbricht die CPU ihre aktuelle Verarbeitung und springt in eine Interrupt-Service-Routine (ISR). Diese ISR ist vom Treiberentwickler als Funktion zu implementieren und beim Kernel zu registrieren.

Wurde eine Hardware-ISR abgearbeitet und der Interrupt wieder freigegeben überprüft der Kernel ob weitere wichtige Routinen auszuführen sind. Diese Routinen werden Softirq genannt. Die Abarbeitung findet im Interrupt-Kontext statt, kann dabei jedoch von einem weiteren Interrupt unterbrochen werden. Dabei gibt es zwei Ausprägungen der Softirqs. Da sie in dieser Arbeit keine Anwendung finden, werden sie nur der Vollständigkeit halber aufgeführt und nicht im Detail besprochen. Für eine genauere Beschreibung sei hier auf die Seiten 159-174 bei Quade (2011) verwiesen.

Tasklet : Ist eine längere Berechnung im Kontext eines Interrupts nötig, so kommen Tasklets zum Einsatz. Dadurch wird die Abarbeitung eines Interrupts in zwei Teile geteilt. Im ersten finden nur die zeitkritischen Aktionen statt und weitere Interrupts sind gesperrt. Im zweiten finden die übrigen Berechnungen statt und der Interrupt wieder freigegeben.

Timer: Ein Timer kommt zum Einsatz, wenn eine Aktion zu einem späteren, genau definierten Zeitpunkt, ausgeführt werden soll. Wird der angegebene Zeitpunkt erreicht, so wird eine bei der Initialisierung angegebene Funktion im Interrupt-Kontext ausgeführt. Eine spezielle Form der Timer sind die *High Resolution Timer*, die im weiteren Verlauf genauer erläutert werden.

Linux und die Zeit

Von besonderer Bedeutung für diese Arbeit ist der Umgang mit Zeit und Zeitquellen unter Linux. Im Folgenden wird ein Überblick über diese gegeben.

Periodische Timerticks: Historisch bedingt werden diese, innerhalb des Kernels, relativ zum Einschaltzeitpunkt gemessen. Es gibt dazu eine globale Variable $jiffies$, die sich mit jedem Timertick erhöht. Dabei ist die Updatefrequenz von System zu System unterschiedlich und liegt typischerweise zwischen $1ms$ und $4ms$. Dies führt allerdings zu einer verminderten Zeitgenauigkeit.

Dynamische Timerticks: Ab der Kernelversion 2.6.16 wurde die Zeitverwaltung auf dynamische Timerticks umgestellt. Hier werden keine kontinuierlichen Timer-Interrupts

mehr generiert, sondern der Kernel berechnet im Voraus wann er aktiv werden muss und aktiviert einen Timer, der zu genau dem berechneten Zeitpunkt einen Interrupt auslöst. Dies ermöglicht unter anderem eine viel höhere Zeitpräzision.

Durch die Umstellung auf dynamische Timerticks fanden sogenannte **High Resolution Timer** Einzug in den Kernel. Sie zeigen ein deutlich genaueres Zeitverhalten. Es lassen sich dabei zwei Zeitquellen einstellen. *CLOCK_MONOTONIC* repräsentiert die Zeit innerhalb des Rechners. Sie wird wird kontinuierlich weiter gezählt, selbst wenn sich von außen die Zeit sprunghaft ändert. *CLOCK_REALTIME* ist die Zeit außerhalb des Rechners. Ändert sich etwas an den Zeiteinstellungen des Systems, so hat dies direkte Auswirkungen auf den Timer.

Zusätzlich kann konfiguriert werden, ob der Zeitpunkt, zu dem der Timer einen Interrupt auslösen soll, relativ (*HRTIMER_MODE_REL*) oder absolut (*HRTIMER_MODE_ABS*) angegeben wird. Listing 2.1 zeigt eine Beispielimplementierung mit absolutem Zeitwert und *CLOCK_REALIME* als Zeitquelle.

```
#include <linux/hrtimer.h>

struct hrtimer bsp_timer;
ktime time = timevalue_in_the_future;

// callback function
static enum hrtimer_restart callback_function( struct hrtimer * hrt){
                    ⋮
  return HRTIMER_RESTART;
}

// hrtimer initialize
hrtimer_init(&bsp_timer, CLOCK_REALTIME, HRTIMER_MODE_ABS);

// set callback function
bsp_timer.function = callback_function;

// start hrtimer
hrtimer_start(&bsp_timer, time , HRTIMER_MODE_ABS)
```

Listing 2.1: HRTIMER Initialisierung Beispiel

Die Zeitverwaltung in Linux verwendet den Datentyp *ktime_t*. Dieser Datentyp hält einen Zeitwert in Sekunden und Nanosekunden. Dabei ist sichergestellt, dass der Nanosekundenanteil nie größer oder gleich einer Sekunde wird. Tabelle 2.3 zeigt die Funktionen zum Umgang mit dem Datentypen *ktime_t*.

Bereich	Funktion
Initialisierung	ktime_set
Rechenfunktionen	ktime_add ktime_sub ktime_add_ns ktime_sub_ns ktime_add_us ktime_sub_us
Vergleiche	ktime_equal
Konvertierung	timespec_to_ktime timeval_to_ktime ktime_to_timespec ktime_to_timeval ns_to_ktime ktime_to_ms ktime_to_us ktime_to_ns ktime_us_delta
Zeit lesen	ktime_get_ts ktime_get_real ktime_get_real_ts

Tabelle 2.3: Funktionen zum Umgang mit ktime (vgl. Quade, 2011, S.224f)

Neben *jiffies* gibt es noch weiter Zeitquellen. Die globale Variable *xtime* vom Typ *struct timespec* zählt ebenfalls Sekunden und Nanosekunden, zeigt die vergangene Zeit jedoch in Unix-Darstellung, also seit dem 1. Januar 1970, an.

Wird eine sehr genau Zeitbasis benötigt, kann, falls vorhanden, auf den internen Taktzyklenzähler (tsc) zurückgegriffen werden. Hierbei handelt es sich um ein prozessorspezifisches Register, welches mit jedem CPU-Takt inkrementiert wird. Die hohe Zeitgenauigkeit wird jedoch mit einigen möglichen Nachteilen erkauft:

- nicht jede CPU besitzt so ein Register
- moderne CPUs besitzen Stromsparmodi, in denen die Taktfrequenz verändert wird, so wird es schwer die tatsächlich verstrichene Zeit zu berechnen.

Zwar ist es möglich direkt auf das tsc zuzugreifen, jedoch ist diese Methode wegen den oben genannten Gründen nicht zu empfehlen, zumal der Code dann auch nur auf einem Prozessor mit tsc ausführbar wäre. Stattdessen empfiehlt sich die Verwendung der Methoden

do_gettimeofday oder *ktime_get_real*, da hier das tsc-Register, falls vorhanden, mit verrechnet wird und so für eine erhöhte Genauigkeit gesorgt ist.

Listing 2.2 zeigt wie unter Linux mit Hilfe des Terminals überprüft werden kann, ob der vorhandene Prozessor ein tsc-Register besitzt.

```
$ dmesg | grep tsc
tsc: Fast TSC calibration using PIT
[    0.001000] tsc: Detected 2494.218 MHz processor
[    1.803596] tsc: Refined TSC clocksource calibration: 2494.333 MHz
[    2.804846] Switched to clocksource tsc
```

Listing 2.2: tsc register

2.3.3 Wichtige Kernel-Strukturen

Für das bessere Verständnis werden nun einige relevante Kernel-Strukturen vorgestellt. Es soll dabei nur ein Überblick über die für diese Arbeit nötigen Felder gegeben werden. Für eine detaillierte Auflistung sei hier auf die Kernel-Quellen auf verwiesen (The Linux Kernel Organization, Inc., 2013a).

struct sk_buff

Diese Struktur ist das Herzstück der Linux-Netzwerkkommunikation. Sie enthält unter anderem Netzwerkpakete und Informationen über das zugrunde liegende Netzwerkgerät. Sie ist definiert in <linux/skbuff.h>. (vgl. Corbet u. a., 2005, S.516,S.528ff)

```
struct sk_buff {
        /* These two members must be first. */
        struct sk_buff          *next;
        struct sk_buff          *prev;

        ktime_t                 tstamp;

        struct sock             *sk;
        struct net_device       *dev;
                        ⋮
}
```

Listing 2.3: struct sk_buff

Die Zeilen 3 und 4 enthalten die Adressen des vorhergehenden bzw. nachfolgenden Pakets innerhalb einer Liste. In Zeile 6 wird der Empfangszeitpunkt, der über die Funktion ktime_get_real() ermittelt wird, gespeichert. Zeile 8 enthält den Socket, an den dieser Socket-Buffer gebunden ist. Das struct net_device wird im weiteren Verlauf erklärt.

struct skb_shared_hwtstamps

Diese Struktur ist ebenfalls in <linux/skbuff.h> definiert und hält den Hardware-Zeitstempel des Pakets, falls dies sowohl von der Netzwerkkarte als auch der Kernel-Konfiguration unterstützt wird. Letztere ist in der Regel nicht standardmäßig aktiviert und muss daher explizit mit kompiliert werden. Eine genaue Beschreibung dazu folgt in Kapitel 5.

```
1 struct skb_shared_hwtstamps {
2         ktime_t hwtstamp;
3         ktime_t syststamp;
4 };
```

Listing 2.4: struct skb_shared_hwtstamps

hwtstamp enthält den von der Netzwerkkarte ermittelten exakten Hardware-Zeitstempel. Dieser Zeitstempel spiegelt die Zeitdauer seit dem aktivieren der Netzwerkkarte wider. Sie sind nur vergleichbar mit den Hardware-Zeitstempeln der selben Netzwerkkarte.

syststamp enthält den in Unix-Zeit umgewandelten hwtstamp. Dieser Wert ist eingeschränkt mit dem Software-Zeitstempel tstamp aus sk_buff vergleichbar. Die Genauigkeit hängt hier von der Umwandlung des hwtsamps in syswtstamp ab.

struct net_device

Netzwerkkartentreiber werden, im Gegensatz zu block- oder zeichenorientierten Treibern, in einer Datenstruktur organisiert. Jedes Netzwerk-Interface wird mit allen dazugehörigen Informationen in einer net_device Struktur gehalten, die in <linux/netdevice.h> definiert ist (vgl. Corbet u. a., 2005, S.502).

```
1 struct net_device {
2     char name[IFNAMSIZ];
3                   :
4 }
```

Listing 2.5: struct net_device

Für diese Arbeit ist lediglich das Feld „name“ interessant. Es enthält den Knoten unter dem das jeweilige Netzwerk-Interface zu erreichen ist.

struct igb_adapter

Bei der Struktur igb_adapter handelt es sich um eine privat deklarierte Struktur aus dem igb-Netzwerkkartentreiber der Firma Intel. Sie kann als Erweiterung der net_device Struktur betrachtet werden und enthält hardwarespezifische Informationen und Einstellungen für die unterstützten Netzwerkkarten. Der Treiber ist quelloffen und findet sich auch in den Kernel-Quellen unter <drivers/net/ethernet/intel/igb/igb.h>. Die zu diesem Zeitpunkt aktuelle Version 4.3.0 dient als Basis für diese Arbeit.

```
/* board specific private data structure */
struct igb_adapter {
        unsigned long active_vlans[BITS_TO_LONGS(VLAN_N_VID)];

        struct net_device *netdev;

        unsigned long state;
        unsigned int flags;

        unsigned int num_q_vectors;
        struct msix_entry *msix_entries;
                  :
}
```

Listing 2.6: struct igb_adapter

In dieser Arbeit wird sie verwendet um auf die net_device Struktur zuzugreifen. Die zusätzlichen Felder finden keine Anwendung.

struct hrtimer

Das Konzept der High-Resolution Timer wurde im Abschnitt Linux und die Zeit erläutert. Die Struktur wird hier der Vollständigkeit halber aufgelistet und ist in <linux/hrtimer.h> zu finden. Sie muss mit der Funktion hrtimer_init() initialisiert werden.

```
struct hrtimer {
        struct rb_node                  node;
        ktime_t                         _expires;
```

```
4         ktime_t                         _softexpires;
5         enum hrtimer_restart            (*function)(struct hrtimer *);
6         struct hrtimer_clock_base       *base;
7         unsigned long                   state;
8         struct list_head                cb_entry;
9 #ifdef CONFIG_TIMER_STATS
10        int                             start_pid;
11        void                            *start_site;
12        char                            start_comm[16];
13 #endif
14 };
```

Listing 2.7: struct hrtimer

Neben dem offensichtlich wichtigen Ablaufzeitpunkt in den Zeilen 3 und 4 sind hier hrtimer_restart in Zeile 5 und hrtimer_clock_base in Zeile 6 interessant. Ersterer hält den Zeiger auf die Funktion, die ausgeführt werden soll wenn der Timer abläuft. Letzterer die Zeitquelle des Timers. Der Ablaufzeitpunkt _softexpires beschreibt den absolut frühesten Zeitpunkt zu dem der Timer ablaufen soll, das Feld _expires die interne Darstellung des absoluten Ablaufzeitpunkts. Diese Zeit bezieht sich auf die Zeitquelle mit der der Timer gestartet wurde. Handelt es sich um einen absoluten Timer ist es identisch zu _softexpires.

struct hwtstamp_config

Die Struktur hwtstamp_config hält die Einstellungen für die Zeitstempelquelle der Netzwerkkarte. Sie ist in <linux/net_tstamp.h> definiert.

```
1 struct hwtstamp_config {
2         int flags;
3         int tx_type;
4         int rx_filter;
5 };
```

Listing 2.8: struct hwtstamp_config

Um den exakten, in Hardware ermittelten Empfangszeitpunkt, für alle eingehenden Pakete zu erhalten muss für rx_filter HWTSTAMP_FILTER_ALL und für tx_type HWTSTAMP_TX_ON eingestellt werden. Eine genaue Beschreibung und Auflistung aller Konfigurationsmöglichkeiten ist in </linux/Documentation/networking/timestamping.txt> zu finden. Das Feld Flags wird bis dato nicht verwendet.

3 Analyse

Da diese Arbeit auf Rick (2012) aufsetzt, soll zuerst ein Überblick über jene gegeben werden, bevor eine Anforderungsanalyse für diese Arbeit gemacht wird. Dazu wird zunächst das ursprüngliche Konzept betrachtet.

3.1 Implementierungsstand der zugrunde liegenden Arbeit

Bevor ein Konzept erstellt werden kann, ist es nötig, den aktuellen Stand der Implementierung zu analysieren. Dazu wurde der Arbeit folgende Aufzählung entnommen. Es soll dabei nur auf die für diese Arbeit nötigen Punkte eingegangen werden.

