Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Frieder Rick

Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Frieder Rick

Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang 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

Abgegeben am 15. Juni 2012

Frieder Rick

Thema der Bachelorarbeit

Entwurf und Entwicklung eines virtuellen TTEthernet Treibers für Linux

Stichworte

TTEthernet, Linux, RT Patch, Real-Time, Time-Triggered, Echtzeit, Ethernet, Echtzeitbetriebssystem, RTOS, Treiber, Zeitkritisch

Kurzzusammenfassung

In dieser Arbeit wird ein Entwurf und die Umsetzung eines TTEthernet Controllers unter Linux mit dem RT Patch beschrieben. Der zu entwickelnde Treiber soll zum einen als virtuelles Netzwerkinterface für die Verarbeitung von Best Effort (standard Ethernet) Traffic auf Socket-Ebene funktionieren. Zum anderen soll über die TTEthernet API von TTTech der Zugriff auf den TTEthernet Controller und das Senden und Empfangen von Time-Triggered Nachrichten realisiert werden. Dabei soll der zeitkritische Traffic nicht von dem unkritischen Best Effort Traffic des virtuellen Netzwerkinterfaces gestört werden.

Frieder Rick

Title of the paper

Design and developement of a virtual TTEthernet driver for Linux

Keywords

TTEthernet, Linux, RT Patch, Real-Time, Time-Triggered, Ethernet, Realtime operating system, RTOS, Driver, Timecritical

Abstract

This paper describes the design and realisation of a TTEthernet controller under Linux with the RT Patch. The driver which should be developed, has to implement a virtual network interface for the handling of best effort traffic (standart Ethernet) with the use of sockets. The TTEthernet controller should be accesible through the TTE API from TTTech for the use with time triggered messages. The critical traffic should not be disturbed by the best effort traffic from the virtual network interface.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Seit 1970 steigt die Anzahl der elektronischen Bauteile im Automobil stetig an. Vorher mechanische oder hydraulische Bauteile wurden mehr und mehr durch elektronische Bauteile ersetzt. Heutzutage ist die Elektronik aus einem modernen Fahrzeug nicht mehr wegzudenken. Zusätzlich zu den ersetzten mechanischen Bauteilen kommen immer neue Erweiterungen zur Erhöhung der Sicherheit wie z. B. das Elektronisches Stabilitätsprogramm (ESP). Weitere Anwendungen wie z. B. eine Rückfahrkamera, Abstandssensoren und Unterhaltungsmedien für die Mitfahrer kommen noch dazu.

Sensoren müssen mit Steuergeräten kommunizieren, welche die Daten wiederum weiterverarbeiten, um entsprechend zu reagieren oder die Informationen für den Fahrer verfügbar zu machen. Dazu wurden für die verschiedenen Anwendungsbereiche jeweils passende Bussysteme für die Kommunikation entwickelt. Die derzeit am meisten verwendeten Bussysteme sind SAE J1850, CAN, LIN, MOST, IDB-1394 und FlexRay. Da die Steuergeräte oft auch auf Informationen von Sensoren angewiesen sind, die auf einem anderen physikalischen Bus liegen, wurden seit 1990 Mechanismen entwickelt, um die Kommunikation verschiedener Bussysteme untereinander über Gateways zu ermöglichen.

Ein Phaeton der Volkswagen AG verfügt über mehr als 60 Steuergeräte, welche über 3 unterschiedlichen Bussystemen kommunizieren und so über mehr als 3800 Meter Kabel bestehend aus 2100 Einzelleitungen 2500 Signale übertragen (Vgl. Marscholik und Subke, 2011), um das Auto unter möglichst optimalen Bedingungen zu steuern und nötige Informationen für den Fahrer aufzubereiten (Geschwindigkeitsinformationen, Staumeldungen, Wetterinformationen usw.). Wenn in Zukunft Bauteile, die als sehr sicherheitskritisch eingestuft werden (Bremse, Lenkstange), auch durch elektronische Bauteile ersetzt werden sollen, wird das zu einem weiteren Zuwachs an Steuergeräten und Leitungen führen.

Durch die steigende Anzahl der elektronischen Bauteile im Automobil in den letzten Jahren werden immer höhere Datenraten auf den Bussen benötigt. Aus diesem Grund werden für verschiedene Anwendungsfelder verschiedene Busse eingesetzt (siehe Tabelle 1). Durch die vielen verschiedenen Bussysteme (siehe Abbildung 1.1) in heutigen Automobilen ist die Komplexität des Systems und die Anzahl der nötigen Einzelleitungen enorm gestiegen und wirkt sich somit direkt auf die Entwicklungs- und Materialkosten aus. Um hier eine neue übersichtlichere und kostengünstigere Lösung zu finden, wird bereits seit einiger Zeit an der Möglichkeit geforscht, Ethernet als Bus im Automobil einzusetzen.

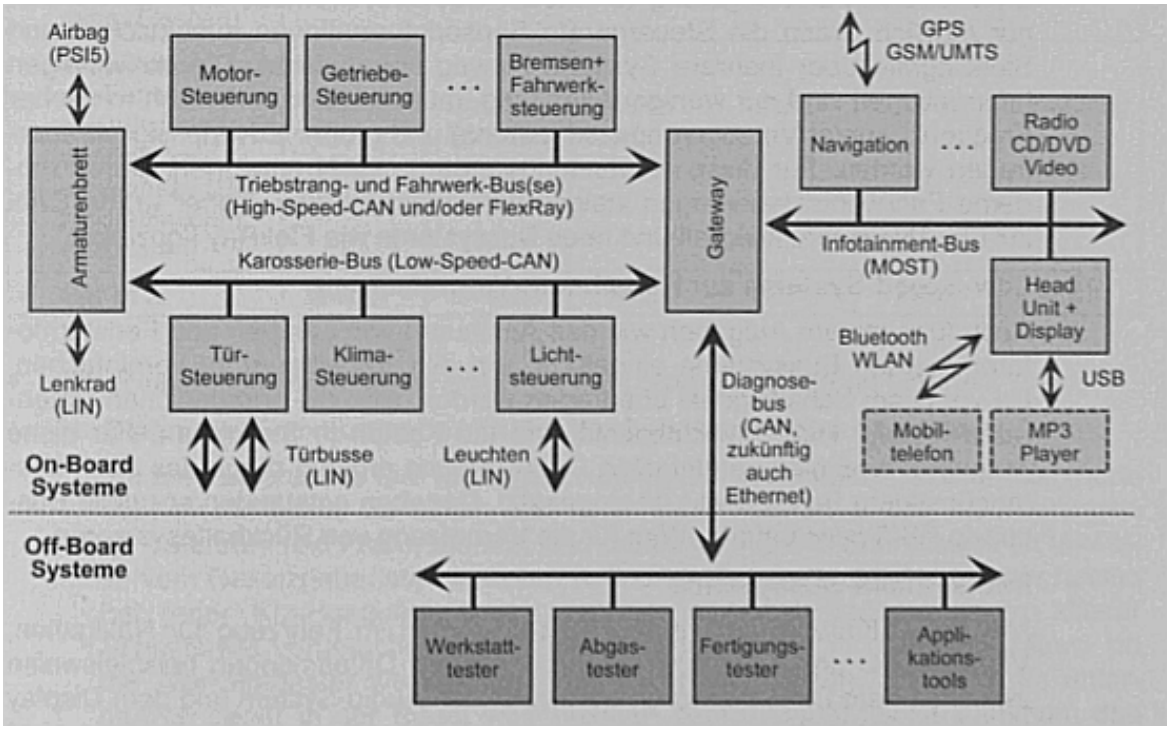

Abbildung 1.1: Bussysteme eines modernen Fahrzeugs (Quelle: Zimmermann und Schmidgall, 2011)

Ethernet ist durch seine bereits heute enorme Verbreitung in den verschiedensten Bereichen eine Lösung, die sich positiv auf die Kosten auswirken wird. Durch seine hohen Bandbreiten und die Unabhängigkeit des physikalischen Layers kann es mit einer Echtzeit-Erweiterung eine optimale Lösung als Backbone und Ersatz für derzeit genutzte Bussysteme im Automobil aber auch der Automatisierungstechnik darstellen.

Zielsetzung

Es soll ein TTEthernet (Time-triggered Ethernet) Stack als Treiber für Linux entwickelt werden, der mit minimalen Änderungen am Treiber der Netzwerkkarte auskommt und mit allen gängigen Netzwerkkarten zusammenarbeitet. TTEthernet ist eine Erweiterung des IEEE (Institute of Electrical and Electronics Engineers) 802.3 Standards (Vgl. Institute of Electrical

Class	Bitrate	Typische Vertreter	Anwendung
Diagnose	< 10 Kbit/s	ISO 9141-K-Line	Werkstatt- und Abgabetester
A	< 25 Kbit/s	LIN, SAE J1587/1707	Karosserieelektronik
B	25 . . . 125 Kbit/s	CAN (Low Speed)	
C	125 . . . 1000 Kbit/s	CAN (High Speed)	Antriebsstrang, Fahrwerk, Diagnose
D	> 1 Mbit/s	FlexRay, TTP	X by Wire, Backbone-Netz
Infotainment	> 10 Mbit/s	MOST	Multimedia (Audio, Video)

Tabelle 1.1: Klassifikation der Bussysteme (Vgl. Zimmermann und Schmidgall, 2011)

and Electronics Engineers, 2005) welcher das Protokoll um harte Echtzeiteigenschaften erweitert und trotzdem konform zum 802.3 Standard bleibt. Es wurde erstmals unter dem Namen TTE an der Uni Wien entwickelt (Vgl. Kopetz u. a., 2005) und später in Zusammenarbeit mit der Firma TTTech unter dem Namen TTEthernet weiterentwickelt und spezifiziert (Vgl. Steiner, 2008). Mittlerweile ist es als SAE AS6802 Spezifikation (Vgl. SAE, 2011) standardisiert worden. Der Stack soll die API von TTTech implementieren und mit dem Ethernet Stack von Linux zusammenarbeiten um Programme die auf Sockets basieren weiterhin zu unterstützen. Der Treiber soll zwischen den Ethernet Stack und den Netzwerkkartentreiber geschoben werden, um die Kontrolle über ankommende und zu sendende Frames zu erhalten. Über die API von TTTech soll der Linux Stack umgangen und kritische Pakete direkt an den virtuellen Treiber übergeben werden, welcher somit eine Priorisierung dieser vornehmen kann und gewährleisten soll, dass die Pakete mit Priorität und Echtzeiteigenschaften in ihrem vorgegebenen Zeitfenster versendet bzw. empfangen werden. Das Modul soll die für TTEthernet spezifizierte Synchronisierung implementieren, um Nachrichten rechtzeitig im Schedule versenden und empfangen zu können. In dieser Arbeit wird zum Testen der Anforderungen nur der Synchronisations-Client implementiert.

Gliederung der Arbeit

In Kapitel 2 wird auf grundlegende Themen eingegangen, welche für die Realisierung der Arbeit relevant sind. Kapitel 3 stellt das Design, des zu entwickelnden Treibers, dar und in Kapitel 4 wird auf die Realisierung dieses und die konkrete Implementierung des Treibers eingegangen. Zum Schluss werden in Kapitel 5 Tests vorgestellt, Ergebnisse erläutert und ein Ausblick für die zukünftige Entwicklung des Treibers gegeben.

2 Grundlagen

In diesem Kapitel wird auf das verwendete Betriebssystem und Protokoll eingegangen sowie auf die für diese Arbeit wichtigen Eigenschaften dieser. Alle Aussagen, welche über den Linux Kernel getroffen werden, beziehen sich auf die Version 3.0.29 mit dem RT Patch in Version 49.

2.1 Echtzeit und Linux

Echtzeit für ein Betriebssystem ist die Einhaltung vordefinierter Fristen, wobei weiterhin die Ergebnisse der Berechnungen korrekt sein müssen. (Vgl. Wörn, 2005)

2.1.1 Echtzeitbetriebssystem

Ein Echtzeitbetriebssystem, unter anderem auch RTOS (Real-Time Operating System) genannt, unterscheidet sich von einem Betriebssystem ohne Echtzeitunterstützung nicht wie oft angenommen in der Geschwindigkeit, sondern die Pünktlichkeit in der Ausführung von Aufgaben ist ausschlaggebend. Dabei ist Geschwindigkeit nicht zu vernachlässigen, da die Ressourcen ausreichen müssen, um die Aufgabe in einem vordefiniertem Zeitfenster zu erledigen. Pünktlichkeit setzt voraus, dass eine Aufgabe nicht vor einem definierten Zeitpunkt erledigt ist und bis zu einem weiteren vorgegebenen Zeitpunkt abgeschlossen sein muss (Rechtzeitigkeit).

Des weiteren wird noch mal zwischen harter und weicher Echtzeit für Aufgaben unterschieden, da die Anforderungen an die Rechtzeitigkeit je nach Anwendungsfall unterschiedlich ausfallen. Harte Echtzeit setzt voraus, dass das Zeitfenster in jedem Fall getroffen wird, da eine Verletzung zu fatalen Fehlern im System führen kann. Als Beispiel kann hier die Auslösung eines Airbags im Automobil genommen werden. Falls bei einem Aufprall eines Autos der Airbag nur einige tausendstel Sekunden zu spät oder zu früh auslöst kann das fatale Folgen für die Insassen haben. Weiche Echtzeit wünscht sich, dass das Zeitfenster eingehalten wird wobei eine Verletzung der Ausführungsfrist nur unschöne Effekte auslöst und keine fatalen Folgen nach sich zieht. Das ist der Fall z. B. bei Abspielen von Musik, wo eine

Verletzung der Ausführungsfrist zu unschönen Verzerrungen in der Wiedergabe führt, der Hörer dabei aber nicht verletzt wird (Vgl. Wörn, 2005, S. 322).

Um in einem RTOS Fristen bei hoher Last einzuhalten ist es nötig, dass die einzelnen Aufgaben priorisiert werden können und unterbrechbar sind. Der Scheduler des Betriebssystems ist dafür zuständig die einzelnen Aufgaben zu verwalten und einen niedriger priorisierten Prozess gegebenenfalls zu verdrängen, wenn ein höher priorisierter die CPU benötigt. Realtime Tasks (RT Tasks) müssen geplant werden und ihre Ausführungszeit muss vorhersagbar sein. Dabei muss sichergestellt werden, dass für deren Ausführung zu jeder Zeit genug Rechenleistung zur Verfügung steht. Wenn die RT Tasks[1] nicht die volle Rechenleistung benötigen, kann die freie CPU-Zeit dynamisch an nicht priorisierte Tasks vergeben werden deren Ausführung gewünscht ist, aber der Zeitpunkt der Ausführung keine Rolle spielt.

2.1.2 Real-Time Linux

Real-Time Linux (RT Linux) ist eine Erweiterung von Linux um Echtzeiteigenschaften. Lange Zeit gab es RT Linux ausschließlich als Patch für die Linux Kernel Sourcen, mittlerweile hat es größtenteils den Einzug in den Mainline[2] Kernel geschafft, sodass es in Zukunft ein fester Bestandteil von Linux wird (Vgl. Gleixner, 2011). Der RT Patch erweitert Linux um voll präemptives Scheduling. Um dies zu ermöglichen, mussten einige Konzepte des Linux Kernels überdacht und den Ansprüchen entsprechend angepasst werden.

Im Linux Kernel ohne RT Patch (im folgenden Linux) haben Interrupt Service Routinen (ISR's), Softirq's und Tasklets die höchste Priorität und jeder Prozess, der gerade auf der CPU läuft, wird von ihnen bei Ausführung verdrängt. Kernelthreads ebenso wie Benutzerprozesse. Dadurch werden ISRs, Softirqs und Tasklets automatisch zu den höchst priorisierten Prozessen im System. Der RT Patch transformiert diese in Kernel Threads um sie ebenfalls unterbrechbar zu machen und es dem Scheduler zu ermöglichen diese anhand definierter Prioritäten gleichrangig anderen Prozessen gegenüber zu behandeln (Vgl. S. 388 f. Yaghmour u. a., 2008).