TTEthernet Controller für CT-Traffic

- TTE API implementieren (erfüllt)
- Konfiguration aus config.c Datei lesen (erfüllt)
- Senden von TT Nachrichten (erfüllt)
- Empfangen von TT Nachrichten (erfüllt)
- Zeitstempel (erfüllt)
- Filtern der Nachrichten nach Typ in Buffer (erfüllt)
- Scheduling der Tasks mit Prioritäten (erfüllt)
- TX TT (Task) (erfüllt)
- Sync (Task) (implementiert, eingeschränkt nutzbar)
- RX-TT Callback (Task) (nicht erfüllt)
- TT-Task (nicht erfüllt)
- TX-BE (Task) (erfüllt)
- RX-BE (Task) (erfüllt)
- Acceptance Window für CT Nachrichten (nicht erfüllt)
- BE Traffic unter Kontrolle des Schedulers stellen (erfüllt)

- Synchronisations Client (teilweise erfüllt)
- Synchronisations Master (nicht erfüllt)
- Synchronisations Compression Master (nicht erfüllt)

Die Konfiguration aus der config.c Datei zu lesen ist erfüllt. Das ist wichtig, da in dieser Datei die Buffer und die Zeitfenster für zeitkritischen Datenverkehr eingestellt werden, sowie die zeitgesteuerten Scheduler-Tasks. Genaueres dazu folgt später bei der Umsetzung.

Da die Zeitsynchronisation durch die sogenannten PCFrames realisiert wird, wie im Kapitel Grundlagen erläutert wurde, ist der Empfang, das Senden und das Filtern der Nachrichten nach Typ in den entsprechenden Buffer auch nötig. Dies ist ebenfalls bereits implementiert. Der Sync-Task, der ebenfalls wichtig ist, ist jedoch erst eingeschränkt nutzbar. Hier muss also die Umsetzung noch vervollständigt werden.

Es ist bereits ein „teilweise funktionierender“ Synchronization Client implementiert, dieser soll jedoch unter Verwendung der hoch-präzisen Hardware-Zeitstempel neu geschrieben und somit ersetzt werden.

Der zugrunde liegenden Arbeit ist ebenfalls zu entnehmen, dass das TTEthernet-Modul für den Linux-Kernel 3.0.29 entwickelt wurde. Da dieser Kernel mittlerweile veraltet ist muss überprüft werden ob das Modul für die aktuelle Version angepasst werden muss.

3.2 Beschreibung der TTEthernet-Treiberimplementierung

Um Modifizierungen des TTEthernet-Treibers vornehmen zu können ist es nötig die vorhandene Implementierung zu analysieren. Dabei soll zunächst ein kurzer Einblick in die Netzwerktreiber-Entwicklung unter Linux im Allgemeinen, sowie ein Überblick über Kernkomponenten des vorliegenden Treibers gegeben werden.

3.2.1 Grundlegendes Design eines Linux Netzwerktreibers

Linux Treiber, auch Module genannt, laufen im sogenannten Kernelspace, also in dem für den Kernel reservierten Speicherbereich. Nachfolgend werden die Unterschiede von Kernelspace und Userspace erläutert.

Kernelspace bezeichnet die Speicherbereiche, auf die Routinen innerhalb des Kernels direkt zugreifen können. Nur Code, der im Kernelkontext abläuft hat Zugriff auf den Kernelspace.

Userspace ist der Speicherbereich, auf den Applikationen direkt zugreifen können. Dabei läuft, sofern nicht explizit anders angegeben, jede Anwendung in ihrem eigenen virtuellen Speicherbereich und kann somit nicht ungewollt Daten überschreiben.

Ein Linux-Treiber inkludiert *<linux/init.h>*, sowie *<linux/module.h>*. Dazu besitzt jedes Modul eine Init- und eine Exit-Funktion, die beim Laden, und Entladen aufgerufen werden, und die jeweils über die Makros *module_init()* bzw. *module_exit()* als solche deklariert werden.

Wird der Zugriff auf eine Funktion oder eine Variable eines Moduls in einem anderen Modul benötigt, so kann dies mit dem Makro *EXPORT_SYMBOL(Funktion oder Variable)* durchgeführt werden. Dabei wird die Referenz auf die Funktion oder Variable, welche auf diese Weise exportiert wurden, in der sogenannten Kernel Symbol Table gespeichert, um sie so anderen Modulen bekannt zu machen.

Sichern kritischer Abschnitte mit Spinlocks

Zum Schützen kritischer Codebereiche und für exklusiven Zugriff auf Variablen implementiert der Linux-Kernel sogenannte Spinlocks. Diese sind in *<linux/spinlock_types.h>* definiert. Spinnlocks können im Gegensatz zu Semaphoren auch in Codeabschnitten benutzt werden, in denen kein *sleep* erlaubt ist, wie etwa in Interrupt-Funktionen. Sie bieten darüber hinaus eine erhöhte Performance.

Die Initialisierung kann zur Compile-Zeit mit

```
spinlock_t lock = SPIN_LOCK_UNLOCKED;
```

oder zur Laufzeit mit

```
void spin_lock_init(spinlock_t *lock);
```

durchgeführt werden.

Prinzipiell gibt es zwei Arten von Spinlocks. Listing 3.1 zeigt in den Zeilen 3 und 4 das Erhalten und Loslassen eines gewöhnlichen Spinlocks. Diese zeichnen sich durch eine hohe Performance aus. Zeile 6 und 7 zeigen die zweite Art. Sie unterscheiden sich dahingehend, dass sie zusätzlich das Auslösen aller lokalen Interrupts abschaltet und somit den durch den Lock gesicherten

Abschnitt nicht unterbrechbar macht. Hierzu wird beim Erhalten des Locks der Kontext in der Variable $flags$ gesichert und beim Loslassen damit wiederhergestellt.

```
1 unsigned long flags
2
3 spin_lock(&lock);
4 spin_unlock(&lock);
5
6 spin_lock_irqsave(&lock, flags)
7 spin_unlock_irqrestore(&lock, flags)
```

Listing 3.1: Spinlocks

Kompilieren eines Moduls

Die zu bevorzugende Methode ein Treiber Modul unter Linux zu kompilieren ist über ein sogenanntes Makefile. Listing 3.2 zeigt ein Minimalbeispiel. Wird das Modul über das Kernel-Build-System kompiliert, so ist die Variable *KERNELRELEASE* gesetzt. Diese Abfrage in Zeile 1 wird benötigt, wenn das Modul im Rahmen eines Kernel-Build-Vorgangs mit gebaut wird. In Zeile 2 wird der Modulname angegeben, in diesem Fall wird die hello.c kompiliert. Gibt es weitere .c Dateien, so werden diese wie in Zeile 3 dargestellt hinzugefügt. Zeile 9 führt schließlich den Befehl zum Kompilieren mit den zuvor ermittelten Parametern aus. Dazu wird in Zeile 5 das Verzeichnis für die Header des aktiven Kernels ermittelt und in Zeile 6 das aktuelle Arbeitsverzeichnis.

```
1 ifneq ($(KERNELRELEASE),)
2   obj-m := hello.o
3   module-objs := file1.o file2.o
4 else
5   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
6   PWD       := $(shell pwd)
7
8 default:
9   $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
```

Listing 3.2: Module Makefile

Laden und Entladen eines Moduls

Wurde der Treiber erfolgreich kompiliert, so muss dieser noch geladen werden damit er verwendet werden kann. Dies geschieht mit dem Befehl

```
$ insmod hello.ko
```

In diesem Beispiel wird das Modul *hello* aus dem Beispiel-Makefile des vorhergehenden Abschnitts geladen. Dabei wird der Code und die Daten in den Kernel geladen, welcher alle nicht aufgelösten Symbole mit seiner Symbol-Table verlinkt. Befindet sich das entsprechende Ziel nicht in der Table, so bricht der Befehl mit der Fehlermeldung *unresolved symbols* ab.

Eine weitere Möglichkeit einen Treiber zu laden ist über den Befehl

```
$ modprobe hello
```

Dieser funktioniert wie *insmod*, jedoch prüft er vorher ob alle Symbole vorhanden sind, wenn nicht wird der aktuelle Modulpfad nach Modulen durchsucht, welche diese Symbole exportieren und lädt diese ebenso in den Kernel.

Wieder entfernt werden kann der Treiber mit dem Befehl:

```
$ rmmod hello
```

3.2.2 Design des TTEthernet-Treibers

Abbildung 3.1 zeigt die Funktionsweise des virtuellen Interfaces, welches im TTEthernet-Treiber realisiert wurde. Ein virtuelles Netzwerkinterface kann dabei, wie ein physikalisches, im Userspace verwendet werden.

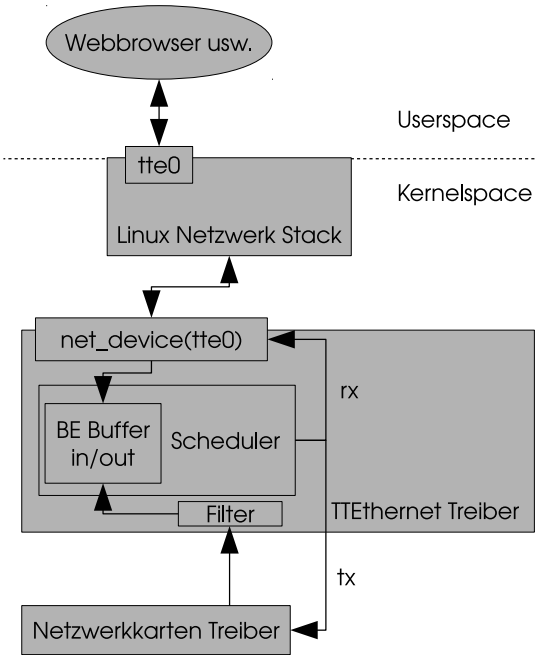

Abbildung 3.1: Aufbau des virtuellen Netzwerkinterfaces (vgl. Rick, 2012, S.32)

Der Treiber implementiert die Schnittstelle zum Linux-Netzwerk-Stack und registriert sich bei diesem als Netzwerkinterface, in diesem Fall tte0. Der TTEthernet Treiber bildet eine Schicht zwischen dem Treiber der physikalischen NIC und des Linux-Netzwerk-Stack. Ankommende Pakete werden nach Best Effort und zeitkritischem Datenverkehr gefiltert und in entsprechende Buffer zur Weiterverarbeitung einsortiert(vgl. Abbildung 3.2). Damit die Frames des normalen Netzwerkverkehrs die zeitkritischen Pakete nicht verdrängen, werden die BE-Nachrichten solange zwischengespeichert, bis ihnen Bandbreite im Netzwerk zugeteilt werden kann, dann werden sie an den Linux-Netzwerk-Stack zur Weiterverarbeitung weitergereicht.

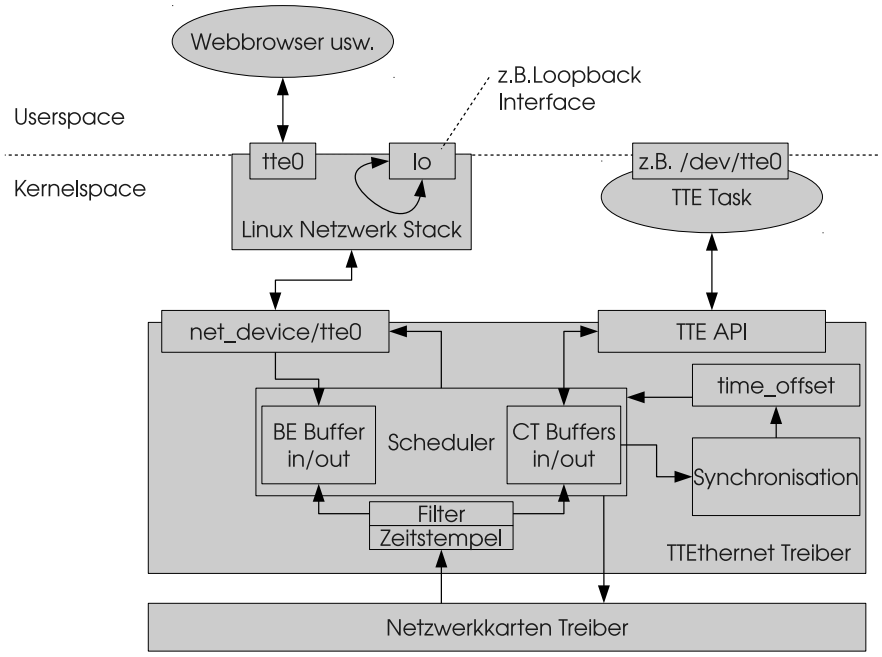

Abbildung 3.2: Aufbau des TTEthernet Treibers mit virtuellem Netzwerkinterface (vgl. Rick, 2012, S.33)

Der Treiber implementiert die TTEthernet API der Firma TTTech. Diese definiert Funktionen zur Konfiguration des Controllers und ermöglicht den Zugriff auf Ein- und Ausgabebuffer für Zeitkritischen Datenverkehr.

Implementierte Tasks und deren Priorisierung

Der TTEthernet-Treiber implementiert eine Reihe von Realtime-Tasks. Diese sind mit ihren Prioritäten im Folgenden dargestellt. Die Auflistung ist dabei (Rick, 2012, S.37) entnommen.

TX-TT, Prio 0: Der TX TT Task ist für das Versenden von CT Nachrichten zuständig und hat die höchste Priorität im TTE System.

Sync, Prio 1: Der Sync Task implementiert den Synchronisations-Algorithmus und ist für die Synchronisierung der Zykluszeit anhand der PCF zuständig.

RX-TT, Callback Prio 2: Der RX-TT Callback wird bei dem Empfang von CT Nachrichten ausgeführt. Er kann für jede Nachricht konfiguriert werden.

TT-Task, Prio 3: Ein TT-Task kann in der config.c angegeben werden und wird vom Scheduler zu gegebener Zeit ausgeführt.

TX-BE, Prio 4 : Der TX-BE Task ist für das Versenden von BE Nachrichten, welche, vom Linux Stack kommend, im BE Ausgangsbuffer gespeichert wurden, zuständig und hat die niedrigste Priorität im TTE System.

RX-BE, Prio 4: Der RX-BE Task ist für das Übergeben von BE Nachrichten, welche von der Empfangsroutine in den BE-Eingangsbuffer gespeichert wurden, zuständig und hat die niedrigste Priorität im TTE System.

Von besonderem Interesse für diese Arbeit ist der Sync-Task, da dieser für den Synchronization-Client verwendet wird. Er befindet sich in der *scheduler.c* und wird, wie in Listing 3.3 dargestellt, initialisiert.

```
1  #include <linux/kthread.h>
2
3  #define SYNC_CLIENT_TASK_PRIORITY 71;
4  DECLARE_WAIT_QUEUE_HEAD(sync_client_event);
5
6  struct task_struct *sync_client_kthread;
7
8  int init_scheduler(void){
9
10   struct sched_param param;
11
12   /* Declare tasks as a real time tasks and run them */
13   param.sched_priority = SYNC_CLIENT_TASK_PRIORITY;
14   tte_thread_init(&sync_client_kthread, sync_client, (void *)
         p_sync_frame_buffer, param);
15
16   int sync_client(void *data){
17                   ⋮
18   }
```

Listing 3.3: scheduler.c

Zeile 3 legt die Realtime Priorität fest und Zeile 4 legt statisch eine Warteschlange an, die dazu verwendet werden kann, einen Prozess schlafen zu legen. Zeile 6 deklariert den Pointer für den anzulegenden Threat. In Zeile 14 wird dann der Thread angelegt. Die Parameter dafür sind der Pointer auf den Thread, die auszuführende Methode (in diesem Fall sync_client), der Zeiger auf den Buffer für die PCFrames, sowie Konfigurationsparameter wie etwa die Priorität.

Der TTEthernet-Scheduler

Eine zentrale Komponente des TTEthernet-Treibers ist der Scheduler. Dieser ist dafür zuständig zu genau definierten Zeiten bestimmte Aktionen, wie etwa das Senden von PC-Frames, oder das Ausführen einer Routine, zu starten. Er ist dabei in dem Zyklus strukturiert, der durch die Periode der TT-Frames festgelegt ist. Hierzu arbeitet der Scheduler mit einem High Resolution Timer und einem internen Zyklus. Beim Initialisieren des Schedulers wird die aktuelle Systemzeit als Referenz für den eigenen Zyklus genommen, wie in Gleichung 3.1 dargestellt.

$$period\_start = ktime\_get(); \tag{3.1}$$

$$period\_start = ktime\_add\_ns(period\_start, period + ulSchedulerOffset); \tag{3.2}$$

$$abs\_time\_from\_next\_event = ktime\_add\_ns(period\_start, aSchedulerEvents[ucSchedulerCurrentEvent].ulPeriodTime); \tag{3.3}$$

Gleichung 3.2 zeigt den Beginn eines neuen Zyklus. Um den absoluten Zeitpunkt für dieses Ereignis zu ermitteln wird zu dem Anfangszeitpunkt des vorhergehenden Zyklus die Periodendauer addiert und wenn der Synchronization Client aktiv ist auch der Korrekturwert für den Offset. Mit dem berechneten Wert wird dann ein neuer High Resolution Timer initialisiert.

Gleichung 3.3 zeigt, wie die Zeit bis zum nächsten Event innerhalb eines Zyklus berechnet wird. Es wird dabei immer die Startzeit relativ zum Zyklusbeginn berechnet.

aSchedulerEvents[ucSchedulerCurrentEvent].ulPeriodTime beinhaltet hierzu immer den Startzeitpunkt des aktuellen Events relativ zum Zyklus. Läuft der High Resolution Timer ab, so wird die bei seiner Initialisierung angegebene Routine ausgeführt.

3.3 Betrachtung der Aufgabenstellung

Der vorhandene TTEthernet-Treiber soll zunächst auf den zu diesem Zeitpunkt aktuellen Linux Kernel 3.6 portiert und anschließend um Hardware-Zeitstempel erweitert werden. Dazu gilt es eine Netzwerkkarte zu verwenden, die in der Lage ist, alle eingehenden Ethernetpakete mit hoch präzisen Zeitstempeln zu versehen. Es ist eine Schnittstelle zu implementieren, so dass diese vom TTEthernet-Treiber verwendet werden kann. Zudem soll auch der Synchronization Client nach AS6802 implementiert werden um die Funktionalität dieser Schnittstelle unter Beweis zu stellen.

4 Konzept

Im Folgenden wird das Konzept erläutert. Dazu wird zunächst die Auswahl einer geeigneten Netzwerkkarte diskutiert und Architekturoptionen aufgezeigt. Danach wird die Verknüpfung des Treibers der Netzwerkkarte mit dem TTEthernet-Treiber dargelegt und aufbauend darauf die interne Synchronisation der Uhr der NIC mit der Linux-Systemzeit, sowie der Synchronization Client besprochen.

4.1 Auswahl der Netzwerkkarte

Die Auswahl der Netzwerkkarte ist ein zentraler Punkt der Arbeit. Die von ihr unterstützten Features und die Verfügbarkeit des Treiber-Quellcodes beeinflussen maßgeblich das gesamte Projekt.

Aus der Aufgabenstellung ergeben sich folgende Anforderungen an die Netzwerkkarte:

- Per-Packet Timestamping
- Quell-offener Treiber für Linux
- Auslesen der HW-Uhrzeit
- Interrupt-Generierung zu frei definierbaren Zeiten
- Verfügbarkeit als Erweiterungskarte für den Cardbus-Slot eines Notebooks

Die wichtigste Eigenschaft hierbei ist das Per-Packet-Tinestamping, für das exakte Erfassen des Empfangszeitpunktes eines Pakets. Auf den ersten Blick sehr vielversprechend wirken die Controller der Firma Napatech (www.napatech.com). Sie versprechen Hardware-gestützte Zeiterfassung für eingehende und ausgehende Pakete, jedoch wird der Quellcode für die jeweiligen Treiber nicht zur Verfügung gestellt. Dies ist eine weitere sehr wichtige Bedingung, da sich nur so eine Schnittstelle zum TTEthernet-Treiber bilden lässt. Eine Option wäre hier, den TTEthernet-Treiber in den Userspace zu portieren. Dies wird im nächsten Abschnitt über die Architekturoptionen weiter diskutiert.

Tabelle 4.1 zeigt einen Ausschnitt aus den Datenblättern aktueller Netzwerkkarten der Firma Intel. Diese lassen die Controller mit den folgenden Bezeichnungen übrig, da nur sie das Per-Packet-Timestamping unterstützen: i210, i211, i350, 82580. Diese Adapter unterstützen Hardware-Zeitstempel für alle eingehende Pakete, jedoch nicht für ausgehende. Auch ist der Quellcode der jeweiligen Treiber offen verfügbar und bereits in den Kernel-Sourcen enthalten.

Feature	i350	82580	82599	82576	i210	i211	82574
...							
IEEE 1588	Y	Y	Y	Y	Y	Y	Y
Per-Packet Timestamp	Y	Y	N	N	Y	Y	N
...							

Tabelle 4.1: Ausschnitt der Features der Netzwerkarten 1(vgl. Intel Corporation, 2013b, S.46) und (vgl. Intel Corporation, 2013a, S.13)

Durch die Betrachtung dieses Treiber-Quellcodes kann ein weiterer Punkt bei den Anforderungen abgehakt werden: Das Auslesen der HW-Zeit. Dies wird in der *igb_ptp.c*, sofern der Treiber mit PTP-Unterstützung kompiliert wurde, umgesetzt. Dies geschieht über das Compiler-Flag *IGB_PTP*.

```
$ make CFLAGS_EXTRA="-DIGB_PTP" install
```

Mit dem Befehl

```
$ ethtool -T ethX
```

kann ein Überblick über unterstütze Fähigkeiten der Netzwerkkarte und des Treibers ausgegeben werden. Damit kann überprüft werden, ob die PTP-Unterstützung aktiv ist.

Die Anforderung, dass der Controller als Cardbus-Karte verfügbar sein soll, wird von keiner der in Frage kommenden Adapter erfüllt. Die Interrupt-Generierung zu frei definierbaren Zeiten wird ebenfalls nicht unterstützt. Betrachtet man die Dokumentation für die PTP-Uhren-Infrastruktur im Kernel, sieht man die möglichen Funktionen einer PTP-Uhr. Ein Ausschnitt davon ist in Abbildung 4.1 dargestellt.

- Grundfunktionen der Uhr
  - Uhrzeit setzen
  - Uhrzeit auslesen
  - Atomares Verschieben um einen Offset
  - Anpassen der Frequenz
- Ergänzende Funktionen
  - Ein kurzer oder periodischer Alarm mit Signalübertragung an ein Programm
  - Zeitsempel für externe Ereignisse
  - Aus dem Userspace konfigurierbare periodische Ausgangssignale
  - Synchronisation der Linux-Systemzeit über das PPS Subsystem

Abbildung 4.1: Dokumentation für die PTP-Uhren-Infrastruktur im Kernel (The Linux Kernel Organization, Inc., 2013b)

Das Datasheet zu dem Intel Controller i350 legt nahe dass die Interrupt-Generierung theoretisch möglich ist, jedoch diese sogenannten ergänzenden Funktionen im Treiber zu diesem Zeitpunkt nicht implementiert sind. In einem späteren Release des Treibers ist es jedoch durchaus möglich dass dies von Seiten des Herstellers nachgereicht wird. Damit, und da durch die Aktualität des Chipsatzes eine bessere Verfügbarkeit gewährleistet ist, fällt die Wahl auf den Netzwerkcontroller i350 der Firma Intel.

4.2 Beschreibung der Architekturoptionen

Aus den Anforderungen und der zur Verfügung stehenden Hardware ergeben sich einige alternative Optionen für die Umsetzung, auf die im nachfolgenden Abschnitt eingegangen werden soll. Hier soll zunächst auf die grundlegende Designfrage, das Portieren des TTEthernet-Treibers in den Userspace eingegangen werden. Die weiteren Architekturoptionen werden in ihren jeweiligen Abschnitten erläutert.

4.2.1 Portierung des TTEthernet-Treibers vom Kernelspace in den Userspace

Grundsätzlich besteht die Möglichkeit, den TTEthernet-Treiber in ein Userspace-Programm umzuwandeln. Dies würde einige Vor- bzw. Nachteile mit sich bringen.

Vorteile: Die Schnittstelle zu den PTP-Funktionen der Netzwerkkarte sieht nur einen Zugriff aus dem Userspace über die Linux-Systemcalls vor. Wird das TTEthernet Modell als

Treiber realisiert, so muss selbst eine Schnittstelle angelegt werden und somit der Quellcode des Intel-Treibers verändert werden.

Ein weiterer Vorteil wäre die Unabhängigkeit von dem Treiber einer physikalischen Netzwerkkarte, was die Verwendung eines Adapters mit einem Closed-Source-Treiber, wie etwa einer der zuvor erwähnten Firma Napatach, ermöglichen würde.

Nachteile: Die Portierung in den Userspace würde allerdings auch einige Nachteile mit sich bringen. Ein wesentlicher Gedanke des ursprünglichen Konzepts war es, die Empfangsroutine des Netzwerkkarten-Treibers zu überschreiben, um schnellst möglich an die ankommenden Pakete zu kommen und sie dann entsprechend ihrer Priorität zu behandeln. Dieser Vorteil würde verloren gehen, da die Pakete dann über den Kernel in die Anwendung gelangen würden und so eine zusätzliche Verzögerung eintritt.

Darüber hinaus wird von der packet-capture Bibliothek libpcap unter Linux bis dato keine Nanosekunden-Genauigkeit unterstützt. Diese Bibliothek stellt ein Interface für den Empfang von Paketen im Userspace bereit. Die Bibliothek nimmt die Pakete vom Treiber entgegen und leitet sie an die entsprechende Anwendung weiter. Wird das TTEthernet-Modell als Anwendung umgesetzt, kann sie nicht als Paketquelle für libpcap fungieren und eine Analyse des zeitkritischen Datenverkehrs mit Programmen wie TCPDUMP oder Wireshark wäre kaum möglich, da keine Unterscheidung bei den empfangenen Paketen stattfindet. Ebenso müssten die empfangenen Daten nachträglich selbst gefiltert werden.

Werden die Vor- bzw. Nachteile gegeneinander abgewogen, so überwiegen die Nachteile. Daher wird dass TTEthernet-Modul nicht in den Userspace portiert und stattdessen muss ein Konzept gefunden werden, wie aus dem Kernelspace auf die PTP-Funktionen der Netzwerkkarte zugegriffen werden kann. Ebenso ergeben sich aus dieser Entscheidung einige Anforderungen an den Linux-Kernel, auf die im folgenden Abschnitt weiter eingegangen wird.

4.2.2 Anforderungen an den Kernel

Bisher haben sich einige Anforderungen an die Kernel-Optionen ergeben, die standardmäßig nicht aktiviert sind. Zunächst muss der Realtime-Patch installiert werden und die Optionen für Fullpreemtive-Scheduling und High Resolution Timer aktiviert werden. Des weiteren müssen, laut der offiziellen Webpräsenz des Linux Realtime Patches alle Optionen für das Energiesparen

wie etwa APM abgeschaltet werden (vgl. Ts'o u. a., 2013). Tabelle 4.2 fasst die nötigen Optionen zusammen.

Option	Aktivieren
CONFIG_PREEMPT_RT_FULL	Y
CONFIG_HIGH_RES_TIMERS	Y
APM und andere Energiesparmaßnahmen	N

Tabelle 4.2: RT-Kernel-Optionen (vgl. Ts'o u. a., 2013)

Darüber hinaus muss, wie bereits festgestellt, der Intel-igb-Treiber mit PTP-Untertützung kompiliert werden, um Zugriff auf die Zeit-relevanten Methoden zu bekommen. Auch im Kernel selbst muss die Unterstützung für PTP erst aktiviert werden, da dies vom igb-Treiber gefordert wird. Tabelle 4.3 zeigt die dafür benötigten Kernel-Optionen.

Option	Beschreibung
CONFIG_PPS	Required
CONFIG_NETWORK_PHY_TIMESTAMPING	Timestamping in PHY devices
PTP_1588_CLOCK	PTP clock support

Tabelle 4.3: PTP-Kernel-Optionen (vgl The Linux PTP Project, 2013)

4.3 Anbindung des NIC-Treibers an den TTE-Treiber

Als erstes gilt es den alten Netzwerkkartentreiber e1000, ebenfalls für Intel-Karten, durch den neuen igb-Treiber zu ersetzen. Dazu gilt es lediglich, wie in der Arbeit von Rick Frieder beschrieben, die Empfangsroutinen des Treibers durch die des TTEthernet-Moduls zu ersetzen. Darauf wird bei der Umsetzung weiter eingegangen.

Im Idealfall kann die Netzwerkkarte als alleinige Zeit- und Interrupt-Quelle fungieren. Da aber die ergänzenden Funktionen im igb-Treiber vom Hersteller, wie bereits erwähnt, nicht implementiert sind und somit der Netzwerkadapter nicht zur Interrupt-Generierung genutzt werden kann, muss hierzu auf die High Resolution Timer von Linux zurückgegriffen werden. Ebenso entsteht dadurch die Notwendigkeit, die Uhr der Netzwerkkarte mit der Linux-Systemzeit zu synchronisieren um die hardware-generierten Empfangszeiten der Pakete mit den Systemzeiten vergleichen zu können.

Um eine Synchronisation der beiden Zeitquellen vorzunehmen, ist es nötig auf die Zeitfunktionen der Netzwerkkarte zugreifen zu können. Vorgesehen ist die Nutzung der PTP-Uhr über die Systemcalls aus dem Userpace. Eine Schnittstelle für den Zugriff von einem anderen Modul aus ist nicht vorgesehen. Es muss also eine solche Schnittstelle definiert werden.

Dazu gibt es zwei Optionen. Zum einen ist es möglich, die benötigten Funktionen einzeln mit dem bereits vorgestellten Makro *EXPORT_SYMBOL()* anderen Modulen bekannt zu machen. Diese Vorgehensweise hat jedoch einen großen Nachteil. Bei einer neuen Treiber-Version müssen alle Anpassungen erneut durchgeführt werden.