Hardwareinterrupts werden von externen Ereignissen ausgelöst und stoßen die für das Ausführen des Handlers zuständige ISR an. Bei Linux unterbricht ein Hardware Interrupt die CPU und gibt sie erst wieder frei, wenn die ISR abgearbeitet ist. Das kann zu Interrupt Inversionen führen, wobei eine ISR einen hoch priorisierten Prozess an der Ausführung hindert. Der RT Patch kann nicht verhindern, dass ein Hardwareinterrupt die CPU unterbricht. Er kann die Unterbrechung aber minimieren, indem die mit der Interrupt Request Queue (IRQ) verbundene ISR einen Kernel Thread aktiviert, welcher dann dem Scheduler unterliegt und

[1] Task = Aufgabe
[2] Hauptentwicklungszweig des Kernels

die eigentliche Funktion der ISR später abarbeitet (siehe Abbildung 2.1). Die einzige Ausnahme ist der System Timer, welcher seine ISR weiterhin direkt aufruft, und somit nicht unterbrechbar ist.

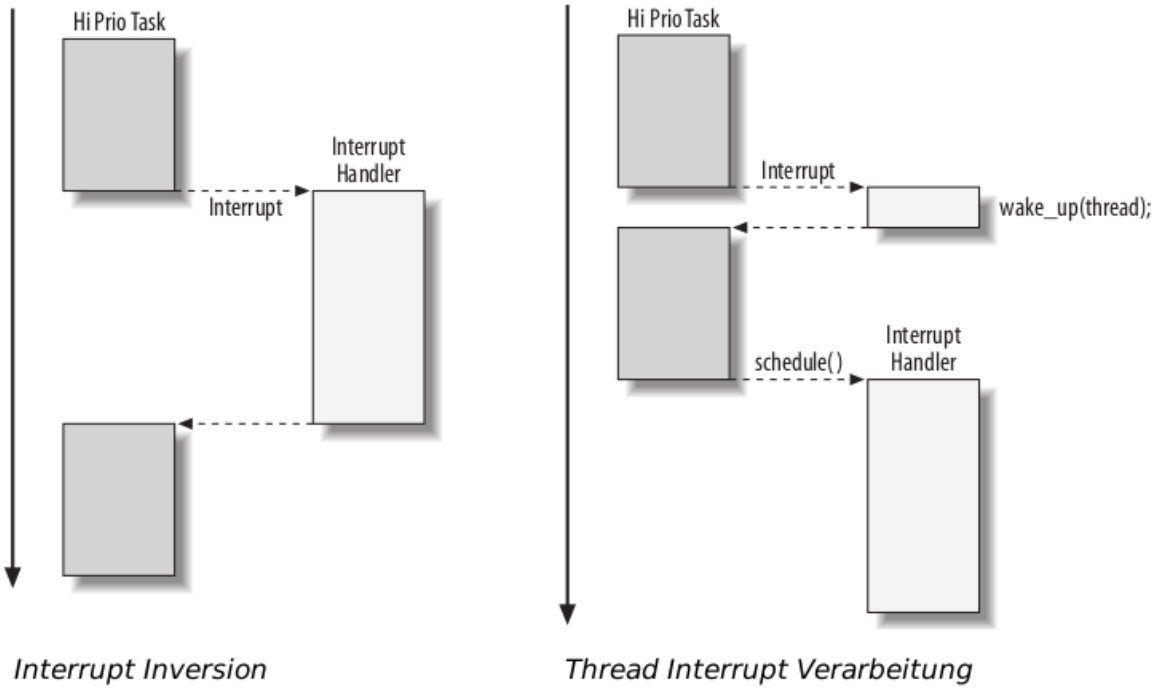

Abbildung 2.1: Interrupt Inversion und RT Linux (Quelle: Yaghmour u. a., 2008)

Da Software Interrupts (Softirqs, ausführlich in Kapitel 2.6) unter Linux nur von ISRs unterbrochen werden können und sie ansonsten alle Tasks einschließlich Kernel Threads verdrängen, kann dies zu Prioritätsinversion führen. Aus diesem Grund werden auch Softirqs durch den RT-Patch in einem Kernel Thread pro CPU namens ksoftirqd/n[3] abgearbeitet. Eine Zeit lang gab es im RT Patch für die einzelnen Interrupt-Typen, jeweils einen Thread auf jeder CPU. Dies hatte unter anderem zum Vorteil dass durch die Priorisierung des softirq-net-tx und softirq-net-rx Softirq Threads der Netzwerkverkehr durch eine hohe Priorität bevorzugt behandelt werden konnte. Doch von diesem Verfahren wurde wieder zu einem ksoftirqd Thread pro CPU zurückgegangen, da unter anderem auch ungewollte Verzögerungen entstanden sind und um näher am Mainline Kernel zu sein (Vgl. Corbet, 2011). Somit unterliegen nun alle Softirqs der gleichen RT Priorität.

Linux mit dem RT-Patch kann zwar als RTOS welches harte Echtzeit unterstützt eingestuft werden, da es in einer getesteten Umgebung deterministisches Verhalten nachweisen kann.

[3] n ist die Nummer des zugehörigen CPUs

Es kann aber nicht den Ansprüchen an ein hoch sicherheitskritisches System gerecht werden, da ein deterministisches Verhalten nicht mathematisch bewiesen ist. Bugs im System können auch eine gut getestete Umgebung in unvorhersehbare Zustände bringen. Aktuelle Informationen über den Patch sind auf der Webseite (Ts'o u. a., 2012) von Real-time Linux zu finden.

2.1.3 Andere RTOS Linux Lösungen

Da diese Arbeit auf den RT Patch für Linux aufbaut, werden hier nur kurz andere Echtzeitbetriebssysteme vorgestellt, um die Designunterschiede zu diesem hervor zu heben.

RTAI-Linux (Vgl. Buttazzo, 2011, S. 433) erweitert den Linux Kernel mit dem RTAI-Patch und fügt diesem so ein Realtime Application Interface (RTAI) hinzu. Durch diese Erweiterung wird zwischen die CPU und den Linux Kernel ein Realtime-Kernel geschoben, welcher sich um die Interrupt Verarbeitung kümmert und das Echtzeitscheduling übernimmt. Der Linux Kernel läuft unter dem RT Kernel als Echtzeit-Task mit der niedrigsten Priorität im System. Realtime Tasks unterbrechen den Linux Kernel und können über vordefinierte FIFOs oder Shared Memory mit dem Linux Userspace kommunizieren. Über das LXRT Modul wird außerdem Zugriff auf die RTAI und RT Scheduler Schnittstellen aus dem Userspace ermöglicht. Ein großer Vorteil von RTAI Linux ist, dass durch den RTAI Patch die volle Kontrolle über die Interrupts im RT Kernel liegt. Durch die Trennung entsteht leider auch der Nachteil, dass eine normale Linux Anwendung nicht ohne Anpassung und die RTAI Schnittstellen als Realtime Task ausführen werden kann. Aktuelle Informationen sind auf der Webseite (RTAI Team, 2012) von RTAI zu finden.

Xenomai (Vgl. Buttazzo, 2011, S. 434) ist vor einiger Zeit aus RTAI entstanden. Das Grundkonzept ist ähnlich da es den Linux Kernel auch um einen RT Kernel, in Xenomai Nucleus genannt, erweitert welcher für das RT Scheduling zuständig ist und den Linux Kernel vollständig unterbrechen kann. Zusätzlich wird Adaptive Domain Environment for Operating Systems (Adeos) eingesetzt, welches zwischen der Hardware und den beiden Kerneln als Interrupt Pipeline arbeitet. Die Interrupts werden in Domänen aufgeteilt und Interrupts, für die sich der Nucleus registriert hat, werden nur wenn er sie explizit weiterreicht an den Linux Kernel übergeben. Interrupts, die in der Echtzeit-Domäne keinen Handler haben, werden von der Pipeline direkt an den Linux Kernel weitergereicht. RT Tasks können für Userspace und Kernelspace programmiert werden. Im Userspace nutzt man die Speicherschutz-Vorteile von Linux, kann dessen Funktionen verwenden und es kann über sogenannte Skins mit dem Nucleus kommuniziert werden. Skins sind Schnittstellen, welche Realtime-APIs verschiedener Echtzeitbetriebssysteme emulieren. So können RT Applikationen von anderen Echtzeitbetriebssystemen, welche ein Skin in Xenomai implementiert haben ohne aufwendiges Redesign portiert werden. Die Skins wiederum greifen auf einen Satz generischer

Echtzeit-Dienste und -Funktionen zu welche die Nucleus API darstellen. RT Treiber werden im Kernelspace implementiert, um einen direkten Zugriff auf die Hardware zu ermöglichen. Für RT Treiber gibt es ein RTMD-Skin (Real-time Driver Model), über welchen die Treiber mit dem Nucleus kommunizieren. Im Gegensatz zu RTAI ist es bei Xenomai nun wieder möglich auch normale Linux Anwendungen im Userspace unter weichen Echtzeitbedingungen laufen zu lassen. Aktuelle Informationen sind auf der Webseite (Xenomai, 2012) von Xenomai zu finden.

2.2 Scheduling unter Linux

Ein Multitasking Betriebssystem kann mehrere Prozesse gleichzeitig, mit einer CPU, ineinander verschachtelt ausführen. Bei nur einem Prozessor entsteht dadurch die Illusion von gleichzeitig laufenden Prozessen. Bei Multiprozessoren laufen Prozesse gleichzeitig auf mehreren CPUs jeweils verschachtelt. Prozesse haben des weiteren die Möglichkeit einen blockierten (Blocked in Abbildung 2.2) Zustand anzunehmen, solange sie auf die Verarbeitung von Daten warten. Dabei wird zwischen kooperativem und präemptivem Multitasking unterschieden. Bei kooperativem Multitasking ist es den einzelnen Prozessen überlassen in gewissen Abständen die CPU für andere wartende Prozesse freizugeben. Präemptives Multitasking gibt dem Prozess Scheduler die Möglichkeit Prozessen die CPU zu entziehen und sie an einen anderen wartenden Prozess abzugeben. Linux implementiert präemptives Multitasking wie die meisten modernen Betriebssysteme heutzutage.

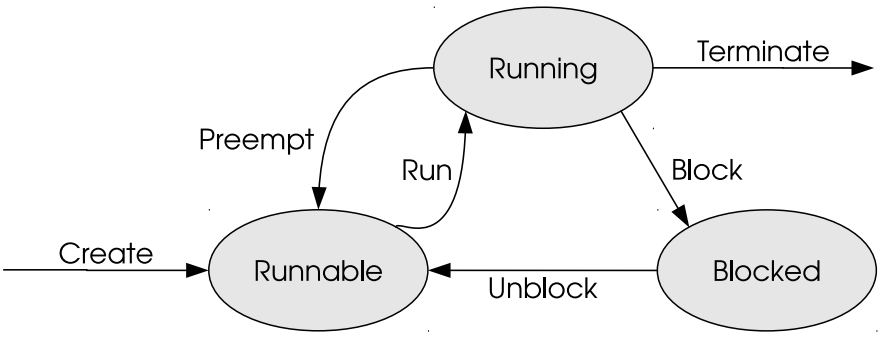

Abbildung 2.2: Vereinfachte Darstellung der Prozesszustände unter Linux

Der Prozess Scheduler, im folgenden Scheduler, eines Betriebssystems ist für die Verwaltung der zu erledigenden Prozesse zuständig. Er entscheidet welcher Prozess wann und wie viel Rechenzeit zugewiesen bekommt. Er teilt die begrenzt verfügbare Rechenzeit der CPU unter den aktiven Prozessen im System auf. Bei einem Multitasking-Betriebssystem wie Linux ist der Scheduler der Kern des Systems. Da ein CPU immer nur einen Prozess zur Zeit bearbeiten kann ist es die Aufgabe des Schedulers Rechenzeit an die laufenden und

wartenden (Running und Runnable in Abbildung 2.2) Prozesse so zu verteilen das es dem Anwender so vorkommt als ob alle Programme auf seinem Computer ihre Arbeit gleichzeitig erledigen.

In den frühen Kernelversionen von Linux wurde eine Scheduling-Strategie angewendet, welche zur Bestimmung des Folgeprozesses immer alle wartenden Prozesse mit einbezogen hat. Durch moderne Computer, auf welchen immer mehr Prozesse gleichzeitig ausgeführt wurden, kam es dabei zu immer größeren Verzögerungen bei der Auswahl je mehr Prozesse ausgeführt wurden. Um dieses Problem zu lösen, wurde der sogenannte O(1) (Big O) Scheduler entwickelt, welcher eine konstante Zeit für die Wahl des Folgeprozesses benötigt. Für Server mit einer hohen Anzahl an CPUs und Prozessen hat diese Strategie gut funktioniert. Da aber heutzutage Linux auch viel im Desktop-Bereich eingesetzt wird, ist diese Scheduling-Strategie für diesen Zweck auch ungeeignet. Ein Desktop Betriebssystem interagiert viel mit dem Nutzer und dieser möchte das seine Eingaben schnell angenommen und verarbeitet werden. Bei dem O(1) Scheduling wurden Prozesse die auf Nutzerinteraktion warten, aber oft nur wenig Zeit benötigen, gleichbehandelt wie rechenintensive Hintergrundprozesse, was die Bedienbarkeit für den Benutzer einschränkt. Um die Antwortzeiten bei Benutzereingaben zu optimieren, wurde das completely fair scheduling (CFS, komplett Faires Scheduling) (Vgl. Love, 2010, S. 46) entwickelt.

Prozesse lassen sich anhand ihres Verhaltens grob in zwei Gruppen aufteilen. Die erste Gruppe sind auf Eingaben wartende (I/O) Prozesse, welche blockieren, solange sie auf ein Ereignis wie z. B. eine Benutzereingabe warten. Diese Prozesse brauchen in der Regel nur wenig CPU-Zeit. Es ist aber gewünscht, dass sie schnell reagieren, damit der Benutzer eine schnelle Rückmeldung über seine Eingabe erhält. Die zweite Gruppe ist CPU gebunden und führt die meiste Zeit Berechnungen aus welche mehr Zeit in Anspruch nehmen aber keine hohen Anforderungen an die Ausführungsgeschwindigkeit haben. Diese Prozesse müssen unterbrochen werden, um eine faire Einteilung der CPU zu gewährleisten.

Unter Linux lassen sich Prozessen zwei Arten von Prioritäten (Vgl. Love, 2010, S. 44) vergeben. Nice-Werte, welche von 19 bis -20 reichen, wobei 20 die niedrigste Priorität ist, da dieser Prozess am nettesten ist und somit Anderen den Vortritt lässt. Des weiteren gibt es Real-time Prioritäten welche von 0 bis 99 reichen. Hier hat der höchste Wert auch die höchste Priorität. Ein Prozess mit einer Real-time Priorität ist immer höher priorisiert als ein Prozess ohne.

SCHED_NORMAL (CFS)

Der Scheduler vergibt Prozessen die Rechenzeit benötigen sogenannte Zeitscheiben die einen numerischen Wert darstellen, welcher angibt wie lange der Prozess die CPU benutzen darf, bevor er unterbrochen wird. Linux vergibt im CFS-Scheduling keine festen Zeitscheiben

an Prozesse, sondern vergibt Anteile der insgesamt verfügbaren Rechenzeit. Die Zeitscheibe welche ein Prozess zugewiesen bekommt, wird anhand einer Funktion berechnet, welche aufgerufen wird wenn ein Prozess Anspruch auf die CPU erhebt indem er in den Runnable-Zustand wechselt. Wenn der zugewiesene Anteil an CPU-Zeit des anfragenden Prozesses größer ist, als der verbleibende Anteil des aktuell laufenden Prozesses, wird der neue Prozess sofort ausgeführt. Anderenfalls wird er für die spätere Ausführung zurückgestellt. Die Funktion zur Berechnung des Anteils eines Prozesses nimmt dessen Nice-Wert als Gewichtung des ihm zustehenden Anteils. Bei n Prozessen, welche Anspruch auf die CPU erheben und die alle den gleichen Nice-Wert haben, ist der jeweilige Anteil die gesamt verfügbare Zeit T geteilt durch die Anzahl der Prozesse (T/n). Bei unterschiedlichen Nice-Werten erhält jeder Prozess seinen Anteil an CPU Zeit proportional zu seiner Gewichtung geteilt durch die Summe der Gewichtung aller laufenden Prozesse. Da die Anzahl der laufenden Prozesse gegen unendlich gehen, es aber nur endlich viel Rechenzeit gibt, wird um nicht durch zu viele Prozesswechsel, welche auch Rechenzeit benötigen, die CPU zu blockieren, eine Untergrenze für die Zeitscheiben festgelegt. Diese beträgt in der Standard Konfiguration 1ms (Vgl. Love, 2010, S. 49).