Die benötigten Methoden implementieren Registerzugriffe, sowie die Steuerung der Hardware der NIC. Da sich die Adressen der Register und die Steuerungscodes innerhalb einer Hardwarerevision nicht ändern, stellt das Neuimplementieren der Funktionen innerhalb des TTEthernet-Treibers eine bessere Alternative dar. Dazu wird die private Struktur *igb_adapter* benötigt, die ebenfalls nach außen geführt und somit bekannt gemacht werden muss.

4.4 Zeitsynchronisation

Nachdem nun eine Netzwerkkarte ausgewählt und eine Strategie für ihre Anbindung an den TTEthernet-Treiber erarbeitet wurde, ist der nächste Schritt die Zeitsynchronisation. Zum einen gilt es die Hardware-Uhr der Netzwerkkarte mit der Linux-Softwarezeit zu synchronisieren, zum anderen das System mit dem TTE-Netzwerk.

4.4.1 Synchronisation der Hardware-Uhr der NIC mit der Linux-Softwarezeit

Die interne Uhren-Synchronisation soll über eine sogenannte Rate-Correction stattfinden. Das bedeutet, dass die Geschwindigkeit einer der beiden Uhren so verändert wird, bis beide gleich schnell laufen. Damit kann sichergestellt werden, dass nach einer gewissen Zeitspanne die selbe Zeit auf den beiden Uhren vergangen ist, jedoch nicht, dass sie die gleiche Uhrzeit anzeigen. Das Ziel ist es die Werte der beiden Zeitquellen in Relation setzen zu können und dazu muss deren Differenz mit eingerechnet werden.

Prinzipiell besteht die Möglichkeit, diese entweder als Scheduler-Task im TTEthernet-Treiber oder als eigenen Kernel-Thread mit einem High-Resolution Timer zu implementieren. Der Vorteil bei einem Scheduler-Task ist, dass der Zeitpunkt der Rate-Correction innerhalb des

TT-Zyklus genau festgelegt werden kann und dadurch eine verlässlichere Zeitbasis für alle Empfangszeitpunkte der Pakte entsteht. Allerdings führt dies zu einer Abhängigkeit vom Scheduler und damit von dessen Zyklus. Da die Uhrenanpassung bei der Umsetzung des Synchronization Client über das Verkürzen bzw. Verlängern der Periodendauer stattfinden wird, hat dies ebenfalls unerwünschte Auswirkungen.

Daher ist die Umsetzung als eigenen Thread die zu bevorzugende Alternative. Die interne Zeitsynchronisation ist damit unabhängig vom Rest der TTEthernet-Implementierung und kann gleich beim Laden des Moduls gestartet werden. Bei dieser Variante ist es ebenfalls möglich, sie Häufigkeit der Frequenzanpassung der Uhr über die Periodendauer des High-Resolution-Timers anzupassen.

4.4.2 Synchronisation mit dem Netzwerk

Nachdem eine einheitliche Zeitbasis der beiden internen Uhren geschaffen wurde, kann diese mit der „Außenwelt“, also dem TTEthernet-Netzwerk, synchronisiert werden. Dies geschieht über den Synchronization Client nach AS6802, welcher im nächsten Abschnitt erläutert wird.

Wichtig für den sogenannten Receive Point In Time ist der exakte Empfangszeitpunkt eines Frames. Er ist definiert als der Zeitpunkt, zu dem der Synchronization Client die stehende Flanke des ersten Bits des ersten Symbols nach dem „Start Of Frame“ Feld registriert (vgl. SAE Aerospace AS6802, 2011, S.28). Der exakte Zeitpunkt, zu dem der Empfang eines Frames in der Netzwerkkarte erfasst wird ist in Abbildung 4.2 dargestellt.

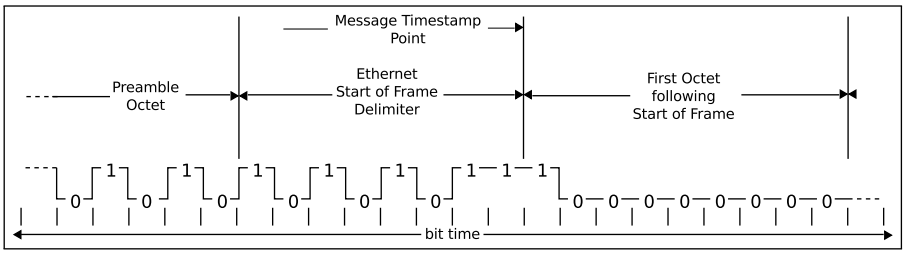

Abbildung 4.2: Zeitstempel Zeitpunkt (vgl. Intel Corporation, 2013b, S.461)

Dieser Zeitpunkt ist ideal für den Receive Point In Time und somit können die Zeitstempel der Netzwerkkarte unverändert verwendet werden.

Synchronization Client

Der Synchronization Client sieht drei Zustände vor, die in Abbildung 4.3 dargestellt sind.

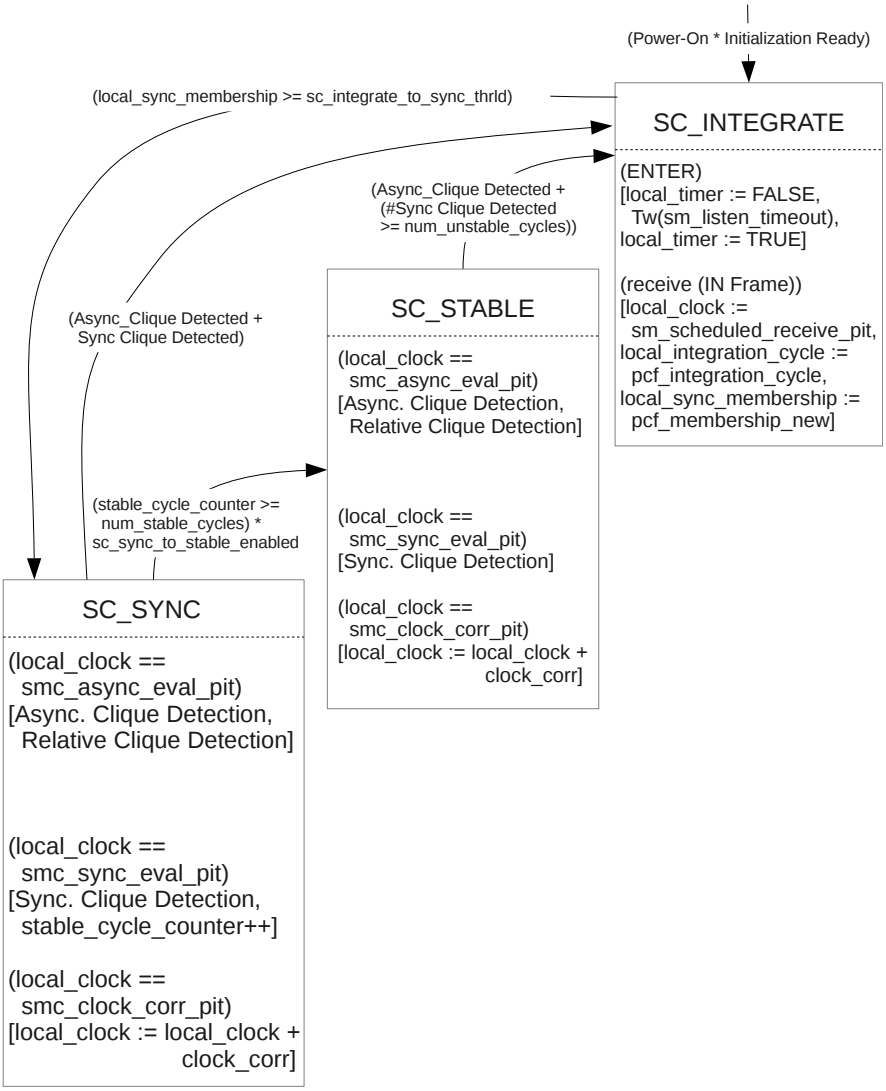

Abbildung 4.3: Statemachine des Synchronization Client (vgl. SAE Aerospace AS6802, 2011, S.65)

SC_INTEGRATE: Dies ist der asynchrone Startzustand des Systems. Er dient zur Erkennung aktiver, bereits synchronisierter, TTEthernet Geräte im Netzwerk. Ist das der Fall, so werden IN-Frames empfangen, anhand derer der Offset des eigenen Zyklus angepasst wird. Daraufhin wird direkt in den Zustand SC_SYNC gewechselt.

SC_SYNC: Der SC_SYNC Zustand ist ein synchroner Zustand, daher laufen sowohl die Uhren-Synchronisation als auch die sogenannte Cliquen-Erkennung. Letztere wird nicht implementiert, da in dem vorhandenen TTEthernet-Netzwerk die Cliquen-Erkennung nicht unterstützt wird und so die Funktionalität nicht überprüft werden kann. Die Uhren-Synchronisation stellt sicher, dass die *local_clock* und der *local_integration_cycle* im Synchronization Client aktualisiert werden. Gab es mindestens *num_stable_cycles*, so wird in den Zustand SC_STABLE gewechselt.

SC_STABLE: Im Grunde ist der Zustand SC_STABLE identisch mit SC_SYNC und zeigt lediglich an, dass sich das System in einem stabilen Zustand befindet. Wird eine bestimmte, konfigurierbare Anzahl an asynchronen Zyklen festgestellt, so wird wieder in den Zustand SC_INTEGRATE gewechselt.

Abbildung 4.4 zeigt die lokale Uhr des Synchronization Clients und ihren Anpassungszeitpunkt. Zunächst werden kurz die einzelnen Variablen erläutert.

local_clock ist die aktuelle Zeit innerhalb eines TT-Zyklus. Sie wird mit den Uhren der anderen Geräte synchronisiert.

sm_dispatch_pit ist der interne Zeitpunkt, zu dem der Scheduler des Synchronization Masters das Versenden eines PCFrames auslöst.

max_transmission_delay wird offline ermittelt und stellt die maximale Übertragungszeit eines PCFrames innerhalb des TT-Netzwerks dar.

compression_master_delay ist die maximale vom CM verursachte Verzögerung. Sie ist ebenfalls offline statisch konfiguriert und dadurch gegeben.

local_integration_cycle dies ist die lokale Zyklus-Nummer in der sicher der SC befindet, sie wird mit jeder neuen Periode hochgezählt.

max_integration_cycle ist die maximale Anzahl an Integrationszyklen, danach wird wieder bei 0 begonnen.

smc_permanence_pit Das smc_permanence_pit beschreibt den Zeitpunkt, zu dem ein eingehender PCFrame permanent wird, also tatsächlich empfangen wurde. Er ist definiert als permanence_pit = smc_receive_pit + permanence_delay, wobei sich das permanence_delay aus max_transmission_delay - PcfTransparentClock zusammensetzt.

smc_scheduled_pit beschreibt den Zeitpunkt, zu dem die PCFrames ankommen sollen. Es ist dabei definiert als smc_scheduled_pit = period_start + 2 * max_transmission_delay + compression_master_delay.

acceptance_window Es wird statisch eine bestimmte Genauigkeit für eingehende Pakete definiert. Diese Genauigkeit ergibt mit dem smc_scheduled_pit das Zeitfenster, in dem die ankommenden Frames als „rechtzeitig empfangen“ bewertet werden.

clock_corr_delay Eine Zeitkorrektur findet nicht unmittelbar statt, sondern zu einem später im Zyklus befindlichen Zeitpunkt. Diese Verzögerung wird als clock_corr_delay bezeichnet.

clock_corr bezeichnet den Wert um den die lokale Uhrzeit korrigiert werden muss.

smc_clock_corr_pit stellt den Zeitpunkt der tatsächlichen Uhren-Anpassung dar.

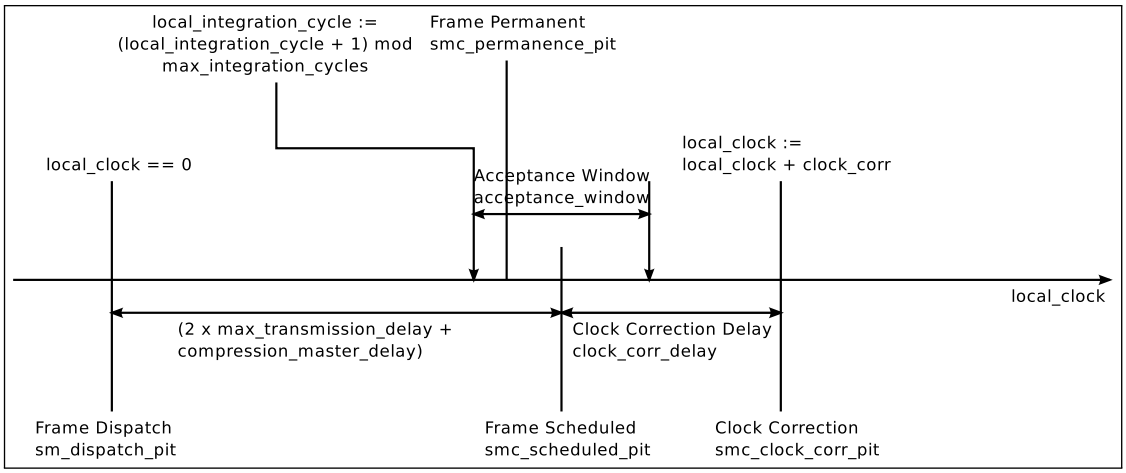

Abbildung 4.4: Lokale Uhr im Synchronization Client (vgl. SAE Aerospace AS6802, 2011, S.44)

Das ursprüngliche Konzept berücksichtigt bereits einen Synchronization Client und wird demnach wieder aufgegriffen. Schematisch ist dies in Abbildung 3.1 auf Seite 24 dargestellt. Die tatsächliche Implementierung wurde im Scheduler vorgenommen. Diese wird in eine eigene synchronisation.c Datei ausgelagert, die sowohl für die interne, als auch die externe Synchronisation zuständig ist.

5 Realisierung der Treibererweiterung

Dieses Kapitel befasst sich mit der Umsetzung des Konzepts. Dabei wird zunächst erläutert, wie die Anbindung des NIC-Treibers an den TTE-Treiber umgesetzt wurde und danach die Implementierung sowohl der internen Synchronisation der beiden Uhren, sowie die Synchronisation mit anderen Geräten im TT-Ethernet Netzwerk, beschrieben.

5.1 Anbindung des NIC-Treibers an den TTE-Treiber

Der Netzwerkkartentreiber des realen Interfaces liefert empfangene Pakete an den Linux-Kernel weiter. Von hier kann das Paket dann an eine Anwendung weitergereicht werden. Dieser Zwischenschritt durch den Kernel verursacht unvorhersehbare Latenzen, welche problematisch für zeitkritischen Datenverkehr sind. Um diese Latenzen zu vermeiden, müssen einige Methoden des Intel-Treibers überschrieben werden (vgl. Rick, 2012, S.46f). Dadurch werden die Empfangsroutinen übergangen und die Nachrichten gehen direkt an das virtuelle Interface. Die Anpassungen sind in Listing 5.1 dargestellt.

```
1 /*
2  * TTE Extension
3  */
4 #include "../../tte.h"
5 #include "../../shared.h"
6 #define HAVE_PTP_1588_CLOCK
7 #define netif_receive_skb tteif_receive_skb
8 #define netif_rx tteif_rx
9 #define napi_gro_receive ttenapi_gro_receive
```

Listing 5.1: Ausschnitt aus der igb.h

Bei der Anbindung des Netzwerkkarten-Treibers an den TTE-Treiber gibt es einige Probleme zu bewältigen. Um auf die Funktionen der Intel-Netzwerk-Interfaces zugreifen zu können, wird, wie im Kapitel 4 Konzept diskutiert, die private Struktur *igb_adapter* öffentlich zugänglich

gemacht. Die verwendete Netzwerkkarte gibt es jedoch sowohl mit zwei als auch mit vier Ports. Um diese Verwalten zu können werden sie in einer vom Kernel bereitgestellten Linked List gehalten. Dies ist in Listing 5.2 dargestellt.

```
1  static int __igb_open(struct net_device *netdev, bool resuming)
2  {
3    struct igb_adapter *adapter = netdev_priv(netdev);
4    struct e1000_hw *hw = &adapter->hw;
5
6                    ⋮
7
8    /*My TTE START*/
9    struct my_tte_igb_adapter *tmp;
10   tmp = (struct my_tte_igb_adapter *) kmalloc(sizeof(struct ↩
         my_tte_igb_adapter), GFP_USER);
11   tmp->tte_adapter = adapter;
12   list_add(&tmp->mylist, &my_linked_list);
13   /*My TTE END*/
14
15                   ⋮
```

Listing 5.2: igb_main.c

Da sich der TTE-Treiber mit nur einem Netzwerkinterface verbindet, wurde die Option implementiert diesen als Parameter beim Laden des Moduls zu übergeben. In Listing 5.3 ist dargestellt, wie dies umgesetzt wurde.

```
1 char *ifname = "eth0";
2 module_param(ifname, charp, 0);
3 MODULE_PARM_DESC(ifname, "phys␣device␣name");
```

Listing 5.3: Modul Übergabeparameter in tte_main.c

Wird kein Parameter beim Laden des Moduls angegeben, so wird der Standardwert „eth0“ aus Zeile 1 verwendet. Zeile 2 enthält das Makro mit dem der Parameter definiert wird. Der erste Wert ist der Name des zu übergebenden , der zweite ist der Typ und der dritte legt die Zugriffsrechte für sysfs fest (0 deaktivert den Eintrag in sysfs komplett). Sysfs ist ein Virtuelles Dateisystem, das Information über Geräte und Treiber im Userspace zur Verfügung stellt. Das Makro in Zeile 3 enthält die Beschreibung für den Modulparameter. Mit dem Befehl

```
$ insmod modulname ifname=ethX
```

kann somit dem TTEthernet-Treiber beim Laden die gewünschte Ethernet-Schnittstelle mitgeteilt werden. Die zur Verfügung stehenden Schnittstellen können nach dem Laden des Intel-Treibers mit

```
$ ifconfig
```

ermittelt werden.

5.2 Zeitsynchronisation

Nachfolgend wird die Umsetzung der Zeitsynchronisation erläutert. Bevor die Synchronisation der Hardware-Uhr der Netzwerkkarte mit der Linux-Softwarezeit und des Synchronization Clients betrachtet wird, soll zunächst die Schnittstelle zu den Timerfunktionen der Netzwerkkarte vorgestellt werden.

5.2.1 Schnittstelle zu den Timerfunktionen der Netzwerkkarte

Für die Schnittstelle zu den PTP-Timerfunktionen der Netzwerkkarte wurden die Methoden, wie im Konzept diskutiert, neu implementiert. Da der Intel igb-Treiber ebenfalls für andere Controller-Modelle gedacht ist, ist auch viel Code enthalten, der die Hardwarezugriffe für diese Modelle steuert. Dieser überflüssige Quellcode zu deren Unterstützung wurde der Übersichtlichkeit halber, für weitere aufbauende Arbeiten, entfernt.

Umgesetzt wurden nachfolgende Methoden, auf die nun näher eingegangen wird.

int activate_hwtstamp(struct hwtstamp_config*): Mit dieser Methode werden die Einstellungen für die Zeitstempel-Quellen gesetzt und aktiviert. Diese Einstellungen werden in Form einer struct hwtstamp_config (beschrieben in Listing 2.8) übergeben. Der Rückgabewert zeigt an, ob die Aktivierung erfolgreich war.

u64 hwtime_read(void): Diese Methode liest das Timerregister der Netzwerkkarte aus und gibt den Zeitwert in Nanosekunden seit dem Aktivieren des Netzwerkinterfaces zurück.

int hwtime_adjfreq(s32): Hiermit kann die Geschwindigkeit der Netzwerkkarten-Uhr beeinflusst werden. Wird ein negativer Wert übergeben, läuft die Uhr langsamer, bei einem positiven Wert schneller. Eine genaue Beschreibung der Zeitanpassung kann dem nachfolgenden Abschnitt entnommen werden. Der Rückgabewert zeigt an, ob die Anpassung erfolgreich war.

void hwtime_write(const struct timespec*): Mit dieser Funktion kann das Timerregister der Netzwerkkarte verändert werden. Sie erwartet hierfür einen Zeitwert in Form eines struct timespec, der zwei 64 Bit Integer, Sekunden und Nanosekunden, enthält. Sie ist der Vollständigkeit halber aufgeführt, findet in dieser Arbeit jedoch keine Verwendung. Die Uhr der Netzwerkkarte darf nicht sprunghaft verändert, bzw. in die Vergangenheit korrigiert werden, da dies zu Sprüngen in den Empfangszeitpunkten der Paketen oder mehrmals des selben Zeitstempels führen kann und somit die Zeitkonsistenz verloren geht.

void systim_to_hwtstamp(struct skb_shared_hwtstamps*, u64): Da die Funktion hwtime_read(), wie bereits erwähnt, den aktuellen Zeitwert nur in Nanosekunden seit dem Aktivieren des Netzwerkinterfaces zurück gibt kann dieser nicht mit anderen Zeitwerten verglichen werden. Deshalb kann dieser Zeitwert mit dieser Methode in Unix-Zeit umgewandelt werden. Sie erwartet hierzu den Zeitwert von hwtime_read() und einen Zeiger auf die zu füllende Struktur skb_shared_hwtstamps, auf die bereits in Kapitel 2 eingegangen wurde.

5.2.2 Synchronisation der Hardware-Uhr der NIC mit der Linux-Softwarezeit

Die Synchronisation der beiden Zeitquellen wird über eine Rate-Correction realisiert. Hier wird die Geschwindigkeit der Netzwerkkarten-Uhr an die Linux-Software-Uhr angepasst. Um eine möglichst genaue Zeitbasis zu erhalten muss die unpräzisere Uhr nach der präziseren gestellt werden, wobei hier die Hardware-Uhr der Netzwerkkarte die präzisere und auf Grund der Natur einer Software-Uhr, die Linux-Uhr die unpräzisere darstellt. Dies kann hier nicht so umgesetzt werden, da die Linux-Software-Uhr später mit den PCFrames und somit dem Rest des TTEthernet-Netzwerkes synchronisiert wird. Auch basiert die Zeitquelle des Schedulers auf der Software-Uhr. Diese außerhalb des durch den Synchronization Clients vorgesehenen Zeitfensters anzupassen kann zu einer Störung des TT-Zyklus' führen.

Für die Zeitanpassungsfunktion wird ein periodischer High-Resolution-Timer konfiguriert, der bei der Initialisierung des Schedulers gestartet wird. Dabei wird alle 5ms die Linux-Systemzeit ausgelesen und die Differenz zum vorhergehenden Zeitstempel gebildet. Diese Periodendauer kann variiert werden und als Maximalwert max(signed 32Bit) annehmen.Eine kürzere Frequenz führt zu einer schnelleren Synchronität, findet die Anpassung jedoch zu häufig statt, so müssen sehr viele Kontextwechsel in dem Thread stattfinden. Dies kann sich negativ auf die

Performance auswirken. Sinnvolle Werte sind daher im einstelligen Millisekunden Bereich. Gleiches geschieht mit der Hardware-Uhr der Netzwerkkarte. So kann festgestellt werden wie viel Zeit auf beiden Uhren tatsächlich vergangen ist. Laufen die beiden Uhren gleich schnell, so sind diese beiden Deltas gleich groß. Ist jedoch das Delta der Netzwerkkarten-Zeit kleiner, so läuft die Uhr der NIC zu langsam. Ist es größer, so ist die Uhr zu schnell. In beiden Fällen muss das *TIMINCA* Register (beschrieben in Tabelle 5.1) der Netzwerkkarte angepasst werden. Wie die Gleichung in Abbildung 5.1 zeigt, wird mit dem *TIMINCA* Register die Geschwindigkeitsanpassung realisiert.

Field	Bit(s)	Initial Value	Description
Incvalue	30:0	0x0	Increment value. Value to be added or subtracted (depending on ISGN value) from 8 ns clock cycle in resolution of $2^{-32}ns$.
ISGN	31	0b	Increment sign. 0 - Each 8 nS cycle add to SYSTIM a value of $8ns + Incvalue * 2^{-32}ns$. 1 - Each 8 nS cycle add to SYSTIM a value of $8ns - Incvalue * 2^{-32}ns$.

Tabelle 5.1: Increment Attributes Register - TIMINCA (vgl. Intel Corporation, 2013b, S.604)

$$NewSYSTIM = OldSySTIM + 8\text{ ns} + \textbf{TIMINCA}\cdot 2^{-32}\text{ ns}$$

Abbildung 5.1: Berechnung des Zeitwertes des Zeitregisters (vgl. Intel Corporation, 2013b, S.465)

Gemäß Abbildung 5.1 wird alle 8ns das Timerregister um 8ns erhöht und der Wert des *TIMINCA* Registers $\cdot 2^{-32}$ addiert oder subtrahiert.

Die Anpassung soll anhand eines Beispiels demonstriert werden. Soll die Uhr um 1% schneller laufen ergibt sich die folgende Rechnung. Es sei noch angemerkt, dass sich diese 1% Geschwindigkeitsveränderung auf die ursprüngliche Startgeschwindigkeit bezieht und nicht auf die aktuelle, evtl. schon angepasste Geschwindigkeit.

$$8 \cdot 1{,}01 = 8 + TIMINCA_i \cdot 2^{-32} + TIMINCA_{i-1} \cdot 2^{-32}; für\ i \in [1; \infty[$$

Im Anfangszustand ist $TIMINCA_0 = 0$. Löst man nun nach *TIMINCA* auf ergibt sich diese Gleichung:

$$TIMINCA = (8{,}08 - 8) \cdot 2^{32} = 343.597.383{,}68$$

Wird nun dieser Wert in das Register geschrieben, so läuft die Uhr um die geforderten 1% schneller. Daraus lässt sich eine allgemeine Formel für den Registerwert ableiten:

$$TIMINCA_i = \sum_{1 \leq i}^{\infty} \frac{Zeit_{soll} - Zeit_{ist}}{Zeit_{ist}} \cdot 2^{32} + TIMINCA_{i-1}$$

Aus dieser Gleichung ergibt sich nun jedoch ein Problem: Im Kernel ist eine Fließkommadivision nicht vorgesehen. Die möglichen Alternativen wären das Shiften um Zweierpotenzen, das aber zu ungenauen Ergebnissen führt und das Herantasten an den richtigen Registerwert durch kleine Änderungen. Hier wird letzteres favorisiert, da es zu einer deutlich einfacheren Anpassungsfunktion führt, wie der folgende Quellcodeausschnitt zeigt.

```
1  enum hrtimer_restart synchronization_intern_callback(struct hrtimer *
       timer) {
2    unsigned long flags;
3    struct skb_shared_hwtstamps shhwtstamps;
4    ktime_t currtime;
5    ktime_t interval;
6
7    spin_lock_irqsave(&mr_lock, flags);
8    /* get current time */
9    systimestamp = ktime_get_real();
10   hwtimestamp = hwtime_read();
11   spin_unlock_irqrestore(&mr_lock, flags);
12
13   systim_to_hwtstamp(&shhwtstamps, hwtimestamp);
14   hwtimestamp = shhwtstamps.hwtstamp.tv64;
15
16   /* careful here, we're doing s32 = u64 - u64, but the resulting size
         should be fine */
17   hwtimestamp_delta = (hwtimestamp - hwtimestamp_old);
18   systimestamp_delta = ktime_sub(systimestamp, systimestamp_old);
19
20   /* calculate new clock speed
21    * if SystemSpeed negative, we are too fast
22    * if SystemSpeed positive, we are too slow
23    */
24   SystemSpeed = (systimestamp_delta.tv64 - hwtimestamp_delta);
25
26   hwtimestamp_old = hwtimestamp;
27   systimestamp_old = systimestamp;
28
```

```
29   hwtime_adjfreq(Frequency_adjustment);
30 }
```

Listing 5.4: Ausschnitt aus der Zeitsynchronisationsfunktion der beiden Zeitquellen

In den Zeilen 9 und 10 werden die aktuelle Hardwarezeit der NIC und die Linux-System-Zeit ausgelesen. Dies geschieht innerhalb des Kernel-Spinlocks spin_lock_irqsave in den Zeilen 7 und 11, da dieser Interrupts auf dem lokalen Prozessor deaktiviert (vgl. Corbet u. a., 2005, S.119) und somit sichergestellt werden kann dass die beiden Zeitstempel möglichst zeitnah ausgelesen werden. Auch spielt hierbei die Reihenfolge beim Auslesen eine entscheidende Rolle. Zuerst wird die Zeit von der ungenaueren Uhr gelesen. In Zeile 13 wird der Hardwarezeitstempel in Unix-Zeit konvertiert um sie später vergleichen zu können. In den Zeilen 17 und 18 wird das Delta der Zeitstempel der beiden Uhren gebildet. Würde die genauere Uhr zuerst ausgelesen werden, so würde sich der Jitter der ungenaueren Uhr nur auf eines dieser Deltas addieren und somit zu Ungenauigkeiten führen. In Abbildung 5.2 ist das Auslesen der Hardware-Zeit zuerst dargestellt und in Abbildung 5.3 der Software-Zeit zuerst. Dargestellt ist jeweils der maximal mögliche Jitter der Software-Uhr. $\Delta t1$ ist in Abbildung 5.2 die Zeit zwischen dem Auslesen der Hardware-Zeit, $\Delta t2$ zwischen der Software-Zeit und in Abbildung 5.3 umgekehrt. $\Delta z1$ ist die Zeit zwischen dem Auslösen der Software- und der Hardware-Zeitmessung, $\Delta z2$ entsprechend anders herum. Im Fall von $\Delta z2$ wirkt sich wie dargestellt der maximale Jitter der Software-Uhr auf $\Delta t2$ aus, nicht aber auf $\Delta t1$ und führt somit zu einem Gesamtjitter, wenn die Differenz zwischen $\Delta t1$ und $\Delta t2$ gebildet wird.