Es werden also vom Scheduler alle Prozesse nacheinander für die ihnen zugewiesene Zeitscheibe ausgeführt. Falls gerade kein Prozess die CPU benötigt, kann ein laufender Prozess auch länger als vorerst geplant die CPU benutzen. Wenn nun ein I/O-lastiger Prozess aufwacht und CPU-Zeit benötigt, wird geprüft, ob der ihm zustehende Anteil größer ist als der verbleibende Anteil des aktuell die CPU beanspruchenden Prozesses. Da I/O-lastige Prozesse meistens ihre zugeteilte CPU Zeit nicht voll benötigen und somit prozessorlastige Prozesse die Möglichkeit bekommen länger als vorerst geplant zu laufen hat ein I/O-lastiger Prozess gute Chancen anhand seiner besseren Zeitquote sofort zur Ausführung zu kommen. Somit ist diese scheduling-Strategie sehr gut um schnelle Antwortzeiten mit hoher Auslastung der Systemressourcen zu vereinen.

SCHED_FIFO und SCHED_RR (RT)

Diese Scheduling Strategien sind für das Real-Time Scheduling vorgesehen. Realtime Prozesse unterbrechen Prozesse, die mit SCHED_NORMAL scheduled sind, immer wenn sie in den Runnable Zustand wechseln. Einzelne RT Prozesse unterbrechen sich gegenseitig, wenn ein Prozess mit einer höheren Priorität in den Runnable Zustand wechselt. Die Scheduling-Strategien SCHED_FIFO und SCHED_RR kommen zum Einsatz, wenn mehrere RT Prozesse mit der gleichen Priorität gleichzeitig arbeiten möchten. Die SCHED_FIFO Strategie arbeitet die RT Prozesse mit gleicher Priorität in der Reihenfolge, in der sie zur Ausführung gebracht wurden ab. Ein RT Task darf so lange arbeiten, bis er die CPU abgibt und dann kommt der nächste Prozess an die Reihe. Bei der SCHED_RR Strategie werden

gleich priorisierte RT Tasks, welche gleichzeitig arbeiten möchten, abwechselnd ausgeführt bis sie die CPU für niedriger priorisierte Prozesse freigeben.

2.3 Ethernet

Ethernet wurde von Bob Metcalfe entwickelt und erstmals 1976 vorgestellt. Es basiert auf dem früher entwickelten Aloha Network, welches als Experiment an der Universität von Hawaii von Norman Abramson und Kollegen für die Kommunikation über Radiowellen entwickelt wurde (Vgl. Spurgeon, 2000).

IEEE veröffentlichte 1985 die erste Version des 802.3 Standards (Vgl. Institute of Electrical and Electronics Engineers, 2005) welcher Ethernet beschreibt und Protokoll sowie physikalische Spezifikationen enthält. Mittlerweile wurde der Standard durch etliche Erweiterungen verbessert. Heute wird hauptsächlich Switched-Ethernet eingesetzt, was kollisionsfrei arbeitet und dadurch die Kollisions-Erkennung überflüssig macht. Durch diesen internationalen Standard hat sich in den folgenden Jahren Ethernet auf dem Markt immer mehr durchgesetzt und sich bis heute gehalten.

Im Folgenden werden die Felder des Ethernet Frames (siehe Abbildung 2.3) beschrieben:

Präambel Ein Frame beginnt mit einer 8 Byte großen Präambel, wobei die ersten 7 Byte das Bitmuster "10101010“ enthalten. Das letzte Byte signalisiert den Framestart und enthält das Bitmuster "10101011“. Dadurch wird die Bit-Synchronisation zwischen Sender und Empfänger ermöglicht.

Zieladresse Die Zieladresse enthält die MAC-Adresse des Empfängers.

Quelladresse Hier trägt der Absender seine eigene MAC-Adresse ein.

Länge / Typ Dieses Feld enthält die Länge des Frames oder den Nachrichtentyp. Werte bis 1500 werden als die Länge des Frames interpretiert, ein größerer Wert steht für den Typ.

Daten Dieses Feld enthält die zu übertragenden Daten des Frames. Die minimale Größe beträgt 46 Byte, die maximale Größe beträgt 1500 Byte.

Prüfsumme Die CRC-Prüfsumme dient zur Erkennung von Übertragungsfehlern.

Abbildung 2.3: Aufbaus eines Ethernet Frames nach IEEE 802.3

2.4 Time-Triggered Ethernet

Time-Triggered Ethernet (TTEthernet) baut auf dem Ethernet Protokoll auf und erweitert dieses um Echtzeiteigenschaften ohne zu den Frames des original 802.3 Standards inkompatibel zu sein. Hier wird nun auf die besonderen Eigenschaften von TTEthernet eingegangen. Diese Übersicht veranschaulicht die grobe Funktionsweise der Erweiterung, ausführliche Informationen sind in der SAE AS6802 Spezifikation (SAE, 2011) zu finden.

Zusätzlich zu den Standard Ethernet Frames gibt es bei TTEthernet zwei weitere Nachrichtenklassen. Die im Protokoll verwendeten Nachrichtenklassen sind:

Time-Triggered Traffic (TT) TT-Pakete haben die höchste Priorität und werden für zeitkritischen Verkehr welcher auf ein deterministisches Zeitverhalten angewiesen ist verwendet. Sie können von anderen Nachrichten nicht verdrängt werden. Kurze Antwortzeiten und minimaler Jitter sind hier von hoher Bedeutung. Sie werden immer zu einem definierten Zeitpunkt im Zyklus gesendet.

Rate-Constrained Traffic (RC) RC-Paketen wird eine vorher definierte Bandbreite zugesichert. Sie können aber von TT-Paketen verdrängt werden. Diese Nachrichten sind zum ARINC-Standard 664 konform (Vgl. S. 11 SAE, 2011).

Best-Effort Traffic (BE) entspricht den Standard Ethernet Paketen. Diese Nachrichten nutzen die verbleibende Bandbreite im System. Eine Übertragung kann wie bei 802.3 nicht garantiert werden.

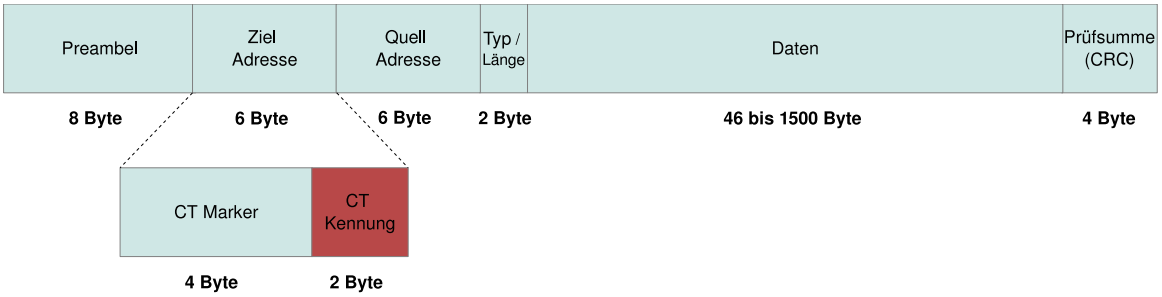

Abbildung 2.4: Aufbaus eines TTEthernet Frames für Critical Traffic (CT)

Da der Ethernet Standard im Header der Frames kein Feld für die Priorisierung von Paketen vorsieht, muss hier auf eine Lösung zurückgegriffen werden, welche mit dem Standard konform ist. In TTEthernet wird aus diesem Grund das Feld der Zieladresse so interpretiert das hier eine Priorisierung vorgenommen werden kann. Die Zieladresse wird dazu aus einem 4 Byte großen CT[4] Marker und einer 2 Byte großen CT Kennung (CT ID) zusammengesetzt (siehe Abbildung 2.4). Somit ist es möglich in einem System 4096 zeitkritische Nachrichten zu versenden.

Um eine Nachricht zu identifizieren, wird die Zieladresse eines Pakets mit 0xFFFFFFFF0000 maskiert (siehe Tabelle 2.1). Falls das Ergebnis dem festgelegten Marker für CT Traffic entspricht, können die letzten 2 Byte als CT ID interpretiert werden und so anhand einer für jedes Gerät im System vergebenen Konfiguration die Priorität der Nachricht abgeleitet werden.

	CT Marker				CT ID	
Ziel	0x03	0x04	0x05	0x06	0x00	0x64
Maske	0xFF	0xFF	0xFF	0xFF	0x00	0x00
Ergebnis	0x03	0x04	0x05	0x06	0x00	0x00

Tabelle 2.1: Erkennung von zeitkritischem Verkehr (Vgl. Bartols, 2010)

Für die Verarbeitung von CT Nachrichten werden spezielle Switches benötigt, welche für die Übertragung dieser konfiguriert werden. CT Nachrichten, welche der Switch über CT Marker und ID erkennt, werden statisch geroutet. Standard Ethernet Pakete können statisch oder dynamisch geroutet werden. TT Nachrichten verdrängen RC und BE Nachrichten und RC verdrängt BE. Die Endgeräte bekommen so nur die für sie konfigurierten CT Nachrichten und dürfen auch nur die für sie in der Konfiguration vorgesehenen Versenden.

Alle Teilnehmer, die in einem Cluster[5] zusammenhängen und CT Nachrichten senden oder empfangen, unterliegen einem Cluster Zyklus (siehe Abbildung 2.5). Dieser richtet sich nach der globalen Zeit auf welche sich die Teilnehmer anhand des in Kapitel 2.4.1 erläuterten Verfahrens Synchronisieren. CT Nachrichten unterliegen einer festen Zeit im Zyklus. Sie dürfen nur zu fest konfigurierten Zeiten im Zyklus versendet werden. RC Nachrichten dürfen immer gesendet werden solange sie ihre zugewiesene Bandbreite nicht überschreiten. Falls die Obergrenze der Bandbreite für eine RC Nachricht erreicht ist muss diese verzögert gesendet werden. BE Nachrichten werden immer gesendet sobald gerade keine CT Nachrichten gesendet werden. Falls der Puffer eines auf der Verarbeitungsstrecke liegenden Geräts nicht

[4] Critical Traffic: Beinhaltet die TT und RC Nachrichtentypen

[5] Alle sich im Cluster befindenden Teilnehmer müssen einen gemeinsamen Cluster Zyklus verwenden, welcher dem kleinsten gemeinsamen Vielfachen der Zyklen im Cluster entspricht, um Kollisionsfrei agieren zu können

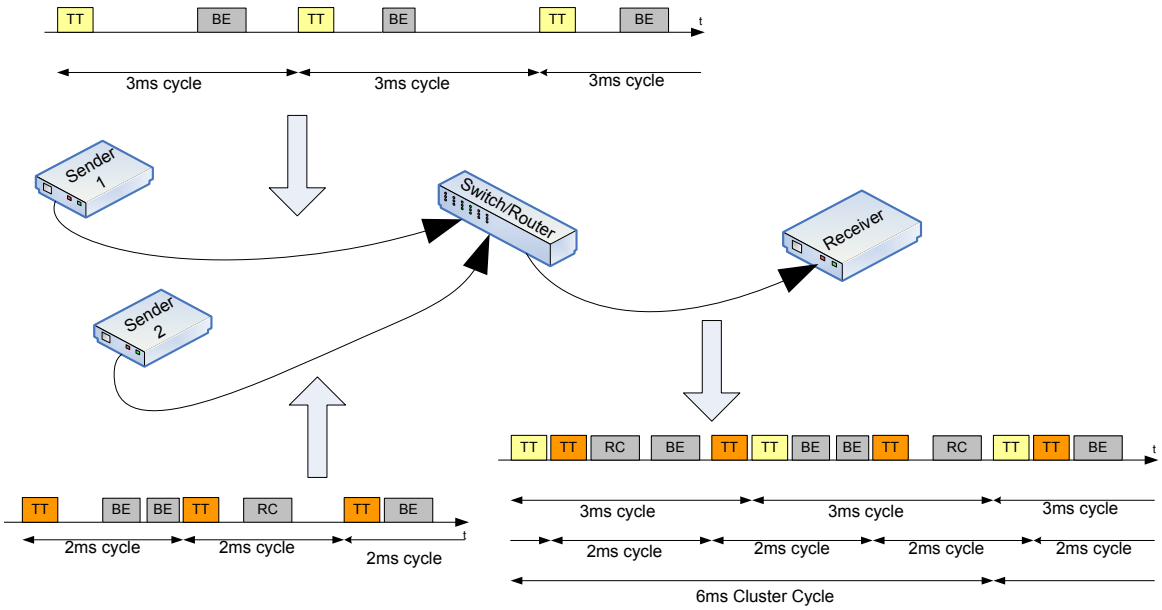

Abbildung 2.5: Integration der Nachrichten in den Zyklus (Quelle: Steiner, 2008)

ausreicht, werden die BE Nachrichten gegebenenfalls verworfen und müssen erneut gesendet werden.

2.4.1 TTEthernet Synchronisierung

Um eine kollisionsfreie Übertragung von CT Nachrichten zu gewährleisten, ist es notwendig das alle Teilnehmer im Cluster sich auf eine globale Zeit Synchronisieren. Dies ist wichtig, da für CT Nachrichten ein deterministisches Verhalten gewünscht ist und aus diesem Grund bei der Übertragung keine unerwarteten Verzögerungen entstehen dürfen.

Eine detaillierte Beschreibung der Synchronisierung ist in der AS6802 Spezifikation (SAE, 2011) zu finden. In Tabelle 2.2 sind die Daten eines Protocol Control Frames (PCF) aufgeführt, welches für die Synchronisierung an die Teilnehmer im Cluster gesendet wird. Abbildung 2.6 ist eine vereinfachte Darstellung der Berechnung der Abweichung der Zeit zur globalen Zeit im Endgerät dargestellt.

2.5 Ethernet im Linux Kernel

Das Netzwerk-Subsystem in Linux ist mächtig und besteht aus einer großen Anzahl von Komponenten und Schnittstellen die wiederum zusammen arbeiten. Hier wird nur auf die

Name	Length	Description
pcf_integration_cycle	32 bits	Integration cycle in which the PCF was sent
pcf_membership_new	32 bits	Bit-vector with a static configured one-to-one relation from a bit to a synchronization master in the system
pcf_sync_priority	8 bits	Static configured value in each synchronization master, synchronization client, and compression master
pcf_sync_domain	8 bits	Static configured value in each synchronization master, synchronization client, and compression master
pcf_type	4 bits	Frame type of a PCF
pcf_transparent_clock	64 bits	Stores the accumulated delay of a Protocol Control Frame from the generator of the PCF up to the consumer of the PCF. Time is represented as multiples of picoseconds

Tabelle 2.2: Protocol Control Frame (PCF) Payload (Quelle. SAE, 2011, S. 31)

Verarbeitung von Frames beim Senden und Empfangen und auf die Zusammenarbeit des Netzwerkkarten Treibers mit dem Linux Kernel eingegangen.