In Zeile 24 wird nun die Differenz der beiden ermittelten Zeitspannen gebildet um zu bestimmen, ob die Hardware-Uhr zu schnell oder zu langsam läuft. Da dieser Wert in Relation zu dem zuvor ermittelten Wert für das *TIMINCA* Register, der nötig ist um eine Geschwindigkeitssteigerung um 1% zu erreichen, um ein Vielfaches kleiner ist, kann dieser direkt für die Geschwindigkeitsanpassung verwendet werden. Diese findet über die Methode hwtime_adjfreq, welche in der Schnittstelle zu den Timerfunktionen der Netzwerkkarte implementiert wurde und im vorhergehenden Abschnitt bereits erklärt wurde, statt.

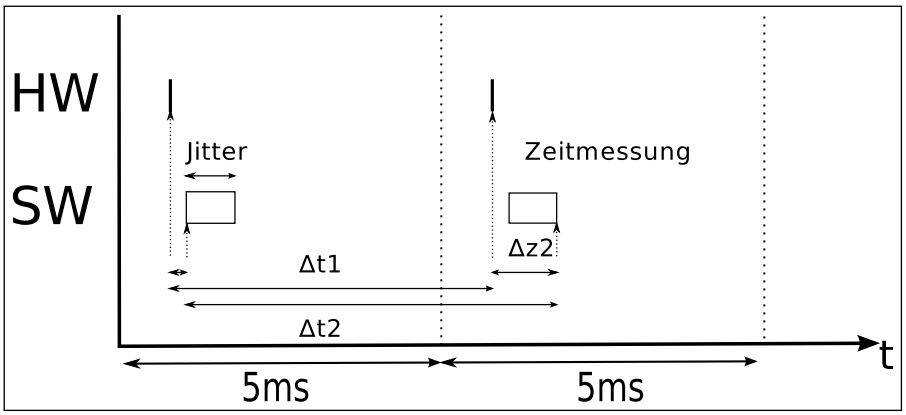

Abbildung 5.2: Auslesen der HW-Zeit zuerst

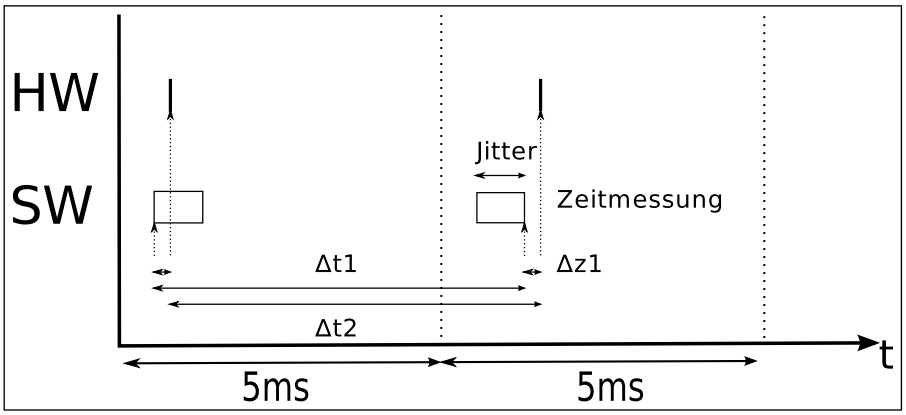

Abbildung 5.3: Auslesen der SW-Zeit zuerst

Ebenfalls implementiert wurde eine Sättigung des Anpassungswertes, falls dieser durch Messfehler zu sehr aus dem Rahmen fallen würde und so eine zu große Änderung vornehmen würde. Zuletzt wird der High-Resolution-Timer mit der voreingestellten Zykluszeit neu gestartet. Beides wird hier aus Platzgründen nicht aufgeführt.

5.2.3 Synchronisation mit dem Netzwerk

Für die TTEthernet-Synchronisation mit dem Netzwerk wurde der Synchronization Client nach AS6802 umgesetzt. Dieser sieht drei Zustände vor, *INTEGRATE*, *SYNC* und *STABLE*, deren detaillierte Beschreibung in Abschnitt 4.4.2 zu finden ist. Der Synchronization Client ist dabei

ein Prozess, der beim Empfang eines PCFrames ausgeführt wird und den *Offset* sowie die *Clock_Correction* gemäß Spezifikation berechnet.

Auf Grund der Implementierung des Schedulers findet die Zeitanpassung über das Verändern der Zyklus-Dauer statt.

```
1  if (ucSchedulerCurrentEvent == 0) {
2      spin_lock_irq(&period_start_lock);
3      /* critical section ... */
4      if(!time_expired && (local_integration_cycle ==
           pcf_integration_cycle)){
5        period += clock_corr;
6      } else {
7        offset = 0;
8      }
9      period_start = ktime_add_ns(period_start, period + offset);
10     spin_unlock_irq(&period_start_lock);
```

Listing 5.5: Anpassung der Periodendauer

Listing 5.5 zeigt den Ausschnitt aus der Implementierung des Schedulers, in dem die Anpassung realisiert wird. Ist kein weiteres Event im Buffer des Schedulers (vgl. Zeile 1), also am Ende des Zyklus, so wird der Startzeitpunkt für den neuen Zyklus berechnet. Im unsynchronisierten Zustand wird zunächst in *Integrate* der Offset zur Periode des Synchronization Masters in Zeile 9 korrigiert und in den Zustand *SYNC* gewechselt. Befindet sich der empfangene PCFrame innerhalb des Acceptance-Windows (Zeile 4 *time_expired*) und stimmt die Zyklusnummer des Synchronization Masters mit der lokalen Zyklusnummer überein (Zeile 4 *local_integration_cycle*), so wird die durch den Zustandsautomaten berechnete Zeitanpassung in Zeile 5 auf die Periodendauer addiert.

6 Ergebnisanalyse und Bewertung

In diesem Kapitel werden die Ergebnisse der Messungen vorgestellt und erörtert. Dazu wird zunächst der Versuchsaufbau, sowie die verwendete Hardware vorgestellt und auf die jeweiligen Konfigurationen eingegangen. Um die Genauigkeit der Uhrensynchronisation der Netzwerkkarte und der Linux-Systemzeit zu ermitteln, wird in einem definierten Zeitraum von 5ms die Differenz der auf der Netzwerkkarten-Uhr vergangenen Zeit und der vergangenen Linux-Systemzeit gebildet.

Ebenso wird die Veränderung der Periodendauer, sowie die Differenz des permanence_pit und des scheduled_receive_pit mit und ohne aktiviertem Synchronization Client gemessen. Dies dient zur Verifizierung der korrekten Funktionsweise des Synchronization Client, dessen Genauigkeit und Stabilität.

6.1 Versuchsaufbau

Betrachtet wird zunächst der Versuchsaufbau. Dieser ist in Abbildung 6.1 dargestellt. Für alle Messungen wurde der Microcontroller NXHX500-ETM direkt per Patchkabel mit der Intel i350-T2 Netzwerkkarte in einem Linux-System verbunden. Zum Einsatz kommt der Linuxkernel 3.6.8 mit Realtime-Patch, sowie mit aktivierter Precision Time Protocol Unterstützung. Die verwendete Linux-Distribution ist Linux Mint 14 'Nadia' in der 32Bit Variante.

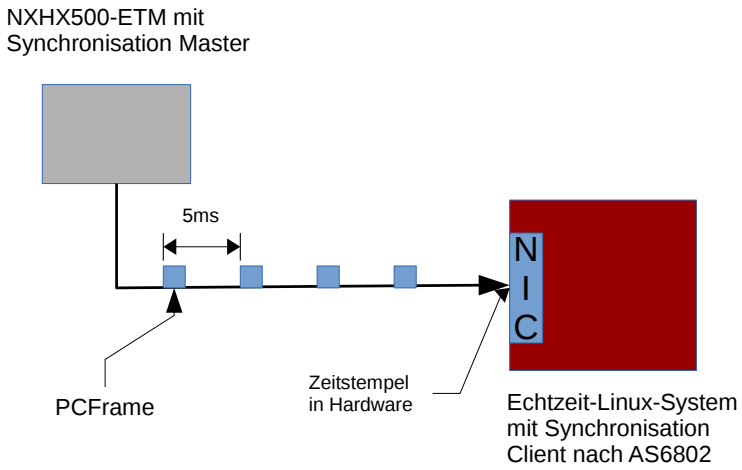

Abbildung 6.1: Versuchsaufbau

6.1.1 Konfiguration des TTEthernet-Netzwerks

Die Konfiguration des Synchronization Clients wurde wie folgt vorgenommen:

- Periodendauer = 5ms
- max_integration_cycles = UINT32_MAX
- precision_in_ns = 50000
- sync_to_stable = 3
- pcf_transparent_clock = 4112ns
- max_transmission_delay = 10000

Der Microcontroller NXHX500-ETM ist als Synchronization Master konfiguriert und sendet im Abstand von 5ms PCFrames auf Grund derer die Zeitanpassung im Client geschehen soll.

Die Konfigurationsdatei config.c

Die Konfigurationsdatei config.c dient dazu, die Buffer für TT- und RC-Nachrichten, sowie PCFrames anzulegen. Hier werden auch die Acceptance-Windows für die PCFrames eingestellt. Darüber hinaus dient sie dazu, die Versandzeitpunkte des zeitkritischen Datenverkehrs im Scheduler festzulegen.

Zeile 3 in Listing 6.1 legt fest, dass es sich um einen Synchronization Client handelt und Zeile 4 die Periodendauer in ns. Um Buffer für eingehende bzw. ausgehende Nachrichten anzulegen,

müssen diese in rx_message_conf_buffer[] bzw. tx_message_conf_buffer[] eingetragen werden. Im angegeben Beispiel werden sowohl ein Queuebuffer für TT-Nachrichten(Zeile 13-20) als auch ein Double-Buffer für PCFrames (Zeile 22-28) angelegt. Diese Einträge müssen dem Scheduler durch die Einträge in tte_conf_buffer in den Zeilen 6 und 8 bekannt gemacht werden.

Double Buffer: Bei einer Leseanfrage an einen Double Buffer liefert dieser immer das aktuellste Element, entfernt dieses jedoch nicht. Dadurch kann es sein, dass das selbe Element mehrfach gelesen wird bis es überschrieben wird.

Queue Buffer: Ein Queue Buffer ist ein FIFO. Er liefert ebenfalls immer das aktuellste Element, entfernt es jedoch auch aus dem Buffer. Jedes Element kann demnach nur einmal gelesen werden.

```
1  tte_swes_conf_t tte_conf_buffer =
2    { 0x3040506 /* cluster_id */
3    , 0 /* sync_client */
4    , (5000000) /* period */
5                 ⋮
6    , &(rx_message_conf_buffer[0]) /* rx_table */
7    , &(schedule_entry_buffer[0]) /* schedule_table */
8    , &(rx_message_conf_buffer[1]) /* pcf_conf */
9                 ⋮
10   };
11
12 tte_message_conf_t rx_message_conf_buffer[] =
13   {
14     { 100 /* msg_id = VL_TT_100 */
15     , TT_MAC(100) /* dst_mac */
16   , 0xE789 /* eth_type */
17   , 1514 /* size (byte) */
18   , TTE_QUEUE_BUF /* buf_type */
19   , 32 /* buf_len */
20     } /* [0] */
21   ,
22     { 0xfcb /* msg_id = */
23     , TT_MAC(0xfcb) /* dst_mac */
24     , TT_SYNC_PROTO /* eth_type */
25     , 60 /* size (byte) */
26     , TTE_DOUBLE_BUF /* buf_type */
27     , 2 /* buf_len */
28     } /* [1] */
29   }; /* end rx_message_conf_buffer */
```

```
30
31 tte_schedule_entry_t schedule_entry_buffer[] =
32   {
33     { (0) /* time */
34     , TTE_SCHED_SYNC,
35       { (tte_message_conf_t *) &(rx_message_conf_buffer[1]) } /* tt_msg
           */
36     , 0 /* deadline */
37     } /* [0] */
38 }; /* end schedule_entry_buffer */
```

Listing 6.1: config.c

Wann welche Ereignisse im Scheduler auftreten bzw. erwartet werden wird in schedule_entry_buffer[] eingetragen. In der in Listing 6.1 aufgeführten Konfiguration wird zu Beginn des Zyklus (Zeile 33, Zeit 0) ein Sync-Event, also ein PCFrame, (Zeile 34) erwartet. Zeile 35 gibt an, welcher Buffer verwendet werden soll und Zeile 36 besagt, dass das Event keine Deadline hat, der Frame als zu jeder Zeit ankommen darf.

6.2 Messergebnisse

Im Folgenden werden die Messergebnisse dargestellt und erörtert. Zunächst wird auf die interne Uhrensynchronisation der Hardware-Uhr der NIC und der Linux-Systemzeit eingegangen und danach auf die Ergebnisse des Synchronization Clients.

Alle nachfolgenden Messungen wurden dabei über die Ausgabe geeigneter Werte über die Konsole mit 'printk' ohne graphische Oberfläche durchgeführt.

6.2.1 Synchronisation der Hardware-Uhr der NIC mit der Linux-Systemzeit

Es wurde ein periodisch laufender Thread implementiert, der die aktuelle Zeit sowohl von der Hardware-Uhr, als auch von der Software_Uhr abliest und das jeweilige Delta zu der vorhergehenden Messung bildet. Auf Grund der Unterschiede dieser Deltas wird die Geschwindigkeit der Netzwerkkarten-Uhr angepasst. Ist die Differenz dieser Deltas 0, so laufen die Uhren synchron. Gemessen wird die Differenz besagter Deltas um die Genauigkeit der Synchronisation zu ermitteln. Die Ergebnisse sind in 6.2 dargestellt.

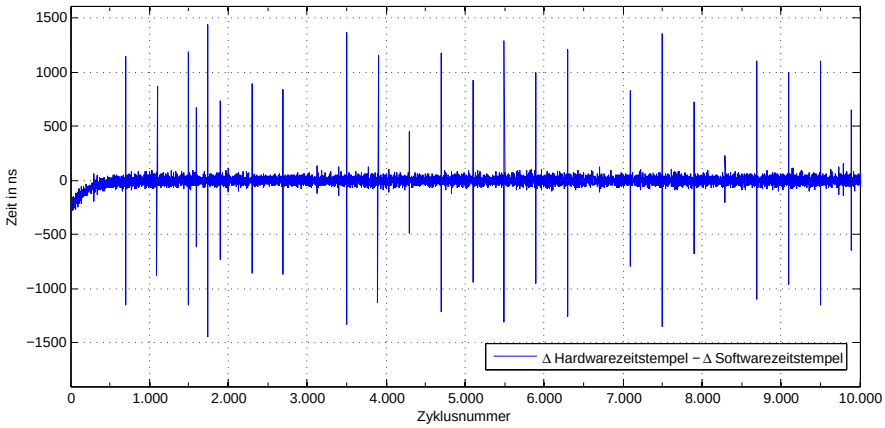

Abbildung 6.2: Genauigkeit der Uhrensynchronisation

Zu sehen ist die Einsynchronisationszeit von ca. 500 Zyklen. Das entspricht in etwa einer Zeit von $500 \cdot 5ms = 2{,}5s$. Diese Zeit bezieht sich auf den Ladezeitpunkt des Moduls. Danach befindet sich der Großteil der Abweichungen in einem Bereich von $+/-200ns$. Der maximale Jitter beträgt etwa $3\mu s$. Die nur relativ selten auftretenden Ausreißer lassen sich eher schwer durch eine konkrete Ursache erklären. Sie weisen eine gewisse Periodizität auf, was vermuten lässt, dass ein wiederkehrendes Ereignis, beispielsweise ein Interrupt, im Betriebssystem dafür verantwortlich ist. Eine genaue Analyse würde jedoch den Rahmen dieser Arbeit sprengen.

6.2.2 Synchronization Client

Im Folgenden sollen nun die Messergebnisse für den Synchronization Client betrachtet werden. Abbildung 6.3 zeigt hier die Differenz des *permanence_pit* und des *smc_scheduled_receive_pit* ohne aktiven Synchronization Client. Bei einem voll synchronen System wäre diese Differenz nahe dem Nullpunkt.

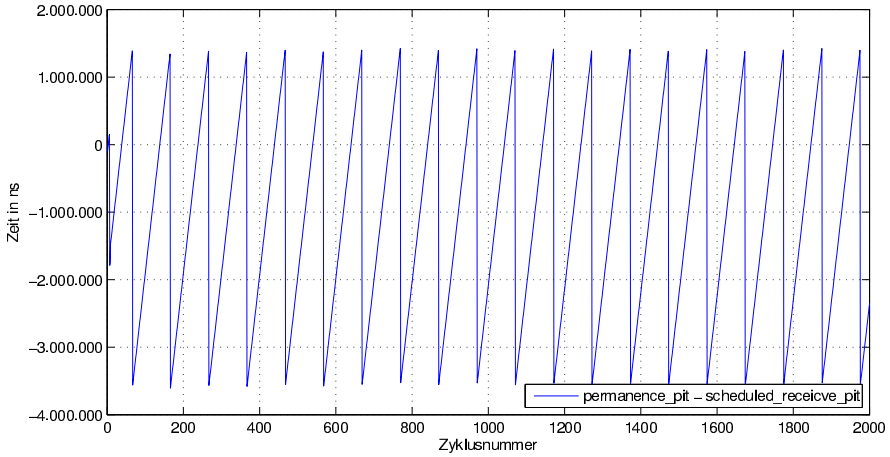

Abbildung 6.3: Differenz von Permanence Pit und Scheduled Receive Pit ohne Synchronization Client

In der Abbildung ist zu sehen dass die Differenz den gesamten Bereich des konfigurierten Zeitzyklus von 5ms einnimmt (von -3,5ms bis 1,5ms). Die Uhr des Synchronization Masters läuft demnach mit einer anderen Geschwindigkeit als das zu synchronisierende System. Etwa alle 100 Zyklen wiederholen sich die Messwerte, also alle $100 \cdot 5ms = 0,5s$. Im Umkehrschluss bedeutet dies, dass die Abweichung der Uhren um $5ms \div 100 = 50\mu s$ pro Zyklus zunimmt.

Schlüsselt man die Differenz des *permanence_pit* und des *smc_scheduled_receive_pit* auf, so lässt sich auch erklären ob die Uhr des Client-Systems zu schnell oder zu langsam läuft.

$$permanence\_pit = hw\_tstamp + max\_transmission\_delay - TransparentClock \tag{6.1}$$

$$smc\_scheduled\_receive\_pit = local\_period\_start + max\_transmission\_delay \tag{6.2}$$

Aus den beiden Gleichungen ergibt sich folglich:

$$\begin{aligned} &permanence\_pit - smc\_scheduled\_receive\_pit \\ &\qquad = hw\_tstamp - TransparentClock - local\_period\_start \end{aligned} \tag{6.3}$$

Wie bereits beim Versuchsaufbau erläutert, ist der TransparentClock-Wert statisch konfiguriert und vernachlässigbar klein. Dies reduziert die Gleichung weiterhin zu:

$$permanence\_pit - smc\_scheduled\_receive\_pit \approx hw\_tstamp - local\_period\_start \tag{6.4}$$

Dies sagt aus, dass die lokale Uhr des zu synchronisierenden Systems schneller läuft als die des SMs. Ebenfalls ist der Graph innerhalb einer Periode steigend, was den selben Schluss zulässt. Zur Erklärung ist noch zu sagen, dass *local_period_start* hier die lokale Uhr des Clients repräsentiert und *hw_tstamp* die Uhr des Synchronization Master dargestellt in lokaler Zeit des Clients, da dieser die Empfangszeitpunkte der PCFrames vornimmt. Auffällig ist der negative Offset von ca. 1ms. Im Rahmen dieser Arbeit konnte dieser jedoch aus zeitlichen Gründen nicht näher analysiert werden. Eine naheliegende Vermutung ist allerdings, dass er mit dem Ausführungszeitpunkt im Scheduler zusammenhängt, da dieser ebenfalls bei 1ms liegt. Bestärkt wird dies dadurch, dass der negative Jitter sowohl bei aktivem, als auch bei ausgeschaltetem Synchronization Client größer als der positive ist, wie die Abbildung 6.3 und Abbildung 6.6 zeigen.

In Abbildung 6.4 ist die gleiche Messung wie oben dargestellt, jedoch mit aktiviertem Synchronization Client. Hier ist zu sehen, dass nicht wie zuvor der gesamte Bereich des Zyklus verwendet wird und nach einer Einschwingzeit von ca. 80 Zyklen, also etwa 0,4s die Differenz des *permanence_pit* und des *smc_scheduled_receive_pit* nahezu bei Null liegt.

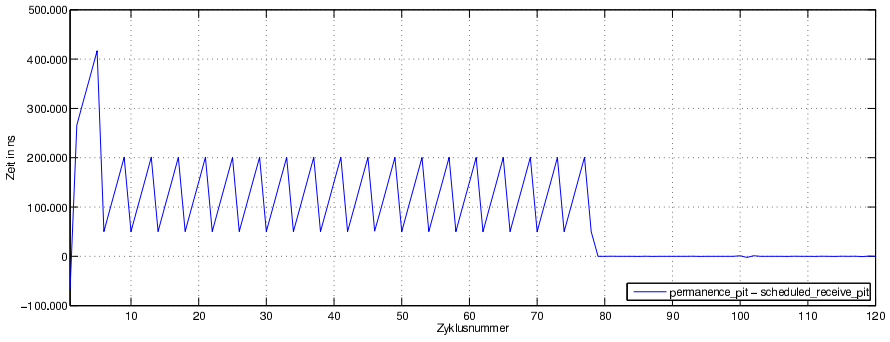

Abbildung 6.4: Einschwingzeit

Auffallend ist hier der gleichbleibende Verlauf im Bereich der Zyklusnummer 6 bis 78. Zu erwarten wäre eine stete Annäherung an die Nulllinie bis zur maximal möglichen Genauigkeit der Zeitanpassung. In dem genannten Bereich ist dies jedoch nicht der Fall. Hier zeigt sich ein

periodisch gleichbleibender Wertebereich. Dieses Verhalten ist auf eine Eigenheit in der Implementierung und der verwendeten Konfiguration des TTEthernet-Netzwerks zurückzuführen. Zur Erinnerung: Die Precision beträgt $50\mu s$, das Acceptance Window demnach $100\mu s$. Der Synchronization Client ist um $50\mu s$ zu schnell.

Zu beginn ist der Synchronization Client im Zustand „Integrate“. Hier wird der Startzeitpunkt des Zyklus korrigiert, also der Offset, und danach in den Zustand „Sync“ gewechselt. Ist das nun empfange PCFrame innerhalb des Acceptance Windows, wo wird die Periodenlänge anhand dieses Frames angepasst. Sind drei Frames außerhalb, so wird wieder in den Zustand „Integrate“ gewechselt. Das Ziel der Synchronisation ist es, einen möglichst kleinen Offset zwischen dem Empfangszeitpunkt des PCF und des scheduled_receive_point zu erreichen. Aufgrund der Implementierung des Schedulers wirkt sich jedoch eine Zeitanpassung immer erst auf den Zyklus nach dem aktuellen aus.

Im Integrate-Zustand wird im ersten Schritt der Offset angepasst, aber da der Synchronization Client um $50\mu s$ zu schnell läuft ist im nächsten Schritt der Offset genau um diese Wert zu groß und das Acceptance Window wird knapp verfehlt. Dabei addieret sich weiter der Offset von $50\mu s$ pro Zyklus auf und die Synchronisation driftet davon. Nach drei Fehlversuchen im Sync-Zustand wird erst wieder in Integrate gewechselt und das ganze beginnt von neuem. Dies geschieht so lang, bis durch den Jitter einmal das Acceptance Window getroffen wird. Danach funktioniert die Synchronisation wie erwartet. Abbildung 6.5 soll diesen Vorgang verdeutlichen.

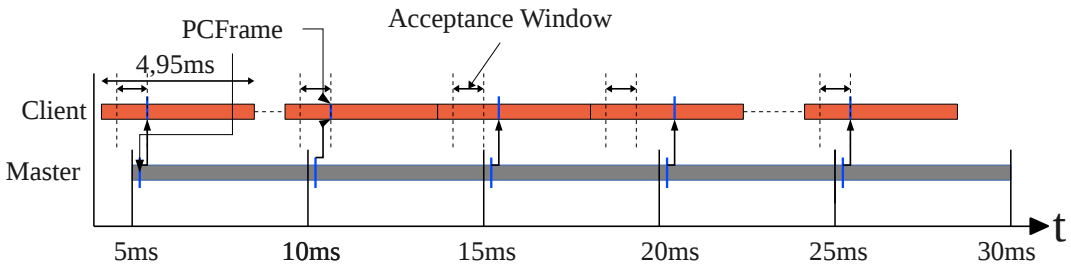

Abbildung 6.5: Erklärung der Synchronisationseigenart

Die x-Achse stellt die real vergangene Zeit dar. Sowohl beim Master als auch beim Client ist eine Periodendauer von $5ms$ eingestellt. Da aber die Uhr des Clients um ca. $50\mu s$ zu schnell läuft, ist die Periode in realer Zeit kürzer als die des Masters.

Eine schnellere und vor allem zuverlässigere Synchronisation ließe sich erreichen, indem entweder das Acceptance Window erhöht wird, oder bereits in Integrate nach der ersten Offsetkorrektur auch eine Anpassung der Zyklusdauer gemacht wird.

Abbildung 6.6 stellt eine Vergrößerung des Bereiches nach den 80 Zyklen Einschwingzeit aus Abbildung 6.4 dar. Die Extremwerte liegen hier bei $-2{,}5\mu s$ und $+1{,}4\mu s$. Dies ergibt einen Jitter von rund $4\mu s$.

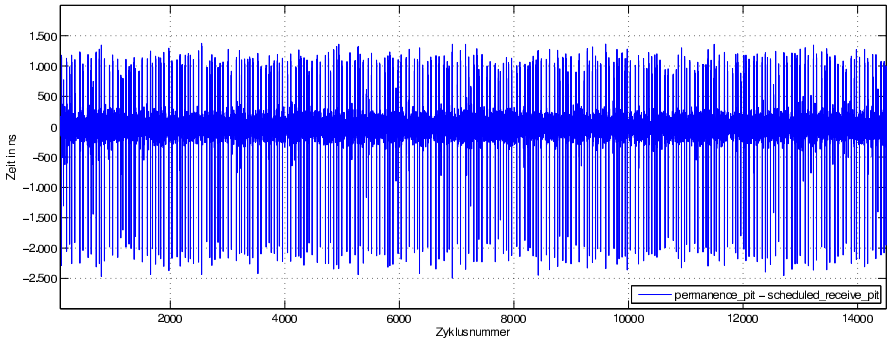

Abbildung 6.6: Vergrößerte Differenz von Permanence Pit und Scheduled Receive Pit mit Synchronization Client

Wie im Konzept bereits festgestellt, hängt die Genauigkeit des Synchronization Client direkt von der Genauigkeit der internen Uhrensychronisation ab, da die Hardwarezeitstempel mit der Linux-Software-Uhr verglichen werden müssen. Dies spiegeln diese beiden Graphen wie erwartet wider.

Betrachtet man nun den Verlauf der Periodenlänge in Abbildung 6.7, so bestätigt sich dich Annahme, dass das zu synchronisierende System um $50\mu s$ zu schnell läuft.

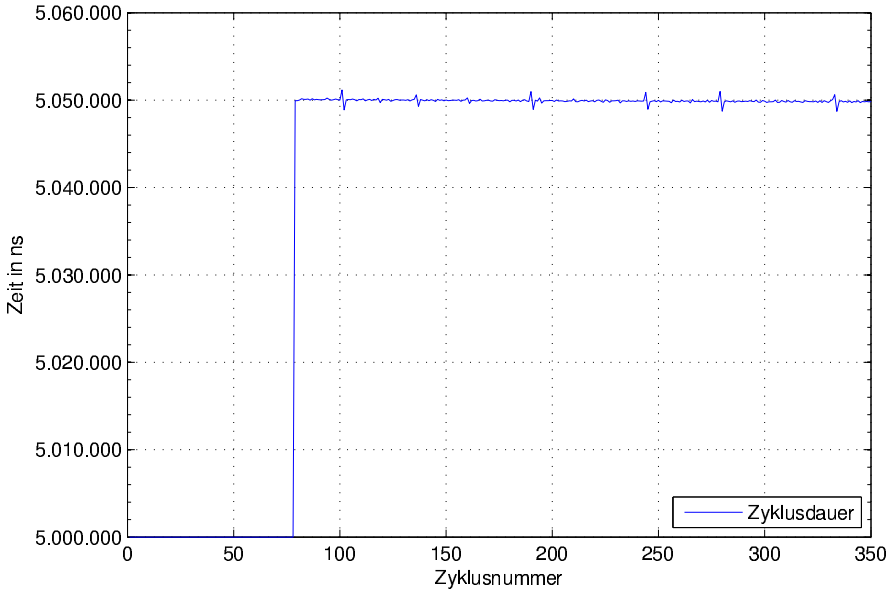

Abbildung 6.7: Periodendauer mit aktivem Synchronization Client

Die ersten 80 Zyklen findet keine Anpassung der Periode statt. Dies erhärtet die Erklärung der vorhergehenden Graphen. Erst als das Acceptance Window durch den Jitter getroffen wird erhöht sich die Zykluslänge um exakt den erwarteten Wert von $50\mu s$ um sie dem Master anzugleichen.