2.5.1 Netzwerkkarten Treiber

Ein Netzwerkkarten Treiber (Vgl. Corbet u. a., 2005, S. 497 ff.) muss die von ihm verwaltete Hardware als Erstes im entsprechenden Hardware-Layer registrieren. Folgend wird am Beispiel einer PCI-Netzwerkkarte ein grober Überblick über die Geräteregistrierung gegeben. Über den Befehl pci_register_driver(struct *pci_driver), welcher üblicherweise direkt bei Initialisieren des Treibers aufgerufen wird, wird der Treiber erstmals dem PCI-Subsystem im Kernel bekannt gemacht. Die Struktur pci_driver (Quellcode 2.1) enthält Zeiger zu Funktionen, die der Treiber bereitstellen muss, damit er vom PCI-Subsystem initialisiert und verwaltet werden kann. Wenn das PCI-Subsystem anhand der Gerätetabelle (id_table) feststellt das ein Gerät für den Treiber verfügbar ist führt es für jedes gefundene Gerät die in der pci_driver Struktur unter “probe“ eingehängte Funktion aus. Hier werden PCI spezifische Konfigurationen vorgenommen und benötigte Ressource vom Kernel angefordert. Anschließend wird über die Funktion alloc_netdev() oder alloc_etherdev(), speziell für Ethernet, die Speicherreservierung und Initialisierung einiger Strukturen, welche für die Verwaltung des Netzwerkinterfaces nötig sind, gemacht. Diese Funktion gibt einen Zeiger auf die net_device Struktur zurück. In dieser Struktur sind alle nötigen Informationen für das Netzwerk-Subsystem gespeichert. Mit der Funktion register_netdev(struct *net_device) wird nun die vollständig initialisierte Struktur dem Netzwerk-Subsystem übergeben und das Interface registriert. Ein

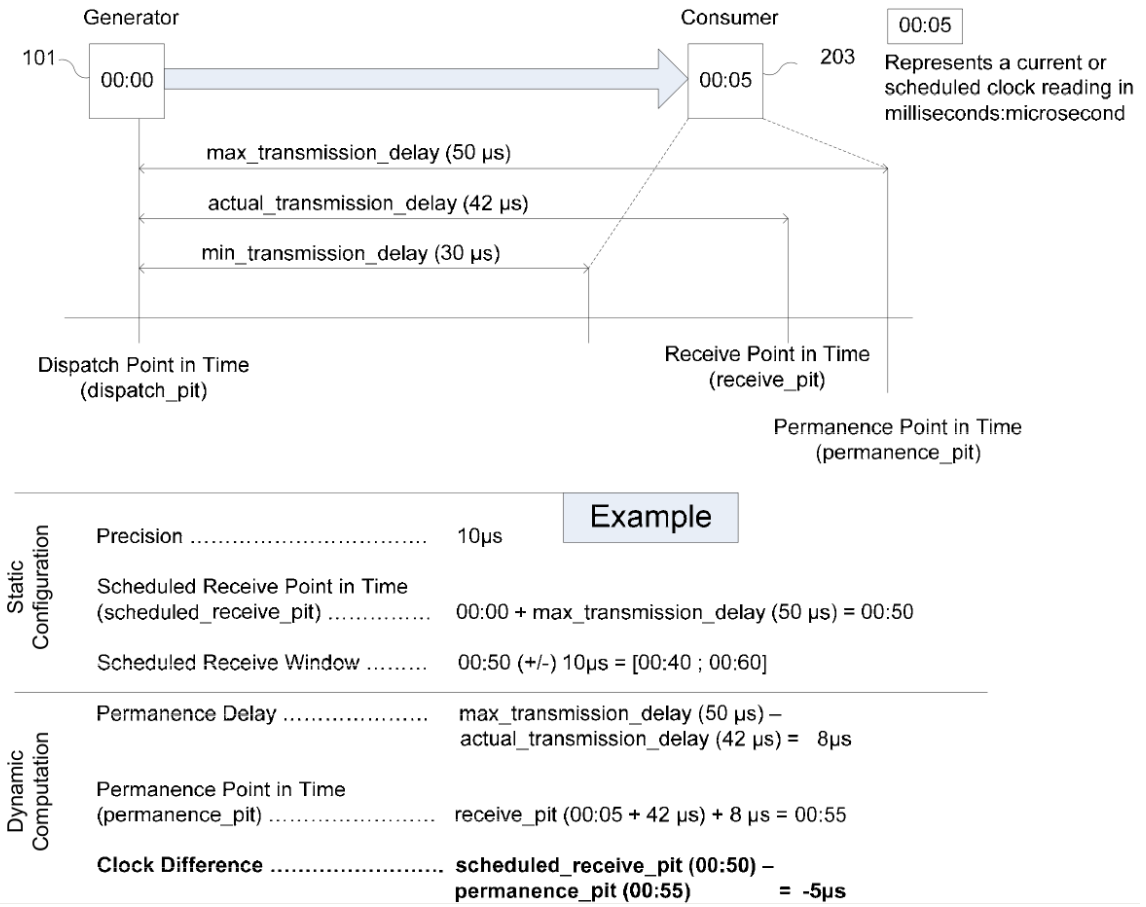

Abbildung 2.6: Berechnung der Abweichung der Zeit im Endgerät (Quelle: SAE, 2011, S. 17)

Teil dieses Containers ist die net_device_opt (Quellcode 4.1) Struktur in welcher die jeweiligen Funktionszeiger, welche das Netzwerk-Subsystem zur Bearbeitung der Pakete und der Konfiguration des Treibers braucht, vorher vom Treibers selbst gesetzt worden sind.

```
struct pci_driver {
        struct list_head node;
        const char *name;
        /* must be non-NULL for probe to be called */
        const struct pci_device_id *id_table;
        /* New device inserted */
        int  (*probe)  (struct pci_dev *dev,
                                const struct pci_device_id *id);
        /* Device removed (NULL if not a hot-plug capable driver) */
        void (*remove) (struct pci_dev *dev);
```

```
        /* Device suspended */
        int  (*suspend) (struct pci_dev *dev, pm_message_t state);
        int  (*suspend_late) (struct pci_dev *dev, pm_message_t state);
        int  (*resume_early) (struct pci_dev *dev);
        /* Device woken up */
        int  (*resume) (struct pci_dev *dev);
        void (*shutdown) (struct pci_dev *dev);
        struct pci_error_handlers *err_handler;
        struct device_driver    driver;
        struct pci_dynids dynids;
};
```

Quellcode 2.1: struct pci_driver aus linux/pci.h

```
struct net_device_ops {
        int             (*ndo_init)(struct net_device *dev);
        void            (*ndo_uninit)(struct net_device *dev);
        int             (*ndo_open)(struct net_device *dev);
        int             (*ndo_stop)(struct net_device *dev);
        netdev_tx_t     (*ndo_start_xmit) (struct sk_buff *skb,
                                           struct net_device *dev);
/* Folgenden Code Abgeschnitten */
```

Quellcode 2.2: struct net_device_ops aus linux/netdevice.h (gekürzt)

Wenn ein Netzwerktreiber initialisiert ist und das Netzwerkinterface im Netzwerk-Subsystem z. B. unter dem Namen eth0 registriert ist kann es automatisch vom System, oder falls dies nicht konfiguriert ist über den Befehl “ifconfig eth0 up“ von einem Benutzer mit root Rechten, aktiviert werden. Bei Aktivierung wird die unter ndo_open (siehe Quellcode 4.1) registrierte Funktion des Treibers aufgerufen und der Treiber macht die letzten Vorbereitungen, wie z. B. eine Interrupt-Anfrage an den Kernel, um die Hardware zum Senden und Empfangen bereit zu machen. Ab diesem Zeitpunkt ist das Interface soweit konfiguriert, dass zumindest RAW-Ethernet Pakete gesendet und empfangen werden können. Das Interface kann nun über den Befehl ifconfig angezeigt werden und es können, falls nicht automatisch geschehen, Konfigurationen für den Empfang von Paketen welche für höhere Netzwerkschichten vorgesehen sind vorgenommen werden.

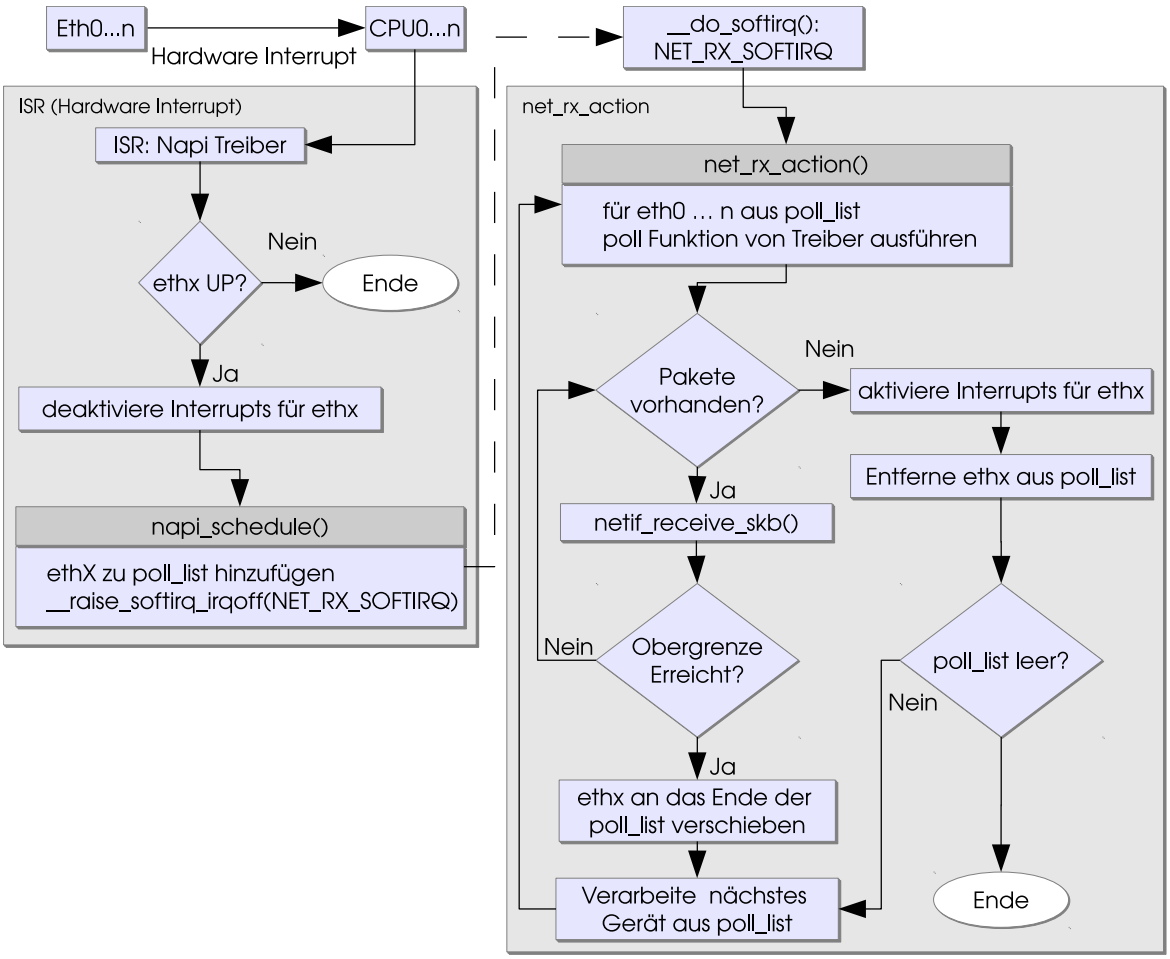

Abbildung 2.7: Vereinfachte Darstellung des Empfangs von Paketen im Linux Kernel

2.5.2 Empfang von Frames

NAPI (New API) (Vgl. Benvenuti, 2006, S. 214) wird die Modifikation des Netzwerk Empfangsframeworks genannt, mit welchem die aktuellen Treiber ihre empfangenen Pakete an den Netzwerk Stack weiterleiten. Napi wurde eingeführt, um ein Ausbremsen des Systems bei einer zu hohen Interrupt-Flut der Netzwerkkarte zu verhindern. Es wird auf eine Mischung aus interruptgestützter und gepollter Verarbeitung zurückgegriffen. Nachfolgend wird auf den Empfang mit dieser Methode etwas näher eingegangen.

Bei Eintreffen eines Pakets in der Netzwerkkarte aktiviert diese die bei der Registrierung zugewiesene (siehe Kapitel 2.5.1) IRQ, was zum Auslösen der vom Treiber bereitgestellten ISR führt. Im Napi Modus wird der Interrupt nicht sofort zurückgesetzt. Die ISR registriert über die Funktion napi_schedule() das entsprechende Netzwerkinterface in einer globalen poll_list und aktiviert den Softirq NET_RX_SOFTIRQ (siehe Kapitel 2.6).

Wenn der NET_RX_SOFTIRQ sofort oder zu einem späteren Zeitpunkt ausgeführt wird (net_rx_action()) werden alle in der poll_list vorhandenen Netzwerkinterfaces durchgegangen und es wird die jeweils zugehörige Pollfunktion, welche der Treiber bereitstellt, aufgerufen. Die Pollfunktion verarbeitet alle angekommenen und während der Verarbeitung ankommenden Pakete, bis zu einer definierten Obergrenze, und gibt sie über die Funktion netif_receive_skb() (oder eine äquivalente Hilfsfunktion) an den Linux Netzwerk Stack weiter. Falls bei Erreichen der Obergrenze an zu bearbeitende Pakete noch nicht alle Pakete verarbeitet wurden wird das Netzwerkinterface wieder an das Ende der poll_list eingehängt, der Softirq NET_RX_SOFTIRQ wieder aktiviert und mit dem nächsten Interface auf die gleiche Art verfahren. Wenn alle Pakete verarbeitet wurden, wird die IRQ des Interfaces wieder aktiviert und das Interface wird aus der poll_list entfernt. Der hier beschriebene Ablauf ist zur Veranschaulichung in Abbildung 2.7 dargestellt.

2.5.3 Senden von Frames

Das Senden von Paketen (Vgl. Benvenuti, 2006, S. 239) wird im Kernel über die Funktion dev_queue_xmit() angestoßen. Diese Funktion nimmt Pakete von höheren Schichten entgegen, reiht diese in die Ausgangsqueue des Interfaces ein und versucht das Paket über die Funktion qdisc_run() zu versenden. Da das Paket vorher in die Ausgangsqueue eingereiht wurde, welche Traffic Kontrolle unterstützt (QoS Schicht), kann es sein, dass schon Pakete vorhanden sind und diese zuerst gesendet werden. Dieser Fall kann eintreten, wenn der Treiber die Queue angehalten hat, da der Speicher in der Hardware keinen Platz für weitere Pakete enthält. Es kann auch vorkommen, dass gerade eine andere CPU die Hardware blockiert, wodurch das Paket nicht versendet werden kann und das Paket dann zu einem späteren Zeitpunkt über die softirq Verarbeitung versendet wird. In dem Fall, dass der Treiber wegen zu wenig Speicher auf der Hardware die Queue gestoppt hat, wird wenn wieder genug Speicher für ein Paket mit maximaler Länge zur Verfügung steht der Softirq NET_TX_SOFTIRQ vom Gerätetreiber ausgelöst. Der Softirq handler net_tx_action() läuft in einer Schleife die Netzwerkinterfaces durch, welche sich für das Senden registriert haben, und ruft für jedes die Funktion qdisk_run auf welche das Versenden über die von dem Treiber unter ndo_start_xmit() (Quellcode 4.1) registrierte Funktion der verbleibenden Pakete in der Queue übernimmt. Falls qdisk_run() ein Paket aus einem der oben erwähnten Fälle erneut nicht versenden kann wird es in der Queue wieder eingereiht und über die Funktion netif_schedule() der Softirq NET_TX_SOFTIRQ erneut aktiviert um die verbleibenden Pakete in einem weiteren Durchlauf von net_tx_action() zu versenden. Der hier beschriebene Ablauf ist zur Veranschaulichung in Abbildung 2.8 dargestellt.