Abbildung 6.8 zeigt den Verlauf der Periodendauer nach den ersten 80 Zyklen. Es lässt sich ein Jitter von $2{,}5\mu s$ ablesen. Dargestellt ist ein Bereich von über 14000 Zyklen, was einer Zeit von mehr als 70 Sekunden entspricht. In dieser Zeit bewegt sich das Synchronisationsergebnis stabil innerhalb der erwähnten Grenzen. Auch längere Messungen (hier aus Gründen der Übersichtlichkeit nicht dargestellt) haben ein ähnliches Bild ergeben.

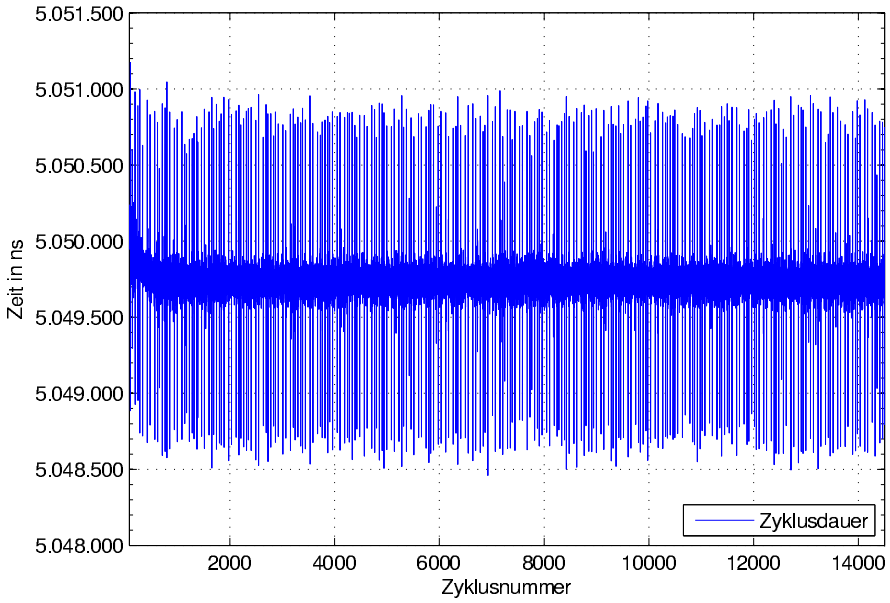

Abbildung 6.8: Bereich mit aktivem Synchronization Client

Die Periodendauer pendelt sich bei ca. 5,049,750ns ein. Durch den verlängerten Zyklus im Client werden die unterschiedlich schnell laufenden Uhren angeglichen. In realer Zeit sind somit die Zyklen des Clients und des Masters gleich groß.

Insgesamt konnte durch die Messungen die Funktion sowohl der internen Uhrensynchronisation sowie die Synchronisation mit dem TTEthernet-Netzwerk nachgewiesen werden.

7 Zusammenfassung, Fazit und Ausblick

Das Ziel dieser Arbeit war die Verbesserung der Zeitpräzision eines Linux-TTEthernet-Treibers mit Hilfe eines Netzwerkcontrollers, der in der Lage ist den Empfangszeitpunkt exakt in Hardware zu bestimmen. Darüber hinaus wurde der Synchronization Client nach AS6802 implementiert.

Die in Kapitel 6, Ergebnisanalyse und Bewertung ermittelten Messwerte bestätigen eine erfolgreiche Modifizierung sowohl des TTEthernet-Treibers als auch des Treibers der Netzwerkkarte und der Implementierung des Synchronization Clients.

Zusammenfassung der Ergebnisse

Die Aufgabenstellung umfasste drei Teilgebiete. Diese sollen im folgenden Verlauf aufgelistet und die Ergebnisse noch einmal zusammengefasst werden.

Anbindung des Netzwerkkartentreibers an den TTEthernet-Treiber: Der Treiber der Intel Netzwerkkarte konnte erfolgreich mit dem TTEthernet-Treiber verbunden werden. Es gab die Möglichkeit, die Methoden für die Zeitanpassung im Intel Treiber nach außen hin für andere Module mit EXPORT_SYMBOL sichtbar zu machen. Dieser Weg wurde jedoch nicht gewählt, da im Falle eines Updates alle Änderungen erneut durchgeführt werden müssten. Stattdessen wurden die Methoden angepasst und im TTEthernet-Treiber neu implementiert. Es werden dabei das Auslesen und das Schreiben des Timer-Registers, sowie die Anpassung der Geschwindigkeit der Uhr unterstützt. Des weiteren wurden Methoden zur Aktivierung des Hardware-Zeitstempels und der Konvertierung in Unix-Zeit implementiert.

interne Synchronisation der Uhr der NIC und der Linux-Softwarezeit: Da die Intel Netzwerkkarte keine Timer mit Interruptgenerierung unterstützt, war es nötig auf die High-Resolution-Software-Timer des Kernels zurückzugreifen. Um die Zeitstempel der

Netzwerkkarte mit der Linux-Zeit vergleichen zu können müssen die Uhren der Netzwerkkarte und die Software-Zeit synchronisiert werden. Dies wurde mit Hilfe einer Rate-Correction realisiert. Dabei konnte eine hohe Genauigkeit erzielt werden. Der maximale Jitter lag bei $3\mu s$. Erwähnenswert ist, dass die Abweichung größtenteils kleiner als $200ns$ war.

Synchronization Client nach AS6802: Aufbauend auf die Ergebnisse der vorhergehenden Teilgebiete wurde der Synchronization Client nach AS6802 implementiert. Dabei wurde eine Eigenart festgestellt. Auf Grund der Umsetzung des TTEthernet-Schedulers wirken sich die Änderungen der Periodendauer erst auf den nächsten Zyklus aus. Dies hat zur Folge, dass unter Umständen das Acceptance Window nach dem Integrationszyklus nicht getroffen wird, bzw die Synchronisation bei zu kleinem Acceptance Window nie funktioniert. Dieses Verhalten kann korrigiert werden, indem entweder das Window größer konfiguriert wird, oder beim ersten Durchlauf nach dem Integrationszyklus trotz Verfehlen des Acceptance Windows eine Zeitanpassung gemacht wird.

Mit der Synchronisation konnte eine ausreichende Genauigkeit erzielt werden. Das Gros der Abweichungen des *permanence_pits* und des *scheduled_receive_pits* lag zwischen $-500ns$ und $+500ns$. Insgesamt ergab sich ein maximaler Jitter von $4\mu s$.

Fazit

Das Ziel der Verbesserung der Zeitgenauigkeit des Linux-TTEthernet-Treibers wurde erreicht. Für die Zeitsynchronisation mit einem TTEthernet-Netzwerk konnte ein Jitter von $4\mu s$ erzielt werden. Darüber hinaus konnte das ursprüngliche Konzept, möglichst wenig Anpassungen am Treiber der verwendeten Netzwerkkarte vorzunehmen, eingehalten werden.

Ausblick

Die ermittelten Messwerte und deren Auswertung legen eine erfolgreiche Erweiterung des TTEthernet-Treibers nahe und sind eine vielversprechende Grundlage für weitere Projekte.

Steer-by-Wire-Anwendung: Stephan Phieler entwickelte im Rahmen seiner Bachelorarbeit unter anderem eine Steer-by-Wire-Anwendung mittels Force-Feedback-Lenkrad. Dieses Lenkrad wird mittels serieller Schnittstelle über einen Mikrocontroller am TTEthernet-

Netzwerk angeschlossen. Die Anwendung könnte für den TTEthernet-Treiber angepasst werden und so als Beispiel zur Verifikation der Treiberfunktionen dienen.

AS6802 Synchronisationsprotokoll: Ein offensichtlich weiterführendes Projekt ist das Implementieren der noch fehlenden Rollen des Synchronisationsprotokolls. Durch die Umsetzung sowohl des Synchronization Masters als auch des Compression Masters wäre der Funktionsumfang des TTEthernet-Treibers vollständig und es somit möglich einen Computer mit Standard-Hardware als vollwertigen TTEthernet-Teilnehmer einzusetzen.

Umstellung auf das Precision Time Protocol: Soll ein Laptop verwendet werden, z.B. als leicht portables Diagnosewerkzeug für ein TTEthernet-Netzwerk, ergibt sich ein Problem: Die Netzwerkcontroller, die in der Lage sind den exakten Empfangszeitpunkt aller eingehenden Nachrichten in Hardware zu ermitteln und eine Basis für diese Arbeit bilden, gibt es nur als PCI-Express-Karten.

Ein weiterer Ansatz für die Zeitsynchronisation von Ethernet-Teilnehmern ist die Verwendung des Precision Time Protocol nach IEEE 1588. Ein White Paper der Firma TTTech legt nahe, dass es möglich ist PTP innerhalb eines TTEthernet-Netzwerks zu verwenden, da die PTP-Frames kompatibel zu PC-Frames sind (vgl. TTTech Computertechnik AG, 2010, S.10). Einem weiteren White Paper der Firma Hirschmann ist zu entnehmen, dass durch eine solche Synchronisation eine Genauigkeit von +/-60ns „und das praktisch unabhängig von der Netzlast oder der Auslastung der CPU“ (Dreher und Mohl, S.8) erreicht werden könnte, was noch einmal eine deutliche Steigerung der Präzision bedeuten würde. Dies ist jedoch auch nur durch die Verwendung der Hardwareunterstützung für PTPv2 möglich, was aber bereits in vielen Netzwerkkarten integriert ist und auch in Laptops zu finden ist. Damit könnten eingehende UND ausgehende Synchronisationspakete mit präzisen Zeitwerten versehen werden.

Ein Umsetzungsmöglichkeit könnte überprüft und je nach Sinnhaftigkeit gegebenenfalls eine Implementierung durchgeführt werden.

Literaturverzeichnis

[Abbot 2006] Abbot, Doug: *Linux for embedded and real-time applications - 2nd Edition.* Butterworth Heinemann, Mai 2006. – ISBN 0-7506-7932-8

[Bartols 2010] Bartols, Florian: *Leistungsmessung von Time-Triggered Ethernet Komponenten unter harten Echtzeitbedingungen mithilfe modifizierter Linux-Treiber.* Hamburg, HAW Hamburg, Bachelorthesis, Juli 2010. – Bachelorthesis

[Benz und CO. 1886] Benz und CO.: *Fahrzeug mit Gasmotorenbetrieb.* 1886

[Bovet und Cesati 2008] Bovet, D.P. ; Cesati, M.: *Understanding the Linux Kernel.* O'Reilly Media, 2008. – URL http://books.google.de/books?id=h0lltXyJ8aIC. – ISBN 9780596554910

[Corbet u. a. 2005] Corbet, Jonathan ; Rubini, Alessandro ; Kroah-Hartman, Greg: *Linux Device Drivers, Third Edition.* O'Reilly Media, Inc., 2005. – ISBN 978-0-596-00590-0

[Dreher und Mohl] Dreher, Andreas ; Mohl, Dirk: Präzise Uhrzeitsynchronisation - Der Standard IEEE 1588 / Hirschmann Automation and Control GmbH. URL http://www.pdv.reutlingen-university.de/rte/White_paper_ieee1588_de_v1-2.pdf. – Forschungsbericht. Zugriffszeitpunkt: 17.10.2013

[Intel Corporation 2013a] Intel Corporation: *Intel Ethernet Controller i210 Datasheet.* Revision Number: 2.4. http://www.intel.com: , 2013. – URL http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/i210-ethernet-controller-datasheet.pdf

[Intel Corporation 2013b] Intel Corporation: *Intel Ethernet Controller I350 Datasheet.* Revision Number: 2.1. http://www.intel.com: , March 2013. – URL http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-controller-i350-datasheet.pdf

[Love 2010] Love, Robert: *Linux kernel development*. Third. Addison-Wesley, 2010 (Developer's library: essential references for programming professionals). – 440 S. – ISBN 0-672-32946-8

[Marscholik und Subke 2011] Marscholik, Christoph ; Subke, Peter: *Datenkommunikation im Automobil: Grundlagen, Bussysteme, Protokolle und Anwendungen*. 2. Auflage. Vde-Verlag, Februar 2011. – ISBN 3800732750

[Quade 2011] Quade, Jürgen: *Linux-Treiber entwickeln : eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung*. Heidelberg : dpunkt-Verl, 2011. – URL https://ezs.kr.hsnr.de/TreiberBuch. – ISBN 9783898646963

[Rick 2012] Rick, Frieder: *Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux*. Hamburg, HAW Hamburg, Bachelorthesis, Juni 2012. – Bachelorthesis

[SAE Aerospace AS6802 2011] SAE Aerospace AS6802: *Time-Triggered Ethernet AS6802*. SAE Aerospace. November 2011. – URL http://standards.sae.org/as6802/

[Sally 2010] Sally, Gene: Real Time. In: *Pro Linux Embedded Systems*. Apress, 2010, S. 257–271. – URL http://dx.doi.org/10.1007/978-1-4302-7226-7_12. – ISBN 978-1-4302-7227-4

[The Linux Kernel Organization, Inc. 2013a] The Linux Kernel Organization, Inc.: *The Linux Kernel Archives*. Oktober 2013. – URL https://www.kernel.org/. – Zugriffszeitpunkt: 17.10.2013

[The Linux Kernel Organization, Inc. 2013b] The Linux Kernel Organization, Inc.: *The Linux Kernel Archives*. Oktober 2013. – URL https://www.kernel.org/doc/Documentation/ptp/ptp.txt. – Zugriffszeitpunkt: 15.10.2013

[The Linux PTP Project 2013] The Linux PTP Project: *The Linux PTP Project*. Oktober 2013. – URL http://linuxptp.sourceforge.net. – Zugriffszeitpunkt: 10.10.2013

[Ts'o u. a. 2013] Ts'o, Theodore ; Hart, Darren ; Kacur, John: *The Real-Time Linux Wiki*. Oktober 2013. – URL https://rt.wiki.kernel.org/. – Zugriffszeitpunkt: 17.10.2013

[TTTech Computertechnik AG] TTTech Computertechnik AG: *TTEthernet in Motion*. – URL http://www.tttech.com/fileadmin/content/general/secure/TTEthernet/TTTech-TTEthernet_in_Motion-Whitepaper.pdf. – Zugriffszeitpunkt: 30.09.2013

[TTTech Computertechnik AG 2010] TTTech Computertechnik AG: Scalable Real-Time Ethernet Platform. URL http://www.tttech.com/fileadmin/content/general/secure/TTEthernet/TTTech-TTEthernet-Scalable_Real-Time_Ethernet_Platform-Whitepaper.pdf, 2010. – Forschungsbericht. Zugriffszeitpunkt: 17.10.2013

[Zimmermann und Schmidgall 2010] Zimmermann, W. ; Schmidgall, R.: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. Vieweg+Teubner Verlag, 2010 (ATZ/MTZ-Fachbuch). – URL http://books.google.de/books?id=muK7cQAACAAJ. – ISBN 9783834809070

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 14. November 2013 Johannes Reidl