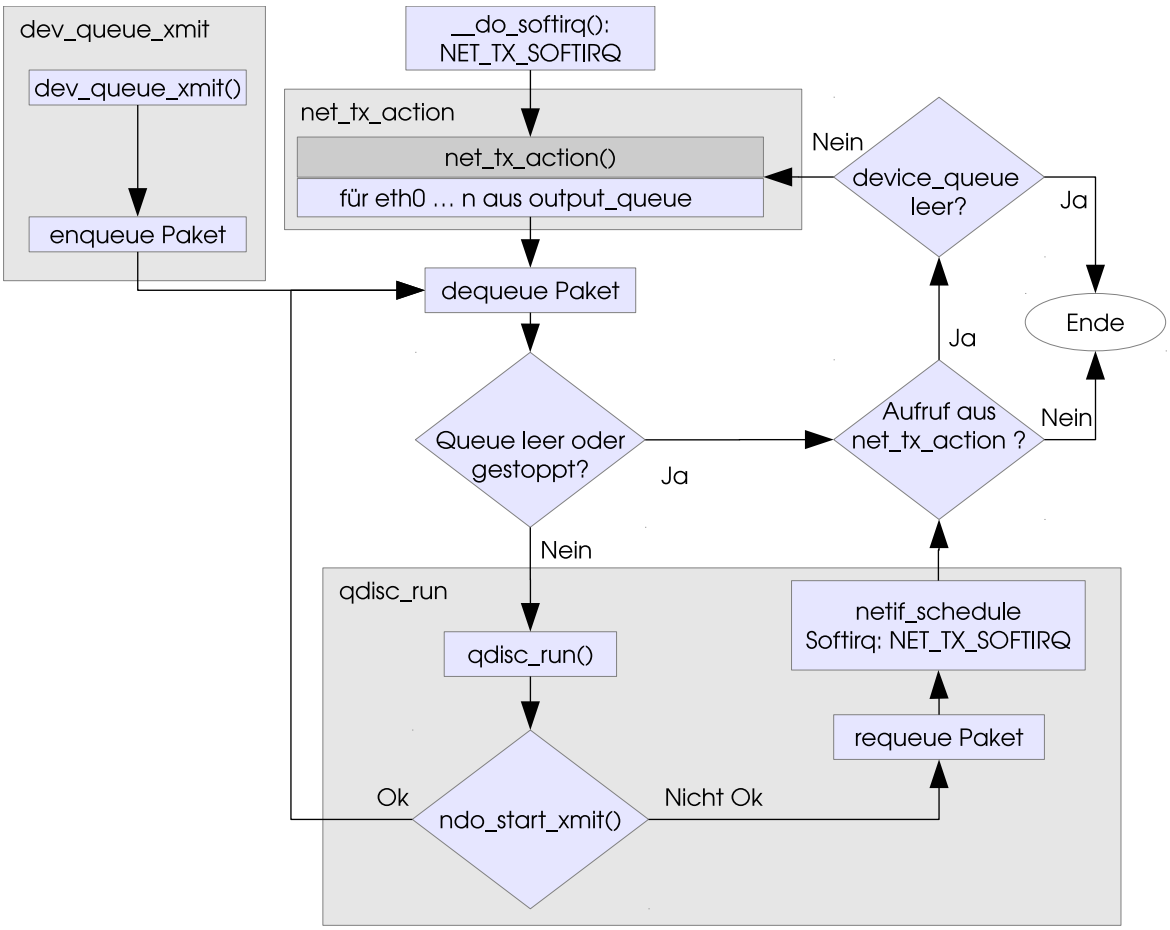

Abbildung 2.8: Vereinfachte Darstellung des Sendens von Paketen im Linux Kernel

2.6 Softirq Verarbeitung unter Linux

Software Interrupts (Softirqs) unter Linux (Vgl. Love, 2010, S. 137) sind statisch und werden zu Kompilierzeit angelegt. Tasklets[6], was spezielle Softirqs sind, können auch dynamisch erstellt und ausgeführt werden. Hier wird hauptsächlich auf die Softirqs eingegangen, da diese von dem Netzwerk-Subsystem intensiv genutzt werden. Eine Auflistung der Softirqs ist in Tabelle 2.6 zu sehen.

Wenn ein Softirq aktiviert wird, muss zur Ausführung danach die Funktion do_softirq() aufgerufen werden. Wenn der CPU sich gerade im Interruptkontext befindet, was mit der Funktion in_interrupt() geprüft wird, tut diese Funktion nichts. Softirqs können nur von Hardware Interrupts unterbrochen werden und haben somit die zweithöchste Priorität im System.

[6]Ein Tasklet kann auch in multiprozessor Systemen nicht parallel ausgeführt werden und wird häufig von Treibern zur Verarbeitung von Daten in einer ISR gestartet.

Name	Priorität	Beschreibung
HI_SOFTIRQ	0	Hochpriorisierte Tasklets
TIMER_SOFTIRQ	1	Timer
NET_TX_SOFTIRQ	2	Senden von Netzerk Paketen
NET_RX_SOFTIRQ	3	Empfangen von Netzwerk Paketen
BLOCK_SOFTIRQ	4	Block Geräte
BLOCK_IOPOLL_SOFTIRQ	5	Block Geräte (polling)
TASKLET_SOFTIRQ	6	Normal priorisierte Tasklets
SCHED_SOFTIRQ	7	Scheduler
HRTIMER_SOFTIRQ	8	High-resulition Timer
RCU_SOFTIRQ	9	RCU locking

Tabelle 2.3: Software Interrupts im Linux Kernel

Software Interrupts können, abgesehen von Tasklets (Vgl. Love, 2010, S. 142), je einmal pro CPU ausgeführt werden. Wenn do_softirq() zur Ausführung kommt, wird mit der Funktion local_softirq_pending() eine Kopie der aktuell aktivierten Softirqs geholt. Anhand der Kopie werden nun alle bis dahin aktivierten Softirqs in der Reihenfolge ihrer Priorität (0 ist die höchste) abgearbeitet und es wird zu jedem der zugehörige Handler aufgerufen. Da Hardware Interrupts aktiviert bleiben, ist es möglich, dass während der Bearbeitung weitere oder dieselben Softirqs wieder aktiviert werden. Softirqs können sich auch selber reaktivieren wie z. B. im net_rx_action (siehe Kapitel 2.5.2) Handler. Hier wird falls die maximale Anzahl der zu verarbeitenden Pakete erreicht ist der NET_RX_SOFTIRQ wieder aktiviert und der nächste Softirq verarbeitet. Wenn alle Softirqs die zu Beginn der Ausführung anstanden abgearbeitet sind, wird geprüft ob in der Zwischenzeit wieder Softirqs aktiviert wurden. Wenn das der Fall ist, werden die neu aktivierten Softirqs ausgeführt bis MAX_SOFTIRQ_RESTART (derzeit ein Wert von 10) erreicht ist. Falls danach noch Softirqs zur Verarbeitung anstehen, wird zur Ausführung der ksoftirqd Thread aufgeweckt.

Der ksoftirqd (Vgl. Love, 2010, S. 146) ist ein Kernel Thread welcher auf jeder CPU zu Systeminitialisierung einmal gestartet wird. Er hat die niedrigste Priorität im System. Dieses Verfahren hat den Hintergrund das bei einer zu hohen Anzahl von Softirqs, was z. B. bei starkem Netzwerkverkehr vorkommen kann, die CPU nur noch im Interrupt-Context arbeitet und Benutzerprozesse solange ausgebremst werden. Dieser Thread tut nichts weiter als in einer Schleife die Softirqs mit do_softirq() erneut zur Ausführung zu bringen und dann wieder zu schlafen. Da er aber nicht im Interrupt-Context läuft, können andere Prozesse bevor die Softirqs erneut ausgeführt werden auch für kurze zeit die CPU beanspruchen. Mit dieser Lösung wurde ein Kompromiss aus schneller Verarbeitung von Softirqs und Fairness gegenüber Benutzerprozessen gewählt.

3 Design

Dieses Kapitel beschäftigt sich mit dem grundlegenden Aufbau des zu entwickelnden Treibers.

3.1 Anforderungen

Virtuelles Netzwerkinterface für BE Traffic

Senden

Empfangen

Nutzung von Socket basierten Anwendungen

ARP, DHCP, Routing, NAT, Firewall etc.

TTEthernet Controller für CT Traffic

TTE API implementieren

Konfiguration aus config.c Datei lesen

Senden von TT Nachrichten

Empfangen von TT Nachrichten

Zeitstempel

Filtern der Nachrichten nach Typ in Buffer

Scheduling der Tasks mit Prioritäten

TX TT (Task)

Sync (Task)

RX-TT Callback (Task)

TT-Task

TX-BE (Task)

RX-BE (Task)

Acceptance Window für CT Nachrichten

BE Traffic unter Kontrolle des Schedulers stellen

Synchronisations Client

Synchronisations Master

Synchronisations Compression Master

3.2 Virtuelles Netzwerkinterface

Ein virtuelles Netzwerkinterface wird im Userspace wie jedes Netzwerkinterface unter Linux angezeigt und kann äquivalent benutzt werden. Im Gegensatz zu einem normalen Netzwerkinterface hat es aber keine eigene Hardware zum Senden und Empfangen der Daten, sondern nutzt ein anderes Interface für die Kommunikation mit der Außenwelt. Der TTEthernet Treiber soll ein virtuelles Netzwerkinterface für die Kommunikation über Ethernet (Best Effort) Traffic auf Socketebene implementieren. Das hat zum Vorteil das alle gängigen Programme für Linux die über das Netzwerk Daten senden und empfangen weiterhin ohne Modifikationen benutzt werden können. Folgend wird die Funktionsweise des virtuellen Interfaces beschrieben, was auch auf Abbildung 3.1 grafisch dargestellt ist.

Um dass zu ermöglichen, muss der Treiber die Schnittstelle zum Linux Stack implementieren und sich bei diesem als Netzwerkinterface (in Abbildung 3.1 tte0) registrieren. Die Frames die der Linux Stack an das virtuelle Interface übergibt müssen zwischengespeichert werden damit sie verzögert gesendet werden können und nicht den Critical Traffic (CT) unterbrechen. Der TTEthernet Treiber soll dann, anhand eines internen Schedulers, wenn gerade kein CT Frame verarbeitet wird die zwischengespeicherten Frames über den Treiber der realen Netzwerkkarte versenden.

Da das virtuelle Interface eine andere MAC[1] Adresse haben kann, als die von der realen Netzwerkkarte, muss diese im Promiscuous-Modus, was den Netzwerkkartentreiber dazu veranlasst alle ankommenden Frames weiter zu reichen, konfiguriert sein. Ankommende Frames werden vom Netzwerkkartentreiber an den TTEthernet Treiber weitergereicht, welcher dann zwischen BE und CT Frames filtert und sie in entsprechende Buffer sortiert. Beim Empfang von BE Frames geht es darum diesen so kurz wie möglich zu machen, um nicht

[1] Die MAC-Adresse ist die weltweit eindeutige Hardware Adresse einer Netzwerkkarte

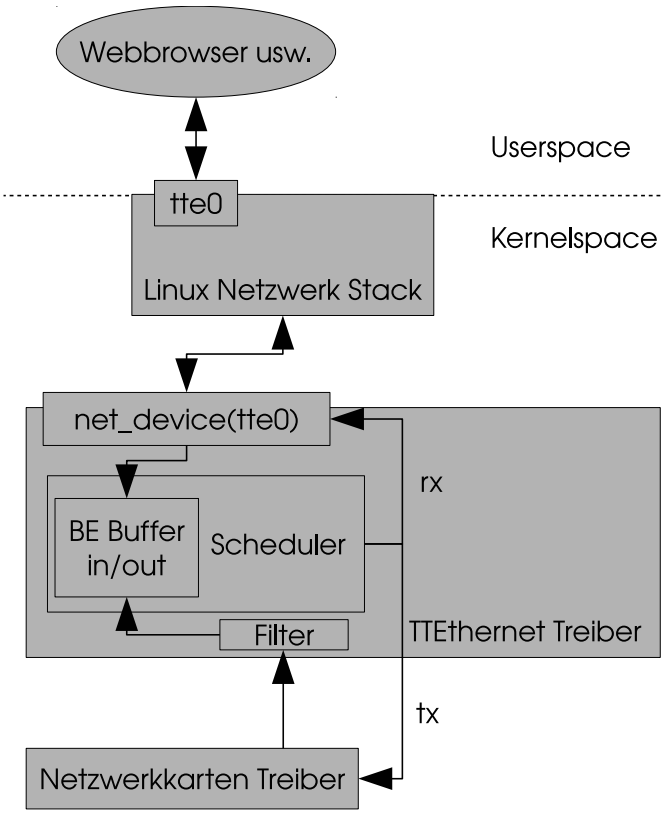

Abbildung 3.1: Aufbau des Virtuellen Netzwerkinterfaces

eine folgende CT Nachricht zu verdrängen. Die zwischengespeicherten BE Frames sollen dann, wenn keine Verarbeitung von CT Frames ansteht, verzögert an den Linux Stack über das virtuelle Interface weitergereicht werden.

3.3 TTEthernet im Treiber

Auf die TTEthernet spezifischen Funktionalitäten des Treibers wird über die TTEthernet API von TTTech[2] (im folgenden API) zugegriffen. Hier werden Funktionen zur Konfiguration des TTEthernet Controllers zur Verfügung gestellt sowie der Zugriff auf die Ein- und Ausgangsbuffer für CT Frames ermöglicht. Hier wird nach erfolgreichem Initialisieren, Konfigurieren und Starten des Controllers über die entsprechenden Funktionen ein Zugriff auf die Buffer ermöglicht. Die API wird im Kernelspace zur Verfügung gestellt. Über Diese kann ein Kernelmodul Daten senden und empfangen, indem es über die API-Funktionen auf den entsprechenden Buffern liest oder schreibt. Der Scheduler ist dann für das rechtzeitige Senden

[2]TTTech Computertechnik AG Schönbrunner Straße 7 A-1040 Wien, Österreich

der Nachrichten im Buffer zuständig. Empfangene Nachrichten werden sofort in den entsprechenden Empfangsbuffer geschrieben und können dann über die API gelesen werden. Eine Übersicht der TTEthernet Treiber Architektur inklusive des virtuellen Netzwerkinterfaces ist in Abbildung 3.2 dargestellt.

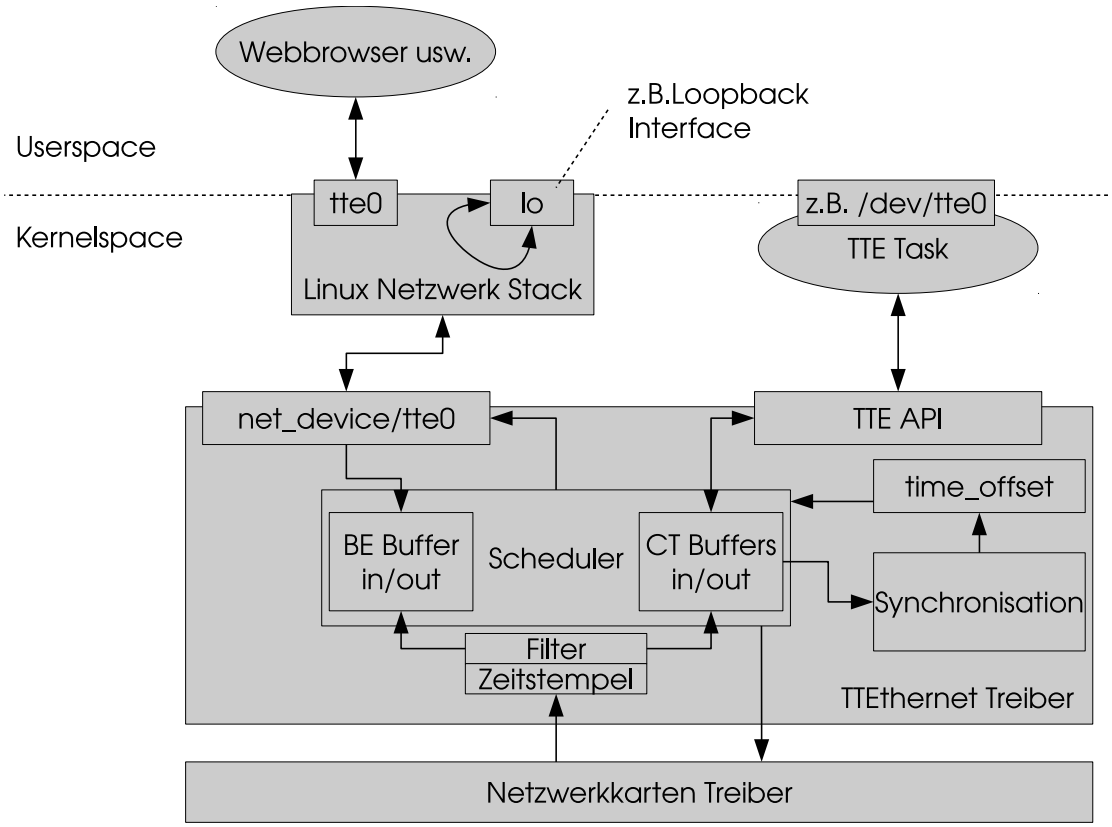

Abbildung 3.2: Aufbau des TTEthernet Treibers mit virtuellem Netzwerinterface

Da die TTEthernet API von TTTech Version 2.0 Rate-Constrained (RC) Nachrichten nicht unterstützt wird auf diese Nachrichtenklasse nicht weiter eingegangen.

Buffer

Wie bereits in Kapitel 2.4 beschrieben, werden CT Frames anhand ihrem CT Marker und ihrer CT ID identifiziert. Es wird in der Konfiguration für jede CT ID festgelegt, wann diese gesendet wird und optional auch, wann diese empfangen wird. Der Zugriff auf die Buffer über die API erfolgt jeweils mit der Angabe der CT ID. Die API bietet die Verwendung von zwei verschiedenen Buffertypen an, welche der Treiber unterstützen soll.

Queued Buffer Der Queued-Buffer arbeitet nach dem FIFO[3] Prinzip. Daten werden hinten eingefügt und vorne raus geholt. In der Konfiguration eines Buffers muss die Länge der Queue angegeben werden.

Double Buffer Der Double-Buffer kann nur ein Frame speichern. Es soll aber gleichzeitiger Lese- und Schreibzugriff möglich sein. Ein neu eintreffender Frame überschreibt den alten Wert erst, wenn kein lesender Zugriff mehr auf dem Buffer stattfindet.

In der Konfiguration muss für jede CT Nachricht ein entsprechender Buffer eingetragen werden. Zusätzlich zum Typ des Buffers wird die Richtung (Eingang/Ausgang) festgelegt. Diese Buffer müssen beim Laden der Konfiguration angelegt werden.

Zeitstempel

Um zu überprüfen, ob eine CT Nachricht rechtzeitig angekommen ist oder um die Verzögerung bei einem Protokoll Control Frame (PCF) vom Empfang bis zur Verarbeitung in der Synchronisationsroutine zu berechnen, muss jedes empfangene CT Frame mit einem Zeitstempel versehen werden. Dies muss so früh wie möglich geschehen, um eine genaue Aussage über die Ankunft der Nachricht machen zu können. Der Zeitstempel ist also das Erste, was bei Eintreffen der Nachricht im Treiber gesetzt wird. Da der TTEthernet Treiber die empfangene Nachricht von dem Treiber der Netzwerkkarte erhält, ist zu prüfen, wie stark der Zeitstempel im TTEthernet Treiber von der realen Empfangszeit abweicht. Wenn ein statisches Delay entsteht, muss dieses auf den Zeitstempel addiert werden.

Acceptance Window

Für CT Nachrichten kann in der Konfiguration eine Deadline angegeben werden. Bei Empfang soll, falls eine Deadline konfiguriert ist, überprüft werden, ob die Nachricht in ihrem Zeitfenster liegt und falls dies nicht der Fall ist verworfen werden.

Filter

Nach dem Setzen des Zeitstempels werden die Nachrichten gefiltert. Anhand des CT Markers und der CT ID, welche dem Controller über die Konfiguration bekannt gemacht werden, sollen CT Nachrichten in die entsprechenden Eingangsbuffer sortiert werden.

[3] FIFO: first in first out

Tasks und Priorisierung

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 PCF Frames 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 kommen 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.

3.4 TTEthernet Treiber Scheduling

Der TTEthernet Stack braucht für die Ausführung von zeitlich abhängigen Ereignissen und Aufgaben die zu solchen in Beziehung stehen, eine Strategie um diese zu planen. Um vorgegebenen Echtzeitbedingungen einzuhalten, ist es wichtig, dass die kritischen Aufgaben ohne Verzögerung ausgeführt werden.

Durch die Möglichkeit der Vergabe von Prioritäten bei dem verwendeten Kernel mit Real-Time (RT) Patch kann ein höher priorisierter Prozess einen niedriger priorisierten unterbrechen. Der Linux Scheduler bietet zwar die Möglichkeit über Tasklets und Kernel Threads zur Ausführung bestimmte Prozesse zu planen, es ist aber nicht vorgesehen einen Zeitpunkt der Ausführung zu definieren. Je nach Priorität wird der Prozess sofort oder erst später ausgeführt. Dies macht es notwendig einen eigenen Scheduler im Stack zu implementieren, der zur angegebenen Zeit die geplanten Ereignisse ausführt.

Über die Konfiguration wird eine Periodendauer (Zyklus) festgelegt. Es gibt Aufgaben, welche relativ zur Periodendauer (In Period) geplant oder spontan (In Impulse) ausgeführt werden müssen.

In Period TX-TT, RX-TT Callback, TT-Task

In Impulse Sync, TX/RX BE

Die In-Period Tasks sollten von dem Systemdesigner, welcher die Konfiguration für den TTEthernet-Controller erstellt, so geplant werden, dass sie sich nicht überschneiden. Da die Tasks unterschiedliche Prioritäten haben werden, falls doch Überschneidungen in der Ausführung entstehen, niedriger priorisierte Tasks gegebenenfalls verdrängt werden. In-Impulse Tasks werden spontan aktiviert und müssen vom Scheduler ihrer Priorität entsprechend zur Ausführung gebracht werden, wenn keine höher priorisierten Tasks anstehen.

Das Versenden von Frames blockiert die Hardware mindestens so lange, bis das aktuelle Frame versendet ist und kann nicht unterbrochen werden. Aus diesem Grund ist es nötig bei Versenden von hoch priorisierten Nachrichten darauf zu achten das für die Zeit vorher, welche für das Senden eines Frames mit maximaler Länge benötigt wird, kein niedriger priorisierter Task die Hardware blockiert. Damit soll sichergestellt werden das TT Nachrichten die Hardware sofort zum Senden zur Verfügung steht. Jedes zu versendende Frame muss also der Kontrolle des Schedulers unterliegen.

Zykluszeit

Um TT Nachrichten und Tasks pünktlich zu planen, muss der Scheduler in einem synchronisierten Zyklus laufen. Da keine zusätzliche Hardware verwendet wird, steht nur der Systemtimer als Zeitreferenz zur Verfügung. Die Zeit des Systemtimers kann von dem Synchronisationsmodul aber nicht einfach korrigiert, werden da sonst andere Prozesse im System, die von dem Timer abhängig sind, nicht mehr korrekt funktionieren. Also kann die Systemzeit nur als Referenz (cycle in folgendem Pseudocode) zum Start des Zyklus dienen.

```
cycle = ktime_get(); #hole Systemzeit = Zyklusstart
event = cycle + event_time; #berechne absolute Systemzeit fuer
                                                    ein Event
next_cycle = cycle + period; #berechne naechsten Zyklus Start
```

Es wird also die aktuelle Systemzeit zum Zyklusstart gespeichert und Ereignisse werden dann relativ zu dieser Zeit geplant. Die Zeitreferenz muss vom Synchronisationsmodul entsprechend korrigiert werden, wenn eine Abweichung von der globalen Zeit festgestellt wird. Aktuelle Linux Kernel benutzen eine Kernelzeit, welche vom Beginn des Systemstarts in Nanosekunden hochzählt.

Wenn auf die Systemzeit, welche den Zyklusstart referenziert, die Zeit eines Events im Zyklus addiert wird, erhält man die absolute Systemzeit, zu welcher das Event ausgeführt werden soll. Mit dieser absoluten Zeit kann ein Timer programmiert werden, um das Event dann zu gewünschtem Zeitpunkt auszuführen. Der Wechsel in einen neuen Zyklus stellt auch ein

Ereignis dar, welches anhand des Zyklusstarts und der konfigurierten Periodendauer berechnet werden kann und zu gegebenen Zeitpunkt die Referenzzeit für den neuen Zyklus aktualisiert.

4 Realisierung

Dieses Kapitel beschäftigt sich mit den Details der Implementierung der einzelnen Komponenten des TTEthernet Treibers.

4.1 Treiber Aufbau

Unter Linux werden Erweiterungen des Kernels die zur Laufzeit geladen werden sollen als Module realisiert. Der TTEthernet Treiber ist dementsprechend ein Kernelmodul, welches wenn es für den laufenden Kernel kompiliert wurde, geladen werden kann. In linux/module.h sind die nötigen Funktionen, welche für die Erstellung eines Moduls gebraucht werden, definiert. Die Makros module_init(tte_init_module); und module_exit(tte_cleanup); definieren die Funktionen des Moduls, welche bei Laden bzw. entladen des Moduls vom Kernel aufgerufen werden. tte_init_module ist die Funktion des Treibers, welche beim Laden des Moduls aufgerufen wird und Speicher für das virtuelle Netzwerkinterface beim Kernel anfordert und es registriert. tte_cleanup wird beim Entladen des Moduls aufgerufen und ist dafür zuständig das virtuelle Netzwerinterface wieder zu deregistrieren, den Speicher und die Ressourcen, welche beim Start und während der Laufzeit des Moduls vom Kernel angefordert wurden, wieder freizugeben.

Über das Makro EXPORT_SYMBOL(funktion_pointer); können Funktionen des eigenen Moduls für andere Module verfügbar gemacht werden. So wird der Zugriff auf die TTE API von anderen Modulen aus realisiert. Der TTEthernet Controller wird dann erst, wenn ein Modul welches die API verwenden möchte, aktiviert indem die Init-Funktion der API aufgerufen wird.

4.2 Tasks als Kernel Threads

Da Linux mit dem RT-Patch eine Priorisierung ermöglicht, aber keine zeitgesteuerte Ausführung von Prozessen vorsieht, werden die einzelnen Tasks (siehe Kapitel 3.3), welche vom TTE Scheduler oder durch externe Ereignisse zu gegebener zeit gestartet werden, als

Kernel Threads realisiert, um sie nach dem Start unter Kontrolle des Betriebssystem Schedulers arbeiten zu lassen. Dieser ist dann für die Ausführung entsprechend der vergebenen Prioritäten der Threads zuständig. Da Tasks, welche Frames versenden auf die Netzwerkhardware zugreifen, indem sie sich ein Lock holen, ist es hier nötig für niedrig priorisierte Sendetasks weitere Maßnahmen zu ergreifen, um Prioritätsinversionen zu vermeiden, auf welche in Kapitel 4.4 im Einzelnen eingegangen wird.

4.3 Virtuelles Netzwerkinterface

Das virtuelle Netzwerkinterface implementiert die net_device Schnittstelle des Linux Kernels.

Registrierung im Kernel

In der Initialisierungsfunktion des Moduls wird dem Kernel das virtuelle Netzwerkinterface mit der Funktion register_netdev() bekannt gemacht. Dieser wird ein Zeiger auf die net_device Struktur übergeben, welche vorher initialisiert wurde und Zeiger auf die Funktionen des Moduls enthält, welche für die Ausführung verschiedener Aufgaben zuständig sind und welche die Schnittstelle vorsieht. Die net_device Struktur enthält hauptsächlich Informationen, die der Kernel intern für die Verwaltung benötigt und um was für eine Art von Netzwerkinterface es sich handelt, in diesem Fall Ethernet. Bei der Initialisierung wird der Name des virtuellen Interfaces mit tte%d angegeben, wobei %d für die nächste freie Nummer steht, falls schon ein Interface mit demselben Namen existiert. In der Struktur net_device_ops, welche im Quellcode 4.1 abgebildet ist, werden die Treiberfunktionen des virtuellen Interfaces definiert. Ein Zeiger auf diese Struktur muss vor der Registrierung beim Kernel in der net_device Struktur der Variable netdev_ops zugewiesen werden.

```
static const struct net_device_ops tte_ops =
{
  .ndo_open = tte_open,
  .ndo_stop = tte_release,
  .ndo_set_config = tte_config,
  .ndo_start_xmit = tte_be_tx,
  .ndo_do_ioctl = tte_ioctl,
  .ndo_get_stats = tte_stats,
  .ndo_change_mtu = tte_change_mtu,
};
```

Quellcode 4.1: struct net_device_ops aus linux/netdevice.h

Best Effort Buffer

Für die Best Effort Frames wird eine Sende- und Empfangsqueue angelegt. Dies geschieht zusammen mit der Initialisierung der Buffer für Critical Traffic, welche in Kapitel 4.4 beschrieben werden. Es werden Queues mit einer Länge von jeweils 1000 Frames zur Verfügung gestellt. Best Effort Frames können somit erst versendet bzw. empfangen werden, wenn die Buffer angelegt sind.

Verarbeitung der Frames vom Linux Stack

Die Ethernet Frames erreichen das Modul über die Funktion tte_be_tx(), welche wie in Quellcode 4.1 zu sehen ist, in der Variablen ndo_start_xmit dem Linux Stack bekannt gemacht wurde. Ankommende Frames werden, wenn der TTE Scheduler und die Buffer initialisiert sind, in dem Ausgangsbuffer für BE Frames gespeichert. Nach erfolgreichem Speichern wird mit der Funktion schedule_bg_tx() dem TX_BE Task (siehe Kapitel 3.3), welcher als Kernel Thread realisiert ist signalisiert, dass ein Frame versendet werden kann. Der Thread, welcher bei der Initialisierung des Schedulers gestartet wird, wartet in einer Wait-Queue, bis das Signal an diese gesendet wird. Der TX_BE Kernel Thread hat die niedrigste RT Priorität im Treiber und wird somit, falls ein höher priorisierter Task ansteht, vom Scheduler des Betriebssystems verdrängt bzw. erst ausgeführt, wenn keine höher priorisierten Aufgaben mehr anstehen. Da beim Senden aber ein exklusiver Zugriff auf die Hardware der Netzwerkkarte erfolgt muss in dem TX_BG Thread überprüft werden ob in der Zeit, die für das Versenden eines maximal zulässigen Ethernet Frames benötigt wird, nicht ein höher priorisierter Task eine Nachricht senden möchte. Diese Funktion ist in dem aktuellen Prototyp noch nicht implementiert. Es müsste die Liste der anstehenden Events und deren Startzeiten mit der maximal erwarteten Endzeit des Sendens des BE Frames abgeglichen und gegebenenfalls gewartet werden, bis der höher priorisierte Task ausgeführt wurde. Der Thread versendet das Frame über die Funktion tte_tx, welche in Kapitel 4.6 genauer beschrieben wird.

Übergeben der Frames an den Linux Stack

Die empfangenen Frames, welche von der Empfangsroutine in Kapitel 4.7 beschrieben, werden von dem RX_BE Task weiterverarbeitet und an den Linux Stack übergeben. Der RX_BE Task hat, wie der Task zum Senden von BE Frames, die niedrigste RT Priorität im Treiber. Er wird ebenfalls bei der Initialisierung des TTE Schedulers gestartet und wartet in einer wait_queue auf ein Signal. Wenn die Empfangsroutine ein Frame in den BE Eingangsbuffer schreibt, wird die entsprechende wait_queue informiert und der Thread wird vom Linux Scheduler entsprechend seiner Priorität zur Ausführung gebracht. Der Thread holt jeweils

ein Frame aus dem Eingangsbuffer und gibt es dann über die Funktion netif_rx, welche im realen Treiber per define überschrieben wurde (siehe 4.7), an den Linux Netzwerk Stack weiter. Der Thread arbeitet so lange, bis keine Frames mehr im Buffer vorhanden sind und wartet anschließend wieder in der wait_queue auf neue Frames. Frames werden im Linux Kernel in einer Socketbuffer Struktur verwaltet, welche Informationen für die Verarbeitung im Stack bereitstellt. Diese Struktur wird bereits von dem realen Netzwerkkartentreiber mit Informationen gefüllt. Hier werden vor der Übergabe an den Stack noch die Informationen des realen Netzwerkinterfaces durch die des virtuellen ausgetauscht, um eine Zuordnung der Frames im Linux Stack zu dem virtuellen Netzwerkinterface zu ermöglichen.

4.4 TTEthernet im Treiber

Der TTEthernet Controller ist für die Verarbeitung der CT Frames verantwortlich. Er implementiert die TTE API von TTTech und arbeitet mit einem internen Scheduler, auf welchen in Kapitel 4.5.1 genauer eingegangen wird. Des weiteren verfügt er über ein Synchronisations Modul und einen Bufferpool, welcher zum Speichern von Frames verwendet wird. Die Details des TTEthernet Controllers werden im Folgenden erläutert.

Buffer

Der verwendete Buffer wurde aus der Bachelorarbeit von Kai Müller (Vgl. Müller, 2011, S. 43 ff.) übernommen und für Linux portiert. Es werden Queued- und Doublebuffer unterstützt, wie von der TTE API gefordert. Der Aufbau sowie die Implementierung ist dort detailliert beschrieben. Aus diesem Grund wird hier darauf nicht genauer eingegangen.

Zeitstempel

Alle ankommenden Frames werden in der Empfangsroutine (siehe Kapitel 4.7) mit einem Zeitstempel versehen. Dieses dient der Berechnung des dynamischen Empfangsdelays, welches für die Berechnung der Zykluszeit (siehe Kapitel 3.4) benötigt wird. Der Zeitstempel wird in der Variable tstamp der Socketbuffer Struktur, welche vom Treiber übergeben wird, gespeichert. Die aktuelle Kernelzeit wird mit der Funktion ktime_get() ausgelesen, da diese auch als Referenzzeit der Zykluszeit dient, welche die Zeit in Nanosekunden seit dem Systemstart zurückgibt.

Acceptance Window

Das Acceptance Window wurde nicht implementiert, da aktuell die Synchronisation keine ausreichende Genauigkeit erzielt, welche für die Überprüfung der Zeit ob das Frame rechtzeitig angekommen ist, gebraucht wird.

Filter

Nachdem die Frames einen Zeitstempel bekommen haben, werden sie in die entsprechenden Empfangsbuffer sortiert. CT Frames werden über die Zieladresse, wie in Kapitel 2.4 beschrieben identifiziert. Als Erstes werden die Zieladresse, welche dem CT Marker entsprechen (siehe Abbildung 2.4), mit der 0xFFFFFFFF0000 maskiert und dann mit der konfigurierten Cluster-ID, welche dem CT-Marker um 2 Byte nach links geshiftet entspricht, verglichen (siehe Tabelle 2.1). Wenn diese Werte übereinstimmen, handelt es sich um ein CT Frame und die letzten 2 Byte werden als CT ID interpretiert. Daraufhin werden sie anhand ihrer ID in den entsprechenden Empfangsbuffer kopiert. Alle Nachrichten, die nicht als CT Nachricht erkannt werden, kommen in den BE Eingangsbuffer. Nachrichten, die in der Zieladresse den CT Marker enthalten, für welche aber der Empfang nicht konfiguriert wurde, werden als fehlerhaft interpretiert und verworfen, da kein Buffer für diese CT ID angelegt wurde.

Tasks und Priorisierung

Die Tasks werden größtenteils als Kernelthreads ausgeführt. Auf die Funktionsweise der einzelnen Tasks wird im Folgenden eingegangen. Als Erstes wird auf die Initialisierung und Priorisierung der Kernelthreads eingegangen.

Mit der Initialisierung des Schedulers werden auch die Kernel Threads initialisiert. Die Funktion, welche für die Initialisierung verwendet wird, ist in Quellcode 4.2 abgebildet und wird kurz erläutert. Für jeden Kernel Thread wird vorher ein Zeiger auf eine task_struct angelegt und zusammen dem Zeiger auf den Thread den Daten und der sched_param Struktur übergeben. Als Erstes wird der Thread mit kthread_create() im Kernel erstellt, indem der Funktion, Thread und Datenzeiger sowie ein Name übergeben wird. Wenn der Thread erstellt und der zurückgegebene Zeiger auf die task_struct gespeichert, ist wird der Task priorisiert und RT Scheduling aktiviert. Dies geschieht mit der Funktion sched_setscheduler(), welcher die Task-Struktur, die Scheduling Strategie sowie ein Zeiger auf die sched_param Struktur, übergeben wird. Als Scheduling Strategie wird SCHED_FIFO übergeben, um RT Scheduling mit der FIFO Strategie (siehe Kapitel 2.2) zu verwenden. In der sched_param Struktur wurde vorher die RT Priorität des zu initialisierenden Threads in der Variable sched_priority

festgelegt. Mit der Funktion wake_up_process(struct task_struct *) wird der entsprechende Thread erstmalig gestartet. In der Funktion, welche den Thread realisiert, muss falls er nicht einmalig ausgeführt werden soll, mithilfe einer Schleife (veranschaulicht in Quellcode 4.3) dafür gesorgt werden, dass er erst terminiert, wenn kthread_stop(struct task_struct *) aufgerufen wird. In diesem Fall wird der Thread direkt in eine wait_queue eingereiht, bis er durch ein Signal auf diese geweckt wird.

```
int tte_thread_init(struct task_struct **kthread, int
(*threadfn)(void *data), void *data, struct sched_param param)
{
  static int num = 0;
  num++;
  // Create Kernel thread
  *kthread = (kthread_create(threadfn, data, "tte_kthread%d", num));
  if (IS_ERR(*kthread))
    {
      return PTR_ERR(*kthread);
    }
  // Set scheduling priority ans strategie
  if (sched_setscheduler(*kthread, SCHED_FIFO, &param) == -1)
    {
      return -1;
    }
  // Start thread
  wake_up_process(*kthread);
  return 0;
}
```

Quellcode 4.2: Funktion zur Initialisierung und Priorisierung der Kernel Threads

```
  while (!kthread_should_stop())
    {
      wait_event_interruptible(bg_tx_event,
          (bg_tx_event_count > 0 || kthread_should_stop()));
      if (kthread_should_stop())
        {
          break;
        }
      //do something
      bg_tx_event_count--;
    }
```

$$
\begin{aligned}
pcf\_transparent\_clock_n = {} & pcf\_transparent\_clock_{n-1} \\
& + dynamic\_receive\_delay_n \\
& + static\_receive\_delay_n \\
& + wire\_delay_n
\end{aligned}
$$

Abbildung 4.1: Berechnung: Transparent Clock im Endgerät (n) (Quelle: SAE, 2011, S. 34)

Quellcode 4.3: Beispiel der Schleife in einem Kernelthread

TX-TT Task Der TX-TT Task wird im Gegensatz zu den anderen Tasks nichts als Kernel Thread aufgerufen sondern wird direkt im Interruptkontext des Timers, welcher von dem Scheduler (siehe Kapitel 4.5) gestartet wurde, ausgeführt. Da er die höchste Priorität hat und der Timer Interrupt auch in einem Kernel mit RT Patch nicht unterbrechbar ist (Vgl. Yaghmour u. a., 2008, S. 390) kann dadurch ein Kontextwechsel vermieden werden. Der Task holt die zu versendende Nachricht aus dem entsprechenden Buffer und übergibt sie der Senderoutine, welche in Kapitel 4.6 näher beschrieben wird.

Sync Task Da während der Entwicklung kein Synchronisations Master/Compressionmaster vorhanden war der den Synchroniations-Algorithmus nach Vorgabe der AS8602 Spezifikation implementiert hat, sondern nur PCF Frames in einem definierten Zyklus sendet, wurde eine Synchronisierung auf die PCF Frames von diesem Master implementiert. Bei dem speziellen Master ist nur pcf_type und pcf_transparent_clock (siehe Tabelle 2.2) gesetzt. Der pcf_type ist immer 0x2, was für ein Integration-Frame steht. Die Transparent Clock des Masters wird nach Vorgaben der AS8602 Spezifikation berechnet. Der Sync Task berechnet anhand seiner Transparent Clock (siehe Abbildung 4.4) welche anhand der Transparent Clock des Geräts, welches das PCF Frame sendet, die Abweichung von seiner eigenen Zykluszeit (Local Clock) nach dem Transparent Synchronization Verfahren wie es in der AS6802 Spezifikation (Vgl. SAE, 2011, S. 15 ff.) beschrieben ist. Der Sync Task ist als Kernel Thread implementiert und wird direkt bei Empfang eines PCF Frames über ein Signal an seine wait_queue aktiviert.

RX-TT Callback Der RX-TT Callback Task ist nicht implementiert.

TT-Task Der TT-Task ist nicht implementiert.

TX-BE Task Der TX-BE Task ist als Kernel Thread implementiert und wird mit der Initialisierung des Schedulers gestartet. Er wird aktiviert, wenn ein ankommendes Frame vom Linux Stack ein dem entsprechenden Ausgangbuffer gespeichert wird, indem ein

Signal an seine wait_queue geschickt wird. Die Funktionsweise wird in kapitel 4.3 beschrieben.

RX-BE Task Der RX-BE Task hat wie der TX-BE Task die niedrigste Priorität im TTE System und ist auch vom Aufbau analog zu diesem. Der Kernel Thread wird in diesem Fall von der Empfangsroutine, welche in Kapitel 4.7 beschrieben wird, über ein Signal an die entsprechende wait_queue aktiviert. Die genaue Funktionsweise wurde bereits in Kapitel 4.3 beschrieben.

4.5 TTEthernet Treiber Scheduling

Der TTE Scheduler wird über die TTE API, nachdem die tte_init() aufgerufen wurde, über die Funktion tte_configure() konfiguriert und initialisiert. Nach der erfolgreichen Initialisierung kann er mittels der Funktion tte_start() aktiviert werden.

In der Konfigurationsphase werden die Ereignisse, welche bei TTEthernet statisch geplant werden, in einer chronologisch sortierten Liste gespeichert. Dadurch ist der Aufwand Ereignisse aufzufinden während der Laufzeit gering und es kann ohne eine neue Berechnung direkt das nächste Ereignis ermittelt werden. Die Ereignisse werden in einer Ereignisstruktur organisiert, welche die Felder Ereignisklasse, Ausführungszeitpunkt und einen Parameter für Daten enthält. Die Felder eines Ereignisses werden vorher über die TTE API definiert. Die Ereignisklasse beschreibt den Typ des Eintrags und wird dem entsprechenden Scheduler Task zugeordnet. Im Ausführungszeitpunkt steht die Zeit des Ereignisses relativ zu dem konfigurierten Zyklus. Der Parameter für Daten enthält einen Zeiger auf den entsprechenden Nachrichtenbuffer.

4.5.1 Scheduler

Der Scheduler arbeitet mit dem High Resolution Timer von Linux, welcher eine Auflösung im Nonosekundenbereich unterstützt. Im Scheduler wird immer eine Referenz auf das aktuelle Ereignis in der Liste der Ereignisse gehalten. Sobald der Scheduler gestartet wird, wird der Timer auf das erste Ereignis in der Liste programmiert. Diese Zeit wird berechnet, indem die Zeit aus der Ereignisstruktur zu der Zykluszeit (siehe Kapitel 3.4) hinzuaddiert wird.

Wenn der Timer zu der programmierten Zeit auslöst, wird die Funktion, welche bei Initialisierung an den Timer übergeben wurde, aufgerufen. Hier wird anhand der Referenz auf das aktuelle Ereignis eine Unterscheidung vorgenommen und der entsprechende Task Thread aktiviert und anschließend der Timer auf das nächste Ereignis programmiert. Das Senden einer TT Nachricht wird, da es die höchste Priorität im TTE System hat, sofort in der ISR des

Timers ausgeführt, um möglichst wenig Overhead zu erzeugen. Wenn alle Ereignisse einer Periode verarbeitet wurden, wird die Zykluszeit angepasst, wie in Kapitel 3.4 beschrieben, und der Timer wird wieder auf das erste Ereignis in der folgenden Periode programmiert.

4.5.2 Synchronisation

Die Synchronisation wird von dem Sync Task, beschrieben in Kapitel 4.4, vorgenommen.

4.6 Frames Senden

Das Senden eines Frames wird über die Funktion tte_tx im TTEthernet Treiber realisiert. Der Funktion wird eine Socketbuffer Struktur, welche von Linux für die Verarbeitung von Frames genutzt wird, übergeben werden. Die Struktur enthält das Ethernetframe nach dem Ethernetstandard (Header und Payload). Um die Queue, welche beim Senden von Frames unter Linux genutzt wird (siehe Kapitel 2.5.3), zu umgehen und das Frame direkt über das reale Netzwerkinterface zu senden wird eine Referenz auf das vorher konfigurierte Interface geholt. Diese Referenz zeigt auf die net_device Struktur der realen Netzwerkkarte. Anhand dieser Struktur können vor dem Senden Prüfungen vorgenommen werden, ob das Gerät sendebereit ist. Ein exklusiver Zugriff auf die Hardware wird ermöglicht indem ein Lock auf das TX-Queue-Lock des realen Netzwerkinterfaces vorgenommen wird. Nach erfolgreichem Lock wird das Frame über die zum Senden bereitgestellte Funktion, welche über die net_device Struktur ermittelt wird, über die Netzwerkkarte gesendet. Anschließend wird das Gerät mit einem Unlock wieder freigegeben.

4.7 Frames Empfangen

Der Treiber der realen Netzwerkkarte ruft im Softirq Handler (siehe Kapitel 2.5.2) direkt eine Funktion des Linux Netzwerk Stacks auf, um angekommene Frames abzuliefern. Frames vom Linux Stack zu empfangen würde wieder eine Verzögerung ergeben welches nicht erwünscht ist. Aus diesem Grund wird der Treiber der realen Netzwerkkarte um ein #define erweitert welches die Funktion des Linux Stacks mit der äquivalenten Funktion für den Empfang im TTEthernet Treiber überschreibt. Im aktuellen Kernel kommen dafür drei Funktionen (siehe Quellcode 4.4) infrage, welche je nach Implementierung des Netzwerkkartentreibers aufgerufen werden.

```
#include "../tte.h"
#define tteif_receive_skb netif_receive_skb
#define tteif_rx netif_rx
#define napi_gro_receive ttenapi_gro_receive
```

Quellcode 4.4: defines im realen Netzwerkkartentreiber um Frames empfangen zu können

Diese 3 Funktionen bekommen jeweils eine Sockebuffer Struktur, welche die Ethernet Framedaten enthält, übergeben. Diese wird an die im TTEthernet Treiber für den Empfang zuständige Funktion tte_rx() übergeben. Das Frame wird mit einem Zeitstempel versehen und anhand der Filterfunktion eine Referenz in den entsprechenden Eingangsbuffer geschrieben. Des weiteren wird bei BE und PCF Frames die entsprechende wait_queue über den Eingang des jeweiligen Frames informiert.

5 Tests, Ergebnisse und Ausblick

In diesem Kapitel werden die abschließenden Tests vorgestellt und die aufgestellten Anforderungen überprüft. Zum Schluss wird eine Bewertung der Ergebnisse sowie ein Ausblick zu weiteren Möglichkeiten in der Zukunft gegeben.

5.1 Funktionstests

In den Funktionstests wird das Senden von TT Frames und die korrekte Funktionsweise des virtuellen Netzwerkinterfaces überprüft.

5.1.1 TT Frames Senden

Für diesen Test wurde eine Konfiguration erstellt, welche mit einem Zyklus von 2ms arbeitet. Innerhalb dieses Zyklus wird nach 180us eine TT Nachricht mit dem CT-Marker 0x03040506 und einer CT-ID von 0x200 gesendet. Die gesendeten Frames werden mit einem Messsystem, welches von Friedrich Groß in seiner Bachelorarbeit (Vgl. Groß, 2011) entwickelt wurde, auf einem netX500 Mikrocontroller gemessen. Es wurde eine Messung ohne Last und eine Messung mit hoher Last, auf beiden CPUs des Rechners, auf welchem der TTEthernet Treiber läuft, gemacht. Die Last wurde mit dem Programm cpuburn, welches versucht die CPU komplett auszulasten, erzeugt.

Bei den gemessenen Zeiten wurden jeweils die Schwankungen der Abstände der gesendeten TT Frames untersucht. Die Zeiten der beiden Messungen sind in den Tabellen 5.1 und 5.2 dargestellt. In der ersten Spalte mit Werten (100%) wurden alle gemessenen Werte einbezogen. In der zweiten Spalte wurden ausreißende Messwerte entfernt, wobei jeweils 97% ohne Last und 96% mit Last der Werte für die Bestimmung des Jitters dienten.

Zu sehen ist, dass der anfängliche hohe Jitter nur durch einen geringen Teil der gesendeten Frames zustande kommt. Der Jitter der gesendeten Frames ist ohne Last bei Einbeziehung von 97% der Testdaten und mit Last bei 96% nur noch im einstelligen Mikrosekundenbereich.

Um die ausreißenden Werte mehr oder vielleicht ganz zu eliminieren muss analysiert werden welche Prozesse im Betriebssystem das Senden verzögern können und ob diese zu unterbinden sind. In Linux (auch mit dem RT Patch) gibt es einige Stellen an denen im Kernel Preemtion deaktiviert ist, um Deadlocks bei Zugriffen auf gemeinsam genutzte Ressource zu vermeiden. Da ein Betriebssystem wie Linux sehr komplex ist und viele Faktoren eine Rolle spielen, ist es nicht einfach diese Prozesse, welche die Verzögerung auslösen zu finden und bedarf umfassender Recherche und tief greifende Kenntnisse des Linux Kernels sowie den Modifizierungen durch den RT Patch.

	Zeit/us (100%)	Zeit/us (97%)
Min	1941,356	1973,524
Max	2022,406	1983,426
Jitter	81,050	9,902

Tabelle 5.1: Messung gesendeter TT Nachrichten ohne Last

	Zeit/us (100%)	Zeit/us (96%)
Min	1934,476	1978,356
Max	2021,683	1979,634
Jitter	87,207	1,278

Tabelle 5.2: Messung gesendeter TT Nachrichten mit Last

Da eine Synchronisation des Systems derzeit noch nicht sauber funktioniert wurden die Frames im unsynchronisierten Zustand versenden. Dadurch entsteht, obwohl im 2ms Zyklus gesendet wird, eine konstante Abweichung von diesen 2ms, da die Uhr im Messsystem eine minimale Abweichung von der des sendenden Rechners aufweist. Somit sind die 2ms auf dem sendenden Rechner etwas kürzer als die des Messsystems, was erklärt warum die gemessenen Abstände außer bei den ausreißenden Werten, kleiner als 2ms sind.

5.1.2 BE Senden und Empfangen

Um die Funktionalität des virtuellen Interfaces des TTEthernet Treibers zu überprüfen wurden folgende Tests gemacht. Die Netzwerkkarte, welche von dem virtuellen Interface genutzt wird, wurde an einen Router angeschlossen, welcher Zugang zum Internet hat. Der Router verfügt zudem über einen DHCP-Server. Als Erstes wurde, wie in Quellcode 5.1 zu sehen, eine IP Konfiguration von dem DHCP-Server für das virtuelle Interface tte0 angefordert. Folgend ist im Quellcode die Konfiguration des Interfaces zu sehen, welche mit dem Befehl ifconfig tte0 aufgerufen wird.

```
timb21 ~ # dhclient tte0
timb21 ~ # ifconfig tte0
tte0   Link encap:Ethernet  Hardware Adresse 00:15:17:0e:09:57
       inet Adresse:192.168.1.122  Bcast:192.168.1.255  Maske:255.255.255.0
       inet6-Adresse: fe80::215:17ff:fe0e:957/64 Gltigkeitsbereich:Verbindung
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metrik:1
       RX packets:111 errors:0 dropped:0 overruns:0 frame:0
       TX packets:133278 errors:0 dropped:0 overruns:0 carrier:0
       Kollisionen:0 Sendewarteschlangenlnge:1000
       RX bytes:31188 (30.4 KiB)  TX bytes:8015655 (7.6 MiB)

timb21 ~ # arp -a
DD-WRT (192.168.1.1) auf 00:1a:70:eb:62:4e [ether] auf tte0
timb21 ~ # route
Kernel-IP-Routentabelle
Ziel            Router      Genmask         Flags Metric Ref    Use Iface
default         DD-WRT      0.0.0.0         UG    0      0        0 tte0
192.168.1.0     *           255.255.255.0   U     0      0        0 tte0
```

Quellcode 5.1: IP Konfiguration des virtuellen Interfaces

Anschließen wird mit dem Programm ping überprüft, ob das Interface Ethernet Frames senden und empfangen kann. Ping sendet an den Server 8.8.8.8 einen ICMP Request über IP und erhält im Erfolgsfall von diesem Server, wenn er für das ICMP Protokoll konfiguriert ist, eine ICMP Response. Die Ergebnisse dieser Messung sind in Quellcode 5.2 dargestellt. Zu sehen ist das von 5 gesendeten Frames über das virtuelle Netzwerkinterface alle beantwortet wurden und die Antwort empfangen werden konnte.

```
timb21 ~ # ping -c 3 -I tte0 8.8.8.8
PING 8.8.8.8 (8.8.8.8) from 192.168.1.122 tte0: 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_req=1 ttl=46 time=12.2 ms
64 bytes from 8.8.8.8: icmp_req=2 ttl=46 time=12.5 ms
64 bytes from 8.8.8.8: icmp_req=3 ttl=46 time=12.1 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 12.133/12.325/12.545/0.169 ms
```

Quellcode 5.2: Ping über das virtuelle Interface

5.2 Überprüfung der Anforderungen

Virtuelles Netzwerkinterface für BE Traffic

Senden (erfüllt)

Empfangen (erfüllt)

Nutzung von Socket basierten Anwendungen (erfüllt)

ARP, DHCP, Routing, NAT, Firewall etc. (erfüllt)

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

BE Traffic unter Kontrolle des Schedulers stellen (erfüllt)

Synchronisations Client (teilweise erfüllt)

Synchronisations Master (nicht erfüllt)

Synchronisations Compression Master (nicht erfüllt)

5.3 Analyse

Virtuelles Netzwerkinterface

Das virtuelle Netzwerkinterface arbeitet ohne Einschränkungen. Es lässt sich über die unter Linux gewohnten Tools konfigurieren und unterstützt die vom Linux Stack implementierten Protokolle voll. Socketbasierte Anwendungen können ohne das es Änderungen bedarf mit dem Interface genutzt werden. Wenn sich ein DHCP Server im Netzwerk befindet, kann über diesen eine IP bezogen werden, es erscheint wie jedes reale Netzwerkinterface in der ARP- sowie Routingtabelle und kann mit einem DHCP-Client konfiguriert werden.

Das Problem mit der Prioritäts Inversion, welche bei gleichzeitigem Senden von BE und TT Nachrichten auftreten kann besteht noch. Dieses Problem ist aber mit einer Überprüfung der TT-TX Events in der Liste des Schedulers vor dem Senden von BE Frames lösbar, konnte aber im Rahmen dieser Arbeit nicht mehr umgesetzt werden.

TTEthernet Controller

Der TTEthernet Controller kann über die TTE API angebrochen werden und unterstützt alle Funktionen. Der Controller kann über eine config.c Datei konfiguriert werden, Ein- und Ausgangsbuffer können gelesen und beschrieben werden. In der config.c konfigurierte TT Nachrichten können gesendet und empfangen werden.

Eine Synchronisierung des TTEthernet Controllers konnte im Rahmen dieser Arbeit nicht erfolgreich getestet werden. Der Algorithmus für die Synchronisierung wurde implementiert. Die Synchronisation kann aber nur korrekt funktionieren, wenn alle statischen Verzögerungen bekannt sind und alle dynamischen Verzögerungen von der Ankunft eines PCF Frames im Controller bis zu seiner Verarbeitung gemessen werden können. Dies stellte sich durch die Komplexität des Linux Kernel als größtes Problem dar. Durch den Zeitstempel, welcher in einem Frame gespeichert wird sobald es den Treiber des Controllers erreicht, lässt sich zwar die Verzögerung von dort bis hin zur Verarbeitung bestimmen, aber die Zeit des Empfangs in der realen Hardware, bis zum Erreichen des Controllers fehlt.

Das Ziel dieser Arbeit war unter anderem einen Treiber zu entwickeln, der mit jedem gängigen Netzwerkkartentreiber funktioniert ohne diesen maßgeblich zu verändern. Dies stellte sich am Ende als Problem dar. Die Verarbeitung der ankommenden Frames (in Kapitel 2.5.2 beschrieben) durch Software Interrupts, wie es unter Linux in aktuellen Kernel gehandhabt wird, ist für eine statische Empfangszeit von Ethernet Frames sehr ungünstig. Gängige Netzwerkkartentreiber arbeiten alle im NAPI Modus, was eine Kombination aus Interrupts und Polling darstellt. Damit ein Netzwerkkartentreiber ohne die von Linux vorgegebenen für RT

ungünstigen Mechanismen arbeitet, sind weitere Änderungen im Treiber nötig. Die Funktionen, welche für das Polling der Frames zuständig sind, werden im Netzwerkkartentreiber implementiert und sind sehr Hardware spezifisch, da dort die Frames aus dem Buffer der Netzwerkkarte ausgelesen und dann an den Stack übergeben werden.

Zusammengefasst heißt dass um vorhersagbare Empfangszeiten zu erreichen, was für die Synchronisierung Voraussetzung ist, muss sich intensiver mit dem jeweiligen Treiber der Netzwerkkarte auseinandergesetzt werden und dieser für den TTEthernet Controller angepasst werden.

5.4 Bewertung

Ein virtuelles Netzwerkinterface für Linux kann ohne Probleme in einen TTEthernet Controller integriert werden, wobei auf keine Eigenschaften von Ethernet unter Linux verzichtet werden muss. Das Senden und Empfangen über Buffer funktioniert ohne Probleme.

Den TTEthernet Stack in Linux mit dem RT Patch zu verwenden ist meiner Einschätzung nach realisierbar. Eine Umsetzung mit modifizierten Treibern von TTTech (Vgl. Grillinger, 2009) existiert bereits und auch eine Implementierung des TTE Controllers für RTAI Linux (Vgl. Grillinger u. a., 2006) erzielte positive Ergebnisse mit modifizierten Treibern der Netzwerkkarte. Für einen stabilen Stack der alle Funktionen unterstützt und sich stabil synchronisiert ist ein sehr gutes Verständnis des Linux Kernels (sowie seiner RT-Erweiterung) und seiner Arbeitsweise nötig. Durch die Komplexität und fehlende oder veraltete Dokumentationen wird es einem oft erschwert, sich in diesem zurechtzufinden. Gerade, wenn es darum geht Verzögerungen zu finden, ist es sehr wichtig zu wissen wie der Kernel arbeitet, um zu wissen wo die Messungen der Zeiten anzusetzen sind. Zusätzlich ist eine Analyse des zu verwendetet Netzwerkkartentreibers und eine spezielle Anpassung dieses nötig, um den Controller mit Synchronisierung zu realisieren.

5.5 Ausblick

Anschließend an diese Arbeit kann der bereits implementierte Synchronisationsalgorithmus durch einen modifizierten Netzwerkkartentreiber und eine Analyse der Verzögerungen, welche bei dem Empfang und der Verarbeitung der Frames entstehen, zum Laufen gebracht werden. Wenn die Synchronisierung in ein bestehendes System erfolgreich umgesetzt wurde, kann über eine Erweiterung des Controllers über die Funktionalitäten des Synchronisations- und Compressionsmasters nachgedacht werden. Ob eine Implementierung dieser sinnvoll ist, hängt von der Genauigkeit in der Bestimmung der Verzögerungen

ab, um eine korrekte Synchronisierung des TTEthernet Systems zu gewährleisten. Von einem Compressionmaster, in einem TTEthernet Controller unter Linux mit RT Patch, ist aber abzuraten. Da eine fehlerfreie Ausführung von Linux nicht mathematisch bewiesen ist, würde bei einem Systemfehler durch den Verlust des Compressionmasters das ganze TTEthernet System zum Erliegen kommen. Ebenfalls, wenn der TTEthernetcontroller als Master realisiert wird, sollte dieser nur unterstützend mit anderen Mastern arbeiten und nicht alleine für die fehlerfreie Übertragung im System zuständig sein.

Literaturverzeichnis

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

[Benvenuti 2006] BENVENUTI, Christian: *Understanding Linux network internals*. O'Reilly & Associates, Inc., 2006. – xxiv + 1035 S. – URL http://www.oreilly.com/catalog/9780596002558. – ISBN 0-596-00255-6

[Buttazzo 2011] BUTTAZZO, Giorgio C.: *Real-Time Systems Series*. Bd. 24: *Hard Real-Time Computing Systems*. Springer US, 2011. – ISBN 978-1-4614-0676-1

[Corbet 2011] CORBET, Jonathan: *The 2011 realtime minisummit*. Oktober 2011. – URL https://lwn.net/Articles/464180/. – Zugriffsdatum: 2012-05-15

[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

[Gleixner 2011] GLEIXNER, Thomas: *[ANNOUNCE] 3.0-rc7-rt0*. Juli 2011. – URL https://lkml.org/lkml/2011/7/19/309. – Zugriffsdatum: 2012-05-09

[Grillinger 2009] GRILLINGER, Petr: *100Mbit/s Evaluation System Introduction*. TTTech Computertechnik AG. Juni 2009. – URL http://www.tttech.com

[Grillinger u. a. 2006] GRILLINGER, Petr ; ADEMAJ, Astrit ; STEINHAMMER, Klaus ; KOPETZ, Hermann: Software Implementation of Time-Triggered Ethernet Controller. In: *Workshop on Factory Communication Systems*, 2006, S. 145–150. – ISBN 1-4244-0379-0

[Groß 2011] GROSS, Friedrich: *Mikrocontroller basierte Messung von Paketlaufzeiten in Time-Triggered-Ethernet Netzwerken*. August 2011. – Bachelorthesis

[Institute of Electrical and Electronics Engineers 2005] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS: IEEE 802.3: LAN/MAN CSMA/CD Access Method / IEEE. 2005 (IEEE 802.3-2005). – Standard

[Kopetz u. a. 2005] KOPETZ, Hermann ; ADEMAJ, Astrit ; GRILLINGER, Petr ; STEINHAMMER, Klaus: The time-triggered Ethernet (TTE) design. In: *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005.*, Mai 2005, S. 22–33

[Love 2010] LOVE, Robert: *Linux kernel development*. Third. Addison-Wesley, 2010 (Developer's library: essential references for programming professionals). – xx + 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

[Müller 2011] MÜLLER, Kai: *Time-Triggered Ethernet für eingebettete Systeme: Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur*. August 2011. – Bachelorthesis

[RTAI Team 2012] RTAI TEAM: *RTAI - the RealTime Application Interface for Linux from DIAPM*. 2012. – URL http://www.rtai.org. – Zugriffsdatum: 2012-03-10

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

[Spurgeon 2000] SPURGEON, Charles E.: *Ethernet - the definitive guide: designing and managing local area networks*. O'Reilly, 2000. – I–XX, 1–498 S. – ISBN 978-1-56592-660-8

[Steiner 2008] STEINER, Wilfried: *TTEthernet Specification*. TTTech Computertechnik AG. November 2008. – URL http://www.tttech.com

[Ts'o u. a. 2012] TS'O, Theodore ; HART, Darren ; KACUR, John: *RT-Patch Official Wiki*. 2012. – URL https://rt.wiki.kernel.org/. – Zugriffsdatum: 2012-01-14

[Wörn 2005] WÖRN, Heinz: *Echtzeitsysteme: Grundlagen, Funktionsweisen, Anwendungen*. 1. Springer Berlin Heidelberg, April 2005. – ISBN 3-5402-0588-8

[Xenomai 2012] XENOMAI: *Real-Time Framework for Linux*. 2012. – URL http://www.xenomai.org. – Zugriffsdatum: 2012-03-10

[Yaghmour u. a. 2008] YAGHMOUR, Karim ; MASTERS, Jon ; BEN-YOSEFF, Gilad ; GERUM, Phillipe: *Building Embedded Linux Systems - 2nd Edition*. O'Reilly Media, Inc., August 2008. – ISBN 978-0-596-52968-0

[Zimmermann und Schmidgall 2011] ZIMMERMANN, Werner ; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik - 4. aktualisierte Auflage*. Vieweg + Teubner, 2011. – ISBN 978-3-8348-0907-0

Abkürzungsverzeichnis

ABS Antiblockiersystem

API Application Programming Interface

CPU Central Processing Unit

ESP Elektronisches Stabilitätsprogramm

IEEE Institute of Electrical and Electronics Engineers

IRQ Interrupt Request Queue

ISR Interrupt Service Routine

MAC Media Access Control

PCI Peripheral Component Interconnect

QoS Quality of Service

RCU Read Copy Update

RT Real Time

RTOS Real-Time Operating System

TTEthernet Time Triggered Ethernet

Glossar

Bug Fehler im Programmcode (Engl. Wanze)

Bussysteme Wird im Automotivebereich als Oberbegriff der Busse im Automobil wie z. B. der CAN- oder MOST-Bus verwendet. In der Informatik klassifiziert man ein Bussystem nach der Verwendung (z. B. Daten- oder Adress-Bus)

Handler Zuständige Funktion zur Verarbeitung einer Aufgabe

Jitter Schwankung der Latenz

Latenz Verzögerungs- oder Signallaufzeit von Paketen

Mainline Kernel Hauptentwicklungszweig des Linux Kernels

root Benutzer mit administrations Rechten unter Linux

Task Aufgabe

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. Juni 2012
Ort, Datum

Unterschrift